Белорусский национальный технический университет
Международный институт дистанционного образования
Кафедра «Информационные системы и технологии»

ЭЛЕКТРОННЫЙ УЧЕБНО-МЕТОДИЧЕСКИЙ КОМПЛЕКС ПО УЧЕБНОЙ ДИСЦИПЛИНЕ

СИСТЕМНОЕ ПРОГРАММИРОВАНИЕ

для специальностей:
1-40 01 01 «Программное обеспечение информационных технологий»
6-05-0612-01 «Программная инженерия»

Составитель:
Бумай Андрей Юрьевич, ассистент

Минск
БНТУ
2024

СИСТЕМНОЕ ПРОГРАММИРОВАНИЕ

БНТУ

— 1920 —

Перечень материалов

Конспект лекций, материалы для лабораторных занятий и контрольных работ, вспомогательный раздел.

Пояснительная записка

Цели данного ЭУМК – повышение эффективности организации учебного процесса с использованием дистанционных технологий; предоставление возможности студентам заниматься самообразованием, пользуясь комплектом учебно-методических материалов по дисциплине «Системное программирование».

ЭУМК содержит четыре раздела: теоретический, практический, контроля знаний и вспомогательный.

Теоретический раздел представлен конспектом лекций. Лекционный материал подготовлен в соответствии с основными разделами и темами учебной программы.

Практический раздел представлен лабораторными работами, которые помогут освоить теоретический материал дисциплины.

Раздел контроля знаний включает контрольную работу, содержащую индивидуальные варианты заданий, требования к оформлению контрольной работы, вопросы к экзамену. Вспомогательный раздел представлен учебной программой. Данное ЭУМК в первую очередь разработано для студентов МИДО дистанционной (заочной) формы получения образования, однако ЭУМК также может быть полезным студентам дневной формы получения образования и всем, кто заинтересован в освоении дисциплины «Системное прогаммирование».

СОДЕРЖАНИЕ

РАЗДЕЛ 1 ТЕОРЕТИЧЕСКИЙ

ВВЕДЕНИЕ

Системными программами называют набор программ, используемых в процессе подготовки прикладных программ на этапах от постановки задачи до выполнения на ЭВМ. Примеры системных программ - трансляторы, редакторы связей, текстовые редакторы, загрузчики, отладчики, канальные программы, драйверы.

Системное программирование - разработка, настройка и прагматическое использование системных программ в конкретной вычислительной среде.

Принципиальное отличие системного программирования от прикладного - зависимость от вычислительной системы. Системное програмирование непосредственно связано с особенностями аппаратуры ЭВМ и механизмов операционной системы.

Основные вопросы системного программиста по архитектуре ЭВМ, построенной на идее совместно запоминаемых программы и данных (Дж.фон Нейман):

– память - минимальная адресуемая единица памяти, объем памяти, схема адресации;

– регистры - количество регистров, их размер, функция и взаимосвязь;

– данные - обрабатываемые типы и формат представления в памяти арифметических, символьных и логических данных;

– команды - классы команд ЭВМ, набор команд обработки арифметических, логических и символьных данных, команд управления, формат команд и схема их хранения в памяти;

– специальные средства - структура системы прерываний, механизмы защиты и их доступность для пользователя, организация ввода-вывода.

Современные вычислительные системы, как правило, представляются наслоением программного обеспечения на аппаратуру. Работа в вычислительной среде конкретной операционной системы требует знания спецификации интерфейсов связи с ее модулями на разных уровнях детализации. В последнее время наблюдается тенденция смещения массовых задач системного программирования на программный уровень.

В настоящее время задачи системного программирования решаются с использованием чаще всего языка ассемблера либо языка С.

Под ассемблером (автокодом, системой символического кодирования) понимают систему программирования, включающую: язык символического кодирования программы на машинном уровне в терминах мнемонических

обозначений команд и символических обозначений операндов; транслятор программ на языке ассемблера в перемещаемые (объектные) модули. Функции ассемблера: отображение символических объектов исходного текста программы на память с учетом формата команд, форм представления данных и системы адресации; подготовка объектного модуля программы стандартного вида; предоставление программисту высокоуровневых услуг по автоматизации кодирования программы (вычислительные операции, макросредства, диагностика ошибок).

Ассемблер применяют после убедительного обоснования в редких классах задач: создание базовых элементов системного программного обеспечения; решение задач со специальными требованиями по вычислительной эффективности либо методам использования вычислительных ресурсов; комплексирование разнородных программных продуктов. Языком ассемблера представляют: заключительный уровень детализации результатов транслятором языка высокого уровня либо генератором прикладных программ; элементы настройки операционных систем и сложных пакетов прикладных программ на конкретные условия применения; фрагменты адаптируемых посредством дизассемблирования чужих программ при отсутствии исходного текста. Язык С, в отличие от языка ассемблера, ориентирован на переносимость программ, по крайней мере, на уровне исходного текста с одной вычислительной системы на другую. Это обеспечивается наличием трансляторов языка С практически на всех современных вычислительных системах.

1.1 ОБЗОР ЯЗЫКА ПРОГРАММИРОВАНИЯ С

1.1.1 Характеристика языка С

Язык С первоначально доминировал в среде мини-ЭВМ, работающих под управлением ОС UNIX, но в дальнейшем получил широкое распространение и в других классах вычислительных систем. Сейчас трудно найти вычислительную систему без поддержки такого языка.

Достоинства языка С:

это современный, эффективный, переносимый, мощный и гибкий, удобный и обладающий рядом присущих ассемблеру управляющих конструкций язык.

наиболее важное положительное качество языка С - переносимость (мобильность), обеспечиваемая наличием стандарта языка и, следовательно, совместимых на уровне исходного текста систем программирования.

Недостаток языка С - относительно плохая "читаемость" текста программ из-за неоднозначности синтаксиса.

Области применения языка С - системное программирование и прикладные задачи с жесткими требованиями по скорости и памяти.

Язык С относится к классу языков процедурного типа с адресной арифметикой и формально является подмножеством языка объектно-ориентированного программирования С++. Освоение языка С создает фундамент для изучения языка С++ [1].

1.1.2 Основные сведения о синтаксисе языка С

Начальные сведения о синтаксисе любого языка программирования включают элементарные правила записи исходного текста программы - идентификация объектов программы, комментарии, формат исходного текста.

Идентификаторы объектов программы на языке С могут включать:

– цифры 0...9;
– латинские прописные и строчные буквы A...Z, a...z;
– символ подчеркивания _.

Первый символ идентификатора не может быть цифрой. Длина идентификатора определяется реализацией транслятора С и редактора связей (компоновщика). Современная тенденция - снятие ограничений длины идентификатора.

Разделители идентификаторов объектов программы:

– пробелы;
– символы табуляции, перевода строки и страницы;
– комментарии (играют роль пробелов).

Комментарий - любая последовательность символов, начинающаяся парой символов /* и заканчивающаяся парой символов */.

Формат записи исходного текста программ на языке С – свободный [1].

```
/* ПРИМЕР ПРОГРАММЫ ТЕСТА ГЕНЕРАТОРА СЛУЧАЙНЫХ ЧИСЕЛ */

#include <stdio.h>
#include <stdlib.h>

#define RANGE 100

void main() {
 int i,j,k,s;

 /* Запрос объема выборки */

 while (printf("\n K-? "), !scanf(" %d",&k);

 /* Генерация случайных чисел */

 printf("\n Значения %d случайных чисел:\n",k);
```

```
 for (s=i=0; i < k; i++)
  s+=(j=random(RANGE)+1);
  printf("\n %3d) %d",i,j);
 }
 printf("\n\n Сумма %d, среднее %f",s,(float)s/k);
}
```

1.1.3 Структура программ на языке С

Программа на языке С включает операторы декларации объектов, преобразования объектов и управления вычислительным процессом. Программирование процесса преобразования объектов программы производится посредством записи выражений. Выражение включает один или несколько операндов и символов операций. Любое выражение, заканчивающееся символом ';', является оператором.

Простейший вид операторов - операторы-выражения:

– оператор присваивания - выполнение операций присваивания;
– оператор вызова функции - операция вызова функции;

пустой оператор.

Классы управляющих операторов в языке С:

– операторы условного и безусловного перехода;
– операторы организации циклов;
– оператор выбора альтернатив (переключатель);
– оператор выхода из функции.

Каждый из управляющих операторов имеет конкретную лексическую конструкцию, образуемую из ключевых слов языка С, выражений и символов-разделителей '{','}',',',':','(',')'. Операторы языка С записываются в свободном формате с использованием разделителей между ключевыми словами. Допустима вложенность операторов. Любой оператор может помечаться меткой - идентификатором и символом ':'. Область действия метки - функция, где эта метка определена. В случае необходимости можно использовать составной оператор (блок) - последовательность любых операторов, заключенная в фигурные скобки { и } (после закрывающей скобки символ ';' не требуется). Элементарным модулем программы на языке С является функция. Любая программа должна содержать, как минимум, головную функцию со стандартным именем main [1].

Пример исходного текста программы:

```
#include <stdio.h>

/* ПРОГРАММА ПЕЧАТИ КОДОВ НАЖАТЫХ КЛАВИШ */

void main () {
```

```
 int i;

 while ((i=getch())!=27) {
 if (i==0) printf("\n* %d",getch());
  else printf("\n  %d",i);
 }
}
```

1.1.4 Функции и библиотеки функций языка С

Большинство трансляторов языка С – компиляторы. Технология подготовки программы к выполнению включает эпапы трансляции исходного текста в объектный модуль и получение после редактирования его связей с другими оъектными модулями исполняемого (загрузочного) модуля. Объектные модули могут размещаться в так называемой библиотеке - файле со специальным методом доступа. Редактор связей может выполнять выборку объектных модулей из библиотеки непосредственно. Включение объектных модулей в библиотеку, их удаление и контроль содержания библиотеки выполняет отдельная программа - библиотекарь. Система программирования Turbo-C включает препроцессор, компилятор, редактор связей, библиотекарь, редактор текста, отладчик и интегрированную управляющую среду.

Программирование на языке С связано с интенсивным использованием библиотечных функций. Мобильность программ на языке С потребовала при его разработке исключения операторов, реализация которых каким-либо образом связано с организацией вычислительной среды. Можно заметить, что операционные средства языка находятся на уровне машинных команд, простейших базовых типов данных, непосредственно отображаемых на регистры процессора.

Библиотеки функций на языке С обычно отражают конкретную вычислительную среду. Структуризация библиотечных функций на уровне пользователя проявляется системой так называемых заголовочных файлов, содержащих описания функций средствами языка. Заголовочные файлы с описаниями фактически используемых функций обычно включатся посредством директивы препроцессора #include в исходный текст программы. Заголовочные файлы имеют содержательные имена.

Пример именования заголовочных файлов в системе программирования:

– alloc.h - функции управления памятью;
– bios.h - функции интерфейса BIOS;
– conio.h - функции консольного ввода-вывода DOS;
– ctype.h - информация для классификации и преобразования символов;
– dir.h - средства доступа к директориям и именам файлов;
– direct.h - POSIX директорий;

– dos.h - константы и объявления DOS;
– errno.h - мнемонические константы и коды ошибок;
– fcntl.h - символические константы функции открытия файлов;
– float.h - интерфейс функций арифметики с плавающей точкой;
– graphics.h - функции для работы с графикой;
– io.h - функции низкоуровневого ввода-вывода;
– limits.h - информация об ограничениях и диапазонах значений параметров вычислительной среды;
– locale.h - информация о поддержке страны и языка;
– malloc.h - функции управления памятью;
– math.h - математические функции и структуры сообщений об ошибках;
– mem.h, memory.h - функции манипулирования блоками памяти;
– process.h - структуры и функции порождения задач;
– search.h - функции сортировки и поиска;
– setjmp.h - структуры данных и функции нелокального перехода;
– share.h - интерфейс доступа к разделяемым файлам;
– signal.h - порождение исключительных ситуаций;
– stdarg.h - средства поддержки списка аргументов функций переменной длины;
– stddef.h - разнообразные стандартные типы данных;
– stdio.h - функции стандарного ввода-вывода и предопределенные файлы ввода-вывода stdin, stdout, stdprn и stderr;
– stdlib.h - массовые вспомогательные функции;
– string.h - операции со строками символов;
– sys\stat.h - символические константы процедур открытия и создания файлов;
– sys\types.h - декларация характеристик некоторых типов данных;
– time.h - структуры данных и функции для работы со временем;
– values.h - важные системно зависимые константы.

Легко заметить, что приведенное типичное для систем программирования группирование функций отражает весьма важные в практическом программировании действия. Проведя эксперимент с любой из реальных систем программирования, можно убедиться в наличии тысяч функций и связанных с ними определений. Таким образом, знание собственно лингвистических конструкций языка С не означает обладание профессиональным уровнем работы в конкретной вычислительной среде. Иногда говорят, что язык С – язык функций.

Интегрированные среды разработки программ современных систем программирования включают, как правило, системы оперативной подсказки, позволяющие получать исчерпывающую информацию об использовании

библиотечных функций. "Точками входа" в справочный материал во многих справочных системах является имя заголовочного файла из приведенного списка. Можно заметить, что состав заголовочных файлов отражает базовый набор интуитивно ожидаемых понятий, характеризующих вычислительную среду.

Исторически язык С создан в период использования текстового режима ввода-вывода информации программы, поэтому представленные в файле stdio.h функции стандартного ввода-вывода предполагают поддержку именно такого режима. Работа в графическом режиме отображения информации, а также с манипулятором "мышь" или другими нестандартными устройствами требует специальных библиотек.

Подобные обстоятельства заставляют с повышенным вниманием относиться к этапу инсталляции и настройки систем программирования на языке С. Следует учитывать, что результат компиляции программ на языке С зависит от параметров настройки, связанных с учетом характеристик вычислительной среды (например, модели памяти в MS-DOS, режим выравнивания, контроль стека, диагностика).

Изучение языка программирования С можно вести с проведением сеансов работы в любой доступной системе программирования. На персональных ЭВМ удобными для обучения и легкодоступными являются Visual Studio Code [1].

1.2 ДЕКЛАРАЦИЯ ОБЪЕКТОВ ПРОГРАММЫ НА ЯЗЫКЕ С

1.2.1 Операционные объекты в языке С

Любые действия программы связаны с понятием операционного объекта.

Классы операционных объектов программ на языке С:

– константы;
– простые переменные;
– массивы;
– структуры;
– объединения;
– указатели объектов;
– функции.

Объекты программы в общем случае имеют атрибуты:

– тип - характеристика механизма интерпретации данных;
– класс памяти - характеристика способа организации размещения объектов в памяти (статическая и динамическая память);
– область действия - характеристика области использования объектов функциями программы (локальные и глобальные объекты).

Единственная разновидность объектов, определяемая в исходном тексте непосредственно по месту использования - константы. Иногда используют термин "самоопределенные константы". Остальные объекты программы должны быть явно описаны в виде

<атрибуты> <список_идентификаторов_объектов>;

(элементы списка разделены запятыми, а атрибуты - разделителями).

Предварительно отметим следующее:

– в языке С атрибуты типа описываются всегда явно;
– класс памяти можно не указывать, используя назначение по умолчанию;
– область действия объекта специальным ключевым словом не задается [1].

1.2.2 Понятие типа объекта в языке С

Тип объекта **рекурсивно** определяется на основе любого базового и производного типа посредством использования:

– символов модификации текущего типа;
– предписания размещения объектов известных типов в памяти.

Модификаторы текущего типа:

– символ * перед идентификатором - описание указателя на объект исходного типа;
– символы [и] после идентификатора объекта - описание массива объектов;
– символы (и) после идентификатора объекта - описание функции.

Отметим, что обязательное условие использования скобочных модификаторов любого вида - баланс открывающих и закрывающих скобок. Внутри скобок может размещаться детализирующая описание типа информация, например, размерность массива или список параметров. Допускается использование более одного модификатора типа с учетом следующих правил:

1) чем ближе модификатор к идентификатору, тем выше его приоритет;
2) модификаторы [] и () обладают приоритетом перед *;
3) дополнительно вводимые круглые скобки позволяют увеличить приоритет объединяемых ими элементов описания.

Примеры описания объектов:

```
type a0;         /* Элемент типа type */
type a1[5];      /* Массив элементов типа type */
type *a2;        /* Указатель на элемент типа type */
type **a3;       /* Указатель на указатель элемента типа type */
type *a4[5];     /* Массив указателей на элементы типа type */
type (*a5)[10]; /* указатель на массив элементов типа type */
type *a6[3][4]; /* 3-элементный массив указателей
                   на 4-элементный массив элементов типа type */
type a7[5][2];  /* Массив массивов элементов типа type */
type a8();       /* Функция, вычисляющая значение типа type */
```

```
type *a9();      /* Функция, вычисляющая указатель
                    на элемент типа type */
type (*aa)();    /* Указатель на функцию, вычисляющую
                    значение типа type */
type *ab()[6];   /* Функция, вычисляющая указатель
                    на массив элементов типа type */
type *ac[4]();   /* Массив указателей на функцию, вычисляющую
                    значение типа type */
```

Здесь type - некоторый известный текущий тип объектов. Возможности определения размещения объектов известных типов в памяти в языке С представлены понятиями массивов, структур и объединений (см. 1.2.4—1.2.5) [1].

1.2.3 Базовые типы данных в языке С

Перечень базовых типов данных:

а) целые числа:

int - целое (слово);

short - короткое целое (полуслово либо слово);

long - длинное целое (слово либо двойное слово);

unsigned - неотрицательное целое (слово);

б) символы:

char - символ (байт);

в) числа с плавающей точкой:

float - число с плавающей точкой ординарной точности;

double - число с плавающей точкой двойной точности.

Атрибут unsigned может сочетаться с атрибутами char, int, short, long. Атрибут int можно добавлять к атрибутам short и long без последствий. Примеры описания простых объектов:

```
int i,j,k;
char r;
double gfd;
```

К основным типам объектов формально относится и тип void - отсутствие значения. Очевидно, что реальные объекты типа void в памяти не могут быть созданы. Варианты применения ключевого слова void:

– нейтрализация значения, возвращаемого функцией (в отличие от других языков, где существуют два вида процедурных модулей - подпрограммы и функции, в языке С исполнимым модулем программы является функция);

– явная пометка пустого списка параметров функции;

– декларация указателей на область памяти без назначения типа хранящихся данных [1].

Примеры описания функций:

```
int sigma(char x, int y);
void betta(void);
char *gamma(int);
```

1.2.4 Массивы в языке С

Массив - простейший вид структурного типа, представляющий поименованную совокупность последовательно размещаемых в памяти данных **одного** типа. Примеры описания массивов:

```
char s[20];        /* Одномерный массив */
int  x[10][20];    /* Двумерный массив  - массив массивов */
```

Особенность индексации элементов массива в языке С - нулевой индекс первого элемента по каждому измерению. Доступ к отдельному элементу массива выполняется после указания его целочисленного индекса двумя способами: использование операции индексации - s[10], x[5][6]; косвенное обращение по указателю - *(s+10), *(*(x+5)+6) [1].

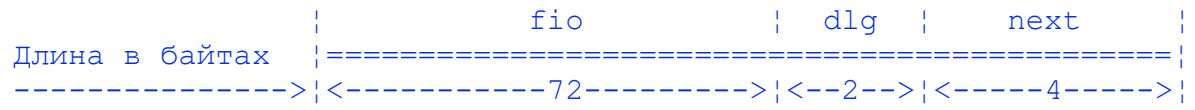
1.2.5 Структуры и объединения в языке С

Структура - поименованная совокупность логически связанных данных не обязательно одинаковых типов, размещаемых в смежных областях памяти.

Структурный тип данных может быть описан в виде:
struct имя_структуры { описание_элементов }; (между символами } и ; иногда помещают список декларируемых структурных переменных, при этом "имя_структуры" можно опустить). "Описание_элементов" производится обычным способом, ограничений на тип элементов нет. Пример определения структурного типа:

```
struct person {
 char fio[72];
 int  dlg;
 struct person *next;
};
```

Интерпретация объекта типа struct person:



Структурный тип "struct имя_структуры" можно использовать для декларации структурных переменных, массивов, функций и т.д.

```
struct person teacher;      /* Структурная переменная */
struct person student[100]; /* Массив структур */
struct person *inplst();    /* Функция - указатель на структуру */
```

Предыдущий пример можно записать кратко:

```
struct person {
 char fio[72];
 int  dlg;
 struct person *next;
} teacher, student[100], *inplst();
```

Имя структуры здесь также можно опустить:

```
struct {
 char fio[72];
 int  dlg;
 void *next;  /* См. операции над указателями */
} teacher, student[100], *inplst();
```

Элементом структуры могут быть битовые поля (строки битов):

```
struct fields {
 unsigned int flag:1;
 unsigned int mask:10;
 unsigned int code:5;
};
```

(после символа ':' указывается длина битового поля, не превышающая разрядность поля типа int).

Битовые поля размещаются последовательно в поле типа int, при нехватке места для очередного битового поля - переход на следующее поле типа int. Возможно объявление безымянных битовых полей, а длина поля 0 означает необходимость перехода на очередное поле int:

```
struct areas {
 unsigned f1:1;
           :2;    /* Безымянное поле длиной 2 бита */
 unsigned f2:5;
           :0     /* Признак перехода на след. поле int */
 unsigned f3:5;
 float data;      /* Структура может содержать   */
 char buffs[100]; /* элементы любых типов данных */
};
```

Битовые поля могут использоваться в выражениях как целые числа соответствующей длине поля разрядности в двоичной системе исчисления. Единственное отличие этих полей от обычных объектов запрет операции определения адреса (&)(см. 3.11). Следует учитывать, что использование битовых полей снижает быстродействие программы по сравнению с представлением данных в полных полях из-за необходимости выделения битового поля. Элементы структур в общем случае размещаются в памяти последовательно с учетом выравнивания начальных адресов. Выравнивание - установка значения адреса, кратного некоторой величине, определяемой особенностями адресации данных на аппаратном уровне. Например, в системе 360/370 используется выравнивание на границу полуслова (2 байта), слова (4 байта) или двойного слова (8 байт). Часто выравнивание не обязательно, но при этом скорость обращения к объекту может снижаться. В системе программирования Turbo-C можно выбирать режим выравнивания. Включение режима выравнивания становится заметным при использовании структур и объединений, где объявлены объекты с разными границами выравнивания. Компиляторы языка С не меняют порядок элементов структуры, поэтому в поле размещения структуры могут появиться неиспользуемые участки памяти. В итоге размер структуры может отличаться от суммы размеров ее элементов. Структурный тип данных удобно применять для группового управления манипулирования логически связанными объектами. Параметрами таких операций являются адрес и размер структуры. Примеры групповых операций:

– захват и освобождение памяти для объекта, представленного совокупностью не обязательно однотипных данных (см. struct person);

– запись и чтение данных, хранящихся на внешних носителях как физические и(или) логические записи с известной структурой.

Объединение - поименованная совокупность данных разных типов, размещаемых с учетом выравнивания в одной и той же области памяти, размер которой достаточен для хранения наибольшего элемента. объединенный тип данных декларируется подобно структурному типу:

```
union имя_объединения {
  описание_элементов
};
```

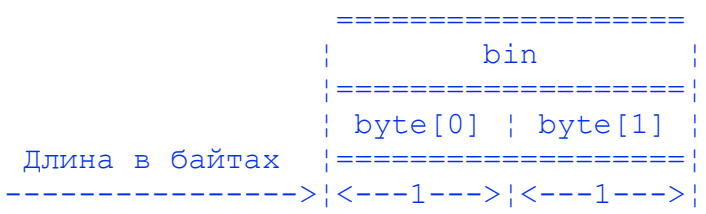
(между символами } и ; иногда помещают список декларируемых переменных объединенного типа, тогда "имя_объединения" можно опустить).

Пример описания объединенного типа:

```
union word {
 int bin;
```

```
 char byte[2];
};
```

Интерпретация объекта типа union word:



Пример объявления объектов объединенного типа:

```
union word *area, buffer[100];
```

Объединения применяют для экономии памяти в случае, когда объединяемые элементы логически существуют в разные моменты времени либо требуется разнотипная интерпретация поля данных. Например, поток сообщений по каналу связи пусть содержит сообщения трех видов:

```
struct m1 {
 char code;
 float data[100];
};

struct m2 {
 char code;
 int mode;
};

struct m3 {
 char code, note[80];
};
```

Элемент code - признак вида сообщения. Удобно описать буфер для хранения сообщений в виде

```
struct m123 {
 char code;
 union {
  float data[100];
  int mode;
  char note[80];
 };
};
```

Обращение к элементам структур или объединений производится посредством операции принадлежности (.) в виде [1]

```
имя_структуры_или_объединения.имя_элемента
```

или

```
(*указатель_структуры_или_объединения).имя_элемента
```
либо операции косвенной адресации (->) в виде
```
указатель_структуры_или_объединения->имя_элемента
```
Примеры обращения:

```
teacher.next
area->bin
buffer[i].byte[1]
```

1.2.6 Перечисления

Перечисления - средство создания типа данных посредством задания ограниченного множества значений. Определение перечислимого типа данных имеет вид

```
enum имя_перечисляемого_типа {
 список_значений
};
```

(между символами } и ; иногда помещают список декларируемых переменных перечисляемого типа, а "имя_перечисляемого_типа" можно опустить). Значения данных перечисляемого типа указываются идентификаторами:

```
enum marks {
 absence, two, three, four, five
};
```

Транслятор последовательно присваивает идентификаторам списка значений целочисленные величины 0,1,... . При необходимости можно явно задать значение идентификатора, тогда очередные элементы списка будут получать последующие возрастающие значения:

```
enum level {
 low=100, medium=500, high=1000, limit
};
```

(значение limit будет равно 1001).
Пример объявления переменной перечисляемого типа:

```
enum marks estimation;
enum level state;
```

Переменная marks может принимать только значения из множества {absence, two, three, four, five}. Возможные операции с данными перечисляемого типа:

– присваивание переменных и констант одного типа;
– сравнение для выявления равенства либо неравенства.

Практическое назначение перечислений - определение множества **различающихся** символических констант целого типа. Назначить значения элементов перечисления можно любым константным выражением, включая ранее определенные значения даже определяемого типа:

```
enum proces_status {
 passive=0x01,              /* Пассивный элемент трассы    */
 actived=0x02,              /* Активный элемент трассы     */
 goalset=actived+passive,   /* Фильтр меток решения        */
 visited=0x04,              /* Просмотренная вершина сети  */
 storage=0x08,              /* Пропускаемая вершина трассы */
 labeled=visited+storage,   /* Фильтр меток поиска решения */
 tempset=~labeled,          /* Фильтр временных меток      */
 pathset=~goalset           /* Фильтр меток трассы         */
};
```

Очевидно, что элементы перечисления не хранятся в памяти, а лишь становятся известными компилятору. Только их упоминание в выражениях приводит к размещению в памяти соответствующего целочисленного значения [1].

1.2.7 Оператор переопределения типа

Обязательность явной декларации типов объектов в языке С в случае использования производных типов требует повышенного внимания при записи идентификаторов базовых типов. Часто новому образуемому посредством модификаторов производному типу удобно назначить уникальный идентификатор, учитывая его содержательный смысл. Например, представлением комплексных чисел может быть структура

```
struct complex {
 float real, image;
};
```

Именем типа здесь выступает пара слов - struct complex, поэтому декларация скалярной переменной, массива и указателя такого типа имеет вид

```
struct complex x,y[10],*z;
```

Вариант декларации тех же объектов в форме записи

```
COMPLEX x,y[10],*z;
```

выглядит предпочтительнее для чтения исходного текста программы. Здесь выполнена формальная замена текста struct complex идентификатором

COMPLEX. Соответствующая такой замене так называемая операция лексемного замещения может быть предписана оператором препроцессора языка С

```
#define COMPLEX struct complex
```

(подробности см. в 1.6.2). Однако по отношению к идентификации типов более удобен оператор переопределения типа **компилятора** языка С:

```
typedef старый_тип новый_тип;
```

(здесь старый_тип, новый_тип - имена типов объектов). Область действия такого оператора определяется его расположением аналогично операторам описания типа объектов. Примеры переопределения типа и использования новых имен типов:

```
typedef int INTEGER;

     INTEGER a,b,c;

typedef struct {
 float real;
 float image;
} COMPLEX;

     COMPLEX x,y[10],*z;

typedef void interrupt (*HANDLER)(void);

     HANDLER old_int_0c;

typedef unsigned int (STREAM)[2];

     STREAM input;
```

Два последних примера показывают, что формальное отношение между элементами текста "старый_тип" и "новый_тип" может быть более сложным, чем при использовании оператора препроцессора #define. Любые виды модификаторов типа могут окружать фразу "новый_тип". Оператор typedef может определять несколько новых типов, если использовать запятые в качестве разделителя:

```
typedef struct _COMSTAT {   /*  Назначение типу        */
 DWORD fCtsHold : 1;        /*  struct _COMSTAT        */
 DWORD fDsrHold : 1;        /*  идентификатора         */
 DWORD fRlsdHold : 1;       /*  COMSTAT,               */
 DWORD fXoffHold : 1;       /*  а типу                 */
 DWORD fXoffSent : 1;       /*  (struct _COMSTAT *) -  */
 DWORD fEof : 1;            /*  идентификатора         */
 DWORD fTxim : 1;           /*  LPCOMSTAT              */
 DWORD fReserved : 25;      /**************************/
```

```
 DWORD cbInQue;
 DWORD cbOutQue;
} COMSTAT, *LPCOMSTAT;

typedef struct _PROCESS_INFORMATION {
 HANDLE hProcess;          /*************************/
 HANDLE hThread;           /* Здесь                 */
 DWORD dwProcessId;        /* PPROCESS_INFORMATION  */
 DWORD dwThreadId;         /* и                     */
} PROCESS_INFORMATION,     /* LPPROCESS_INFORMATION - */
 *PPROCESS_INFORMATION,    /* синонимы имени типа   */
 *LPPROCESS_INFORMATION;   /* (PROCESS_INFORMATION *) */
```

Использование оператора typedef позволяет улучшить читаемость и переносимость программы [1].

1.2.8 Константы в программах на языке С

Константы - объекты, не подлежащие использованию в левой части оператора присваивания.

В языке С константами являются:

– самоопределенные арифметические, символьные и строковые константы;
– имена массивов и функций;
– элементы перечислений.

Арифметические константы могут быть целого типа или числами с плавающей точкой. Целые константы:

а) десятичные - последовательность цифр 0...9, первая из которых не должна быть 0;

б) восьмеричные - последовательность цифр 0...7, первая из которых должна быть 0;

в) шестнадцатеричные - последовательность цифр 0...9 или букв A...F (a...f), начинающаяся символами 0x (0X).

Память для перечисленных видов констант выделяется в зависимости от значения. Можно предписать размещение константы по требуемому формату, добавив после значения константы символ l(L) - long и(или) u(U) unsigned. Примеры целочисленных констант:

```
1992, 13, 777, 1000L - десятичные; 0777, 00033, 011 - восьмеричные;
0x123, 0X00ff, 0xb80001 - шестнадцатеричные.
```

Константы с плавающей точкой размещаются в памяти по формату double, а во внешнем представлении состоят из частей:

– целая часть - цифры 0...9;
– десятичная точка;
– дробная часть - цифры 0...9;

– символ экспоненты e либо E;
– порядок - целая десятичная константа, возможно со знаком.

Примеры констант с плавающей точкой:

```
1.0, .125, 100e-10, .125E+13
```

Символьные константы - заключенные в апострофы символы кода ASCII, рассматриваемые как данные типа int. Целочисленному значению кода символа в диапазоне 0...255 (0...0377 или 0...0xff) может соответствовать:
– изображаемый текстовой символ (алфавитно-цифровой, пробел, символы пунктуации, знаки операций);
– специальный символ (псевдографика, национальный алфавит);
– управляющий символ (разделители блоков данных, управление обменом и др.).

Текстовые символы можно непосредственно вводить с клавиатуры,а специальные и управляющие приходится в исходном тексте представлять парами текстовых символов.

Примеры представления управляющих символов:

\n - новая строка;

\t - горизонтальная табуляция;

\v - вертикальная табуляция;

\b - возврат на шаг;

\r - возврат каретки;

\f - новая страница;

\a - звуковой сигнал.

Примеры представления специальных символов языка С:

\\ - обратная косая черта;

\' - апостроф;

\" - кавычки;

\0 - нулевой символ (пусто).

Любой символ может быть представлен определением значения кода в форматах:

восьмеричного числа \ddd, где ddd - значение кода символа (0...377).

шестнадцатеричного числа \xdd, где dd - значение кода символа (0...ff).

Примеры символьных констант:

```
'A', '9', '$', '\n', '\072','\x6a'
```

Строковые константы - заключенная в кавычки последовательность символов кода ASCII. Во внутреннем представлении к строковойконстанте добавляется нулевой символ '\0', отмечающий конец строки. Примеры строковых констант:

```
"Система", "\n Аргумент %d\n", "Состояние \"WAIT\""
```

Внутреннее представление строковой константы "01234\0ABCDEF" будет [1]

```
'0','1','2','3','4','\0','A','B','C','D','E','F','\0'
```

1.3 ОПЕРАТОРЫ И ВЫРАЖЕНИЯ

1.3.1 Обзор операций языка С

Операции - атомарные на уровне любого языка действия над операндами, приводящие к получению некоторого результата и возможному изменению состояния операнда.

– В языке С можно выделить следующие группы операций:
– операции присваивания;
– арифметические операции;
– операции отношений;
– логические операции;
– операции над битами;
– условное вычисление;
– определение размера объекта по имени или типу;
– операция приведения типа;
– последовательное вычисление выражений;
– операции над указателями;
– операция вызова функции.

1.3.2 Операции присваивания

Присваивание значения в языке С в отличие от традиционной интерпретации

идентификатор=выражение;

рассматривается как **выражение**, имеющее значение левого операнда после присваивания. **Оператор присваивания**, таким образом, может включать несколько операций присваивания и, как следствие, изменить значения нескольких операндов как побочный результат вычисления выражения:

```
int i,j,k;
float x,y,z;
char a,b;
...
x=i+(y=3)-(z=0);   <===>   z=0; y=3; x=i+y-z;
i=j=k=0;           <===>   k=0; j=k; i=j;
a=(i+b);           <===>   a=i+b; /* Результат - ? */
```

Очевидно, что в случае использования в выражении неодинаковых операций или разнотипных операндов возникают вопросы:

– порядок выполнения операций в выражении;
– последовательность вычисления операндов;
– корректность преобразования операндов.
– Предварительные ответы:

1) порядок выполнения операций можно определять круглыми скобками и(или) учитывать их **приоритет**, а в случае одинакового приоритета – **направление** выполнения каждой операции;

2) последовательность вычисления операндов большинства операций стандартом языка С не регламентирована, а для коммутативных операций может оптимизироваться компилятором даже при наличии круглых скобок;

3) корректность преобразования операндов гарантируется в рамках реальных аппаратных возможностей, подозрительные ситуации стремится обнаружить компилятор, а любые неопределенности устраняются явным использованием операции **приведения типа** (см. 1.3.9).

Таким образом, конструирование выражений в языке С следует проводить с нетрадиционной для языков высокого уровня осторожностью. Из выделенных понятий только приоритет и направление выполнения операций стандартны в языке С, а остальные в общем случае являются системно зависимыми. Традиционная форма **операции присваивания**

```
v = e
```

(здесь v - операнд-переменная, e - выражение). В языке С допускается две разновидности сокращений записи операции присваивания:

1) вместо записи

```
v = v operator_ab e
```

рекомендуется использовать запись

```
v operator_ab = e
```

(здесь operator_ab - арифметическая операция либо операция над битовым представлением операндов);

2) вместо записи

```
ipv = ipv + 1
```

либо

```
ipv = ipv - 1
```

рекомендуется использовать запись

```
ipv++
```

либо

```
ipv--,
```

а также

`++ipv` либо `--ipv`

(здесь символ '+' обозначает операцию инкремента, символ '-' операцию декремента, ipv - целочисленная переменная или переменная - указатель).

Значением выражения ++x или --x является значение x после операции инкремента или декремента, а значением выражения x++ или x-- - значение x до операции (подробности операций над указателями см. ниже). Примеры использования сокращений:

```
int i,j,k;
float x,y;
...

x*=y;      <===>  x=x*y;
i+=2;      <===>  i=i+2;
x/=y+15;   <===>  x=x/(y+15);
k--;       <===>  k=k-1;
--k;       <===>  k=k-1;
j=i++;     <===>  j=i; i=i+1;
j=--i;     <===>  i=i-1; j=i;
```

Рекомендации использования сокращений обоснованы возможностью оптимизации программы. Например, схема выражения вида

```
v operator_ab = e
```

соответствует схеме выполнения многих машинных команд типа «регистр-память». Ограничения на целочисленность операндов операций вида ++ipv (--ipv) или ipv++ (ipv--) следуют из наличия машинных команд инкремента и декремента целочисленных операндов. Отметим, что левым операндом операции присваивания может быть только именованная либо косвенно адресуемая указателем переменная (в диагностических сообщения компилятора - LVALUE). Примеры недопустимых выражений:

а) присваивание константе:

```
2=x+y getch()=i z=&y /* z - имя массива */
```

б) присваивание результату операции:

```
(i+1)=(j=2+y); (float)i=1.012
```

Элементы массивов адресуются косвенно (операции над указателями см. в 1.3.11), поэтому допустимы выражения

```
int area[];

area[i]=log2(i);
area[i]++;
area[i]*=125;
```

Особенность языка С: любые операции допускаются только со **скалярными** объектами, причем небольшого размера порядка размера регистров процессора. Это объясняется ориентацией языка на задачи системного программирования. Любые действия с громоздкими объектами - массивами, строками (частный случай массива), структурами реализуются посредством операции вызова функций. Например, рассмотрим сложные объекты

```
int x[100], y[100];
char c[80];
struct {
 int code;
 float data[1000];
} s1, s2;
```

Примеры правильной реализации операций присваивания для таких объектов:
содержимое массива y присвоить содержимому массива x

```
memcpy(x,y,sizeof(x));
```

массиву c присвоить значение строки "0123456789"

```
strcpy(c,"0123456789");
```

содержимое структуры s2 присвоить содержимому структуры s1

```
memcpy(&s1,&s2,sizeof(s1));
```

Операции присваивания в стиле языка PL/1 или dBASE вида

```
x=y, c="0123456789", s1=s2
```

в языке С интерпретируются как присваивание значений указателей на объекты. Здесь в левой части - константы, поэтому приведенные операции обнаружит как ошибочные компилятор [1].

1.3.3 Арифметические операции

Перечень арифметических операций в языке С и их обозначений:
+ - сложение;
- - вычитание либо изменение знака;
/ - деление (при целочисленных операндах - с отбрасыванием остатка) ;
* - умножение;
% - остаток от деления целочисленных операндов со знаком первого операнда (деление по модулю).

Как и в других языках высокого уровня, допустимым являются унарные операции (+ -). Операндами арифметических операций могут быть любые арифметические выражения. Тип выражений при необходимости приводится к

более масштабному для обеспечения правильности результата. Обязательные преобразования даже однотипных операндов перед выполнением арифметических операции:

```
float --> double; char, short --> int.
```

Необязательные преобразования разнотипных операндов:

```
int --> unsigned -->long --> double.
```

Единственной исключительной ситуацией при выполнении арифметических операций считается деление на нуль, а другие виды ситуаций (переполнение, исчезновение порядка или потеря значимости) игнорируются. Операции (* / %) обладают приоритетом над операциями (+ -), поэтому при записи сложных выражений можно использовать общепринятые математические правила [1]:

```
x+y*z-a/b <===> x+(y*z)-(a/b)
```

1.3.4 Операции отношений

Перечень операций отношений между объектами в языке С и их обозначения:

== - равно или эквивалентно;

!= - не равно;

< - меньше;

<= - меньше либо равно;

> - больше;

>= - больше либо равно.

Символы пар символов {==,!=,<=,>=} разделять нельзя.

Общий вид операций отношений

```
<выражение_1> <знак_операции> <выражение_2>
```

Операндами могут быть любые скалярные типы. Значения операндов после вычисления перед сравнением преобразуются к одному типу. Арифметические операнды преобразуются по правилам, аналогичным для арифметических операций. Операнды-указатели преобразуются в целые числа необходимого типа. Результат сравнения указателей будет корректным в арифметическом смысле лишь для объектов одного массива (операции над указателями см. в 3.11). Результат операции отношения - целое значение 1, если отношение истинно, или 0 в противном случае. Следовательно, операция отношения может использоваться в любых арифметических выражениях.

Примеры использования операций отношений:

```
y>0
```

```
x==y
x!=2
y=x*(z>2)+(d==1);
```

Рассмотрим примеры ошибочного использования операций отношений в синтаксически правильных выражениях.

ПРИМЕР 1. Попытка конструирования выражений в математическом стиле записи:

```
0<x<100  <===> (0<x)<100  <===> 1 /* Тавтология */
```

Правильные варианты выражения:

```
 0<x && x<100
(0<x)&&(x<100)
(0<x) * (x<100)  /* Допустимо, но неэффективно */
```

(символы && обозначают операцию конъюнкции - см. 3.5).

ПРИМЕР 2. Попытка сравнения сложных объектов в различных областях памяти по их адресам:

```
char x[40]; x=="Фамилия" <===> 0
```

Отношения между нескалярными объектами приходится проверять посредством последовательного сравнения их элементов. Удобно для этих целей воспользоваться библиотечными функциями. Например, требуемое выше сравнение строк символов можно записать в виде

```
strcmp(x,"Фамилия")
```

(функция int strcmp(char *x,char *y) выполняет лексикографическое сравнение двух строк: нуль - совпадение строк, положительное число - признак "x>y", отрицательное число - признак "x<y").

Сложные объекты можно рассматривать как массивы символов типа unsigned char. Сравнение массивов s1 и s2 символов такого типа длиной n байт выполняет функция

```
int memcmp(void *s1, void *s2, unsigned n),
```

(результат формируется подобно функции strcmp).

Например, рассмотрим сложные объекты

```
int x[100], y[100];

struct {
 int code;
 float data[1000];
} s1, s2;
```

Примеры правильной реализации операций сравнения для таких объектов:

```
memcmp(x,y,sizeof(x)) - сравнение массивов x и y;
memcmp(&s1,&s2,sizeof(s1)) - сравнение структур s1 и s2.
```

Упомянутые библиотечные функции могут правильно выявлять эквивалентность содержимого полей памяти. Использование их для установления отношений порядка для арифметических данных (напр., "больше", "меньше") приведет к ошибке, если игнорировать внутреннее представление данных. Например, рассмотрим два числа: 0x01020304l и 0x00000005l. Их представление в памяти ППЭВМ на основе микропроцессора 8086/808286 <04030201> и <05000000>. Функция memcmp при упрощенной интерпретации ее результата "установит", что первое число больше второго [1].

1.3.5 Логические операции

Перечень логических операций в языке С в порядке убывания относительного приоритета и их обозначения:

! - отрицание (логическое НЕТ);

&& - конъюнкция (логическое И);

|| - дизъюнкция (логическое ИЛИ).

Символы пар символов {||,&&} разделять нельзя.

Общий вид операции отрицания

```
!<выражение>
```

Общий вид операций конъюнкции и дизъюнкции

```
<выражение_1><знак_операции><выражение_2>
```

Операндами логической операции могут быть любые скалярные типы. Ненулевое значение операнда трактуется как "истина", а нулевое - "ложь". Результатом операции может быть 1 либо 0 целого типа.

```
!0 <===> 1 !5 <===> 0
int x,y; x=10; !((x=y)>0) <===> 0
```

Относительный приоритет логических операций позволяет пользоваться общепринятым математическим стилем запмси сложных логических выражений. Например, выражение

```
!x && (y>0) || (z==1) && (k>0)
```

эквивалентно следующему варианту выражения со скобками

```
((!x) && (y>0)) ||((z==1) && (k>0))
```

Особенность операций конъюнкции и дизъюнкции - экономное последовательное вычисление выражений-операндов:

– если выражение_1 операции конъюнкции равно нулю, то результат операции - нуль, а выражение_2 не вычисляется;

– если выражение_1 операции дизъюнкции не равно нулю, то результат операции - единица, а выражение_2 не вычисляется.

Таким образом, появляется возможность записью логического выражения задать условную последовательность вычисления выражений в направлении слева направо:

`scanf("%d",&i)&&test1(i)&&test2(i)` - нулевой результат одной из функций приведет к игнорированию вызова остальных;

`search1(x)||search2(x)||search3(x)` - только ненулевой результат одной из функций приведет к игнорированию вызова остальных;

`char *p=NULL; /* ... */ p && p[0]=='*'` - проверить первый элемент массива, если указатель на массив не пустой [1].

1.3.6 Операции над битами

Перечень операций над битами в языке С и их обозначения:

~ - инвертирование (одноместная операция);

& - конъюнкция;

| - дизъюнкция;

^ - исключающее ИЛИ (сложение по модулю 2);

>> - сдвиг вправо;

<< - сдвиг влево.

Символы пар символов {>>,<<} разделять нельзя.

Общий вид операции инвертирования

```
~<выражение>
```

Остальные операции над битами имеют вид

```
<выражение_1><знак_операции><выражение_2>
```

Операндами операций над битами могут быть только выражения, приводимые к целому типу. Операции {~,&,|,^} выполняются поразрядно над всеми битами операндов (знаковый разряд особо не выделяет ся):

```
     ~ 0xF0 == 0x0F
0xFF & 0x0F == 0x0F
0xF0 | 0x11 == 0xF1
0xF4 ^ 0xF5 == 0x01
```

Операции сдвига выполняются также для всех разрядов с потерей выходящих за границы битов:

```
0x8001<<1 == 0x0002
0x8001>>1 == 0x4000
```

Если "выражение_1" имеет тип unsigned, то при сдвиге вправо освобождающиеся разряды гарантированно заполняются нулями (логический сдвиг). Выражения типа signed могут, но не обязательно, сдвигаться вправо с копированием знакового разряда (арифметический сдвиг). При сдвиге влево освобождающиеся разряды всегда заполняются нулями. Если "выражение_2" отрицательно либо больше длины "выражения_1" в битах, то результат операции сдвига не определен и является системно зависимым. Операции сдвига вправо на k разрядов весьма эффективны для деления, а сдвиг влево - для умножения целых чисел на 2**k:

```
x<<1  <===> x*2
x>>1  <===> x/2
x<<3  <===> x*8
```

Подобное применение операций сдвига безопасно для беззнаковых и положительных значений "выражения_1". Для ППЭВМ на базе микропроцессоров типа 8086/80286 выигрыш во времени выполнения составляет 10...90 раз. Двухместные операции над битами (&,|,^,<<,>>) могут использоваться в сокращенных формах записи операции присваивания:

```
int i,j,k; ...
i|=j <===> i=i|j      /* Включение в поле i единиц из поля j */

i&=0xFF <===> i=i&0xFF      /* Выделение в поле i единиц по маске
поля 0x00FF */

k^=j      /* Выделение в поле k отличающихся разрядов в полях k и j
*/

i^=i      /* Обнуление всех разрядов поля i */
```

Операции над битами реализуются, как правило, одной машинной командой и рекомендуются для использования во всех подходящих случаях. В математическом смысле операнды логических операций над битами можно рассматривать как отображение некоторых множеств с размерностью не более разрядности операнда на значения {0,1}. Пусть единица означает обладание элемента множества некоторым свойством, тогда очевидна теоретико-множественная интерпретация рассматриваемых операций:

```
~ - дополнение; | - объединение; & - пересечение.
```

Простейшее применение - проверка нечетности целого числа:

```
int i; ... if (i&1) printf("значение i четно!");
```

Комбинирование операций над битами с арифметическими операциями часто позволяет упростить выражения. Например, получение размера области в параграфах размером 16 байт для размещения объекта размером x байт:

```
(x+15)>>4
```

Другие возможности оперирования над битами:

– использование структур с битовыми полями (см. 2.5);
– доступ к битам как разрядам арифметических данных.

Например, байт атрибута символов, выводимых на экран в текстовом режиме работы дисплея, имеет структуру:

```
struct attr {
 unsigned m:1; /* Признак мерцания символа */
 unsigned b:3; /* Цвет фона */
 unsigned f:4; /* Цвет символа */
};
```

Содержимое байта атрибута можно определить выражением [1]:

```
int bl; /* Признак мерцания символа */
int bg; /* Цвет фона */
int fg; /* Цвет символа */

bg*16+fg+128*(bl>0)
```

1.3.7 Условное вычисление

Вид записи операции:

```
условие? выражение_1:выражение_2
```

Результат операции - значение "выражения_1", если выражение "условие" отлично от нуля, иначе - значение "выражения_2". Условное вычисление применимо к арифметическим операндам или операндам-указателям:

```
int a,x;
x=(a<0)? -a: a
printf("Значение %d %sнулевое !",x,x? "не":"");
```

Отличительная особенность операции условного вычисления последовательное вычисление сначала выражения-условия, а затем выражения_1 или выражения_2.

Другая особенность операции условного вычисления - возможность использования результата операции в левой части оператора присваивания [1]:

```
#include "stdio.h"

int main(int argc, char* argv[])
{
 int i=1, j=2, k=3;

 (i>0? j : k )=0;
```

```
 printf("\n i=%d, j=%d, k=%d",i,j,k);
 (i>0? j : k )++;
 printf("\n i=%d, j=%d, k=%d",i,j,k);
 return 0;
}
```

1.3.8 Определение размера объекта

Операция определения размера объекта требуется, например, на этапе программного управления памятью. Виды записи операции:

```
sizeof выражение
sizeof(тип)
```

Частным случаем выражения является имя объекта. Результат операции - число байт для размещения объекта с указанным именем либо заданного типа:

```
sizeof(int) /* Длина поля данных типа int */
sizeof(a)   /* Длина поля объекта a */
sizeof(2+3) /* Длина поля хранения результата суммирования */
```

Операция sizeof - операция этапа компиляции. По существу, она является специальным обозначением константы типа unsigned int. Очевидно, что операция sizeof не может быть в левом операндом операции присваивания. Одно из полезных применений операции sizeof - определение количества элементов массива с объявленной размерностью:

```
char s[111];
...
sizeof(s)/sizeof(*s) /* Количество элементов массива s */
```

(применение подобного подхода для многомерных массивов рассмотрено в 3.11). Наиболее часто операция sizeof применяется при динамическом распределении памяти:

```
float *x; /* Указатель массива */
int n;    /* Количество элементов массива */

/* Захват и очистка памяти */

x=calloc(n,sizeof(*x));
```

Следует учитывать, что размер структуры при учете компилятором правил выравнивания адресов ее элементов может оказаться больше суммы размеров элементов. По этой причине планирование размещения в памяти составных объектов следует проводить на основе обращения к операции sizeof с именем

объекта. Например, пусть требуется определить размер буфера для размещения различных структур данных:

```
struct s1 { ... };
struct s2 { ... };
...
struct sn { ... };
```

Определив вспомогательный тип

```
union buffer {
 struct s1 x1;
 struct s2 x2;
 ...
 struct sn xn;
};
```

легко получить размер буфера: sizeof(union buffer) [1];

1.3.9 Операция приведения типа

Вид записи операции:

```
(тип)выражение
```

Ее результат - значение "выражения", преобразованное к заданному типу представления. Операция приведения типа вынуждает компилятор согласиться с выполнением указанного преобразования, но ответственность за последствия возлагается на программиста. Рекомендуется использовать эту операцию в исключительных случаях. Рассмотрим примеры таких случаев.

Ранее отмечалось, что результат арифметической операции в математическом отношении будет корректным в рамках конкретных аппаратных возможностей. Например:

```
unsigned char x=150, y=128;
unsigned char z=x+y;          // z!=150+128
```

Ошибка, появляющаяся здесь в операции присваивания, устраняется использованием другого типа поля для сохранения результата:

```
int  a;                  // a==150+128
a=x+y;                   // a==150+128
```

Пример оправданного приведения типа:

```
float x;
int n,l;
...
```

```
x=(float)(n+l)/3;
```

Использование операции приведения здесь позволяет избежать округления результата деления целочисленных операндов, т.к. после преобразования одного из операндов к типу float все операнды будут автоматически приведены к типу double. Корректными и прогнозируемыми являются операции сложения и вычитания над парами операндов (указатель_объекта_типа_X - арифме тическое_значение), а также вычитания указателей объектов одного массива). Потребность в операции приведения возникает в случае появления неопределенности типа операнда и при необходимости обхода запретов языка на комбинации операндов. Неопределенность типа операнда возникает, например, при передаче значения аргумента функции при отсутствии в ее описании описания списка параметров. Компилятор предупреждает программиста о подобных ситуациях, но иногда неопределенность порождена отсутствием достаточной информации.

Пример неопределенности связи параметров:

```
#include <stdio.h>

void main() {
 float f=125.0;

/* Надежда на интеллект компилятора */

 printf("\n??? %f %d %x %f",f,f,f,f);

/* На компилятор надейся, но приводи тип в соответствие */

 printf("\n*** %f %d %x %f",f,(int)f,(int)f,f);
}
```

Результаты работы программы:

```
??? 125.000000 0 0 0.000000
*** 125.000000 125 7d 125.000000
```

Примеры запрещенных видов операции присваивания:

– указатель=арифметическое_значение;
– арифметическая_переменная=указатель;
– указатель_объекта_типа_1=указатель_объекта_типа_2, если (тип_1!=тип2)&&(тип_1!=void)&&(тип_2!=void).

Многие задачи системного и прикладного программирования вынуждают использовать перечисленные операции. При этом приходится непосредственно сталкиваться с особенностями реализации вычислительной среды, а программы легко теряют свойство мобильности. Например, пусть необходимо получить доступ к видеопамяти текстового режима работы дисплея по известному адресу 0xb8000 для видеоадаптеров CGA, EGA или VGA ПЭВМ класса IBM PC/XT/AT:

```
/* Правильный вариант описания видеопамяти */

     struct item {
      char byte; // отображаемый символ
      char attr; // атрибуты отображения
     } *screen_pointer=(struct item far *)MK_FP(0xb800,0);
```

Здесь учтены: формат элемента видеопамяти; невозможность относительной адресации видеопамяти в сегментах программы пользователя ("близкими" указателями с атрибутом near); использование уникального способа представления "далекого" (far) указателя и т.д.

Игнорирование, например, последнего фактора приведено в синтаксически правильном операторе

```
screen_pointer=(struct item far *)0xb8000l;
```

Попытка после этого очистить поле экрана

```
for (int i=0; i<2000; screen_pointer[i++].byte=' ');
```

закончится крахом системы. Способ представления далеких указателей в системах программирования C и C++ определен макрокомандой

```
#define MK_FP(seg,ofs) \((void far *)(((unsigned
long)(seg)<<16)|(unsigned)(ofs)))
```

Результат преобразования объекта, не являющегося указателем, не может быть LVALUE, поэтому значение ему присваивать нельзя. Для указателя же допустимым является выражение

/* Правильный вариант установки указателя видеопамяти */

```
(unsigned long)screen_pointer=0xb80000001;
```

Подобные операции характерны для системного программирования, но при этом желательно максимально использовать высокоуровневые лингвистические конструкции (см. первоначальный вариант).

Явное приведение типов указателей позволяет получить адрес объекта любого типа:

```
any_type* p;


p=(any_type*)&some_object;
```

Известное значение указателя p позволяет работать с некоторым объектом some_object как объектом типа any_type:

```
   /* Пример программы печати представления числа типа long */

#include <stdio.h>
```

```
long x=0xB80000001;
unsigned char *c;

void main () {
 int i;

 c=(char *)&x;
 for (i=0; i<sizeof(x); i++)
  printf(" %02x",c[i]);
}
```

Результат работы программы:

```
00 00 00 b8
```

Явного преобразования типов рекомендуется, по возможности, из бегать. Появление неожиданных результатов операций во внешне правильных выражениях и исходных данных - сигнал для анализа и коррекции хода преобразований. Рекомендуется максимально использовать стандартные библиотечные функции и макрокоманды [1].

1.3.10 Последовательное вычисление выражений

Вид записи операции:

```
e1, e2, ..., eN
```

Смысл операции - гарантированно последовательно вычисляются выражения e1... eN, а результат операции - значение выражения eN [1].

Пример:

```
m=(i=1, j=i++, k=0, l=i+j); /* i=1; j=i++; k=0; l=i+j; m=l; */
```

1.3.11 Операции над указателями

Обращение к объектам любого типа как операндам операций в языке С может проводиться как по имени так и по указателю. Использование в исходном тексте операции косвенной адресации, записываемой в виде

```
*указатель_объекта
```

позволяет получить доступ к полю объекта, адрес которого - "указатель_объекта". Значение адреса объекта указателю можно присвоить, например, используя операцию определения адреса, вид записи которой

```
&идентификатор_объекта
```

(здесь в качестве объекта нельзя использовать битовые поля структур или объединений, а также объекты с атрибутом класса памяти register). Пример идентификации объектов:

```
int i,  /* Переменная типа int */
    *x; /* Указатель на элемент данных типа int */

x=&i;  /* x <- адрес переменной i */
*x=1;  /* Косвенная адресация указателем поля i */
```

Говорят, что использование указателя означает отказ от именования (разыменование) адресуемого им объекта:

```
int i,j,k, *x;
x=&i;
*x=0;     /* i=0;  */
x=&j;
*x+=i;    /* j+=i; */
x=&k;
k+=*x;    /* k+=k; */
(*x)++;   /* k++;  */
```

Отказ от именования объектов при наличии возможности доступа по указателю приближает язык С по гибкости отображения "объект память" к языку ассемблера. При вычислении адресов объектов следует учитывать, что идентификатор массива и функции именует переместимую адресную константу (термин ассемблера) - константу типа указатель (термин языка С). Такую константу можно присвоить переменной типа указатель, но нельзя подвергать преобразованиям:

```
int x[100], *y;
void getsn(), (*proc)();

y=x;         /* Присваивание константы переменной */
proc=getsn;
/* ... */
x=y;     /* Ошибка: в левой части - указатель-константа */
```

Указателю-переменной можно присвоить значение другого указателя либо выражения типа указатель с использованием при необходимости операции приведения типа (приведение необязательно, если один из указателей имеет тип "void *").

```
int i,*x;
char *y;

x=&i;              /* x --> поле типа int */
y=(char *)x;       /* y --> поле типа char */
y=(char *)&i;      /* y --> поле типа char */
```

Указатель может использоваться в выражениях вида

```
p+ie, p-ie,
++p, --p,
```

```
p++, p--,
p+=ie, p-=ie,
```

где p - указатель, ie - целочисленное выражение. Значение таких выражений - увеличенное или уменьшенное значение указателя на величину ie*sizeof(*p). Текущее значение указателя всегда ссылается на позицию некоторого объекта в памяти с учетом правил выравнивания для соответствующего типа данных. Таким образом, значение p [+|-] ie указывает на объект того же типа, расположенный в памяти со смещением на [+|-] ie позиций:

```
#include <stdio.h>
void main() {
 int i;
 double *pd;
 char *pc;
 int  *pi;
 struct test {
  int x[10];
  char b[83];
  float c[15];
 } *ps;
 pd=(double *)&i;
 pc=(char   *)&i;
 pi=(int    *)&i;
 ps=(struct test *)&i;
 printf("\n pd %p %p %p %d",pd,pd+1,pd-1,sizeof(*pd));
 printf("\n pc %p %p %p %d",pc,pc+1,pc-1,sizeof(*pc));
 printf("\n pi %p %p %p %d",pi,pi+1,pi-1,sizeof(*pi));
 printf("\n ps %p %p %p %d",ps,ps+1,ps-1,sizeof(*ps));
}
```

Результаты работы программы:

```
pd 0FD0 0FD8 0FC8 8
pc 0FD0 0FD1 0FCF 1
pi 0FD0 0FD2 0FCE 2
ps 0FD0 1073 0F2D 163
```

Как следствие, разрешается сравнивать указатели и вычислять разность двух указателей. При сравнении могут проверятся отношения любого вида (">",">=","<","<=","==","!="). Наиболее важными видами проверок являются отношения равенства или неравенства. Отношения порядка имеют смысл только для указателей на последовательно размещенные объекты (элементы одного массива). Разность двух указателей дает число объектов адресуемого ими типа в соответствующем диапазоне адресов. Очевидно, что уменьшаемый и вычитаемый указатель также должны соответствовать одному массиву, иначе результат операции не имеет практической ценности. Любой указатель можно сравнивать со значением NULL, которое означает недействительный адрес.

Значение NULL можно присваивать указателю как признак пустого указателя. NULL заменяется препроцессором на выражение (void *)0 (см. 6.2). Рассмотрим связь указателей и массивов. Пусть объявлены:

```
type p[100]; /* Массив 100 последовательно
                размещенных объектов типа type */
type *q;     /* Указатель на объект типа type */
int i;       /* Индекс элемента */
```

Обращение к элементу массива в языке С возможно в традиционном для многих других языков стиле записи операции обращения по индексу (элементы массива нумеруются 0,1,...) :

```
p[0]=1; /* Первый элемент массивов имеет нулевой индекс */
p[i]++; /* Инкремент элемента массива */
p[3]=p[i]+p[i+1]; /* Ограничений на доступ к массиву нет */
```

Если выполнен оператор

```
q=p; /* Присваивание константы переменной */
```

то из раннее сказанного следует, что выражения p[i] и *(q+i) приводят к одинаковым результатам. Учитывая, что p и q - указатели, легко догадаться, что в языке С выражения p[i] и *(p+i) эквивалентны. Отсюда следует, что операция обращения к элементу массива по индексу применима и при его именовании указателем-переменной. Таким образом, для любых указателей можно использовать две эквивалентые формы выражений для доступа к элементам массива: q[i] и *(q+i). Первая форма удобнее для читаемости текста. Дополнительно отметим, что в языке С с целью повышения быстродействия программы отсутствует механизм контроля границ изменения индексов массивов. При необходимости такой механизм должен быть запрограммирован явно. Рассмотрим некоторые важные для практического программирования следствия.

Следствие 1. Очевидна эквивалентность выражений

```
&q[0] <===>&(*q) <===> q


*q <===> q[0]
```

Последнее выражение объясняет правильность выражения для получения количества элементов массива с объявленной размерностью:

```
type x[100]; /* Размерность должно быть константой */
...
int n=sizeof(x)/sizeof(*x);  /* n <-- 100 */
```

Следствие 2. В языке С размерность массива при объявлении должна задаваться константным выражением. При необходимости работы с массивами переменной размерности вместо массива достаточно объявить указатель требуемого типа и присвоить ему адрес свободной области памяти. Библиотечные функции для получения памяти, ее освобождения и оценки размера остатка памяти описаны в файле alloc.h.

```
/* Пример создания динамического массива */

     float *x;
     int    n;
     ...
     while (printf("\nРазмерность - ? "), !scanf(" %d",&n));

     if ((x=calloc(n,sizeof(*x)))!=NULL) {
      printf("\nМассив создан и очищен!");
      ...
      for (i=0; i<n; i++)
       printf("\n%f",x[i]);
      ...
      free(x); /* Освобождение памяти */
     } else {
      printf("\nПредел размерности %d",coreleft()/sizeof(*x));
      exit(1);
     }
```

Следствие 3. Ранее отмечалось, что строковые константы в памяти представляются массивом символов, дополненным символом конца строки '\0'. Отсюда следует допустимость выражений:

```
char *x;
...
x="МРТИ";
x=(i>0)? "положительное":(i<0)? "отрицательное":"нулевое";
```

Сказанное относительно связи указателей и массивов с одним измерением справедливо и для массивов с большим числом измерений. Например, рассмотрим двумерный массив

```
type name[5][10];
```

Если рассматривать его как массив пяти массивов размерностью по десять элементов каждый, то очевидна схема его размещения в памяти - последовательное размещение "строк". Обращению к элементам name[i][j] соответствует эквивалентное выражение

```
*(*(name+i)+j),
```

а интерпретация такого массива указателем переменной выглядит в форме

```
type **pname;
```

Таким образом, имя двумерного массива - имя указателя на указатель. Аналогичным образом можно установить соответствие между указателями и массивами с произвольным числом измерений:

```
type name[][][][]; <===> type ****pname;
```

В последнем случае эквивалентными являются выражения:

```
name[i][j][k][l]
*(*(*(*(name+i)+j)+k)+l)
*(*(*(name+i)+j)+k)[l]
*(*(name+i)+j)[k][l]
*(name+i)[j][k][l]
```

Приведем пример программы конструирования массива массивов:

```
#include <stdio.h>

int x0[3]={ 1, 2, 3};     /*********************************/
int x1[3]={11,12,13};     /*  Декларация и инициализация   */
int x2[3]={21,22,23};     /*     массивов целых чисел      */
int x3[3]={31,32,33};     /*********************************/

int *y[4]={x0,x1,x2,x3}; /* Создание массива указателей  */

void main() {
 int i,j;

 for (i=0; i<4; i++) {
  printf("\n %2d)",i);
  for (j=0; j<3; j++)
   printf(" %2d",y[i][j]);
 }
}
```

Результаты работы программы:

```
  0)  1  2  3
  1) 11 12 13
  2) 21 22 23
  3) 31 32 33
```

Такие же результаты будут получены и следующей программой:

```
#include <stdio.h>

int z[4][3]={{ 1, 2, 3},    /*********************************/
             {11,12,13},    /*  Декларация и инициализация   */
             {21,22,23},    /* массива массивов целых чисел */
             {31,32,33}     /*********************************/
             };

void main() {
 int i,j;
```

```
 for (i=0; i<4; i++) {
  printf("\n %2d)",i);
  for (j=0; j<3; j++)
   printf(" %2d",z[i][j]);
 }
}
```

В последней программе массив указателей на массивы создается компилятором. Здесь собственно данные массива располагаются в памяти последовательно по строкам, что является основанием для декларации массива z в виде:

```
int z[4][3]={  1, 2, 3, /*********************************/
              11,12,13, /*  Декларация и инициализация  */
              21,22,23, /* массива массивов целых чисел */
              31,32,33  /*********************************/
            };
```

Замена скобочного выражения z[4][3] на z[12] здесь не допускается, так как массив указателей не будет создан. Таким образом, использование многомерных массивов в языке С связано с накладными расходами памяти на массивы указателей (этот недостаток в современных версиях устранен). Можно избежать таких расходов, если ввести адресную функцию для доступа к элементам одномерного массива по значениям индексов многомерного массива. Например, функция вывода двумерного массива произвольной размерности в стиле рассмотренных выше примеров имеет вид:

```
print(int *array, int N_row, int N_col) {
 int i,j;

 for (i=0; i<N_row; i++) {
  printf("\n %2d)",i);
  for (j=0; j<N_col; j++)
   printf(" %2d",array[N_col*i+j]);
 }
}
```

Нетрудно заметить, что после объявления многомерного массива, например, в виде

```
type s[10][5][3];
```

синтаксически правильными будут выражения:

```
s[i]       <===>  *(s+i)            <===> &s[i][0]
s[i][j]    <===>  *(*(s+i)+j)       <===> &s[i][j][0]
s[i][j][k] <===>  *(*(*(s+i)+j)+k)
```

Отсутствие контроля за значениями индексов влечет допустимость выражений

```
s[1][-2][4]
```

Отсюда следует, что ни количество измерений массива, ни диапазон значений индекса по отдельному измерению в языке С не контролируются, а имена массива и указателя на данные часто взаимозаменяемы.

Декларация массива перекладывает операции распределения памяти на компилятор (см. 5.3). К любым компонентам многомерного массива применима операция sizeof:

```
int s[10][5][3];

    /* Истинные выражения:

       sizeof(s)==300
       sizeof(*s)==30
       sizeof(**s)==6
       sizeof(***s)==2
       sizeof(s)/sizeof(*s)==10
       sizeof(*s)/sizeof(**s)==5
       sizeof(**s)/sizeof(***s)==3

     */
```

Здесь компилятор полностью осведомлен об интерпретации операндов. В случае косвенной ссылки на существующий массив, например, после записи оператора

```
int ***t=s;
```

синтаксически правильным будет только выражение sizeof(t). Указатели позволяют:

– работать с массивами переменной размерности и другими сложными структурами данных;

– заменить пересылку данных большого размера изменением значения указателя [1].

1.3.12 Операция вызова функции

Операция вызова функции в общем случае требует наличия в файле исходного текста декларации функции в форме определения или описания. При отсутствии декларации функции предполагается, что вызывается функция, возвращающая значение типа int.

Определение функции как процедурного модуля включает:

а) тип результата;

б) идентификатор (указатель-константа);

в) список формальных параметров с описанием их типов;

г) составной оператор (блок), представляющий выполняемые функцией действия.

Пример определения функции:

а) классический формат записи

```
                                  ---------------------------------
int poly2(a,b,c,x)
 int a;                              Заголовок функции
 int b;
 int c;
 int x;
{                                 ---------------------------------
 return (c+x*(b+x*a));               Тело функции
}                                 ---------------------------------
```

б) современный формат записи (стандарт языка C++)

```
int poly2(int a, int b, int c, int x) {
 return (c+x*(b+x*a));
}
```

Описание функции используется лишь для уведомления компилятора о том, что функция определена позднее в текущем или другом файле исходного текста либо находится в библиотеке. Формы описания приведенного примера функции:

а) классический формат записи - тип результата, идентификатор и круглые скобки

```
int poly2();
```

б) современный формат записи - классический формат дополнен списком типов параметров с необязательными идентификаторами параметров:

```
int poly2(int a, int b, int c, int x);
int poly2(int, int, int, int);
```

Описание (прототип) функции в современных системах программирования позволяет улучшить диагностику правильности ее вызова и выполнить преобразования параметров. Наличие определения функции делает излишним запись описания в остатке файла исходного текста. Функция, которая не возвращает значения, должна описываться как имеющая тип void. Способы вызова функции

а) вызов по имени

```
имя_функции(e1,e2,...eN)
```

б) вызов по указателю

```
(*указатель_на_функцию)(e1,e2,...eN)
```

Здесь e1,e2,... eN - выражения, определяющие значения аргументов (фактических параметров). Порядок вычисления выражений перед вызовом не регламентирован. Список аргументов может быть пустым, но круглые скобки опускать в этом случае нельзя. Указатель_на_функцию - переменная, содержащая адрес функции, которая декларируется, напр., в виде

```
тип_функции (*указатель_на_функцию)();
```

Адрес функции присваивается указателю оператором

```
указатель_на_функцию = имя_функции;
```

```
(имя_функции - указатель_константа).
```

Результат операции вызова функции определяется типом возвращаемого функцией результата в ее описании. Операция вызова функций, не возвращающих значений, может применяться только в операторе вызова функции в форматах:

```
имя_функции(e1,e2,...eN); (*указатель_на_функцию)(e1,e2,...eN);
```

Другие функции могут использоваться в любых выражениях:

```
int getchar();
...
i=getchar(); /* Ввод символа */
...
getchar();   /* Ожидание нажатия любой клавиши */
```

Синтаксис записи списка аргументов совпадает с синтаксисом операции последовательного вычисления выражений. Конфликт устраняется использованием круглых скобок: ... printf("\n %d",(x=(i==1), i++));

Проверка числа и типа аргументов, переданных функции, а также типа возвращаемого значения во время выполнения не производится. Предполагается, что тип возвращаемого функцией результата совпадает с объявленным в ее описании. Аргументы (фактические параметры) функций в языке С всегда передаются по значению. Преобразование параметров в теле функции не вызывает изменения полей аргументов в вызывающей функции.

Особенности использования параметров функций:

– при необходимости изменения функцией некоторого поля небходимо использовать в качестве параметра указатель соответствующего типа;

– массивы (функции) всегда адресуются косвенно, т.е. имя массива - указатель-константа, использование которого в качестве параметра позволяет получить доступ к любому элементу массива;

– способ адресации производных типов данных (структуры, объединения и их массивы) зависит от реализации транслятора языка С, поэтому для обеспечения переносимости программ рекомендуется использовать в качестве параметра указатель таких типов данных.

```
int i;              /* Пример функции ввода данных */
scanf("%d",&i);     /* ************************** */
```

функция может вызываться рекурсивно, причем глубина рекурсии ограничивается только объемом доступной памяти для раззмещения переменных в динамической (стековой) памяти.

```
#include

void trans(register int x) {
 if (x) {
  trans(x>>4);
  printf("%c","0123456789abcdef"[x&0x0f]);
 } else printf("0x");
}


int main(int argc, char *argv[]) {
 int n;
 n = 0x01020304;
 printf("\n n = %d = %#08x = ",n,n);
 trans(n);
 return 0;
}
```

Описания библиотечных функций помещены в так называемые заголовочные файлы, содержимое которых может быть включено в исходный текст программы на этапе компиляции директивой препроцессора #include (см.1.6.4) [1].

1.3.13 Приоритет и порядок выполнения операций

В языке С операции делят на группы, имеющие атрибуты приоритета и порядка выполнения. Очередность выполнения операций в выражении при отсутствии скобок определяется приоритетом группы, а при принадлежности операций одной группе - порядком выполнения (см. таблицу).

Приоритеты и порядок выполнения операций

Приоритет	Группа операций	Порядок выполнения
1	2	3
1	() вызов функции	Слева направо

	[] выделение элемента массива или объединения -> выделение элемента структуры(объединения), адресуемой(ого) указателем	
2	! логическое отрицание ~ побитовое отрицание (дополнение) - изменение знака ++ увеличение на единицу -- уменьшение на единицу & определение адреса (указателя) * обращение по адресу (указателю) (тип) преобразование (приведение) типа sizeof определение размера в байтах	Справа налево
3	* умножение / деление % деление по модулю	Слева направо
4	+ сложение - вычитание	- "
5	<< сдвиг влево >> сдвиг вправо	- "
6	> больше >= больше или равно < меньше <= меньше или равно	- "
7	== равно != не равно	- "
8	& побитовая конъюнкция (И)	- "
9	^ побитовое исключающее ИЛИ	- "
10	| побитовая дизъюнкция (ИЛИ)	- "
11	&& логическая конъюнкция (И)	- "
12	|| логическая дизъюнкция (ИЛИ)	- "
13	?: условное вычисление	Справа налево
14	= присваивание *= умножение и присваивание /= деление и присваивание %= деление по модулю и присваивание += сложение и присваивание -= вычитание и присваивание <<= сдвиг влево и присваивание >>= сдвиг вправо и присваивание &= побитовая конъюнкция и присваивание ^= побитовое исключающее ИЛИ и присваивание |= побитовая дизъюнкция и присваивание	- " Справа налево
15	, последовательное вычисление	Слева направо

Примеры:

```
x*y/c      <=>  (x*y)/c       /* ==> */
a=b=c      <=>  a=(b=c)       /* <== */
alfa[i]()  <=>  (alfa[i])()   /* ==> */
```

В сложных выражениях иногда играет роль порядок обработки операндов. Только для четырех операций (&& || ?: ,) в языке С левый операнд гарантированно обрабатывается первым. С целью обеспечения переносимости (мобильности) программы рекомендуется не использовать повторно в выражении переменную, значение которой присваивается любым образом в этом выражении. Пример:

```
y=(x=1)-(++x);
```

Перепишем оператор формирования результата в виде

```
y=ol-or;
```

Если компилятор планирует вычисление операндов в скобках слева направо, то схема формирования значения y:

```
ol=1; or=2; y=-1;
```

(переменная x последовательно принимает значения 1 и 2). При вычислении операндов в обратной последовательности

```
or=x+1; ol=1; y=x;
```

(переменная x вначале увеличивается на 1, затем принимает значение 1). Результат присваивания переменных x и y оказался существенно различен [1].

1.4 УПРАВЛЯЮЩИЕ ОПЕРАТОРЫ

1.4.1 Условные операторы

В языке С имеется две разновидности условных операторов: простой и полный операторы условного выполнения. Синтаксис простого оператора условного выполнения:

```
if (выражение) оператор
```

Элемент "оператор" подлежит выполнению, если "выражение" от лично от нуля. Примеры записи:

```
if (x>0) x=0;

if (i!=1) j++, l=1; <===> if (i!=1) { j++, l=1; }

if (getch()!=27) {
 k=0;
}

if (i) exit(1);  <===>  if (i!=0) exit(1);
if (i>0) if (i<n) k++; <===> if ((i>0)&&(i<n)) k++;
if (1) i=0; <===> i=0;
```

Синтаксис полного оператора условного выполнения:

```
if (выражение) оператор_1
else оператор_2
```

Если "выражение" отлично от нуля, то подлежит выполнению "оператор_1", иначе - "оператор_2". Примеры записи:

```
if (x>0) j=k+1;
else m=i+10;
if (x) {
 y=i++;
 k=sfn(i);
} else {
 printf("\n ???");
 exit(0);
}
```

Элементы "оператор_1" и(или) "оператор_2" могут быть любым оператором, в том числе и условным оператором. Фраза "else ..." относится к непосредственно предшествующей ей фразе "if ...", поэтому для устранения коллизий условных операторов разных уровней вложенности необходимо заключать их в фигурные скобки [1]:

```
if (!x) if (!y) printf("\n x=YES, y=YES");
else printf("\n x=NO"); /* x=0 <===> (x!=0)&&(y==0) ??? */

if (!x) {
 if (!y) printf("\n x=YES, y=YES");
} else printf("\n x=NO");
```

1.4.2 Операторы цикла

Перечень разновидностей операторов цикла:

– оператор цикла с предусловием;
– оператор цикла с постусловием;
– оператор цикла с предусловием и коррекцией.

Синтаксис оператора цикла с предусловием:

```
while (выражение) оператор
```

(элемент "оператор" может быть пустым оператором, оператором-выражением или операторным блоком). Смысл оператора: на каждой итерации цикла предварительно проверяется условие продолжения цикла - ненулевое значение "выражения", затем выполняется "оператор". Элемент "оператор" может включать любое количество управляющих операторов, связанных с конструкцией while:

continue - переход к следующей итерации цикла;
break - выход из цикла.
Примеры записи:

```
while (i>0) i<<=1, j++;
while (printf("\n n-?"), scahf(" %d",&n));
   ...
while (1) { /* Организация бесконечного цикла */
 /* ... */
 if (kbhit()&&(getch()==27)) break; /* Выход по ESC */
 /* ... */
}
while (!kbhit()); /* Активное ожидание нажатия клавиши */
```

Синтаксис оператора цикла с постусловием:

```
do оператор while (выражение);
```

(элемент "оператор" может быть пустым оператором, оператором выражением или операторным блоком). Смысл оператора: на каждой итерации цикла предварительно выполняется "оператор", затем проверяется условие продолжения цикла - ненулевое значение "выражения". Назначение используемых в элементе "оператор" управляющих операторов continue и break совпадает с ранее рассмотренным, но оператор continue вызывает переход к этапу оценки "выражения".

```
do printf("\n ???");
while (!scanf(" %d",&n));
...
float *x;
int i=0;
...
m=coreleft()/sizeof(*x);
do {
 printf("\n n-?");
 if (!scanf(" %d",&n)) {
  sound(300);
  printf(" Вводите цифры !");
  continue;
 }
if (n>m) continue;
 if (!scanf(" %f",x+i)) break;
} while (++i<n);
```

Синтаксис оператора цикла с предусловием и коррекцией:

```
for (инициализация; условие_выполнения; коррекция) оператор
```

Здесь "инициализация", "условие_выполнения" и "коррекция" выражения, которые могут отсутствовать (пустые выражения), но символы ';' опускать нельзя. Оператор-выражение "инициализация" выполняется один раз перед

началом итераций цикла. Итерации цикла продолжаются, пока истинно "условие_выполнения" - выражение пустое либо непустое выражение отлично от нуля. На каждой итерации последовательно выполняются "оператор" и оператор-выражение "коррекция". Назначение используемых в элементе "оператор" управляющих операторов continue и break совпадает с ранее рассмотренным.

```
float  x[10], y;
int  i,n;
...
for (i=n=sizeof(x)/sizeof(*x); i>0; x[--i]=0);
...
for (i=0; i<n; x[i++]=0);

for (i=0; i<n; i++) x[i]=0;
...
for (i=0, i=0; i<n; i++)
 if (x[i]<0) y+=x[i];
...
```

Различные формы операторов цикла могут выражаться друг через друга, например:

а) оператор

```
for (инициализация; условие_выполнения; коррекция) оператор
```

эквивалентен последовательности операторов

```
инициализация;
while (условие_выполнения) {
 оператор;
 коррекция;
}
```

(здесь "оператор" не может включать операторы break или continue);

б) оператор

```
for (; условие_выполнения;) оператор
```

эквивалентен оператору

```
while (условие_выполнения) оператор;
```

в) оператор

```
for (оператор; условие_выполнения;) оператор
```

эквивалентен оператору

```
do оператор while (условие_выполнения);
```

г) оператор

```
for (;;) оператор
```

эквивалентен оператору

```
while (1) оператор;
```

(вместо 1 здесь может быть любое число, отличное от нуля).

При использовании вложенных циклов следует учитывать, что управляющие операторы break и continue действуют внутри собственного цикла. Для выхода из вложенного цикла приходится использовать оператор **безусловного** перехода goto [1]:

```
float x[10][20];
int i,j;

/* Проверка наличия отрицательных значений */

for (i=0; i<10; i++)
 for (j=0; j<20; j++)
  if (x[i][j]<0) goto next_step;

next_step: printf("\nЭлемент (%d,%d) отрицателен...");
```

1.4.3 Оператор выбора альтернатив (переключатель)

Синтаксис:

```
switch (выражение) { ----------------------------------

 case константа_1: оператор_1                Тело
 case константа_2: оператор_2              оператора
                    ...                     switch
 default:  оператор_N
} ------------------------------------------------------
```

Ключевое слово default и целочисленные значения констант рассматриваются как специальные метки операторов, область действия которых - тело оператора switch. Порядок следования таких меток не регламентирован. Целочисленное выражение сравнивается после вычис ления со значениями констант-меток. При совпадении с одной из них выполняется передача управления соответствующему оператору в теле оператора switch. В случае несовпадения значения выражения с одной из констант - переход на метку default либо, при ее отсутствии, к оператору, следующему за оператором switch. Упомянутые здесь специальные метки в теле оператора switch недолжны повторяться или использоваться для ссылок в операторе goto. Для выхода из тела оператора switch используют управляющий оператор break (дополнительно можно воспользоваться операторами goto или return, а при вложенности в оператор цикла и оператором continue). Рассмотрим пример построения

простейшего калькулятора для вычисления значений функций sin(), cos(), log(), sqrt(), tan().

```
#include <stdio.h>
#include <math.h>
#include <conio.h<

char f[]="\n %s(%lf)=%lf";

void main() {
 double x;

 while (sound(1000),
        printf("\n x-? "),
        nosound(),
        scanf("%lf",&x)) {
  for (;;) {
   printf("\n x=%lf, Sin, Cos, Log, sQrt, Tan - ? ",x);
   switch(getch()) {
    case 27 : goto cont; /* 27 - код клавиши ESC */
    case 's':
    case 'S': printf(f,"sin",x,sin(x));
              break;
    case 'c':
    case 'C': printf(f,"cos",x,cos(x));
              break;
    case 'l':
    case 'L': printf(f,"log",x,log(x));
              break;
    case 'q':
    case 'Q': printf(f,"sqrt",x,sqrt(x));
              break;
    case 't':
    case 'T': printf(f,"tan",x,tan(x));
              break;
    default:  sound(100);
    printf("\n Select Sin, Cos, Log, sQrt, Tan or ESC");
    nosound();
   }
  }
  cont:;
 }
}
```

Очевидно, что здесь оператор switch можно заменить вложенными условными операторами:

```
int i;
...
i=getch();
if ((i=='s')||(i=='S')) printf(f,"sin",x,sin(x));
else if ((i=='c')||(i=='C')) printf(f,"cos",x,cos(x));
...
```

```
else if ((i=='t')||(i=='T')) printf(f,"tan",x,tan(x));
else {
 sound(100);
 printf ...
 nosound();
}
```

В последнем варианте используется последовательная проверка условий, его вычислительная сложность 6/2 (половина количества альтернатив). Оператор switch реализуется переходом по адресу, выбираемому из инвертированной таблицы меток по значению выражения, поэтому его вычислительная сложность близка к единице [1].

1.4.4 Операторы передачи управления

Формально к операторам передачи управления относятся:

а) оператор безусловного перехода

```
goto метка;
```

б) оператор перехода к следующему шагу (итерации) цикла

```
continue;
```

(игнорирование оставшихся операторов тела цикла);

в) выход из цикла либо оператора switch

```
break;
```

(передача управления следующему оператору);

г) оператор возврата их функции

```
return;
return выражение;
```

(прекращение выполнения текущей функции и возврат управления вызвавшей программе с передачей, при необходимости, значения выражения).

Операторы вида а...в рассматривались ранее. Оператор return обязательно необходим в функциях, возвращающих значения. В функциях, не возвращающих результат, он неявно присутствует после последнего оператора. Значение "выражения" при необходимости будет преобразовано к типу возвращаемого функцией значения.

```
void error(void *x, int n) {
 if (!x) printf("\nМассив не создан");
 if (!n) printf("\nМассив пустой");
}

float estim(float *x, int n) {
 int i;
```

```
 float y;
 if ((!x)||(!n) {
  error(x,n);
  return 0;
 }
 for (y=i=0; i<n; i++)
  y+=x[i];
 return y/n;
}
```

Строго говоря, в языке С имеются дополнительные возможности передачи управления:

– нелокальный переход, организуемый парой функций longjump и setjump;

– операторы структурного управления исключениями _try/_except и _try/finally [1].

1.5 СТРУКТУРИЗАЦИЯ ПРОГРАММ И ДАННЫХ

1.5.1 Области действия объектов программ

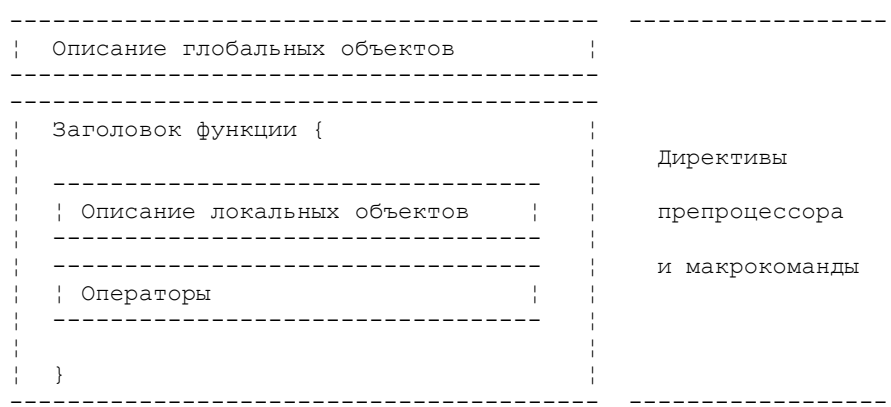
В языке С можно выделить два уровня детализации модуля программы: файл исходного текста и функцию. Файл исходного текста может включать определения данных и функций. Операционные объекты, кроме атрибута типа, имеют атрибуты области действия и класса памяти. По области действия разделяют объекты:

– глобальные - доступны во всех функциях файла исходного текста;

– локальные - доступны только в теле функции или операторного блока.

В языке С нет ключевого слова, указывающего область действия объекта. Область действия определяется местоположением оператора описания объекта в файле исходного текста программы (любой объект полностью описывается в одном операторе).

Структура исходного текста программ на языке С



Файл исходного текста может включать любое количество определений функций и(или) глобальных данных. Параметры функции являются локальными объектами и должны отличаться по именам от используемых в теле функции глобальных объектов. Локальные объекты, описанные в теле функции, имеют приоритет перед объектами, описанными вне функции:

```
int n;       /* Глобальное n */

void main (int n, char **l) {
 int i;
/* ... */
 f1(i);
/* ... */
 f2(n);      /* Локальное n */
}

f1(int i) {
 /* ... */
  i=n;       /* Глобальное n */
 /* ... */
}
f2(int n) {
 int i;
 /* ... */
  i=n;        /* Локальное n */
 /* ... */
}
```

Локальные объекты в программах на языке С можно декларировать в начале любого операторного блока, а операторный блок тела функции - частный случай такого правила (в языке С++ оператор декларации может размещаться в любом месте операторного блока). В любом месте файла исходного текста можно ссылаться на глобальные объекты, определенные ниже в остатке текущего файла или в других файлах. Для этого необходимо описать тип объекта и добавить к описанию ключевое слово extern. Описания функций подразумевают атрибут extern по умолчанию. Разрешается опускать длину внешних одномерных массивов, но операция sizeof к таким массивам становится бессмысленной. Следует учитывать, что любой оператор описания действует только на остаток файла исходного текста [1].

1.5.2 Классы памяти объектов программ

Классы памяти объектов в языке С:

– статическая память - распределяется на этапе трансляции и заполняется по умолчанию нулями;

– динамическая память - выделяется при вызове функции и освобождается при выходе из функции.

Объекты, размещаемые в статической памяти, описываются с атрибутом static и могут иметь любой атрибут области действия. Значения локальных статических объектов сохраняются при повторном вызове функции. Глобальные объекты всегда являются статическими. Атрибут static, использованный при описании глобального объекта, предписывает ограничение области его применимости только в пределах остатка текущего файла. Таким образом, в языке С ключевое слово static имеет разный смысл для локальных и глобальных объектов. В динамической памяти могут размещаться только локальные объекты, объявленные в теле функции. Такие объекты существуют временно на этапе активности функции.

Имеется две разновидности динамической памяти:

– автоматическая память (атрибут auto) - объекты располагаются в стековой памяти;

– регистровая память (атрибут register) - объекты располагаются в регистрах общего назначения, а при нехватке регистров - в стековой памяти (размер объекта не должен превышать разрядности регистра).

По умолчанию, локальные объекты, объявленные в теле функции, имеют атрибут класса памяти auto. Регистровая память позволяет увеличить быстродействие программы, но к размещаемым в ней объектам неприменима операция адресации & (см. 3.11).

```
void swap(int *x, int *y) {
 register int t;
 t=*x, *x=*y, *y=t;
}
```

Понятие области действия приходится учитывать при использовании многофайловых программных комплексов [1]:

```
/********************** Файл  f1.c ********************/

static void f0(); /* f0 доступна только в файле f1.c */
extern int n; /* Переменная n определена в файле f2.c */
char x[]="n"; /* Глобальная переменная без атрибута static */

void f2() { /* Функция f2 доступна из всех файлов */
 printf("\nЗначение %s=%d",x,n);
 f0(); /* Вызов f0 из текущего файла f1.c */
}


void f0() {
 printf("\n Функция f0 из файла f1.c");
}
```

```
/************************ Файл  f2.c ********************/

 static void f0() { /* f0 доступна только в файле f2.c */
  printf("\n Функция f0 из файла f2.c");
 }

 int n=10; /* Глобальная переменная без атрибута static */

 extern char x[];  /* Массив x определен в файле f1.c */

 void main() {
  f2(); /* Вызов f2 из файла f1.c */
  printf("\n Справедливо ли (%s==%d) ?",x,n);
  f0(); /* Вызов f0 из текущего файла f2.c */
 }
```

1.5.3 Инициализации объектов программ

Любые объекты, кроме масивов, структур и объединений, имеющих атрибут auto, а также формальных параметров, при определении могут получать начальные значения - константные выражения. Признак инициализации - символ '=' в поле оператора описания объекта.

Начальные значения записываются по следующим правилам:

а) основные типы данных

```
int x=0;
float y=3.14;
char z='\0';
int *h=&x;
char *u=(char *)&y;
int a=sizeof(float);
int print(void), (*prnt)(void)=print;
```

б) массивы

```
int t[10]={1,2,3,4,5};
int e[]={1,2,3,4,5};

char x[]={'Y','E','S','\0'}; /*  Эквивалентные    */
char x[]="YES";              /*  описания         */
char *x="YES";               /*  массива символов */
```

(список значений - в фигурных скобках, недостающие значения заменяются нулями, если размер массива опущен, то он определяется фактическим числом начальных значений, массив символов может быть инициализирован строковой константой);

в) структуры

```
struct part {
 int code;
 char *name;
};

struct part x={121,"Блок питания"};
struct part y[]= {
  {10,"Болт"},
  {20,"Гайка"},
  {30,"Шайба"}
};

struct part z[10]= {
  {11},
  {12},
  {13}
};
```

(список значений каждой структурной переменной может заключаться в фигурные скобки, значения элементам структуры присваиваются в порядке размещения элементов в определении структурного типа, список значений может быть неполным, тогда оставшиеся поля заполняются нулями).

Перепишем ранее рассмотренный пример простейшего калькулятора [1]:

```
#include <stdio.h>
#include <math.h>
#include <conio.h>
#include <ctype.h>

char f[]="\n %s(%lf)=%lf";

struct {
 char *fn;
 double (*fa)();
} fd[]={{"Sin",sin},
        {"Cos",cos},
        {"Exp",exp},
        {"Log",log},
        {"sQrt",sqrt},
        {"Tan",tan}
       };

void main() {
 double x;
 int n=sizeof(fd)/sizeof(*fd);
 int in[256], i,j,k;

 for (i=0; i<256; in[i++]=n);
 for (i=0; i<n; i++)
  for (j=0; (k=fd[i].fn[j])!=0; j++)
   if (!islower(k)) {
    in[k]=in[tolower(k)]=i;
```

```
    break;
   }

 while (sound(1000),
        printf("\n x-? "),
        nosound(),
        scanf("%lf",&x)) {

  for (;;) {
   printf("\n x=%lf, f()-?",x);
   switch(k=getch()) {
    case 27 : goto cont;

    default : if ((i=in[k])<n) printf(f,fd[i].fn,x,(*(fd[i].fa))(x));
              else {
               sound(100);
               printf("\n Набор функций:");
               for (i=0; i<n; i++)
                printf("\n %s",fd[i].fn);
               nosound();
              }
   }
  }
  cont:;
 }
}
```

1.5.4 Управляемая память

Любой именованный объект программы размещается в статически либо автоматически распределяемой памяти. Статический объект размещается во время запуска программы и существует в течение всего времени ее выполнения. Автоматический объект размещается каждый раз при входе в его блок и существует только до момента выхода из блока. Часто возникает потребность управляемого размещения объектов в памяти в соответствии с алгоритмом решения задачи без привязки к блокам программы. В языках C и C++ такие объекты могут адресоваться только косвенно по значению указателя. Указатель может иметь при этом имя, но адресуемый им объект является безымянным. Область памяти для размещения объектов может быть получена запросом к операционной системе на выделение блока требуемого размера либо назначена на место размещения некоторого известного объекта достаточного размера. Управление размещением объектов осуществляется операциями захвата и освобождения памяти. В языке C для этих целей приходится пользоваться библиотечными функциями. Например, в файле alloc.h декларированы функции:

– void *malloc(unsigned nbytes) - возврат указателя на выделенную область размером nbytes (NULL при недостатке памяти или nbytes==0);

– void free(void *block_pointer) - освобождение захваченной памяти по заданному адресу.

Операции захвата и освобождения памяти в стиле языка С имеют вид:

```
указатель_на_объект=malloc(sizeof(атрибуты_типа_объекта));
free(указатель_на_объект);
```

Пример программы с работой в управляемой памяти:

```
#include <stdio.h>
#include <stdlib.h>
#include <mem.h>

/* Процедура сортировки обменным методом */

     int excngs(void *base,
                int number,
                int width,
                int fcp(const void *x,const void *y)) {
      char *x,*y,*t;
      int i,j,n;
      t=(char *)malloc(width);
      if (t) {
       n=number-1;
       do {
        for (i=j=0; i<n; i++) {
         x=(char *)base+i*width;
         y=x+width;
         if (fcp(x,y)>0) {
          memcpy(t,x,width);
          memcpy(x,y,width);
          memcpy(y,t,width);
          j=n;
         }
        }
       } while (j);
       free(t);
      }
      return (t!=NULL);
```

```
     }

typedef struct {
 int attr, app, group, obj;
} TEST;

int fcp(const void *x,const void *y) {
 TEST *X=(TEST *)x,
      *Y=(TEST *)y;
 return(X->attr-Y->attr);
}

/* Исходный массив структур */

TEST x[]={
 { -1, 1, 1, 1},
 {-10, 2, 1, 1},
 { -2, 3, 1, 1},
 { -3, 4, 1, 1},
 { -1, 5, 1, 1}
};

/* Печать массива структур */

void print(char *title) {
 int n=sizeof(x)/sizeof(*x);
 printf("\n %s",title);
 for (int i=0; i<n; i++)
  printf("\n %3d) %6d %6d %6d %6d",
         i,x[i].attr,x[i].app,x[i].group,x[i].obj);
 printf("\n");
}

/* Тестовая программа */

void main () {
 print("Исходный массив:");
 if (excngs(x,sizeof(x)/sizeof(*x),sizeof(*x),fcp))
```

```
  print("Массив после сортировки:");
}
```

Безопасное использование управляемой памяти требует обязательной проверки успешности выделения памяти [1].

1.6 ПРЕПРОЦЕССОР ЯЗЫКА С

1.6.1 Возможности препроцессора и его вызов

Препроцессор - программа предварительной обработки исходного текста программы перед этапом компиляции. Способ включения препроцессора в систему программирования определяется стилем ее реализации. Например, в системе программирования Turbo-C препроцессор совмещен с компиляторами TC и TCC. Кроме этого, имеется и автономный пакетный препроцессор CPP. Чаще всего препроцессор автоматически вызывается на этапе компиляции, если в исходном тексте обнаружен хотя бы один препро цессорный оператор. Признаком препроцессорного оператора в языке С является символ '#'(обычно в начале строки). Такой оператор заканчивается символом перевода на новую строку '\n'. При необходимости продолжения оператора в следующей строке текущую строку должен завершать символ '\'.

Возможности препроцессора языка С:

– лексемное замещение идентификаторов;
– макрозамещение;
– включение файлов исходного текста;
– условная компиляция;
– изменение нумерации строк и текущего имени файла [1].

1.6.2 Операторы лексемного замещения идентификаторов

Оператор определения значения идентификатора:

```
#define идентификатор строка
```

В результате каждое вхождение в исходный текст элемента "идентификатор" заменяется на значение элемента "строка":

```
#define L_bufs 2048
#define binary int
#define WAIT ffluch(stdin); getch()
#define BEEP sound(800);\
```

```
delay(100);nosound()
```

Лексемное замещение весьма удобно для сокращения записи повторяющихся фрагментов текста и определения символических констант:

```
#define YES 1
#define NO  2
#define ESC 27
#define Enter 30
```

Примеры использования:

```
if (x==ESC) break;
BEEP;
return(YES);
```

Оператор отмены определения идентификатора

```
#undef идентификатор
```

Далее по исходному тексту можно назначить новое значение такого идентификатора. Функцию оператора препроцессора #define по отношению к именованию типов объектов может выполнять оператор переопределения типа компилятора языка C typedef (п. 1.2.7) [1].

1.6.3 Макрозамещение

Макрозамещение - обобщение лексемного замещения посредством параметризации строки оператора #define в виде:

```
#define идентификатор(параметр1,... ) строка
```

(между элементом "идентификатор" и символом '(' пробелы не допускаются).

Такой вариант оператора #define иногда называют макроопределением (макросом). Элемент "строка" обычно содержит параметры, которые будут заменены препроцессором фактическими аргументами так называемой макрокоманды, записываемой в формате

```
идентификатор(аргумент1,... )
```

Пример макроопределения и макрокоманд:

```
#define P(X) printf("\n%s",X)

char *x;

/* Использование макроопределения P(X) */

P(x);
P(" НАЧАЛО ОПТИМИЗАЦИИ");
```

```
/* Эквивалентные операторы */

   printf("\n%s",x);
   printf("\n%s"," НАЧАЛО ОПТИМИЗАЦИИ");
```

Идентификаторы параметров в строке макроопределений сложных выражений рекомендуется заключать в круглые скобки:

```
#define MAX(A,B) ((A)>(B)? (A):(B))
#define UP(x) ((x)-'a'+'A')
#define LOW(x) ((x)-'A'+'a')
```

Потребность в круглых скобках возникает при опасности искажения смысла вложенных выражений из-за действия правил приоритета операций. Пример искажения смысла операций:

```
#define BP(X) X*X

int x,y,z;

x=BP(y+z); <===> x=y+z*y+z; <===> x=y+(z*y)+z;
```

Очевидно, что ошибки будут и при следующих вариантах:

```
#define BP(X) (X*X)
#define BP(X) (X)*(X)
```

Безопасный вариант:

```
#define BP(X) ((X)*(X))
```

Иногда источником ошибок может быть символ ';' в конце строки макроопределения:

```
#define BP(X) ((X)*(X));

int x,y,z;

x=BP(z)-BP(y); <===> y=((z)*(z));-((y)*(y));
```

Макроопределение отменяется оператором #undef.

Идентификаторы макроопределений обычно составляют из прописных букв латинского алфавита. Это позволяет отличать макрокоманды от вызова функций. Например, в Turbo-C построение "далекого" указателя может быть выполнено макрокомандой MK_FP:

```
#define MK_FP(seg,ofs) \
 ((void far *)(((unsigned long)(seg)<<16)|(unsigned)(ofs)))
```

Макрокоманда внешне синтаксически эквивалентна операции вызова функции, но смысл их существенно различен. Функция в программе имеется в одном экземпляре, но на ее вызов тратится время для подготовки параметров и передачи управления. Каждая макрокоманда замещается соответствующей частью макроопределения, но потерь на передачу управления нет. Дополнительное отличие: параметры функции контролируются на соответствие типа и могут автоматически преобразовываться в заданный прототипом тип [1].

1.6.4 Оператор включения файлов исходного текста

Имеются два варианта запроса включения в текущий файл содер жимого другого файла. Оператор

```
#include <имя_файла>
```

вводит содержимое файла из стандартного каталога (обычно принято именовать его \include\), а оператор

```
#include "имя_файла"
```

организует последовательный поиск в текущем, системном и стандарт ном каталогах.

Примеры (Turbo-C MS-DOS):

```
#include <stdio.h>    /* Стандартные средства ввода-вывода */
#include <alloc.h>    /* Средства распределения памяти */
#include <dos.h>      /* Обращения к функциям ОС */
#include <conio.h>    /* Консольный ввод-вывод */

#include "a:\prs\head.h"  /* Включение файла пользователя */
```

В стандартных каталогах системы программирования помещены операторы описания внутренних функций языка С, системных переменных среды исполнения, структур данных файловой системы, строк диагностических сообщений, определения констант и т.п.

Рекомендуется описания системных объектов включать из стандартных каталогов и размещать их в начале файла исходного текста программы. Системные объекты в результате получают атрибут области действия "глобальный", что устранит неоднозначность их описания. Включаемый оператором #include исходный текст может содержать любые операторы препроцессора [1].

1.6.5 Условная компиляция

Условность компиляции понимается по отношению к включению фрагментов текста в рабочий вариант программы. Операторы условной компиляции и реализуемые правила включения исходного текста:

а) условное включение

```
#if<предикат_условия>
 ТЕКСТ
#endif
```

(если условие истинно, то ТЕКСТ обрабатывается компилятором);

б) альтернативное включение

```
#if<предикат_условия>
 ТЕКСТ_1
#else
 ТЕКСТ_2
#endif
```

(если условие истинно, то компилятором обрабатывается ТЕКСТ_1, иначе - ТЕКСТ_2).

Виды предикатов условий:

– константное_выражение - истина, если его значение не равно нулю;

– def идентификатор - истина, если идентификатор был определен ранее оператором #define;

– ndef идентификатор - истина, если идентификатор не был определен оператором #define.

Константное_выражение отделяется от ключевого слова if разделителем, а def и ndef - нет.

Пример:

```
#ifdef DEBUG
 print_state();
#endif
```

Элементы исходного текста "ТЕКСТ_1" или "ТЕКСТ_2" могут содержать любые операторы препроцессора.

Примеры:

```
#ifndef EOF
#define EOF -1
#endif

#if UNIT==CON
#include "conproc.c"
```

```
#else
#include "outproc.c"
#endif

/* Блокировка повторного включения заголовочного файла */

#ifndef _DESCRIPTOR
#define _DESCRIPTOR
 /*
  Текст
 */
#endif
```

Предикаты условий могут использовать системно определенные макросы, например:

__DATE__ - дата,

__TIME__ - время компиляции;

__FILE__ - имя компилируемого файла;

__LINE__ - целочисленное значение номера текущей строки.

Пример использования системных макроопределений:

```
#include <stdio.h>
void main() {
 printf("\nФайл %s",__FILE__);
}
```

Кроме заданных операторами #define и системоопределенных макросов доступными оказываются и идентификаторы, задаваемые в командной строке вызова компилятора и(или) опциями установки интегрированной среды.

В качестве предиката условия в современных препроцессорах может выступать "функция" препроцессора defined():

```
#if defined(DOS_TARGET)
 puts(msg);
#else
 MessageBox(NULL,msg,"MSG",MB_OK | MB_TASKMODAL);
#endi
```

Использование defined() позволяет сокращать текст записи сложных условий:

```
#if defined(DOS_TARGET) && !defined(NO_DEBUG)
 puts(msg);
#endif
```

Предикат условия может включать логические операции языка С и операции отношений (==, !=, >, >=, <, <=).

Последнее расширение операторов условной компиляции - оператор #elif:

```
#if выражение_1        /* Включение, если выражение_1 - истина */

#elif выражение_2      /* Включение, если выражение_2 - истина */

#else                  /* Включение, если все выражения ложны */

#endif
```

В последней синтаксической конструкции обязательны лишь первый и последний операторы. Типичное применение #elif - альтернативный выбор фрагмента исходного текста [1]:

```
void sort(void *x,int n) {
 #if 0
  /* Новая версия программы не отлажена */

 #elif MODEL==__HUGE__
  /* Работа с "дальними указателями" */
 #else
  /* Использование библиотечной функции */
  qsorts(x,n);
 #endif
}
```

1.6.6 Изменение нумерации строк и имени файла

По умолчанию диагностические сообщения компилятора привязываются к номеру строки и имени файла исходного текста.

Оператор

```
#line номер_строки идентификатор_файла
```

позволяет с целью более приметной привязки к фрагментам текста изменить номер текущей строки __LINE__ и имя файла __FILE__ на новые значения ("идентификатор_файла" можно опустить) [1].

1.6.7 Расширенные возможности современных процессоров

Современные версии препроцессоров поддерживают следующие возможности:

– вывод диагностических сообщений;
– преобразование аргументов макроопределений в строку;
– конкатенацию(склейку) лексем.

Синтаксис оператора **вывода диагностических сообщений:**

```
#error сообщение_об_ошибке
```

(сообщение об ошибке здесь может включать идентификаторы макроопределений).

```
#if !defined(__HUGE__)
 #error Файл __FILE__: компиляция только в режиме HUGE
#endif
```

Рассмотрим возможность **преобразования аргументов макроопределений в строку.** В операторе макроопределения

```
#define идентификатор(парам_1,...) строка
```

именам параметров в "строке" может предшествовать символ '#', что предписывает преобразование аргумента в строку. Результат преобразования объединяется со смежными строками, если он отделен только пробелом:

```
#define DEBUG_OUT(intvar) \
 printf(#intvar "%d\n", (int)(intvar))

void main() {
 int alpha=1, betta=2;
 DEBUG_OUT(alpha);

 DEBUG_OUT(betta);
}
```

Результаты работы программы:

```
alpha=1 betta=2
```

Конкатенация, или склейка, лексем программируется следующим образом: оператор X##Y объединяет лексемы X и Y, причем результат снова обрабатывается препроцессором.

Пример:

```
#define DEF_Var(n) int __var_##n
#define USE_Var(n) __var_##n

DEF_Var(100);
DEF_Var(200);

USE_Var(100)=1;
USE_Var(200)=USE_Var(100)++;
```

Приведенный фрагмент программы на вход компилятора поступит в виде

```
int __var_100;
int __var_200;

__var_100=1;
__var_200++;
```

Очевидно, что рассмотренные здесь расширенные возможности препроцессора облегчают параметризацию исходного текста программы [1].

1.7 ЗАПУСК И ЗАВЕРШЕНИЕ ПРОГРАММ

1.7.1 Головная функция программ на языке С

Головной функцией любой программы на языке С является функция main, которая может получать аргументы из командной строки вызова программы и среды оболочки. Интерпретация командной строки вызова программы

```
имя арг_1 арг_1 ... арг_N
```

средствами языка С выглядит так:

```
char *argv[argc]={"имя","арг_1",..."арг_N"};
```

(здесь argc=N+1). Интерпретация переменных среды оболочки

```
char *envp[]={
 "имя_1=значение_1",
 "имя_2=значение_2",
 ...
 "имя_M=значение_M",
 NULL  /* Признак конца списка */
};
```

Набор переменных среды оболочки в MS-DOS может меняться и контролироваться оператором SET: а) добавление или коррекция переменной

```
set имя=значение
```

б) исключение переменной из среды

```
set имя=
```

в) вывод на экран переменных среды

```
set
```

Доступ к аргументам командной строки вызова программы и переменным среды оболочки возможен в случае использования списка формальных параметров в головной функции main. Пример процедуры main с полным списком параметров:

```
#include <stdio.h>
void main(int argc, char *argv[], char *envp[]) {
 int i;
 char **p;

 /* Печать параметров командной строки */
```

```
 for (i=0; i<argc; i++)
  printf("\nАргумент %d): %s",i,argv[i]);

 /* Печать переменных среды оболочки */

  printf("\n\nПеременные среды оболочки:");
  for (p=envp; *p; p++)
   printf("\n   %s",*p);
}
```

Результаты работы программы:

```
        Аргумент 0): D:\RMPL\SP\P1.EXE
        Аргумент 1): x1
        Аргумент 2): x2
    Переменные среды оболочки:
       winbootdir=C:\WINDOWS
       COMSPEC=C:\COMMAND.COM
       PROMPT=$p$g

PATH=C:\WINDOWS;C:\WINDOWS\COMMAND;D:\LEXNEW;E:\CBUILDER\
BIN
       TEMP=C:\TMP
       INCLUDE=e:\msdev\include
       LIB=e:\msdev\lib
       windir=C:\WINDOWS
       CMDLINE=p1.exe  x1 x2
```

Результаты выполнения команды set:

```
    winbootdir=C:\WINDOWS
    COMSPEC=C:\COMMAND.COM
    PROMPT=$p$g
    PATH=C:\WINDOWS;C:\WINDOWS\COMMAND;D:\LEXNEW;E:\CBUIL
DER\BIN
    TEMP=C:\TMP
    INCLUDE=e:\msdev\include
    LIB=e:\msdev\lib
    windir=C:\WINDOWS
```

Функция main может вызываться рекурсивно из любой функции:

```
#include <stdio.h>

int n=5;

void main() {
  printf("\n %*i",n,n--);
 if (n) main();
}
```

Результаты работы программы:

```
    5
   4
  3
 2
1
```

Следует отметить, что в языке C++ рекурсивный вызов функции main не всегда допускается [1].

1.7.2 Порождение и идентификация задач

В однопользовательской операционной системе MS-DOS процесс запуска задач (программ) активизируется интерпретатором команд COMMAND.COM с использованием по умолчанию консоли CON. Имеется возможность назначения вместо консоли других стандартных устройств COM1,COM2,... AUX:

1) оператором SHELL в файле CONFIG.SYS:

```
SHELL=[[dos-drive:]dos-path]COMMAND.COMM [device] ...
```

2) при запуске нового экземпляра COMMAND.COM;

```
[[dos-drive:]dos-path]COMMAND.COMM [device] ...
```

3) оператором CTTY:

```
CTTY device
```

Здесь device - текущее устройство ввода-вывода команд и сообщений пользователя (терминал). COMMAND.COM запускает корневую задачу иерархии процессов пользователя, интерпретируя входные строки

```
[[dos-drive:]dos-path]exec_file p1 p2 ... [>stdout] [<stdin]
```

как запрос на запуск исполнимого файла (программы или пакетного командного файла) exec_file с параметрами p1,p2,... и возможным перенаправлением

стандартных потоков ввода-вывода. Любая задача может породить другие задачи-потомки. Примеры функций запуска задач-потомков:

```
int spawnv(int mode, // Режим запуска
           char *path, // Путь к загрузочному модулю
           char *argv[]); // Аргументы командной строки

int spawnve(int mode, // Режим запуска
            char *path, // Путь к загрузочному модулю
            char *argv[], // Аргументы командной строки
            char *envp[]); // Переменные среды окружения
```

Виды режимов запуска (Turbo-C, Borland C++ 3.1):

P_WAIT - ожидание момента завершения задачи-потомка;

P_NOWAIT - не реализованное в MS-DOS параллельное выполнение задач (применение приводит к ошибке);

P_OVERLAY - оверлейная загрузка задачи - потомка на место родителя (трансформация задач).

При успешном запуске после исполнения задачи-потомка возвращается значение кода возврата, в противном случае - значение -1 и код ошибки в глобальной переменной errno:

E2BIG - длинный список аргументов;

EINVAL - ошибочный аргумент;

ENOENT - файл не найден;

ENOEXEC - ошибка формата файла;

ENOMEM - нехватка памяти.

Задача-потомок в MS-DOS пользуется стандартными файлами stdin, stdout, stderr задачи родителя (их можно переопределить).

Пример программы:

```
#include <process.h>
#include <stdio.h>
#include <errno.h>

int main(void) {
 if (freopen("TEMP.OUT","w",stdout)) {
  if (spawnl(P_WAIT,"work.exe",NULL)<0) {
   printf("\n Ошибка %d",errno);
   exit(errno);
  }
 }
 return 0;
}
```

Приведенная программа моделирует ввод команды

```
work > temp.out
```

В MS-DOS доступны следующие виды внешнего вмешательства в процесс исполнения активной задачи:

Ctrl+Break - снятие задачи;

Pause - приостановка.

Межзадачное взаимодействие, организуемое средствами MS-DOS, сводится лишь к образованию канала обмена данными через временный файл. Например, команда

```
dir|sort|more
```

приводит к последовательному выполнению системных программ dir, sort, more и выдачи результата на устройство CON. Пусть файл ds.exe получен в результате трансляции программы

```
#include <stdio.h>

void main() {
 int i;
 while ((i=getch())!=EOF)
  putchar(' '), putchar(i);
}
```

Команда

```
dir|ds|more
```

позволит просмотреть результат работы программы dir в виде разреженного текста.

Функции семейства spawn в многозадачных операционных системах позволяют организовать динамическую параллельную структуру программы. Для образования динамической последовательной структуры можно воспользоваться функциями семейства exec:

```
int execl(char _FAR *__path, char _FAR *__arg0, ...);
int execle(char _FAR *__path, char _FAR *__arg0, ...);
int execlp(char _FAR *__path, char _FAR *__arg0, ...);

int execlpe(char _FAR *__path, char _FAR *__arg0, ...);
int execv(char _FAR *__path, char _FAR *__argv[]);
int execve(char _FAR *__path, char _FAR *__argv[],
           char _FAR *_FAR *__env);
int execvp(char _FAR *__path, char _FAR *__argv[]);
int execvpe(char _FAR *__path, char _FAR *__argv[],
            char _FAR *_FAR *__env);
```

Такие функции возвращают код завершения задачи-потомка. Можно выполнить запуск задачи-потомка через вызов интерпретатора командных строк, используя функцию

```
int system(char *command);
```

Здесь строка command может содержать любую команду и/или директиву, обрабатываемую интерпретатором командных строк:

```
system("dir>test.txt");
```

(в MS-DOS команда dir выполняется непосредственно интерпретатором командных строк) [1].

1.7.3 Завершение программ

Любая функция в языке С обычно завершается после выполнения последнего оператора в теле функции либо оператора return. Возврат из любой точки программы можно выполнить посредством вызова специальных библиотечных функций. Например, в файле stdlib.h определены функции:

– void exit(int status) - вывод содержимого буферов, закрытие всех файлов и возврат в порождающий процесс кода завершения status;

– void abort(void) - возврат в порождающий процесс с кодом завершения 3 [1].

1.7.4 Идентификация задач и виды межзадачных взаимодействий

В многозадачных средах каждая задача получает уникальный системный идентификатор (task ident). Например, в операционной системе QNX функции на языке С могут использовать глобальные переменные

```
unsigned int My_tid, /* Идентификатор текущей задачи */
             Dads_tid, /* Идентификатор задачи - родителя */
             My_nid; /* Номер узла вычислительной сети */
```

(QNX - сетевая многозадачная многопользовательская операционная система, быстрореактивная версия операционной системы семейства UNIX) [1]. Во время исполнения задача-родитель информирована об идентификаторах потомков результатом функций порождения. Иногда требуется получить идентификатор некоторой задачи, не связанной с текущей процессом порождения. Например, это может потребоваться для обмена информацией между задачами. Для удобства установления идентификаторов задач QNX предоставляет возможность подсоединения к задаче системного имени в виде строки символов. Набор функций для работы с системными именами:

```
unsigned name_attach(char *name, unsigned node);
unsigned name_detach(char *name, unsigned tid);
unsigned name_locate(char *name, unsigned node, unsigned size);
```

Если в качестве номера узла node указать нуль, то имя задачи будет глобальным, т.е. известным во всех узлах сети. В других случаях имя name локально на конкретном узле node. QNX поддерживает три механизма межзадачных связей:

– сообщения;
– порты;
– исключения [1].

1.8 СИСТЕМНО ЗАВИСИМЫЕ КОНСТРУКЦИИ ЯЗЫКА С

1.8.1 Системно зависимые расширения языка С

Наибольшая зависимость системы программирования на языке С от особенностей вычислительной среды проявляется в библиотеках стандартных функций. Система программирования языка С отражает особенности конкретной вычислительной среды посредством:

– дополнительных операторов;
– атрибутов типа (данных и функций);
– системно-определенных объектов (константы, переменные, функции);
– макроопределений.

Далее рассмотрим примеры системно зависимых расширений применительно к системе программирования С++ [1].

1.8.2 Понятие псевдо-регистров

Непосредственный доступ к регистрам процессора можно получить, используя зарезервированные идентификаторы:

```
_AX    _AL    _AH    _SI    _ES
_BX    _BL    _BH    _DI    _SS
_CX    _CL    _CH    _BP    _CS
_DX    _DL    _DH    _SP    _DS
```

(набор идентификаторов в Borland C++ 3.1 дополнен обозначением регистра флагов _FLAGS, а в Borland C++ 4.0 - регистрами _EAX, _EBX, _ECX, _EDX,_ESI, _EDI, _ESP, _EBP и т.п.). Представленные идентификаторы без символа подчеркивания соответствуют стандартным мнемоническим обозначениям регистров микропроцессора. Псевдо-регистры используют, например, для обращения к функциям BIOS или MS-DOS посредством программного прерывания.

Пример обращения к функции BIOS:

```
/* Установка формы курсора в текстовом режиме работы дисплея */

  #include <dos.h>

  void cursor(int top_line, int bottom_line) {
   _CH=top_line;
   _CL=bottom_line;
   _AH=0x01;
   geninterrupt(0x10);
  }
```

При работе с псевдо-регистрами следует учитывать наличие побочных эффектов:

```
  #include <dos.h>

  char *msg="Проба пера...";

  void main() {
   _AH=0x09;
   _DX=FP_OFF(msg);
   _DS=FP_SEG(msg);
   geninterrupt(0x21);
  }
```

Здесь присваивание регистров _DX и _DS проводится с временным использованием регистра _AX, приводящим к разрушению содержимого регистра _AH. В результате требуемая функция MS-DOS наверняка не будет указана [1].

1.8.3 Функции - обработчики прерываний

Функции обработки прерываний должны удовлетворять условиям, определяемыми аппаратурой. В Turbo-C MS-DOS такие функции должны описываться с атрибутом interrupt:

```
  void interrupt int_06h(void);
  void interrupt (*interp)(void);
```

Функция с атрибутом interrupt сохраняет при вызове все регистры процессора и завершается машинной командой IRET (возврат из обработчика прерываний). При компиляции таких функций необходимо отключать режимы диагностики о переполнении стека и использования регистровых переменных. Естественным здесь является отсутствие входных параметров и возвращаемых значений. Допускается использование глобальных объектов. Операция вызова рассматриваемых функций возможна: компилятор планирует программную имитацию аппаратных действий, связанных с сохранением и восстановлением

регистра флагов. Механизм обработки прерываний строится на основе вектора прерываний, поэтому для любого типа прерывания необходимо задать соответствие "номер прерывания - адрес процедуры обработки прерываний".

Пример программы обработки прерываний от таймера [1]:

```
#include <stdio.h>
#include <dos.h>

long Tick_Counter=100;

void interrupt (*old_1ch)(void);

void interrupt new_1ch(void) {
 Tick_Counter--;
 old_1ch(); /* Разрешенная операция вызова */
}

void main() {
 old_1ch=getvect(0x1c);
 setvect(0x1c,new_1ch);
 while (Tick_Counter>0) {
  /*****************************
   Произвольный набор операторов
  *****************************/
 }
 setvect(0x1c,old_1ch);
 printf("\nФИНИШ...");
 exit(0);
}
```

1.8.4 Дополнительные атрибуты указателей

Понятие указателя в языке С отражает адрес объекта. Однако в реальных вычислительных средах может отсутствовать взаимно однозначные соответствие математических и физических адресов памяти. Значения указателей формируются с учетом особенностей организации памяти. В шестнадцатирязрядных микропроцессорных системах, использующих относительную адресацию с базированием, физический адрес FA определяется следующим образом:

```
FA=Segm*16+Offs,
```

где Segm - содержимое сегментного регистра, Offs - смещение относительно начала сегмента. Для внешнего представления значения FA принято использовать запись Segm:Offs. Таким образом, значение адреса A000:0000 соответствует 640 K. В большинстве случаев организация памяти программы учитывается автоматически, но учет конкретных особенностей задачи может

потребовать назначения свойств указателей дополнительными атрибутами. Например, указатели объектов в программах, создаваемых в среде Turbo-C, имеют один из атрибутов:

– near - адрес в текущем сегменте (смещение, 2 байта);

– far - адрес в любом сегменте (сегмент:смещение, 4 байта), но размер объекта не более 64 Кбайт;

– huge - адрес в любом сегменте (сегмент:смещение, 4 байта), размер объекта любой, но выполняется нормализация значения адреса для корректного выполнения операций сравнения, инкремента и декремента.

Нормализация указателя с атрибутом huge заключается в представлении значения адреса единственным образом путем назначения минимально возможного "смещения" и максимально возможного "сегмента". По умолчанию указателю приписывается атрибут в зависимости от указанного при компиляции вида организации памяти программы.

Характеристика моделей памяти в Turbo-C

Модель памяти	Тип указателя по умолчанию		Количество сегментов размером 64 Кбайта			
	Данные	Функции	Данные	Функции	Всего	Один объект
TINY	near	near	1	1	1	1
SMALL	near	near	1	1	2	1
MEDIUM	near	far	1	>1	>1	1
COMPACT	far	near	>1	1	>1	1
LARGE	far	far	>1	>1	>1	1
HUGE	far	far	>1	>1	>1	>1

Доступ к значениям "дальних" указателей реализуется посредством макрокоманд:

– void far *MK_FP(segm, offset) - образовать "дальний" указатель по заданным значениям сегмента segm и смещения offset;

– unsigned FP_SEG(far_pointer),

– unsigned FP_OFF(far_pointer) - получить значения сегмента и смещения заданного "дальнего" указателя far_pointer.

Перечисленные макрокоманды позволяют работать с памятью на основе абсолютных значений адресов, например:

```
char far *x=MK_FP(0,0x466);
printf("\n Байт режима дисплея %#x",*x);
```

Другая возможность учета механизма сегментации - использование явной привязки "ближних" указателей к сегментным регистрам специальными модификаторами типа указателей _ds, _es, _cs, _ss. Например, оператор

```
char _ss *x;
```

объявляет указатель с атрибутом near для работы относительно базового регистра стека. Значение регистра стека можно загрузить, используя идентификатор псевдо-регистра _SS. Рассмотрим два варианта программы подсчета контрольной суммы двухбайтных слов области BIOS с целью идентификации ПЭВМ. Предположим, что интересующая нас область начинается с адреса F000:0000 и заканчивается в поле с адресом FFFF:000E.

```
    /* Вариант 1. Использование "дальнего" указателя */

.    #include <stdio.h>
     #include <dos.h>

     void main(void) {
      long s=0;
      unsigned huge *p=(unsigned huge *)MK_FP(0xF000,0);
      while (p)
       s+=*p++;
      printf("\nКонтрольная сумма: %#x",s);
     }
    /* Вариант 2. Использование "ближнего" указателя */

     #include <stdio.h>
     #include <dos.h>

     void main(void) {
      long s=0;
      unsigned _es *p=(unsigned _es *)0;
      _ES=0xF000;
      do s+=*p++;
      while (p)
      printf("\nКонтрольная сумма: %#x",s);
     }
```

Результаты работы двух вариантов программ:

```
Контрольная сумма: 0x4636
```

Второй вариант программы предпочтительнее по быстродействию, но требует повышенного внимания при отладке [1].

1.8.5 Модификаторы типа объектов

Если размер объекта превышает размер одного сегмента, то приходится использовать модификатор huge, например:

```
long huge work_area[40000];
```

Очевидно, что такой объект должен быть глобальным.

Объект может размещаться в одном сегменте, но если подобных объектов несколько, то модификатор far заставит компилятор разместить их в отдельных сегментах:

```
int far table_1[20000];
int far table_2[20000];
```

Другие примеры комбинирования атрибутов [1]:

```
int i; /* Переменная в сегменте по умолчанию */
int far j; /* Переменная в дальнем сегменте */
int *ip1=&i; /* Указатель в сегменте по умолчанию */
int far *ip2=&i; /* Дальний указатель в сегменте по умолчанию */
int * far ip3=&i; /* Стандартный указатель в дальнем сегменте */
int far * far ip4=&i; /* Дальний указатель в дальнем сегменте */
void far f1(char *); /* Адресуемая дальним указателем функция */
int near f2(float); /* Адресуемая ближним указателем функция */
```

1.8.6 Использование ассемблера

В тело любой функции на языке С можно встроить операторы языка ассемблера. В простейшем случае включение отдельного оператора оформляется так:

```
asm <код операции> <операнды> <';' или '\n'>
```

(код операции должен соответствовать мнемоническому обозначению кода команды на языке ассемблера, операндами могут быть константы, регистры или любые подходящие объекты программы на языке С). Несколько операторов ассемблера можно поместить в операторный блок:

```
asm {
 <операторы языка ассемблера>
}
```

Пример программы с включением операторов языка ассемблера:

```
#include <stdio.h>

int Number_1=1997,
    Number_2=1998;

void main() {
 printf("\n Number_1=%d, Number_2=%d ?!",Number_1,Number_2);
 asm {
  mov ax, Number_1
  mov dx, Number_2
  xor ax,dx
  xor dx,ax
  xor ax,dx
  mov Number_1, ax
  mov Number_2, dx
 }
 printf("\n Number_1=%d, Number_2=%d ??",Number_1,Number_2);
}
```

Вставка на ассемблере здесь реализует оператор языка С вида [1]

```
Number_1^=Number_2^=Number_1^=Number_2;
```

1.9 ВВОД-ВЫВОД ДАННЫХ

1.9.1 Организация ввода-вывода данных в С

Система программирования С и операционная система MS-DOS поддерживают понятие файла как поименованной последовательности однобайтных символов. Операции с такими последовательностями рассматриваются как однонаправленная последовательная передача потоков символов. Состояние процесса ввода-вывода определяется смещением текущей позиции от начала файла, которое увеличивается после выполнения операции передачи данных. Исходное состояние устанавливается обязательной операцией открытия файла. Операция закрытия файла автоматически выполняется при завершении программы в MS-DOS, но при необходимости может запрашиваться явно. Любые операции, связанные с организацией ввода-вывода, на языке С программируются посредством обращения к библиотечным функциям.

Файл в операционной системе MS-DOS представляется тремя взаимосвязанными понятиями:

идентификатор файла - символьная строка, имеющая согласно правил именования файлов в MS-DOS формат [диск:][путь\]имя_файла[.расширение_имени_файла] (здесь в квадратные скобки заключены необязательные элементы);

– номер обработчика (драйвера) ввода-вывода;
– указатель блока управления файлом.
– Набор средств ввода-вывода потоком в системе программирования С декларирован в файле stdio.h, где представлены:
– определение типа данных FILE (блока управления файлом);
– определения стандартных параметров и констант;
– макроопределения процедур ввода-вывода.

Среди определенных в файле stdio.h констант наибольший интерес представляют:

NULL - значение пустого указателя;

EOF - признак конца файла (целая константа, равная -1).

Структура блока управления файлом ввода-вывода потоком:

```
typedef struct {
 short          level;  /* Заполненность буфера потока */
 unsigned       flags;  /* Флажки состояния файла */
```

```
 char           fd;     /* Дескриптор файла(номер обработчика) */
 unsigned char  hold;   /* Возвращенный символ */
 short          bsize;  /* Размер буфера */
 unsigned char *buffer; /* Указатель буфера */
 unsigned char *curp;   /* Указатель текущей позиции */
 unsigned       istemp; /* Индикатор временного файла */
 short          token;  /* Рабочее поле процедур диагностики */
} FILE;
```

Практическое программирование операций ввода-вывода обычно не требует явного обращения к элементам структуры FILE. Такие элементы используются библиотечными функциями и макроопределениями, например:

```
#define ferror(f)   ((f)->flags & _F_ERR)
#define feof(f)     ((f)->flags & _F_EOF)
#define fileno(f)   ((f)->fd)
```

(интерпретация макроопределений приведена ниже). Любое использование файлов должно начинаться с открытия потока ввода-вывода и завершаться закрытием этого потока. Операция открытия устанавливает взаимно однозначное соответствие между идентификатором файла, его обработчиком и указателем блока управления вводом-выводом. В момент открытия проверяется корректность запроса на организацию потока ввода-вывода и проводится установка начальных значений переменных состояния потока. Операция закрытия выводит содержимое буферов потоков вывода и освобождает связанную с потоками системно распределенную память. После закрытия потока доступ к нему невозможен без операции открытия

Перейдем к обзору важнейших библиотечных функций системы Turbo-C, предназначенных для организации ввода-вывода потоком. Функция FILE *fopen(char *path, char *mode) открывает поток ввода-вывода по полной спецификации файла path, режиму обработки файла mode и возвращает указатель успешно открытого файла или NULL при неудаче. Допустимые символы управляющей строки mode:

'r' - открытие для чтения;

'w' - создание файла для вывода;

'a' - открытие для дозаписи в конце существующего файла или создания нового файла;

'+' - разрешение обновления файла;

'b' - открытие двоичного файла;

't' - открытие текстового файла.

Примеры управляющих строк:

"rb" - двоичный файл ввода;

"w+t" - текстовый файл для вывода и обновления;

"a" - расширяемый существующий или создаваемый новый файл.

Тип файла (текстовый или двоичный) назначается по умолчанию значением глобальной переменной _fmode.

Текстовый файл характеризуется тем, что выводимый символ перевода строки заменяется парой символов возврата каретки и переводастроки, а при вводе выполняется обратное преобразование. Операции ввода-вывода для двоичных файлов выполняются без подобного преобразования.

Функция FILE *tmpfile(void) позволяет создать временный файл (на периоде активности программы). Иногда полезным оказывается получение уникального имени s для временного рабочего файла посредством вызова функции char *tmpnam(char *s).

Система программирования Turbo-C предоставляет возможность использования стандартных потоков ввода-вывода, которые по умолчанию автоматически открываются при запуске программы и закрываются при ее завершении.

Стандартные файлы:

stdin - файл ввода;

stdout - файл вывода;

stderr - файл вывода сообщений об ошибках;

stdaux - файл дополнительных данных;

stdprn - файл печати.

По умолчанию файлы stdin, stdout и stderr назначаются на терминал.

Функция FILE *freopen(char *path, char *mode, FILE *stream) закрывает открытый поток ввода вывода stream и назначает ему новый файл path в режиме mode. При нормальном открытии нового файла возвращается значение stream, в противном случае - NULL.

Функция freopen используется, например, для переназначения потоков стандартных файлов ввода-вывода.

Функция int fclose(FILE *stream) закрывает поток stream и возвращает при нормальном завершении 0, а в противном случае - EOF.

Функция int fcloseall(void) закрывает все открытые потоки ввода-вывода, за исключением stdin и stdout. Закрытие всех открытых потоков выполняет и функция void exit(int status).

Макрос int ferror(FILE *stream) определяет тип ошибки в потоке stream и возвращает 0 при отсутствии ошибок.

Макрос int fileno(FILE *stream) определяет дескриптор файла (номер обработчика), связанного с потоком stream.

Функция int eof(int handle) и макрос int feof(stream) проверяют состояние конца файла, связанного с обработчиком handle или потоком stream, и возвращают значения:

1 - конец файла;

0 - конец файла не достигнут;

-1 - обнаружены ошибки.

Функция void clearerr (FILE *stream) обнуляет признаки ошибок в потоке stream.

Функция int fflush(FILE *stream) очищает буфер ввода-вывода потока stream: для вводного файла удаляются необработанные символы, а для выводного выполняются операции передачи символов из буфера на носитель данных. Функция int flushall(void) выполняет те же действия для всех открытых потоков ввода-вывода.

Функция long filelength(int handle) возвращает размер файла в байтах, связанного с обработчиком handle или -1 в случае ошибок.

Функция int rename(char *oldname, char *newname) позволяет заменить имя файла oldname на новое имя newname.

Функция int unlink(char *path) удаляет файл с полной спецификацией path.

Функция void rewind(FILE *stream) устанавливает указатель текущей позиции потока stream на начало файла.

Функция int fseek(FILE *stream, long offset, int whence) смещает указатель текущей позиции потока stream на величину offset относительно позиции, отсчитываемой в зависимости от значения параметра whence:

SEEK_SET - от начала файла;

SEEK_CUR - от текущей позиции;

SEEK_END - от конца файла.

Константы SEEK_SET=0, SEEK_CUR=1 и SEEK_END=2 определены в файле stdio.h, где находится и определение функции fseek.

Функция int fsetpos(FILE *stream, long *pos) присваивает указателю текущей позиции в потоке stream значение целого числа, адресуемого указателем pos, и возвращает 0 в случае успешного завершения.

Функция font color = "blue"long ftell(FILE *stream) возвращает значение указателя текущей позиции, а в случае ошибок - EOF.

Функция int fgetpos (FILE *stream, long *pos) присваивает значение указателя текущей позиции в потоке stream целому числу, адресуемому указателем pos, и возвращает 0 в случае успешного завершения.

Рассматриваемые далее функции позволяют управлять буферизацией процесса ввода-вывода. Возможно использование трех режимов буферизации обмена, запрашиваемых соответствующим значением параметра type функции setvbuf:

_IOFBF - полная буферизация - операция вывода на внешнее устройство выполняется после полного заполнения буфера, а операция ввода при пустом буфере включает попытку заполнения буфера с внешнего устройства;

_IOLBF - строчная буферизация - вывод содержимого полностью заполненного буфера на внешнее устройство прерывается после передачи символа перевода строки '\n', а ввод выполняется подобно режиму полной буферизации;

_IONBF - отсутствие буферизации - каждый оператор ввода-вывода реализует обмен непосредственно с внешним устройством.

Функция int setvbuf(FILE *stream,char *buf,int type,int size) устанавливает режим буферизации type для потока stream, причем для режимов полной или строчной буферизации размер буфера определяется параметром size, а адрес буфера - buf(допускается значение buf=NULL, тогда память будет захвачена с помощью функции malloc); при установке не буферизованного обмена параметры buf и size игнорируются.

Функция void setbuf(FILE *stream, char *buf) устанавливает режим полной буферизации потока stream с буфером стандартного размера BUFSIZ, адресуемого непустым указателем buf; в случае обращения с параметром buf=NULL устанавливается режим не буферизованного обмена.

Особенности использования функций setbuf и setvbuf:

– буфер потока не может размещаться в автоматической памяти;

– назначение нового буфера взамен автоматически создаваемого должно производиться после открытия потока.

По умолчанию, потоки stdin и stdout являются не буферизованными. При их переопределении функцией freopen они становятся полностью буферизованными. Изменить вид буферизации можно и функциями setbuf и setvbuf.

При программировании ввода-вывода иногда полезным оказывается предвосхищение исключительных ситуаций посредством анализа текущей обстановки.

Например, функция int access(char *path,int mode) выполняет проверку доступности файла, идентифицируемого символьной строкой path. Вид проверки задается параметром mode:

0 - существование файла;

1 - исполняемость файла;

2 - разрешение записи;

4 - разрешение чтения;

6 - разрешение записи и чтения.

Функция access возвращает нулевое значение в случае успешного результата проверки, в противном случае возвращается -1 и устанавливается значение глобальной переменной errno:

ENOENT - путь либо файл не найден;

EACCES - доступ запрещен.

Функция char *searchpath(char *filename) осуществляет поиск пути к файлу с именем filename сначала в текущем каталоге, а затем по всем каталогам, определенным переменной PATH среды окружения. Возвращаемое значение - указатель на строку с полным путем доступа к файлу или NULL при безуспешном поиске [1].

1.9.2 Бесформатный ввод-вывод

Рассматриваемые здесь функции ввода-вывода предназначены для обмена между полями данных в оперативной и внешней памяти. Понятие поля данных здесь относится к символу, слову, строке и блоку - непрерывной области, размер которой указан в байтах.

Функция int fgetc(FILE *stream) возвращает очередной прочитанный символ из потока stream, а в случае обнаружения конца файла или ошибок - значение EOF. Аналогичные действия выполняет и макрос int getc(FILE *stream).

Функция int fgetchar(void) реализует оператор вызова функции fgetc(stdin).

Функция int ungetc(int chr, FILE *stream) позволяет вернуть прочитанный символ chr в поток stream для последующего повторного считывания. Макрос int ungetch(int chr) возвращает символ в поток stdin.

Функция int getw(FILE *stream) возвращает очередное прочитанное слово из потока stream (без учета выравнивания), а в случае обнаружения конца файла или ошибок - значение EOF. Для выявления исключительной ситуации при получении значения EOF можно воспользоваться функциями feof или ferror.

Функция char *fgets(char *s, int n, FILE *stream) читает из потока stream символы в строку s. Чтение прекращается после передачи n-1 символа либо обнаружения символа новой строки '\n', который заменяется на символ конца строки '\0'.

Функция char *gets(char *s) читает из потока stdin символы в строку s до обнаружения символа '\n', который заменяется на символ '\0'.

Функция int fputc(int chr, FILE *stream) или макрос putc выводят символ с внутренним кодом chr в поток stream, а функция int fputchar(int chr) - в поток stdout.

Функция int putw(int w, FILE *stream) выводит слово w в поток stream.

Функция int fputs(char *s, FILE *stream) выводит строку символов s в поток stream без символа конца строки '\0', а функция int puts(char *s) - в поток stdout с заменой символа '\0' на символ новой строки '\n'. Возвращаемые значения - последний выведенный символ или EOF в случае ошибок.

Следующие функции выполняют операции над n одинаковыми элементами размером size байт блока, адресуемого указателем ptr и потокомstream:

int fread(void *ptr,int size, int n,FILE *stream) - чтение;

int fwrite(void *ptr, int size, int n,FILE *stream) - запись.

Возвращаемое значение - фактическое количество переданных элементов блока.

```
/* Пример использования функций бесформатного ввода данных */
#include <stdio.h>
#include <io.h>
#include <alloc.h>

void main() {
 FILE *inpfil;
 char *buffer;
 size_t n;
 if ((inpfil=fopen("b1.c","rb"))!=NULL) {
  n=(size_t)filelength(fileno(inpfil));
  if ((buffer=calloc(sizeof(*buffer),n))!=NULL) {
   fread(buffer,sizeof(*buffer),n,inpfil);
   printf("\n Файл в памяти");
  }
 }
}
```

Бесформатный ввод-вывод отличается высоким быстродействием и часто применяется для операций с рабочими файлами, когда представленные в файле данные не подлежат визуальному просмотру или обработке с помощью текстовых редакторов [1].

1.9.3 Форматный ввод-вывод

Средства форматного ввода-вывода предназначены для преобразования внешнего представления данных в текстовой форме во внутреннее кодовое преставление и наоборот в соответствии с заданным форматом преобразования. Функции форматного ввода-вывода используют три класса параметров - поток ввода-вывода, управляющую форматную строку и список идентификаторов объектов ввода-вывода.

Примеры декларации функций форматного вывода в Turbo-C:

```
int fprintf(FILE *stream, char *format, ...);
int printf(char *format, ...);
int cprintf(char *buffer, char *format, ...);
int sprintf(char *buffer, char *format, ...);
int vfprintf(FILE *stream, char *format, va_list arglist);
int vprintf(char *format, va_list arglist);
int vsprintf(char *buffer, char *format, va_list arglist).
```

Примеры декларации функций форматного ввода в Turbo-C:

```
int fscanf(FILE *stream, char *format, ...);
     int scanf(char *format, ...);
     int cscanf(char *buffer, char *format, ...);
     int sscanf(char *buffer, char *format, ...);
     int vfscanf(FILE *stream, char *format, va_list arglist);
     int vscanf(char *format, va_list arglist);
     int vsscanf(char *buffer, char *format, va_list arglist).
```

Идентификация аргументов функций форматного ввода-вывода:

stream - указатель потока;

format - указатель форматной строки;

buffer - указатель массива символов;

arglist - переменный список аргументов (способы использования переменных списков аргументов в системе Turbo-C изложены в [1,2]).

Идентификация функций форматного ввода-вывода построена на основе добавления префикса к имени функции ввода scanf или вывода printf:

отсутствие префикса - ввод из стандартного потока stdin и вывод в стандартный поток stdout;

префикс f - операции с потоком stream;

префикс c - ввод с клавиатуры и вывод на экран терминала;

префикс s - операции с массивом символов buffer.

дополнительный префикс v - использование переменного списка аргументов.

Функции форматного ввода-вывода потоком имеют переменное число аргументов. Количество и тип аргументов, определяющих передаваемые данные, должны соответствовать спецификации преобразования данных в форматной (управляющей) строке. При недостатке аргументов по отношению к количеству спецификаций данных в форматной строке результат работы функций ввода-вывода непредсказуем, а лишние аргументы игнорируются.

Форматная строка, определяющая способ преобразования данных, является стандартной в языке С символьной строкой. Рассмотрим вопросы программирования вывода данных потоком. Пример вывода данных:

```
...
int n,m;
 ...
printf("\n Значение n=%d, m=%d",n,m);
...
```

Здесь первый параметр - форматная строка, определенная символьной константой, а n и m - элементы множества выводимых данных.

Особенности интерпретации форматной строки функциями вывода printf:

– символы форматной строки разделяются на обычные и символы спецификации преобразования;

– обычные символы без преобразования копируются в поток вывода;

– спецификация преобразования в форматной строке включает начальный символ '%' и символы описания формата преобразования аргумента функции вывода.

Спецификация формата вывода данных: %[flags][width][.prec][mode]type (в квадратных скобках указаны необязательные элементы). Здесь:

flags - флаги, определяющие выравнивание результата в поле вывода, символы знака числа, десятичные точки, незначащие нули, восьмеричные и шестнадцатеричные префиксы;

width - ширина поля вывода - минимальное количество выводимых символов, дополняемых пробелами либо нулями;

prec - точность представления результата;

mode - модификатор размера поля преобразуемого аргумента;

type - тип преобразования.

Интерпретация обязательного элемента type:

d - десятичное целое число со знаком (альтернатива i);

i - десятичное целое число со знаком (альтернатива d);

o - восьмеричное целое число без знака;

u - десятичное целое число без знака;

x - шестнадцатеричное целое число без знака, представленное строчными символами;

X - шестнадцатеричное целое число без знака, представленное прописными символами;

f - десятичное число со знаком с фиксированной точкой;

e - десятичное число со знаком с плавающей точкой, представленное в виде [-]d.ddd e [+/-]ddd строчными символами;

g - наиболее короткий формат из e или f;

E - аналог формата e, представляемый прописными символами;

G - аналог формата g, представляемый прописными символами;

c - символ;

s - строка символов, ограниченная символом '\0' или [.prec];

% - символ '%';

p - значение указателя (формат вывода указателя типа near - YYYY, а указателя типа far или huge - XXXX:YYYY, где X и Y - шестнадцатеричные цифры);

n - запоминание текущего количества выведенных символов в поле целого типа, адресуемым очередным аргументом (например, оператор printf("%d%n %d",i,&j,k) помещает в переменную j количество символов в поле вывода переменной i).

Интерпретация необязательных элементов flags:

флаг '-' - выравнивание результата по левому краю и дополнение пробелами справа;

флаг '+' - результат преобразования значений со знаком начинается с символа '+' или '-';

флаг ' ' (пробел) - результат преобразования положительных значений начинается с пробела, а отрицательных - с символа '-';

флаг '#' - вывод в альтернативной форме представления результата (при использовании форматов c,s,d,i,u флаг '#' игнорируется; для формата o символ '0' предшествует ненулевому значению, x или X - результат начинается с пары символов "0x" или "0X", e,E,f - всегда используется десятичная точка, g или G - аналогично e или E, но незначащие нули отсутствуют).

Флаги могут использоваться в произвольной комбинации, но флаг '+' обладает приоритетом перед флагом ' '. По умолчанию, результат преобразования выравнивается по правому краю и дополняется слева нулями или пробелами.

Интерпретация необязательного элемента width:

n - вывод не менее n символов, дополненных, при необходимости, пробелами (n - десятичная константа);

0n - вывод не менее n символов, дополненных, при необходимости, нулями слева;

* - значение width определяется очередным аргументом функции.

По умолчанию, а также при недостаточной явно указанной ширине поля вывода, производится его расширение до размеров результата. Интерпретация необязательного элемента prec:

.0 - для форматов d,i,o,u и x устанавливается точность по умолчанию, а для e, E и f десятичная точка не выводится;

.n - вывод не более n символов с округлением или усечением, в зависимости от типа данных, результата;

* - значение prec определяется очередным аргументом функции.

Интерпретация модификаторов mode:

F - аргумент рассматривается как указатель типа far;

N - аргумент рассматривается как указатель типа near;

h - аргумент, преобразуемый по формату d,i,o,u,x или X, имеет тип short int;

l - аргумент, преобразуемый по формату o,u,x или X, имеет тип long int.

В поле mode может записываться один из перечисленных символов - модификаторов. Применение модификаторов обязательно для корректного представления указателей и данных типов short int и long int. Каждая функция вывода возвращает количество выведенных символов, а в случае ошибки -

значение EOF. Рассмотрим вопросы программирования ввода данных потоком. Пример ввода данных:

```
...
int n, m;
...
scanf("%d%i",&m,&n);
...
```

Здесь первый параметр - форматная строка, определенная символьной константой, а n и m - элементы множества вводимых данных. Отметим, что список аргументов функций семейства scanf состоит из указателей полей. Последнее обусловлено принятой в языке С схемой связи аргумент-параметр, где из-за передачи значений аргументов, а не их адресов, приходится при необходимости изменения функцией поля аргумента передавать указатель поля аргумента.

Особенности интерпретации форматной строки функциями ввода scanf:

– символы форматной строки разделяются на неотображаемые и отображаемые обычные символы и символы спецификации преобразования;

– любой из неотображаемых символов ' ', '\t' и '\n' предписывает пропуск всех символов входного потока до первого отображаемого символа;

– любой символ, за исключением ' ', '\t', '\n' и '%', считается отображаемым, а его появление в форматной строке предписывает проверку наличия такого же символа во входном потоке;

– спецификация преобразования в форматной строке включает начальный символ '%' и символы описания формата преобразования аргумента функции ввода.

Спецификация формата ввода данных: %[*][width][mode]type (в квадратных скобках указаны необязательные элементы). Здесь:

символ '*' указывает на необходимость чтения данных из входного потока без присваивания результата преобразования;

width - максимальное количество читаемых символов;

mode - явное назначение размера и типа поля, адресуемого аргументом функции ввода;

type - тип преобразования.

Интерпретация обязательного элемента type:

d или D - десятичное целое число со знаком типа int или long;

i или I - десятичное, восьмеричное или шестнадцатеричное целое число со знаком типа int или long;

o или O - восьмеричное целое число без знака типа int или long;

u или U - десятичное целое число без знака типа int или long;

x или X - шестнадцатеричное целое число со знаком типа int или long;

f или F - десятичное число со знаком с фиксированной точкой;

e или E - десятичное число со знаком с плавающей точкой;

c - символ;

s - строка символов, ограниченная во входном потоке символом ' ' или '\n', представляемая в памяти с дополнительным символом '\0' (лидирующие пустые символы при вводе пропускаются);

p - значение указателя, представленное в форме YYYY (near) или XXXX:YYYY (far или huge), где X и Y - шестнадцатеричные цифры;

n - запоминание текущего количества введенных символов в поле целого типа, адресуемым очередным аргументом (например, оператор scanf("%d%d%n%d",&i,&j,&k,&l) помещает в переменную k количество символов, прочитанных при вводе переменных i и j).

Интерпретация элемента width:

n - ввод не более n символов (n - десятичная константа).

Интерпретация модификаторов mode аналогична ранее рассмотренной интерпретации для функций вывода printf, разница лишь в направлении передачи данных.

Особый случай преобразования - ввод символьных строк по образцу. Образец определяет множество допустимых символов во входном потоке и представляется строкой символов, заключенных в квадратные скобки. Если первым символом образца является '^', то допустимыми символами считаются все, кроме перечисленных далее в образце. Непрерывный в коде ASCI диапазон символов образца можно представить первым и последним символом, разделенными символом '-'.

Примеры использования образца в спецификациях формата ввода:

%[abcd] - перечисление допустимых символов;

%[a-d] - указание диапазона допустимых символов;

%[A-F0-9] - определение символов шестнадцатеричных чисел;

%[^0-9] - определение нецифровых символов.

Функции ввода данных семейства scanf прекращают обработку входного потока по следующим причинам:

– форматная строка обработана;

– достигнут конец файла или обнаружен символ конца строки при вводе данных из области памяти;

– очередной символ во входном потоке не совпадает с отображаемым символом в форматной строке.

Все разновидности функций scanf возвращают количество успешно введенных и размещенных в памяти данных (пропущенные поля входного потока в общую сумму не включаются). Если при чтении входного потока

достигнут конец файла, а в случае ввода данных из памяти функциями ssanf или vssanf обнаружен символ конца строки '\0', то возвращается значение EOF [1].

1.10 СТРУКТУРЫ ДАННЫХ

1.10.1 Виды организации хранения данных в памяти

Любая система программирования предоставляет возможность манипулирования некоторым множеством структур данных. Например, в языке С рассматриваются скалярные переменные, массивы, строки и структуры как агрегаты данных. При разработке программ часто удобнее оперировать естественным по отношению к решаемой задаче понятием абстрактной структуры (множества, графа или сети, очереди, конечного автомата и т.п.). В случаях, когда программирование ведется не на специализированном языке, приходится вводить промежуточные структуры, надстраиваемые над памятью ЭВМ из элементов используемого языка.

Нижний уровень надстраиваемых структур (уровень хранения) данных может иметь векторную либо списковую организацию памяти.

Векторная память поддерживается почти всеми языками высокого уровня и предназначена для хранения массивов различной размерности. Каждому массиву выделяется н е п р е р ы в н ы й участок памяти достаточного размера. Адресация элементов массива определяется некоторой адресной функцией, связывающей адрес и индексы элемента. Подобным образом может быть организовано и представление агрегатов данных, где элементы не обязательно однородны, но размещаются последовательно.

Пример адресной функции для однородного трехмерного массива X(1...n1,1...n2,1...n3):

```
I(i,j,k)=n1*n2*(i-1)+n2*(j-1)+k, i={1...n1}, j={1...n2}, k={1...n3}.
```

Для размещения такого массива требуется область размером n1*n2*n3 элементов. Рассматривая такую область как вектор (массив) Y(1...n1*n2*n3), можно установить соответствие элемента массива X элементу памяти для его размещения: X(i,j,k) <===> Y(I(i,j,k)).

Элементы массива X размещаются в последовательности

```
X(1,1,1), X(1,1,2), X(1,1,3),... X(n1,n2,n3-1), X(n1,n2,n3).
```

Современные языки высокого уровня поддерживают понятие многомерного массива, отображая его подобным рассмотренному способом на непрерывную область памяти. При программировании на таких языках необходимость введения адресных функций возникает лишь в ситуациях, когда

требуется изменить способ отображения с учетом особенностей конкретной задачи.

Например, пусть требуется обеспечить компактное представление верхней треугольной матрицы X с диагональю порядка n.

Адресная функция здесь

```
K(i,j)=(n-i/2)*(i-1)+j, i={1...n}, j={i...n}.
```

Для хранения элементов матрицы необходимо выделить в рабочей памяти массив Y(1...n*(n+1)/2), а для доступа к ним использовать отображение

```
X(i,j) <===> Y(K(i,j)), i={1...n}, j={i...n}
```

(допустимые комбинации значений индексов здесь строго фиксированы условиями задачи).

Списковая организация памяти применяется в случаях, когда возможность доступа к любому именованному элементу памяти не используется, а потребности задачи практически удовлетворяются лишь возможностью перехода от одного элемента данных к другому. Очевидно, что в таких случаях *необязательно* размещать последовательно обрабатываемые элементы в смежных областях памяти. Для адресации элементов данных достаточно каждый элемент дополнить полем ссылки (указателем) на область размещения следующего элемента и отдельно задать ссылку на первый элемент.

Использование указателей для связи элементов данных позволяет легко отразить различные виды взаимосвязей данных. Важным понятием при манипулировании указателями является значение "пустого" указателя, используемое для обнаружения конца последовательности связей.

Линейный список (список) – организация некоторого множества элементов данных, при которой каждый элемент содержит собственно данные и указатель на расположение в памяти следующего элемента множества.

Циклический список, в отличие от линейного, допускает многократное прохождение списка, начиная с любого элемента.

Симметричный список позволяет организовать продвижение по связям в любом направлении и удобен для выполнения операций включения и исключения элементов в заданном месте.

Очевидно, что нет принципиальных ограничений на вид структур, представляемых списками. Если элементом списка является другой список, то говорят о списковых структурах.

Наиболее естественно и эффективно списки используются для представления множеств, элементы которых имеют переменную длину. Поле данных элемента списка в таких случаях включает указатель на размещение элемента множества. В общем случае, элемент списка должен содержать явно либо неявно механизм интерпретации поля данных.

Пример образования и использования списка на языке С:

```
#define N 100

int   head;         /*  Указатель списка */
int   next[N];      /*  Поля ссылок */
float data[N];      /*  Поля данных */

float Summa;    /* Текущее значение суммы */

/* Построение списка положительных элементов */

for (head=-1, i=0; i<N; i++) {
 if (data[i] > 0) {
  next[i]=head, head=i;
 }
}


/* Использование списка - суммирование полей данных */

for (Summa=0, i=head; i>=0; i=next[i])
 Summa+=data[i];
```

Отметим наиболее существенные достоинства списков.

Во-первых, большое разнообразие и гибкость операций над списками:

– включение новых элементов;
– исключение элементов;
– объединение и разбиение списков произвольного размера;
– переупорядочение элементов списка и другие операции.

Во-вторых, операции над элементами списка и списками выполняются над указателями без перемещения полей данных.

Недостаток списков - неэффективность выполнения произвольного доступа к элементам [1].

1.10.2 Абстрактные структуры данных

Совокупность обрабатываемых элементов данных и взаимосвязей между ними вне аспекта физического размещения - абстрактная структура данных.

Примеры абстрактных структур:

– множества,
– очереди и стеки,
– строки (последовательность элементов некоторого алфавита),
– графы и их важная разновидность - деревья,
– таблицы и т.п.

Напомним определения некоторых абстрактных структур.

Множество – неупорядоченная совокупность элементов (данных). По определению множество не содержит повторяющихся элементов. Если значения элементов повторяются, то говорят о комплекте.

Очереди и стеки – множества, элементы которых упорядочены по времени включения.

Очередь предполагает доступность только того элемента, который был помещен в нее ранее других.

Очередь с двумя концами - дек, допускает выполнение операций записи - выборки с обоих концов.

В **стеке** доступен только тот элемент, который был помещен позже других. Обращение к стеку связано с одной единицей хранения - вершиной стека.

Конечная последовательность элементов некоторого алфавита – **строка**.

Ориентированный **граф** – пара множеств вершин и дуг, где элементы множества дуг – упорядоченные пары вершин. Для нагруженного графа дополнительно определяется соответствующее каждой дуге значение (вес дуги).

Дерево – частный случай ориентированного графа, где имеется лишь одна вершина с нулевой степенью захода, а остальные вершины имеют единичную степень захода (степень захода - число дуг, входящих в вершину графа).

Таблица – множество элементов, организованное таким образом, что каждый элемент и, возможно, его расположение однозначно определяются его ключом. В отличие от списков, элемент таблицы идентифицирует только самого себя. В контексте понятия систем баз данных таблица - базовый элемент реляционной модели данных, а ее элементы называют записями(кортежами). Идентификация записей ключами оправдана тем, что по существу решаемых задач обработки данных обычно для различения записей существенна только небольшая часть записи, называемая ключом.

Абстрактные структуры данных могут отображаться как в векторной, так и списковой памяти. Выбор вида организации памяти определяется задачами использования представляемых данных [1].

1.10.3 Отображение структур данных в памяти

Любые структуры данных могут отображаться в памяти, распределяемой как статически на этапе написания программы, так и динамически на этапе выполнения программы. Статическое распределение реализуется средствами отображения операционных объектов языка программирования на этапе трансляции и/или вызова программного модуля. Операционные объекты программ на языке С могут размещаться в статической, динамической (стековой) и регистровой памяти, но схема их размещения является статической. Динамическое распределение памяти ассоциируется с операциями порождения

и уничтожения новых объектов по запросу программы, которая в языке С не реализована. Программист вынужден заниматься явным описанием таких операций посредством обращения к функциям захвата и освобождения областей памяти.

Рассмотрим набор библиотечных функций языка С, используемых для организации динамического распределения памяти (см. файл alloc.h,где декларировано и значение пустого указателя NULL):

выделение памяти для размещения nelem объектов размером elsize байт и заполнение полученной области нулями – ункция

void *calloc(unsigned nelem, unsigned elsize) возвращает ука- затель на распределенную область или NULL при нехватке памяти.

coreleft - получение размера неиспользуемой памяти в байтах:

unsigned coreleft(void) - модели tiny, small и medium,

unsigned long coreleft(void) - модели compact, large и huge;

free - освобождение блока памяти, адресуемого указателем blk:

void free(void *blk);

malloc - выделение области памяти для размещения блока размером nbytes байт:

void *malloc(unsigned nbytes) возвращает указатель на распределенную область или NULL при нехватке памяти.

realloc - изменение размера размещенного по адресу blk блока на новое значение nbytes и копирование, при необходимости, содержимого блока:

void *realloc(void *blk,unsigned nbytes) возвращает указатель на перераспределенную область или NULL при нехватке памяти.

Представленные далее функции предназначены для операций над памятью, распределяемой в дальних областях памяти. Идентификаторы функций дополнительно снабжены префиксом far, а типы аргументов изменены с учетом возможности манипулирования большими размерами блоков памяти:

```
void far *farcalloc(unsigned long nelem, unsigned long elsize),
unsigned long farcoreleft(void),
void farfree(void far *blk),
void far *farmalloc(unsigned long nbytes),
void far *farrealloc(void far *blk, unsigned long nbytes).
```

При использовании динамически распределяемой памяти следует учитывать, что освобождение памяти некоторой области не вызывает сжатия пустых областей [1].

1.10.4 Примеры представления структур данных

Базовым понятием любой абстрактной структуры является понятие множества. Множество может быть задано либо перечислением его элементов, либо назначением правила установления принадлежности любого объекта множеству. Множество можно представить в памяти как в виде массива, так и списка элементов. Зарезервированная для представления множества память может рассматриваться с точки зрения типа и значения хранящихся данных.

Использование типа данных - простейший и наиболее распространенный прием определения правила установления принадлежности множеству в программировании. Здесь адресуемый некоторым образом элемент памяти взаимно однозначно соответствует элементу множества. Понятие типа поддерживается на уровне используемого языка программирования. В языке С каждый операционный объект имеет атрибут типа. Динамическое порождение объектов осуществляется посредством захвата памяти, адресуемого указателями на объекты заданного типа (пример со структурой ARC приведен ниже). Уничтожение объекта означает освобождение области памяти для его представления. Выбор способа представления любой абстрактной структуры зависит от набора операций над данными. Например, над множествами могут выполняться операции объединения, пересечения, вычитания и другие. Элементарное действие при выполнении подобных операций - проверка вхождения некоторого значения в заданное множество. Такое действие реализуется процедурами поиска, вопросы построения которых рассмотрены ранее.

```
/* Пример процедуры проверки вложенности множеств */

     int subsetis(int a[], int na, int b[], int nb) {
      int i;
      for (i=0; i<na; i++) {
       if (psearch(b,nb,a[i])<0) return(0);
      }
      return(1);
     }
```

Очевидно, эффективность процедуры subsetis можно повысить, если упорядочить массивы a и b и использовать процедуру дихотомического поиска dsearch.

```
/* Пример улучшенной процедуры проверки вложенности множеств */

     int subsetis(int a[], int na, int b[], int nb) {
      int dsearch(int x[], int nx, int y);
      int i,j;
      for (i=j=0; i<na; i++) {
```

```
      if ((j=dsearch(b+j,nb-j,a[i]))<0) return(0);
     }
     return(1);
    }
```

Наиболее эффективно операция проверки принадлежности множеству числовых значений реализуется с использованием понятия характеристической функции. Характеристическая функция такого множества - массив логических величин, в котором элемент i указывает принадлежность значения i множеству. Подобно ранее рассмотренной процедуре поиска на основе инвертированного массива, операция проверки принадлежности при этом выполняется за один шаг. Использование характеристических функций позволяет реализовать операции объединения, пересечения или разности множеств посредством соответствующих элементарных логических операций над элементами массивов. В языке С нет понятия массива битов, поэтому характеристические массивы множеств приходится представлять целочисленными либо символьными массивами, а в случае ограничений по памяти выполнять программное моделирование массивов битов. Рассмотрим пример компактного представления характеристического массива множества целых неотрицательных чисел.

```
/* Отображаемое множество неотрицательных целых чисел */

      int X[N];
      ...
/* Указатель характеристического массива */

      char *x_mask;

/* Рабочие переменные */
      int i,j
      ...
/* Захват памяти для характеристического массива */

      for (i=j=0; i<N; i++) if (X[i]>j) j=X[i];
      if ((x_mask=calloc(1,(++j>>3)+1))==NULL) {
       printf("\n Выход из-за нехватки памяти...");
       exit(1);
      }

/* Инициализация характеристического массива */

      for (i=0; i<N; i++) {
       j=X[i], x_mask[(j>>3)]|=(1<<(j&7));
      }
      ...
/* Функция проверки принадлежности значения z множеству X */

      int z_in_x(int z) {
```

```
        return((x_mask[(z>>3)]&(1<<(z&7)))? 1:0);
       }
```

Существенными чертами очередей, деков и стеков является зависимость количества элементов от времени. Возможны два конкурирующих варианта стратегии распределения памяти: статическое распределение требует знания предельного размера множества, а динамическое влечет накладные расходы на манипулирование блоками памяти. Пример процедуры с динамическим распределением памяти:

```
#include <alloc.h>
#include <stdlib.h>

/* Структура элемента списка дуг */

struct ARC {
 unsigned a; /* Начальная вершина дуги */
 unsigned b; /* Конечная вершина дуги */
 long     w; /* Вес дуги a->b */
 struct ARC *next; /* Указатель описания следующей дуги */
};

struct ARC *arcs=NULL; /* Указатель списка дуг графа */

/* Процедура ввода описания графа в виде списка дуг */

  void main(void) {
   unsigned A,B;
   long W;
   struct ARC *p;

   W=coreleft()/sizeof(struct ARC));
   printf("\nДопустимое количество дуг %ld",W);

/* Ввод списка дуг графа */

   while(printf("\n a,b,w-? "), scanf("%u,%u,%d",&A,&B,&W)) {
    if ((p=(struct ARC *)malloc(sizeof(struct ARC)))==NULL)
     exit(1);
    p->next=arcs, arcs=p; p->a=A, p->b=B, p->w=W;
   }

/* Печать введенного списка дуг */

   printf("\nВведенный список дуг: \n");
   for (p=arcs; p!=NULL; p=p->next)
    printf("\n %u %u %d",p->a,p->b,p->w);


/* Освобождение памяти */

   while (arcs!=NULL) {
    p=arcs, arcs=p->next, free(p);
```

```
    }
  }
```

Единственным видом из рассмотренных ранее абстрактных структур данных, для которого в языке С имеется операционный набор библиотечных функций, являются строки.

Строка в системах программирования на языке С - последовательность однобайтных символов, завершающаяся символом конца строки '\0'(внутренний код 0). Очевидно, что такое представление обусловливает последовательную схему обработки элементов строки.

Набор групп операций над строками в системе программирования Turbo-C - изменение, связывание, сравнение, преобразование, копирование и поиск (описание функций приведено в файле string.h) [1].

1.11 СОРТИРОВКА И ПОИСК ДАННЫХ

1.11.1 Характеристика проблемы сортировки

Сортировка – процесс упорядочения размещения данных в оперативной или внешней памяти в соответствии с заданным законом изменения ключевого признака на множестве адресов или номеров записей. Цель сортировки - установление соответствия между физическим и логическим порядками следования записей.

Логическая упорядоченность записей в последовательно организованных структурах данных может быть задана отношением между любыми двумя записями посредством назначения операции сравнения ключевых признаков. Обычно используются операции сравнения вида "не меньше" либо "не больше", определяющие соответственно упорядоченность записей по неубыванию либо невозрастанию значений ключевых признаков.

В общем случае запрос на сортировку данных должен включать правило проверки упорядоченности элементов любой пары записей. Цель операции сортировки – истинность условия упорядочения для всех пар записей. Очевидно, что все случаи сортировки можно свести к задаче упорядочения массива по возрастанию значений его элементов.

Процесс сортировки реализуется с проведением следующих операций:

– сравнение ключевых признаков записей;
– пересылка содержимого записей.

Операция пересылки содержимого записей имеет более значимую трудоемкость по отношению к операции сравнения ключевых признаков.

Критерии оценки эффективности процедур сортировки:

– количество операций пересылки содержимого записей;
– объем требуемой памяти помимо памяти исходного массива.

Теоретический предел оценки трудоемкости сортировки C= n*log2(n), где n - количество сортируемых элементов данных [1].

1.11.2 Методы внутренней сортировки

Здесь будут представлены основные идеи конструирования процедур сортировки, реализующих наиболее распространенные базовые методы упорядочения данных.

Рассмотрим задачу упорядочения массива целых чисел X={ x(i), i=0...n-1 } в порядке возрастания их значений. Представляемые процедуры сортировки написаны на языке C, их параметрами является указатель массива x и количество сортируемых элементов n.

Метод «вставки». Сущность метода "вставки": при рассмотрении элемента x(i) организуется его "вставка" в предварительно упорядоченное подмножество из (i-1) элемента, i=1...n-1. "Вставка" включает операции поиска места элемента x(i) и его последующего размещения с целью формирования очередного упорядоченного подмножества размерностью i. Упорядоченность подмножеств на каждой итерации предоставляет возможность совместить операции поиска и размещения в цикле просмотра текущего подмножества.

```
/* Пример процедуры сортировки методом вставки  */

     void insert(int x[], int n) {
      int i,j,t;
      for (i=1; i<n; i++) {
       for (j=i; (j>0)&&(x[j-1]>x[j]); j--) {
        t=x[j-1], x[j-1]=x[j], x[j]=t;
       }
      }
     }
```

Очевидно, что:
– в худшем случае, когда исходный массив упорядочен в обратном направлении, количество операций сравнения и пересылки равно n*(n-1)/2;
– в лучшем случае, когда исходный массив уже упорядочен, количество операций сравнения равно n-1;
– среднее количество операций n*n/4.

Отсюда следует, что вычислительная эффективность процедуры сортировки методом "вставки" далека от идеальной. Однако схема такой процедуры позволяет естественно включать ее в процесс обработки потока буферизуемого потока данных.

Метод «пузырька» (обменный метод). Пусть цель сортировки - упорядочить массив чисел X={ x(i), i=0...n-1 } по возрастанию. Схема сортировки обменным методом: на первой итерации найти максимальное число и разместить его на последнем месте, на второй итерации - мак- симальное число в неотсортированном массиве из n-1 элемента и так далее до тех пор, пока не будет упорядочен весь массив.

Термин "пузырек" ассоциируется с процессом поиска очередного максимального элемента, который совмещается с перемещением элементов массива. Текущее значение "пузырька" меняется по мере продвижения к окончательной позиции. Отличительная черта такой процедуры состоит в том, что итерации сортировки можно завершить, если не было зафиксировано ни одного обмена элементов.

```
/*  Пример простой процедуры сортировки обменным методом */

     void excngs(int x[], int n) {
      int i,j,t;
      do {
       for (j=0, i=1; i<n; i++)
        if (x[i-1]>x[i]) {
         t=x[i], x[i]=x[i-1], x[i-1]=t, j=i;
        }
      } while (j);
     }
```

Очевидно, что на каждой итерации сортировки выявляется дополнительная информация об упорядоченности массива. Ниже приводится одна из наиболее совершенных версий процедур сортировки обменным методом.

```
/*  Пример улучшенной процедуры сортировки обменным методом */

     void shakers(int x[], int n) {
      int i,j,L,R,t;
      L=1, R=j=n-1;
      while (L<=R) {
       for (i=R; i>=L; i--) {
        if (x[i-1]>x[i]) {
         t=x[i-1], x[i-1]=x[i], x[i]=t, j=i;
        }
       }
       L=j+1;
       for (i=L; i<=R; i++) {
        if (x[i-1]>x[i]) {
         t=x[i-1], x[i-1]=x[i], x[i]=t, j=i;
        }
       }
       R=j-1;
      }
     }
```

Эффективность метода "пузырька" в худшем и среднем случаях аналогична методу "вставки".

Метод Шелла. Трудоемкость сортировки ранее рассмотренными методами зависит квадратично от размерности сортируемого массива. Идея метода Шелла базируется на разбиении исходной задачи на подзадачи сортировки массивов меньшей размерности. Сортируемый массив здесь последовательно разбивается на ряд групп, каждая из которых упорядочивается по методу "вставки". Разбиение на группы продолжается до тех пор, пока не будет образована группа, совпадающая по размеру с исходным массивом. На каждом этапе разбиения размер групп увеличивается, их количество сокращается, но степень упорядоченности массива повышается. Эффективность метода обусловлена сокращением общего количества пересылок записей.

Рассмотрим схему алгоритма сортировки по методу Шелла.

Шаг 1. Положим начальный шаг группы a(i)=[n/2], i=1, где [x] - операция выделения целой части числа x.

Шаг 2. Разделим исходный массив на подмножества размером a(i) (на первом шаге получится два подмножества, если значение n четно, в противном случае третье подмножество будет состоять из одного элемента x(n)).

Шаг 3. Образуем группы из элементов, имеющих одинаковые номера (индексы) внутри выделенных подмножеств (размер групп, за исключением, возможно, первой, равен 2*i, где i - номер шага).

Шаг 4. Проведем упорядочение элементов внутри каждой группы, используя метод "вставки".

Шаг 5. Если шаг группы a(i)>2, то установим новый шаг группы a(i+1)=[a(i)/2] и выполним возврат к шагу 2, в противном случае сортировка завершена.

Процедура образования подмножеств может быть произвольной, если гарантируется на последнем этапе применение процедуры сортировки методом "вставки" к исходному множеству данных.

```
/*  Пример процедуры сортировки методом Шелла */

     void shells(int x[], int n) {
      int i,j,k,m,t;
      if (n<=1) return;
      for (k=1; k<=n; k<=1);
      for (m=k-1; m; m>>=1) {
       for (k=n-m, j=0; j<k; j++)
        for (i=j; (i>=0)&&(x[i+m]<x[i]); i-=m) {
         t=x[i+m], x[i+m]=x[i], x[i]=t;
        }
      }
     }
```

Трудоемкость сортировки по методу Шелла в среднем оценивается зависимостью n**(3/2)/2.

Наиболее эффективным по быстродействию методом сортировки является метод так называемой **"быстрой" сортировки Хоара**. Основная идея метода быстрой сортировки основывается на перестановке элементов в последовательности, зависящей от расстояния. Например, для массива n элементов, первоначально размещенных в обратном порядке, количество пересылок равно n/2, если вначале поменять местами левый и правый элементы, а затем последовательно продолжить движение с обеих сторон. Схема процедуры быстрой сортировки базируется на рекурсивном разделении исходного массива на частично упорядоченные подмножества до тех пор, пока каждое из подмножеств не будет состоять из одного элемента.

```
/*  Пример процедуры быстрой сортировки */

     void qsorts(int x[], int n) {
      void sort(int a[], int L, int R);

 /* Вызов рекурсивной процедуры сортировки */
      sort(x,0,n-1);
     }

 /* Рекурсивная процедура быстрой сортировки */

     void sort(int a[], int L, int R) {
      int i,j,x,t;
      i=L, j=R, x=a[(L+R)/2];
      do {
       while (a[i]<x) i++;
       while (x<a[j]) j--;
       if (i<=j) {
        t=a[i], a[i++]=a[j], a[j--]=t;
       }
      } while (i<=j);
      if (L<j) sort(a,L,j);
      if (i<R) sort(a,i,R);
     }
```

Вычислительная сложность процедуры быстрой сортировки оценивается зависимостью 2*log(2)*n*log2(n) [1].

1.11.3 Поиск данных

Поиск данных - процесс выделения из некоторого множества записей определенного подмножества, элементы которого удовлетворяют заранее заданному условию. Область определения условия поиска включает элементы

записи. Любую из разновидностей задач поиска (например, поиск по совпадению, интервалу, близости или логическому выражению) можно свести к задаче поиска по совпадению некоторого эталонного значения (ключа поиска) со значением ключа записи.

Эффективность методов поиска оценивается числом сравнений пар признаков, проведенных до момента окончания поиска. При сравнении различных методов поиска моментом его окончания принято считать момент обнаружения первой требуемой записи.

В общем случае эффективность поиска зависит от организации записей в массиве и метода доступа к отдельным записям. Для последовательно организованных массивов наилучшие условия поиска соответствуют упорядоченности записей по возрастанию либо убыванию ключевого признака.

Параметрами представленных далее процедур поиска являются указатель массива целых чисел a, количество его элементов n и искомое(эталонное) значение b. Возвращаемое значение - индекс элемента массива x, совпадающего с искомым значением b, а в случае неудачного поиска - значение -1.

```
/*  Пример процедуры последовательного поиска по совпадению
    в неупорядоченном линейном массиве */

    int psearch(int a[], int n, int b) {
      for (int i=0, i<n; i++)
       if (a[i]==b) return i;
      return -1;
     }
```

Наиболее эффективным методом поиска произвольных данных в последовательно организованных упорядоченных массивах является **дихотомический метод (метод двоичного поиска).**

```
/*  Пример процедуры дихотомического поиска по совпадению
    в упорядоченном по возрастанию линейном массиве */

     int dsearch(int a[], int n, int b) {
      int i,j,k;
      if (n<1) return -1;
      j=0, k=n-1;
      while (j<k) {
       i=(j+k)>>1;
       if (a[i]<b) j=i+1;
       else k=i;
      }
      return (a[j]!=b)? -1:j;
     }
```

Вычислительная трудоемкость процедур дихотомического поиска – log2(n) итераций поиска.

Рассмотрим другой метод поиска в упорядоченном массиве – **последовательный перебор с некоторым шагом**. Вначале здесь выбирается некоторый шаг просмотра m, задающий величину интервала записей. Далее последовательно анализируются ключи записей с номерами 1,m+1,2m+1,... n. При обнаружении интервала с требуемой записью организуется последовательный поиск внутри этого интервала.

Очевидно, что число операций сравнения составляет в среднем L(n,m)=(n/m+m)/2, откуда легко определить оптимальную величину шага просмотра m'=n**(1/2).

Вычислительная трудоемкость процедуры последовательного поиска с равномерным оптимальным шагом - n**(1/2) итераций поиска.

Аналогичный по эффективности метод поиска - использование справочника размещения записей.

Наиболее эффективный поиск реализуется на основе **адресной функции** вида i=f(p), где i - номер или адрес записи, p - значение ключа поиска.

Например, пусть ключи записей принимают значения в интервале[a,b]. Очевидно, что количество записей здесь не превышает значения (b-a+1). Предполагая, что для хранения записей используется линейный массив X(0...b-a), можно предложить для адресации записей зависимость i=(p-a).

Использование адресной функции предоставляет возможность доступа к требуемой записи за один шаг, однако, на практике не всегда удается найти подходящий вид зависимости f(p).

Вырожденный случай адресной функции i=p соответствует понятию инвертированного массива. Рассмотрим пример построения и использования инвертированного массива.

```
/* Пример исходного массива целых чисел */
     int x[]={3,5,1,7,9};
/* Размерность исходного массива */
     int n=sizeof(x)/sizeof(*x);
     ...
/* Указатель инвертированного массива */
     int *y;
/* Размерность инвертированного массива */
     int m;
     ...
/* Создание инвертированного массива */

    void prepar(int x[], int n, int *y) {
     int i,m;
/* Оценка размерности инвертированного массива */
     for (i=0; i<n; i++)
      if (x[i]>m) m=x[i];
/* Захват памяти для размещения инвертированного массива */
     if ((y=malloc(m*sizeof(*y)))==NULL) return;
```

```
/* Инициализация инвертированного массива */
     for (i=0; i<m; i++) y[i]=m;
/* Окончательное формирование инвертированного массива */
     for (i=0; i<n; i++) y[x[i]]=i;
    }
    ...
/*  Процедура поиска в инвертированном массиве */

    int invsrc(int y[], int m, int b) {
     return (b<m)? y[b]: -1;
    }
```

Рассмотренные здесь процедуры поиска не учитывают частоту выборки конкретных данных. Очевидно, что возможно размещение данных с учетом частоты их использования, при котором среднее время поиска будет минимальным. Например, данные могут быть связаны в древовидную структуру, вид которой определяется априорными частотами выборки [1].

1.11.4 Стандартные процедуры сортировки и поиска данных

Рассмотрим набор библиотечных процедур поиска и сортировки данных в системе программирования С. Представляемые процедуры предназначены для обработки массивов, последовательно размещенных в памяти одинаковых по размеру элементов данных. Внутреннее представление данных может быть произвольным, но программист должен определить понятие ключа сортировки либо поиска посредством задания процедуры-функции проверки порядка следования любой пары элементов массива.

Мнемонические обозначения аргументов библиотечных функций сортировки и поиска данных:

base - указатель последовательного массива данных в памяти;
nelem - количество элементов данных;
width - длина элемента данных в байтах;
fcmp - указатель функции сравнения элементов данных;
x,y - указатели аргументов функции fcmp;
key - эталонный элемент процедур поиска данных;
pnelem - указатель поля, содержащего количество элементов данных.

Декларация функций поиска и сортировки данных:

```
/* Сортировка линейного массива по возрастанию
   значений ключевых элементов */

     void qsort(void *base, int nelen, int width,
                int (*fcmp)(void *x, void *y));

/* Дихотомический поиск в линейном массиве,
```

```
   упорядоченном по возрастанию ключевых элементов */

     void *bsearch(void *key, void *base, int nelen, int width,
                   int (*fcmp)(void *x, void *y));

/* Последовательный поиск в линейном неупорядоченном массиве
   с добавлением отсутствующих элементов данных */

     void *lsearch(void *key, void *base, int *pnelen, int width,
                   int (*fcmp)(void *x, void *y));

/* Последовательный поиск в линейном неупорядоченном массиве */

     void *lfind(void *key, void *base, int *pnelen, int width,
                 int (*fcmp)(void *x, void *y));
```

Особенности использования функций поиска и сортировки:

– процедура-функция *fcmp должна возвращать целочисленное значение K(*x)-K(*y), где K(z) - значение ключа элемента данных z (абсолютная величина получаемой разности принципиального значения не имеет, учитывается лишь ее знак для установления относительного порядка следования элементов данных *x и *y при условии упорядочения массива по возрастанию значений ключей - знак '-' означает, что элемент *x предшествует элементу *y, знак '+' - элемент *x следует за элементом *y, а нуль означает совпадение ключей элементов *x и *y);

– процедуры поиска bsearch и lfind возвращают указатель на первый найденный элемент данных или 0 (пустой указатель NULL) в случае неудачи;

– процедура поиска lsearch возвращает указатель на первый найденный элемент, а в случае неудачи - на место размещения добавленного в конец массива элемента;

– поиск второго и последующих элементов данных с совпадающими значениями ключа возможен посредством рекуррентного обращения к функции поиска с коррекцией на каждом шаге значений указателя массива и количества оставшихся элементов.

Примеры использования процедур поиска и сортировки

```
#include <stdio.h>
#include <string.h>

char *x[]={
 "3","1","2","5","4"
};

void print(char **x, int n, char *t) {
 int i;
 printf("\n %s:",t);
 for (i=0; i<n; i++)
  printf(" %s",x[i]);
}
```

```
int fcmp(void *x, void *y) {
 return strcmp(*(char **)x,*(char **)y);
}

void main() {
 int n=sizeof(x)/sizeof(*x);
 print(x,n," Вход");
 qsort(x,n,sizeof(*x),fcmp);
 print(x,n,"Выход");
}
```

Результаты работы программы:

```
Вход: 3 1 2 5 4
Выход: 1 2 3 4 5
```

Поиск в строках символов можно выполнить обращением к функциям strchr, strstr, strbrk, strtok и др. (см. файл string.h) [1].

1.12 ПРИМЕРЫ РЕШЕНИЯ ЗАДАЧ СИСТЕМНОГО ПРОГРАММИРОВАНИЯ

1.12.1 Анализ схемы распределения памяти в MS-DOS

Пусть возникла необходимость оценки схемы использования памяти программами, загруженных любым способом в вычислительной среде MS-DOS. В такой ОС любая загружаемая программа первоначально захватывает всю память, но затем неиспользуемая память выделяется в отдельный блок, используемый для динамического перераспределения. Динамическое перераспределение обычно ведется посредством обращения к ОС через программное прерывание 0x21 со следующими видами запросов:

– выделить блок памяти указанного размера;
– освободить блок памяти;
– перераспределить блок памяти (уменьшить либо увеличить его размер).

Модуль обработки прерывания 0x21 различает перечисленные запросы по коду, помещаемому в регистр _AH (0x48, 0x49 или 0x4a). Программист на языке C может воспользоваться для тех же целей обращением к библиотечным функциям, например, malloc, free или realloc, декларированными в файле alloc.h.

Механизм управления памятью в MS DOS строится на основе однотипного представления любого ее блока в виде следующей структуры данных:

```
struct Memory_Control_Block { /* Структура блока памяти */
 char type; /* Тип: 'M' - промежуточный, 'Z' - последний блок */
 unsigned int ownr; /* PSP: сегментный адрес владельца */
```

```
 unsigned int size; /* Размер блока памяти в параграфах (16 байт) */
 char reserved[11]; /* Резервируемая область */
 char area[1]; /* Начало области памяти для использования */
};
```

Первые 16 байт называются **блоком управления памятью** (Memory Control Block - MCB). Функции типа alloc возвращают указатель на "область памяти для прикладных программ", а функции типа free по такому адресу обнуляют с учетом известного смещения поле "указатель владельца" и/или корректируют поле типа.

Все блоки памяти оказываются выравненными на границу параграфа, а размер их кратен размеру параграфа (16 байтам). ОС может просматривать блоки только последовательно - последующий блок отыскивается путем смещения на размер текущего блока и его MCB. Освобождение памяти возможно только от конца последовательности.

В среде разработке С можно получить сегментный адрес своей программы (Program Segment Prefix - PSP), используя значение глобальной системно определенной переменной

```
unsigned int _psp;
```

Зная свой PSP, можно программировать нетривиальное распределение оставшейся памяти.

Если требуется анализ списка всех блоков памяти, то необходимо иметь адрес первого выделенного блока памяти. Такой адрес можно получить косвенным путем, если воспользоваться недокументированной функцией 0x52 прерывания 0x21 MS-DOS.

Результатом обращения к такой функции являются помещаемый в регистр _ES сегментный адрес, а в регистр _BX - увеличенное на 2 значение смещения области со следующей важной системной информацией:

1) сегментный адрес первого блока памяти (смещение от начала области 0 байт, размер - 2 байта);

2) смещение и сегментный адрес описания устройств прямого доступа (смещение от начала - 2 байта, размер - 4 байта);

3) смещение и сегментный адрес описания первого драйвера из списка драйверов устройств (смещение от начала - 36 байт, размер - 4 байта) и др.

Рассмотрим пример программы, которая позволяет выделить:

– выделенные ОС блоки памяти;
– имена загруженных в память программ;
– содержимое среды оболочки для каждой программы.

Сегментный адрес среды оболочки легко определяется по документированному описанию PSP - двухбайтное поле со смещением 0x2c. Среда окружения представляется последовательно размещенными строками

символов с нулевым символом в качестве признака конца строки. Признак конца данных среды окружения - пустая строка.

Пример содержимого среды окружения:

COMSPEC=C:\COMAND.COM\0PROMPT=PG\0\0

После пары нулевых символов стандартно размещается значение 0x0001, за которым следует строка символов с именем пути и файла программы [1].

```
#include <stdio.h>
#include <dos.h>
struct { /* Структура блока памяти */
 char type;          /* Тип блока */
 unsigned int ownr;  /* Указатель владельца */
 unsigned int size;  /* Размер блока памяти */
 char reserved[11];  /* Резервируемая область */
 char area[1];       /* Начало блока памяти */
} far *mcb;

struct work { /* Структура описания имени файла программы */
 int mark; /* Признак имени файла программы */
 char name[1]; /* Начало строки имени файла программы */
} far *wrk;

void main(void) {
 unsigned int es, ax, bx, i;
 int far *env;
 char far *end;

 /* Поиск указателя списка блоков памяти */

  _AH=0x52;
  geninterrupt(0x21);
  bx=_BX-2;  /* Выделение смещения и сегмента */
  es=_ES;    /*  указателя системной области  */
  ax=peek(es,bx); /* Чтение сегментного адреса первого MCB */

 /* Просмотр блоков памяти */

  for (es=0;;es=1) {
   mcb=MK_FP(ax,0); /* Формирование указателя текущего MCB */
   printf("\nАдрес блока %p",mcb);
   printf("\nТип блока \'%c\'",mcb->type);
   printf("\nВладелец %4x",mcb->ownr);
   printf("\nРазмер блока %u",mcb->size);

  /* Вывод данных о программе-владельце блока */

   if (es && mcb->ownr) { /* Проверка наличия владельца */

   /* Вывод строк оболочки */

    printf("\nСодержимое среды оболочки:");
    env=MK_FP(mcb->ownr,0x2c);
    for (end=MK_FP(*env,0); (bx=strlen(end))>0; end+=bx+1)
```

```
      printf("\n %s",end);

    /* Вывод пути и имени файла программы */

     wrk=(struct work far *)(end+1);
     if (wrk->mark==1)  /* Проверка корректности структуры данных */
      printf("\nИмя файла программы %s",wrk->name);
    }
    if (mcb->type=='Z') break; /* Проверка полноты просмотра */
    ax+=mcb->size+1; /* Вычисление сегм. адр. следующего МСВ */
   }
}
```

1.12.2 Вывод списка установленных драйверов устройств

Операции ввода-вывода на ПЭВМ реализуются на физическом уровне программным путем микропроцессора. Программы обслуживания устройств называют **драйверами**, отражая их непосредственное участие в управлении устройством. Некоторые драйверы являются элементами системы управления вводом-выводом, но пользователь может установить собственные драйверы. В MS-DOS такая установка производится директивой DEVICE в файле CONFIG.SYS.

Иногда представляет интерес получение информации об активных драйверах по ходу работы программы.

Драйверы устройств представлены списком со следующей структурой элементов:

```
struct Driver_Code_Area {
 unsigned int NextOff;
 unsigned int NextSeg;
 unsigned int Strategy;
 unsigned int Interrupt;
 char DeviceName[10];
 char DriverProgramCodeArea[1];
};
```

Здесь

NextSeg и NextOff - сегментный адрес и смещение заголовка следующего драйвера("далекий" указатель следующего элемента списка; признак "пустого" указателя: NextSeg=NextOff=0xffff);

DevAttr - атрибуты драйвера;

Strategy - смещение программы отработки стратегии обслуживания устройства;

Interrupt - смещение программы обработки прерываний с запросом ввода-вывода.

Рассмотрим пример программы для вывода списка установленных драйверов. Для получения указателя головного элемента воспользуемся обсужденным в предыдущей задаче приемом (прерывание 0x21, функция 0x52). Компоненты указателя списка будут прочитаны из области системных данных.

В отличие от предшествующей программы, здесь не будет использоваться отображение структур данных на память средствами языка С. Причина - нестандартное хранение поля указателя. Для выборки элементов данных будут применены библиотечные функции:

int peek(int seg, int off) - чтение слова,

char peekb(int seg, int off) - чтение байта по заданным значениям сегментного адреса seg и смещения off.

```
#include <stdio.h>
#include <dos.h>
void main(void) {
 unsigned int es, bx, seg, off, i,j,k=0;

 /* Поиск указателя списка блоков памяти */

 _AH=0x52;
 geninterrupt(0x21);

 bx=_BX-2;  /* Выделение смещения и сегмента */
 es=_ES;    /*  указателя системной области  */
 seg=peek(es,bx+38); /* Выделение сегмента и смещения */
 off=peek(es,bx+36); /*  указателя первого драйвера   */
 while(off!=0xffff) {  /* Просмотр списка драйверов */
  printf("\nДрайвер %d: ",++k);
  for (i=10; i<18; i++)
   if ((j=peekb(seg,off+i))==' ') break;
   else printf("%c",j);
  bx=peek(seg,off+2); /*********************************/
  es=peek(seg,off);   /* Выделение сегмента и смещения */
  seg=bx;             /* указателя следующего драйвера */
  off=es;             /*********************************/
 }
}
```

Результаты работы программы [1]:

1) CON
2)
3)
4) CON
5) AUX
6) PRN
7) CLOCK$
8)

```
 9) COM1
10) LPT1
11) LPT2
12) LPT3
13) COM2
14) COM3
15) COM4
```

1.12.3 Получение информации о системных ресурсах

Способы получения системной информации о конкретной ПЭВМ во время выполнения программы:

– чтение стандартно распределенных областей памяти;
– чтение регистров состояния оборудования через порты обмена;
– запрос через программное прерывание к BIOS;
– использование библиотечных функций.

Сведения о распределении памяти и портов ввода-вывода - исходная рабочая информация системного программиста.

Укрупненная схема распределения памяти ПЭВМ IBM PC/XT/AT (в форме "сегментный адрес - содержимое"):

0000:0000 - векторы прерываний (256*4 байт);

0040:0000 - область данных BIOS;

0050:0000 - область памяти, управляемая ОС MS DOS (модули ОС IBMBIO.COM, IBMDOS.COM, буферы ОС, резидент COMAND.COM, резидентные программы пользователя, область памяти прикладных программ);

A000:0000 - начало области буферов экрана;

A000:0000 - буфер экрана EGA/VGA;

B000:0000 - буфер экрана MGA;

B800:0000 - буфер экрана CGA;

C000:0000 - начало области ПЗУ;

F600:0000 - интерпретатор BASIC;

FE00:0000 - модуль BIOS.

Важнейшая оперативная информация хранится в области данных BIOS, как, например:

0000:0410 (2) - флажки установленного оборудования;

0000:0413 (2) - объем оперативной памяти в килобайтах;

0000:041A (2) - адрес начала очереди введенных кодов клавиш;

0000:041C (2) - адрес конца очереди введенных кодов клавиш;

0000:041E (2) - начало буфера клавиатуры;

0000:0450 (16) - координаты курсора (колонка,столбец) по четырем возможным видеостраницам и т.д.

Некоторая информация находится в области BIOS:

F000:FFF5 (8) - номер версии BIOS (дата);

F000:FFFE (1) - код типа ПЭВМ:

FF - IBM PC;

FE - IBM PC/XT;

FD - PCjr;

FC - IBM PC/AT.

Интерпретация флажков установленного оборудования:

0 - наличие НГМД;

1 - наличие математического сопроцессора;

2,3 - размер базовой памяти (устарело);

4,5 - тип активного видеоадаптера:

00 - нет видеоадаптера;

01 - цветной 40*25 (PCjr);

10 - цветной 80*25;

11 - монохромный;

6,7 - уменьшенное на единицу количество НГМД;

9,A,B - уменьшенное на единицу количество коммуникационных адаптеров;

C - наличие игрового адаптера;

E,F - количество принтеров.

Пример программы отображения системных ресурсов:

```
#include <stdio.h>
#include <dos.h>
void main(void) {
  char *name;
  unsigned dflags;

 /* Вывод типа ПЭВМ */

  printf("\nМодель ПЭВМ: ");
  switch(peekb(0xf000,0xfffe)) {
   case 0xff: name="IBM PC";
              break;
   case 0xfe: name="IBM PC/XT";
              break;
   case 0xfd: name="IBM PCjr";
              break;
   case 0xfc: name="IBM PC/AT";
              break;
   default:   name="Unknown";
  }
  printf("%s",name);
```

```
 /* Вывод объема памяти */

  printf("\nОбьем памяти: %d K",peek(0,0x413)+1);

 /* Вывод описания оборудования */

  dflags=peek(0,0x410);
  printf("\nФлаги конфигурации оборудования: %#04x",dflags);
  if (dflags&1)
   printf("\nКоличество НГМД: %d",((dflags&0x00c0)>>6)+1);
  printf("\nСопроцессор %sустановлен",(dflags&2)? NULL:"не ");
  printf("\nКоличество коммуникационных адаптеров: %d",
         (dflags&0x0e00)>>9);
  printf("\nКоличество обслуживаемых принтеров: %d",
         (dflags&0x0c00)>>9);
  printf("\nТип видеоадаптера: ");
  switch ((dflags&0x0030)>>4) {
   case 1: name="C 40*25";
           break;
   case 2: name="C 80*25";
           break;
   case 3: name="MDA";
           break;
  }
  printf("%s",name);
}
```

Результаты работы программы:

```
Модель ПЭВМ: IBM PC/AT
Обьем памяти: 640 K
Флаги конфигурации оборудования: 0x4461
Количество НГМД: 2
Сопроцессор не установлен
Количество коммуникационных адаптеров: 2
Количество обслуживаемых принтеров: 2
Тип видеоадаптера: C 80*25
```

Другой способ получения информации о системных ресурсах - запрос через программные прерывания к BIOS:

– прерывание 0x11 - получение флагов конфигурации оборудования в регистре _AX;

– прерывание 0x12 - получение в регистре _AX размера оперативной памяти в килобайтах (значение уменьшено на единицу).

```
#include <stdio.h>
#include <dos.h>
void main(void) {
 unsigned dflags;
 unsigned memsiz;
```

```
 geninterrupt(0x11);
 dflags=_AX;
 printf("\nФлаги конфигурации оборудования: %#04x",dflags);

 geninterrupt(0x12);
 memsiz=_AX+1;
 printf("\nОбьем основной памяти: %u К",memsiz);
}
```

Результаты работы программы:

```
Флаги конфигурации оборудования: 0x4461
Обьем основной памяти: 640 К
```

Третий способ получения информации о системных ресурсах - использование библиотечных функций. Представленная ниже программа демонстрирует такие функции в системах программирования С и С++:

```
#include <stdio.h>
#include <bios.h>

void main(void) {
 unsigned dflags;
 unsigned memsiz;

 dflags=biosequip();
 printf("\nКод конфигурации оборудования: %#04x",dflags);

 memsiz=biosmemory()+1;
 printf("\nОбьем основной памяти: %u К",memsiz);
}
```

Результаты работы программы двух последних программ совпадают.

Все три представленных способа функционально эквивалентны, но последний, как самый высокоуровневый, здесь наиболее предпочтителен. Однако часто для реализации рассмотренных способов подходящие библиотечные отсутствуют, а запрос по прерыванию слишком медлителен. Первый способ всегда наиболее быстродействующий.

Рассмотрим пример ситуации, когда детализированная информация получается путем обращения непосредственно к оборудованию через порты ввода-вывода.

ПЭВМ класса IBM PC/AT хранит информацию о конфигурации в специальной CMOS-памяти (микросхема с батарейным питанием). Она имеет 64 однобайтных регистра, доступ к которым осуществляется посылкой номера регистра в порт 0x70 и выборкой или установкой содержимого регистра операцией чтения или записи порта 0x71.

Например, информация о наличии памяти:

0x15-0x16 - объем памяти на системной плате;

0x17-0x18 - объем установленной расширенной (expansion) памяти (свыше 1 Мбайта);

0x30-0x31 - объем работоспособной расширенной памяти (по данным программы самотестирования).

Пример программы вывода сведений о памяти ПЭВМ класса IBM PC/AT из CMOS-памяти:

```
#include <stdio.h>
#include <dos.h>

/* Порты адреса(номера регистра) и данных CMOS-памяти */

#define ADDR_PORT 0x70
#define DATA_PORT 0x71

/* Чтение слова из однобайтных регистров CMOS-памяти */

unsigned getwrd(int port_s) {
 union {
  unsigned word;
  char byte[2];
 } x;
 outportb(ADDR_PORT,port_s);
 x.byte[0]=inportb(DATA_PORT);
 outportb(ADDR_PORT,port_s+1);
 x.byte[1]=inportb(DATA_PORT);
 return x.word;
}

char *format="\n%s: %u K"; /* Формат вывода данных */

/* Макрос вывода данных */

#define P(X,Y) printf(format,X,getwrd(Y))

/* Регистровые адреса данных описания памяти */
enum {
 MMS=0x15, /* Обьем основной памяти */
 EMS=0x17, /* Обьем установленной расширенной памяти */
 AMS=0x30  /* Обьем работоспособной расширенной памяти */
};

void main(void) {
 P("Обьем основной памяти",MMS);
 P("Обьем установленной расширенной памяти",EMS);
 P("Обьем работоспособной расширенной памяти",AMS);
}
```

Результаты работы программы:

```
Обьем основной памяти: 640 K
Обьем установленной расширенной памяти: 384 K
```

```
Обьем работоспособной расширенной памяти: 384 К
```

Кроме сведений о конфигурации ПЭВМ, CMOS-память хранит данные о текущем времени и моменте запланированного прерывания от таймера, контрольную сумму байтов байтов описания оборудования и т.д.

Хранение информации о конфигурации системы в CMOS-памяти избавляет от необходимости аппаратной настройки ПЭВМ при ее изготовлении или замене элементов (периферийных устройств, блоков памяти). Вместе с тем, возникает опасность разрушения содержимого CMOS из-за неисправности источника питания или неправильной работы прикладных программ. Содержимое CMOS можно контролировать и изменять автоматически вызываемой на этапе начальной загрузки операционной системы программы SETUP. Нетрудно разработать программу сохранения-восстановления CMOS памяти [1].

1.12.4 Обработка прерываний в ПЭВМ типа IBM PC/XT/AT

Под **прерыванием** понимают передачу управления заранее подготовленной программе обработки некоторой ситуации в ответ на асинхронный по отношению к текущему вычислительному процессу внешний аппаратный сигнал или запрос команды программного прерывания (команда типа INT).

Механизм обработки прерываний в ПЭВМ типа IBM PC/XT/AT базируется на основе вектора прерываний, который имеет 256 четырехбайтных элементов и располагается в начале оперативной памяти. Элементы векторов содержат "далекий" адрес программы обработки прерывания в формате «сегмент : смещение».

Любые прерывания с точки зрения микропроцессора идентифицируются номером в диапазоне 0...255. При возникновении прерывания с некоторым номером слово состояния текущей программы (регистры микропроцессора CS, IP и регистр флагов) запоминается в стеке. Далее по номеру прерывания в регистры CS и IP загружается адрес точки входа в соответствующую программу обработки прерывания. Именно последнее означает переход к программной обработке прерывания. Программа обработки прерывания должна, как правило, завершаться командой IRET, которая восстанавливает из стека содержимое регистров слова состояния текущей программы. В системе программирования Turbo-C программа обработки прерывания должна оформляться как функция с атрибутом interrupt.

Программирование процессов обработки аппаратных прерываний требует знакомства со следующими понятиями:

– регистр флагов микропроцессора;

– уровни и приоритеты аппаратных прерываний;
– обслуживание контроллера прерываний.

Интерпретация разрядов регистра флагов микропроцессоров фирмы Intel:

а) микропроцессор Intel 8080: 0: CF - перенос или заем; 2: PF - четность результата; 4: AF - вспомогательный (десятичный) перенос; 6: ZF - нулевой результат; 7: SF - знак результата;

б) дополнительные флаги микропроцессора Intel 8086: 8: TF - пошаговый режим; 9: IF - разрешение прерываний; 10: DF - направление (автоинкремент); 11: OF - переполнение.

Единичное значение флага IF разрешает микропроцессору реагировать на аппаратные прерывания. Установка и сброс этого флага может выполняться, соответственно, командами ассемблера STI и CLI. При программировании на языке C в системе Turbo-C для тех же целей можно воспользоваться библиотечными функциями enable() и disable(). Различают три вида причин прерываний:

– процессорные прерывания - обнаружение исключительных ситуаций микропроцессором;
– аппаратные прерывания - сигнал от внешнего к микропроцессору контроллера прерываний (от аппаратуры ЭВМ);
– программные прерывания - инициатива программы.

Процессорные и аппаратные прерывания жестко привязаны к вектору прерываний.

Привязка номера прерывания и причины возникновения процессорных прерываний предопределена изготовителем микропроцессора:

0 - деление на нуль;

1 - пошаговый режим;

2 - немаскируемое прерывание (напр., из-за падения напряжения питания, сбоя памяти);

3 - ситуация приостановки;

4 - переполнение.

Привязка аппаратных прерываний к вектору прерываний реализуется посредством программируемого контроллера прерываний и предопределена изготовителем ПЭВМ. Контроллер прерываний ПЭВМ обслуживает 8 уровней прерываний, но у IBM PC/AT таких контроллеров два. Каждому уровню прерывания соответствует определенный аппаратный сигнал, обозначаемый как IRQx, где x - номер сигнала. Набор сигналов прерывания ПЭВМ типа IBM PC/XT (в скобках - первого контроллера для IBM PC/AT):

0 - таймер;

1 - клавиатура;

2 - сигнал второго контроллера прерываний;

3 - коммуникационный адаптер COM1 (COM2);
4 - коммуникационный адаптер COM2 (COM1);
5 - винчестер (принтер);
6 - НГМД;
7 - принтер.
Набор сигналов прерывания второго контроллера IBM PC/AT:
8 - таймер времени дня (CMOS);
9 - программный аналог IRQ2;
13 - сопроцессор;
14 - винчестер
(остальные сигналы зарезервированы).

Приоритет прерываний убывает с возрастанием номера сигнала, но учитывая связь контроллеров в ПЭВМ типа IBM PC/AT прерывания IRQ8...IRQ15 имеют приоритет 2.

Сигналы прерывания отображаются на вектор прерываний следующим образом (номер сигнала - номер вектора):

IRQ0...IRQ7 - Ox8...0xf; IRQ8...IRQ15 - 0x70...0x77.

Программируемый контроллер прерывания для программиста можно условно представить однобайтными регистрами данных и управления. Регистр данных первого контроллера прерываний доступен для обмена через порт 0x21, а второго - через порт 0xa1. Адреса портов регистров управления - 0x20 и 0xa0 соответственно. Регистр данных предназначен для чтения и установки маски прерываний сигналов. Сигналам IRQ0 и IRQ8 соответствуют младшие разряды регистра данных.

```
/* Программа чтения текущих масок прерывания */

#include <stdio.h>

#define PIC_D0 0x21
#define PIC_D1 0xa1

#define P(X)\
 printf("\nПорт %#02x: маска прерывания %#02x",X,inportb(X))

void main() {
 P(PIC_D0);
 P(PIC_D1);
}
```

Результаты работы программы:

```
Порт 0x21: маска прерывания 0xa8
Порт 0xa1: маска прерывания 0xbd
```

Интерпретация результата:

1010 1000 - запрет IRQ3, IRQ5, IRQ7 1011 1101 - запрет IRQ8, IRQ10, IRQ11, IRQ12, IRQ13, IRQ15

Процедура установки маски прерывания является критичной к аппаратным прерываниям, поэтому на момент ее исполнения рекомендуется сбрасывать в нуль флаг IF.

```
#include <stdio.h>
#include <dos.h>

#define PIC_D0 0x21

void main() {
 unsigned char mask; /* Исходная маска прерываний */

/* Запрет прерываний от клавиатуры */

 disable();
 mask=inportb(PIC_D0);
 outportb(PIC_D0,mask&0xfd);
 enable();
 printf("\nКлавиши не обслуживаются...");

/* Работа без клавиатуры */

 sleep(10);

/* Разрешение прерываний от клавиатуры */

 disable();
 outportb(PIC_D0,mask);
 enable();
 printf("\nКлавиши обслуживаются...");
 /* ... */
}
```

После возникновения аппаратного прерывания программа его обработки должна сбросить через регистр управления контроллер прерываний в исходное состояние. Это реализуется посылкой значения 0x20 в порт 0x20 (для сигналов IRQ8...IRQ15, кроме этого, необходимо послать значение 0x20 и в порт 0xa0). Схема типичного обработчика аппаратных прерываний на языке С:

```
void interrupt() {

 /***********************************
    Операторы обработки прерывания
 ***********************************/

 outportb(0x20,0x20);  /* Сброс контроллера прерываний */
}
```

Для работы в вычислительной среде конкретной ОС необходимо знание схемы вектора прерываний, формируемого на этапе первоначальной загрузки [1].

1.12.5 Понятие системы типа "клиент-сервер"

Под системой типа **«клиент-сервер»** подразумевают систему централизованной обработки запросов. Полезность централизации в системах обработки данных обычно обусловлена необходимостью управления разделением некоторого вычислительного ресурса (программ, данных, устройств или памяти) между многими пользователями. Примеры таких систем:

– администраторы устройств;
– файловые системы(локальные и распределенные);
– системы управления базами данных и т.п.

Внутренние механизмы операционных систем и прикладных систем, построенные по рассматриваемой схеме, как правило, ориентированы на обслуживание переменного во времени количества пользователей. Интерфейс связи удобно строить на основе механизма обмена сообщениями, рассмотренного ранее. При этом сервер обычно является задачей-получателем сообщений от задач пользователя, а обратная связь реализуется репликами ответов. Протокол передачи сообщений может быть любым. При работе в среде многозадачных ОС (QNX, UNIX, OS/2, Nowell Netware) средства его построения предоставляет система. Формально сервер как механизм синхронизации процессов играет роль монитора.

Разделяют два вида серверов:

– поддержка транзакций - обслуживание независимых запросов;
– поддержка зависимых запросов задачи.

Транзакция проще в программировании - каждый запрос рассматривается как новый, поэтому сервер не должен учитывать состояние задачи пользователя или следить за ее уничтожением. Однако пользователь обязан предоставлять в каждом запросе полное описание задачи.

Поддержка зависимых запросов задачи требует от нее начального запроса на открытие ресурса и конечного запроса на его закрытие. Открытие обычно использует символическое имя ресурса, которому сервером ставится в соответствие некоторый идентификатор ресурса для последующих обращений. Например, файловая система назначает при открытии имени файла указатель структуры описания файла с его номером в системной таблице (см. функции fopen, fileno, _open). Очевидно, чтосервер здесь должен быть информирован о фактах уничтожения задач, которые могут не закрывать ресурс [1].

1.12.6 Пример построения системы "клиент-сервер" в QNX

Пусть в системе управления транспортными процессами возникает потребность определения множества справок по кратчайшим маршрутам между произвольными вершинами некоторой транспортной сети. Можно показать, что выдачу справок для реальных транспортных сетей эффективнее производить не путем выборки из файла матрицы кратчайших маршрутов, а посредством оптимизации маршрута по запросу.

Задача поиска кратчайших маршрутов на графе транспортной сети в математическом отношении является хорошо изученной, а практический интерес в последнее время представляют лишь способы реализации вычислительных схем ее решения. Здесь мы будем использовать вариант эффективной реализации алгоритма Дийкстры. Таким образом, каждый запрос на оптимизацию маршрута интерпретируется в терминах получения характеристик дерева кратчайших маршрутов от некоторой исходной вершины до конкретно заданной вершины либо до всех остальных вершин графа сети. Взаимозависимостью запросов, ассоциированных при этом лишь с некоторой корневой вершиной, не будем пренебрегать, но для простоты рассуждений естественно возникающую идею диспетчеризации запросов оставим вне рассмотрения.

Реализация схемы решения любой задачи вынуждает выполнить предварительное планирование распределения памяти для представления переменных состояния (по крайней мере, с целью оценки реальной возможности решения задачи).

Область определения процесса оптимизации маршрутов на транспортной сети включает:

– множество данных представления графа транспортной сети;

– массивы результата оптимизации в виде описания дерева кратчайших маршрутов и расстояний от заданной корневой вершины;

– множество переменных описания инвариантных характеристик задачи оптимизации маршрутов на графе транспортной сети.

Рассмотрим последовательно состав перечисленных элементов с целью определения структуры глобальных переменных процедур оптимизации.

Графы реальных транспортных сетей обычно характеризуются незначительной степенью связности, поэтому их представление в памяти удобно выбирать в виде списковых структур данных (см. 4). Модель транспортной сети в этом случае включает следующие элементы данных:

N_dot - количество вершин графа транспортной сети;

N_lst - массив указателей списков смежности;

B_arc - массив элементов списков смежности (конечных вершин дуг графа);

W_arc - массив весов дуг.

Предполагается, что вершины графа транспортной сети пронумерованы числами 0,1,... N_dot-1.

Элементы массивов N_lst, B_arc, W_arc после нумерации вершин фиксируются следующим образом:

```
N_lst[0]=0;

N_lst[i+1]=N_lst[i]+card(i'), i=0,1,... N_dot-1;

B_arc[N_lst[i]+j]=i'(j),
W_arc[N_lst[i]+j]=w(i,i'(j)),
j=0,1,...card(i')-1, i=0,1,... N_dot-1.
```

Здесь x' – множество номеров вершин, для которых существуют направленные дуги из вершины графа x;

w(x,y) – вес дуги из вершины x в вершину y;

card(z) – размерность множества z.

Очевидно, что

```
card(N_lst)=N_dot+1,
card(B_arc)=card(W_arc)=N_lst[N_dot]                    (12.1)
```

(элемент N_lst[N_dot] содержит количество дуг графа сети по определению).

Результат построения дерева кратчайших маршрутов размещается в следующих массивах:

D_dot - массив расстояний от корня дерева;

P_dot - массив вершин кратчайшего пути.

Размерность таких массивов

```
card(D_dot)=card(P_dot)=N_dot.                          (12.2)
```

Состав инвариантных параметров задачи оптимизации маршрутов на графе транспортной сети, которые целесообразно определить один раз на этапе инициализации резидентной части программы, предопределен процедурой оптимизации маршрутов. Можно показать, что такими пара- метрами являются:

L_arc - максимальная длина дуги;

t_root - номер корневой вершины последнего полностью построенного дерева кратчайших маршрутов;

x_next и x_prec - массивы указателей следующих и предыдущих элементов списка очередей вершин;

t_size - размер горизонта планирования дерева;

t_base и t_stop - начало и конец индексов указателей очередей вершин.

Списки очередей вершин представимы в массивах следующей размерности [3]:

```
card(x_prec)=N_dot;                                    (12.3)
card(x_next)=N_dot+L_arc+1.                            (12.4)
```

Очевидно, что наиболее существенный вес здесь имеют области массивов. Заключение о возможности решения задачи при известном объеме доступной памяти практически можно обосновать его сравнением с суммой значений, получаемых на основе (1.6.1-1.6.4).

Рассмотрим пример программной реализации в среде ОС QNX справочника кратчайших маршрутов задачей-администратором со следующей схемой функционирования:

– чтение параметров размерности и описания графа транспортной сети из файла сжатого описания сети;
– подключение идентификационного имени задачи-администратора;
– переход в состояние обработки запросов.

Предполагается, что файл сжатого описания сети подготовлен заранее с помощью, загружаемой в пакетном режиме программы. Соответствие имени такого файла идентификационному имени задачи - администратора задается командной строкой запуска этой задачи:

```
interx netfile tskname
```

Здесь interx - имя файла - загрузочного модуля задачи - администратора (результата компиляции и компоновки исходного текста из файла interx.c);

netfile - имя файла сжатого описания транспортной сети;

tskname - идентификационное имя задачи - администратора. Задача - администратор обрабатывает следующие запросы:

– запрос операции расчета;
– запрос параметров сети;
– команда завершения работы.

Структуры сообщений потока информационного обмена и коды запросов определены в заголовочном файле interz.h.

Содержимое заголовочного файла interz.h: /* Декларация структур сообщений */

```
#ifndef INTERZ_H
#define INTERZ_H

typedef struct { /* Запрос на расчет маршрута */
 int code;       /* код сообщения */
 int sour;       /* номер исходной вершины */
 int dest;       /* номер конечной вершины */
} QUERY;
```

```
typedef struct { /* Запрос размерности описания графа сети */
 int code;       /* код сообщения */
 int dots;       /* количество вершин */
 int arcs;       /* количество дуг */
} CARDS;

typedef struct { /* Элемент результата расчета */
 int code;       /* код сообщения */
 int prec;       /* номер предшествующей вершины маршрута */
 long dist;      /* значение расстояния от корневой вершины */
} RESULT;

/* Коды сообщений потока межзадачного обмена */

enum {
 CALC=1, /* Запрос операции расчета */
 EXIT,   /* Команда завершения работы */
 SIZE    /* Запрос параметров сети */
};

#endif
```

Исходный текст задачи - администратора (размещается в файле interx.c):

```
#include <stdio.h>
#include <magic.h>
#include "interz.h"

int t_root;

/* Параметры описания графа сети */

int   N_dot;  /* количество вершин */
int   N_arc;  /* количество дуг */
int  *N_lst;  /* массив указателей списков смежности */
int  *B_arc;  /* массив конечных вершин дуг графа */
int  *W_arc;  /* массив весов дуг */

/* Параметры описания дерева кратчайших путей */

int *x_next;
int *x_prec;
int  t_size;
int  t_base;
int  t_stop;

/* Массивы результатов */

int  *P_dot;  /* массив вершин кратчайшего пути */
long *D_dot;  /* массив расстояний */

int mread(void *p, int s, int n, FILE *f) {
 return (fread(p,s,n,f)!=n);
}
```

```
void main(int na, char **la) {
 FILE *stream;
 QUERY buf;      /* Буфер приема запросов */
 RESULT out;     /* Буфер выдачи результатов */
 int L_arc;      /* Максимальная длина дуги */
 unsigned xid;   /* Идентификатор прикладной задачи */
 long length;

 if (na<3) {
  printf("\n\n %s","СПРАВОЧНИК КРАТЧАЙШИХ РАССТОЯНИЙ");
  printf("\n\nСинтаксис вызова:\n\n %s netfile tskname",la[0]);
  printf("\n   netname - имя файла сжатого описания сети");
  printf("\n   tskname - локальное имя задачи");
  exit(1);
 }
 if (!(stream=fopen(la[1],"r")))
  error("Ошибка открытия файла модели сети");

/* Чтение параметров размерности сети */

 if (
  mread(&N_dot,sizeof(N_dot),1,stream) ||
  mread(&N_arc,sizeof(N_arc),1,stream) ||
  mread(&L_arc,sizeof(L_arc),1,stream)
 ) error("Ошибка чтения размерности сети");

/* Захват памяти для рабочих массивов */

 t_base=N_dot+1, t_size=L_arc+1, t_stop=N_dot+t_size;
 length=(long)N_dot*(long)(sizeof(*D_dot));
 length+=(long)N_dot*(long)(sizeof(*P_dot)+sizeof(*x_prec));
 length+=(long)t_stop*(long)sizeof(*x_next);
 length+=(long)(N_dot+1)*(long)sizeof(*N_lst);
 length+=(long)N_arc*(long)(sizeof(*B_arc)+sizeof(*W_arc));
 x_next=malloc((unsigned)length);
 if (!x_next)
  error("Ошибка захвата памяти");

/* Инициализация указателей рабочих массивов */

 x_prec=x_next+t_stop;
 P_dot=x_prec+N_dot;
 N_lst=P_dot+N_dot;
 B_arc=N_lst+N_dot+1;
 W_arc=B_arc+N_arc;
 D_dot=(long *)(W_arc+N_arc);

/* Чтение массивов описания структуры сети */

 *N_lst=0, t_root=N_dot;

 if (
  mread(N_lst+1,sizeof(*N_lst),N_dot,stream) ||
  mread(B_arc,sizeof(*B_arc),N_arc,stream) ||
  mread(W_arc,sizeof(*W_arc),N_arc,stream)
```

```
 ) error("Ошибка чтения структуры сети");
 fclose(stream);

/* Переход в режим обслуживания запросов */

 if (name_attach(la[2],My_nid)) {
  while (N_dot)
   if ((xid=receive(0,&buf,sizeof(buf))) && (~xid))
    switch(buf.code) {
     case SIZE: /* Запрос параметров графа сети */
                buf.sour=N_dot;
                buf.dest=N_arc;
                reply(xid,&buf,sizeof(buf));
                break;

     case CALC: /* Заявка на оптимизацию маршрута */
                if (buf.sour!=t_root)
                 shrtpt(buf.sour,buf.dest);
                out.code=buf.code;
                out.dist=D_dot[buf.dest];
                out.prec=P_dot[buf.dest];
                reply(xid,&out,sizeof(out));
                break;

     case EXIT: /* Завершение работы */
                N_dot=0;
                break;
   }
 } else error("Ошибка подключения имени задачи");
}

/* Процедура построения кратчайших путей на графе */

void shrtpt(int i, int z) {
 int  j,k,l,m,n,w,t;
 long Q,R,S;
 static long infinity=0x7fffffffL;

/* Инициализация рабочих массивов */

 for (j=0; j<N_dot; D_dot[P_dot[j]=j]=infinity, j++);
 D_dot[x_next[t_root=N_dot]=i]=0, w=1;
 for (j=t_base; j<t_stop; x_next[j++]=N_dot);
 x_next[i]=x_prec[i]=N_dot;

/* Организация процесса ветвления */

 while (w>0) {
  for (m=N_dot; m<t_stop; m++)
   while((j=x_next[m])<N_dot) {
    if (j==z) return;
    k=x_next[m]=x_next[j], w--;
    if (k<N_dot) x_prec[k]=m;
    for (l=N_lst[j], Q=D_dot[j]; l<N_lst[j+1]; l++) {
     if ((S=D_dot[(n=B_arc[l])])>=Q)
```

```
      if ((R=Q+(t=W_arc[l]))<S) {
       if (S==infinity) w++;
       else {
        k=x_next[x_prec[n]]=x_next[n];
        if (k<N_dot) x_prec[k]=x_prec[n];
       }
       D_dot[n]=R, P_dot[n]=j, t+=m;
       if (t>=t_stop) t-=t_size;
       k=x_next[n]=x_next[t], x_prec[n]=t, x_next[t]=n;
       if (k<N_dot) x_prec[k]=n;
      }
     }
    }
  }
  t_root=i;
}
```

Прикладные задачи могут установить связь с задачей - администратором посредством проверки активности задачи - владельца идентификационного имени. После установления связи и получения параметров размерности описания графа транспортной сети можно порождать поток запросов на получение сведений о кратчайших маршрутах между любыми вершинами сети.

Исходный текст примера прикладной программы формирования запросов (размещается в файле intery.c):

```
/* Пример прикладной программы формирования запросов */

#include <stdio.h>
#include <magic.h>
#include "interz.h"

void main(na,la) int na; char **la; {
 unsigned xid;  /* Идентификатор задачи - получателя */
 QUERY buf;     /* Буфер приема запросов */
 RESULT out;    /* Буфер выдачи результатов */
 union buffer { /* Объединение буферов обмена */
  QUERY  buf;
  RESULT out;
 };

 if (na<2) {
  printf("\n\n %s","ПРОВЕРКА СПРАВОЧНИКА КРАТЧАЙШИХ РАССТОЯНИЙ");
  printf("\n\nСинтаксис вызова:\n\n %s tskname",la[0]);
  printf("\n   tskname - локальное имя задачи");
  exit(1);
 }
 if (xid=name_locate(la[1],0,sizeof(union buffer))) {

/* Запрос параметров сети */

  buf.code=SIZE;
  send(xid,&buf,&buf,sizeof(buf));
```

```
  printf("\n Размерность транспортной сети:");
  printf("\n  количество вершин %d",buf.sour);
  printf("\n  количество дуг %d",buf.dest);

/* Цикл рабочих запросов */

  buf.code=CALC;
  while(printf("\nОткуда, куда - ? "),
        scanf(" %d %d",&buf.sour,&buf.dest)) {
   send(xid,&buf,&out,sizeof(buf));
   if (out.code==buf.code)
    printf("\n Расстояние %ld через вершину %d",out.dist,out.prec);
  }

/* Завершение работы */

  buf.code=EXIT;
  send(xid,&buf,&out,sizeof(buf));
 } else error("\n СПРАВОЧНИК КРАТЧАЙШИХ РАССТОЯНИЙ ПАССИВЕН");
 printf("\n Конец работы !\n\n");
}
```

Пусть intery - имя файла загрузочного модуля прикладной задачи формирования запросов (результата компиляции и компоновки исходного текста из файла intery.c). Взаимодействие с задачей - администратором interx реализуется лишь после указания при запуске программы intery в командной строке вида

```
intery tskname
```

идентификационного имени tskname, использованного при запуске задачи interx [1].

1.12.7 Защита файлов от копирования

Ниже приведен пример программы защиты от несанкционированного копирования файлов в файловой системе FAT. Метод организации защиты - запись и контроль ключевой информации в неиспользуемую область последнего кластера выделенных файлов [2].

```
// Модуль тестирования привязки

#include <stdlib.h>
#include <stdio.h>

#include <io.h>
#include <string.h>
#include <ctype.h>
#include <direct.h>
#include <dos.h>
#include <alloc.h>
```

```
#include <conio.h>
#pragma warn -par

void far handler(unsigned deverr,
                 unsigned errval, unsigned far *devhdr) {
 _hardretn(5);  /* Code 5: DOS "access denied" error */
}

enum exit_code {
 ok=0,
 f_call,    // Ошибка обращения
 f_open,    // Контролируемый файл не найден
 f_search,  // Неудача поиска в DTF
 f_core,    // Нехватка памяти
 r_disk,    // Ошибка чтения сектора
 f_random,  // Пароль неверен
 w_disk,    // Ошибка записи сектора
 f_time     // Таймаут контроля
};

struct dtfitm { // Элемент таблицы DTF
 char s1[17];
 unsigned long l;
 char s2[11];
 char x[11]; // Имя файла
 char s3[10];
 unsigned int r; // Номер последнего прочитанного кластера
 char s4[4];
};
struct dtflst {
 unsigned int o,s,n;
 struct dtfitm e[1];
};

void isname(char *t, char *x) {
 char *y;
 for (int i=0; i<11; t[i++]=' ');
 if ((y=strrchr(x,'.'))!=0)
  for (y++, i=0; (i<3)&&y[i]; i++) t[i+8]=toupper(y[i]);
 if (((y=strrchr(x,'\\'))!=0)||
      ((y=strrchr(x,':'))!=0)||((y=x-1)!=0))
  for (y++, i=0; (i<8)&&y[i]&&(y[i]!='.'); i++)
   t[i]=toupper(y[i]);
}

struct dtfitm far *search(char *t) {
 unsigned int ps,po;
 _AX=0x5200;
 geninterrupt(0x21);
 ps=_ES, po=_BX;
 struct dtfhdr {
  unsigned int tx,ty,to,ts;
 } far *tp=(struct dtfhdr far *)MK_FP(ps,po);
 struct dtflst far *ft=(struct dtflst far *)MK_FP(tp->ts,tp->to);
 while (1) {
```

```
  for (int i=0; (i<ft->n); i++) {
   for (int j=0; j<11; j++)
    if (ft->e[i].x[j]!=t[j]) break;
   if (j==11) return &(ft->e[i]);
  }
  if (ft->o==0xffff) break;
  ft=(struct dtflst far *)MK_FP(ft->s,ft->o);
 }
 return 0;
}

int diskon(char *x) {
 return (x[1]==':')? toupper(x[0])-'A':getdisk();
}

#define INTR 0x1C

#ifdef __cplusplus

#define __CPPARGS ...
#else
#define __CPPARGS
#endif

void interrupt (*old1ch)(__CPPARGS);

static int countc=0;

void exiter(int x) {
 setvect(0x1c,old1ch);
 nosound();
 delay(300);
 for (int i=0; i<x; ) {
  sound(++i*100);
  delay(200);
  nosound();
  delay(100);
 }
 exit(x);
}

// Управление звуковыми сигналами

void interrupt new1ch(__CPPARGS) {
 countc++;
 switch (countc%20) {
  case 16: sound(800);
           break;
  case 19: nosound();
           break;
 }
}

// Контроль таймаута
```

```
void protec() {
 if (countc>50) exiter(f_time);
}

// Контроль и модификация сигнатуры файла

void main(int na,char **la) {
 old1ch=getvect(0x1c);
 countc=0;
 setvect(0x1c,new1ch);
 if (na>1) {
  _harderr(handler);
  FILE *fp=fopen(la[1],"r+b");
  protec();
  if (fp) {
   char t[11];
   isname(t,la[1]);
   protec();
   struct dtfitm far *fx=search(t);
   if (fx) {
    struct {
     char s1[11];
     unsigned int  sl;     // Размер сектора
     unsigned char ks;     // Число секторов кластера
     unsigned int  rs;     // Число резервируемых секторов
     unsigned char fn;     // Количество FAT
     unsigned int  rn;     // Число записей корневого каталога
     unsigned int  vs;     // Число секторов на томе
     unsigned char bd;     // Байт описания носителя
     unsigned int  fs;     // Число секторов в FAT
     unsigned int  ts;     // Число секторов на дорожке
     unsigned int  hn;     // Число головок
     char s2[11];
     unsigned long vi;     // Серийный номер тома
     unsigned char vl[11]; // Метка тома
     char s3[458];
    } sector;

    int disk=diskon(la[1]);
    if (absread(disk,1,0,$or))
     exiter(r_disk);
    protec();
    unsigned int clustbyte=sector.sl*sector.ks;
    char *clsbuf=new char[clustbyte];
    if (clsbuf) {
     unsigned int tailsz=fx->l%clustbyte;
     fseek(fp,long(fx->l-tailsz),0);
     fread(clsbuf,1,tailsz,fp);
     long clsnum=1l;
     clsnum+=long(sector.fn)*long(sector.fs);
     protec();
     clsnum+=long(sector.rn>>4);
     clsnum+=long(fx->r-2)*long(sector.ks);
     protec();
     if (absread(disk,sector.ks,clsnum,clsbuf))
```

```
      exiter(r_disk);
     srand(clsbuf[tailsz]);
     protec();
     for (int i=tailsz+1; i<clustbyte; i++)
      if (clsbuf[i]!=char(random(256)))
       exiter(f_random);
     protec();
     randomize();
     clsbuf[tailsz]=random(256);
     protec();
     srand(clsbuf[tailsz]);
     for (i=tailsz+1; i<clustbyte; i++) {
      protec();
      clsbuf[i]=random(256);
     }
     protec();
     if (abswrite(disk,sector.ks,clsnum,clsbuf))
      exiter(w_disk);
     delete clsbuf;
    } else exiter(f_core);
   } else exiter(f_search);
  } else exiter(f_open);
  fclose(fp);
 } else exiter(f_call);
 protec();
 exiter(ok);
}
```

1.13 ИНТЕРПРЕТАЦИЯ СИСТЕМ ПРОДУКЦИЙ НА ЯЗЫКЕ С

1.13.1 Сетевое представление системы продукций

Под **продукцией** понимают элементарную систему типа «условие-действие». Частный случай систем продукций - сети Петри и их расширения. Очевидно, что системы продукций характеризуются дискретностью поведения как упорядоченной последовательности действий, управляемой соответствующим набором условий. Управление здесь подразумевает не обязательно связанные с понятием времени этапы анализа состояния сети с целью активизации соответствующих действиям переходов. Можно показать, что система продукций эквивалентна расширенной сети Петри с задержками в переходах (TPN).

Переход t системы продукций представим четверкой функций, определяемых на множестве переменных состояния системы продукций:

A(t) - проверка условия активизации;

B(t) - изменение переменных состояния в момент активизации (входные действия);

C(t) - автономный процесс актисного состояния;

D(t) - изменение переменных состояния в момент пассивизации (выходные действия).

В случае TPN переменные состояния – разметка позиций, функции A, B, D связывают разметку позиций с весами дуг сети, а функция C – задержка во времени.

Нетрудно заметить, что цель интерпретации системы продукций - построение последовательности активизаций переходов. Решение такой задачи может решаться на основе схемы интерпретации процессов на TPN. Действительно, если некоторый переход t=i стал пассивным, то это особое событие порождает необходимость проверки условий активизации переходов j, функции A(j) которых включают переменные, измененные функцией D(i). Если условие A(j) истинно, то выполняются действия B(j), после чего активизируется переход j и проверяется возможность активизации переходов, условия активизации которых определены и на переменных функции B(j).

Таким образом, порождение процесса активизации переходов основано на восприимчивости отдельных переходов системы продукций к изменению только локальных переменных состояния. Отсюда следует, что на переходах системы продукций можно формально построить сеть, связывающую переходы с потенциальной возможностью активизации. Например, пусть TPN со стартовым переходом as и начальной нулевой разметкой позиций преставлена структурой смежности

```
as:b1;b2;b6
a1:b3;b4
a2:b5
a3:b2;b1
a4:b7
b1:af;a1
b2:af;a1
b3:a2
b4:a3
b5:a3
b6:a4
b7:af
```

Сеть интерпретации процессов в этом случае имеет вид:

```
as:a1,a4
a1:a2,a3
a2:a3
a3:af,a1
a4:af
```

Переменные состояния здесь образуют множество

```
X={b1,b2,b3,b4,b5,b6,b7}.
```

При этом предполагается, что в исходный момент времени начальное состояние

```
X={0,0,0,0,0,0,0},
```

а веса дуг TPN являются единичными. Соответствующая система продукций будет полностью определена, если записать следующие из определения TPN определения функций:

```
A(as)=0;
A(a1)=b1&&b2;
A(a2)=b3;
A(a3)=b4&&b5;
A(a4)=b6;
A(af)=b1&&b2&&b7;

B(as)=0;
B(a1)=b1--, b2--;
B(a2)=b3--;
B(a3)=b4--, b5--;
B(a4)=b6--;
B(af)=b1--, b2--, b7--;

C(as)=0;
C(a1)=d1;
C(a2)=d2;
C(a3)=d3;
C(a4)=d4;
C(af)=b1--, b2--, b7--;

D(as)=b1++, b2++, b5++;
D(a1)=b3++, b4++;
D(a2)=b5++;
D(a3)=b1++, b5++;
D(a4)=b7++;
D(af)=0;
```

(логические и арифметические операции здесь обозначены символами операции языка C/C++) [1].

1.13.2 Входное описание систем продукций

Для решения практических задач описание системы продукций конкретизируется в реальной вычислительной среде. Например, в системе моделирования дискретных процессов в реальном времени СМДП-РВ TPN (1.7.1-1.7.5) может быть закодирована в выде:

```
// РАЗДЕЛ СТАТИЧЕСКОГО ОПИСАНИЯ МОДЕЛИ
PLACES
```

```
// Позиции тестовой TPN
   int b1,b2,b3,b4,b5,b6,b7;

// РАЗДЕЛ ДИНАМИЧЕСКОГО ОПИСАНИЯ МОДЕЛИРУЕМЫХ ПРОЦЕССОВ */

TRANSITION

as: {0},
    { printf("\nМОДЕЛИРОВАНИЕ ПРОЦЕССОВ НА TPN");
      b1=b2=b3=b4=b5=b6=b7=0;
      count=0;
    },,{b1++,b2++,b6++;};

af: { b1 && b2 && b7 },
    { b1--, b2--, b7--;
      printf("\n ФИНИШ... "); };

a1: { b1 && b2 },
    { b1--, b2--; },
    { d1 },
    { b3++, b4++; };

a2: { b3 },
    { b3--; },
    { d2 },
    { b5++; };

a3: { b4 && b5 },
    { b4--, b5--; },
    { d3 },
    { b1++, b2++;
      printf("\n Время %ld, счетчик %d, ",Event_Time,++count);
    };

a4: { b6 },
    { b6--; },
    { d4 },
    { b7++; };

SOURCE

#include <stdio.h>
long d1=2, d2=3, d3=4, d4=10;
int count;
```

После трансляции такого исходного текста система СМДП-РВ подготовит следующий текст программы, пригодный для обработки системой программирования на языках С/С++:

```
// ТЕКСТ ДЕТАЛИЗАЦИИ ОПИСАНИЯ МОДЕЛИ

#include <stdio.h>
long d1=2, d2=3, d3=4, d4=10;
```

```
int count;

// ОПРЕДЕЛЕНИЕ ПЕРЕМЕННЫХ СОСТОЯНИЯ

int b1, // 0
    b2, // 1
    b3, // 2
    b4, // 3
    b5, // 4
    b6, // 5
    b7; // 6

// ОПИСАНИЕ ФУНКЦИЙ УСЛОВИЙ АКТИВИЗАЦИИ И ДЕЙСТВИЙ ПЕРЕХОДОВ

int as_1(void), // 0
    af_1(void), // 1
    a1_1(void), // 2
    a2_1(void), // 3
    a3_1(void), // 4
    a4_1(void); // 5

int (*smdp_1[])(void) = {
 as_1,af_1,a1_1,a2_1,a3_1,a4_1
};

void as_2(void),
     af_2(void),
     a1_2(void),
     a2_2(void),
     a3_2(void),
     a4_2(void);

void (*smdp_2[])(void) = {
 0,af_2,a1_2,a2_2,a3_2,a4_2
};

long a1_3(void),
     a2_3(void),
     a3_3(void),
     a4_3(void);

long (*smdp_3[])(void) = {
 0,0,a1_3,a2_3,a3_3,a4_3
};

void as_4(void),
     a1_4(void),
     a2_4(void),
     a3_4(void),
     a4_4(void);

void (*smdp_4[])(void) = {
 as_4,0,a1_4,a2_4,a3_4,a4_4
};
```

```
// ОПРЕДЕЛЕНИЕ ФУНКЦИЙ УСЛОВИЙ АКТИВИЗАЦИИ И ДЕЙСТВИЙ ПЕРЕХОДОВ

int as_1(void) { return(0); }

void as_4(void) {
 printf("\nМОДЕЛИРОВАНИЕ ПРОЦЕССОВ НА TPN");

 b1=b2=b3=b4=b5=b6=b7=0;
 count=0;
 b1++,b2++,b6++;
}

int af_1(void) {
 return(b1 && b2 && b7);
}

void af_2(void) {
 b1--, b2--, b7--;
 printf("\n ФИНИШ... ");
}

int a1_1(void) { return(b1 && b2); }

long a1_3(void) { return(d1); }

void a1_2(void) { b1--, b2--; }

void a1_4(void) { b3++, b4++; }
int a2_1(void) { return(b3); }
long a2_3(void) { return(d2); }
void a2_2(void) { b3--; }
void a2_4(void) { b5++; }
int a3_1(void) { return(b4 && b5); }
long a3_3(void) { return(d3); }
void a3_2(void) { b4--, b5--; }
void a3_4(void) {
 b1++, b2++;
 printf("\n Время %ld, счетчик %d, ",EL_time,++count); }

int a4_1(void) { return(b6); }
long a4_3(void) { return(d4); }
void a4_2(void) { b6--; }
void a4_4(void) { b7++; }

// СХЕМА СВЯЗИ ПЕРЕХОДОВ

#define N_tran 6

    int L_tran[] = { // Индексы списка связанных позиций переходов
     0,7,10,14,16,20,22 };

    int M_tran[] = { // Связи переходов
     0,1,2,3,4,5,6, 0,1,6, 0,1,2,3, 2,4, 3,4,0,1, 5,6
//   as             af     a1       a2   a3       a4
    };
```

```
// СХЕМА СВЯЗИ ПОЗИЦИЙ

    int L_plac[] = { // Индексы списка связанных переходов позиций
     0,2,4,5,6,7,8,9,9
    };

    int M_plac[] = { // Связи позиций
     1,2, 1,2, 3, 4, 4, 5, 1
//   b1   b2   b3 b4 b5 b6 b7
    };
```

Схема связи переходов и позиций по существу определяют TPN, соответствующую сети переходов системы продукций [1].

1.13.3 Интерпретация систем продукций

Здесь представлен результат конструирования процедуры интерпретации системы продукций посредством построения обрамляющей сети типа TPN на исходном тексте описания такой системы.

```
/*****************************************************************
  Процедура интерпретации системы продукций в ускоренном времени
 *****************************************************************/

// Включение описания варианта сети

#include "test.c"

// Описание списка событий

int   EL_next[N_tran]; // Поля ссылок
long  EL_time[N_tran]; // Моменты времени
#define HEAD EL_next[0]  // Указатель списка активных переходов
#define TEMP EL_time[0]  // Момент очередного особого события

void main(void) {
 int i,j,k,l,m,n,p,q,t;
 long f;

// Инициализация списка событий

 HEAD=0; // Стартовый переход имеет нулевой номер
 TEMP=0; // Отсчет времени от нуля

// Фиксация пассивности переходов

 for (i=1; i<N_tran; EL_next[i++]=N_tran);

// Выходные действия стартового перехода

 if (smdp_2[0]) smdp_2[0]();
```

```
// Обработка последовательности событий

 do {
  t=HEAD, HEAD=EL_next[t]; // Выборка очередного перехода
  TEMP=EL_time[t];   // Момент пассивизации
  if (t) EL_next[t]=N_tran; // Фиксация пассивности перехода t

// Обработка последствий пассивизации перехода t

  if (smdp_4[t]) smdp_4[t](); // Выходные действия
  for (i=L_tran[t], j=L_tran[t+1]; i<j; i++)
   for (k=M_tran[i], l=L_plac[k], m=L_plac[k+1]; l<m; l++) {

// Обработка последствий изменения разметки позиции k

    if (EL_next[n=M_plac[l]]!=N_tran)
     continue; // Отказ от активизации активного перехода
    if (smdp_1[n]()) { // Проверка условий активизации
     if (smdp_2[n]) smdp_2[n](); // Входные действия

// Планирование момента пассивизации перехода n

     f=EL_time[n]=TEMP+(smdp_3[n]? smdp_3[n]():0);
     for (p=0, q=HEAD; (q>0)&&(f>=EL_time[q]); p=q, q=EL_next[p]);
     EL_next[n]=q, EL_next[p]=n;
    }
   }
 } while (HEAD>0);

 printf("\n* Завершение моделирования в момент %ld\n",TEMP);
}
```

Результаты работы программы

```
МОДЕЛИРОВАНИЕ ПРОЦЕССОВ НА TPN
 Время 9, счетчик 1
 Время 18, счетчик 2
 ФИНИШ...
* Завершение моделирования в момент 18
```

Таким образом, полученные результаты полностью соответствуют результатам интерпретации представленной рассматриваемой системой продукций TPN [1].

ЗАКЛЮЧЕНИЕ

Говоря о программировании на языке С, следует предполагать знание не только собственно лингвистических конструкций, но и библиотек функций конкретной системы программирования. Язык С появился в годы широкого применения многопользовательского режима использования ЭВМ Понятие

терминала или консоли оператора при программировании в среде повсеместно распространенных в настоящее время операционных систем семейства Windows потеряло привлекательность для пользователей новых систем. Однако задачи системного программирования с непритязательным интерфейсом ввода-вывода, а также критичные по эффективности реализации модули программ могут программироваться на языке С.

Среди других алгоритмических языков программирования процедурного типа наиболее отличительным свойством языка С явилась поддержка адресной арифметики. Другое важное свойство - мобильность. Последнее позволило достичь успеха в создании крупных, развиваемых и долгоживущих программных систем. Логическим следствием эволюции языка С в направлении улучшения технологии программирования явилось создание языка объектно-ориентированного программирования С++.

Технология объектно-ориентированного программирования основана на конструктивном использовании принципов структуризации, модульности и абстракции. Ключевые понятия систем объектно-ориентированного прогаммирования:

– пакетирование – связывание в единое целое объектов существующих типов и функций доступа к ним с целью определения объектов нового типа;
– наследование – использование элементов данных и функций ранее определенных объектов для образования иерархии производных объектов;
– полиморфизм – возможность ассоциации некоторого имени с множеством уникальных для каждого уровня иерархии производных объектов понятий.

Объектно-ориентированное программирование на языке С++ - предмет изучения в следующем учебном курсе. Знание языка С и опыт его практического использования - непременное условие понимания языка С++, а также концепций построения современных систем программирования [1].

РАЗДЕЛ 2 ПРАКТИЧЕСКИЙ

ЛАБОРАТОРНЫЙ ПРАКТИКУМ ПО КУРСУ СИСТЕМНОЕ ПРОГРАММИРОВАНИЕ

ЛАБОРАТОРНАЯ РАБОТА №1

ОБЗОР ЯЗЫКА ПРОГРАММИРОВАНИЯ С

1. Цель работы
1.1. Ознакомление с языкот Си.
1.2. Приобретение практических навыков работы с Си
2. Основные сведения о синтаксисе языка С

Начальные сведения о синтаксисе любого языка программирования включают элементарные правила записи исходного текста программы - идентификация объектов программы, комментарии, формат исходного текста. Идентификаторы объектов программы на языке С могут включать:

– цифры 0...9;
– латинские прописные и строчные буквы A...Z, a...z;
– символ подчеркивания _.

Первый символ идентификатора не может быть цифрой. Длина идентификатора определяется реализацией транслятора С и редактора связей (компоновщика). Современная тенденция - снятие ограничений длины идентификатора.

Разделители идентификаторов объектов программы:

– пробелы;
– символы табуляции, перевода строки и страницы;
– комментарии (играют роль пробелов).

Комментарий - любая последовательность символов, начинающаяся парой символов /* и заканчивающаяся парой символов */. Формат записи исходного текста программ на языке С - свободный.

```
/* ПРИМЕР ПРОГРАММЫ ТЕСТА ГЕНЕРАТОРА СЛУЧАЙНЫХ ЧИСЕЛ */

#include
#include

#define RANGE 100

void main() {
 int i,j,k,s;

 /* Запрос объема выборки */
```

```
 while (printf("\n K-? "), !scanf(" %d",&k);

 /* Генерация случайных чисел */

 printf("\n Значения %d случайных чисел:\n",k);
 for (s=i=0; i < k; i++)
  s+=(j=random(RANGE)+1);
  printf("\n %3d) %d",i,j);
 }
 printf("\n\n Сумма %d, среднее %f",s,(float)s/k);
}
```

3. Структура программ на языке С

Программа на языке С включает операторы декларации объектов, преобразования объектов и управления вычислительным процессом. Программирование процесса преобразования объектов программы производится посредством записи выражений. Выражение включает один или несколько операндов и символов операций. Любое выражение, заканчивающееся символом ';', является оператором. Простейший вид операторов - операторы-выражения:

– оператор присваивания - выполнение операций присваивания;
– оператор вызова функции - операция вызова функции;
– пустой оператор.

Классы управляющих операторов в языке С:

– операторы условного и безусловного перехода;
– операторы организации циклов;
– оператор выбора альтернатив (переключатель);
– оператор выхода из функции.

Каждый из управляющих операторов имеет конкретную лексическую конструкцию, образуемую из ключевых слов языка С, выражений и символов-разделителей **'{','}',',',':','(',')'**.

Операторы языка С записываются в свободном формате с использованием разделителей между ключевыми словами. Допустима вложенность операторов. Любой оператор может помечаться меткой - идентификатором и символом **':'**. Область действия метки - функция, где эта метка определена. В случае необходимости можно использовать составной оператор (блок) – последовательность любых операторов, заключенная в фигурные скобки { и } (после закрывающей скобки символ **';'** не требуется). Элементарным модулем программы на языке С является функция. Любая программа должна содержать, как минимум, головную функцию со стандартным именем **main**. Пример исходного текста программы:

```
#include
```

```
/* ПРОГРАММА ПЕЧАТИ КОДОВ НАЖАТЫХ КЛАВИШ */

void main () {
 int i;

 while ((i=getch())!=27) {
 if (i==0) printf("\n* %d",getch());
  else printf("\n  %d",i);
 }
}
```

4. Операционные объекты в языке С

Классы операционных объектов программ на языке С:

– константы;
– простые переменные;
– массивы;
– структуры;
– объединения;
– указатели объектов;
– функции.

Объекты программы в общем случае имеют атрибуты:

– тип - характеристика механизма интерпретации данных;
– класс памяти - характеристика способа организации размещения объектов в памяти (статическая и динамическая память);
– область действия - характеристика области использования объектов функциями программы (локальные и глобальные объекты).

Единственная разновидность объектов, определяемая в исходном тексте непосредственно по месту использования - константы. Иногда используют термин "самоопределенные константы". Остальные объекты программы должны быть явно описаны в виде <атрибуты> <список_идентификаторов_объектов>; (элементы списка разделены запятыми, а атрибуты – разделителями). Предварительно отметим следующее:

– в языке С атрибуты типа описываются всегда явно;
– класс памяти можно не указывать, используя назначение по умолчанию;
– область действия объекта специальным ключевым словом не задается.

5. Понятие типа объекта в языке С

Тип объекта рекурсивно определяется на основе любого базового и производного типа посредством использования:

– символов модификации текущего типа;
– предписания размещения объектов известных типов в памяти.

Модификаторы текущего типа:

– символ * перед идентификатором - описание указателя на объект исходного типа;

– символы [и] после идентификатора объекта - описание массива объектов;
– символы (и) после идентификатора объекта - описание функции.

Отметим, что обязательное условие использования скобочных модификаторов любого вида - баланс открывающих и закрывающих скобок. Внутри скобок может размещаться детализирующая описание типа информация, например, размерность массива или список параметров. Допускается использование более одного модификатора типа с учетом следующих правил:

– чем ближе модификатор к идентификатору, тем выше его приоритет;
– модификаторы [] и () обладают приоритетом перед *;
– дополнительно вводимые круглые скобки позволяют увеличить приоритет объединяемых ими элементов описания.

Примеры описания объектов:

```
type a0;         /* Элемент типа type */
type a1[5];      /* Массив элементов типа type */
type *a2;        /* Указатель на элемент типа type */
type **a3;       /* Указатель на указатель элемента типа type */
type *a4[5];     /* Массив указателей на элементы типа type */
type (*a5)[10]; /* указатель на массив элементов типа type */
type *a6[3][4]; /* 3-элементный массив указателей
                   на 4-элементный массив элементов типа type */
type a7[5][2];   /* Массив массивов элементов типа type */
type a8();       /* Функция, вычисляющая значение типа type */
type *a9();      /* Функция, вычисляющая указатель
                   на элемент типа type */
type (*aa)();    /* Указатель на функцию, вычисляющую
                   значение типа type */
type *ab()[6];   /* Функция, вычисляющая указатель
                   на массив элементов типа type */
type *ac[4]();   /* Массив указателей на функцию, вычисляющую
                   значение типа type */
```

Здесь **type** – некоторый известный текущий тип объектов. Возможности определения размещения объектов известных типов в памяти в языке С представлены понятиями массивов, структур и объединений (см. 1.2.4-1.2.5).

6. Массивы в языке С

Массив - простейший вид структурного типа, представляющий поименованную совокупность последовательно размещаемых в памяти данных одного типа. Примеры описания массивов:

```
char s[20];        /* Одномерный массив */
int  x[10][20];    /* Двумерный массив  - массив массивов */
```

Особенность индексации элементов массива в языке С - нулевой индекс первого элемента по каждому измерению. Доступ к отдельному элементу массива выполняется после указания его целочисленного индекса двумя

способами: использование операции индексации - s[10], x[5][6]; косвенное обращение по указателю - *(s+10), *(*(x+5)+6).

7. Операции

7.1 Операции присваивания

Присваивание значения в языке С в отличие от традиционной интерпретации

```
идентификатор=выражение;
```

рассматривается как выражение, имеющее значение левого операнда после присваивания. Оператор присваивания, таким образом, может включать несколько операций присваивания и, как следствие, изменить значения нескольких операндов как побочный результат вычисления выражения:

```
int i,j,k;
float x,y,z;
char a,b;
...
x=i+(y=3)-(z=0);   <===>   z=0; y=3; x=i+y-z;
i=j=k=0;           <===>   k=0; j=k; i=j;
a=(i+b);           <===>   a=i+b; /* Результат - ? */
```

Очевидно, что в случае использования в выражении неодинаковых операций или разнотипных операндов возникают вопросы:

– порядок выполнения операций в выражении;
– последовательность вычисления операндов;
– корректность преобразования операндов.

Предварительные ответы:

– порядок выполнения операций можно определять круглыми скобками и(или) учитывать их приоритет, а в случае одинакового приоритета - направление выполнения каждой операции;
– последовательность вычисления операндов большинства операций стандартом языка С не регламентирована, а для коммутативных операций может оптимизироваться компилятором даже при наличии круглых скобок;
– корректность преобразования операндов гарантируется в рамках реальных аппаратных возможностей, подозрительные ситуации стремится обнаружить компилятор, а любые неопределенности устраняются явным использованием операции приведения типа.

Таким образом, конструирование выражений в языке С следует проводить с нетрадиционной для языков высокого уровня осторожностью. Из выделенных понятий только приоритет и направление выполнения операций стандартны в языке С, а остальные в общем случае являются системно зависимыми. Традиционная форма операции присваивания v = e (здесь v – операнд-переменная, e – выражение). В языке С допускается две разновидности

сокращений записи операции присваивания: 1) вместо записи v = v operator_ab e рекомендуется использовать запись v operator_ab = e (здесь operator_ab – арифметическая операция либо операция над битовым представлением операндов);

2) вместо записи ipv = ipv + 1 либо ipv = ipv – 1 рекомендуется использовать запись ipv++ либо ipv--, а также ++ipv либо –ipv (здесь символ '+' обозначает операцию инкремента, символ '-' операцию декремента, ipv - целочисленная переменная или переменная - указатель). Значением выражения ++x или --x является значение x после операции инкремента или декремента, а значением выражения x++ или x-- - значение x до операции (подробности операций над указателями см. ниже). Примеры использования сокращений:

```
int i,j,k;
float x,y;
...


x*=y;      <===>  x=x*y;
i+=2;      <===>  i=i+2;
x/=y+15;   <===>  x=x/(y+15);
k--;       <===>  k=k-1;
--k;       <===>  k=k-1;
j=i++;     <===>  j=i; i=i+1;
j=--i;     <===>  i=i-1; j=i;
```

Рекомендации использования сокращений обоснованы возможностью оптимизации программы. Например, схема выражения вида v operator_ab = e соответствует схеме выполнения многих машинных команд типа "регистр-память". Ограничения на целочисленность операндов операций вида ++ipv (--ipv) или ipv++ (ipv--) следуют из наличия машинных команд инкремента и декремента целочисленных операндов. Отметим, что левым операндом операции присваивания может быть только именованная либо косвенно адресуемая указателем переменная (в диагностических сообщения компилятора - LVALUE). Примеры недопустимых выражений: а) присваивание константе: 2=x+y getch()=i z=&y /* z - имя массива */ б) присваивание результату операции: (i+1)=(j=2+y); (float)i=1.012 Элементы массивов адресуются косвенно поэтому допустимы выражения

```
int area[];
area[i]=log2(i);
area[i]++;
area[i]*=125;
```

Особенность языка С: любые операции допускаются только со скалярными объектами, причем небольшого размера порядка размера регистров процессора.

Это объясняется ориентацией языка на задачи системного программирования. Любые действия с громоздкими объектами - массивами, строками (частный случай массива), структурами реализуются посредством операции вызова функций. Например, рассмотрим сложные объекты

```
int x[100], y[100];
char c[80];
struct {
int code;
float data[1000];
} s1, s2;
```

Примеры правильной реализации операций присваивания для таких объектов:
содержимое массива y присвоить содержимому массива x

```
memcpy(x,y,sizeof(x));
```

массиву c присвоить значение строки "0123456789"

```
strcpy(c,"0123456789");
```

содержимое структуры s2 присвоить содержимому структуры s1

```
memcpy(&s1,&s2,sizeof(s1));
```

Операции присваивания в стиле языка PL/1 или dBASE вида

```
x=y, c="0123456789", s1=s2
```

в языке C интерпретируются как присваивание значений указателей на объекты. Здесь в левой части - константы, поэтому приведенные операции обнаружит как ошибочные компилятор.

7.2 Арифметические операции

Перечень арифметических операций в языке C и их обозначений:

+ - сложение;
- - вычитание либо изменение знака;
/ - деление (при целочисленных операндах - с отбрасыванием остатка) ;
* - умножение;
% - остаток от деления целочисленных операндов со знаком первого операнда (деление по модулю).

Как и в других языках высокого уровня, допустимым являются унарные операции (+ -). Операндами арифметических операций могут быть любые арифметические выражения. Тип выражений при необходимости приводится к более масштабному для обеспечения правильности результата.

Обязательные преобразования даже однотипных операндов перед выполнением арифметических операции:
float --> double; char, short --> int.
Необязательные преобразования разнотипных операндов:
int --> unsigned -->long --> double.

Единственной исключительной ситуацией при выполнении арифметических операций считается деление на нуль, а другие виды ситуаций (переполнение, исчезновение порядка или потеря значимости) игнорируются. Операции (* / %) обладают приоритетом над операциями (+ -), поэтому при записи сложных выражений можно использовать общепринятые математические правила:
x+y*z-a/b <===> x+(y*z)-(a/b)

7.3 Операции отношений

Перечень операций отношений между объектами в языке С и их обозначения:
== - равно или эквивалентно;
!= - не равно;
< - меньше;
<= - меньше либо равно;
> - больше;
>= - больше либо равно.

Символы пар символов {==,!=,<=,>=} разделять нельзя.
Общий вид операций отношений
<выражение_1> <знак_операции> <выражение_2>

Операндами могут быть любые скалярные типы. Значения операндов после вычисления перед сравнением преобразуются к одному типу. Арифметические операнды преобразуются по правилам, аналогичным для арифметических операций. Операнды-указатели преобразуются в целые числа необходимого типа. Результат сравнения указателей будет корректным в арифметическом смысле лишь для объектов одного массива. Результат операции отношения - целое значение 1, если отношение истинно, или 0 в противном случае. Следовательно, операция отношения может использоваться в любых арифметических выражениях.
Примеры использования операций отношений:
y>0
x==y
x!=2
y=x*(z>2)+(d==1);

Рассмотрим примеры ошибочного использования операций отношений в синтаксически правильных выражениях:
Попытка конструирования выражений в математическом стиле записи:

0<x<="==> (0<x)<="==> 1 /* Тавтология */
Правильные варианты выражения:
0 (0<x)&&(x (0(символы && обозначают операцию конъюнкции – см. 1.3.5).
Попытка сравнения сложных объектов в различных областях памяти по их адресам:
char x[40]; x=="Фамилия" <===> 0
Отношения между нескалярными объектами приходится проверять посредством последовательного сравнения их элементов. Удобно для этих целей воспользоваться библиотечными функциями. Например, требуемое выше сравнение строк символов можно записать в виде
strcmp(x,"Фамилия")
(функция int strcmp(char *x,char *y) выполняет лексикографическое сравнение двух строк: нуль - совпадение строк, положительное число - признак "x>y", отрицательное число - признак "x
Сложные объекты можно рассматривать как массивы символов типа unsigned char. Сравнение массивов s1 и s2 символов такого типа длиной n байт выполняет функция
int memcmp(void *s1, void *s2, unsigned n),
(результат формируется подобно функции strcmp).

Например, рассмотрим сложные объекты
int x[100], y[100];
struct {
int code;
float data[1000];
} s1, s2;

Примеры правильной реализации операций сравнения для таких объектов: memcmp(x,y,sizeof(x)) - сравнение массивов x и y; memcmp(&s1,&s2,sizeof(s1)) – сравнение структур s1 и s2.

Упомянутые библиотечные функции могут правильно выявлять эквивалентность содержимого полей памяти. Использование их для установления отношений порядка для арифметических данных (напр., "больше", "меньше") приведет к ошибке, если игнорировать внутреннее представление данных. Например, рассмотрим два числа: 0x01020304l и 0x00000005l. Их представление в памяти ППЭВМ на основе микропроцессора 8086/808286 <04030201> и <05000000>. Функция memcmp при упрощенной интерпретации ее результата "установит", что первое число больше второго.</x)&&(x</x)</x

7.4 Логические операции

Перечень логических операций в языке С в порядке убывания относительного приоритета и их обозначения:

! - отрицание (логическое НЕТ);

&& - конъюнкция (логическое И);

|| - дизъюнкция (логическое ИЛИ).

Символы пар символов {||,&&} разделять нельзя.

Общий вид операции отрицания

!<выражение>

Общий вид операций конъюнкции и дизъюнкции

<выражение_1><знак_операции><выражение_2>

Операндами логической операции могут быть любые скалярные типы. Ненулевое значение операнда трактуется как "истина", а нулевое - "ложь". Результатом операции может быть 1 либо 0 целого типа.

!0 <===> 1 !5 <===> 0

int x,y; x=10; !((x=y)>0) <===> 0

Относительный приоритет логических операций позволяет пользоваться общепринятым математическим стилем запмси сложных логических выражений. Например, выражение

!x && (y>0) || (z==1) && (k>0)

эквивалентно следующему варианту выражения со скобками

((!x) && (y>0)) ||((z==1) && (k>0))

Особенность операций конъюнкции и дизъюнкции - экономное последовательное вычисление выражений-операндов:

если выражение_1 операции конъюнкции равно нулю, то результат операции - нуль, а выражение_2 не вычисляется;

если выражение_1 операции дизъюнкции не равно нулю, то результат операции - единица, а выражение_2 не вычисляется.

Таким образом, появляется возможность записью логического выражения задать условную последовательность вычисления выражений в направлении слева направо:

scanf("%d",&i)&&test1(i)&&test2(i) - нулевой результат одной из функций приведет к игнорированию вызова остальных;

search1(x)||search2(x)||search3(x) - только ненулевой результат одной из функций приведет к игнорированию вызова остальных; char *p=NULL; /* ... */ p && p[0]=='*' - проверить первый элемент массива, если указатель на массив не пустой.

8. Операторы

8.1 Условные операторы

В языке С имеется две разновидности условных операторов: простой и полный операторы условного выполнения.

Синтаксис простого оператора условного выполнения:

if (выражение) оператор

Элемент "оператор" подлежит выполнению, если "выражение" от лично от нуля.

Примеры записи:

```
if (x>0) x=0;
if (i!=1) j++, l=1; <===> if (i!=1) { j++, l=1; }
if (getch()!=27) {
k=0;
}

if (i) exit(1);  <===>  if (i!=0) exit(1);
if (i>0) if (i<="=="> if ((i>0)&&(i<="=="> i=0;
```

Синтаксис полного оператора условного выполнения:

if (выражение) оператор_1

else оператор_2

Если "выражение" отлично от нуля, то подлежит выполнению "оператор_1", иначе - "оператор_2".

Примеры записи:

```
if (x>0) j=k+1;
else m=i+10;
if (x) {
y=i++;
k=sfn(i);
} else {
printf("\n ???");
exit(0);
}
```

Элементы "оператор_1" и(или) "оператор_2" могут быть любым оператором, в том числе и условным оператором. Фраза "else ..." относится к непосредственно предшествующей ей фразе "if ...", поэтому для устранения коллизий условных операторов разных уровней вложенности необходимо заключать их в фигурные скобки:

```
if (!x) if (!y) printf("\n x=YES, y=YES");
else printf("\n x=NO"); /* x=0 <===> (x!=0)&&(y==0) ??? */
```

```
if (!x) {
if (!y) printf("\n x=YES, y=YES");
} else printf("\n x=NO");
```

8.2 Операторы цикла

Перечень разновидностей операторов цикла:

– оператор цикла с предусловием;
– оператор цикла с постусловием;
– оператор цикла с предусловием и коррекцией.

Синтаксис оператора цикла с предусловием:

while (выражение) оператор (элемент "оператор" может быть пустым оператором, оператором-выражением или операторным блоком). Смысл оператора: на каждой итерации цикла предварительно проверяется условие продолжения цикла - ненулевое значение "выражения", затем выполняется "оператор". Элемент "оператор" может включать любое количество управляющих операторов, связанных с конструкцией while:
continue - переход к следующей итерации цикла;
break - выход из цикла.

Примеры записи:

```
while (i>0) i<<=1, j++;
     while (printf("\n n-?"), scahf(" %d",&n));
        ...
     while (1) { /* Организация бесконечного цикла */
      /* ... */
      if (kbhit()&&(getch()==27)) break; /* Выход по ESC */
      /* ... */
     }
     while (!kbhit()); /* Активное ожидание нажатия клавиши */
```

Синтаксис оператора цикла с постусловием:

do оператор while (выражение);
(элемент "оператор" может быть пустым оператором, оператором выражением или операторным блоком). Смысл оператора: на каждой итерации цикла предварительно выполняется "оператор", затем проверяется условие продолжения цикла - ненулевое значение "выражения". Назначение используемых в элементе "оператор" управляющих операторов continue и break совпадает с ранее рассмотренным, но оператор continue вызывает переход к этапу оценки "выражения".

```
     do printf("\n ???");
      while (!scanf(" %d",&n));
      ...
```

```
float *x;
int i=0;
...
m=coreleft()/sizeof(*x);
do {
 printf("\n n-?");
 if (!scanf(" %d",&n)) {
  sound(300);
  printf(" Вводите цифры !");
  continue;
 }
if (n>m) continue;
 if (!scanf(" %f",x+i)) break;
} while (++i<="" div="">
```

Синтаксис оператора цикла с предусловием и коррекцией:
for (инициализация; условие_выполнения; коррекция) оператор
Здесь "инициализация", "условие_выполнения" и "коррекция" выражения, которые могут отсутствовать (пустые выражения), но символы ';' опускать нельзя. Оператор-выражение "инициализация" выполняется один раз перед началом итераций цикла. Итерации цикла продолжаются, пока истинно "условие_выполнения" - выражение пустое либо непустое выражение отлично от нуля. На каждой итерации последовательно выполняются "оператор" и оператор-выражение "коррекция". Назначение используемых в элементе "оператор" управляющих операторов continue и break совпадает с ранее рассмотренным.

```
float  x[10], y;
int  i,n;
...
for (i=n=sizeof(x)/sizeof(*x); i>0; x[--i]=0);
...
for (i=0; i<="" div="">
```

Различные формы операторов цикла могут выражаться друг через друга, например:

а) оператор
for (инициализация; условие_выполнения; коррекция) операторэквивалентен последовательности операторов инициализация;
while (условие_выполнения) {оператор; коррекция;}
(здесь "оператор" не может включать операторы break или continue);

б) оператор
for (; условие_выполнения;) оператор
эквивалентен оператору
while (условие_выполнения) оператор;

в) оператор
for (оператор; условие_выполнения;) оператор
эквивалентен оператору
do оператор while (условие_выполнения);

г) оператор
for (;;) оператор
эквивалентен оператору
while (1) оператор;

(вместо 1 здесь может быть любое число, отличное от нуля).

При использовании вложенных циклов следует учитывать, что управляющие операторы break и continue действуют внутри собственного цикла. Для выхода из вложенного цикла приходится использовать оператор безусловного перехода goto:

```
float x[10][20];
int i,j;

  /* Проверка наличия отрицательных значений */

     for (i=0; i<10; i++)
      for (j=0; j<20; j++)
       if (x[i][j]<0) goto next_step;

     next_step: printf("\nЭлемент (%d,%d) отрицателен...");
```

8.3 Оператор выбора альтернатив (переключатель) Синтаксис:
switch (выражение) { ----------
case константа_1: оператор_1 Тело
case константа_2: оператор_2 оператора
... switch
default: оператор_N
} ----------

Ключевое слово default и целочисленные значения констант рассматриваются как специальные метки операторов, область действия которых - тело оператора switch. Порядок следования таких меток не регламентирован. Целочисленное выражение сравнивается после вычис ления со значениями констант-меток. При совпадении с одной из них выполняется передача управления соответствующему оператору в теле оператора switch. В случае несовпадения значения выражения

с одной из констант - переход на метку default либо, при ее отсутствии, к оператору, следующему за оператором switch. Упомянутые здесь специальные метки в теле оператора switch недолжны повторяться или использоваться для ссылок в операторе goto. Для выхода из тела оператора switch используют управляющий оператор break (дополнительно можно воспользоваться операторами goto или return, а при вложенности в оператор цикла и оператором continue). Рассмотрим пример построения простейшего калькулятора для вычисления значений функций sin(),cos(),log(),sqrt(),tan().

```
#include
#include
#include <conio.h< char="" f[]="\n %s(%lf)=%lf" ;="" void=""
main()="" {="" double="" x;="" while="" (sound(1000),=""
printf("\n="" x-?="" "),="" nosound(),="" scanf("%lf",&x))="" for=""
(;;)="" x="%lf," sin,="" cos,="" log,="" sqrt,="" tan="" -="" ?=""
",x);="" switch(getch())="" case="" 27="" :="" goto="" cont;="" *=""
код="" клавиши="" esc="" 's':="" printf(f,"sin",x,sin(x));=""
break;="" 'c':="" printf(f,"cos",x,cos(x));="" 'l':=""
printf(f,"log",x,log(x));="" 'q':="" printf(f,"sqrt",x,sqrt(x));=""
't':="" printf(f,"tan",x,tan(x));="" default:="" sound(100);=""
select="" or="" esc");="" nosound();="" }="" cont:;="" <=""
div=""></conio.h<>
```

Очевидно, что здесь оператор switch можно заменить вложенными условными операторами:

```
int i;
...
i=getch();
if ((i=='s')||(i=='S')) printf(f,"sin",x,sin(x));
else if ((i=='c')||(i=='C')) printf(f,"cos",x,cos(x));
...
else if ((i=='t')||(i=='T')) printf(f,"tan",x,tan(x));
else {
 sound(100);
 printf ...
 nosound();
}
```

В последнем варианте используется последовательная проверка условий, его вычислительная сложность 6/2 (половина количества альтернатив). Оператор switch реализуется переходом по адресу, выбираемому из инвертированной таблицы меток по значению выражения, поэтому его вычислительная сложность близка к единице.

8.4 Операторы передачи управления

Формально к операторам передачи управления относятся:

а) оператор безусловного перехода

goto метка;

б) оператор перехода к следующему шагу (итерации) цикла
continue;
(игнорирование оставшихся операторов тела цикла);

в) выход из цикла либо оператора switch
break;
(передача управления следующему оператору);

г) оператор возврата их функции
return;
return выражение;
(прекращение выполнения текущей функции и возврат управления вызвавшей программе с передачей, при необходимости, значения выражения).

Операторы вида а...в рассматривались ранее. Оператор return обязательно необходим в функциях, возвращающих значения. В функциях, не возвращающих результат, он неявно присутствует после последнего оператора. Значение "выражения" при необходимости будет преобразовано к типу возвращаемого функцией значения.

```
void error(void *x, int n) {
 if (!x) printf("\nМассив не создан");
 if (!n) printf("\nМассив пустой");
}

float estim(float *x, int n) {
 int i;
 float y;
 if ((!x)||(!n) {
  error(x,n);
  return 0;
 }
 for (y=i=0; i<="" div="">
```

Строго говоря, в языке С имеются дополнительные возможности передачи управления:
– нелокальный переход, организуемый парой функций longjump и setjump;
– операторы структурного управления исключениями _try/_except и _try/finally [1].

ЛАБОРАТОРНАЯ РАБОТА №2

ФУНКЦИИ В ЯЗЫКЕ С

1. Цель работы
1.1 Знакомство с правилами организации функций в языке С
1.2 Получение навыков практического программирования с использованием собственных функций
2. Функции в языке С и структура программы

Функция является не только средством написания некоторой части программы, но и служит для оформления логически завершенного действия с собственным набором входных и выходных параметров. Термин «функция», принятый в С, имеет в других языках программирования родственные термины – процедура, модуль. Функция является основной программной единицей уже потому, что вся программа представляет собой множество вызывающих друг друга функций. Часть из них может быть получена «со стороны» – из библиотек или из программ, написанных в другое время, в другом месте, другими людьми и даже на другом языке программирования. То есть на уровне функций осуществляется «сборочный процесс» программы из отдельных составляющих.

2.1 Определение функции

Функция состоит из двух частей: *заголовка* создающего «интерфейс» функции к внешнему миру, и *тела функции* реализующего заложенный и нее алгоритм с использованием внутренних локальных данных. Вместе заголовок и тело составляют *определение функции*.

Интерфейс функции состоит из *имени функции, списка формальных параметров* (вход) и *типа результата* (выход). Формальные параметры – это собственные переменные функции, которым при ее вызове присваиваются значения фактических параметров. Список формальных параметров имеет синтаксис определений обычных переменных. Использование их в теле функции не отличается от использования обычных переменных.

Результат функции – это временная переменная, которая возвращается функцией и может быть использована как операнд в контексте (окружении) выражения, где был произведен ее вызов.

Поскольку все переменные в С имеют типы, тип результата также должен быть определен. Это делается в заголовке функции тем же способом, что и для обычных переменных. Используется тот же самый синтаксис, в котором имя функции выступает в роли переменной-результата:

```
int sum(...);   // Результат – целая переменная
char * FF(...); // Результат – указатель на символ
```

Значение переменной-результата устанавливается в операторе return, который производит это действие наряду с завершением выполнения функции и выходом из нее. Между ключевым словом return и ограничивающим символом «;» может стоять любое выражение, значение которого и становится результатом функции. Если необходимо, производиться преобразование типа, соответствующего выражению, к типу результата функции:

```
double FF() {
  int nn;          // Эквивалент
  return (nn+1);   // FF = (double)(nn + 1)
}
```

Имеется специальный пустой тип результата – void, который обозначает, что функция не возвращает никакого результата и, соответственно, не может быть вызвана внутри выражения. Оператор return в такой функции также не содержит никакого выражения:

```
void Nothing() {
  ...
  return;
}
```

Вызов функции выглядит как имя функции, за которым в скобках следует список *фактических параметров*.

Фактические параметры – переменные, константы или выражения, значения которых при вызове присваиваются соответствующим по списку формальным параметрам.

В **C** формальными параметрами и результатом функции могут быть только переменные, занимающие ограниченный объем памяти: базовые типы данных и указатели. Это сделано, исходя из общего положения о том, что транслятор не должен оказывать сильное влияние на эффективность программы путем включения каких-либо неявных операций копирования. Использование же массивов и структур в этом качестве приведет к появлению таких операций копирования.

Тело функции представляет собой уже известную нам синтаксическую конструкцию – блок. Это простая последовательность операторов, заключенная в фигурные скобки. После открывающейся скобки в блоке могут стоять определения переменных. Это *локальные переменные* функции. Они обладают следующими свойствами:

– локальные переменные создаются в момент входа в блок (тело функции) и уничтожаются при выходе из него;

– локальные переменные могут использоваться только в том блоке, в котором они определены. Это значит, что за пределами блока они «не видны»;

– инициализация локальных переменных заменяется присваиванием им значений во время их создания при входе в блок. Поскольку под инициализацией понимается процесс установки начальных значений переменных в процессе трансляции (которые затем попадают в программный код), то для локальных переменных это сделать принципиально невозможно.

Локальные переменные в теле функции обозначаются в С термином *автоматические*.

2.2 Способ передачи параметров

В **С** принят способ передачи параметров, который называется *передачей по значению*. Выглядит он так:

– формальные параметры являются собственными переменными функции;

– при вызове функции происходит присваивание значений фактических параметров формальным (копирование первых во вторые);

– при изменении формальных параметров значения соответствующих им фактических параметров не меняются.

Единственным исключением из этого правила является передача имени массива в качестве параметра. В этом случае формальный параметр также является собственной переменной, но не массивом, а указателем на него. Поэтому размерность такого массива в функции несущественна и может отсутствовать, а изменение элементов массива – формального параметра приводит к изменению значений массива – фактического параметра функции:

```
int sum(int s[], int n) { // сумма элементов массива
  int z = 0;              // размерность передается
  int i;                  // отдельным параметром
  for (i=0; i < n; i++) z += s[i];
  return(z);
}
int c[10] = {1,6,4,7,3,56,43,7,55,33};
void main() {
  int nn;
  nn = sum(c, 10);
}
```

Коль скоро формальные параметры функции являются псевдопеременными, которые при вызове содержат копии фактических параметров, на них распространяются все соглашения о типах данных и переменных. В частности, в заголовке функции используется стандартный контекстный способ определения типа переменной:

```
void f(char **p, int A[], void (*f)())
{...}
```

Исторически сложилось, что первоначальный синтаксис определения функции принципиально исключал возможность контроля транслятора за соответствием количества и типов формальных и фактических параметров. С одной стороны, это позволяло использовать механизм вызова функций и передачи параметров нестандартными способами, а, с другой стороны, являлось причиной многочисленных, трудно обнаруживаемых ошибок.

В связи с этим был введен новый синтаксис определения, а также объявления функции, называемый прототипом. Если вызывается функция, определенная или объявленная по прототипу, то транслятор проверяет соответствие формальных и фактических параметров и, по возможности, выполняет неявные преобразования типов.

2.3 Функция main

Функция **main**, с которой начинается выполнение С-программы, может быть определена с параметрами, которые передаются из внешнего окружения, например, из командной строки. Во внешнем окружении действуют свои правила представления данных, а точнее, все данные представляются в виде строк символов. Для передачи этих строк в функцию **main** используются два параметра, первый параметр служит для передачи числа передаваемых строк, второй для передачи самих строк. Общепринятые (но не обязательные) имена этих параметров <>bargc и argv. Параметр **argc** имеет тип **int**, его значение формируется из анализа командной строки и равно количеству слов в командной строке, включая и имя вызываемой программы (под словом понимается любой текст, не содержащий символа пробел). Параметр **argv** это массив указателей на строки, каждая из которых содержит одно слово из командной строки. Если слово должно содержать символ пробел, то при записи его в командную строку оно должно быть заключено в кавычки. Функция **main** может иметь третий параметр, который принято называть **argp**, и который служит для передачи в функцию **main** параметров операционной системы (среды) в которой выполняется С-программа. Заголовок функции main может иметь следующий вид:

```
int main (int argc, char *argv[], char *argp[])
```

Если, например, командная строка С-программы имеет вид:

```
A:\>cprog working 'C program' 1
```

то аргументы **argc, argv, argp** могут представляться в памяти как показано в схеме

```
argc = 4
argv [0] = “A:\\cprog.exe\0”
     [1] = “working\0”
     [2] = “C program\0”
```

```
      [3] = "1\0"
      [4] = NULL
argp [0] = "path=A:\\;C:\\\0"
      [1] = "lib=D:\\LIB\0"
      [2] = "include=D:\\INCLUDE\0"
      [3] = "conspec=C:\\COMMAND.COM\"
      [4] = NULL
```

Операционная система поддерживает передачу значений для параметров **argc, argv, argp**, а на пользователе лежит ответственность за передачу и использование фактических аргументов функции **main**.

Следующий пример представляет программу печати фактических аргументов, передаваемых в функцию **main** из операционной системы и параметров операционной системы. Пример:

```
int main(int argc, char *argv[], char *argp[]) {
  int i = 0;
  printf("\n Имя программы %s", argv[0]);
  for(i = 1; i <= argc; i++)
  printf("\n аргумент %d равен %s", argv[i]);
  printf("\n Параметры операционной системы:");
  while(*argp) {
    printf("\n %s", *argp);
    argp++;
  }
  return 0;
}
```

Доступ к параметрам операционной системы можно также получить при помощи библиотечной функции getenv, ее прототип имеет следующий вид:

```
char * geteuv(const char *varname);
```

Аргумент этой функции задает имя параметра среды, указатель на значение которой выдаст функция **getenv**. Если указанный параметр не определен в среде в данный момент, то возвращаемое значение NULL.

Используя указатель, полученный функцией **getenv**, можно только прочитать значение параметра операционной системы, но нельзя его изменить. Для изменения значения параметра системы предназначена функция **putenv**.

2.4 Глобальные (внешние) переменные. Инициализация

Программа в целом представляет собой просто набор функций с обязательной функцией **main**, имеющих каждая собственный набор локальных переменных. Но кроме этого в ее состав включаются еще переменные, доступные сразу нескольким функциям. Такие переменные называются *глобальными* (в С – *внешними*). Будучи определенными в любом месте программы вне тела функции, они становятся доступными любой функции, следующей за их объявлением по тексту программы:

```
-int B[10];
¦int sum()
¦{...B[i]... }
¦ -int n;
¦ ¦
¦ ¦void nf()
¦ ¦{...B[i]...n...}
¦ ¦
¦ ¦ -char c[80];
¦ ¦ ¦ void main()
¦ ¦ ¦ {...B[i]...n...c[k]...}
L-+-+---------- Области действия переменных B,n,c
```

Глобальные (внешние) переменные являются наиболее «стабильными» данными в программе. Транслятор переводит их во внутреннее представление, в котором им соответствуют определенные адреса выделенной памяти. Можно сказать, что эти переменные находятся в программе (программном файле) еще до загрузки ее в память. Поэтому их можно инициализировать.

Инициализация – присваивание переменным во время трансляции начальных значений, которые сохраняются во внутреннем представлении программы и устанавливаются при загрузке программы в память перед началом ее работы.

Инициализация включается в синтаксис определения переменной:

```
int a = 5, B[10] = {1,5,4,2}, C[] = { 0,1,2,3,4,5 };
```

Инициализатор отделяется от переменной в ее определении знаком «=». Для простой переменной – это обычная константа, для массива – список констант, заключенных в фигурные скобки и разделенных запятыми. Заметим, что размерность массива может отсутствовать, если транслятор в состоянии определить ее из инициализирующего списка.

2.5 Области действия функций. Определения и объявления

До сих пор ничего не говорилось ни о взаимном расположении в программе определения функции и ее вызова, ни о соответствии формальных и фактических параметров, ни о контроле такого соответствия. Конечно, нельзя считать, что транслятор «знает» о всех функциях, когда-либо написанных, либо находящихся в библиотеках, текстовых файлах и т.д. Каждая программа должна сама сообщать транслятору необходимую информацию о функциях, которые она собирается вызывать, а именно:

– имя функции;
– тип результата;
– список формальных параметров (переменные и типы).

При ее наличии транслятор может корректно сформировать вызов функции, даже если ее текст (определение) отсутствует в программе.

Вся перечисленная информация о функции находится в ее заголовке. Таким образом, достаточно этот заголовок привести отдельно, и проблема корректного вызова решается. Такой заголовок называется *объявлением функции* или в рассматриваемом варианте синтаксиса *прототипом*.

Объявление функции – заголовок функции (без тела функции), необходимый транслятору для формирования корректного вызова функции, если она по каким-либо причинам ему недоступна. Причины такого «незнания» транслятора следующие. Во-первых, трансляторы обычно используют довольно простые алгоритмы просмотра текста программы, «не заглядывая» вперед. Поэтому обычно на данный момент трансляции содержание текста программы за текущим транслируемым оператором ему неизвестно. Во-вторых, функция может быть в библиотеке. В-третьих – в другом текстовом файле, содержащем часть С-программы. Во всех этих случаях необходимо использовать объявления. Единственный случай, когда этого делать не надо – когда определение функции присутствует ранее по тексту программы:

```
intB[10];
int sum(int s[],int n); // Объявление функции,
// определенной далее по тексту
// Объявление библиотечной функции
// с переменным числом параметров
extern int printf(char *,...);
// Объявление функции без
// параметров из другого файла программы
extern int other(void);

void main() {
  sum(B,10);// Вызовы объявленных функций
  printf("%d",B[i]);
  other();
}

int sum(int s[], int n) {
  ...
}
```

Из примера видно, что объявление функции практически дублирует заголовок, отличаясь в некоторых деталях:

– объявление заканчивается символом «;»;

– если функция находится вне текущего файла, то объявление предваряется служебным словом **extern**;

– имена переменных в списке формальных параметров объявления могут отсутствовать;

– если функция не имеет формальных параметров, то в объявлении присутствует формальный параметр типа **void**.

Имея предварительно определенную функцию или ее объявление (прототип), транслятор в состоянии проверить соответствие формальных и фактических параметров функции как по их количеству, так и по типам. При этом транслятор может выполнить неявные преобразования типов фактических параметров к типам формальных, если это потребуется:

```
extern double sin(double); int x; double y; y = sin(x); // неявное
преобразование (double)x
```

2.6 Вызов функции

Фактические параметры записываются в стек перед вызовом функции, начиная с последнего в списке. Поскольку аппаратный стек расположен «вверх дном» и «растет» от старших адресов к младшим, то этим обеспечивается прямой порядок размещения их в памяти. Формальные параметры представляют собой «ожидаемые» смещения в стеке, по которым должны после вызова находиться соответствующие фактические параметры. Таким образом, сам механизм вызова функции соответствие параметров устанавливает только «по договоренности» между вызывающей и вызываемой функциями, а транслятор при использовании прототипа проверяет эти соглашения [1].

ЛАБОРАТОРНАЯ РАБОТА №3

АДРЕСНАЯ АРИФМЕТИКА И УПРАВЛЕНИЕ ПАМЯТЬЮ

1. Цель работы
1.1 Знакомство с указателями в языке Си.
1.2 Ознакомления с функциями работы с памятью языка Си и Windows API.
1.3 Приобретение практических навыков управления памятью в программах.
2. Адресная арифметика, управление памятью
2.1 Общие положения

При работе с памятью редко (может быть, и никогда) придется использовать что-нибудь кроме функций из стандартной библиотеки языка С. Причина, по которой рекомендуется использовать библиотечные функции С (**malloc, realloc, calloc, free** и т. д.), состоит в том, что они просты и понятны. Но самое главное заключается в том, что не возникнет никаких проблем при использовании этих функций в программах, написанных для Windows. Хотя использовать библиотечные функции языка С удобно, возможно, в принципе, написание программы для Windows вообще без использования этих функций.

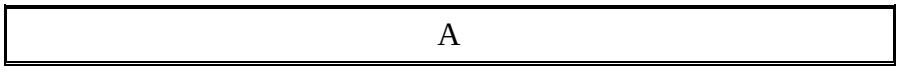
Каждая библиотечная функция, которая требует обращения к операционной системе (в частности функции управления памятью) имеет соответствующую, и, как правило, более развитую и гибкую функцию операционной системы. Под управлением памятью имеются в виду возможности программы по размещению и манипулированию данными. Поскольку единственным «представителем» памяти в программе выступают переменные, то управление памятью определяется тем, каким образом работает с ними и с образованными ими структурами данных язык программирования. Большинство языков программирования однозначно закрепляет за переменными их типы данных и ограничивает работу с памятью только теми областями, в которых эти переменные размещены. Программист не может выйти за пределы самим же определенного шаблона структуры данных. С другой стороны, это позволяет транслятору обнаруживать допущенные ошибки как в процессе трансляции, так и в процессе выполнения программы.
В языке Си ситуация принципиально иная по двум причинам. Во-первых, наличие операции адресной арифметики при работе с указателями позволяет, в принципе, выйти за пределы памяти, выделенной транслятором под переменную и адресовать память как «до» так и «после» нее. Другое дело, что это должно производиться *осознанно и корректно*. Во-вторых, присваивание и преобразование указателей различных типов, речь о котором пойдет ниже, позволяет рассматривать одну и ту же память «под различным углом зрения» в смысле типов заполняющих ее переменных.

2.2 Присваивание указателей различного типа

Операцию присваивания указателей различных типов следует понимать как назначение указателя в левой части на ту же самую область памяти, на которую назначен указатель в правой. Оба указателя после присваивания содержат один и тот же адрес. Но поскольку тип переменных у них может быть разный, то эта область памяти по правилам интерпретации указателя будет рассматриваться как заполненная переменными либо одного, либо другого типа:

```
char *pc, A[20];
int  *pi;
long *pl;
pc = A;    pi = pc; pl = pc;
```

В этом примере *pc* – указатель на область байтов, *pi* – на область целых, *pl* – на область длинных целых. Соответственно операции адресной арифметики **(pc+i), *(pi+i), *(pl+i)* адресуют *i*-ый байт, *i*-ое целое и *i*-ое длинное целое от начала области (рис. 2.1):



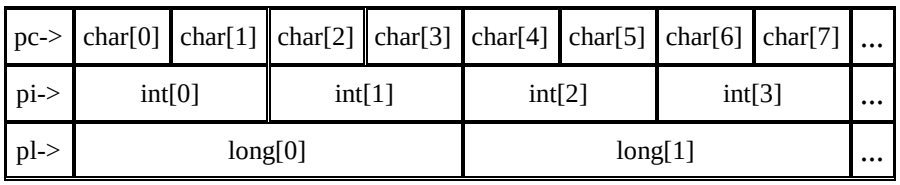


Рисунок 2.1

```
*(pc + 2) = 5;// записать 5 во 2-й байт области A
*(pi + 1) = 7;// записать 7 в 1-е слово области A
*(pl + 0) = 9;// записать 9 в 0-е двойное слово области A
```

Таким образом, область памяти имеет различную структуру (байты, слова и т.д.) в зависимости от того, через какой указатель к ней обращаться. При этом неважно, что сама область определена как массив типа **char** – это имеет отношение только к операциям использующим идентификатор массива. Присваивание значения указателя одного типа указателю другого типа сопровождается действием, которое называется *преобразованием типа указателя*. На самом деле это действие не приводит к каким-либо преобразованиям данных (команды транслятором не генерируются). Транслятор просто запоминает, что тип переменной изменился и операции адресной арифметики и косвенного обращения нужно выполнять с учетом нового типа указателя. При присваивании происходит автоматическое неявное преобразование типа указателя, которое в транслятор обычно сопровождает соответствующим предупреждением.

Таким образом, операция присваивания указателя включает в себя:

– присваивание адреса от правого указателя к левому;
– неявное преобразование типа указателя от правого к левому.

2.3 Явное преобразование типа указателя

Рассмотрим три операции присваивания:

```
char *pc, A[20];
int  *pi;
pc = A;
pi = pc;
*(pi + 2) = 5;
*((int*)pc + 2) = 5;
```

Все они дают одинаковый результат – записывают целое 5 во второе слово области памяти, определенной как массив байтов (символов) с именем A. Однако, если в первом случае используется промежуточный указатель типа **int***, то в последнем случае такой указатель создается как рабочая переменная, которая получает тип **int*** и значение переменной pc. Такая операция называется явным преобразованием типа указателя. В скобках указывается абстрактный тип данных – указатель на требуемый тип (например, **int***).

2.4 Роль операции sizeof в управлении памятью

Операция **sizeof** вызывает подстановку транслятором соответствующего значения размерности указанного в ней типа данных в байтах. С этой точки зрения она является универсальным измерителем, который должен использоваться для корректного размещения данных различных типов в памяти. Сказанное можно продемонстрировать на простом примере размещения переменных типа **double** в массиве типа **char**:

```
#define N 40
double *d;
char   A[N];

for (int i = 0, d = A; i < N / sizeof(double); i++)
       d[i] = (double)i;
```

Следует заметить, что использование операции **sizeof** позволяет сделать программу переносимой, то есть нечувствительной к разрядности представления данных в различных трансляторах.

2.5 Указатель типа void*

Наличие указателя определенного типа предполагает известную организацию памяти, на которую он ссылается. Но в некоторых случаях фрагмент программы «не должен знать» или просто не имеет достаточной информации о структуре данных в этой области. Тогда указатель должен пониматься как адрес памяти как таковой, с неопределенной организацией и неизвестной размерностью указываемых данных. Такой указатель можно присваивать, передавать в качестве параметра и результата функции, но операции косвенного обращения и адресной арифметики с ним недопустимы.

Именно такими свойствами обладает указатель типа **void*** – указатель на пустой тип **void** (или другими словами *указатель неопределенный тип данных*). Наличие его в данном месте программы говорит о том, что она не имеет достаточных оснований для работы с адресуемой областью памяти. Наиболее часто тип **void*** является формальным параметром или результатом функции.

Приведем несколько примеров:

```
extern void * malloc(int);
int *p;
p = malloc(sizeof(int)*20);
...
p[i] = i;
```

Функция **malloc** резервирует память в динамической области памяти и возвращает ее адрес в виде указателя **void***. Это означает, что функцией выделяется память как таковая, безотносительно к размещаемым в ней переменным. Вызывающая функция неявно преобразует тип указателя **void*** в

требуемый тип **int*** для работы с этой областью как с массивом целых переменных.

```
extern int fread(void *, int, int, FILE *);
int A[20];
...
fread(A, sizeof(int), 20, fd);
```

Функция **fread** выполняет чтение из двоичного файла *n* записей длиной по *m* байтов, при этом структура записи для функции неизвестна. Поэтому начальный адрес области памяти передается формальным параметром типа **void***. При подстановке фактического параметра A типа int* производится неявное преобразование его к типу **void***.

Как видно из примеров, преобразование типа указателя **void*** к любому другому типу указателя соответствует «смене точки зрения» программы на адресуемую память от «данных вообще» к «конкретным данным» и наоборот.

2.6 Работа с областью памяти переменного формата

Иногда требуется организовать данные в памяти таким образом, что заранее неизвестна точная последовательность переменных различных типов – она определяется некоторым форматом. В этом случае требуется просматривать последовательно область памяти, извлекая из нее переменные разных типов. Такая задача может быть решена с использованием нескольких указателей различного типа, которые сохраняют одинаковое значение (адрес) путем взаимного присваивания. Заметим, что операция **p++* применительно к любому указателю интерпретируется как «взять переменную, на которую указывает *p*, и перейти к следующей», следовательно, значением указателя после выполнения операции будет адрес переменной, следующей за выбранной:

```
char *pc, A[100], c;
int  *pi, i;
long *pl  l;
pl = pi = pc = A;     // назначить все указатели на
                      // общий начальный адрес A
i = *pi++;       // взять целое по указателю
pc = pl = pi;         // выровнять значения всех указателей
l = *pl++;       // взять длинное целое по указателю
pc = pi = pl;         // выровнять значения всех указателей
c = *pc++;       // взять байт (символ) по указателю
pi = pl = pc;         // выровнять значения всех указателей
```

Более простой вариант заключается в использовании объединения (**union**), которое позволяет использовать общую память для размещения своих элементов. Если элементами **union** являются указатели, то операции присваивания можно исключить (более подробно объединения рассматриваются в следующих работах):

```
union PTR
{
      char *pc;
      int  *pi;
      long *pl;
} ptr;

i = *(ptr.pi)++; l = *(ptr.pl)++; c = *(ptr.pc)++;
```

2.7 Указатели и многомерные массивы

Для двумерных и многомерных массивов в **C** существуют особенные взаимоотношения с указателями. Для их понимания необходимо напомнить те соглашения, которые заложены в языке для многомерных массивов:

– двумерный массив всегда реализован как одномерный массив с количеством элементов, соответствующих первому индексу, причем каждый элемент представляет собой массив элементов базового типа с количеством, соответствующим второму индексу.

Например

```
char A[20][80];
```

– определяет массив из 20 массивов по 80 символов в каждом и никак иначе. Массив, таким образом, располагается в памяти по строкам;

– идентификатор массива без скобок интерпретируется как адрес нулевого элемента нулевой строки, или указатель на базовый тип данных. В нашем примере идентификатору A будет соответствовать выражение *&A[0][0]* с типом **char***;

– по аналогии имя двумерного массива с единственным индексом интерпретируется как начальный адрес соответствующего внутреннего одномерного массива. *A[i]* понимается как *&A[i][0]*, то есть начальный адрес *i*-го массива символов.

Сказанное справедливо и для N-мерных массивов: многомерный массив представляет собой массив элементов первого индекса, каждый из который представляет собой массив элементов второго индекса и т.д.

```
char A[20][80];
for (int i = 0; i < 20; i++)
{
      //A[i] – указатель на i-ю строку
      //в двумерном массиве символов
      gets(A[i]);     // ввод строки с клавиатуры
}
```

Обычный указатель работает с линейной последовательностью элементов. В этом случае при присваивании начального адреса двумерного массива

обычному указателю его двумерная структура «теряется». В следующем примере указатель используется для работы с двумерным массивом с учетом его реального размещения в памяти:

```
char *pc, A[20][80];
pc = A;
//обращение к элементам массива по указателю на базовый тип
... *(pc + i*80 + j) ...  // или
... pc[i*80 + j] ...
```

Для работы с многомерными массивами вводятся особые указатели – указатели на массивы. Они представляют собой обычные указатели, адресуемым элементом которых является не базовый тип, а массив элементов этого типа:

```
char (*p)[][80];
char (*q)[5][80];
```

Круглые скобки имеют здесь принципиальное значение. В контексте определения p является переменной, при косвенном обращении к которой получается двумерный массив символов, то есть p является указателем на двумерный массив. При отсутствии скобок имел бы место двумерный массив указателей на строки. Кроме того, если не используются операции вида «p++» или «p += n», связанные с размерностью указуемого массива, то наличия первого индекса не требуется.

```
p = q = A; // назначить p и q на двумерный массив
(*p)[i][j] // j-ый символ в i-ой строке
                // в массиве по указателю p
p += 2;         // переместить p на 2 массива
                // по 5 строк по 80 символов
```

2.8 Библиотечные функции C

Вы можете определить в программе указатель (например, на массив целых чисел) следующим образом :

```
int *p; // указатель не инициализирован
```

Можно выделить блок памяти, на который будет указывать p, следующим образом :

```
p = (int *)malloc(1024);
```

При этом выделяется блок памяти размером 1024 байта, который может хранить 512 16-разрядных целых. Указатель, равный *NULL*, показывает, что выделение памяти не было успешным. Можно также выделить такой блок памяти, используя следующий вызов:

```
p = (int *)calloc(512, sizeof(int));
```

Два параметра функции **calloc** перемножаются и в результате получается 1024 байта. Кроме того, функция **calloc** производит заполнение блока памяти нулями.

Если необходимо увеличить размер блока памяти (например, удвоить его), то можно вызвать функцию:

```
p = (int *)realloc(p, 2048);
```

Указатель является параметром функции, и указатель (возможно, отличающийся по значению от первого, особенно, если блок увеличивается) является возвращаемым значением функции.

После того как работа с памятью, выделенной с помощью библиотечных функций, будет завершена ее целесообразно освободить для дальнейшего использования (например другими функциями). Это можно выполнить с помощью функции **free**, которой необходимо передать адрес освобождаемой памяти. Например:

```
free(p);
```

2.9 Динамическое размещение массивов средствами языка С

При динамическом распределении памяти для массивов следует описать соответствующий указатель и присваивать ему значение при помощи функции **calloc**. Одномерный массив *a[10]* из элементов типа **float** можно создать следующим образом

```
float * a;
a = (float*)calloc(10, sizeof(float));
```

Для создания двумерного массива вначале нужно распределить память для массива указателей на одномерные массивы, а затем распределять память для одномерных массивов. Пусть, например, требуется создать массив *a[n][m]*, это можно сделать при помощи следующего фрагмента программы:

```
void main()
{
      double **a;
      int n, m, i;
      scanf("%d %d", &n, &m);
      a = (double **)calloc(m, sizeof(double *));
      for (i = 0; i <= m; i++)
            a[i] = (double *)calloc(n, sizeof(double));
}
```

Аналогичным образом можно распределить память и для трехмерного массива размером NxMxL. Следует только помнить, что ненужную для дальнейшего выполнения программы память следует освобождать при помощи функции **free**.

```
void main ()
{
      long ***a;
      int n, m, l, i, j;
      scanf("%d %d %d", &n, &m, &l);
/* --------  распределение памяти -------- */
      a = (long ***)calloc(m, sizeof(long **));
      for (i = 0; i <= m; i++)
      {
            a[i] = (long **)calloc(n, sizeof(long *));
            for (j = 0; j <= l; j++)
                  a[i][j] = (long *)calloc(l, sizeof(long));
      }
      ...
/* --------- освобождение памяти ----------*/
      for (i = 0; i <= m; i++)
      {
      for (j = 0; j <= l; j++) free(a[i][j]);
      free(a[i]);
      }
      free(a);
}
```

Следующий пример интересен тем, что память для массивов распределяется в вызываемой функции, а используется в вызывающей. В таком случае в вызываемую функцию требуется передавать указатели, которым будут присвоены адреса выделяемой для массивов памяти. Пример:

```
int vvod(double ***, long **);
void main()
{
      double **a;      /*  указатель для массива  a[n][m] */
      long *b;         /*  указатель для массива  b[n]    */
      vvod(&a, &b);
      /* в функцию vvod передаются адреса */
      /* указателей, а не их значения */
      ...
}
int vvod(double ***a, long **b)
{
      int n, m, i, j;
      scanf("%d %d", &n, &m);
      *a = (double **)calloc(n, sizeof(double *));
      *b = (long *)calloc(n, sizeof(long));
      for (i = 0; i <= n; i++)
            *a[i] = (double *)calloc(m, sizeof(double));
      ...
}
```

Указатель на массив не обязательно должен показывать на начальный элемент некоторого массива. Он может быть сдвинут так, что начальный элемент

будет иметь индекс отличный от нуля, причем он может быть, как положительным, так и отрицательным.

Пример:

```
void main()
{
     float *q, **b;
     int i, j, k, n, m;
     scanf("%d %d", &n, &m);
     q = (float *)calloc(m, sizeof(float));
/* сейчас указатель q показывает */
/* на начало массива */
     q[0] = 22.3;
     q -= 5;
/* теперь начальный элемент массива имеет */
/* индекс 5,а конечный элемент индекс n-5 */
     q[5] = 1.5;
/* сдвиг индекса не приводит к перераспределению */
/* массива в памяти и изменится начальный элемент*/
     q[6] = 2.5;     /*  -  это второй элемент     */
     q[7] = 3.5;     /*  -  это третий элемент     */
     q+=5;
/* теперь начальный элемент вновь имеет индекс 0,*/
/* а значения элементов q[0], q[1], q[2] равны   */
/* соответственно 1.5, 2.5, 3.5              */
     q+=2;
/* теперь начальный элемент имеет индекс -2,*/
/* следующий -1, затем 0 и т.д. по порядку*/
     q[-2]=8.2;
     q[-1]=4.5;
     q-=2;
/* возвращаем начальную индексацию, три первых*/
/* элемента массива q[0], q[1], q[2], имеют*/
/* значения 8.2, 4.5, 3.5                  */
     q--;
/* вновь изменим индексацию.             */
/* Для освобождения области памяти в которой */
/* размещен  массив q используется функция    */
/* free(q), но поскольку значение указателя  */
/*  *q смещено, то выполнение функции* free(q)*/
/*  приведет к непредсказуемым последствиям. */
/* Для правильного выполнения этой функции    */
/* указатель q должен быть возвращен          */
/* в первоначальное положение                 */
     free(++q);
/* Рассмотрим возможность изменения индексации и*/
/* освобождения памяти для двумерного массива*/
     b=(float **)calloc(m,sizeof(float *));
     for (i=0; i < m; i++)
     b[i]=(float *)calloc(n,sizeof(float));
/* После распределения памяти начальным  */
/* элементом массива будет элемент b[0][0]  */
/* Выполним сдвиг индексов так, чтобы начальным */
/* элементом стал элемент b[1][1]          */
```

```
     for (i=0; i < m ; i++) --b[i];
     b--;
/* Теперь присвоим каждому элементу массива */
/* сумму его индексов                       */
     for (i=1; i<=m; i++)
          for (j=1; j<=n; j++)
               b[i][j]=(float)(i+j);
/* Обратите внимание на начальные значения */
/* счетчиков циклов i и j, он начинаются с 1 */
/* а не с 0 возвратимся к прежней индексации */
     for (i=1; i<=m; i++) ++b[i];
     b++;
/* Выполним освобождение памяти          */
     for (i=0; i < m; i++) free(b[i]);
     free(b);
     ...
}
```

3. Управление памятью средствами Win32 API

3.1 Функции для работы с памятью

Как уже говорилось ранее, все, что можно делать с помощью библиотечных функций С, можно делать самостоятельно, или используя вызовы функций ядра Windows. Ниже приведен пример вызова функции Windows для выделения блока памяти для указателя на целые:

```
DWORD dwSize = 1024;
UINT uiFlags = 0;
p = (int *)GlobalAlloc(uiFlags, dwSize);
```

За исключением одной, для каждой функции, начинающейся со слова **Global**, существует другая, начинающаяся со слова **Local**. Эти два набора функций в Windows идентичны. Два различных слова сохранены для совместимости с предыдущими версиями Windows, где функции Global возвращали дальние указатели, а функции **Local** – ближние. Если первый параметр задать нулевым, то это эквивалентно использованию флага **GMEM_FIXED**. Такой вызов функции **GlobalAlloc** эквивалентен вызову функции **malloc**.

Следующий пример демонстрирует использование функции изменения размера блока памяти:

```
p = (int *)GlobalReAlloc(p, dwSize, uiFlags);
```

Для определения размера выделенного блока памяти используется функция **GlobalSize**:

```
dwSize = GlobalSize(p);
```

Функция освобождения памяти:

```
GlobalFree(p);
```

3.2 Перемещаемая память

Функция **GlobalAlloc** поддерживает флаг **GMEM_MOVEABLE** и комбинированный флаг для дополнительного обнуления блока памяти (описано в заголовочных файлах Windows):

```
#define GHND (GMEM_MOVEABLE | GMEM_ZEROINIT)
```

Флаг **GMEM_MOVEABLE** позволяет перемещать блок памяти в виртуальной памяти, при этом функция возвращает не адрес выделенного блока, а 32-разрядный описатель (дескриптор) блока памяти. Это необязательно означает, что блок памяти будет перемещен в физической памяти, но адрес, которым пользуется программа для чтения и записи, может измениться. Для фиксации блока используется вызов:

```
p =(int *)GlobalLock(hGlobal);
```

Эта функция преобразует описатель памяти в указатель. Пока блок зафиксирован, Windows не изменяет его виртуальный адрес. Когда работа с блоком заканчивается, для снятия фиксации вызывается функция:

```
GlobalUnlock(hGlobal);
```

Этот вызов дает Windows свободу перемещать блок в виртуальной памяти. Для того чтобы правильно осуществлять этот процесс следует фиксировать и снимать фиксацию блока памяти в ходе обработки одного сообщения. Когда нужно освободить перемещаемую память, надо вызывать функцию **GlobalFree** с описателем, но не с указателем на блок памяти. Если в данный момент нет доступа к описателю, то необходимо использовать функцию:

```
hGlobal = GlobalHandle(p);
```

Для преднамеренного удаления блока памяти можно использовать следующий вызов:

```
GlobalDiscard(hGlobal);
```

3.3 Другие функции и флаги

Другим доступным для использования в функции **GlobalAlloc** является флаг **GMEM_SHARE** или **GMEM_DDESHARE** (идентичны). Как следует из его имени, этот флаг предназначен для динамического обмена данными. Функции **GlobalAlloc** и **GlobalReAlloc** могут также включать флаги **GMEM_NODISCARD** и **GMEM_NOCOMPACT**. Эти флаги дают указание Windows не удалять и не перемещать блоки памяти для удовлетворения запросов памяти.

Функция **GlobalFlags** возвращает комбинацию флагов **GMEM_DISCARDABLE, GMEM_DISCARDED** и **GMEM_SHARE**. Наконец, вы можете вызвать функцию **GlobalMemoryStatus** (для этой функции нет

функции – двойника со словом **Local**) с указателем на структуру типа **MEMORYSTATUS** для определения количества физической и виртуальной памяти, доступной приложению.

Windows также поддерживает некоторые функции, реализованные программистом или дублирующие библиотечные функции С. Это функции **FreeMemory** (заполнение конкретным байтом), **ZeroMemory** (обнуление памяти), **CopyMemory** и **MoveMemory** – обе копируют данные из одной области памяти в другую. Если эти области перекрываются, то функция **CopyMemory** может работать некорректно. Вместо нее используйте функцию **MoveMemory**.

3.4 Функции управления виртуальной памятью

Windows поддерживает ряд функций, начинающихся со слова **Virtual**. Эти функции предоставляют значительно больше возможностей управления памятью. Однако, только очень необычные приложения требуют использования этих функций.

Последняя группа функций работы с памятью – это функции, имена которых начинаются со слова **Heap**. Эти функции создают и поддерживают непрерывный блок виртуальной памяти, из которого можно выделять память более мелкими блоками. Следует начинать с вызова функции **HeapCreate**. Затем, использовать функции **HeapAllocate, HeapReAllocate** и **HeapFree** для выделения и освобождения блоков памяти в рамках «кучи» [1].

ЛАБОРАТОРНАЯ РАБОТА №4

ОБРАБОТКА СТРУКТУРИРОВАННЫХ ДАННЫХ

1. Цель работы

1.1 Ознакомление со стандартными структурированными типами данных языка С

1.2 Приобретение практических навыков организации и работы с нестандартными типами данных

2. Обработка встроенных структурированных типов данных

2.1 Массивы

В программе на языке Си можно использовать структурированные типы данных. К ним будем относить массивы, структуры и объединения. Массив состоит из многих элементов одного и того же типа. Ко всему массиву целиком можно обращаться по имени. Кроме того, можно выбирать любой элемент массива. Для этого необходимо задать индекс, который указывает его относительную позицию. Число элементов массива назначается при его

определении и в дальнейшем не меняется. Если массив объявлен, то к любому его элементу можно обратиться следующим образом: указать имя массива и индекс элемента в квадратных скобках (индексация всегда начинается с нуля). Массивы определяются так же, как и переменные:

```
int a[100];
char b[20];
float d[50];
```

В первой строке объявлен массив *a* из 100 элементов целого типа. Во второй строке элементы массива *b* имеют тип **char**, а в третьей – **float**. Двумерный массив представляется как одномерный, элементы которого тоже массивы. Например, определение

```
char a[10][20];
```

задает такой массив. По аналогии можно установить и большее число измерений. Элементы двумерного массива хранятся по строкам, т.е. если проходить по ним в порядке их расположения в памяти, то быстрее всего изменяется самый правый индекс. Например, обращение к девятому элементу пятой строки запишется так: *a[5][9]*.

Пусть задан массив:

```
int a[2][3];
```

Тогда элементы массива a будут размещаться в памяти следующим образом:

```
a[0][0], a[0][1], a[0][2], a[1][0], a[1][1], a[1][2]
```

Имя массива – это константа, которая содержит адрес его первого элемента (в данном примере a содержит адрес *a[0][0]*). Предположим, что *a* = 1000. Тогда адрес элемента *a[0][1]* будет равен 1002 (элемент типа **int** занимает в памяти 2 байта), адрес следующего элемента *a[0][2]* – 1004 и т.д. Если выбрать элемент, для которого не выделена память, может возникнуть ошибка времени выполнения. Т.к компилятор не следит за этим, программа будет откомпилирована, но работать будет неверно.

В языке С существует сильная взаимосвязь между указателями и массивами. Любое действие, которое достигается индексированием массива, можно выполнить и с помощью указателей, причем последний вариант будет быстрее.

Определение int *a[5]*; задает массив из пяти элементов a[0], a[1], a[2], a[3], a[4]. Если объект *y* определен как

```
int *y;
```

то оператор *y = &a[0]*; присваивает переменной *y* адрес элемента *a[0]*. Если переменная y указывает на очередной элемент массива a, то *y+1* указывает на

следующий элемент, причем здесь выполняется соответствующее масштабирование для приращения адреса с учетом длины объекта (для типа **int** – байт, **long** – 4 байта, (**double** – 8 байт и т.д.).

Так как само имя массива есть адрес его нулевого элемента, то оператор *y = &a[0]*; можно записать и в другом виде: *y = a*. Тогда элемент *a[1]* можно представить как **(a+1)*. С другой стороны, если *y* – указатель на массив *a*, то следующие две записи: *a[i]* и **(y+i)* эквивалентны.

Между именем массива и соответствующим указателем есть одно важное различие. Указатель – это переменная и *y = a*; или *y++*; – допустимые операции. Имя же массива – константа, поэтому конструкции вида *a = y; a++*; использовать нельзя, так как значение константы постоянно и не может быть изменено.

Переменные с адресами могут образовывать некоторую иерархическую структуру (могут быть многоуровневыми) типа указатель на указатель (т.е. он содержит адрес другого указателя), указатель на указатель на указатель и т.д. Если указатели адресуют элементы одного массива, то их можно сравнивать (отношения вида, «==», «!=» и другие работают правильно). В то же время нельзя сравнивать либо использовать в арифметических операциях указатели на разные массивы (соответствующие выражения не приводят к ошибкам при компиляции, но в большинстве случаев не имеют смысла). Любой адрес можно проверить на равенство или неравенство со значением NULL. Указатели на элементы одного массива можно также вычитать. Тогда результатом будет число элементов массива, расположенных между уменьшаемым и вычитаемым объектами. Язык Си позволяет инициализировать массив при его определении. Для этого используется следующая форма:

```
тип имя_массива[...] = {список значений};
```

Примеры:

```
int a[5] = {0, 1, 2, 3, 4};
char ch[3] = {'d', 'e', '9'};
int b[2][3] = {{1, 2, 3}, {4, 5, 6}};
```

В языке допускаются массивы указателей, которые определяются, например, следующим образом:

```
char *m[5];
```

Здесь *m[5]* – массив, содержащий адреса элементов типа **char**.

2.2 Строки символов

Язык С не поддерживает отдельный строковый тип данных, но он позволяет определить строки двумя различными способами. В первом используется массив символов, а во втором – указатель на первый символ массива.

Определение *char a[10]*; указывает компилятору на необходимость резервирования места для максимум 10 символов. Константа *a* содержит адрес ячейки памяти, в которой помещено значение первого из десяти объектов типа **char**. Процедуры, связанные с занесением конкретной строки в массив *a*, копируют ее по одному символу в область памяти, на которую указывает константа *a*, до тех пор, пока не будет скопирован нулевой символ, оканчивающий строку. Когда выполняется функция типа *printf("%s", a)* ей передается значение *a*, т.е. адрес первого символа, на который указывает *a*. Если первый символ нулевой, то работа функции *printf()* заканчивается, а если нет, то она выводит его на экран, прибавляет к адресу единицу и снова начинает проверку на нулевой символ. Такая обработка позволяет снять ограничения на длину строки (конечно, в пределах объявленной размерности): строка может быть любой длины, до тех пор, пока есть место в памяти, куда ее можно поместить.

Инициализировать строку при таком способе определения можно следующим образом:

```
char array[7] = "Строка";
char s[] = {'С', 'т', 'р', 'о', 'к', 'а', '\0'};
```

(при определении массива с одновременной инициализацией пределы изменения индекса можно не указывать).
Второй способ определения строки – это использование указателя на символ. Определение

```
char * b;
```

задает переменную *b*, которая может содержать адрес некоторого объекта. Однако в данном случае компилятор не резервирует место для хранения символов и не инициализирует переменную *b* конкретным значением. Когда компилятор встречает оператор вида

```
b = «IBM PC»;
```

он производит следующие действия. Во-первых, как и в предыдущем случае, он создает в каком-либо месте объектного модуля строку «IBM PC», за которой следует нулевой символ ('\0'). Во-вторых, он присваивает значение начального адреса этой строки (адрес символа 'I') переменной *b*. Функция *printf("%s", b)* работает так же, как и в предыдущем случае, осуществляя вывод символов до тех пор, пока не встретится заключительный нуль.

Массив указателей можно инициализировать, т.е. назначать его элементам конкретные адреса некоторых заданных строк при определении.

Для ввода и вывода строк символов помимо *scanf()* и *printf()* могут использоваться функции *gets()* и *puts()* (их прототипы находятся в файле *stdio.h*).

Если *string* – массив символов, то ввести строку с клавиатуры можно так:

```
gets(string);
```

(ввод оканчивается нажатием клавиши). Вывести строку на экран можно следующим образом:

```
puts(string);
```

Для работы со строками существует специальная библиотека функций, прототипы которых находятся в файле *string.h*.

Наиболее часто используются функции *strcpy(), strcat(), strlen()* и *strcmp()*. Если *string1* и *string2* – массивы символов, то вызов функции *strcpy()* имеет вид:

```
strcpy(string1, string2);
```

Эта функция служит для копирования содержимого строки *string2* в строку *string1*. Массив *string1* должен быть достаточно большим, чтобы в него поместилась строка *string2*. Так как компилятор не отслеживает эту ситуацию, то недостаток места приведет к потере данных.

Вызов функции strcat() имеет вид:

```
strcat(string1, string2);
```

Эта функция присоединяет строку *string2* к строке *string1* и помещает ее в массив, где находилась строка *string1*, при этом строка *string2* не изменяется. Нулевой байт, который завершал первую строку, заменяется первым байтом второй строки.

Функция *strlen()* возвращает длину строки, при этом завершающий нулевой байт не учитывается. Если *a* – целое, то вызов функции имеет вид:

```
a = strlen(string);
```

Функция *strcmp()* сравнивает две строки и возвращает 0, если они равны.

2.3 Структуры

Структурированная переменная (или просто структура) играет в языке программирования роль, противоположную массиву. Так, если массив представляет из себя упорядоченное множество переменных одного типа, последовательно размещенных в памяти, то структура -аналогичное множество, состоящее из переменных разных типов. Синтаксис определения структурированных переменных в **C** имеет следующий вид

```
struct man // имя структуры
{
      char name[10]; // элементы структуры
      int dd, mm, yy;
      char * address;
}
// Определение структурированных переменных
man A, B, X[10];
```

Составляющие структуру переменные имеют различные типы и имена, по которым они идентифицируются в структуре. Их называют *элементами структуры*, и они имеют синтаксис определения обычных переменных. Использоваться где-либо еще, кроме как в составе структурированной переменной, они не могут. В данном примере структура состоит из массива 10 символов *name*, целых переменных *dd, mm* и *yy* и указателя на строку *address*. После определения элементов структуры следует список структурированных переменных. Каждая из них имеет внутреннюю организацию описанной структуры, то есть полный набор перечисленных элементов. Имя структурированной переменной идентифицирует всю структуру в целом. В данном случае имеются переменные *A, B* и массив *X* из 10 структурированных переменных:
Другое важное свойство структуры это наличие у нее имени. Имя структуры идентифицирует данную последовательность элементов, поэтому в программе в дальнейшем можно определять новые структурированные переменные, не раскрывая содержания уже определенной структуры:

```
man C, D, *p;
```

В этом примере видно, что можно определять, как сами структурированные переменные, так и указатели на них. Аналогичным образом формальные параметры функции и ее результат также могут быть указателями на структурированные переменные:

```
man *create() {... }
void f(man *q) {... }
```

После определения структуры ее имя приобретает самостоятельное значение и используется в синтаксисе языка аналогично таким ключевым словам как **int, long** и т.д. Имя структуры при этом является не базовым типом данных, а производным.

При определении глобальной (внешней) структурированной переменной или массива таких переменных они могут быть инициализированы списками значений элементов, заключенных в фигурные скобки и перечисленных через запятую. Например:

```
man A = {"Петров", 1, 10, 1969, "Морская-12"};
man X[10] = {{"Смирнов", 12, 12, 1977, "Дачная-13"},
             {"Иванов", 21, 03, 1945, "Северная-21"}};
```

Способ работы со структурированной переменной вытекает из ее аналогии с массивом. Точно так же, как нельзя выполнить операцию над всем массивом, но можно над отдельным его элементом, структуру можно обрабатывать,

выделяя отдельные ее элементы. Для этой цели существует операция «.» (точка), аналогичная операции «[]» в массиве. В структурированной переменной она выделяет элемент с заданным именем:

```
A.name... // элемент name структурированной переменной A
B.dd... // элемент dd структурированной переменной B
```

Если элемент структуры является не простой переменной, а массивом или указателем, то для него применимы соответствующие ему операции ([], * и адресная арифметика):

```
A.name[i]...  // i-й элемент массива name, который является
              // эл-том структурированной переменной A
*B.address... // косвенное обращение по указателю address,
              // который является элементом структурированной
              // переменной B
```

Структура играет особую роль среди всех других способов представления данных. Элементы структуры связаны между собой не только физически (общая память), но и логически, поскольку обычно представляют собой характеристики и свойства одной сущности или предмета, состояние которого отображается в программе. Иначе говоря, структурированная переменная соответствует в программе понятию *объекта*.

То, что указатели на структурированные переменные имеют широкое распространение, подтверждается наличием в **C** специальной операции «->» (стрелка, минус-больше), которая понимается как выделение элемента в структурированной переменной, адресуемой указателем. То есть операндами здесь являются указатель на структуру и элемент структуры. Операция имеет полный аналог в виде сочетания операций «*» и «.»:

```
struct man *p,A;
p = &A;
p->mm // эквивалентно (*p).mm
```

2.4 Объединения

Объединение представляет собой структурированную переменную с несколько иным способом размещения элементов в памяти. Если в структуре (как и в массиве) элементы расположены последовательно друг за другом, то в объединении «параллельно». То есть для их размещения выделяется одна общая память, в которой они перекрывают друг друга и имеют в ней общий адрес. Размерность ее определяется максимальной размерностью элемента объединения. Синтаксис объединения полностью совпадает с синтаксисом структуры, только ключевое слово **struct** заменяется на **union**:

```
union Dword
{
      long ll;
      int ii[2];
      char cc[4];
      int xx;
} DW;
```

Назначение объединения заключается не в экономии памяти, как может показаться на первый взгляд. На самом деле оно является одним из инструментов управления памятью на принципах, принятых в С. У объединений есть одно важное свойство: если записать в один элемент объединения некоторое значение, то через другой элемент это же значение можно прочитать уже в другой форме представления (как переменную другого типа). То есть форму представления данных в памяти можно менять совершенно свободно:

```
char z;
DW.ll = 0x12345678;
z = DW.cc[2]; // Второй байт в массиве байтов cc в DW имеет
              // значение 0x34. Результат: z получает
              // значение второго байта длинного целого
```

Естественно, что при таком манипулировании внутренним представлением данных, необходимо знать их форматы и размерность [1].

ЛАБОРАТОРНАЯ РАБОТА №5

РАБОТА С ФАЙЛАМИ

1. Цель работы

1.1 Ознакомление со стандартными функциями работы с файлами языка Си.

1.2 Приобретение практических навыков работы с файлами в ОС Windows

2. Работа с файлами, синхронный ввод/вывод

2.1 Общие понятия, определение файла

Файлом называют способ хранения информации на физическом устройстве. Файл – это понятие, которое применимо ко всему – от файла на диске до терминала.

В языке Си отсутствуют операторы для работы с файлами. Все необходимые действия выполняются с помощью функций, включенных в стандартную библиотеку. Они позволяют работать с различными устройствами, такими, как диски, принтер, коммуникационные каналы и т.д. Эти устройства

сильно отличаются друг от друга. Однако файловая система преобразует их в единое абстрактное логическое устройство, называемое потоком.

В Си существует два типа потоков: текстовые (text) и двоичные (binary).

Текстовый поток – это последовательность символов. При передаче символов из потока на экран, часть из них не выводится (например, символ возврата каретки, перевода строки).

Двоичный поток – это последовательность байтов, которые однозначно соответствуют тому, что находится на внешнем устройстве.

Прежде чем читать или записывать информацию в файл, он должен быть открыт и тем самым связан с потоком. Это можно сделать с помощью библиотечной функции **fopen**. Она берет внешнее представление файла (например, «c:\my_prog.txt») и связывает его с внутренним логическим именем, которое используется далее в программе.

Логическое имя – это указатель на требуемый файл. Его необходимо определить; делается это, например, так:

```
FILE * fp;
```

Здесь *FILE* – имя типа, описанное в стандартном заголовочном файле , *fp* – указатель на файл. Обращение к функции *fopen()* в программе осуществляется выражением:

```
fp = fopen(спецификация_файла, "способ_использования_файла")
```

Спецификация файла (т.е. имя файла и путь к нему) может, например, иметь вид: *“C:\\my_prog.txt”* – для файла *my_prog.txt* на диске C:.

Способ использования файла задается специальными символами (или их комбинацией) ‘r’, ‘w’, ‘a’ и т.д. Если в результате обращения к функции *fopen()* возникает ошибка, то она возвращает указатель на константу *NULL*.

Рекомендуется использовать следующий способ открытия файла:

```
if ((fp = fopen("c:\\my_prog.txt", "rt")) == NULL)
{
     puts("Открыть файл не удалось\n");
     exit(1);
}
```

После окончания работы с файлом он должен быть закрыт. Это делается с помощью библиотечной функции *fclose()*. Она имеет следующий прототип:

```
int fclose(FILE *fp);
```

При успешном завершении операции функция *fclose()* возвращает значение нуль. Любое другое значение говорит об ошибке.

2.2 Функции работы с файлами библиотек языка Си

Функция *putc()* записывает символ в файл и имеет следующий прототип:

```
int putc(int c, FILE *fp);
```

Здесь *fp* – указатель на файл, возвращенный функцией *fopen()*, c – символ для записи (переменная c имеет тип **int**, но используется только младший байт). При успешном завершении *putc()* возвращает записанный символ, в противном случае возвращается константа *EOF*. Она определена в файле и имеет значение -1.

Функция *getc()* читает символ из файла и имеет следующий прототип:

```
int getc(FILE *fp);
```

Здесь *fp* – указатель на файл, возвращенный функцией *fopen()*. Эта функция возвращает прочитанный символ. Соответствующее значение имеет **int**, но старший байт равен нулю. Если достигнут конец файла, то getc() возвращает значение *EOF*.

Функция *feof()* определяет конец файла при чтении двоичных данных и имеет следующий прототип:

```
int feof(FILE *fp);
```

Здесь *fp* – указатель на файл, возвращенный функцией *fopen()*. При достижении конца файла возвращается ненулевое значение, в противном случае возвращается 0.

Функция *fputs()* записывает строку символов в файл. Она отличается от функции *puts()* только тем, что в качестве второго параметра должен быть записан указатель на переменную файлового типа.

Например:

```
fputs("Example", fp);
```

При возникновении ошибки возвращается значение *EOF*.

Функция *fgets()* читает строку символов из файла. Она отличается от функции *gets()* тем, что в качестве второго параметра должно быть записано максимальное число вводимых символов плюс единица, а в качестве третьего – указатель на переменную файлового типа. Строка считывается целиком, если ее длина не превышает указанного числа символов, в противном случае функция возвращает только заданное число символов.

Рассмотрим пример:

```
fgets(string, n, fp);
```

Функция возвращает указатель на строку *string* при успешном завершении и константу *NULL* в случае ошибки либо достижения конца файла.

Функция *fprintf()* выполняет те же действия, что и функция *printf()*, но работает с файлом. Ее отличием является то, что в качестве первого параметра задается указатель на переменную файлового типа.

Например:

```
fprintf(fp, "%x", a);
```

Функция *fscanf()* выполняет те же действия, что и функция *scanf()*, но работает с файлом. Ее отличием является то, что в качестве первого параметра задается указатель на переменную файлового типа.

Например:

```
fscanf(fp, "%x", &a);
```

При достижении конца файла возвращается значение *EOF*.

Функция *fseek()* позволяет выполнять чтение и запись с произвольным доступом и имеет следующий прототип:

```
int fseek(FILE *fp, long count, int access);
```

Здесь *fp* – указатель на файл, возвращенный функцией *fopen()*, *count* – номер байта относительно заданной начальной позиции, начиная с которого будет выполняться операция, *access* – способ задания начальной позиции. Переменная *access* может принимать следующие значения:

0 – начальная позиция задана в начале файла;

1 – начальная позиция считается текущей;

2 – начальная позиция задана в конце файла.

При успешном завершении возвращается нуль, при ошибке – ненулевое значение.

Функция *ferror()* позволяет проверить правильность выполнения последней операции при работе с файлами. Имеет следующий прототип:

```
int ferror(FILE *fp);
```

В случае ошибки возвращается ненулевое значение, в противном случае возвращается нуль.

Функция *remove()* удаляет файл и имеет следующий прототип:

```
int remove(char *file_name);
```

Здесь *file_name* – указатель на строку со спецификацией файла. При успешном завершении возвращается нуль, в противном случае возвращается ненулевое значение.

Функция *rewind()* устанавливает указатель текущей позиции в начало файла и имеет следующий прототип:

```
void rewind(FILE *fp);
```

Функция *fread()* предназначена для чтения блоков данных из потока. Имеет прототип:

```
unsigned fread(void *ptr,unsigned size,unsigned n,FILE *fp);
```

Она читает *n* элементов данных, длиной *size* байт каждый, из заданного входного потока *fp* в блок, на который указывает указатель *ptr*. Общее число

прочитанных байтов равно произведению n*size. При успешном завершении функция *fread()* возвращает число прочитанных элементов данных, при ошибке – 0.

Функция *fwrite()* предназначена для записи в файл блоков данных. Имеет прототип:

```
unsigned fwrite(void *ptr,unsigned size,unsigned n,FILE *fp);
```

Она добавляет *n* элементов данных, длиной *size* байт каждый, в заданный выходной файл *fp*. Данные записываются с позиции, на которую указывает указатель *ptr*. При успешном завершении операции функция *fwrite()* возвращает число записанных элементов данных, при ошибке – неверное число элементов данных.

В языке С открываются пять стандартных файлов со следующими логическими именами:

stdin – для ввода данных из стандартного входного потока (по умолчанию с клавиатуры);

stdout – для вывода данных в стандартный выходной поток (по умолчанию на экран дисплея);

stderr – файл для вывода сообщений об ошибках (всегда связан с экраном дисплея);

stdprn – для вывода данных на принтер;

stdaux – для ввода и вывода данных в коммуникационный канал.

В языке С имеется также система низкоуровневого ввода/вывода (без буферизации и форматирования данных), соответствующая стандарту системы UNIX. Прототипы составляющих ее функций находятся в файле.

2.3 Функции Windows для работы с файлами

2.3.1 Обзор файлового ввода/вывода

Обычно, если необходимо открыть файл, используется один из стандартных вызовов библиотеки С (такой как, например, *fopen()*). В большинстве языков программирования предусмотрены достаточно удобные высокоуровневые средства работы с файлами. Однако в некоторых ситуациях требуется открыть файл и работать с ним на уровне операционной системы, не используя высокоуровневые функции. Например, прямое обращение к операционной системе может потребоваться в случае, если не намерены использовать асинхронный ввод/вывод. Системный вызов, с помощью которого осуществляется открытие файла, называется *CreateFile()*. На самом деле название этого вызова (в переводе на русский *CreateFile* – «создать файл») плохо отражает функции, которые он выполняет. В зависимости от флагов, которые передаются этому вызову, он может либо действительно создать новый файл, либо открыть уже существующий. Если файл с указанным именем уже

существует на жестком диске, вызов *CreateFile()* может либо открыть его, либо уничтожить его и создать новый файл с таким же именем. Если файла с указанным именем на жестком диске еще нет, вызов *CreateFile()* создает такой файл. В любом случае вызов *CreateFile()* создает дескриптор файла и возвращает его вызвавшей программе, которая может использовать этот дескриптор для дальнейшей работы с файлом.

2.3.2 Функции открытия/закрытия файлов

2.3.2.1 Функция создания файла (CreateFile)

Функция *CreateFile()* предназначена не только для создания нового файла, но и открытия существующего файла или каталога, а также изменения длины существующего файла. Кроме этого эта функция может выполнять операции над каналами передачи данных (*pipe*), дисковыми устройствами и консолями. Прототип функции *CreateFile()* следующий:

```
HANDLE CreateFile (
  LPCTSTR lpFileName,           // адрес строки имени файла
  DWORD dwDesiredAccess,   // режим доступа
  DWORD dwShareMode,       // режим совместного использования
                                // файла
  LPSECURITY_ATTRIBUTES
  lpSecurityAttributes,         // дескриптор защиты
  DWORD dwCreationDistribution, // параметры создания
  DWORD  dwFlagsAndAttributes,  // атрибуты файла
  HANDLE hTemplateFile
);
```

Через параметр *lpFileName* передается адрес строки, содержащей имя файла, который необходимо создать или открыть. Строка должна быть заканчиваться нулем. Параметр *dwDesiredAccess* определяет тип доступа, который должен быть предоставлен к открываемому файлу. Здесь можно использовать логическую комбинацию следующих констант:

0 – доступ запрещен, однако приложение может определять атрибуты файла или устройства, открываемого при помощи функции *CreateFile()*;

GENERIC_READ – разрешен доступ на чтение;

GENERIC_WRITE – разрешен доступ на запись.

С помощью параметра *dwShareMode* задаются режимы совместного использования открываемого или создаваемого файла. Для этого параметра может быть использована комбинация следующих констант:

0 – совместное использование файла запрещено;

FILE_SHARE_READ – другие приложения могут открывать файл с помощью функции *CreateFile()* для чтения;

FILE_SHARE_WRITE – аналогично предыдущему, но на запись.

Через параметр *lpSecurityAttributes* необходимо передать указатель на дескриптор защиты или значение *NULL*, если этот дескриптор не используется. В данных примерах не используется дескриптор защиты. Параметр *dwCreationDistribution* определяет действия, выполняемые функцией *CreateFile()*, если приложение пытается создать файл, который уже существует.

Параметр *dwFlagsAndAttributes* задает атрибуты и флаги для файла.

В дополнение к перечисленным выше атрибутам, через параметр *dwFlagsAndAttributes* вы можете передать любую логическую комбинацию флагов.

И, наконец, последний параметр *hTemplateFile* предназначен для доступа к файлу шаблона с расширенными атрибутами для создаваемого файла. Этот параметр здесь не рассматривается.

В случае успешного завершения функция *CreateFile()* возвращает идентификатор созданного или открытого файла (или каталога). При ошибке возвращается значение *INVALID_HANDLE_VALUE* (а не *NULL*, как можно было бы предположить). Код ошибки можно определить при помощи функции *GetLastError()*.

В том случае, если файл уже существует и были указаны константы *CREATE_ALWAYS* или *OPEN_ALWAYS*, функция *CreateFile()* не возвращает код ошибки. В то же время в этой ситуации функция *GetLastError()* возвращает значение *ERROR_ALREADY_EXISTS*.

*2.3.2.2 Функ*ция CloseHandle

Функция *CloseHandle()* позволяет закрыть файл. Она имеет единственный параметр – идентификатор закрываемого файла. Заметим, что если указать функции *CreateFile()* флаг *FILE_FLAG_DELETE_ON_CLOSE*, то сразу после закрытия файл будет удален. Как ранее говорилось, такая методика очень удобна при работе с временными файлами.

2.4 Роль операции sizeof в управлении памятью

Операция **sizeof** вызывает подстановку транслятором соответствующего значения размерности указанного в ней типа данных в байтах. С этой точки зрения она является универсальным измерителем, который должен использоваться для корректного размещения данных различных типов в памяти. Сказанное можно продемонстрировать на простом примере размещения переменных типа **double** в массиве типа **char**:

2.4.1 Функции работы с файлами

2.4.1.1 Функции чтения/записи файлов (ReadFile и WriteFile)

С помощью функций *ReadFile()* и *WriteFile()* приложение может выполнять, соответственно, чтение из файла и запись в файл. Прототипы функций *ReadFile()* и *WriteFile()* следующие:

```
BOOL ReadFile(
  HANDLE  hFile,                // идентификатор файла
  LPVOID  lpBuffer,             // адрес буфера для данных
  DWORD nNumberOfBytesToRead,   //количество байт,
                                // которые необходимо прочесть
                                // в буфер
  LPDWORD lpNumberOfBytesRead,  // адрес слова,
                                // в которое будет записано
                                // количество прочитанных байт
  LPOVERLAPPED lpOverlapped);   // адресс структуры типа OVERLAPPED


BOOL WriteFile (
  HANDLE  hFile,                // идентификатор файла
  LPVOID  lpBuffer,             // адрес записываемого блока данных
  DWORD   nNumberOfBytesToWrite,     // количество, байт
                                  // которые необходимо
                                  // записать
  LPDWORD lpNumberOfBytesWrite, // адрес слова, в котором
                                  // будет сохранено
                                  // количество записанных байт
  LPOVERLAPPED lpOverlapped);   // адрес структуры OVERLAPPED
```

Через параметр *hFile* этим функциям необходимо передать идентификатор файла, полученный от функции *CreateFile()*.

Параметр *lpBuffer* должен содержать адрес буфера, в котором будут сохранены прочитанные данные (для функции *ReadFile()*), или из которого будет выполняться запись данных (для функции *WriteFile()*).

Параметр *nNumberOfBytesToRead* используется для функции *ReadFile()* и задает количество байт данных, которые должны быть прочитаны в буфер *lpBuffer*. Аналогично, параметр *nNumberOfBytesToWrite* задает функции *WriteFile()* размер блока данных, имеющего адрес *lpBuffer*, который должен быть записан в файл.

Так как в процессе чтения возможно возникновение ошибки или достижение конца файла, количество прочитанных или записанный байт может отличаться от значений, заданных, соответственно, параметрами *nNumberOfBytesToRead* и *nNumberOfBytesToWrite*. Функции *ReadFile()* и *WriteFile()* записывают количество действительно прочитанных или записанных байт в двойное слово с адресом, соответственно, *lpNumberOfBytesRead* и *lpNumberOfBytesWrite*.

Параметр *lpOverlapped* используется в функциях *ReadFile()* и *WriteFile()* для организации асинхронного режима чтения и записи. Если запись выполняется синхронно, в качестве этого параметра следует указать значение *NULL*. Способы выполнения асинхронного чтения и записи будут рассмотрены позже. Заметим только, что для использования асинхронного режима файл должен быть открыт функцией *CreateFile()* с использованием флага

FILE_FLAG_OVERLAPPED. Если указан этот флаг, параметр *lpOverlapped* не может иметь значение NULL. Он обязательно должен содержать адрес подготовленной структуры типа *OVERLAPPED*.

Если функции *ReadFile()* и *WriteFile()* были выполнены успешно, они возвращают значение *TRUE*. При возникновении ошибки возвращается значение *FALSE*. В последнем случае вы можете получить код ошибки, вызвав функцию *GetLastError()*. В процессе чтения может быть достигнут конец файла. При этом количество действительно прочитанных байт (записывается по адресу *lpNumberOfBytesRead*) будет равно нулю. В случае достижения конца файла при чтении ошибка не возникает, поэтому функция *ReadFile()* вернет значение *TRUE*.

2.4.1.2 Функция FlushFileBuffers

Так как ввод и вывод данных на диск в операционной системе Windows буферизуется, запись данных на диск может быть отложена до тех пор, пока система не освободится от выполнения текущей работы. С помощью функции *FlushFileBuffers()* можно принудительно заставить операционную систему записать на диск все изменения для файла, идентификатор которого передается этой функции через единственный параметр:

```
BOOL FlushFileBuffers(HANDLE hFile);
```

В случае успешного завершения функция возвращает значение *TRUE*, при ошибке – *FALSE*. Код ошибки вы можете получить при помощи функции *GetLastError()*.

Заметим, что при закрытии файла функцией *CloseHandle()* содержимое всех буферов, связанных с этим файлом, записывается на диск автоматически. Поэтому использовать функцию *FlushFileBuffers()* нужно только в том случае, если запись содержимого буферов нужно выполнить до закрытия файла.

2.4.1.3 Функция SetFilePointer

С помощью функции *SetFilePointer()* приложение может выполнять прямой доступ к файлу, перемещая указатель текущей позиции, связанный с файлом. Сразу после открытия файла этот указатель устанавливается в начало файла. Затем он передвигается функциями *ReadFile()* и *WriteFile()* на количество прочитанных или записанных байт, соответственно.

Функция *SetFilePointer()* позволяет выполнить установку текущей позиции:

```
DWORD SetFilePointer (
  HANDLE hFile,            // идентификатор файла
  LONG lDistanceToMove,        // количество байт, на которое
                               // будет передвинута текущая
                               // позиция
  PLONG lpDistanceToMoveHigh,  // адрес старшего слова,
                               // содержащего расстояние для
                               // перемещения позиции
```

```
  DWORD  dwMoveMethod          // способ перемещения позиции
);
```

Через параметр *hFile* передается идентификатор файла, для которого выполняется изменение текущей позиции.

Параметр *lDistanceToMove*, определяющий дистанцию, на которую будет передвинута текущая позиция, может принимать как положительные, так и отрицательные значения. В первом случае текущая позиция переместится по направлению к концу файла, во втором – к началу файла.

Если планируется работа с файлами, размер которых не превышает $2^{32} – 2$ байта, для параметра *lpDistanceToMoveHigh* можно указать значение *NULL*. В том случае, когда файл очень большой, для указания смещения может потребоваться 64-разрядное значение. Для того чтобы указать очень большое смещение, необходимо записать старшее 32-разрядное слово этого 64-разрядного значения в переменную, и передать функции *SetFilePointer()* адрес этой переменной через параметр *lpDistanceToMoveHigh*. Младшее слово смещения следует передавать как и раньше, через параметр *lDistanceToMove*.

Параметр *dwMoveMethod* определяет способ изменения текущей позиции.

В случае успешного завершения функция *SetFilePointer()* возвращает младшее слово новой 64-разрядной позиции в файле. Старшее слово при этом записывается по адресу, заданному параметром *lpDistanceToMoveHigh*.

При ошибке функция возвращает значение 0xFFFFFFFF. При этом в слово по адресу *lpDistanceToMoveHigh* записывается значение *NULL*. Код ошибки можно получить при помощи функции *GetLastError()*.

2.4.1.4 Функция SetEndOfFile

При необходимости изменить длину файла (уменьшить или увеличить), можно воспользоваться функцией *SetEndOfFile()*. Эта функция устанавливает новую длину файла в соответствии с текущей позицией:

```
BOOL SetEndOfFile(HANDLE hFile);
```

Для изменения длины файла достаточно установить текущую позицию в нужное место с помощью функции *SetFilePointer()*, а затем вызвать функцию *SetEndOfFile()*.

2.4.2 Функции блокирования работы с файлами (LockFile и UnlockFile)

Так как операционная система Windows является мультизадачной и допускает одновременную работу многих процессов, возможно возникновение ситуаций, в которых несколько задач попытаются выполнять запись или чтение для одних и тех же файлов. Например, два процесса могут попытаться изменить одни и те же записи файла базы данных, при этом третий процесс будет в то же самое время выполнять выборку этой записи.

Напомним, что если функции *CreateFile()* указать режимы совместного использования файла *FILE_SHARE_READ* или *FILE_SHARE_WRITE*, несколько процессов смогут одновременно открыть файлы и выполнять операции чтения и записи, соответственно. Если же эти режимы не указаны, совместное использование файлов будет невозможно. Первый же процесс, открывший файл, заблокирует возможность работы с этим файлом для других процессов.

Очевидно, что в ряде случаев все же необходимо обеспечить возможность одновременной работы нескольких процессов с одним и тем же файлом. В этом случае при необходимости процессы могут блокировать доступ к отдельным фрагментам файлов для других процессов. Например, процесс, изменяющий запись в базе данных, перед выполнением изменения может заблокировать участок файла, содержащий эту запись, и затем после выполнения записи разблокировать его. Другие процессы не смогут выполнить запись или чтение для заблокированных участков файла. Блокировка участка файла выполняется функцией *LockFile()*, прототип которой представлен ниже:

```
BOOL LockFile(
  HANDLE hFile,       // идентификатор файла
  DWORD  dwFileOffsetLow, // младшее слово смещения области
  DWORD  dwFileOffsetHigh,// старшее слово смещения области
  DWORD  nNumberOfBytesToLockLow,    // младшее слово длины
                                     // области
  DWORD  nNumberOfBytesToLockHigh    // старшее слово длины
                                     // области
);
```

Параметр *hFile* задает идентификатор файла, для которого выполняется блокировка области. Смещение блокируемой области (64-разрядное) задается при помощи параметров *dwFileOffsetLow* (младшее слово) и *dwFileOffsetHigh* (старшее слово). Размер области в байтах задается параметрами *nNumberOfBytesToLockLow* (младшее слово) и *nNumberOfBytesToLockHigh* (старшее слово). Заметим, что если в файле блокируется несколько областей, они не должны перекрывать друг друга.

В случае успешного завершения функция *LockFile()* возвращает значение *TRUE*, при ошибке – *FALSE*. Код ошибки можно получить при помощи функции *GetLastError()*.

После использования заблокированной области, а также перед завершением своей работы процессы должны разблокировать все заблокированные ранее области, вызвав для этого функцию *UnlockFile()*:

```
BOOL UnlockFile(
  HANDLE hFile,       // идентификатор файла
  DWORD  dwFileOffsetLow, // младшее слово смещения области
  DWORD  dwFileOffsetHigh,// старшее слово смещения области
```

```
  DWORD  nNumberOfBytesToUnlockLow,  // младшее слово длины
                                     // области
  DWORD  nNumberOfBytesToUnlockHigh  // старшее слово длины
);                                   // области
```

Заметим, что в программном интерфейсе операционной системы Windows есть еще две функции, предназначенные для блокирования и разблокирования областей файлов. Эти функции имеют имена, соответственно, *LockFileEx()* и *UnlockFileEx()*.

Главное отличие функции *LockFileEx()* от функции *LockFile()* заключается в том, что она может выполнять частичную блокировку файла, например, только блокировку от записи. В этом случае другие процессы могут выполнять чтение заблокированной области [1].

ЛАБОРАТОРНАЯ РАБОТА №6

ДИНАМИЧЕСКИ ПОДКЛЮЧАЕМЫЕ БИБЛИОТЕКИ

1. Цель работы

Изучение возможностей применения динамически подключаемых библиотек в ОС Windows, приобретение практических навыков создания и использования DLL.

2. Назначение и функции динамически подключаемых библиотек

2.1. Основные понятия

Динамически подключаемые библиотеки (DLL, или динамические библиотеки, или библиотеки динамической компоновки, или модули библиотек) являются одним из наиболее важных структурных элементов Windows. Большинство файлов, из которых состоит Windows, представляют из себя либо программные модули, либо модули динамически подключаемых библиотек. Большая часть принципов, относящихся к написанию обычных программ, вполне подходит и для написания этих библиотек, но есть несколько важных отличий.

Как известно, Windows-программа представляет собой исполняемый файл, который обычно создает одно или более окон, а для получения данных от пользователя использует цикл обработки сообщений. Динамически подключаемые библиотеки, как правило, непосредственно не выполняются и обычно не получают сообщений. **Они представляют собой отдельные файлы с функциями, которые вызываются программами или другими динамическими библиотеками для выполнения определенных задач. Динамически подключаемая библиотека активизируется только тогда,**

когда другой модуль вызывает одну из функций, находящихся в библиотеке.

Термин «динамическое связывание» (*dinamic linking*) относится к процессам, которые Windows использует для того, чтобы связать вызов функции в одном из модулей с реальной функцией из модуля библиотеки. Статическое связывание (*static linking*) имеет место в процессе создания программы, когда в процессе построения исполняемого (EXE) файла связываются воедино разные объектные (OBJ) модули, файлы библиотек (LIB) и, как правило, скомпилированные файлы описания ресурсов (RES). В отличие от этого динамическое связывание имеет место во время выполнения программы.

Файлы *kernel32.dll*, *user32.dll* и *gdi32.dll*, файлы различных драйверов, например *keyboard.drv, system.drv* и *mouse.drv*, драйверы мониторов и принтеров – все это динамически подключаемые библиотеки. Их можно использовать во всех программах Windows. Некоторые динамически подключаемые библиотеки (например, файлы шрифтов) содержат только ресурсы (resource only) и нет текстов программ. Таким образом, одной из целей существования динамически подключаемых библиотек должно быть обеспечение функциями и ресурсами, которые можно использовать во многих, совершенно разных программах.

В традиционной операционной системе содержатся программы, которые для решения каких-то задач могут вызывать другие программы. В Windows принцип вызова одним модулем функций из другого модуля распространен на всю операционную систему. Динамически подключаемые библиотеки (включая те, которые составляют Windows) можно считать дополнением программы.

Хотя модуль динамически подключаемой библиотеки может иметь любое расширение (например, .exe или .fon), **стандартным расширением**, принятым в Windows, является **.dll**. Только те динамически подключаемые библиотеки, которые имеют расширение .dll, Windows загрузит автоматически. **Если файл имеет другое расширение, то программа должна загрузить модуль библиотеки явно. Для этого используется функция *LoadLibrary()* или *LoadLibraryEx().***

2.2. Создание DLL модуля

Зачастую создать DLL проще, чем написать приложение, потому что она является лишь набором автономных функций, пригодных для использования любой программой, причем в DLL обычно отсутствует код, предназначенный для обработки циклов выборки сообщений или создания окон. Функции DLL пишутся в расчете на то, что их будет вызывать какое-то приложение (EXE-файл) или другая DLL. Файлы с исходным кодом компилируются и компонуются так же, как и при создании EXE-файла. Но, создавая DLL, следует указывать компоновщику **ключ /DLL**, который заставляет компоновщика записывать в

конечный файл информацию, по которой загрузчик операционной системы определит, что данный файл – DLL, а не приложение.

Чтобы приложение (или другая DLL) могло вызывать функции из DLL, исполняемый файл **нужно сначала спроецировать на адресное пространство вызывающего процесса.** Это делается либо неявной компоновкой при загрузке, либо явной – в период выполнения программы.

Как только DLL спроецирована на адресное пространство вызывающего процесса, ее функции доступны всем потокам этого процесса. Фактически библиотеки при этом теряют почти всю индивидуальность: для потоков код и данные DLL – просто дополнительный код и данные, оказавшиеся в адресном пространстве процесса. Когда поток вызывает из DLL какую-то функцию, та считывает свои параметры из стека потока и размещает в этом стеке собственные локальные переменные. Кроме того, любые созданные кодом DLL объекты принадлежат вызывающему потоку или процессу – DLL в Win32 ничем не владеет.

Например, если функция из DLL вызывает *VirtualAlloc()*, резервируется регион в адресном пространстве того процесса, которому принадлежит поток, обратившийся к функции из DLL. Если DLL будет выгружена из адресного пространства процесса, зарезервированный регион не освободится, так как система не фиксирует того, что регион выделен библиотечной функцией. Считается, что он принадлежит процессу и поэтому освободится, только если поток этого процесса вызовет *VirtualFree()* или завершится сам процесс.

Когда какой-то процесс проецирует образ DLL файла на свое адресное пространство, система создает также экземпляры глобальных и статических переменных.

Пример использования DLL для разделения данных между двумя приложениями:

```
HGLOBAL g_hData = NULL;
void SetData(LPVOID lpvData, int nSize)
{
     LPVOID lpv;
     g_hData = LocalAlloc(LMEM_MOVEABLE,   nSize);
     lpv = LocalLock(g_hData);
     memcpy(lpv, lpvData, nSize);
     LocalUnlock(g_hData);
}
void GetData(LPVOID lpvData, int nSize)
{
     LPVOID lpv = LocalLock(g_hData);
     memcpy(lpvData, lpv, nSize);
     LocalUnlock(g_hData);
}
```

Вызов *SetData()* приводит к выделению блока памяти из сегмента данных DLL, копированию в него данных, на которые указывает параметр *lpvData*, и сохранению описателя блока в глобальной переменной *g_hData*. Теперь другое приложение может вызвать *GetData()*. Та, используя глобальную переменную *g_hData*, блокирует выделенную область локальной памяти, копирует данные из нее в буфер, идентифицируемый параметром *lpvData*, и возвращает управление.

Вот насколько прост способ разделения данных между двумя процессами в 16-разрядной Windows. В Win32 он не работает: во-первых, у DLL в Win32 нет собственных локальных куч. Во-вторых, глобальные и статические переменные не разделяются между разными проекциями одной DLL; система создает отдельный экземпляр глобальной переменной *g_hData* для каждого процесса, и значения, хранящиеся в разных экземплярах переменной, не обязательно одинаковы.

2.3. Проецирование DLL на адресное пространство процесса

2.3.1. Неявная компоновка

Чтобы поток мог вызвать функцию из DLL-библиотеки, последнюю нужно сначала спроецировать на адресное пространство процесса, которому принадлежит вызывающий поток. Сделать это можно одним из двух способов: неявной компоновкой с функциями DLL и явной загрузкой DLL.

Неявная компоновка (*implicit linking*) – самый распространенный метод проецирования образа DLL-файла на адресное пространство процесса. При сборке приложения компоновщику нужно указать набор LIB-файлов. Каждый такой файл содержит список функций данной DLL, вызов которых разрешен приложениям (или другой DLL). Обнаружив, что приложение ссылается на функции, упомянутые в LIB-файле для DLL, компоновщик внедряет имя этой DLL в конечный исполняемый файл. При загрузке EXE-файла система просматривает его образ на предмет определения необходимых ему DLL, после чего пытается спроецировать их на адресное пространство процесса. Поиск DLL осуществляется в:

- каталоге, содержащем EXE-файл;
- текущем каталоге процесса;
- системном каталоге Windows;
- основном каталоге Windows;
- каталогах, указанных в переменной окружения PATH.

Если файл DLL не найден, система отображает окно с соответствующим сообщением и немедленно завершает процесс. Библиотеки, спроецированные на адресное пространство этим методом, не отключаются от него до завершения процесса.

2.3.2. Явная компоновка

Образ DLL-файла можно спроецировать на адресное пространство процесса явным образом, для чего один из потоков должен вызвать либо *LoadLibrary()*, либо *LoadLibraryEx()*:

```
HINSTANCE LoadLibrary(LPCTSTR lpszLibFile);
HINSTANCE LoadLibraryEx(LPCTSTR  lpszLibFile,
                        HANDLE hFile, DWORD dwFlags);
```

Обе функции ищут образ DLL-файла (в каталогах, список которых приведен в предыдущем разделе) и пытаются спроецировать его на адресное пространство вызывающего процесса. Значение типа ***HINSTANCE***, возвращаемое обеими функциями, **сообщает адрес виртуальной памяти, по которому спроецирован образ файла.** Если спроецировать DLL на адресное пространство процесса не удалось, функции возвращают NULL.

Следует обратить внимание на 2 дополнительных параметра функции *LoadLibraryEx(): hFile* и *dwFlags*. Первый зарезервирован и должен быть NULL. Во втором можно передать либо 0, либо комбинацию флагов *DONT_RESOLVE_DLL_REFERENCES*, *LOAD_LIBRARY_AS_DATAFILE* и *LOAD_WITH_ALTERED_SEARCH_PATH*.

DONT_RESOLVE_DLL_REFERENCES указывает системе спроецировать DLL на адресное пространство вызывающего процесса. Проецируя DLL, система обычно вызывает из нее специальную функцию *DllMain()* (о ней чуть позже) и с ее помощью инициализирует библиотеку. Данный флаг заставляет систему проецировать DLL, не обращаясь к *DllMain()*. Кроме того, DLL может импортировать функции из других DLL. При загрузке библиотеки система проверяет, используются ли ею другие DLL; если да, то загружает и их. При установке флага *DONT_RESOLVE_DLL_REFERENCES* дополнительные DLL автоматически не загружаются.

Флаг *LOAD_LIBRARY_AS_DATAFILE* очень похож на предыдущий – DLL проецируется на адресное пространство процесса так, будто это файл данных. При этом система не тратит дополнительного времени на подготовку к исполнению какого-либо кода из данного файла.

Данный флаг может понадобиться по нескольким причинам. Во-первых, его стоит указать, если DLL содержит только ресурсы и никаких функций. Тогда DLL проецируется и адресное пространство процесса, после чего при вызове функций, загружающих ресурсы, можно использовать значение *HINSTANCE*, возвращенное *LoadLibraryEx()*. Во-вторых, данный флаг может потребоваться, если нужны ресурсы, содержащиеся в каком-нибудь EXE-файле. Обычно загрузка такого файла приводит к запуску нового процесса, но этого не произойдет, если его загрузить вызовом *LoadLibraryEx()* в адресное

пространство процесса. Получив значение *HINSTANCE* для спроецированного EXE-файла, фактически получают доступ к его ресурсам. Так как в EXE-файле нет *DllMain()*, при вызове *LoadLibraryEx()* для загрузки EXE-файла нужно указать флаг *LOAD_LIBRARY_AS_DATAFILE*.

Флаг *LOAD_WITH_ALTERED_SEARCH_PATH* изменяет алгоритм, используемый *LoadLibraryEx()* при поиске DLL-файла. Обычно поиск осуществляется так, как было сказано ранее. Однако, если данный флаг установлен, функция ищет файл, просматривая каталоги в таком порядке:

– каталог, заданный в параметре *lpszLibFile*;
– текущий каталог процесса;
– системный каталог Windows;
– основной каталог Windows;
– каталоги, перечисленные в переменной окружения PATH.

Существует еще один фактор, который может повлиять на то, где система ищет файлы DLL. В реестре есть раздел:

```
HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\KnownDLLs
```

Здесь содержится набор параметров, имена которых совпадают с именами некоторых DLL-файлов. Значения параметров представляют собой строки, идентичные именам параметров.

При вызове *LoadLibrary()* или *LoadLibraryEx()*, каждая из них сначала проверяет, указано ли имя DLL вместе с расширением *.dll*. Если нет, поиск DLL ведется по описанным ранее правилам. Если же расширение *.dll* указано, функция его отбрасывает и ищет в разделе реестра *KnownDLLs* параметр с идентичным именем. Если его нет, вновь применяются описанные ранее правила. Если параметр есть – система обращается к значению, связанному с параметром, и пытается загрузить определенную в нем DLL. При этом система ищет DLL в каталоге, на который указывает значение, связанное с параметром реестра *DllDirectory*. В Windows NT параметру *DllDirectory* по умолчанию присваивается значение "%SystemRoot%\System32".

Если DLL загружается явно, ее можно отключить от адресного пространства процесса функцией *FreeLibrary()*.

```
BOOL FreeLibrary(HINSTANCE hinstDll);
```

При вызове *FreeLibrary()* следует передать значение типа *HINSTANCE*, которое идентифицирует выгружаемую DLL. Это значение можно получить, предварительно вызвав *LoadLibrary()* или *LoadLibraryEx()*.

На самом деле *LoadLibrary()* и *LoadLibraryEx()* лишь увеличивают счетчик числа пользователей указанной библиотеки, а *FreeLibrary()* его уменьшает. Например, при первом вызове *LoadLibrary()* для загрузки DLL система

проецирует образ DLL-файла на адресное пространство вызывающего процесса и присваивает единицу счетчику числа пользователей этой DLL Если поток того же процесса вызывает *LoadLibrary()* для той же DLL еще раз, DLL больше не проецируется; система просто увеличивает счетчик числа ее пользователей. Чтобы выгрузить DLL из адресного пространства процесса, *FreeLibrary()* придется теперь вызывать дважды: первый вызов уменьшит счетчик до 1, второй – до 0. Обнаружив, что счетчик числа пользователей DLL обнулен, система отключит ее. После этого попытка вызова какой-либо функции из данной библиотеки приведет к нарушению доступа, так как код по указанному адресу уже не отображается на адресное пространство процесса. Система поддерживает в каждом процессе свой счетчик DLL, т.е. если поток процесса A вызывает:

```
HINSTANCE hinstDll = LoadLibrary("MyLib.DLL");
```

а затем тот же вызов делает поток в процессе B, то *mylib.dll* проецируется на адресное пространство обоих процессов, а счетчики числа пользователей DLL в каждом из них приравниваются к 1. Если же поток процесса **B** вызовет далее

```
FreeLibrary(hinstDll);
```

счетчик числа пользователей DLL в процессе B обнулится, что приведет к отключению DLL от адресного пространства процесса B. Но проекция DLL на адресное пространство процесса A не затрагивается, и счетчик числа пользователей DLL в нем остается прежним.

Чтобы определить, спроецирована ли DLL на адресное пространство процесса, поток может вызвать функцию

GetModuleHandle().

```
HINSTANCE GetModuleHandle(LPCTSTR IpszModuleName);
```

Например, следующий код загружает *mylib.dll*, только если она еще не спроецирована на адресное пространство процесса:

```
HINSTANCE hinstDll;
hinstDll = GetModuleHandle("MyLib");
if (hinstDll == NULL)
{
     hinstDll = LoadLibrary("MyLib");
}
```

Если имеется значение HINSTANCE для DLL, можно определить и полное имя DLL (или EXE) с помощью *GetModuleFileName()*:

```
DWORD GetModuleFileName(HINSTANCE hinstModule,
                        LPTSTR IpszPath, DWORD cchPath);
```

Первый параметр функции – это значение *HINSTANCE* для EXE или DLL. Второй задает адрес буфера, в который функция запишет полное имя образа файла. Последний параметр (*cchPath*) определяет размер буфера в символах. Уменьшить счетчик числа пользователей DLL можно и с помощью другой Win32-функции:

```
VOID FreeLibraryAndExitThread(HINSTANCE hinstDll,
                              DWORD dwExitCode);
//Она реализована в KERNEL32.DLL так:
VOID FreeLibraryAndExitThread(HINSTANCE hinstDll,
                              DWORD dwExitCode) {
     FreeLibrary(hinstDll);
     ExitThread(dwExitCode);
}
```

Если поток станет сам вызывать *FreeLibrary()* и *ExitThread()* по отдельности, возникнет очень серьезная проблема: вызов *FreeLibrary()* тут же отключит DLL от адресного пространства процесса. После возврата из *FreeLibrary()* код, содержащий вызов *ExitThread()*, окажется недоступен, и поток попытается выполнить неопределенный код. Это приведет к нарушению доступа и завершению всего процесса.

Если же поток обратится к *FreeLibraryAndExitThread()*, она вызовет *FreeLibrary()* и сразу же отключит DLL. Но следующая исполняемая инструкция находится в *kernel32.dll*, а не в только что отключенной DLL. Значит, поток сможет продолжить выполнение и вызвать *ExitThread()*, которая корректно завершит его, не возвращая управления.

2.3.3. Функция входа/выхода

У DLL может быть одна функция входа/выхода – *DllMain()*. Система вызывает ее в некоторых ситуациях сугубо в информационных целях, и обычно она используется DLL для инициализации и очистки в конкретных процессах или потоках. Если DLL подобные уведомления не нужны, эту функцию можно не реализовывать. Пример – DLL, содержащая исключительно ресурсы. Но если эта функция в DLL все же есть, она должна выглядеть так:

```
BOOL WINAPI DllMain(HINSTANCE hinstDll, DWORD fdwReason,
                    LPVOID flmpLoad)
{
     switch (fdwReason) {
          case DLL_PROCESS_ATTACH:
               // DLL проецируется на адресное пр-во процесса
               break;
          case DLL_THREAD_ATTACH:
               // создается поток
               break;
          case DLL_THREAD_DETACH:
               // поток завершается корректно
```

```
                break;
            case DLL_PROCESS_DETACH:
                // DLL отключается от адресного пр-ва процесса
                break;
        }
        return(TRUE);
}
```

Операционная система вызывает функцию входа/выхода в различных ситуациях, при этом параметр *hinstDll* содержит описатель экземпляра DLL. Значение этого параметра равно виртуальному адресу, по которому файл DLL проецируется на адресное пространство процесса (как и *hinstExe* функции *WinMain()*). Обычно последнее значение сохраняется в глобальной переменной, чтобы его можно было использовать и при вызовах функций, загружающих ресурсы (типа *DialogBox()* или *LoadString()*). Если файл DLL загружен неявно, параметр *flmpLoad* отличен от 0, а если явно – равен 0. Параметр *fdwReason* сообщает о причине, по которой система вызвала функцию *DllMain()*. Он принимает одно из 4 значений:

DLL_PROCESS_ATTACH – DLL проецируется на адресное пространство процесса (при первой загрузке);

DLL_PROCESS_DETACH – DLL отключается от адресного пространства процесса;

DLL_THREAD_ATTACH – в процессе создается новый поток;

DLL_THREAD_DETACH – DLL в процессе уничтожается поток.

2.4. Использование DLL

2.4.1. Экспорт функций и переменных из DLL

При создании DLL определяется набор функций доступных для других EXE- или DLL-модулей. Если DLL-функция доступна для вызова из других программ, то говорят, что это экспортируемая (*exported*) функция. Кроме функций Win32 позволяет экспортировать и глобальные переменные.

Пример экспорта из DLL функции *Add()* и глобальной целочисленной переменной *g_nUsageCount*:

```
__declspec(dllexport) int Add(int nLeft, int nRight)
{
    return(nLeft + nRight);
}
__declspec(dllexport) int g_nUsageCount = 0;
```

Компилятор языка C/C++, компилируя функцию *Add()* и *g_nUsageCount*, встраивает в конечный OBJ-файл дополнительную информацию, необходимую для компоновщика при сборке DLL из OBJ-файлов.

Обнаружив такую информацию, компоновщик создает LIB-файл со списком идентификаторов, экспортируемых из DLL. Этот LIB-файл нужен при сборке любого EXE-модуля, вызывающего функции из данной DLL. Кроме того, компоновщик вставляет в конечный файл DLL и таблицу экспортируемых идентификаторов (*exported symbols*). Каждый элемент в этой таблице содержит имя экспортируемой функции или переменной, а также адрес этой функции или переменной внутри DLL-файла. Все списки сортируются по алфавиту.

2.4.2. Импорт функций и переменных из DLL

Чтобы EXE-модуль мог вызывать функции или получать доступ к переменным из DLL, нужно сообщить компилятору, что они находятся именно в DLL. Следующий фрагмент кода показывает, как импортировать функцию *Add()* и переменную *g_nUsageCount*, экспортируемые DLL-модулем:

```
__declspec(dllimport) int Add(int nLeft, int nRight);
__declspec(dllimport) int g_nUsageCount;
```

Конструкция *__declspec(dllimport)* сообщает компилятору, что функция *Add()* и переменная *g_nUsageCount* находятся в DLL, доступ к которой EXE-модуль должен получить при загрузке. Это заставит компилятор сгенерировать специальный код для импортируемых идентификаторов (*imported symbols*). Кроме того, компилятор встроит специальную информацию в конечный объектный файл. Она используется при компоновке EXE-модуля, подсказывая компоновщику, какие функции следует искать в LIB-файлах. Собирая EXE-файл, компоновщик отыскивает импортируемые функции и переменные. Затем, определив, какой LIB-файл содержит эти идентификаторы, добавляет в таблицу импорта новые элементы. Каждый элемент содержит имя соответствующего DLL-файла, а также имя самого идентификатора. Таблица импорта вносится в конечный EXE – файл при его записи на жесткий диск.

2.4.3. Заголовочный файл DLL

Обычно при создании DLL создается и ее заголовочный файл, содержащий прототипы всех экспортируемых из DLL функций и переменных. Он понадобится при компиляции EXE-модулй. Часто его включают и при компиляции исходных файлов самой DLL. Чтобы один и тот же заголовочный файл можно было использовать при компиляции исходных файлов как EXE-, так и DLL-модулей, он должен выглядеть так:

```
//MYLIBAPI определена в файле реализаций MyLib.c как
//__declspec(dllexport). Поэтому при включении заголовочного
//файла в MyLib.c, функции будут экспортироваться, а не
//импортироваться.

#ifndef    MYLIBAPI
     #define    MYLIBAPI __declspec(dllimport)
#endif
```

```
MYLIBAPI int Add(int nLeft,   int riRight);
MYLIBAPI int g_nUsageCount;
```

Этот заголовочный файл надо включать в самое начало исходных файлов DLL следующим образом:

```
//Необходимо определить MYLIBAPI до включения файла MyLib.H.
//Тогда MyLib.H, увидев, что MYLIBAPI уже определена, не
//станет переопределять ее как __declspec(dllimport).

#define   MYLIBAPI   __declspec(dllexport)
#include "MyLib.h"
MYLIBAPI int Add(int nLeft, int nRight)
{
     return(nLeft + nRight);
}
MYLIBAPI int g_nUsageCount;
```

Поскольку *MYLIBAPI* определена как *__declspec(dllexport)* явно, то при компиляции исходного кода программы компилятор узнает, что функции экспортируются. А это позволяет не повторять *__declspec(dllexport)* перед каждой процедурой или переменной.

Что касается исходного кода, импортирующего из DLL идентификаторы, то нужно всего лишь включить в него заголовочный файл. Теперь MYLIBAPI будет определена как *__declspec(dllimport)*, и компилятор «поймет», какие идентификаторы содержатся в DLL. Просмотрев заголовочные файлы Windows, например, можно увидеть, что Microsoft применяет тот же метод.

2.4.4. Динамическое связывание без импорта

Вместо того, чтобы Windows выполняла динамическое связывание при первой загрузке программы в оперативную память, можно связать программу с модулем библиотеки во время выполнении программы. Например, можно было бы просто вызвать функцию *Rectangle()*:

```
Rectangle(hdc, xLeft, yTop, xRight, yBottom);
```

Это работает, поскольку программа была скомпонована с библиотекой импорта *gdi32.lib*, в которой имеется адрес функции *Rectangle()*.

Можно также вызвать функцию *Rectangle()* и совершенно необычным образом. Сначала используется оператор ***typedef*** для определения типа функции *Rectangle()*:

```
typedef BOOL(WINAPI *PFNRECT)(HDC, int, int, int, int);
```

Затем определяются две переменные:

```
HANDLE hLibrary;
PFNRECT pfnRectangle;
```

Теперь устанавливаются значения переменных *hLibrary* равное описателю библиотеки, а значение переменной *pfnRectangle* – равным адресу функции *Rectangle()*:

```
hLibrary = LoadLibrary("gdi32.dll");
if (hLibrary)
{
     pfnRectangle =
          (PFNRECT)GetProcAddress(hLibrary,"Rectangle");

//Теперь можно вызывать функцию и затем освободить библиотеку

     if (pfnRectangle)
          pfnRectangle(hdc, xLeft, yTop, xRight, yBottom);
     FreeLibrary(hLibrary);
}
```

Если этот прием динамического связывания во время выполнения не имеет особого смысла для функции *Rectangle()*, то смысл определенно появляется, если до начала выполнения программы неизвестно имя модуля библиотеки [1].

ЛАБОРАТОРНАЯ РАБОТА №7

ПРОЦЕССЫ И ПОТОКИ

1. Цель работы

Ознакомление с понятиями процессов и потоков в ОС Windows, приобретение практических навыков создания и использования процессов и потоков.

2. Многозадачность и многопоточность в Windows

Многозадачность (multitasking) – это способность операционной системы выполнять несколько программ одновременно. В основе этого принципа лежит использование операционной системой аппаратного таймера для выделения отрезков времени (*time slices*) для каждого из одновременно выполняемых процессов. Если эти отрезки времени достаточно малы и машина не перегружена слишком большим числом программ, то пользователю кажется, что все эти программы выполняются параллельно.

Идея многозадачности не нова. Многозадачность реализуется на больших компьютерах типа *мэйнфрэйм (mainframe)*, к которым подключены десятки, а иногда и сотни терминалов. У каждого пользователя, сидящего за экраном такого терминала, создается впечатление, что он имеет эксклюзивный доступ ко всей машине. Кроме того, операционные системы мэйнфрэймов часто дают

возможность пользователям перевести задачу в фоновый режим, где она выполняется, в то время как пользователь работает с другой программой. Windows NT и Windows 9x – 32-разрядные версии Windows – поддерживают кроме многозадачности еще и *многопоточность (multithreading)*.

Многопоточность – это возможность программы самой быть многозадачной. Программа может быть разделена на отдельные потоки выполнения (*threads*), которые выполняются псевдопараллельно. На первый взгляд эта концепция может показаться едва ли полезной, но оказывается, что программы могут использовать многопоточность для выполнения протяженных во времени операций в фоновом режиме, не вынуждая пользователя надолго отрываться от машины.

2.1 Процессы

Если нужно запустить новую программу, следует создать новый процесс. Для этого используется стандартный системный вызов *CreateProcess()*. Однако пользоваться им не очень удобно – приходится определять значения множества аргументов. Что делать, если необходимо запустить WordPad только для того, чтобы отобразить содержимое файла *«readme.txt»*? Для этой цели можно использовать менее сложный механизм – вызов функции *WinExec()*. При обращении к *WinExec()* необходимо сообщить имя программы (полный путь или короткое имя исполняемого файла, расположенного в пути поиска), а также способ отображения окна программы (константа, используемая функцией *ShowWindow(), SW_SHOW, SW_HIDE* и т. д.).

В случае, если произошла ошибка, функция *WinExec()* возвращает значение, меньшее 32. Если программа успешно запущена, функция возвращает дескриптор новой программы (который не может быть меньше 32). После запуска новой программы функция *WinExec()* немедленно передает управление программе, из которой был осуществлен вызов, т.е. она не ждет момента, пока вновь запущенная программа завершит работу.

Еще один простой вызов, который можно использовать для запуска программ, – это *ShellExecute()*. Он во многом напоминает *WinExec(),* но еще поддерживает обработку типов файлов, зарегистрированных графической оболочкой операционной системы. Например, если при помощи *ShellExecute()* попробовать запустить файл с расширением *.txt,* будет запущена программа notepad.exe или любая другая программа, которая используется в системе для просмотра текстовых файлов. В качестве аргументов функция *ShellExecute()* принимает дескриптор окна (на случай, если возникнет необходимость в сообщениях об ошибках) и операционную строку, такую как "open" (открыть), "print" (распечатать) или "explore" (исследовать). В качестве операционной строки можно передать *NULL*-строку, в этом случае указанный файл будет открыт ("open"). Также функции *ShellExecute()* необходимо сообщить имя файла

и любые параметры командной строки (обычно *NULL*). Наконец, последние два аргумента – это текущий каталог и константа функции *ShowWindow()* (как и в случае с *WinExec()*).

Возвращаемое значение точно такое же, как и у *WinExec()*. Если указать в качестве третьего аргумента функции *ShellExecute()* имя исполняемого файла, можно не использовать другие аргументы, кроме аргумента параметров командной строки и константы *ShowWindow()*. Для файлов документов (например, *.txt или *.doc) значение этих аргументов обычно равно *NULL*.

Функцию *ShellExecute()* можно использовать, например, для того, чтобы открыть корневой каталог диска C:

```
ShellExecute(hWnd,"open","c:\\",NULL,NULL,SW_SHOWNORMAL);
```

Можно заменить строку "open" на строку "explore", а также указать в качестве третьего параметра имя абсолютно любого каталога. Ниже приведен пример простой консольной программы.

```
#include
#include

void main()
{
     printf("Opening with WinExec\n");

     if (WinExec("notepad.exe readme.txt", SW_SHOW) < 32)
          MessageBox(NULL, "Ошибка открытия", NULL, MB_OK);
     else
          MessageBox(NULL,"Нажмите OK для продолжения",
                          "Программа открыта",MB_OK);

     printf("Opening with ShellExecute\n)";

     if (ShellExecute(NULL,"open","readme.txt",NULL,
               NULL,SW_SHOW) < 32)
          MessageBox(NULL, "Ошибка открытия", NULL, MB_OK);
     else
          MessageBox(NULL,"Нажмите OK для продолжения",
                          "Программа открыта",MB_OK);
}
```

Программа открывает текстовый файл, используя при этом два различных способа. Сначала происходит обращение к функции *WinExec()*, которой напрямую сообщаются имя программы текстового редактора и имя текстового файла, который должна открыть эта программа. Затем для открытия того же файла используется системный вызов *ShellExecute()* (при этом, скорее всего, будет запущен *WordPad* или другая программа, ответственная в системе за открытие текстовых файлов).

2.2 Вызов CreateProcess

Вызовы наподобие *ShellExecute()* и *WinExec()* очень удобно использовать для выполнения простейших действий, вроде открытия файлов и запуска программ. Если же необходимо создать новый процесс, используя при этом некоторые дополнительные параметры, следует использовать системный вызов *CreateProcess()*. Описание аргументов, принимаемых этим вызовом, приведено ниже.

Аргументы вызова *CreateProcess()*:

lpApplicationName – имя программы (или *NULL*, если имя программы указано в командной строке);

lpCommandLine – командная строка;

lpProcessAttributes – атрибуты безопасности для дескриптора процесса, возвращаемого функцией

lpThreadAttributes – атрибуты безопасности для дескриптора потока, возвращаемого функцией

bInheritHandlers – указывает, наследует ли новый процесс дескрипторы, принадлежащие текущему процессу

dwCreationFlags – параметры создания процесса (см. таблицу 2)

lpEnvironment – значения переменных окружения (или *NULL*, в случае если наследуется текущее окружение)

lpCurrentDlrectory – текущий каталог (или *NULL*, если используется текущий каталог текущего процесса)

lpProcessInformation – возвращаемые функцией дескрипторы и идентификаторы ID процесса и потока

lpStartupInfo – указатель на структуру *STARTUPINFO*, содержащую информацию о запуске процесса

Значения параметра *dwCreationFlags*, используемые при создании процесса:

CREATE_DEFAULT_ERROR_MODE – не наследовать текущий режим сообщений об ошибках (см. *SetErrorMode()*);

CREATE_NEW_CONSOLE – создать новую консоль;

CREATE_NEW_PROCESS_GROUP – создать новую группу процессов;

CREATE_SEPARATE_WOW_VDM – запустить 16-битное приложение в его собственном адресном пространстве;

CREATE_SHARED_WOW_VDM – запустить 16-битное приложение в адресном пространстве общего доступа;

CREATE_SUSPENDED – создать процесс в приостановленном состоянии (см. *ResumeThread()*);

CREATE_UNICODE_ENVIRONMENT – блок переменных окружения записан в формате *UNICODE*;

DEBUG_PROCESS – запустить процесс в отладочном режиме;

DEBUG_ONLY_TH1S_PROCESS – предотвратить отладку процесса текущим отладчиком (используется при отладке родительского процесса);
DETACHED_PROCESS – новый консольный процесс не имеет доступа к консоли родительского процесса.

Аргумент *lpStartupInfo* – это указатель на структуру *STARTUPINFO*. Поля этой структуры содержат заголовок консоли, начальный размер и позицию нового окна, и перенаправление стандартных потоков ввода/вывода. Новая программа может проигнорировать все эти параметры в зависимости от собственного желания.

Поле *dwFlags* этой структуры содержит флаги, установленные в соответствии с тем, какие из остальных полей структуры необходимо было использовать при запуске новой программы. Например, если сбросить флаг *STARTUSEPOSITION*, поля *dwX* и *dwY* структуры *STARTUPINFO*, содержащие координаты основного окна запускаемой программы, будут проигнорированы.

Функция *CreateProcess()* записывает в аргумент *lpProcessInformation* указатель на структуру, содержащую дескрипторы и идентификаторы ID нового процесса и потока. Доступ к этим дескрипторам определяется аргументами *lpProcessAttributes* и *lpThreadAttributes*.

Некоторые аргументы функции *CreateProcess()* относятся к консольным приложениям, другие - имеют значение для всех типов программ. Зачастую при обращении к *CreateProcess()* можно не заполнять структуру *STARTUPINFO*, однако в любом случае нужно передать функции указатель на существующую в памяти структуру, даже если эта структура не заполнена.

Функция *CreateProcess()* возвращает значение типа *BOOL*, но по завершении работы чрезвычайно полезная для программиста информация размещается функцией в структуре типа *PROCESS_INFORMATION*, указатель на которую возвращается при помощи параметра *lpProcessInformation* этой функции. В структуре *PROCESS_INFORMATION* содержатся идентификатор и дескриптор нового процесса, а также идентификатор и дескриптор самого первого потока, принадлежащего новому процессу. Эти сведения могут использоваться для того, чтобы сообщить о новом процессе другим программам, а также для того, чтобы контролировать новый процесс.

При создании процесса с использованием вызова *CreateProcess()* можно передать новому процессу по наследству некоторые объекты, в частности дескрипторы открытых файлов. Однако, к сожалению, это редко, когда бывает полезным, за исключением случаев, когда необходимо объединить стандартные потоки ввода/вывода нескольких консольных приложений. В этой ситуации дескрипторы файлов стандартных потоков ввода/вывода обладают заранее определенными значениями. Даже если новый процесс получает по наследству дескриптор открытого файла, он не может определить его значения, если только

оно не определено заранее. Можно передать дескриптор в командной строке или в переменной окружения, однако, это не очень удобный вариант.

Обладая дескриптором процесса, можно управлять процессом при помощи следующих вызовов:

GetExitCodeProcess() – возвращает код завершения процесса;

GetGuiRecources() – определяет, сколько объектов USER или GDI используется процессом;

SetPrlorityClass() – устанавливает базовый приоритет процесса;

GetPriorityClass() – возвращает базовый приоритет процесса;

SetProcessAffinityMask() – определяет, какие из процессоров используются процессом в качестве основных;

GetProcessAffinityMask() – устанавливает, какие из процессоров используются процессом в качестве основных;

SetProcessPriorityBoost() – позволяет или запрещает Windows динамически изменять приоритет процесса;

GetProcessPriorityBoost() – возвращает статус изменения приоритета процесса;

SetProcessShutDownParameters() – определяет, в каком порядке система закрывает процессы при завершении работы всей системы;

GetProcessShutDownParameter() – возвращает статус механизма завершения работы системы;

SetProcessWorkingSetSize() – устанавливает минимальный и максимальный допустимый объем физической оперативной памяти, используемый процессом;

GetProcessWorkingSetSize() – возвращает информацию об использованиифизической памяти процессом;

TerminateProcess() – корректное завершение работы процесса;

ExitProcess() – немедленное завершение процесса;

GetProcessversion() – возвращает версию Windows, в среде которой хотел бы работать процесс;

GetProcessTimes() – возвращает степень использования CPU процессом;

GetStartUpInfo() – возвращает переданную процессу при обращениик CreateProcess() структуру *STARTUPINFO*.

Чтобы определить момент завершения процесса, можно воспользоваться одним из нескольких методов, Во-первых, можно использовать вызов *GetExitCodeProcess(),* который возвращает либо значение *STILL_ACTIVE* (если процесс еще продолжает работу), либо код завершения процесса (если процесс завершен). В качестве одного из аргументов этой функции передается указатель на переменную, в которую помещается возвращаемое значение. Узнать дескриптор текущего процесса можно при помощи функции *GetCurrentProcess()*. Чтобы управлять процессом из другого процесса, следует обратиться к функции *OpenProcess(),* которой необходимо передать идентификатор процесса. Чтобы

открыть процесс для желаемого доступа, нужно обладать необходимыми разрешениями на доступ к процессу. Процесс, создающий новый процесс при помощи функции *CreateProcess()*, уже обладает дескриптором нового процесса.

Еще один способ определения текущего состояния процесса подразумевает использование функции *WaitForSingleObject()*. Этот вызов можно использовать для самых разных целей. Основное назначение *WaitForSingleObject()* – определить, находится ли некоторый дескриптор в сигнальном состоянии. Дескриптор процесса переходит в сигнальное состояние тогда, когда процесс завершает свою работу. При обращении к функции *WaitForSingleObject()* необходимо указать дескриптор процесса и интервал времени в миллисекундах. Если интервал времени равен 0, функция завершает работу немедленно, возвращая текущее состояние процесса. Если интервал времени равен константе *INFINIT*, функция будет ждать до тех пор, пока интересующий вас процесс не завершит работу. Если указать конкретное значение интервала времени, функция будет ожидать завершения процесса в течение указанного времени, а затем вернет управление вызвавшей ее программе. Если в течение указанного времени процесс завершит работу, *WaitForSingleObject()* вернет управление вызвавшей программе и сообщит ей, что дескриптор целевого процесса перешел в сигнальное состояние. В противном случае эта функция вернет отрицательный ответ.

Вне зависимости от того, в каком состоянии находится дескриптор целевого процесса, функция *WaitForSingleObject()* также возвращает значение, отражающее успешное выполнение – дескриптор так и не перешел в сигнальное состояние, что не является ошибкой. Чтобы определить состояние процесса, необходимо сравнить значение, которое вернула функция, со значениями *WAIT_OBJECT_0* (сигнальное состояние) и WAIT_TIMEOUT (процесс продолжает функционировать). В случае ошибки функция вернет значение *WAIT_FAILED*.

Чтобы иметь возможность ожидать завершения процесса, нужно обладать открытым дескриптором этого процесса с привилегией *SYNCHRONIZE*. Следует помнить, что идентификатор процесса – это не то же самое, что дескриптор процесса. Дескрипторы процессов нельзя просто так передавать из процесса в процесс. Это означает, что если требуется управлять процессом из другого процесса, то следует, прежде всего, каким-либо образом получить дескриптор управляемого процесса. Для идентификации процесса другими процессами служит идентификатор ID этого процесса, который можно передать из процесса в процесс. Преобразовать идентификатор ID процесса в его дескриптор можно при помощи функции *OpenProcess()*, однако для этого требуется обладать необходимыми привилегиями. Узнать идентификатор текущего процесса можно при помощи функции GetCurrentProcessId(). Используя этот вызов, можно узнать

идентификатор собственного процесса и передать этот идентификатор другому процессу. Получив ID вашего процесса, другой процесс сможет открыть его дескриптор.

При обращении к функции *OpenProcess()* необходимо указать требуемый уровень доступа к открываемому процессу. Иногда получить доступ к процессу на любом из возможных уровней нельзя, поэтому, выбирая уровень, нужно использовать именно тот, который необходим для выполнения задачи, и не более того. Например, чтобы узнать код завершения процесса, достаточно владеть уровнем доступа *PROCESS_QUERY_INFORMATION*. Чтобы иметь возможность завершить работу процесса, необходимо обладать уровнем доступа *PROCESS_TERMINATE*. Можно запросить предоставление полного набора прав доступа к процессу, для этого предназначен уровень доступа *PROCESS_ALL_ACCESS*.

При помощи вызова *LoginUser()* программа, обладающая необходимой привилегией (конкретно *SE_TCB_NAME*), может определить лексему (*token*) подключенного к системе пользователя. Обладая этой лексемой, можно запустить какой-либо процесс от имени пользователя системы. Другими словами, действия, которые выполняет процесс, будут рассматриваться системой как действия, выполняемые пользователем системы. Запуск процесса от имени пользователя осуществляется при помощи вызова *CreateProcessAsUser()*, при этом программа будет запущена от лица пользователя, обладающего указанной лексемой.

В приведенных ниже примерах создаются два простых консольных приложения. Первая программа (MASTER) запускает вторую (SLAVE) и переходит в режим ожидания. Программа SLAVE читает идентификатор процесса (PID – Process Identifier) запустившей ее программы из командной строки и ожидает завершения работы программы MASTER. В командной строке программы MASTER можно указать полный путь к исполняемому файлу программы SLAVE. Обе программы иллюстрируют несколько важных технологий использования функций *CreateProcess()*, *OpenProcess()* и *WaitForSingleObject()*.

Следует обратить внимание, что MASTER не использует большую часть аргументов функции *CreateProcess()*, поэтому в данном конкретном случае для запуска SLAVE вполне можно использовать вызов *WinExec()*.

Программа MASTER:

```
#include
#include
#include

void main(int argc, char *argv[])
```

```
{
  char cmd[128];
  if (argc!=1) strcpy(cmd, argv[1]);
  else strcpy(cmd, "slave.exe");
  int pid = GetCurrentProcessId();
  sprintf(cmd + strlen(cmd)," %d", pid);
  printf("Master: Starting:%d \n", pid);

//In this case, might just as well use WinExec
//if (WinExec(cmd,SW_SHOW) < 32)
//  printf("Master: Slave process did not start\n");
  printf("Master: Try naming slave process on the command line\n");
     }
     cout<<"Master: Sleeping\n";
     cout.flush();

     Sleep(15000);
     cout<<"Master: Exiting\n";
     exit(0);
 }
```

Программа SLAVE:

```
#include
#include
#include

void main(int argc,char *argv[])
{
    if (argc!=2)
    {
       cerr<<"Slave: Please run MASTER.EXE instead.\n";
       exit(1);
    }

   int pid=atoi(argv[1]);
   HANDLE process=OpenProcess
    (PROCESS_QUERY_INFORMATION|SYNCHRONIZE,FALSE,pid);

   if (!process) cout<<"Slave: Error opening process\n";
   cout<<"Slave: Waiting for master to finish\n";
   cout.flush();

   if (WaitForSingleObject(process,INFINITE)==
       STATUS_WAIT_0)
       cout<<"Slave: Master completed\n";
   else
       cout<<"Slave: Unexpected error\n";
       exit(0);
 }
```

2.3 Задания и рабочие наборы

Обычно каждому процессу в Windows 2000 назначается так называемый рабочий набор (*working set*). Рабочий набор процесса определяет, какой объем физической памяти диспетчер памяти Windows пытается сохранить за данным процессом. В рабочем наборе указываются минимальное и максимальное количества страниц памяти, которые должны принадлежать данному процессу. Узнать текущий размер можно при помощи вызова *GetProcessWorkingSize()*. Если объем памяти, доступный для других приложений, становится неприемлемо маленьким, система может нарушить границы, установленные в рабочем наборе той или иной программы. Обладая необходимыми для этого привилегиями, программа может изменить собственный рабочий набор при помощи функции *SetProcessWorkingSetSize()*. Если обе границы становятся равными 0xFFFFFFFF, Windows сбрасывает на диск всю память, принадлежащую процессу.

Еще одним методом управления процессами в Windows 2000 является задание (*job*) – это группа связанных между собой процессов. Создать задание можно при помощи функции *CreateJobObject()*, а открыть существующее задание можно при помощи функции *OpenJobObject()*. Обладая дескриптором задания, можно добавить к нему любые другие процессы при помощи функции *AssignProcessToJobObject()*.

Для объединения процессов в группу может быть несколько причин. Например, используя функцию *TerminateJobObject()*, можно разом завершить работу всех процессов одного задания. При помощи вызова *SetInformationJobObject()* можно установить рабочий набор одновременно для всей группы процессов, входящих в задание. Этот же вызов можно использовать для назначения ограничений всем процессам, входящим в задание. В частности, можно запретить всему заданию обращаться к системному вызову *ExitWindows()*, читать содержимое системного буфера обмена или использовать какие-либо дескрипторы (например, дескрипторы окон) других процессов. Если запрещен доступ к пользовательским дескрипторам процессов одного задания, то процесс, не принадлежащий этому заданию, может предоставить его процессам доступ к пользовательскому дескриптору при помощи функции *UserHandleGrantAccess()*.

Помимо прочего операционная система позволяет получать статистику, связанную с процессами задания. Для этого служит вызов *QueryInformationObject()*, используя который можно получить информацию о том, какую нагрузку на центральный процессор создают процессы, входящие в состав задания, а также другую подобную информацию. Кроме того, при помощи этого вызова можно получить информацию о параметрах конфигурации задания, значения которым присваиваются при помощи функции *SetInformationObject()*.

3. Потоки. Общие сведения

Термин поток (*thread*) означает выполнение некоторой последовательности инструкций кода программы. Все программы, представленные ранее в лабораторных работах, выполняются одним потоком, называемым первичным потоком. Однако программа, написанная для Windows, может запустить один или более вторичных потоков, каждый из которых независимо выполняет набор инструкций программного кода. С точки зрения пользователя, потоки в программе выполняются одновременно. Операционная система (ОС) обычно достигает этого за счет быстрого переключения управления с одного потока на другой (однако, если компьютер имеет более одного процессора, ОС может выполнять потоки действительно одновременно).

3.1 Создание вторичных потоков

Многопоточность особенно полезна в Windows-программе в тех случаях, когда первичный поток программы, допустим, посвящен обработке сообщений, при этом можно обеспечить быстрые ответы программы на команды и другие события. Вторичный поток может быть использован для выполнения некоторой длинной задачи, которая должна блокировать обработку сообщений программы, если выполняется первичный поток. Например, рисование сложной графики, пересчет электронных таблиц, выполнение дисковых операций, связь с последовательным портом. Запуск отдельного потока программы выполняется относительно быстро и занимает немного памяти. Кроме того, все потоки внутри программы выполняются в одном и том же адресном пространстве памяти и используют один и тот же набор ресурсов Windows.

В программах Win32 можно достаточно свободно использовать многопоточность, при условии, что соблюдаются определенные правила.

Для запуска нового потока вызывается функция *CreateThread()*, которая имеет следующий формат:

```
HANDLE CreateThread(
  LPSECURITY_ATTRIBUTES lpThreadAttributes,
  DWORD dwStackSize,
  LPTHREAD_START_ROUTINE lpStartAddress,
  LPVOID lpParameter,
  DWORD dwCreationFlags,
  LPDWORD lpThreadId);
```

Параметры функции следующие:

lpThreadAttributes – указатель на структуру *SECURITY_ATTRIBUTES*, которая содержит дескриптор защиты. Используется в Windows NT (значение, равное *NULL*, указывает на то, что новый поток получит дескриптор защиты по умолчанию), для Windows 9x этот параметр игнорируется;

dwStackSize – размер стека для потока (в байтах);

lpStartAddress – адрес функции, которая будет выполняться в потоке.

Ее прототип должен быть следующим:

```
DWORD WINAPI ThreadFunc(LPVOID);
```

lpParameter – 32-разрядное значение, которое будет передаваться в функцию потока;

dwCreationFlags – дополнительный флаг для управления созданием потока. Если использовать значение, равное *CREATE_SUSPENDED*, то поток будет создан в приостановленном состоянии и не выполняется, пока для него не будет вызвана функция *ResumeThread()*. Если значение флага равно 0, то поток начнет свое выполнение немедленно после создания. Других значений флага не предусмотрено;

lpThreadId – указатель на 32-разрядную переменную, которая получит идентификатор созданного потока.

CreateThread() запускает выполнение нового потока и быстро возвращает управление. С этого момента оба потока – новый и тот, который вызвал *CreateThread()*, работают одновременно. Третий параметр *lpStartAddress* задает функцию потока; новый поток начинает выполнять эту функцию. Функция потока может вызывать другие функции, но когда происходит возврат из функции потока, созданный поток завершается.

Функция потока возвращает значение типа *DWORD*, это значение называется кодом завершения, и другие потоки могут его использовать. Обычно функция потока возвращает значение 0, указывающее на нормальное его завершение.

Информацию в новый поток можно передать через указатель *lpParameter*, который передается в функцию потока. Этот указатель может содержать адрес простого числа, например типа **int**, или адрес структуры, содержащей любое количество информации. Аналогично поток может возвращать информацию начальному потоку, присваивая значение элементу данных, на который указывает параметр lpParameter.

Следующий пример кода начинает новый поток, выполняющий функцию *ThreadFunction()*:

```
DWORD WINAPI ThreadFunction (LPVOID lpParam)
{
      // предложения и функциональные вызовы,
      // которые должны быть выполнены новым потоком
      //...
      return 0;
}

void SomeFunction (void)
{
      //...
      int Code = 1;
```

```
    DWORD ThreadId;
    HANDLE hThread;

    hThread = CreateThread(NULL, 0, ThreadFunction,
                           &Code, 0, &ThreadId);

    CloseHandle(hThread);
}
```

3.2 Прекращение выполнения потока

Можно завершить запущенный поток одним из двух способов. Во-первых, возможно просто получить возврат потока из функции потока (как в примере выше), передавая обратно желаемый код выхода. Это наиболее правильный способ завершения потока; стек, используемый потоком, будет освобожден и все данные объектов, автоматически созданные потоком, будут разрушены (т.е. будут вызваны деструкторы для автоматически созданных объектов). Во-вторых, поток может вызвать функцию API *ExitThread()*, передавая ей желаемый код выхода:

```
VOID ExitThread(DWORD dwExitCode);
```

Вызов *ExitThread()* – это удобный способ немедленно завершить поток из вложенной функции (вместо возврата в начальную функцию потока). При использовании этого метода стек потока будет освобожден, но деструкторы для автоматических данных объектов не будут вызваны. Оба эти способа завершения потока должны быть выполнены самим потоком. Если необходимо завершить поток из другого потока, то этому другому потоку следует передавать сигнал в поток, который нужно закончить, требуя завершения его своими средствами. Кроме этого, поток можно завершить из другого с помощью функции *TerminateThread()*. Прототип ее следующий:

```
BOOL TerminateThread(HANDLE hThread,
                     DWORD dwExitCode);
```

где *hThread* – идентификатор завершаемого потока, а *dwExitCode* – код возврата.

3.3 Управление потоками

CreateThread() возвращает дескриптор созданного потока, который затем может использоваться для управления потоком. Можно вызвать функцию *SuspendThread()*, чтобы временно приостановить выполнение потока:

```
DWORD SuspendThread(HANDLE hThread);
```

Чтобы снова запустить выполнение потока можно вызвать функцию *ResumeThread()*:

```
DWORD ResumeThread(HANDLE hThread);
```

Можно также вызвать *ResumeThread()* для запуска потока, созданного в приостановленном состоянии присваиванием значения *CREATE_SUSPEND* параметру *dwCreationFlags* функции *CreateThread()*. Кроме того, можно изменить приоритет потока с уровня, изначально присвоенного в вызове CreateThread(), вызывая *SetThreadPriority()*.

```
BOOL SetThreadPriority(HANDLE hThread,
                       int nPriority);
```

Например, следующий код поднимает приоритет потока:

```
HANDLE hThread = CreateThread(/*... */);
//...
SetThreadPriority(hThread, THREAD_PRIORITY_ABOVE_NORMAL);
```

Текущий уровень приоритета потока возвращается вызовом:

```
GetThreadPriority.
int GetThreadPriority(HANDLE hThread);
```

Функция *GetExitCodeThread()* определяет, продолжается ли выполнение потока, и, если выполнение остановлено, получить его код возврата (т.е. значение, возвращенное из функции потока или функцией *ExitThread()*):

```
BOOL GetExitCodeThread(HANDLE hThread, LPDWORD lpExitCode);
```

Пример использования функции *GetExitCodeThread()*:

```
DWORD ExitCode;
HANDLE hThread;
hThread = CreateThread(/*... */);
//...
GetExitCodeThread(hThread, &ExitCode);
if(ExitCode == STILL_ACTIVE )
{
     // поток продолжает выполнение
}
else
{
     // поток завершен и ExitCode содержит код выхода
}
```

Как видно из примера, функция *GetExitCodeThread()* присваивает значение кода возврата переменной типа *DWORD*, адрес которой передан функции вторым параметром. Если поток продолжает выполнение, переменной присваивается значение *STILL_ACTIVE*; если поток завершился, переменной присваивается значение кода выхода [1].

ЛАБОРАТОРНАЯ РАБОТА №8

СИНХРОНИЗАЦИЯ ПРОЦЕССОВ И ПОТОКОВ

1. Цель работы

1.1 Ознакомление с существующими механизмами синхронизации процессов и потоков.

1.2 Приобретение практических навыков синхронизации.

2. Синхронизация процессов и потоков

2.1 Необходимость использования механизма синхронизации потоков

В среде, позволяющей исполнять несколько потоков одновременно, очень важно синхронизировать их деятельность. Для этого в операционных системах, базирующихся на Win32, предусмотрен целый ряд синхронизирующих объектов. В данной лабораторной рассматриваются четыре таких объекта: критические секции, объекты-мьютексы, семафоры и события. Но существуют и другие, часть из них описанные в документации.

Здесь можно познакомиться с самыми разными способами применения основных синхронизирующих объектов. Во многих случаях они ведут себя почти одинаково, но различия между ними все же есть, и зачастую именно эти различия диктуют выбор того или иного объекта для конкретной задачи.

Все перечисленные объекты, за исключением критических секций, принадлежат ядру. Таким образом, критические секции не управляются низкоуровневыми компонентами операционной системы, и в работе с ними описатели не используются.

В общем случае поток синхронизирует себя с другим так: он засыпает, и операционная система, не выделяя ему процессорного времени, приостанавливает его выполнение. Но прежде чем заснуть, поток сообщает системе, какое особое событие должно произойти, чтобы его исполнение возобновилось. Как только указанное событие произойдет, поток вновь получит право на выделение ему процессорного времени, и все пойдет своим чередом. Таким образом, отныне выполнение потока синхронизировано с определенным событием.

Для 32-битовых программ, выполняемых под Windows 95 или Windows NT, потоки выполняются асинхронно, это значит, что при выполнении одним потоком отдельной инструкции нельзя даже сказать, какая инструкция другого потока выполняется в данный момент.

Данное свойство многопоточности может создать проблемы, когда два или более потоков получают доступ к таким общим ресурсам, как глобальные переменные. Рассмотрим для примера, как два потока выполняют блок кода,

который наращивает на 1 глобальный счетчик, а затем печатает новое значение счетчика:

```
// глобальное объявление:
 int Count = 0;
// ...
// функция, выполняемая двумя потоками
void SharedFunctions()
 {
  // ...
  ++Count;
  cout << Count << '\n' ;
  // ...
 }
```

Если эта функция вызывается повторно, ожидаемым результатом является вывод последовательных целочисленных значений, начиная с 1. Однако, поскольку код выполняется двумя потоками, может сработать следующий сценарий:

Первый поток наращивает Count;

Первый поток приостанавливается, и второй поток получает управление;

Второй поток инкрементирует Count и печатает новое значение;

Первый поток снова получает управление. Затем он печатает то же значение, что напечатано вторым потоком: его значение, которое нужно было напечатать, будет пропущено.

Чтобы предотвратить этот тип ошибки, отдельные потоки должны синхронизировать свои действия. Существует множество различных ситуаций, в которых потоки необходимо синхронизировать. Во-первых, существуют такие разновидности ресурсов, которые могут не позволять одновременный доступ многим потокам, например: объекты MFC классов, графические объекты, неразделяемые файлы и неразделяемые аппаратные устройства. Кроме того, может понадобиться синхронизировать действия потока-производителя и потока-потребителя; например, если один поток записывает символы в буфер, а другой поток читает и удаляет символы из буфера, читающему потоку понадобиться ждать, пока записывающий поток добавит символ, а записывающему потоку понадобиться ждать, пока читающий поток удалит символ.

Win32 API содержит множество объектов синхронизации, которые можно использовать для синхронизации действий отдельных потоков (здесь термин объект ссылается не на объект C++). Используя эти объекты, можно выполнить различные типы синхронизации: можно запретить одновременный доступ к ресурсам более чем одному потоку, можно ограничить число потоков,

одновременно получающих доступ к ресурсам, можно выполнить иные типы сигнализации между потоками.

2.2 Синхронизация задач с помощью критических секций

Критические секции являются наиболее простым средством синхронизации задач для обеспечения последовательного доступа к ресурсам. Они работают быстро, не снижая заметно производительность системы, но обладают одним существенным недостатком – их можно использовать только для синхронизации задач в рамках одного процесса. В отличие от критических секций объекты Mutex, которые будут рассмотрены позже, допускают синхронизацию задач, созданных разными процессами. Критическая секция создается как структура типа CRITICAL_SECTION:

```
CRITICAL_SECTION csWindowPaint;
```

Обычно эта структура располагается в области глобальных переменных, доступной всем запущенным задачам процесса. Так как каждый процесс работает в своем собственном адресном пространстве, вы не сможете передать адрес критической секции из одного процесса в другой. Именно поэтому критические секции нельзя использовать для синхронизации задач, созданных разными процессами.

В файле winbase.h (который включается автоматически при включении файла windows.h) структура CRITICAL_SECTION и указатели на нее определены следующим образом:

```
typedef RTL_CRITICAL_SECTION CRITICAL_SECTION;
typedef PRTL_CRITICAL_SECTION PCRITICAL_SECTION;
typedef PRTL_CRITICAL_SECTION LPCRITICAL_SECTION;
```

Определение недокументированной структуры RTL_CRITICAL_SECTION вы можно найти в файле winnt.h:

```
typedef struct _RTL_CRITICAL_SECTION
 {
  PRTL_CRITICAL_SECTION_DEBUG DebugInfo;
  LONG   LockCount;     // счетчик блокировок
  LONG   RecursionCount;// счетчик рекурсий
  HANDLE OwningThread;  // идентификатор задачи,
                        //владеющей  секцией
  HANDLE LockSemaphore; // идентификатор семафора
  DWORD  Reserved;       // зарезервировано
 } RTL_CRITICAL_SECTION, *PRTL_CRITICAL_SECTION;
```

Нет никакой необходимости изменять поля этой структуры вручную, так как для этого есть специальные функции. Более того, только эти функции и можно использовать для работы с критическими секциями. В документации SDK

также отмечено, что структуру типа CRITICAL_SECTION нельзя перемещать или копировать.

2.2.1 Инициализация критической секции

Перед использованием критической секции ее необходимо проинициализировать, вызвав для этого функцию InitializeCriticalSection:

```
CRITICAL_SECTION csWindowPaint;
InitializeCriticalSection (&csWindowPaint);
```

Функция InitializeCriticalSection имеет только один параметр (адрес структуры типа CRITICAL_SECTION) и не возвращает никакого значения.

2.2.2 Удаление критической секции

Если критическая секция больше не нужна, ее нужно удалить при помощи функции DeleteCriticalSection, как это показано ниже:

```
DeleteCriticalSection (&csWindowPaint);
```

При этом освобождаются все ресурсы, созданные операционной системой для критической секции.

2.2.3 Вход в критическую секцию и выход из нее

Две основные операции, выполняемые задачами над критическими секциями, это вход в критическую секцию и выход из критической секции. Первая операция выполняется при помощи функции EnterCriticalSection, вторая – при помощи функции LeaveCriticalSection. Эти функции, не возвращающие никакого значения, всегда используются в паре, как это показано в следующем фрагменте исходного текста:

```
EnterCriticalSection (&csWindowPaint);
 hdc = BeginPaint(hWnd, &ps);
 GetClientRect(hWnd, &rc);
DrawText(hdc, "SDI Window", -1, &rc,
  DT_SINGLELINE | DT_CENTER | DT_VCENTER);
EndPaint(hWnd, &ps);
LeaveCriticalSection (&csWindowPaint);
```

В качестве единственного параметра функциям EnterCriticalSection и LeaveCriticalSection необходимо передать адрес структуры типа CRITICAL_SECTION, проинициализированной предварительно функцией InitializeCriticalSection.

Если одна задача вошла в критическую секцию, но еще не вышла ее, то при попытке других задач войти в ту же самую критическую секцию они будут переведены в состояние ожидания. Задачи пробудут в этом состоянии до тех пор, пока задача, которая вошла в критическую секцию, не выйдет из нее.

Таким образом гарантируется, что фрагмент кода, заключенный между вызовами функций EnterCriticalSection и LeaveCriticalSection, будет выполняться

задачами последовательно, если все они работают с одной и той же критической секцией.

2.2.4 Рекурсивный вход в критическую секцию

Операционная система Microsoft Windows NT допускает рекурсивный вход в критическую секцию. Например, приведенный выше фрагмент кода можно было бы составить следующим образом:

```
EnterCriticalSection (&csWindowPaint);
PaintClient(hWnd);
LeaveCriticalSection (&csWindowPaint);
. . .
void PaintClient(HWND hWnd)
 {
  . . .
  EnterCriticalSection (&csWindowPaint);
  hdc = BeginPaint(hWnd, &ps);
  GetClientRect(hWnd, &rc);
  DrawText (hdc, "SDI Window", -1, &rc,
  DT_SINGLELINE | DT_CENTER | DT_VCENTER);
  EndPaint(hWnd, &ps);
  LeaveCriticalSection (&csWindowPaint);
 }
```

Здесь выполняетеся вызов функции PaintClient, находясь в критической секции csWindowPaint. При этом сама функция PaintClient также пользуется той же критической секцией.

Рекурсивный вход задачи в ту же самую критическую секцию не приводит к тому, что задача переходит в состояние ожидания. Однако для освобождения критической секции необходимо вызывать функцию LeaveCriticalSection столько же раз, сколько раз вызывается функция EnterCriticalSection.

2.2.5 Работа задачи с несколькими критическими секциями

В том случае, когда задача работает с двумя ресурсами, доступ к которым должен выполняться последовательно, она может создать несколько критических секций. Эти секции выполняют синхронизацию главной задачи приложения с задачами, создаваемыми для окон.

Заметим, что когда задачи работают с несколькими критическими секциями, все они должны использовать одинаковую последовательность входа в эти критические секции и выхода из них, иначе возможны взаимные блокировки задач.

Пусть, например, в приложении определены две критические секции, синхронизирующие рисование в двух окнах:

```
CRITICAL_SECTION csWindowOnePaint;
CRITICAL_SECTION csWindowTwoPaint;
```

Пусть первая задача выполняет рисование в этих окнах следующим образом:

```
EnterCriticalSection(&csWindowOnePaint);
EnterCriticalSection(&csWindowTwoPaint);
PaintClientWindow(hWndOne);
PaintClientWindow(hWndTwo);
LeaveCriticalSection(&csWindowTwoPaint);
LeaveCriticalSection(&csWindowOnePaint);
```

Пусть вторая задача использует другой порядок входа в критические секции и выхода из них:

```
EnterCriticalSection(&csWindowTwoPaint);
EnterCriticalSection(&csWindowOnePaint);
PaintClientWindow(hWndOne);
PaintClientWindow(hWndTwo);
LeaveCriticalSection(&csWindowOnePaint);
LeaveCriticalSection(&csWindowTwoPaint);
```

При этом есть вероятность того что когда первая задача войдет в критическую секцию csWindowOnePaint, управление будет передано второй задаче, которая войдет в критическую секцию csWindowTwoPaint и перейдет в состояние ожидания. Она будет ждать освобождения критической секции csWindowOnePaint, занятой первой задачей. Однако первая задача тоже перейдет в состояние ожидания, так как ей нужна критическая секция csWindowTwoPaint, занятая второй задачей. В результате обе задачи навсегда останутся в состоянии ожидания, так как они не смогут освободить критические секции, нужные друг другу.

Если нет необходимости выполнять рисование во втором окне сразу после рисования в первом окне, возможно избежать взаимной блокировки задач не только используя правильную последовательность входа и выхода в критические секции, но и выполняя последовательное использование этих критических секций:

```
// Рисование в первом окне
EnterCriticalSection (&csWindowOnePaint);
PaintClientWindow(hWndOne);
LeaveCriticalSection (&csWindowOnePaint);

// Рисование во втором окне
EnterCriticalSection (&csWindowTwoPaint);
PaintClientWindow(hWndTwo);
LeaveCriticalSection (&csWindowTwoPaint);
```

2.3 Синхронизация задач с помощью Mutexов

Простейшим и наиболее типичным объектом синхронизации является взаимное исключение (mutex). Имя этого объекта синхронизации происходит из выражения mutual exclusion (взаимное исключение). Он используется для ограничения доступа к данному ресурсу единственным потоком в одно время. При использовании взаимного исключения первый шаг должен вызывать функцию Win32 API CreateMutex, чтобы создать синхронизирующий объект взаимное исключение (это может сделать любой поток программы).

```
HANDLE CreateMutex(
  LPSECURITY_ATTRIBUTES lpMutexAttributes,
  BOOL bInitialOwner,
  LPCTSTR lpName);
```

Первый параметр описывает атрибуты прав доступа взаимного исключения; передача значения NULL присваивает ему эти атрибуты по умолчанию и делает управление взаимным исключением ненаследуемым. Второй параметр описывает начальное состояние взаимного исключения, передача FALSE создает взаимное исключение, которое изначально свободно (если передать TRUE, взаимное исключение изначально занято). Третий параметр задает имя взаимного исключения; передача NULL создаетnt.

В операционной системе Microsoft Windows не предусмотрено средств, с помощью которых можно было бы определить текущее значение семафора, не изменяя его. В частности, вы не можете задать функции ReleaseSemaphore нулевое значение инкремента, так как в этом случае указанная функция просто вернет соответствующий код ошибки [1].

РАЗДЕЛ 3 КОНТРОЛЬ ЗНАНИЙ

ОБЩИЕ ТРЕБОВАНИЯ К КОНТРОЛЬНОЙ РАБОТЕ

Контрольная работа выполняется на кафедре «Информационные системы и технологии» по дисциплине «Операционные системы и системное программирование».

Контрольная работа представляет собой логически завершенное и оформленное в виде текста произведение индивидуального научно теоретически-практического содержания, направленное на решение определенных проблем и задач в области изучаемых дисциплин.

Выполнение контрольной работы направлено на достижение следующих целей:

– систематизация, обобщение, закрепление и углубление теоретических и практических знаний по циклам дисциплин, изучаемых студентами в процессе их профессиональной подготовки в университете;

– совершенствование навыков применения полученных знаний для решения конкретных задачи, а также навыков самостоятельной работы с научной литературой и обработки результатов теоретических или экспериментальных исследований.

Задание на контрольную работу формируется так, чтобы студент получил навыки инженерной деятельности.

ЦЕЛЬ КОНТРОЛЬНОЙ РАБОТЫ

Целью контрольной работы (КР) является применение теоретических и практических навыков, полученных в ходе обучения студентов по дисциплине «Операционные системы и системное программирование». Предлагаемые индивидуальные задания по разделу «Системного программирования» преследуют цель изучения программирования в среде Win32 API. Индивидуальные задания по разделу «Операционные системы» преследуют цель изучения работы операционных систем и их отладки. Источник необходимой информации – системы оперативной подсказки MSDN и MS SDK Help Files/Win32 Programmer’s Reference. Рекомендуемая литература перечислена в разделе «Список литературных источников».

ЗАЩИТА КОНТРОЛЬНОЙ РАБОТЫ

Выполненная контрольная работа решением преподавателя допускается к защите. Перед этим контрольная работа должна быть подписана студентом-автором. Защита контрольной работы проводится перед преподавателем. Допускается открытая защита в присутствии всей учебной группы, где обучается

автор контрольной работы. При защите может использоваться мультимедийное оборудование.

При защите КР студент делает устное сообщение (защита) продолжительностью несколько минут, в котором показывает соответствие полученных результатов требованиям контрольной работы. При этом следует выделить основные этапы выполнения контрольной работы, отметить стандартные и оригинальные приемы решения поставленной задачи.

Обязательным является демонстрация выполнения рабочего кода и/или приемов работы с операционной системой.

Вопросы, задаваемые студенту, могут касаться как содержания контрольной работы, так и соответствующих разделов курса лекций. Также может быть рассмотрено и оценено владения навыками написания рабочего программного кода.

При определении оценки за работу учитываются:

владение материалом и навыками написания кода;

полнота выполненных задач;

оригинальность решения;

оформление контрольной работы.

Результат аттестации контрольной работы оценивается отметками в баллах по десятибалльной шкале и/или допуском к экзамену/зачету. Положительными являются отметки не ниже 4 (четырех) баллов.

Студент, не представивший в установленный срок контрольную работу или не защитивший его, считается не допущенным к экзамену/зачету.

ОБЩИЕ ТРЕБОВАНИЯ ПО СОДЕРЖАНИЮ КОНТРОЛЬНОЙ РАБОТЫ

Индивидуальные задания контрольных работ определяются преподавателем.

Контрольная работа должна включать описание изучаемого механизма системы, методов, рабочего кода и приемов его использования, а также пример **демонстрационной программы**.

Контрольная работа включает:

1**. Титульный лист**, с указанием названия учебного заведения, кафедры, изучаемого предмета, ее автора, преподавателя, года выполнения работы и вариант индивидуальных заданий. Подпись руководителя ставится после проверки материалов контрольной работы и свидетельствует о допуске работы к защите. После защиты на титульном листе проставляется оценка результатов защиты работы.

2**. Введение,** краткие сведения о работе (1-2 стр.).

Слово **ВВЕДЕНИЕ** записывают прописными буквами полужирным шрифтом по центру, страницу нумеруют. Содержание введения включает пять-

шесть ключевых (значимых) слов, краткое и точное основных сведений о контрольной работе**. Например,** рекомендуется включить в разделе «Операционные системы» сведения о представленной операционной системе, версии и ее кратких характеристик, иные вспомогательные программы с их кратким описанием, а также цель выполнения контрольной работы. В разделе «Системного программирования» рекомендуется указать программный язык выполнения индивидуальных заданий и его краткую характеристику, а также указать среду разработки, ее версию, версию компилятора, и иные вспомогательные программы с их кратким описанием.

3**. Содержание**, где указывается название и страницы размещения в работе введения, глав, параграфов, заключения, списка использованных источников, приложения и т. п. Слово **СОДЕРЖАНИЕ** записывают прописными буквами полужирным шрифтом по центру. Расположение заголовков в содержании должно точно отражать последовательность и соподчиненность разделов и подразделов в тексте контрольной работы.

4**. Основная часть** (изложение соответствующего теме материала). Контрольная работа должна соответствовать стандартам Единой системы конструкторской документации (**ЕСКД**), Единой системы технологической документации (**ЕСТД**), Единой системы программной документации (**ЕСПД**), другим действующим техническим нормативным правовым актам. Приводимые по тексту сведения и решения должны сопровождаться ссылками на источник. Использование заимствованных сведений без ссылок является плагиатом, свидетельствует о несамостоятельном выполнении работы, и служит основанием для недопуска курсового проекта к защите.

Рекомендуемая структура основной части для раздела «Системное программирование», пример:

Работа 1. «Обзор язык программирования C/C++». (здесь пишется *задание согласно варианту*, при необходимости дополнительная информация)

Исходный текс программы (Листинг) (здесь предоставляются *листинг кода*, рекомендуется сохранять нативные настройки среды разработки, ниже пример оформления)

main.cpp

```
#include <iostream>
#include <ctime>

using namespace std;

int main()
{
    srand((unsigned int)time(nullptr));

    int size = 0;
    cout << "Enter array size: ";
```

```
    cin >> size;

    int * array = new int[size];
    for (int i = 0; i < size; ++i) {
        array[i] = rand() % 10;
    }
    cout << endl;

    cout << "Original array: ";
    for (int i = 0; i < size; i++)
    {
        cout << "[" << array[i] << "]";
    }
    cout << endl;
    cout << endl;

    for (int i = 0; i < size; i++)
    {
        for (int j = 0; j < size - 1; j++)
        {
            if (array[j] == 0 && array[j + 1] != 0)
            {
                int temp = array[j];
                array[j] = array[j + 1];
                array[j + 1] = temp;
            }
        }
    }
    cout << "Modified array: ";
    for (int i = 0; i < size; i++)
    {
        cout << "[" << array[i] << "]";
    }
    cout << endl;

    delete[] array;

    return 0;
}
```

Результаты работы программы (здесь предоставляются *копии экранов* результата работы программы)

8. **Заключение** (½ стр.), в котором подводятся итоги выполнения индивидуальных заданий, обобщаются и формулируются **выводы**. Слово **ЗАКЛЮЧЕНИЕ** записывают прописными буквами полужирным шрифтом по центру.

9. **Список использованных источников**, записывают прописными буквами полужирным шрифтом по центру строки, в котором перечень источников с указанием их порядкового номера приводится в порядке очередности цитирования либо в алфавитном порядке. Источник – это книга, журнал и т. п. Ссылки на сайты и имена файлов допустимо использовать только как дополнение к названию документа. При приведении в пояснительной

записке теоретических или справочных сведений обязательно приводится ссылка на источник (номер ссылки в квадратных скобках – [5]). Список использованных источников оформляется отдельным разделом. Использование заимствованных сведений без ссылок является плагиатом, свидетельствует о несамостоятельном выполнении работы, и служит основанием для недопуска к защите контрольной работы. Рекомендуется оформлять список источников по примеру на сайте ВАК «Образцы оформления библиографического описания в списке источников, приводимых в диссертации и автореферате».

Каждый раздел работы (введение, главы, заключение и т. д.) должен иметь заглавие.

ОБЩИЕ ТРЕБОВАНИЯ ПО ОФОРМЛЕНИЮ КОНТРОЛЬНОЙ РАБОТЫ

Контрольную работу выполняют с применением печатающих и графических устройств вывода ПЭВМ. Текст располагают на одной стороне листа формата А4 с соблюдением размеров полей и интервалов. Объем контрольной работы рекомендуется текста через 1,5 интервала, шрифт – 13-14 pt Times New Roman, (поля страниц: левое – 3 см., правое – 2 см., верхнее и нижнее – 1 см.). Номера разделов, подразделов, пунктов и подпунктов следует выделять полужирным шрифтом. Заголовки разделов рекомендуется оформлять полужирным шрифтом размером 14-16 пунктов, а подразделов – полужирным шрифтом 13 - 14 пунктов. Номера страниц обозначаются в правом верхнем углу (титульный лист не нумеруется). Рисунки нумеруются согласно появлению в тексте (выравнивание по центру, шрифт на 2 пункта меньше основного текста)

При оформлении рекомендуется следовать принципам оформления типовых учебных текстов и рекомендациям оформления ВАК. Контрольная работа должна быть оформлена строго в едином стиле!

Необходимо предоставить печатный вариант контрольной работы, а также при необходимости предоставить электронный вариант в рекомендуемом формате doc, docx. Также для программного кода иметь возможность предоставить рабочие файлы.

ЗАДАНИЕ НА КОНТРОЛЬНУЮ РАБОТУ

Основой для контрольной работы должно быть соответствующее задание, которое выдается студенту преподавателем в первые две недели обучения семестра, в котором учебным планом он предусмотрен. Варианты заданий могут отличаться друг от друга как исходными данными, так и характером решаемых задач. Для раздела «Системное программирование» имеется несколько вариантов заданий. Рекомендуется *выбирать вариант индивидуальных заданий по последнему номеру зачетной книжки (или кратно количеству вариантов при превышающем номере зачетной книжки)*. **Общее количество заданий по контрольной равно 8**. Темы заданий соответствуют темам лабораторных работ.

ВАРИАНТЫ ЗАДАНИЯ НА КОНТРОЛЬНУЮ РАБОТУ

ЗАДАНИЕ 1 ОБЗОР ЯЗЫКА ПРОГРАММИРОВАНИЯ С

Вариант №1

Дан двумерный целочисленный массив A(2, 10). Известно, что среди его элементов два и только два равны между собой. Напечатать их индексы. Для удовлетворительного решения этой задачи надо не брать для сравнения одну и ту же пару элементов (A[i][j], A[p][q]) дважды и не запутаться в случаях, когда i = p и j = q.

Вариант №2

Составить программу вывода всех трехзначных десятичных чисел, сумма цифр которых равна данному целому числу M.

Программа должна содержать двойной цикл по i и по j, а k вычисляется по заданной сумме M:

k = M – i – j;

Тогда

N = i + 10*j + 100*k;

Вариант №3

В массиве A(N) каждый элемент равен 0, 1 или 2. Переставить элементы массива так, чтобы сначала располагались все нули, затем все двойки и, наконец, все единицы (дополнительного массива не заводить).

Вариант №4 Напечатать все простые числа, не превосходящие заданное число M. Для ускорения вычислений полезно завести таблицу для уже найденных простых чисел и проверять делимость очередного числа на числа из этой таблицы. Четные числа, естественно, не рассматривать.

Вариант №5

В написанном выражении ((((1?2)?3)?4)?5)?6 вместо каждого знака «?» вставить знак одной из четырех арифметических операций +, –, ?, / так, чтобы результат вычислений равнялся 35 (при делении дробная часть в частном отбрасывается). Достаточно найти одно решение.

Вариант №6

Дан одномерный массив. Все его элементы, не равные нулю, переписать (сохраняя их порядок) в начало массива, а нулевые элементы – в конец массива (новый массив не заводить).

Вариант №7

Натуральное число называется совершенным, если оно равно сумме всех своих собственных делителей, включая 1. Напечатать все совершенные числа, меньшие, чем заданное M.

Вариант №8

Заданы три числа D, M, Y, которые обозначают число, месяц и год. Найти номер N этого дня с начала года (високосные года – это те, у которых номер делится на 400, и те, у которых номер делится на 4, но не делится на 100).

Вариант №9

Последовательность определяется следующим образом:

– начальный элемент – произвольное натуральное число, кратное 3;

– за любым элементом последовательности следует число, равное сумме кубов всех цифр данного элемента.

Теорема: Любая такая последовательность становится постоянной, равной 153. Докажите теорему программно.

Вариант №10

Дан одномерный массив положительных вещественных чисел. Преобразовать этот массив следующим образом: сначала обнуляется минимальный элемент, затем максимальный из оставшихся, далее минимальный из оставшихся и т.д. до тех пор, пока не останется единственный элемент. Вывести на экран значение и индекс оставшегося элемента.

Вариант №11

Задан двумерный массив вещественных чисел размерностью (M+1)x(N+1). В строку m+1 записать суммы элементов по столбцам, в столбец n+1 записать суммы элементов по строкам, а в элемент Am+1,n+1 записать сумму всех элементов массива. Результат вывести на экран.

Вариант №12

Задана матрица (двумерный массив) вещественных чисел размерностью MxN. Транспонировать матрицу, не используя вспомогательного массива. Результат вывести на экран.

ЗАДАНИЕ 2 ФУНКЦИИ В ЯЗЫКЕ С

Вариант №1

Дан двумерный целочисленный массив A(2, N). Известно, что среди его элементов два и только два равны между собой. Напечатать их индексы.

Для удовлетворительного решения этой задачи надо не брать для сравнения одну и ту же пару элементов (A[i][j], A[p][q]) дважды и не запутаться в случаях, когда i = p и j = q.

Решение задачи оформит в виде функции, которая получает в качестве параметров двумерный массив, размер N и указатель на массив, в который необходимо поместить найденные индексы.

Вариант №2

Составить программу вывода всех трехзначных десятичных чисел, сумма цифр которых равна данному целому числу M.

Программа должна содержать двойной цикл по i и по j, а k вычисляется по заданной сумме M:

k = M – i – j;

Тогда

N = i + 10*j + 100*k;

Решение задачи оформит в виде функции, которая получает в качестве параметра число M.

Вариант №3

В массиве A(N) каждый элемент равен 0, 1 или 2. Переставить элементы массива так, чтобы сначала располагались все нули, затем все двойки и, наконец, все единицы (дополнительного массива не заводить).

Решение задачи оформит в виде функции, которая получает в качестве параметров указатель на массив и количество элементов массива.

Вариант №4

Напечатать все простые числа, не превосходящие заданное число M. Для ускорения вычислений полезно завести таблицу для уже найденных простых чисел и проверять делимость очередного числа на числа из этой таблицы. Четные числа, естественно, не рассматривать. Таблица понадобится менее чем на

Решение задачи оформит в виде функции, которая получает в качестве параметров число M, указатель на массив, в который будут помещаться найденные простые числа.

Вариант №5

В написанном выражении ((((1?2)?3)?4)?5)?6 вместо каждого знака «?» вставить знак одной из четырех арифметических операций +, –, ?, / так, чтобы результат вычислений равнялся 35 (при делении дробная часть в частном отбрасывается). Достаточно найти одно решение.

Решение задачи оформит в виде функции, которая получает в качестве параметров указатель на массив с числами выражения, количество чисел в выражении и значение выражения.

Вариант №6

Дан одномерный массив. Все его элементы, не равные нулю, переписать (сохраняя их порядок) в начало массива, а нулевые элементы – в конец массива (новый массив не заводить).

Решение задачи оформит в виде функции, которая получает в качестве параметров указатель на массив и количество элементов массива.

Вариант №7

Натуральное число называется совершенным, если оно равно сумме всех своих собственных делителей, включая 1. Напечатать все совершенные числа, меньшие, чем заданное M.

Решение задачи оформит в виде функции, которая получает в качестве параметра число. Функция возвращает 1 если число совершенно и 0 в противном случае.

Вариант №8

Заданы три числа D, M, Y, которые обозначают число, месяц и год. Найти номер N этого дня с начала года (високосные года учитывать).

Решение задачи оформит в виде функции, которая получает в качестве параметров значения D, M, Y. Функция возвращает количество дней.

ЗАДАНИЕ 3 АДРЕСНАЯ АРИФМЕТИКА И УПРАВЛЕНИЕ ПАМЯТЬЮ

Во всех вариантах необходимо написать три функции, которые будут вызываться из функции main().

Первая функция получает размерность массива, создает динамический массив и возвращает указатель на начало созданного массива.

Вторая – получает адрес массива и его размерность и решает одну из ниже перечисленных задач.

Третья функция получает адрес массива и его размерность и освобождает память, занятую массивом.

Размерность вводится с клавиатуры в функции main() и передается в первую функцию. Значения элементов вводятся с клавиатуры в первой функции.

В заданиях с нечетным номером использовать функции управления памятью библиотеки языка C (#include). В заданиях с четным номером использовать функции управления памятью Win32 API (**#include**).

Варианты заданий

Вариант №1

Массив размерностью MxN. Необходимо найти наибольший и наименьший элементы.

Вариант №2

Массив размерностью MxN. Необходимо каждый элемент строки разделить на сумму элементов строки.

Вариант №3

Массив размерностью MxN. Необходимо каждый элемент строки разделить на наибольший элемент строки.

Вариант №4

Массив размерностью M. Необходимо рассчитать среднее арифметическое по формуле и выборочную дисперсию по формуле, где n = M.

Вариант №5

Массив размерностью MxN. Необходимо дополнить его (M+1)-й строкой и (N+1)-м столбцом, в которых записать суммы элементов соответствующих строк и столбцов. В элементе aM+1,N+1 должна храниться сумма всех элементов массива.

Вариант №6

Массив размерностью MxN. Необходимо в каждой строке найти элемент с наименьшим значением, а затем среди этих чисел найти наибольшее. На экран вывести индексы этого элемента.

Вариант №7

Массив размерностью MxM. Необходимо, не используя дополнительного массива, транспонировать данную матрицу.

Вариант №8

Массив размерностью MxN. Необходимо найти номер строки и номер столбца, в которых находится наименьший элемент.

Вариант №9

Массив размерностью MxM. Необходимо, не используя дополнительного массива, получить обратную матрицу.

Вариант №10

Массив из M строк по N символов каждая. Необходимо вывести только те строки, которые являются палиндромами, т.е. читаются одинаково слева направо и справа налево. При проверке строки необходимо определять ее длину с помощью функции strlen() (#include)

ЗАДАНИЕ 4 ОБРАБОТКА СТРУКТУРИРОВАННЫХ ДАННЫХ

Для всех вариантов необходимо выполнить следующее:

- определить типы и функции в соответствии с вариантом задания;
- в функции **main()** реализовать демонстрацию работы созданных функций;

Вариант №1

Определите структуру Date для хранения даты:

```
struct Date

{
        unsigned y;  //год
        unsigned m; //месяц
        unsigned d;  //день
```

```
};
```

Определите следующие функции:

```
void GetDate(Date* d); // ввод даты с клавиатуры в формате «дд.мм.гггг»;
void PutDate(Date d); // вывод даты в формате «дд.мм.гггг»;
void AddDate(Date* d1, Date d2); // сложение двух дат (результат помещается в d1);
int DiffDate(Date d1, Date d2); // вычисляет разницу в днях между двумя датами.
```

Високосными годами можно пренебречь. Для определения количества дней в месяце можно определить следующий массив:

```
int M[12] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

Вариант №2

Определите структуру Time для хранения времени:

```
struct Time
{
	unsigned h;  //часы
	unsigned m; //минуты
	unsigned s;  //секунды
};
```

Определите следующие функции:

```
void GetTime(Time* t); // ввод времени с клавиатуры в формате «чч:мм:сс»;
void PutTime(Time t); // вывод времени в формате «чч:мм:сс»;
void AddTime(Time* t1, Time t2); // сложение двух времен (результат помещается в t1);
int DiffTime(Time t1, Time t2); // вычисляет разницу в секундах между двумя временами.
```

Вариант №3

Имеется узел бинарного дерева:

```
struct Node
{
	char name[20];	//имя узла
	Node * left;	//левая ветвь
	Node * right;		//правая ветвь
};
```

Определите следующие функции:

Node* AddNode(Node* node, char* name); // добавление нового узла в дерево (если у узла отсутствует левая ветвь, то узел добавляется слева, иначе, если отсутствует правая ветвь, то узел добавляется справа, иначе узел не добавляется). Функция должна возвращать указатель на добавленный узел или 0;
void DelLeftNode(Node* node); void DelRightNode(Node* node); // удаление левых и правых поддеревьев;
void PrintTree(Node* node); // рекурсивная функция вывода бинарного дерева на экран.

Вариант №4

Имеется элемент односвязного списка:

```
struct List
{
        char * data;  //указатель на данные
        List * next;  //указатель на следующий элемент
}
* head;                        //указатель на начало списка
```

Определите следующие функции:

void Add(List** list, int i); // добавление нового элемента в список после i-го элемента;
void PrintList(List* list); // вывод содержимого списка на экран;
void Delete(List** list, int i); // удалить i-й элемент из списка.

Вариант №5

Имеется элемент стека (дисциплина обслуживания LIFO):

```
struct Stack
{
        char * data;  //указатель на данные
        Stack * prev;         //указатель на предыдущий элемент
}
* top;                  //указатель на вершину стека
```

Определите следующие функции:

void Push(Stack** stack, char* data); // поместить данные в стек;

char* Pop(Stack** stack); // извлечь данные из стека (при этом элемент удаляется из стека);
PrintStack(Stack* stack); // вывод на экран содержимого стека.

Вариант №6

Бинарное дерево называется сбалансированным тогда и только тогда, когда высоты двух его поддеревьев отличаются не более чем на единицу. Используя описание узла дерева из задания 3 определите рекурсивную функцию
int TreeHeight(Node* node);
которая вычисляет высоту дерева. Высота дерева, состоящего из единственного узла равна 0. Если узел имеет ветви, то высота такого дерева вычисляется рекуррентно следующим образом:

высота = 1 + max(высота_левого_поддерева, высота_правого_поддерева)

Определите функцию
int IsBalancedTree(Node* node);
которая возвращает 1 если дерево сбалансировано и 0 в противном случае.

Вариант №7

Имеется элемент очереди (дисциплина обслуживания FIFO):

```
struct Queue
{
	char * data;  //указатель на данные
	Queue * next;	//указатель на следующий элемент очереди
} * begin;		//указатель на начало очереди
```

Определите следующие функции:

void Put(Queue** queue, char* data); // поместить данные в конец очереди;
char* Get(Queue** queue); // извлечь данные из начала очереди (при этом элемент удаляется из очереди);
void PrintQueue(Queue* queue); // вывод на экран содержимого очереди.

Вариант №8

Определите структуру Complex для хранения комплексных чисел:

```
struct Complex
{
	double re;	//вещественная часть
	double im;	//мнимая часть
};
```

Определите следующие функции:

Complex Add(Complex c1, Complex c2);
Complex Sub(Complex c1, Complex c2);
Complex Mul(Complex c1, Complex c2);
Complex Div(Complex c1, Complex c2); // сложение, вычитание, умножение и деление комплексных чисел. Все функции должны возвращать новое комплексное число, содержащее результат операции;

void PrintComplex(Complex c); // вывод значения комплексного числа на экран в алгебраической или показательной формах.

Вариант №9

Определите структуру TreeNode – узел дерева:

```
struct TreeNode
{
	char name[20];	//имя узла
	TreeNode * nodes;//список дочерних узлов
	TreeNode * next;	//следующий узел на том же уровне
}
* root;		//корневой узел (первый узел на нулевом уровне)
```

Определите следующие функции:

TreeNode* AddNode(TreeNode* node, char* name); // добавление нового дочернего узла. Функция должна возвращать указатель на новый узел;
TreeNode* FindNode(TreeNode* node, char* name); // поиск узла по его имени;
void DelTree(TreeNode* node); // удаление всех дочерних узлов дерева;
void PrintTree(TreeNode* node); // вывод дерева (имен узлов) на экран.

Вариант №10

Определите структуру Pair – пара «имя = значение» и структуру Pairs – массив пар:

```
struct Pair
{
	char * name;		//имя
	int value;	//значение
};
```

```
#define MAX_PAIRS 100

struct Pairs
{
	Pair p[MAX_PAIRS];	//массив пар
	int count;		//количество пар в массиве
};
```

Определите следующие функции для работы с массивом:

int GetValue(Pairs* p, char* name, int* var); // получить значение для заданного имени (функция должна возвратить 0 если нет такого имени и 1 в случае успеха, а значение возвращать через второй параметр функции);
void SetValue(Pairs* p, char* name, int value); // установить значение для заданного имени (если такого имени в массиве нет, то добавить в массив новую пару);
void PrintPairs(Pairs* p); – вывод всех пар на экран.

ЗАДАНИЕ 5 РАБОТА С ФАЙЛАМИ

Условием выполнения данной работы является выполненная предыдущее задание.
Для всех вариантов необходимо выполнить следующее:

- определить функции в соответствии с вариантом задания;
- в функции main() реализовать демонстрацию работы созданных функций; Во всех заданиях необходимо использовать функции Win32 API для работы с файлами.

Варианты заданий

Вариант №1

Определить две функции:
*void WriteDate(char * fname, Date * buffer, int count)*; – функция записывает *count* элементов типа *Date* из массива *buffer* в файл с именем *fname*;
*int ReadDate(char * fname, Date * buffer, int count)*; – функция читает *count* элементов типа *Date* из файла с именем *fname* в массив *buffer* и возвращает количество фактически прочитанных элементов.

Вариант №2

Определить две функции:
*void WriteTime(char * fname, Time * buffer, int count)*; – функция записывает *count* элементов типа *Time* из массива *buffer* в файл с именем *fname*;

*int ReadTime(char * fname, Time * buffer, int count)*; – функция читает *count* элементов типа *Time* из файла с именем *fname* в массив *buffer* и возвращает количество фактически прочитанных элементов.

Вариант №3

Определите следующие функции:

*void WriteTree(char * fname, Node * node)*; – функция записывает бинарное дерево с корневым элементом *node* в файл с именем *fname*. Для записи отдельного узла имеет смысл определить специальную функцию:

*void WriteNode(HANDLE hf, Node * node)*;

Алгоритм этой функции аналогичен алгоритму вывода дерева на экран, но выполнять она будет следующие действия (обход узлов дерева начинается слева):

- если node == 0, то записать в файл значение 0 (int) и выйти из функции;
- иначе определить количество символов (count) в имени узла;
- записать в файл count и имя узла (count символов из поля name);
- вызвать функцию WriteNode для всех ветвей узла.

*Node * ReadTree(char * fname)*; – функция создает из файла с именем *fname* бинарное дерево. Функция возвращает указатель на корневой узел. Для чтения отдельного узла можно определить специальную функцию

*Node * ReadNode(HANDLE hf)*;

Алгоритм функции может быть следующим:

- прочитать значение типа int в переменную count;
- если count != 0, то прочитать count символов в массив name; создать узел с именем name и для каждой его ветви вызвать функцию ReadNode; возвратить указатель на созданный узел;
- иначе возвратить 0.

Вариант №4

Определите следующие функции:

*void WriteList(char * fname, List * plst)*; – функция записывает элементы списка в файл;

*List * ReadList(char * fname)*; – функция читает значения элементов из файла и создает список;

Вариант №5

Определите следующие функции:

*void WriteStack(char * fname, Stack * pstk)*; – функция записывает элементы стека в файл;

*Stack * ReadStack(char * fname)*; – функция читает значения элементов из файла и создает стек;

Вариант №6

Определить функцию:

*Node * CreateBalansedTree(char * fname)*; – функция создает сбалансированное бинарное дерево. Функция вначале открывает файл, читает первую строку (имя узла) и создает корневой узел. Далее для всех оставшихся строк из файла вызывает функцию:

*void AddNode(Node * node, char * name)*;

Функция AddNode() должна выполнять следующие действия:

- если у узла нет дочерних узлов слева/справа, то добавить слева/справа новый узел;
- иначе выбрать дочерний узел, у которого высота меньше (а если высоты равны то выбрать левый) и вызвать функцию *AddNode()* для выбранного узла.

Вариант №7

Определите следующие функции:

*void WriteQueue(char * fname, Queue * pque)*; – функция записывает элементы очереди в файл;

*Queue * ReadQueue(char * fname); – функция читает значения элементов из файла и создает очередь*

Вариант №8

Определить две функции:

*void WriteComplex(char * fname, Complex * buffer, int count); – функция записывает count элементов типа Complex из массива buffer в файл с именем fname;*

*int ReadComplex(char * fname, Complex * buffer, int count); – функция читает count элементов типа Complex из файла с именем fname в массив buffer и возвращает количество фактически прочитанных элементов.*

Вариант №9

Определить две функции:

*void WriteTree(char * fname, Node * node); – функция записи дерева в файл.*

*Node * ReadTree(char * fname); – функция чтения дерева из файла.*

Вариант №10

Определить две функции:

*void WritePairs(char * fname, Pairs * prs); – функция записывает все элементы типа Pair из массива пар prs в файл с именем fname;*

*void ReadPairs(char * fname, Pairs * prs); – функция читает все элементы типа Pair из файла с именем fname в массив пар prs.*

ЗАДАНИЕ 6 ДИНАМИЧЕСКИ ПОДКЛЮЧАЕМЫЕ БИБЛИОТЕКИ

Условием выполнения данного задания является выполненная (и имеющаяся в наличии) предыдущее задание.
Для всех вариантов необходимо выполнить следующее:

- функции, определенные в работах 4 и 5, оформить в виде библиотеки DLL;
- создать новый проект для демонстрации работы с DLL и функциями;
- в **нечетных** номерах заданий необходимо использовать динамическое связывание без импорта, в **четных** – статическое связывание с импортом;
- при динамическом связывании необходимо в проекте с главной программой объявить указатели на функции, содержащиеся в DLL (см. п. 1.2.4.4);
- при статическом связывании необходимо создать специальный заголовочный файл с объявлениями функций (см. п. 1.2.4.3);

ПРИМЕРЫ
1) Динамическое связывание:
Файл MyDate.cpp

```
struct Date {
	//...
};
__declspec(dllexport) void GetDate(Date * d)
{
	//...
}
//...
```

Текст основной программы

```
struct Date {
	//...
}
typedef void(__cdecl * PFGETDATE)(Date *);
//...
typedef void (__cdecl * PFWRITEDATE)(char *, Date *, int);
//...
void main()
{
```

```
        HANDLE hLibrary;
        PFGETDATE pfGetDate;
        //...
        Date dt;
        hLibrary = LoadLibrary("MyDate.dll");
        if (!hLibrary) return;
        pfGetDate =
                (PFGETDATE)GetProcAddress(hLibrary,"GetDate");
        if (pfGetDate) pfGetDate(&dt);

        //...
        FreeLibrary(hLibrary);
}
```

2) Статическое связывание:

Заголовочный файл MyTime.h

```
#ifndef        MYLIBAPI
        #define        MYLIBAPI __declspec(dllimport)
#endif

MYLIBAPI struct Time
{
        //...
};
MYLIBAPI void GetTime(Time *);
//...
void WriteTime(char *, Time *, int);
//...
```

Файл MyTime.cpp

```
#define        MYLIBAPI __declspec(dllexport)
#include "MyTime.h"

MYLIBAPI void GetTime(Time * t)
{
        //...
}
//...
```

Текст основной программы

```
//укзание компоновщику, что при компоновке необходимо использовать MyTime.lib
#pragma comment(linker, "MyTime.lib")

#include "MyTime.h"

void main()
{
	Time t;
	GetTime(&t);
	//...
}
```

ЗАДАНИЕ 7 ПРОЦЕССЫ И ПОТОКИ

Данное задание выполняется на основе предыдущих заданий №4-6.

Вариант №1

Имеется массив элементов типа Date в виде структуры

```
struct Dates
{
int count = 0;//количество имеющихся элементов в массиве
Date dates[100];
} dts = {0};
```

Главный поток программы (функция *main()*) создает вторичный поток, передав в него указатель на *структуру dts*.

Вторичный поток запоминает значение из поля *count*, открывает файл и затем в цикле, если значение *count* изменилось, то записывает последний элемент массива *dates* в файл. Так продолжается до тех пор, пока *count* не достигнет некоторого максимального значения, после этого поток закрывает файл и завершается;

Далее главный поток организует цикл ввода дат следующим образом:

- инициализируется временная переменная tmp типа Date (ввод с клавиатуры);
- с помощью функции SuspendThread() приостанавливается поток;
- значение временной переменной заносится в массив
- dts.dates[dts.count] = tmp;
- dts.count++;
- с помощью функции ResumeTread() поток запускается на выполнение;

Так продолжается до тех пор, пока *count* не достигнет некоторого максимального значения.

Вариант №2

Главный поток программы создает вторичный поток, в который передает указатель на глобальную переменную типа *Time*.

Поток вычисляет количество секунд пошедших от начала суток до указанного времени и выводит на экран. Затем запоминает время и в последующем вычисляет разницу между новым и старым значениями. Так продолжается до тех пор, пока не будет передано время 00:00:00, после этого поток завершается.

Главный поток в цикле вводит с клавиатуры значение времени во временную переменную tmp, с помощью функции *SuspendThread()* приостанавливает поток, записывает введенное значение в глобальную переменную и затем с помощью функции *ResumeTread()* запускает поток на выполнение. Завершение цикла – ввод значения 00:00:00.

Вариант №3

Функция записи узла дерева в файл *(WriteNode())* выполняется во вторичном потоке. При создании потоку передается указатель на переменную типа *Node**.

Поток должен записать новый узел в файл если адрес узла изменился.

Главный поток в цикле добавляет узлы в дерево (новые узлы добавляются слева на право). Перед добавлением нового узла поток приостанавливается, а в конце добавления запускается. С помощью программы из 4-й работы проверить правильность записанного в файл дерева.

Вариант №4

Аналогично как в варианте 3, но для элементов списка.

Вариант №5

Аналогично как в варианте 4, но для элементов стека.

Вариант №6

Аналогично как в варианте 3, но вторичный поток проверяет сбалансировано дерево или нет. Необходимо добавить две функции: добавления узла (см. лаб. раб. №4 вариант 3) и поиска узла по его имени (рекурсивная функция аналогичная функции вывода дерева, см. там же).

Вариант №7

Аналогично как в варианте 5, но для элементов очереди.

Вариант №8

Аналогично как в варианте 1, но для комплексных чисел.

Вариант №9

Аналогично как в варианте 3, но для узлов TreeNode.

Вариант №10

Аналогично как в варианте 1, но для элементов типа Pair.

ЗАДАНИЕ 8 СИНХРОНИЗАЦИЯ ПРОЦЕССОВ И ПОТОКОВ

Данная работа выполняется на основе предыдущего задания №7.
Во всех вариантах заданий требуется синхронизировать потоки с помощью одного из следующих методов синхронизации:

- критическая секция;
- Mutex;
- событие;
- семафоры.

ВОПРОСЫ ДЛЯ КОНТРОЛЯ ЗНАНИЙ

Характеристика языка С.
Основные сведения о синтаксисе языка С.
Структура программ на языке С.
Функции и библиотеки функций языка С.
Операционные объекты в языке С.
Понятие типа объекта в языке С.
Базовые типы данных в языке С.
Массивы в языке С.
Структуры и объединения в языке С.
Перечисления в языке С.
Оператор переопределения типа в языке С.
Константы в программах на языке С.
Обзор операций языка С.
Операции присваивания в языке С.
Арифметические операции в языке С.
Операции отношений в языке С.
Логические операции в языке С.
Операции над битами в языке С.
Условное вычисление в языке С.
Определение размера объекта в языке С.
Операция приведения типа в языке С.
Последовательное вычисление выражений в языке С.
Операции над указателями в языке С.
Операция вызова функции в языке С.
Приоритет и порядок выполнения операций в языке С.
Условные операторы в языке С.
Операторы цикла в языке С.
Оператор выбора альтернатив (переключатель) в языке С.
Операторы передачи управления в языке С.
Области действия объектов программ в языке С.
Классы памяти объектов программ в языке С.
Инициализации объектов программ в языке С.
Управляемая память в языке С.
Возможности препроцессора и его вызов в языке С.
Операторы лексемного замещения идентификаторов в языке С.
Макрозамещение в языке С.
Оператор включения файлов исходного текста в языке С.
Условная компиляция в языке С.

Изменение нумерации строк и имени файла в языке С.
Расширенные возможности современных процессоров в языке С.
Головная функция программ на языке С.
Порождение и идентификация задач в языке С.
Завершение программ в языке С.
Идентификация задач и виды межзадачных взаимодействий в языке С.
Системно зависимые расширения языка С.
Понятие псевдо-регистров в языке С.
Функции - обработчики прерываний в языке С.
Дополнительные атрибуты указателей в языке С.
Модификаторы типа объектов в языке С.
Использование ассемблера в языке С.
Организация ввода-вывода данных в С.
Бесформатный ввод-вывод в языке С.
Форматный ввод-вывод в языке С.
Виды организации хранения данных в памяти в языке С.
Абстрактные структуры данных в языке С.
Отображение структур данных в памяти в языке С.
Примеры представления структур данных в языке С.
Характеристика проблемы сортировки в языке С.
Методы внутренней сортировки в языке С.
Поиск данных в языке С.
Стандартные процедуры сортировки и поиска данных в языке С.
Анализ схемы распределения памяти в MS-DOS в языке С.
Вывод списка установленных драйверов устройств в языке С.
Получение информации о системных ресурсах в языке С.
Обработка прерываний в ПЭВМ типа IBM PC/XT/AT в языке С.
Понятие системы типа "клиент-сервер" в языке С.
Пример построения системы "клиент-сервер" в QNX в языке С.
Защита файлов от копирования в языке С.
Сетевое представление системы продукций в языке С.
Входное описание систем продукций в языке С.
Интерпретация систем продукций в языке С.

РАЗДЕЛ 4 ВСПОМОГАТЕЛЬНЫЙ

УЧЕБНАЯ ПРОГРАММА 1-40 01 01

Учебная программа по учебной дисциплине «Системное программирование» разработана для по специальности 1-40 01 01 «Программное обеспечение информационных технологий» специализации «Веб-технологии и программное обеспечение мобильных систем». Учебная дисциплина «Системное программирование» знакомит студентов с основными принципами построения и организации работы операционных систем семейства Windows.

Подробно рассматриваются вопросы системного программирования с использованием интерфейса Win32 API. Описываются управление потоками и процессами, включая их диспетчеризацию; передача данных между процессами, с использованием анонимных и именованных каналов, а также почтовых ящиков; структурная обработка исключений; управление виртуальной памятью; управление файлами и каталогами; асинхронная обработка данных; создание динамически подключаемых библиотек; разработка сервисов.

Особое внимание уделено вопросам отладки программного обеспечения. Дается обзор существующих инструментов поиска и устранения дефектов, приводится ряд практических рекомендаций по настройке отладчиков, рассматриваются различные сценарии исследования программного обеспечения.

Рассматриваются основы взаимодействия приложений по сети с использованием библиотеки WinSock.

Отдельно рассматриваются методы перехвата вызовов функций и модификации возвращаемых значений. Главной задачей данных тем является формирование у студентов четкого представления функционирования программного обеспечения, передачи управления между функциями, использование стека потока. Предусмотрены соответствующие лабораторные задания.

Цели учебной дисциплины «Системное программирование»: изучение принципов организации операционных систем на примере операционных систем семейства Windows, изучение способов и принципов создания системного программного обеспечения.

Задачи учебной дисциплины:

– изучение архитектуры операционной системы Windows, способов и принципов организации ее работы;
– ознакомление с возможностями, предоставленными интерфейсом прикладного программирования Win32 API;

– изучение принципов использования средств разработки и отладки, предоставляемых операционной системой, для создания эффективного и безопасного программного обеспечения.

В результате изучения учебной дисциплины студент должен:

знать:

- основные функции операционной системы;
- основные компоненты операционной системы
- методы взаимодействия процессов;
- методы синхронизации потоков;
- модель памяти в защищенном режиме;
- методы управления виртуальной памятью;
- методы управления файлами;
- принципы построения клиент-серверных приложений с использованием библиотеки WinSock;
- принципы обработки исключительных ситуаций в ОС Windows;
- методы перехвата вызовов функций;
- основные виды уязвимостей программного обеспечения;
- механизмы защиты программ, предоставляемые операционной системой;
- методы отладки и поиска дефектов в программном обеспечении;

уметь:

- программировать многопоточные приложения;
- организовать обмен данными между двумя процессами;
- создавать приложения, взаимодействующие по сети;
- создавать и использовать динамически подключаемые библиотеки;
- пользоваться отладчиком, исследовать аварийные дампы памяти, проблемы утечки памяти;

владеть:

- методами и инструментами отладки и поиска дефектов в системном и прикладном программном обеспечении;
- языком программирования низкого уровня С.

Освоение данной учебной дисциплины обеспечивает формирование следующей компетенции:

БПК-16. Применять алгоритмические и программные решения в области системного программного обеспечения, включая программные реализации систем с параллельной обработкой данных и высокопроизводительных систем

Согласно учебным планам на изучение учебной дисциплины отведено:

– для заочной (дистанционной) формы получения высшего образования всего 276, часов из них аудиторных 30 часов. На курсовой проект отведено 60 часов самостоятельной работы;

– для заочной (дистанционной) формы получения высшего образования, интегрированного со средним специальным образованием всего 276, часов из них аудиторных 30 часов. На курсовой проект отведено 60 часов самостоятельной работы.

Распределение аудиторных часов по курсам, семестрам и видам занятий приведено в таблицах 1 и 2.

Таблица 1.

Заочная (дистанционная) форма получения высшего образования					
Курс	Семестр	Лекции, ч.	Лабораторные занятия, ч.	Практические занятия, ч.	Форма промежуточной аттестации
4	7	16	14	-	экзамен, защита курсового проекта

Таблица 2.

Заочная (дистанционная) форма получения высшего образования интегрированного со средним специальным образованием					
Курс	Семестр	Лекции, ч.	Лабораторные занятия, ч.	Практические занятия, ч.	Форма промежуточной аттестации
3	5	16	14	-	экзамен
3	6	-	-	-	защита курсового проекта

СОДЕРЖАНИЕ УЧЕБНОГО МАТЕРИАЛА

РАЗДЕЛ 2 Язык программирования С

2.1 Особенности языка программирования С.

Особенности языка. Отличия от С++. Массивы, строки, адресная арифметика, указатели на функции, функции с переменным числом аргументов. Построение программы. Построение основных структур данных. Стандартная библиотека языка С. Безопасные аналоги стандартных функций языка С.

РАЗДЕЛ 3 Операционная система Windows

3.1 Функции и архитектура операционных систем.

Операционные системы. Функции. Архитектура. Выполнение задач. Процессы, потоки. Многозадачность. Windows API. UNICODE.

3.2 Управление процессами и потоками.

Понятие процесса. Ресурсы, принадлежащие процессу. Создание и завершение процессов. Дескрипторы процесса. Взаимодействие процессов (файлы, командная строка, разделяемая память). Безопасность. Маркер доступа. Понятие потока. Контекст потока.

3.3 Управление памятью.

Модель памяти в защищенном режиме. Виды памяти (стек, куча, пулы памяти режима ядра, тегирование пула).

3.4 Управление файлами.

Обзор средств управления файлами. Представление файлов на жестком диске. Открытие, закрытие, чтение, запись, управление курсором. Чтение и изменение атрибутов, создание и удаление каталогов, наблюдение за изменениями. Копирование и перемещение файлов.

3.5 Динамически подключаемые библиотеки.

Назначение динамически подключаемых библиотек. Варианты использования библиотек. Зависимости библиотек. Формат файла PE. Таблицы импорта и экспорта. Загрузка библиотек. Внедрение кода в другой процесс.

3.6 Сервисы и драйверы Windows.

Сервисы Windows. Создание сервиса. Регистрация сервиса в системе. Менеджер сервисов. Запуск и остановка сервисов. Драйверы, точки входа в драйвер. Объект, описывающий драйвер. Объект, описывающий файл. Взаимосвязь объектов. Запрос ввода-вывода. Менеджер ввода-вывода. Стек драйверов. Прерывания, уровни прерываний. Подпрограммы обработки прерываний. Отложенные вызовы процедур. Асинхронные вызовы процедур. Типы асинхронных процедур.

3.7 Программирование сети.

Обзор модели OSI. Обзор сетевых API Windows. Использование Windows Sockets.

3.8 Перехват API вызовов.

Выполнение кода. Стек потока. Соглашения о вызовах. Перехват функций путем модификации исходного кода. Перехват функций путем модификации таблиц импорта/экспорта. Перехват функций путем модификации системных таблиц. Использование драйверов-фильтров.

РАЗДЕЛ 4 Безопасное программирование

4.1 Уязвимости ПО. Безопасное программирование.

Классификация уязвимостей ПО. Ошибка переполнения буфера. Ошибка переполнения переменных. Ошибки форматирования строк. Механизмы защиты

программ, предоставляемые операционной системой (ASLR, DEP). Безопасное программирование (переполнение чисел, буфера, неправильное использование памяти, проверка возвращаемых значений). Проверка входных данных.

4.2 Структурная обработка исключений.

Обработчики завершения. Фильтры и обработчики исключений. Необработанные исключения и исключения C++.

4.3 Отладка ПО

Обзор отладчиков. Обзор пакета Debugging Tools for Windows. Отладочные символы. Исследование аварийных завершений приложений. Исследование ошибок синхронизации. Исследование утечки памяти. Настройка отладки драйверов в режиме ядра. Настройка аварийного дампа памяти операционной системы.

4.4 Утилиты SysInternals.

Process monitor. Process explorer. Autoruns. Handle

4.5 Инструменты статического анализа кода.

Использование аннотации исходного кода (SAL). PREFast. Code analysis для Visual Studio.

ТРЕБОВАНИЯ К КУРСОВМУ ПРОЕКТУ

В соответствии с учебным планом на выполнение курсового проекта отводится всего 60 часов самостоятельной работы.

Целью курсового проекта (КП) является применение теоретических и практических навыков, полученных в ходе обучения студентов по дисциплине «Системное программирование» и создание системного программного обеспечения для операционной системы.

Курсовой проект представляет собой логически завершенное и оформленное в виде текста произведение индивидуального научно теоретически-практического содержания, направленное на решение определенных проблем и задач в области изучаемой дисциплин.

Тема курсового проекта утверждается на соответствующей кафедре, а задание на ее выполнение оформляется руководителем. Объектом проектирования является системное программное обеспечение для операционной системы. Задание на курсовой проект формируется так, чтобы студент получил навыки инженерной деятельности.

Курсовой проект должен соответствовать стандартам Единой системы конструкторской документации (ЕСКД), Единой системы технологической документации (ЕСТД), Единой системы программной документации (ЕСПД), другим действующим техническим нормативным правовым актам.

В состав курсового проекта входят:

- пояснительная записка;
- графическая часть;
- работающее программное обеспечение.

Пояснительная записка должна отражать основные этапы разработки программного обеспечения.

УЧЕБНО-МЕТОДИЧЕСКАЯ КАРТА УЧЕБНОЙ ДИСЦИПЛИНЫ

Заочная (дистанционная) форма получения высшего образования

Номер раздела, темы	Название раздела, темы, занятия	Количество аудиторных часов					Количество часов УСР	Форма контроля знаний
		Лекции	Практические занятия	Семинарские занятия	Лабораторные занятия	Иное		
1	2	3	4	5	6	7	8	9
	7 семестр							
1.	**Язык программирования С**	0			0			
1.1	Особенности языка программирования С.	2			0			
	Лабораторное занятие №1 Построение программы.	0			2			Защита лабораторной работы
	Лабораторное занятие №2 Построение основных структур данных.	0			2			Защита лабораторной работы
2.	**Операционная система Windows**	0			0			
2.1	Функции и архитектура операционных систем.	2			0			
2.2	Управление процессами и потоками.	2			0			
2.3	Управление памятью.	2			0			

2.4	Управление файлами.	2			0			
	Лабораторное занятие №3 Обзор средств управления файлами.	0			2			Защита лабораторной работы
2.5	Динамически подключаемые библиотеки.	2			0			
2.6	Сервисы и драйверы Windows.	2			0			
	Лабораторное занятие №4 Регистрация сервиса в системе. Менеджер сервисов.	0			2			Защита лабораторной работы
3.	**Безопасное программирование**	0			0			
3.1	Уязвимости ПО. Безопасное программирование.	2			0			Контрольная работа
	Лабораторное занятие №5 Безопасное программирование (переполнение чисел, буфера, неправильное использование памяти, проверка возвращаемых значений).	0			2			Защита лабораторной работы
	Лабораторное занятие №6 Обзор пакета Debugging Tools for Windows.	0			2			Защита лабораторной работы
	Лабораторное занятие №7 Code analysis для Visual Studio.	0			2			Защита лабораторной работы
	Курсовой проект	0			0			Защита курсового проекта
	Итого за семестр	16			14			Экзамен

	Всего аудиторных часов	30						

[1]Темы учебного материала, не указанные в Учебно-методической карте, отводятся на самостоятельное изучение студента

Заочная (дистанционная) форма получения высшего образования, интегрированного со средним специальным.

Номер раздела,	Название раздела, темы, занятия	Количество аудиторных часов					Количество часов УСР	Форма контроля знаний
		Лекции	Практические занятия	Семинарские занятия	Лабораторные занятия	Иное		
1	2	3	4	5	6	7	8	9
	5 семестр							
1.	**Язык программирования С**	0			0			
1.1	Особенности языка программирования С.	2			0			
	Лабораторное занятие №1 Построение программы.	0			2			Защита лабораторной работы
	Лабораторное занятие №2 Построение основных структур данных.	0			2			Защита лабораторной работы
2.	**Операционная система Windows**	0			0			
2.1	Функции и архитектура операционных систем.	2			0			
2.2	Управление процессами и потоками.	2			0			
2.3	Управление памятью.	2			0			

2.4	Управление файлами.	2			0			
	Лабораторное занятие №3 Обзор средств управления файлами.	0			2			Защита лабораторной работы
2.5	Динамически подключаемые библиотеки.	2			0			
2.6	Сервисы и драйверы Windows.	2			0			
	Лабораторное занятие №4 Регистрация сервиса в системе. Менеджер сервисов.	0			2			Защита лабораторной работы
3.	**Безопасное программирование**	0			0			
3.1	Уязвимости ПО. Безопасное программирование.	2			0			Контрольная работа
	Лабораторное занятие №5 Безопасное программирование (переполнение чисел, буфера, неправильное использование памяти, проверка возвращаемых значений).	0			2			Защита лабораторной работы
	Лабораторное занятие №6 Обзор пакета Debugging Tools for Windows.	0			2			Защита лабораторной работы
	Лабораторное занятие №7 Code analysis для Visual Studio.	0			2			Защита лабораторной работы
	Итого за семестр	16			14			Экзамен
	6 семестр	0			0			

	Курсовой проект	0			0			Защита курсового проекта
	Итого за семестр							
	Всего аудиторных часов	30						

[1]Темы учебного материала, не указанные в Учебно-методической карте, отводятся на самостоятельное изучение студента

ИНФОРМАЦИОННО-МЕТОДИЧЕСКАЯ ЧАСТЬ

Список литературы

Основная литература

1. Электронный учебно-методический комплекс по учебной дисциплине «Системное программирование» для специальности: I -53 01 02 «Автоматизированные системы обработки информации» [Электронный ресурс] / Белорусский национальный технический университет, Кафедра «Информационных технологий автоматизированных систем» ; сост. М. П. Ревотюк. – Минск : БНТУ, 2006.
2. Руссинович, М. Внутреннее устройство Windows / М. Руссинович [и др.]. – 7-е изд. – СПб. : Питер, 2018. – 944 с.
3. Рихтер, Дж. Windows для профессионалов. Создание эффективных WIN32-приложений с учетом специфики 64-разрядной версии Windows / Дж. Рихтер. – СПб. : Питер, 2001. – 752 с.
4. Побегайло, А. П. Системное программирование в Windows / А. П. Побегайло. – СПб. : БХВ-Петербург, 2006. – 1056 с.
5. Hewardt, M. Advanced Windows Debugging / Hewardt M., D. Pravat. – Boston : Addison-Wesley Professional, 2007. – 840 с.
6. Шилдт, Г. Полный справочник по C++ / Г. Шилдт. – 4-е изд. – М. : Издательский дом «Вильямс», 2006. – 800 с.
7. Таненбаум, Э. Современные операционные системы / Э. Таненбаум, Х. Бос. – 4-е изд. – СПб. : Питер, 2015. – 1120 с.

Дополнительная литература

8. Беляев, А. Централизованная обработка исключений / А. Беляев // RSDN Magazine, 25.09.2004. – [б.м.].
9. Лохас П. Debugging: Развертывание сервера отладочной информации / П. Лохас // Habrahabr [Электронный ресурс]. – 2010. – Режим доступа : http://habrahabr.ru/blogs/development/89094/. – Дата доступа : 29.11.2023.
10. Лохас П. Debugging: Введение в postmortem debugging / П. Лохас // Habrahabr [Электронный ресурс]. – 2010. – Режим доступа : http://habrahabr.ru/blogs/development/89220/. – Дата доступа : 29.11.2023.
11. Харт, Дж. М. Системное программирование в среде Windows / Дж. М. Харт. – 3-е изд. – М. : Издательский дом «Вильямс», 2005. – 592 с.

Средства диагностики результатов учебной деятельности

Оценка уровня знаний студента производится по десятибалльной шкале в соответствии с критериями, утвержденными Министерством образования Республики Беларусь. Для оценки достижений студента рекомендуется использовать следующий диагностический инструментарий:

- контрольная работа;
- защита выполненных в рамках лабораторных занятий заданий;
- защита курсового проекта;
- экзамен.

ПЕРЕЧЕНЬ ТЕМ КУРСОВЫХ ПРОЕКТОВ

1. Мониторинг потока событий (Hooks)
2. Сетевое окружение (Windows Networking)
3. Мониторинг изменений файлов и каталогов (File I/O)
4. Работа с реестром (Registry)
5. Процессы и потоки (Processes and Threads)
6. Синхронизация процессов и потоков (Synchronization)
7. Обработка исключений (Structured Exception Handling)
8. Динамический обмен данными (DDEML)
9. Механизмы межпроцессных взаимодействий (IPC)
10. Панель управления (Control Panel Application)
11. Именованные каналы (Pipes)
12. Почтовые ящики (MailSlots)
13. Выгрузка системы (System Shutdown)
14. Получение сведений о системе (System Informatio)
15. Оконные интерфейсы (Windows Station and Desktops)
16. Интерфейс интерпретатора команд (Shell Library)
17. Хранитель экрана (Screen Saver)
18. Проецирование файлов (File Mapping)
19. Синхронный ввод/вывод и временные файлы (Files)
20. Асинхронный ввод вывод (Files)
21. Протоколирование событий (Event Logging)
22. Динамически загружаемые библиотеки (DLL)
23. Управление памятью (Memory Management)
24. Отладка (Debugging)
25. Буфер обмена (ClipBoard)
26. Управление энергосбережением (Power Management)
27. Сжатие данных (Data Decompression Library)
28. Таймеры (Timers)
29. Инструментальная библиотека (Tool Help Library)

Методические рекомендации по организации и выполнению самостоятельной работы студентов

При изучении дисциплины рекомендуется использовать следующие формы самостоятельной работы:

- составление тематической подборки литературных источников;
- проработка тем, вынесенных на самостоятельное изучение;
- подготовка курсового проекта по индивидуальным заданиям.

УЧЕБНАЯ ПРОГРАММА 6-05-0612-01

Учебная программа по учебной дисциплине «Системное программирование» разработана для по специальности 6-05-0612-01 «Программная инженерия». Учебная дисциплина «Системное программирование» знакомит студентов с основными принципами построения и организации работы операционных систем семейства Windows/Linux.

Подробно рассматриваются вопросы системного программирования. Описываются управление потоками и процессами, включая их диспетчеризацию; передача данных между процессами, с использованием анонимных и именованных каналов, а также почтовых ящиков; структурная обработка исключений; управление виртуальной памятью; управление файлами и каталогами; асинхронная обработка данных; создание динамически подключаемых библиотек; разработка сервисов.

Особое внимание уделено вопросам отладки программного обеспечения. Дается обзор существующих инструментов поиска и устранения дефектов, приводится ряд практических рекомендаций по настройке отладчиков, рассматриваются различные сценарии исследования программного обеспечения.

Рассматриваются основы взаимодействия приложений по сети. Отдельно рассматриваются методы перехвата вызовов функций и модификации возвращаемых значений. Главной задачей данных тем является формирование у студентов четкого представления функционирования программного обеспечения, передачи управления между функциями, использование стека потока. Предусмотрены соответствующие лабораторные задания.

Базовыми учебными дисциплинами по курсу «Системное программирование» являются «Основы алгоритмизации и программирования», «Алгоритмы и структуры данных», «Конструирование программного обеспечения», «Компьютерные системы и сети».

Цели учебной дисциплины «Системное программирование»: изучение принципов построения и методологии разработки системного программного обеспечения для современных процессоров с использованием современных алгоритмических языков и систем программирования.

Задачи учебной дисциплины:

- приобретение базовых знаний в области принципов построения и методологии разработки системного программного обеспечения;
- изучение способов разработки системного программного обеспечения с учетом особенностей современных операционных систем;
- овладение методами разработки, тестирования, отладки и документирования программ, направленных на решение системных задач, с использованием современных инструментальных средств.

В результате изучения учебной дисциплины студент должен:

знать:

- программные интерфейсы современных операционных систем в пользовательском режиме работы;
- принципы построения приложений с графическим пользовательским интерфейсом для современных операционных систем;
- понятие динамически-загружаемой библиотеки и средства построения таких библиотек в современных операционных системах;
- средства поддержки многозадачности в современных операционных системах;
- способы синхронизации задач в многозадачных операционных системах;
- программные интерфейсы современных операционных систем в режиме ядра;
- системные механизмы современных операционных систем: прерывания, исключения и системные вызовы;
- системные механизмы отложенных процедур, асинхронных процедур и рабочих элементов в ядре операционной системы;
- модель памяти в ядре современных операционных систем;

уметь:

- создавать программы пользовательского режима для современных операционных систем;
- создавать программы с графическим пользовательским интерфейсом для современных операционных систем;
- создавать динамически-загружаемые библиотеки для современных операционных систем;
- создавать программы по организации взаимодействия между процессами и потоками в современных операционных системах;
- создавать многопоточные программы с синхронизацией данных для современных операционных систем;
- создавать системные службы для современных операционных систем;
- создавать драйверы для современных операционных систем.

владеть:

- принципами построения служб для современных операционных систем;
- принципами построения драйверов для современных операционных систем;
- навыками применения средств программирования для современных операционных систем.

Освоение данной учебной дисциплины обеспечивает формирование следующей компетенции:

БПК-16. Применять алгоритмические и программные решения в области системного программного обеспечения, включая программные реализации систем с параллельной обработкой данных и высокопроизводительных систем

Согласно учебным планам на изучение учебной дисциплины отведено:

- для заочной формы получения высшего образования всего 276, часов из них аудиторных 12 часов. На курсовой проект отведено 60 часов самостоятельной работы;
- для за формы получения высшего образования, интегрированного со средним специальным образованием всего 276, часов из них аудиторных 12 часов. На курсовой проект отведено 60 часов самостоятельной работы.

Распределение аудиторных часов по курсам, семестрам и видам занятий приведено в таблицах 1 и 2.

Таблица 1.

Заочная форма получения высшего образования					
Курс	Семестр	Лекции, ч.	Лабораторные занятия, ч.	Практические занятия, ч.	Форма текущей аттестации
3	5	6	6	-	экзамен, защита курсового проекта

Таблица 2.

Заочная форма получения высшего образования, интегрированного со средним специальным					
Курс	Семестр	Лекции, ч.	Лабораторные занятия, ч.	Практические занятия, ч.	Форма текущей аттестации
3	5	6	6	-	экзамен, защита курсового проекта

СОДЕРЖАНИЕ УЧЕБНОГО МАТЕРИАЛА

РАЗДЕЛ 1 Системное программирование в пользовательском режиме

1.1 Оконный пользовательский интерфейс

Основные элементы программ с оконным пользовательским интерфейсом. Минимальная программа для ОС Windows и/или Linux с окном на экране. Создание и отображение окна. Понятие оконного сообщения. Источники сообщений. Очередь сообщений. Цикл приема и обработки сообщений. Процедура обработки сообщений. Синхронные и асинхронные сообщения, их передача и обработка. Ввод данных с манипулятора «мышь». Обработка сообщений мыши. Ввод данных с клавиатуры. Понятие фокуса ввода. Обработка сообщений от клавиатуры. Вывод информации в окно. Механизм перерисовки окна. Понятие ресурсов программ. Виды ресурсов. Работа с ресурсами.

1.2 Интерфейс графических устройств

Принципы построения графической подсистемы операционной системы. Понятие контекста устройства. Вывод графической информации на физическое устройство. Рисование геометрических фигур. Графические инструменты. Управление цветом. Палитры цветов. Растровые изображения. Вывод растровых изображений. Значки и курсоры. Вывод растровых изображений с эффектом прозрачного фона. Вывод текста. Логические и физические шрифты. Системы координат. Трансформации. Метафайлы.

1.3 Многозадачность

Организация многозадачности в операционных системах. Понятие процесса и потока. Контекст потока. Создание и завершение процессов и потоков. Синхронизация потоков одного и того же процесса. Критические секции. Синхронизация потоков разных процессов. Объекты синхронизации: флаги, семафоры, события, ожидаемые таймеры, трубы.

1.4 Динамически загружаемые библиотеки

Понятие динамически загружаемой библиотеки. Структура динамически загружаемой библиотеки. Создание динамически загружаемой библиотеки. Использование динамически загружаемой библиотеки в программе. Импорт динамически загружаемой библиотеки на старте программы. Импорт динамически загружаемой библиотеки во время работы программы.

РАЗДЕЛ 2 Системное программирование в режиме ядра

2.1 Отладка программ в режиме ядра

Отладчики для режима ядра. Режимы отладки. Компоненты отладчика. Представление в памяти строк формата Unicode. Представление в памяти двусвязных списков. Создание дампа памяти. Анализ дампов памяти.

2.2 Системные механизмы ядра

Прерывания. Уровни прерываний. Подпрограммы обработки прерываний. Отложенные процедуры. Асинхронные процедуры. Типы асинхронных процедур.

Рабочие элементы. Переход из пользовательского режима в режим ядра. Таблицы функций операционной системы.

2.3 Виртуальное адресное пространство

Пулы памяти. Пул подкачиваемой памяти, пул неподкачиваемой памяти, пул сессии, особый пул. Тегирование пулов. Структура данных пула. Представление объекта в памяти ядра. Менеджер объектов. Ассоциативные списки. Блокирование страниц в памяти. Списки описателей памяти и их использование для работы с аппаратурой.

2.4 Драйверы

Структура драйвера. Точки входа в драйвер. Объект, описывающий драйвер. Объект, описывающий устройство. Объект, описывающий файл. Взаимосвязь объектов. Запрос ввода-вывода. Менеджер ввода-вывода. Стек драйверов. Организация сетевых драйверов.

2.5 Перехват функций

Технологии перехвата функций операционной системы в пользовательском режиме. Технологии перехвата функций операционной системы в режиме ядра. Стандартные методы перехвата функций в режиме ядра: обратные вызовы системного реестра, обратные вызовы менеджера объектов, обратные вызовы процессов, обратные вызовы потоков, обратные вызовы загрузчика модулей, мини-фильтры файловой системы.

ТРЕБОВАНИЯ К КУРСОВМУ ПРОЕКТУ

В соответствии с учебным планом на выполнение курсового проекта отводится всего 60 часов самостоятельной работы.

Целью курсового проекта (КП) является применение теоретических и практических навыков, полученных в ходе обучения студентов по дисциплине «Системное программирование» и создание системного программного обеспечения для операционной системы.

Курсовой проект представляет собой логически завершенное и оформленное в виде текста произведение индивидуального научно теоретически-практического содержания, направленное на решение определенных проблем и задач в области изучаемой дисциплин.

Тема курсового проекта утверждается на соответствующей кафедре, а задание на ее выполнение оформляется руководителем. Объектом проектирования является системное программное обеспечение для операционной системы. Задание на курсовой проект формируется так, чтобы студент получил навыки инженерной деятельности.

Курсовой проект должен соответствовать стандартам Единой системы конструкторской документации (ЕСКД), Единой системы технологической

документации (ЕСТД), Единой системы программной документации (ЕСПД), другим действующим техническим нормативным правовым актам.

В состав курсового проекта входят:

- пояснительная записка;
- графическая часть;
- работающее программное обеспечение.

Пояснительная записка должна отражать основные этапы разработки программного обеспечения.

УЧЕБНО-МЕТОДИЧЕСКАЯ КАРТА УЧЕБНОЙ ДИСЦИПЛИНЫ

Заочная форма получения высшего образования

Номер раздела, темы	Название раздела, темы, занятия	Количество аудиторных часов					Количество часов УСР	Форма контроля знаний
		Лекции	Практические занятия	Семинарские занятия	Лабораторные занятия	Иное		
1	2	3	4	5	6	7	8	9
	5 семестр							
1.	Системное программирование в пользовательском режиме	0			0			
1.1	Оконный пользовательский интерфейс	2			0			
1.2	Динамически загружаемые библиотеки	2						
	Лабораторное занятие №1 Построение программы.	0			2			Защита лабораторной работы
	Лабораторное занятие №2 Построение основных структур данных.	0			2			Защита лабораторной работы
2.	Системное программирование в режиме ядра	0			0			
2.4	Драйверы	2			0			

	Лабораторное занятие №3 Обзор средств управления файлами.	0			2			Защита лабораторной работы
	Курсовой проект	0			0			Защита курсового проекта
	Итого за семестр	6			6			Экзамен
	Всего аудиторных часов	12						

[1]Темы учебного материала, не указанные в Учебно-методической карте, отводятся на самостоятельное изучение студента

Заочная форма получения высшего образования, интегрированного со средним специальным

Номер раздела, темы	Название раздела, темы, занятия	Количество аудиторных часов					Количество часов УСР	Форма контроля знаний
		Лекции	Практические занятия	Семинарские занятия	Лабораторные занятия	Иное		
1	2	3	4	5	6	7	8	9
	5 семестр							
1.	Системное программирование в пользовательском режиме	0			0			
1.1	Оконный пользовательский интерфейс	2			0			
1.2	Динамически загружаемые библиотеки	2						

	Лабораторное занятие №1 Построение программы.	0			2			Защита лабораторной работы
	Лабораторное занятие №2 Построение основных структур данных.	0			2			Защита лабораторной работы
2.	Системное программирование в режиме ядра	0			0			
2.4	Драйверы	2			0			
	Лабораторное занятие №3 Обзор средств управления файлами.	0			2			Защита лабораторной работы
	Курсовой проект	0			0			Защита курсового проекта
	Итого за семестр	6			6			Экзамен
	Всего аудиторных часов	12						

[1]Темы учебного материала, не указанные в Учебно-методической карте, отводятся на самостоятельное изучение студента

ИНФОРМАЦИОННО-МЕТОДИЧЕСКАЯ ЧАСТЬ

Список литературы

Основная литература

1. Электронный учебно-методический комплекс по учебной дисциплине «Системное программирование» для специальности: I -53 01 02 «Автоматизированные системы обработки информации» [Электронный ресурс] / Белорусский национальный технический университет, Кафедра «Информационных технологий автоматизированных систем» ; сост. М. П. Ревотюк. – Минск : БНТУ, 2006.
2. Рихтер, Дж. Windows для профессионалов : создание эффективных Win32 приложений с учетом специфики 64-разрядной версии Windows / Дж. Рихтер. – пер. с англ. – 4-е изд. – Москва : Русская редакция ; СанктПетербург : Питер, 2001. – 752 с.
3. Руссинович, М. Внутреннее устройство Windows / М. Руссинович [и др.]. – пер. с англ. – 7-е изд. – Санкт-Петербург : Питер, 2018. – 944 с.
4. Уорд, Б. Внутреннее устройство Linux / Б. Уорд. – Санкт-Петербург : Питер, 2016. – 384 с.
5. Гордеев, А. В. Системное программное обеспечение / А. В. Гордеев, А. Ю. Молчанов. – Санкт-Петербург : Питер, 2002. – 736 с.
6. Таненбаум, Э. Современные операционные системы / Э. Таненбаум, Х. Бос. – 4-е изд. – Санкт-Петербург : Питер, 2015. – 1120 с.
7. Лав, Р. Linux. Системное программирование / Р. Лав. – 2- изд. – Санкт-Петербург : Питер, 2014. – 448 с.
8. Dabak, P. Undocumented Windows NT / P. Dabak, S. Phadke, M. Borate. – IDG Books Worldwide, Inc.; M&T Books, 1999. – 327 с.
9. Nebbett, G. Windows NT/2000 Native API Reference / G. Nebbett. – MTP, 2000. – 496 с.
10. Солдатов, В. П. Программирование драйверов Windows / В. П. Солдатов. – Москва : Бином-Пресс, 2006. – 576 с.
11. Комиссарова, В. Программирование драйверов для Windows / В. Комиссарова. – Санкт-Петербург : БХВ-Петербург, 2007. – 256 с.
12. Цирюлик, О. Расширения ядра Linux : драйверы и модули / О. Цирюлик. – Санкт-Петербург : БХВ, 2023. – 688 с.

Дополнительная литература

13. Зальцман, П. Пособие по программированию модулей ядра Linux / П. Зальцман [и др.] [Электронный ресурс]. – Режим доступа: https://habr.com/ru/companies/ruvds/articles/681880/. – Дата доступа: 17.05.2023.
14. Разработка, тестирование и развертывание драйверов / Microsoft Corporation [Электронный ресурс]. – Режим доступа:

https://learn.microsoft.com/ruru/windows-hardware/drivers/develop/. – Дата доступа: 17.05.2023.
15. The Undocumented Functions Microsoft Windows NT/2000 / NTAPI Undocumented Functions [Электронный ресурс]. – Режим доступа: http://undocumented.ntinternals.net/. – Дата доступа: 17.05.2023.
16. Сурков, К. Операционные системы и системное программирование / К. А. Сурков [Электронный ресурс]. – Режим доступа https://nezaboodka.by/assets/docs/OSiSP.2021-09-29.pdf. – Дата доступа: 17.05.2023.

Средства диагностики результатов учебной деятельности

Оценка уровня знаний студента производится по десятибалльной шкале в соответствии с критериями, утвержденными Министерством образования Республики Беларусь. Для оценки достижений студента рекомендуется использовать следующий диагностический инструментарий:

- защита выполненных в рамках лабораторных занятий заданий;
- защита курсового проекта;
- экзамен.

Перечень тем курсовых проектов

30. Мониторинг потока событий (Hooks)
31. Сетевое окружение (Windows Networking)
32. Мониторинг изменений файлов и каталогов (File I/O)
33. Работа с реестром (Registry)
34. Процессы и потоки (Processes and Threads)
35. Синхронизация процессов и потоков (Synchronization)
36. Обработка исключений (Structured Exception Handling)
37. Динамический обмен данными (DDEML)
38. Механизмы межпроцессных взаимодействий (IPC)
39. Панель управления (Control Panel Application)
40. Именованные каналы (Pipes)
41. Почтовые ящики (MailSlots)
42. Выгрузка системы (System Shutdown)
43. Получение сведений о системе (System Informatio)
44. Оконные интерфейсы (Windows Station and Desktops)
45. Интерфейс интерпретатора команд (Shell Library)
46. Хранитель экрана (Screen Saver)
47. Проецирование файлов (File Mapping)
48. Синхронный ввод/вывод и временные файлы (Files)
49. Асинхронный ввод вывод (Files)
50. Протоколирование событий (Event Logging)
51. Динамически загружаемые библиотеки (DLL)

52. Управление памятью (Memory Management)
53. Отладка (Debugging)
54. Буфер обмена (ClipBoard)
55. Управление энергосбережением (Power Management)
56. Сжатие данных (Data Decompression Library)
57. Таймеры (Timers)
58. Инструментальная библиотека (Tool Help Library)

Методические рекомендации по организации и выполнению самостоятельной работы студентов

При изучении дисциплины рекомендуется использовать следующие формы самостоятельной работы:

- составление тематической подборки литературных источников;
- проработка тем, вынесенных на самостоятельное изучение;
- подготовка курсового проекта по индивидуальным заданиям.

ОБЩИЕ ТРЕБОВАНИЯ К КУРСОВОМУ ПРОЕКТУ

Курсовой проект выполняется на кафедре «Информационные системы и технологии» по дисциплине «Системное программирование».

В системе профессиональной подготовки специалистов важное место занимает научно-исследовательская работа студентов, в частности такая форма её организации, как написание и защита курсовой работы.

Курсовой проект представляет собой логически завершенное и оформленное в виде текста произведение индивидуального научно теоретически-практического содержания, направленное на решение определенных проблем и задач в области изучаемых дисциплин.

Выполнение курсового проекта направлено на достижение следующих целей:

– систематизация, обобщение, закрепление и углубление теоретических и практических знаний по циклам дисциплин, изучаемых студентами в процессе их профессиональной подготовки в университете;

– совершенствование навыков применения полученных знаний для решения конкретных задачи, а также навыков самостоятельной работы с научной литературой и обработки результатов теоретических или экспериментальных исследований.

Тема курсового проекта утверждается на соответствующей кафедре, а задание на ее выполнение оформляется руководителем. Объектом проектирования является программное обеспечение для операционной системы. Задание на курсовой проект формируется так, чтобы студент получил навыки инженерной деятельности.

ЦЕЛЬ КУРСОВОГО ПРОЕКТА

Целью курсового проекта (КП) является применение теоретических и практических навыков, полученных в ходе обучения студентов по дисциплине «Операционные системы и системное программирование». Создание программного обеспечения для операционной системы.

ЗАДАНИЕ НА КУРСОВОЙ ПРОЕКТ

Основой для разработки проекта должно быть соответствующее задание по КП (ПРИЛОЖЕНИЕ Б), которое выдается студенту преподавателем в первые две недели обучения семестра, в котором учебным планом он предусмотрен. Задание содержит сведения о проекте с подписями исполнителя и руководителя, заверяется подписью заведующего кафедрой. Варианты заданий могут отличаться друг от друга как исходными данными, так и характером решаемых задач. Примеры тем по курсовому проекту приведены в разделе «ПРИМЕРНЫЙ ПЕРЕЧЕНЬ ТЕМ КУРСОВЫХ ПРОЕКТОВ».

ЗАЩИТА КУРСОВОГО ПРОЕКТА

Выполненный курсовой проект решением руководителя проектирования допускается к защите, о чем руководитель делает соответствующую надпись: «К

защите» на обложке пояснительной записки. Перед этим чертежи, приложения, презентация и пояснительная записка должны быть подписаны студентом-автором проекта. Защита курсового проекта проводится перед комиссией, которая формируется в составе не менее 2 человек с участием руководителя курсового проекта. Защита КП, выполненных по групповому заданию, производится в один день. Допускается открытая защита в присутствии всей учебной группы, где обучается автор курсового проекта. При защите может использоваться мультимедийное оборудование. Графическая часть может быть представлена на защите проекта в виде электронной презентации (9-15 слайдов) с распечаткой бумажного раздаточного материала. Наличие электронной презентации не исключает необходимость представления графической части на бумажном носителе, которая должна быть включена в расчетно- пояснительную записку.

При защите КП студент делает устное сообщение (защита) продолжительностью 8-10 минут, в котором показывает соответствие полученных результатов требованиям проекта. При этом следует выделить основные этапы выполнения проекта, отметить стандартные и оригинальные приемы решения поставленной задачи.

Вопросы, задаваемые студенту, могут касаться как содержания проекта, так и соответствующих разделов курса лекций.

При определении оценки за работу учитываются:

- владение материалом;
- полнота выполненных задач;
- обоснованность выбора технических и программных средств;
- оригинальность решения;
- оформление расчетно-пояснительной записки.

Результат итоговой аттестации в форме защиты проекта оценивается отметками в баллах по десятибалльной шкале. Положительными являются отметки не ниже 4 (четырех) баллов.

Студент, не представивший в установленный срок курсовой проект или не защитивший его, считается имеющим академическую задолженность.

ОБЩИЕ ТРЕБОВАНИЯ ПО ОФОРМЛЕНИЮ РАСЧЕТНО-ПОЯСНИТЕЛЬНОЙ ЗАПИСКИ

Тематика курсовых проектов определяется кафедрой. Студентам предоставляется право выбора темы курсового проекта. Студент может предложить свою тему, название которой должно быть уточнено научным руководителем.

Пояснительная записка должна включать описание изучаемого механизма системы, методов и приемов его использования, а также пример **демонстрационной программы**.

Курсовой проект включает:

1**. Титульный лист** (ПРИЛОЖЕНИЕ А), с указанием названия учебного заведения, кафедры, темы курсового проекта, ее автора, научного руководителя, года выполнения работы. Подпись руководителя ставится после проверки материалов проекта и свидетельствует о допуске проекта к защите. После защиты на титульном листе проставляется оценка результатов защиты проекта.

2**. Задание на курсовой проект** (ПРИЛОЖЕНИЕ Б), выдается руководителем. Задание содержит сведения о проекте с подписями исполнителя и руководителя, заверяется подписью заведующего кафедрой**.**

3**. Реферат,** краткие сведения о работе (½ стр.).

Слово **РЕФЕРАТ** записывают прописными буквами полужирным шрифтом по центру, страницу не нумеруют, но включают в общее количество страниц расчетно-пояснительной записки. Содержание реферата включает пять-шесть ключевых (значимых) слов, краткое и точное изложение результатов проекта, т. е. основных сведений и выводов, к которым пришел обучающийся**.**

4**. Содержание** (ПРИЛОЖЕНИЕ В), где указывается название и страницы размещения в работе введения, глав, параграфов, заключения, списка использованных источников, приложения и т. п. Слово **СОДЕРЖАНИЕ** записывают прописными буквами полужирным шрифтом по центру. Расположение заголовков в содержании должно точно отражать последовательность и соподчиненность разделов и подразделов в тексте расчетно-пояснительной записки.

5**. Перечень условных обозначений, символов, терминов** (при необходимости).

6. **Введение,** (1-2стр) в котором излагаются следующие разделы: ***актуальность темы работы*** (включает обоснование необходимости исследования решаемых вопросов); ***задачи и цель исследования*** (включает формулировку конкретных теоретических и практических задач исследования); **характеристика области применения**. Введение должно быть кратким и четким, не должно быть общих мест и отступлений, непосредственно не связанных с разрабатываемой темой. Объем введения не должен превышать двух страниц. Слово **ВВЕДЕНИЕ** записывают прописными буквами полужирным шрифтом по центру. Введение завершается фразой «Целью курсового проекта является …».

7**. Основная часть** (изложение соответствующего теме материала). Курсовой проект должен соответствовать стандартам Единой системы конструкторской документации (**ЕСКД**), Единой системы технологической документации (**ЕСТД**), Единой системы программной документации (**ЕСПД**), другим действующим техническим нормативным правовым актам. Приводимые по тексту сведения и решения должны сопровождаться ссылками на источник.

Использование заимствованных сведений без ссылок является плагиатом, свидетельствует о несамостоятельном выполнении работы, и служит основанием для недопуска курсового проекта к защите.

8. **Заключение** (1-2стр), в котором подводятся итоги исследования, обобщаются и формулируются **выводы**. Слово **ЗАКЛЮЧЕНИЕ** записывают прописными буквами полужирным шрифтом по центру.

9**. Список использованных источников**, записывают прописными буквами полужирным шрифтом по центру строки, в котором перечень источников с указанием их порядкового номера приводится в порядке очередности цитирования либо в алфавитном порядке. Источник – это книга, журнал и т. п. Ссылки на сайты и имена файлов допустимо использовать только как дополнение к названию документа. При приведении в пояснительной записке теоретических или справочных сведений обязательно приводится ссылка на источник (номер ссылки в квадратных скобках – [5]). Список использованных источников оформляется отдельным разделом. Использование заимствованных сведений без ссылок является плагиатом, свидетельствует о несамостоятельном выполнении работы, и служит основанием для недопуска курсового проекта к защите.

Приложения (обязательно) включают схемы, графики, таблицы, презентацию и т.д.

В приложения расчетно-пояснительной записки рекомендуется выносить информацию, имеющую справочное или второстепенное значение, но необходимую для более полного освещения темы проекта, или помещать отдельные материалы (распечатки программ и т. п.) для удобства работы с текстом расчетно-пояснительной записки. Приложениями могут быть математические формулы, программные коды, номограммы, вспомогательные вычисления и расчеты, описания алгоритмов и программ, технические характеристики различных устройств, спецификации, схемы, рисунки и т.п. Допускается использовать в качестве приложений конструкторские документы. Все приложения включают в общую нумерацию страниц. В тексте расчетно-пояснительной записки на все приложения должны быть ссылки. Приложения располагают в порядке ссылок на них в тексте.

Приложения обозначают заглавными буквами русского алфавита, за исключением букв Ё, З, Й, О, Ч, Ъ, Ы, Ь. Если в расчетно-пояснительной записке одно приложение, оно также должно быть обозначено: ПРИЛОЖЕНИЕ А. Каждое приложение начинают с новой страницы. Вверху по центру страницы пишут слово ПРИЛОЖЕНИЕ прописными буквами и его буквенное обозначение. Еще ниже по центру размещают заголовок, который записывают с прописной буквы.

Каждый раздел работы (введение, главы, заключение и т. д.) должен иметь заглавие.

Расчетно-пояснительную записку выполняют с применением печатающих и графических устройств вывода ПЭВМ. Текст располагают на одной стороне листа формата А4 с соблюдением размеров полей и интервалов. Объем курсового проекта (без учета списка использованных источников и приложений) 20 – 30 страниц текста через 1,5 интервала, шрифт – 13-14 pt Times New Roman, (поля страниц: левое – 3 см., правое – 2 см., верхнее и нижнее – 1 см.). Номера разделов, подразделов, пунктов и подпунктов следует выделять полужирным шрифтом. Заголовки разделов рекомендуется оформлять полужирным шрифтом размером 14-16 пунктов, а подразделов – полужирным шрифтом 13 - 14 пунктов. Номера страниц обозначаются в правом верхнем углу (титульный лист не нумеруется).

Схемы – формат А4, А3, А2 либо А1 (*по согласованию с научным руководителем*). Презентация – формат А4.

ПРИЛОЖЕНИЕ А
Титульный лист (пример)

МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ
Белорусский национальный технический университет
Международный институт дистанционного образования

Кафедра "Информационные системы и технологии"

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

к курсовому проекту

по дисциплине: «Операционные системы и системное прграммирование»

_____

(наименование темы)

Исполнитель Иванов И.И.

Руководитель Бумай А.Ю.

Минск 2024

ПРИЛОЖЕНИЕ Б
Лист-задания (пример)

БЕЛОРУССКИЙ НАЦИОНАЛЬНЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Факультет: Международный институт дистанционного образования

Утверждаю
Ст. преподаватель _____ Русак Л.В.
(подпись) (фамилия, инициалы)
« » 2024 г.

Задание на курсовой проект (курсовую работу)

Обучающемуся Иванову Иван Иванычу
(фамилия, собственное имя, отчество)

группа 41703120

1. Тема курсового проекта
(указать: курсового проекта или курсовой работы)
Динамически загружаемые библиотеки (DLL) (вариант 22)

2. Сроки сдачи законченного проекта (работы) 31.05.2024 г.

3. Исходные данные к курсовому проекту
(указать: к курсовому проекту или курсовой работе)

Создать библиотеку функций для работы с двумерными матрицами и скомпоновать ее в виде DLL. Набор функций должен включать: транспонирование матриц, вычисление определителя квадратной матрицы, определение обратной матрицы, умножение двух матриц, сложение матриц, умножение матрицы на скаляр.

Разработать программу, демонстрирующую использование функций из библиотеки. В программе использовать динамическое связывание.

4. Содержание пояснительной записки (перечень вопросов, которые подлежат разработке)

титульный лист;

содержание;

реферат;

введение;

задание (актуальность задачи, область применения);

описание разработанных классов, диаграммы классов;

описание программы (подробно прокомментированный код);

методика испытаний (тестирование, скриншоты работы приложения);

выводы (заключение);

список использованной литературы.

приложения

5. Перечень графического материала (с точным указанием обязательных чертежей и графиков)

копии консольного окна;

содержимое текстовых файлов;

диаграммы классов.

6. Дата выдачи задания 31.01.2024 г.

7. Примерный календарный график выполнения курсового проекта

(указать: курсового проекта или курсовой работы)

с указанием сроков выполнения и трудоёмкости отдельных этапов

Анализ литературных источников	17.02	10%
Выбор методов и алгоритмов	05.03	20%
Написание кода	28.03	50%
Отладка программы	15.04	60%
Тестирование	15.05	80%
Написание отчета	29.05	100%

Руководитель курсового проекта _____ Бумай А.Ю.

(указать: курсового проекта или курсовой работы) (подпись) (фамилия, инициалы)

Иванов И.И.

Подпись обучающегося _____ _____

(подпись) (фамилия, инициалы)

Дата _____

ПРИЛОЖЕНИЕ В
Примерный образец структуры и содержания пояснительной записки к курсовому проекту

Титульный лист
Лист задания
Реферат
Содержание
Перечень условных обозначений и сокращений
Введение
1 ОБЩИЕ СВЕДЕНИЯ
1.1 Назначение DLL-библиотек
1.2 Использование DLL-библиотек
1.3 Цели создания DLL-библиотек
1.4 Цели курсового проекта
2 РАЗРАБОТКА БИБЛИОТЕКИ DLL
2.1 Структура DLL-библиотеки (*схема, описание разработанных классов, диаграммы классов*)
2.2 Функции DLL-библиотеки
2.3 Режим функционирования DLL-библиотеки
2.4 Надежность DLL-библиотеки
2.5 Эксплуатация DLL-библиотеки
3 ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ
3.1 Программное обеспечение для работы с DLL-библиотекой (*выбор*)
3.2 Решения по размещению DLL-библиотеки
Заключение
Список использованных источников
Перечень нормативно-технических документов
Приложения (схемы, таблицы, рисунки, чертежи, презентация, эксплуатационные документы)

СПИСОК ЛИТЕРАТУРНЫХ ИСТОЧНИКОВ

1. Электронный учебно-методический комплекс по учебной дисциплине «Системное программирование» для специальности: I -53 01 02 «Автоматизированные системы обработки информации» [Электронный ресурс] / Белорусский национальный технический университет, Кафедра «Информационных технологий автоматизированных систем» ; сост. М. П. Ревотюк. – Минск : БНТУ, 2006.
2. Руссинович, М. Внутреннее устройство Windows / М. Руссинович [и др.]. – 7-е изд. – СПб. : Питер, 2018. – 944 с.
3. Рихтер, Дж. Windows для профессионалов. Создание эффективных WIN32-приложений с учетом специфики 64-разрядной версии Windows / Дж. Рихтер. – СПб. : Питер, 2001. – 752 с.
4. Побегайло, А. П. Системное программирование в Windows / А. П. Побегайло. – СПб. : БХВ-Петербург, 2006. – 1056 с.
5. Hewardt, M. Advanced Windows Debugging / Hewardt M., D. Pravat. – Boston : Addison-Wesley Professional, 2007. – 840 с.
6. Шилдт, Г. Полный справочник по C++ / Г. Шилдт. – 4-е изд. – М. : Издательский дом «Вильямс», 2006. – 800 с.
7. Таненбаум, Э. Современные операционные системы / Э. Таненбаум, Х. Бос. – 4-е изд. – СПб. : Питер, 2015. – 1120 с.
8. Беляев, А. Централизованная обработка исключений / А. Беляев // RSDN Magazine, 25.09.2004. – [б.м.].
9. Лохас П. Debugging: Развертывание сервера отладочной информации / П. Лохас // Habrahabr [Электронный ресурс]. – 2010. – Режим доступа : http://habrahabr.ru/blogs/development/89094/. – Дата доступа : 29.11.2023.
10. Лохас П. Debugging: Введение в postmortem debugging / П. Лохас // Habrahabr [Электронный ресурс]. – 2010. – Режим доступа : http://habrahabr.ru/blogs/development/89220/. – Дата доступа : 29.11.2023.
11. Харт, Дж. М. Системное программирование в среде Windows / Дж. М. Харт. – 3-е изд. – М. : Издательский дом «Вильямс», 2005. – 592 с.
12. Рихтер, Дж. Windows для профессионалов : создание эффективных Win32 приложений с учетом специфики 64-разрядной версии Windows / Дж. Рихтер. – пер. с англ. – 4-е изд. – Москва : Русская редакция ; СанктПетербург : Питер, 2001. – 752 с.
13. Руссинович, М. Внутреннее устройство Windows / М. Руссинович [и др.]. – пер. с англ. – 7-е изд. – Санкт-Петербург : Питер, 2018. – 944 с.
14. Уорд, Б. Внутреннее устройство Linux / Б. Уорд. – Санкт-Петербург : Питер, 2016. – 384 с.

15. Гордеев, А. В. Системное программное обеспечение / А. В. Гордеев, А. Ю. Молчанов. – Санкт-Петербург : Питер, 2002. – 736 с.
16. Таненбаум, Э. Современные операционные системы / Э. Таненбаум, Х. Бос. – 4-е изд. – Санкт-Петербург : Питер, 2015. – 1120 с.
17. Лав, Р. Linux. Системное программирование / Р. Лав. – 2- изд. – Санкт-Петербург : Питер, 2014. – 448 с.
18. Dabak, P. Undocumented Windows NT / P. Dabak, S. Phadke, M. Borate. – IDG Books Worldwide, Inc.; M&T Books, 1999. – 327 с.
19. Nebbett, G. Windows NT/2000 Native API Reference / G. Nebbett. – MTP, 2000. – 496 с.
20. Солдатов, В. П. Программирование драйверов Windows / В. П. Солдатов. – Москва : Бином-Пресс, 2006. – 576 с.
21. Комиссарова, В. Программирование драйверов для Windows / В. Комиссарова. – Санкт-Петербург : БХВ-Петербург, 2007. – 256 с.
22. Цирюлик, О. Расширения ядра Linux : драйверы и модули / О. Цирюлик. – Санкт-Петербург : БХВ, 2023. – 688 с.
23. Зальцман, П. Пособие по программированию модулей ядра Linux / П. Зальцман [и др.] [Электронный ресурс]. – Режим доступа: https://habr.com/ru/companies/ruvds/articles/681880/. – Дата доступа: 17.05.2023.
24. Разработка, тестирование и развертывание драйверов / Microsoft Corporation [Электронный ресурс]. – Режим доступа: https://learn.microsoft.com/ruru/windows-hardware/drivers/develop/. – Дата доступа: 17.05.2023.
25. The Undocumented Functions Microsoft Windows NT/2000 / NTAPI Undocumented Functions [Электронный ресурс]. – Режим доступа: http://undocumented.ntinternals.net/. – Дата доступа: 17.05.2023.
26. Сурков, К. Операционные системы и системное программирование / К. А. Сурков [Электронный ресурс]. – Режим доступа https://nezaboodka.by/assets/docs/OSiSP.2021-09-29.pdf. – Дата доступа: 17.05.2023.