Министерство образования Республики Беларусь
БЕЛОРУССКИЙ НАЦИОНАЛЬНЫЙ
ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Кафедра «Технология и методика преподавания»

Электронный учебно-методический комплекс
по учебной дисциплине

«КОНСТРУИРОВАНИЕ ПРОГРАММ И ЯЗЫКИ ПРОГРАММИРОВАНИЯ (раздел 1. Язык программирования С++)»

для студентов специальности
1-08 01 01 «Профессиональное обучение
(по направлениям)» направления специальности
1-08 01 01-07 «Профессиональное обучение
(информатика)»

Минск
БНТУ
2020

Составитель:
А.А. Дробыш

К 29 Конструирование программ и языки программирования (раздел 1. Язык программирования С++): учебно-методический комплекс / сост.: А.А. Дробыш. – Минск: БНТУ, 2020. – 232 с.

В теоретической части представлен конспект лекций по изучению языка программирования С++.

В практической части приведены указания для выполнения лабораторных работ по изучению языка программирования С++. Приведены основные теоретические сведения, сформулированы цели лабораторных работ, описан ход работ, перечислены контрольные вопросы к защите лабораторных работ.

Пособие предназначено для студентов инженерно-педагогического факультета БНТУ, а также может быть использовано для самостоятельного изучения студентами университета.

Оглавление

Раздел 1. Введение в предмет
Тема 1.1 Основные понятия и определения

Программирование – это техническая творческая деятельность, цель которой заключается в решении важных для человека задач или выполнении определенных действий с помощью компьютера.

Программирование – процесс и искусство создания компьютерных программ с помощью языков программирования.

Язык программирования – формальный язык, предназначенный для записи компьютерных программ. Язык программирования определяет набор лексических, синтаксических и семантических правил, определяющих внешний вид программы и действия, которые выполнит исполнитель (обычно – ЭВМ) под её управлением.

Программа в целом аналогична предложению, описывающему последовательность действий над заданными предметами с целью получения результата. Можно сказать, что компьютерная программа – один из способов реализации понятия алгоритма, а язык программирования – средство описания алгоритмов. Программа – описание на языке программирования структур данных и алгоритма решения задачи, автоматически переводимое, при помощи специальной программы-транслятора (компилятора или интерпретатора), на язык команд компьютера для последующего выполнения. Программа, в отличие от абстрактного алгоритма, имеет данные – собственные элементы, над которыми она совершает действия, и которые являются ее составной частью. Данные (синтаксически) являются аналогом существительных (объектов, над которыми производятся действия), набор операций – аналогом глаголов (выполняемых действий). Алгоритмическая компонента программы – описание последовательности выполняемых действий – обычно состоит из операторов, задающих эту последовательность действий, и

базируется на наборе операций над данными (арифметические операции, присваивание, проверка значения переменной и т.п.), соответствующем системе команд процессора данного компьютера.

Структура данных – вид представления данных в программе, описание точки зрения пользователя на представление данных. Выбор подходящего представления данных – один из основных вопросов при проектировании программы. Неправильное представление данных может сделать программу ненадежной, неэкономичной, сложной и даже вообще неадекватной задаче. При решении задачи на компьютере, анализе исходных данных программы и ее результата, необходимо выбирать экономичный алгоритм решения, который и определит представление исходных, промежуточных и конечных данных.

В самых общих чертах алгоритм – это однозначное описание последовательности выполнения исполнителем действий из заданного набора, позволяющее получить требуемый результат за конечное число шагов. Однако в определении алгоритма не говорится над чем производятся действия, выполняемые в нем.

Определение программы дано в формуле: «Программа = данные + алгоритм». В ней данные и алгоритм являются двумя взаимозависимыми элементами. Если данные в какой-то мере обладают свойствами пространства (объем, протяженность), то алгоритм – свойствами времени (эффективность, быстродействие); тезис «проигрывая в пространстве, выигрываем во времени» здесь также уместен: эффективность программ может быть принципиально повышена за счет использования дополнительных структур, данных в памяти.

Тема 1.2 Язык программирования C/C++

С. Си придумали инженеры. Если Паскаль придумал учёный, то Си придумали Керниган и Ритчи, они работали инженерами в Bell. Как это произошло? В то время на этих языках (лектор указывает на Fortran, COBOL, Algol) ничего системного написать было нельзя. Что такое «системное»? Например, операционную систему, драйвера какие-нибудь, ещё что-то. Эти языки предназначались для математических расчётов, для бизнес-расчётов, для всего такого. А всё остальное писали на Ассемблере. Были какие-то языки, они сейчас умерли, то есть язык Си появился не сразу от Ассемблера, а через какие-то промежуточные вещи.

История появления языка C++

С. Си придумали инженеры. Если Паскаль придумал учёный, то Си придумали Керниган и Ритчи, они работали инженерами в Bell. В то время ничего системного написать было нельзя. Что такое «системное»? Например, операционную систему, драйвера какие-нибудь, ещё что-то. Эти языки предназначались для математических расчётов, для бизнес-расчётов, для всего такого. А всё остальное писали на Ассемблере. Были какие-то языки, они сейчас умерли, то есть язык Си появился не сразу от Ассемблера, а через какие-то промежуточные вещи. В соответствие с потребностью решать такую главную задачу как написание кода ОС, целью авторов было создание удобного и полезного языка.

Эти критерии, конечно, учитывались и при разработке множества других языков. Но разработка других языков преследовала и другие цели, например:

– *Pascal* – язык на основе которого можно было бы обучать фундаментальным основам и принципам программирования;

– *Basic* – синтаксис языка близок к английскому языку; предназначен для быстрого освоения программирования непрофессионалами.

В эпоху развития объектно-ориентированного программирования и появления объектно- ориентированных языков и такой универсальный язык как С тоже получил развитие в этой области. Новый язык, включающий в себя объектно-ориентированное расширение, получил название С++ («си плас плас» от английского “c plus plus”; «си плюс плюс»).

Язык С имеет огромное количество недостатков (ну просто вообще огромное) – на нём можно делать вообще всё, в том числе стрелять себе в ногу, стрелять себе в ногу с выдумкой, в другую ногу, одной ногой стрелять в другую ногу, в общем – что угодно делать. Но при этом некоторые архитектурные вещи там делаются довольно сложно – опять же, как и в Ассемблере, нам приходится всё время следить, где мы, чего и какую память выделили; она там всё время «течёт» куда-то эта память -– то есть мы выделили, забыли удалить, удалили не то, вылезли за пределы памяти, в общем – огребли кучу проблем.

С++ создавался сначала как набор дополнений к языку С, который облегчит разработку. В то время стало модно объектно-ориентированное программирование и люди решили, что всё можно описать в виде иерархии, то есть – есть у вас мячик (абстрактный), вы от него наследуете футбольный мяч, волейбольный мяч, ещё один абстрактный мяч. Тогда было модно, что «мы сейчас пишем всё в виде какой-то иерархии, и всё будет хорошо, жизнь наладится, всё станет прекрасно и всё». С++ в каком-то смысле реализовывал этот объектный подход – это не был первый язык объектно-ориентированного программирования, но он стал достаточно популярным. При этом С++ сохранял почти полную совместимость (на тот момент) с языком С, программа, написанная на Си в 99% случаев успешной компилировалась как С++-ная и даже работала также. Это было задумано, чтобы с Си было легко перейти на С++.

Достоинства языка С++

К основным достоинствам языка С++ необходимо отнести следующие моменты:

– Актуальный язык. Он включает в себя управляющие конструкции, рекомендуемые теоретическим и практическим программированием.

– Эффективный. Структура позволяет наилучшим образом использовать возможности современных ЭВМ. Программы отличаются компактностью и быстротой исполнения.

– Переносимый или мобильный. Программа, написанная для одной вычислительной системы, может быть перенесена почти без изменений на другую. Компиляторы реализованы почти на 40 типах вычислительных систем, начиная от 8-ми разрядных процессоров и кончая CRAY-1 один из мощных суперкомпьютеров.

– Мощный и гибкий. Большая часть ОС UNIX написана на С.

– Удобный язык. Достаточно структурирован, чтобы поддерживать хороший стиль программирования, в то же время не накладывает больших ограничений

– Язык «компилирующего» типа.

Структура программы

Исходная программа на языке С состоит из следующих частей: директив препроцессора; указаний компилятору; объявлений; определений.

Эти части имеют разное предназначение в тексте программы:

Директивы – специфицируют действия препроцессора по преобразованию текста программы перед компиляцией.

Указания – это специальные инструкции, которым компилятор С++ следует во время компиляции.

Объявления – задают имя и атрибуты данных, их начальные значения явно или по умолчанию.

Различия

Модификаторы доступа – это слова public, private и protected. В С вместо них была внимательность программиста: public – значит с этими полями делаю, что хочу; private – значит к этим полям обращаюсь только с помощью методов этой структуры; protected – то же, что public, но еще можно обращаться из методов унаследованных структур.

То, что в С++ - наследование, в С – это просто структура в структуре. При программировании в стиле С++ применяются такие красивые и звучные слова, как «класс Circle порожден от класса Point» или «класс Point наследуется от класса Circle и является производным от него». На практике все это словоблудие заключается в том, что структура Point – это первое поле структуры Circle.

При этом реальных усовершенствований два. Первое – поля Point считаются так же и полями Circle, в результате доступ к ним записывается короче, чем в С. Второе – в обоих структурах можно иметь функции-методы, у которых имена совпадают с точностью до имени структуры. Например, Point::paint и Circle::paint. Следствие – не надо изобретать имена вроде Point_paint и Circle_paint, как это было в С, а префиксы Point:: и Circle:: в большинстве случаев можно опускать.

В С++ появились две новые операции: new и delete. В первую очередь это – сокращения для распространенных вызовов функций malloc и free:

В стиле С:

```
Point *p = (Point*) malloc(sizeof(Point));
free(p);
```

В стиле С++:

```
Point *p = new Point;
delete p;
```

При вызове new автоматически вызывается конструктор, а при вызове delete – деструктор. Так что нововведение можно описать формулой: new = malloc + конструктор, delete = free + деструктор.

Когда программируешь в стиле C, после того, как завел новую переменную типа структуры, часто надо ее инициализировать и об этом легко забыть. А перед тем как такая структура закончит свое существование, надо ее почистить, если там внутри есть ссылки на какие-то ресурсы. Опять-таки легко забыть.

В C++ появились функции, которые вызываются автоматически после создания переменной структуры (конструкторы) и перед ее уничтожением (деструкторы). Во всех остальных отношениях это – обычные функции, на которые наложен ряд ограничений. Некоторые из этих ограничений ничем не оправданы и мешают: например, конструктор нельзя вызвать напрямую (дестркутор, к счастью, можно). Нельзя вернуть из конструктора или деструктора значение. Что особенно неприятно для конструктора. А деструктору нельзя задать параметры.

При программировании на C часто бывает так, что имеется несколько вариантов одной и той же структуры, для которых есть аналогичные функции. Например, есть структура, описывающая точку (Point) и структура, описывающая окружность (Circle). Для них обоих часто приходится выполнять операцию рисования (point). Так что, если у нас есть блок данных, где перемешаны точки, окружности и прочие графические примитивы, то перед нами стоит задача быстро вызвать для каждого из них свою функцию рисования.

Обычное решение – построить таблицу соответствия «вариант структуры – фукция». Затем берется очередной примитив, определяется его тип, и по таблице вызывается нужная функция. В C этот метод применять довольно нудно из-за того, что, во-первых, надо строить эту таблицу, а во-

вторых, внутри структур заводить поле, сигнализирующее о том, какого она типа, и следить за тем, чтобы это поле содержало правильное значение.

В C++ всем этим занимается компилятор: достаточно обозначить функцию-метод как virtual, и для всех одноименных функций будет создана таблица и поле типа, за которыми следить будет опять-таки компилятор. Вам останется только пользоваться ими: при попыке вызвать функцию с таким именем, будет вызвана одна из серии одноименных функций в зависимости от типа структуры.

Многие программисты изучали C на основе языка Pascal. В Pascal есть возможность возвращать из функции больше одного параметра. Для этого применялось магическое слово «var». В C для того, чтобы сделать то же самое, приходилось расставлять в тексте уйму символов «*».

Разработчики C++ вняли стонам несчастных программистов и ввели слово var. А чтобы все это выглядел ооригинально, "var" они переименовали в "&" и назвали «ссылкой». Это вызвало большую путаницу, так как в C уже были понятия «указатель» (та самая звездочка) и «адрес» (обозначался тем же символом &), а понятие "ссылка" звучит тоже как что-то указующе-адресующее.

С одной стороны, использование ссылок намного сокращает текст программы. Но есть и неприятности. Во-первых, вызов функции, в которой параметр является ссылкой, выглядит так же, как вызов с обычным параметром. В результате «на глаз» незаметно, что параметр может измениться. А в C это заметно по значку &. Во-вторых, многочисленные звездочки в C напоминают программисту о том, что каждый раз выполняется дополнительная операция * разыменования указателя. Что побуждает сделать разумную оптимизацию. В C++ эти операции остаются незамеченными.

Раздел 2. Основы программирования на языке С
Тема 2.1. Структура программы

Вы можете воспользоваться шаблоном проекта для создания базовой структуры программы, меню, панелей инструментов, значков, ссылок и инструкций #include, подходящих для разрабатываемого проекта. Visual Studio содержит несколько видов шаблонов проектов Visual C++ и предоставляет для многих из них мастера, позволяющие настраивать проекты во время их создания.

Использовать шаблон при создании проекта необязательно, однако в большинстве случаев это гораздо эффективнее, так как проще изменять имеющиеся файлы и структуру проекта, чем создавать их «с нуля».

Вы можете создать проект на языке из разряда С, используя шаблоны проектов C++. Найдите в созданном проекте файлы с расширением .CPP и измените его на С. Затем на странице Свойства проекта разверните узлы Свойства конфигурации, C/C++ и выберите Дополнительно. Измените значение параметра на С код (/TC).

Файлы, создаваемые мастером приложений, содержат комментарии TODO в форме комментариев. Это области, в которых можно задать собственный код или использовать дополнительные мастера для разработки приложения. Список комментариев TODO нового проекта выводится в Списке задач.

Для каждого типа проектов Visual C++ имеется мастер приложений, помогающий быстро и легко создавать проекты по шаблону. Чтобы открыть мастер приложений, воспользуйтесь диалоговым окном Новый проект и укажите в нем свойства проекта, такие как имя проекта и каталог или решение, в котором будет размещен проект. Каждый мастер помогает создавать соответствующие проекты:

1) ATL:

– Проект ATL.

2) Среда CLR:
– Библиотека классов, консольное приложение CLR;
– пустой проект CLR.
3) Общие:
– Пустой проект;
– Настраиваемый мастер;
– Проект, использующий файл makefile;
– Проект общих элементов.
4) MFC:
– Приложение MFC;
– Элемент управления ActiveX библиотеки MFC;
– Библиотека DLL MFC.
5) Тест:
– Управляемый тестовый проект;
– Проект машинного модульного теста.
6) Win32:
– Консольное приложение Win32;
– проект Win32.

Библиотека классов FCL – статический компонент каркаса: все языки программирования среды используют классы одной библиотеки.

Типы каркаса покрывают все множество встроенных типов, встречающихся в языках программирования. Типы языка программирования проецируются на соответствующие типы каркаса. Тип, называемый в языке Visual Basic – Integer, а в языке C# - int, проецируется на один и тот же тип каркаса t32. В каждом языке программирования наряду с «родными» для языка названиями типов разрешается пользоваться именами типов, принятыми в каркасе.

Число классов библиотеки FCL велико (несколько тысяч). Поэтому понадобился способ их структуризации. Логически классы с близкой функциональностью объединятся в группы, называемые пространством имен (Namespace). Для динамического компонента CLR физической единицей,

объединяющей классы и другие ресурсы, является сборка (assembly).

Основным пространством имен библиотеки FCL является пространство System, содержащей как классы, так и другие вложенные пространства имен. Так уже упоминавшийся примитивный тип int32 непосредственно вложен в пространство имен System и его полное имя, включающее имя пространства – System.int32.

В пространство System вложен целый ряд других пространств имен. Например, в пространстве System.Collections находятся классы и интерфейсы, поддерживающие работу с коллекциями объектов – списками, очередями, словарями.

Общеязыковая исполнительная среда CLR – динамический компонент каркаса.

Компиляторы языков программирования, включенные в Visual Studio .Net, создают модули на промежуточном языке MSIL (Microsoft Intermediate Language), называемом далее просто – IL. Фактически компиляторы создают так называемый управляемый модуль – переносимый исполняемый файл (Portable Executable или PE-файл). Этот файл содержит код на IL и метаданные – всю необходимую информацию как для CLR, так и конечных пользователей, работающих с приложением. В зависимости от выбранного типа проекта PE-файл может иметь уточнения exe, dll, mod или mdl.

PE-файл, имеющий уточнение exe, хотя и является exe файлом, но это не совсем обычный исполняемый Windows файл. При его запуске он распознается как специальный PE-файл и передается CLR для обработки. Исполнительная среда начинает работать с кодом, в котором специфика исходного языка программирования исчезла. Код на IL начинает выполняться под управлением CLR (по этой причине код

называется управляемым). Исполнительную среду можно рассматривать, как своеобразную виртуальную IL-машину.

Отделение каркаса от студии явилось естественным шагом. Каркас Framework .Net перестал быть частью студии, а стал надстройкой над операционной системой. Теперь компиляция и создание PE модулей на IL отделено от выполнения, и эти процессы могут быть реализованы на разных платформах. В состав CLR входят трансляторы JIT (Just In Time Compiler), которые и выполняют трансляцию IL в командный код той машины, на которой установлена и функционирует исполнительная среда CLR.

Переносимый исполняемый PE-файл является самодокументируемым файлом и, как уже говорилось, содержит как код, так и метаданные, описывающие код. Файл начинается с манифеста и содержит описание всех классов, хранимых в PE-файле, их свойств, методов, всех аргументов этих методов – всю необходимую CLR информацию. Поэтому помимо PE-файла не требуется никаких дополнительных файлов, записей в реестр, вся нужная информация извлекается из самого файла. Среди классов библиотеки FCL имеется класс Reflection, методы которого позволяют извлекать необходимую информацию.

У CLR есть свое видение того, что представляет собой тип. Есть формальное описание общей системы типов CTS – Common Type System. В соответствие с этим описанием каждый тип помимо полей, методов и свойств может содержать и события. При возникновении событий в процессе работы с тем или иным объектом данного типа посылаются сообщения, которые могут получать другие объекты. Механизм обмена сообщениями основан на делегатах – функциональном типе.

Структура программы, написанной на языке C++, выглядит следующим образом:

подключение головных файлов

запись макроопределений
описание глобальных переменных
int main( ) // заголовок главной функции
{
описание переменных
текст программы//алгоритм
return1; // завершение работы функции int main( )
}

1. Файл с программой на языке С++ вначале обрабатывает препроцессор, который распознаёт директивы препроцессора (каждая из них начинается с символа#)и выполняет их. Одной из таких директив является директива #include, которая используется для подключения головных файлов. Осуществляется эта возможность при помощи команды

#include< имя головного файла>

Подключение головных файлов необходимо для возможности работы с библиотечными (определёнными разработчиками языка) функциями, а также для возможности работы с функциями, описанными разработчиками программы и помещёнными в некоторый головной файл.

Некоторые библиотечные головные файлы:

stdio.h – функции ввода-вывода информации, работы с файлами и др.;

math.h – математические функции;

string.h – работа со строками;

stdlib.h – работа с динамической памятью.

Запись макроопределений выполняется с помощью директивы:

#define <имя_макропеременной> <определение_макропеременной>

При записи этой директивы между #define, <имя макропеременной> и <определение макропеременной> должны быть пробелы. Именем макропеременной может быть

любая последовательность символов, не начинающаяся и не содержащая специальных символов. Под определением макропеременной понимается любая константа (числовая), любое арифметическое выражение, содержащее константу и имена макропеременных ранее определённых. В любом месте текста программы, где будет встречаться имя макропеременной, его будет заменять соответствующее макроопределение, описанное после директивы #define.

Ресурсы – это элементы интерфейса, предоставляющие информацию пользователю. К ресурсам относятся точечные рисунки, значки, панели инструментов и курсоры. Некоторые ресурсы можно использовать для выполнения действия, такого как выбор в меню или ввод данных в диалоговом окне.

Таблица 2.1.1 – Стурктура проекта

Имя файла	Расположение каталога	Расположение в обозревателе решений	Описание
1	*2*	3	4
Имя_проекта.rc	*Имя_проекта*	Исходные файлы	Файл скрипта ресурса для проекта. В зависимости от типа проекта и выбранной для него поддержки (например, панели инструментов, диалоговые окна или HTML), файл скрипта ресурса содержит следующее: – Определение меню по умолчанию. – Таблицы сочетаний клавиш и строк. – Диалоговое окно **О программе** по умолчанию. – Прочие диалоговые окна. – Файл значка (res*Имя_проекта*.ico). – Сведения о версии. – Растровые изображения. – Панель инструментов. – HTML-файлы. Файл ресурсов включает файл Afxres.rc для стандартных ресурсов Microsoft Foundation Class.

1	*2*	3	4
Resource.h	*Имя_проекта*	Файлы заголовка	Файл заголовка ресурсов, который содержит определения для ресурсов, используемых в проекте
Имя_проекта.rc2	*Имя_проекта*\res	Исходные файлы	Файл скрипта с дополнительными ресурсами, используемыми в проекте. RC2-файл можно включить в RC-файл проекта. RC2-файл удобно использовать для включения ресурсов, используемых в нескольких разных проектах. Вместо того чтобы создать одни и те же ресурсы несколько раз для различных проектов, можно поместить их в RC2-файл и включить его в главный RC-файл
Имя_проекта.def	*Имя_проекта*	Исходные файлы	Файл определения модуля для проекта DLL. Для элемента управления он предоставляет имя и описание элемента управления, а также размер кучи времени выполнения
Имя_проекта.ico	*Имя_проекта*\res	Файлы ресурсов	Файл значка для проекта или элемента управления. Этот значок отображается, когда приложение свернуто. Он также используется в поле **О программе** приложения. По умолчанию MFC предоставляет значок MFC, а ATL – значок ATL
*Имя_проекта*Doc.ico	*Имя_проекта*\res	Файлы ресурсов	Файл значка для проекта MFC, который включает поддержку архитектуры document/view
Toolbar.bmp	*Имя_проекта*\res	Файлы ресурсов	Файл точечного рисунка, представляющий приложение или элемент управления в панели инструментов или на палитре. Этот точечный рисунок включается в файл ресурсов проекта. Первичная панель инструментов и строка состояния создаются в классе **CMainFrame**
ribbon.mfcribbon-ms	*Имя_проекта*\res	Файлы ресурсов	Файл ресурсов, содержащий код XML, который определяет кнопки, элементы управления и атрибуты на ленте

Тема 2.2. Система базовых типов данных

В алфавит языка С++ входят:

– прописные и строчные буквы латинского алфавита;
– цифры 0 1 2 3 4 5 6 7 8 9;
– специальные знаки: " { } , | [ ] ( ) + - % \ ; ‘ : ? < = > _ ! & # ~ ^ . *

Таблица 2.2.1 – Диапазон значений типов данных

Тип данных	Объем занимаемой оперативной памяти в байтах	Диапазон значений
1	2	3
char	1	–128 ÷ 127
unsigned char	1	0 ÷ 255
signed char	1	-128 ÷ 127
int	4	–2147483648 ÷ 2147483647
unsigned int	4	0 ÷ 4294967295
signed int	4	-2147483648 ÷ 2147483647
short int	2	-32768 ÷ 32767
unsigned short int	2	0 ÷ 65535
signed short int	2	–32768 ÷ 32767
long int	4	-2147483648 ÷ 2147483647
unsigned long int	4	0 ÷ 4294967295
signed long int	4	-2147483648 ÷ 2147483647

1	2	3
float	4	3,4e-38 ÷ 3.4e+38
double	8	1.7e-308 ÷ 1.7e+308
long double	10	3.4e-4932 ÷ 1.1e+4932

Тип данных bool

Первый в таблице – это тип данных bool – целочисленный тип данных, так как диапазон допустимых значений – целые числа от 0 до 255. Но как Вы уже заметили, в круглых скобочках написано – логический тип данных, и это тоже верно. Так как bool используется исключительно для хранения результатов логических выражений. У логического выражения может быть один из двух результатов true или false. true – если логическое выражение истинно, false – если логическое выражение ложно.

Но так как диапазон допустимых значений типа данных bool от 0 до 255, то необходимо было как-то сопоставить данный диапазон с определёнными в языке программирования логическими константами true и false. Таким образом, константе true эквивалентны все числа от 1 до 255 включительно, тогда как константе false эквивалентно только одно целое число – 0. Рассмотрим программу с использованием типа данных bool:

// data_type.cpp: определяет точку входа для консольного приложения.

```
#include "stdafx.h"
#include <iostream>
using namespace std;

int main(int argc, char* argv[])
{
```

bool boolean = 25; // переменная типа bool с именем boolean

if (boolean) // условие оператора if

cout << "true = " << boolean << endl; // выполнится в случае истинности условия

else

cout << "false = " << boolean << endl; // выполнится в случае, если условие ложно

system("pause");

return 0;

}

В строке 9 объявлена переменная типа bool, которая инициализирована значением 25. Теоретически после строки 9, в переменной boolean должно содержаться число 25, но на самом деле в этой переменной содержится число 1. Как я уже говорил, число 0 – это ложное значение, число 1 – это истинное значение. Суть в том, что в переменной типа bool могут содержаться два значения – 0 (ложь) или 1 (истина). Тогда как под тип данных bool отводится целый байт, а это значит, что переменная типа bool может содержать числа от 0 до 255. Для определения ложного и истинного значений необходимо всего два значения 0 и 1. Возникает вопрос: «Для чего остальные 253 значения?».

Исходя из этой ситуации, договорились использовать числа от 2 до 255 как эквивалент числу 1, то есть истина. Вот именно по этому в переменной boolean содержится число 25 а не 1. В строках 10-13 объявлен оператор условного выбора if, который передает управление оператору в строке 11, если условие истинно, и оператору в строке 13, если условие ложно.

Тип данных char

Тип данных char – это целочисленный тип данных, который используется для представления символов. То есть, каждому символу соответствует определённое число из диапазона

[0;255]. Тип данных char также ещё называют символьным типом данных, так как графическое представление символов в C++ возможно благодаря char. Для представления символов в C++ типу данных char отводится один байт, в одном байте – 8 бит, тогда возведем двойку в степень 8 и получим значение 256 – количество символов, которое можно закодировать. Таким образом, используя тип данных char можно отобразить любой из 256 символов. Все закодированные символы представлены в таблице ASCII.

ASCII (от англ. American Standard Code for Information Interchange) – американский стандартный код для обмена информацией.

Рассмотрим программу с использованием типа данных char.

```
// symbols.cpp: определяет точку входа для консольного приложения.
#include "stdafx.h"
#include <iostream>
using namespace std;

int main(int argc, char* argv[])
{
 char symbol = 'a'; // объявление переменной типа char и инициализация её символом 'a'
 cout << "symbol = " << symbol << endl; // печать символа, содержащегося в переменной symbol
 char string[] = "cppstudio.com"; // объявление символьного массива (строки)
 cout << "string = " << string << endl; // печать строки
 system("pause");
 return 0;
}
```

В строке 9 объявлена переменная с именем symbol, ей присвоено значение символа 'a' (ASCII код). В строке 10

оператор cout печатает символ, содержащийся в переменной symbol. В строке 11 объявлен строковый массив с именем string, причём размер массива задан неявно. В строковый массив сохранена строка "cppstudio.com". Обратите внимание на то, что, когда мы сохраняли символ в переменную типа char, то после знака равно мы ставили одинарные кавычки, в которых и записывали символ. При инициализации строкового массива некоторой строкой, после знака равно ставятся двойные кавычки, в которых и записывается некоторая строка. Как и обычный символ, строки выводятся с помощью оператора cout, строка 12**.**

Целочисленные типы данных

Целочисленные типы данных используются для представления чисел. В таблице 1 их шесть штук: short int, unsigned short int, int, unsigned int, long int, unsigned long int**.** Все они имеют свой собственный размер занимаемой памяти и диапазоном принимаемых значений. В зависимости от компилятора, размер занимаемой памяти и диапазон принимаемых значений могут изменяться. В таблице 1 все диапазоны принимаемых значений и размеры занимаемой памяти взяты для компилятора. Причём все типы данных в таблице 1 расположены в порядке возрастания размера занимаемой памяти и диапазона принимаемых значений. Диапазон принимаемых значений, так или иначе, зависит от размера занимаемой памяти. Соответственно, чем больше размер занимаемой памяти, тем больше диапазон принимаемых значений. Также диапазон принимаемых значений меняется в случае, если тип данных объявляется с приставкой unsigned – без знака. Приставка unsigned говорит о том, что тип данных не может хранить знаковые значения, тогда и диапазон положительных значений увеличивается в два раза, например, типы данных short int и unsigned short int.

Приставки целочисленных типов данных:

short – приставка укорачивает тип данных, к которому применяется, путём уменьшения размера занимаемой памяти;

long – приставка удлиняет тип данных, к которому применяется, путём увеличения размера занимаемой памяти;

unsigned (без знака) – приставка увеличивает диапазон положительных значений в два раза, при этом диапазон отрицательных значений в таком типе данных храниться не может.

Так, что, по сути, мы имеем один целочисленный тип для представления целых чисел – это тип данных int. Благодаря приставкам short, long, unsigned появляется некоторое разнообразие типов данных int, различающихся размером занимаемой памяти и (или) диапазоном принимаемых значений.

Типы данных с плавающей точкой

В C++ существуют два типа данных с плавающей точкой: float и double. Типы данных с плавающей точкой предназначены для хранения чисел с плавающей точкой. Типы данных float и double могут хранить как положительные, так и отрицательные числа с плавающей точкой. У типа данных float размер занимаемой памяти в два раза меньше, чем у типа данных double, а значит и диапазон принимаемых значений тоже меньше. Если тип данных float объявить с приставкой long, то диапазон принимаемых значений станет равен диапазону принимаемых значений типа данных double. В основном, типы данных с плавающей точкой нужны для решения задач с высокой точностью вычислений, например, операции с деньгами.

Итак, мы рассмотрели главные моменты, касающиеся основных типов данных в C++. Осталось только показать, откуда взялись все эти диапазоны принимаемых значений и размеры занимаемой памяти. А для этого разработаем программу, которая будет вычислять основные характеристики всех, выше рассмотренных, типов данных.

```
// data_types.cpp: определяет точку входа для консольного приложения.

#include "stdafx.h"
#include <iostream>
// библиотека манипулирования вводом/выводом
#include <iomanip>
// заголовочный файл математических функций
#include <cmath>
using namespace std;

int main(int argc, char* argv[])
{
 cout << " data type " << "byte" << " " << " max value " << endl // заголовки столбцов
 << "bool = " << sizeof(bool) << " " << fixed << setprecision(2)
 /*вычисляем максимальное значение для типа данных bool*/ << (pow(2,sizeof(bool) * 8.0) - 1) << endl
 << "char = " << sizeof(char) << " " << fixed << setprecision(2)
 /*вычисляем максимальное значение для типа данных char*/ << (pow(2,sizeof(char) * 8.0) - 1) << endl
 << "short int = " << sizeof(short int) << " " << fixed << setprecision(2)
 /*вычисляем максимальное значение для типа данных short int*/ << (pow(2,sizeof(short int) * 8.0 - 1) - 1) << endl
 << "unsigned short int = " << sizeof(unsigned short int) << " " << fixed << setprecision(2)
 /*вычисляем максимальное значение для типа данных unsigned short int*/ << (pow(2,sizeof(unsigned short int) * 8.0) - 1) << endl
 << "int = " << sizeof(int) << " " << fixed << setprecision(2)
 /*вычисляем максимальное значение для типа данных int*/
```

*<< (pow(2,sizeof(int) * 8.0 - 1) - 1) << endl*

<< "unsigned int = " << sizeof(unsigned int) << " " << fixed << setprecision(2)

*/*вычисляем максимальное значение для типа данных unsigned int*/ << (pow(2,sizeof(unsigned int) * 8.0) - 1) << endl*

<< "long int = " << sizeof(long int) << " " << fixed << setprecision(2)

*/*вычисляем максимальное значение для типа данных long int*/ << (pow(2,sizeof(long int) * 8.0 - 1) - 1) << endl*

<< "unsigned long int = " << sizeof(unsigned long int) << " " << fixed << setprecision(2)

*/*вычисляем максимальное значение для типа данных undigned long int*/ << (pow(2,sizeof(unsigned long int) * 8.0) - 1) << endl*

<< "float = " << sizeof(float) << " " << fixed << setprecision(2)

*/*вычисляем максимальное значение для типа данных float*/ << (pow(2,sizeof(float) * 8.0 - 1) - 1) << endl*

<< "double = " << sizeof(double) << " " << fixed << setprecision(2)

*/*вычисляем максимальное значение для типа данных double*/ << (pow(2,sizeof(double) * 8.0 - 1) - 1) << endl;*

system("pause");

return 0; }

Для поверхностного ознакомления с кодом программы, поясним некоторые моменты. Оператор sizeof() вычисляет количество байт, отводимое под тип данных или переменную. Функция pow(x,y) возводит значение x в степень y, данная функция доступна из заголовочного файла <cmath>. Манипуляторы fixed и setprecision() доступны из заголовочного файла <iomanip>. Первый манипулятор – fixed**,** передаёт в поток вывода значения в фиксированной форме. Манипулятор setprecision(n) отображает n знаков после

запятой. Максимальное значение некоторого типа данных вычисляется по такой формуле:

*max_val_type = 2^(b * 8 - 1) - 1; // для типов данных с отрицательными и положительными числами*

// где, b - количество байт выделяемое в памяти под переменную с таким типом данных

// умножаем на 8, так как в одном байте 8 бит

// вычитаем 1 в скобочках, так как диапазон чисел надо разделить надвое для положительных и отрицательных значений

// вычитаем 1 в конце, так как диапазон чисел начинается с нуля

// типы данных с приставкой unsigned

*max_val_type = 2^(b * 8) - 1; // для типов данных только с положительными числами*

// пояснения к формуле аналогичные, только в скобочка не вычитается единица

В первом столбце показаны основные типы данных в C++, во втором столбце размер памяти, отводимый под каждый тип данных и в третьем столбце – максимальное значение, которое может содержать соответствующий тип данных. Минимальное значение находится аналогично максимальному. В типах данных с приставкой unsigned минимальное значение равно 0.

Если, например, переменной типа short int присвоить значение 33000, то произойдет переполнение разрядной сетки, так как максимальное значение в переменной типа short int это 32767. То есть в переменной типа short int сохранится какое-то другое значение, скорее всего будет отрицательным. Раз уж мы затронули тип данных int, стоит отметить, что можно опускать ключевое слово int и писать, например, просто short. Компилятор будет интерпретировать такую запись как short int. Тоже самое относится и к приставкам long и unsigned. Например:

// сокращённая запись типа данных int
short a1; // тоже самое, что и short int
long a1; // тоже самое, что и long int
unsigned a1; // тоже самое, что и unsigned int
unsigned short a1; // тоже самое, что и unsigned short int

Тема 2.3. Операторы объявлений

Текст программы начинается с объявления переменных. Под объявлением переменных понимаются строки:

<имя типа данных 1> <имя переменной 1>,…, <имя переменной N>;

<имя типа данных M> <имя переменной 1>,…, <имя переменной K>;

Пример объявления переменных:

int f, k; //объявлены две целочисленные переменные с именами f и k

float dr; //объявлена вещественная переменная с именем dr

char ch; //объявлена символьная переменная с именем ch

При объявлении переменной под неё в оперативной памяти (ОП) отводится конкретное место (участок ОП). Размер участка оперативной памяти, который занимает переменная, определяется типом этой переменной. Для определения объема оперативной памяти, который занимает переменная какого-либо типа можно воспользоваться операцией sizeof(), например:

i=sizeof(int);

В этом примере значение переменной i будет равно 4.

При объявлении переменных можно сообщить им начальные значения путём помещения знака равенства после имени переменной. Этот процесс называется инициализацией и в общем случае имеет вид:

<имя_типа_данных><имя_переменной1>=знач1,…,<имя_переменнойN>=значN;

Например:

int f = 10, k = 56;

Кроме переменных в программе могут быть использованы именованные и неименованные константы. Неименованная константа – число, именованная – идентификатор, которому поставлено в соответствие постоянное значение, не изменяющееся в процессе выполнения программы. Объявление констант выглядит следующим образом:

const <имя_типа_данных>
<имя_константы1>=знач1,...,<имя_константыN>=значN;

При инициализации вещественных переменных и при определении вещественных констант вместо запятой в записи числа всегда ставится точка, например:

const float pi= 3.14159;

При задании имен переменных и констант нужно учитывать то обстоятельство, что компилятор С++ различает строковые и заглавные символы. Например, переменные с именами Alfa и alfa – это различные переменные.

Оператор goto – это оператор управления порядком выполнения кода, который заставляет ЦП сделать переход из одного участка кода на другой, осуществить так называемый прыжок. Другой участок кода идентифицируется с помощью statement label. Ниже приведен пример с использованием goto и statement label:

```
#include <iostream>
#include <cmath> // для функции sqrt()

int main()
{
 double z;
tryAgain: // это statement label
 std::cout << "Enter a non-negative number";
 std::cin >> z;

 if (z < 0.0)
```

```
 goto tryAgain; // а это оператор goto
  std::cout << "The sqrt of " << z << " is " << sqrt(z) <<
std::endl;
 return 0;
}
```

В этой программе пользователю предлагается ввести неотрицательное число. Однако, если пользователь введет отрицательное число, то программа, используя оператор goto, сделает переход назад к строчке tryAgain. Затем пользователю снова нужно будет ввести число. Таким образом, мы можем постоянно запрашивать у пользователя ввод числа, пока он или она не введут корректные числа.

Существуют некоторые ограничения на использование операторов goto. Например, вы не сможете перепрыгнуть вперед через переменную, которая инициализирована в том же блоке, что и goto:

```
int main()
{
 goto skip; // прыжок вперед недопустим
 int z = 7;
skip:
 z += 4; // какое значение будет в этой переменной
вообще?
 return 0;
}
```

В целом, программисты избегают использования оператора goto в C++ (и в большинстве других высокоуровневых языков). Основная проблема с ним заключается в том, что он позволяет программисту управлять выполнением кода так, что точка выполнения может прыгать по коду произвольно. А это в свою очередь создает то, что бывалые программисты называют «спагетти-код». Спагетти-код – это код, порядок выполнения которого напоминает тарелку со спагетти (всё

запутано и закручено), что крайне затрудняет следованию и пониманию логике выполнения такого кода.

Как говорил один известный специалист в информатике и программировании, Эдсгер Дейкстра: «Качество программистов – это уменьшающаяся функция плотности использования операторов goto в программах, которые они пишут».

Оператор goto часто используется в некоторых старых языках, таких как Basic или Fortran, и даже в C. Однако в C++ goto почти никогда не используется, поскольку любой код, написанный с ним, можно более эффективно переписать с использованием других конструкций в C++, таких как циклы, обработчики исключений или деструкторы.

Комментарий – это строка (или несколько строк) текста, которая вставляется в исходный код для объяснения того, что делает этот код. В C++ есть два типа комментариев: однострочные и многострочные.

Однострочные комментарии – это комментарии, которые пишутся между символами //. Они размещаются в отдельных строках и всё, что находится после этих символов комментирования – игнорируется компилятором. Например:

Как правило, однострочные комментарии используются для объяснения одной строчки кода

Размещая комментарии справа от кода, мы затрудняем себе как чтение кода, так и чтение комментариев. Следовательно, однострочные комментарии лучше размещать над строками кода.

Многострочные комментарии – это комментарии, которые пишутся между символами **/* */**. Всё, что находится между звёздочками – игнорируется компилятором

Так как всё, что находится между звёздочками – игнорируется, то иногда вы можете наблюдать следующее:

Многострочные комментарии не могут быть вложенными (т.е. одни комментарии внутри других):

Как правильно писать комментарии?

Во-первых, на уровне библиотек/программ/функций комментарии отвечают на вопрос «ЧТО?»**:** «Что делают эти библиотеки/программы/функции?»

Все эти комментарии позволяют понять, что делает программа, без необходимости смотреть на исходный код. Это особенно важно специалистам, работающим в команде, где не каждый специалист будет знаком со всем имеющимся кодом.

Во-вторых, внутри библиотек/программ/функций комментарии отвечают на вопрос «КАК?»: «Как код выполняет задание?»:

Эти комментарии позволяют пользователю понять, каким образом код выполняет поставленное ему задание.

В-третьих, на уровне стейтментов (однострочного кода) комментарии отвечают на вопрос «ПОЧЕМУ?»: «Почему код выполняет задание именно таким образом, а не другим?». Плохой комментарий на уровне стейтментов объясняет, что делает код. Если вы когда-нибудь писали код, который был настолько сложным, что нужен был комментарий, который бы объяснял, что он делает, то вам нужно было бы не писать комментарий, а переписывать целый код.

Программистам часто приходится принимать трудные решения по поводу того, каким способом решить проблему. А комментарии и существуют для того, чтобы напомнить себе (или объяснить другим) причину, почему вы написали код именно так, а не иначе.

И, наконец, комментарии нужно писать так, чтобы человек, который не имеет ни малейшего представления о том, что делает ваш код – смог в нём разобраться. Очень часто случаются ситуации, когда программист говорит: «Это же совершенно очевидно, что делает код! Я это точно не забуду». Угадайте, что случится через несколько недель или даже дней? Это не совершенно очевидно, и вы удивитесь, как скоро вы забудете, что делает ваш код. Вы (или кто-то другой)

будете очень благодарны себе за то, что оставите комментарии, объясняя на человеческом языке что, как и почему делает ваш код. Читать отдельные строки кода – легко, понимать их логику и смысл – сложно.

Тема 2.4. Операторы вызова функций

Создание функций

Согласно стандарту ANSI C каждая функция должна иметь свой прототип. Прототип представляет собой объявление функции в следующем формате:

<тип> имя_функции (параметры функции);

Прототип может располагаться либо в отдельном заголовочном файле либо в том же файле, где функция используется. Прототип функции должен располагаться вне описаний функций, а также выше того места в программе, где она впервые будет использоваться. Пример:

```
#include <stdio.h>
void MyFunc();//прототип функции
main()
{
MyFunc();
return 0;
}

void MyFunc()
{
printf("Функция выполнена");
}
```

Если функция не возвращает значения, то оператор return при её завершении вызывать не обязательно.

Передача параметров

Если функция имеет параметры, то её прототип должен содержать как минимум типы используемых параметров, имена переменных указывать не обязательно. Пример:

```
 int MyFunc(int, int);
main()
{
printf(«%d»,MyFunc(10, 5));
return 0;
}
int MyFunc(int a, int)
{
return a+b:
}
```

Параметры функции, имеющие значения по умолчанию.

Значения параметров по умолчанию указываются только в прототипе функции их более не нужно указывать в описании функции. Пример:

```
int MyFunc(int a, int b=5);
int MyFunc(int a, int b )
 {
      return a+b;
 }
```

Если в прототипе функции какой-либо параметр имеет значение по умолчанию, то для всех параметров справа также должны быть определены параметры по умолчанию.

При вызове функции, параметры по умолчанию можно опускать.

Пример:

```
 int MyFunc(int a, int b=4, c=2, int d=10);
  main()
{
printf("%d\n", MyFunc(2,3,5,4) ); // на экране 14
printf("%d\n", MyFunc(10) ); // на экране 26
return;
}
 int MyFunc(int a, int b, int c, int d)
{
```

```
return a+b+c+d;
}
```

Функции с неопределенным числом параметров.

Функция с неопределенным числом параметров объявляется с помощью операции многоточия.

<тип> имя_функции(<обязательные параметры>, ...);

Операция многоточие определяет, что после обязательных параметров, следуют неопределенное число параметров. Согласно стандарту ANSI C, подобные функции должны не менее одного обязательного параметра.

Для выборки параметров функции можно воспользоваться макросами va_start, va_arg, va_end, которые определены в библиотеке <stdarg.h>.

Пример:

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

void f(int count, ...)
{
 int arg;
 va_list s;
 va_start(s, count);
 for (int i = 1; i <= count; i++) {
 arg = va_arg(s, int);
 printf("arg N %d = %d\n", i, arg);
 }
 va_end(s);
}

void main(void)
{
 f(1, 12);
 f(3, -10, -20, -30);
}
```

1. Описывается переменная типа va_list (эта структура, содержащая указатель на текущий параметр):va_list s;

2. s инициализируется первым в списке параметром, а также числом переданных параметров:

va_start(s, count);

3. Очередной аргумент получается вызовом макроса va_arg. , имеющего следующие параметры: переменная типа va_list и тип параметра, который Вы хотите получить На основании типа происходит копирование аргумента в Вашу переменную и вычисление смещения следующего аргумента. int d = va_arg(s, int);

4. Очистка s осуществляется вызовом макроса va_end.

va_end(s);

Перегрузка

В С можно определить несколько функций с одним и тем же именем, однако, они должны отличаться по типу принимаемых параметров. Функции не могут отличаться друг от друга лишь типом возвращаемого значения. Пример:

```
#include <stdio.h>

double Calc(double f1, int N)
{
    return f1/N;
}

int Calc(int N1, int N2)
{
    return N1*N2;
}
double Calc(double f1, int N1, int N2)
{
    return (double)(N1*N2)+f1;
}
```

```
main()
{
int n;
double f;
in = Calc( 2, 5 );    //n=10
f= Calc( 5.5, 5 );   //f=1.1
f=Calc(2.0, 5, 9); //f=47.0

    return 0;
}
```

Компилятор отличает одну функцию от другой по типу передаваемых ей в качестве параметров значений.

Пространства имен

Пространства имен – предназначены для предотвращения конфликтов идентификаторов (имен) внутри программы.

Например, std – содержит имена стандартных функций С++ и назвать что либо новое такими именами вы уже не сможете.

Пользовательское пространство имен, синтакис на примере:

Используем переменную x: std::cout<<ns::x;

Импортирование идентификаторов в глобальную или локальные области с помощью слова using.

```
#include <iostream>
Namespace NS {
Int x=10;}
Using name space NS;
Int main () {
Cout<<x;
Cin.get();
Return 0;}
```

Существует неименованное пространство имен, идентификаторы которого видны в пределах всего файла с кодом:

```
Namespace {Int x=10;}
```

Локальные переменные

Переменные, объявляемые внутри функций, называются локальными переменными. В некоторой литературе по С данные переменные могут называться автоматическими из-за использования ключевого слова auto. Поскольку термин локальные переменные широко используется, то далее мы будем пользоваться им. С локальными переменными могут работать только операторы, находящиеся в блоке, где данные переменные объявлены. Вне этого блока локальные переменные неизвестны. Следует помнить, что блок кода начинается открытием фигурной скобки и заканчивается закрытием фигурной скобки.

Наиболее важно понять то, что локальные переменные существуют только в блоке кода, в котором они объявлены. Таким образом, локальные переменные создаются при входе в блок и уничтожаются при выходе из него.

Наиболее типичным блоком кода, в котором объявляются локальные переменные, является функция.

Язык С содержит ключевое слово auto, которое можно использовать для объявления локальных переменных. Тем не менее, поскольку предполагается, что все неглобальные переменные по умолчанию созданы с ключевым словом auto, то оно на самом деле никогда не используется.

Наиболее типично объявление всех необходимых для функции переменных в начале блока кода функции. Это выполняется, главным образом, для того, чтобы любой читающий код, знал об используемых переменных. Тем не менее, нет необходимости выполнять это, поскольку локальные переменные могут быть объявлены в любом блоке кода. (Они должны быть объявлены в начале блока, перед операторами, выполняющими какие-либо действия.)

Возможность объявления переменной в собственном блоке кода, а не в начале функции, может предотвратить случайное ненужное употребление переменной где-либо в функции. По

существу, объявление переменных в блоке кода, использующего их, позволяет отделить код от данных.

Поскольку локальные переменные уничтожаются при выходе из функции, в которой они объявлялись, то эти переменные не могут хранить значение между вызовами функций. (Как будет видно, имеется возможность заставить компилятор сохранять значения путем использования модификатора static.)

Если не определено место для хранения локальных переменных, то они будут храниться в стеке. Тот факт, что стек является динамически изменяющейся областью памяти, объясняет, почему локальные переменные в общем не могут содержать значения между вызовами функций.

Ввод-вывод данных в языке C++ осуществляется либо с помощью функций ввода-вывода в стили C, либо с использованием *библиотеки классов* C++. Преимущество объектов C++ в том, что они легче в использовании, поэтому предлагаю рассмотреть именно их.

Описание объектов для управления *вводом-выводом* содержится в файле iostream.h. При подключении этого файла с помощью директивы #include <iostream.h> в программе автоматически создаются виртуальные каналы связи cin для *ввода* с клавиатуры и cout для *вывода* на экран, а также операции помещения в поток << и чтения из потока >>.

С помощью объекта cin и операции >> можно присвоить значение любой переменной. Например, если переменная x описана как целочисленная, то команда cin>>x; означает, что в переменную x будет записано некое целое число, введенное с клавиатуры. Если необходимо ввести несколько переменных, то следует написать cin>>x>>y>>z;.

Объект cout и операция << позволяет вывести на экран значение любой переменной или текст. Текст необходимо заключать в двойные кавычки. Запись cout<<x; означает вывод на экран значения переменной x.

Тема 2.5. Операторы присвоения

Операции присваивания позволяют присвоить некоторое значения. Эти операции выполняются над двумя операндами, причем левый операнд может представлять только модифицируемое именованное выражение, например, переменную.

Базовая операция присваивания = позволяет присвоить значение правого операнда левому операнду:

```
int x;
x = 2
```

То есть в данном случае переменная x (левый операнд) будет иметь значение 2 (правый операнд).

Стоит отметить, что тип значения правого операнда не всегда может совпадать с типом левого операнда. В этом случае компилятор пытается преобразовать значение правого операнда к типу левого операнда.

При этом операции присваивания имеют правосторонний порядок, то есть выполняются справа налево. И, таким образом, можно выполнять множественное присваивание:

```
int a, b, c;
a = b = c = 34;
```

Здесь сначала вычисляется значение выражения c = 34. Значение правого операнда - 34 присваивается левому операнду c. Далее вычисляется выражение b = c: значение правого операнда c (34) присваивается левому операнду b. И в конце вычисляется выражение a = b: значение правого операнда b (34) присваивается левому операнду a.

Кроме того, следует отметить, что операции присваивания имеют наименьший приоритет по сравнению с другими типами операций, поэтому выполняются в последнюю очередь:

```
int x;
x = 3 + 5;
```

В соответствии с приоритетом операций вначале выполняется выражение 3 + 5, и только потом его значение присваивается переменной x.

Все остальные операции присваивания являются сочетанием простой операции присваивания с другими операциями:

– +=: присваивание после сложения. Присваивает левому операнду сумму левого и правого операндов: A += B эквивалентно A = A + B;

– -=: присваивание после вычитания. Присваивает левому операнду разность левого и правого операндов: A -= B эквивалентно A = A – B;

– *=: присваивание после умножения. Присваивает левому операнду произведение левого и правого операндов: A *= B эквивалентно A = A * B;

– /=: присваивание после деления. Присваивает левому операнду частное левого и правого операндов: A /= B эквивалентно A = A / B;

– %=: присваивание после деления по модулю. Присваивает левому операнду остаток от целочисленного деления левого операнда на правый: A %= B эквивалентно A = A % B;

– <<=: присваивание после сдвига разрядов влево. Присваивает левому операнду результат сдвига его битового представления влево на определенное количество разрядов, равное значению правого операнда: A <<= B эквивалентно A = A << B;

– >>=: присваивание после сдвига разрядов вправо. Присваивает левому операнду результат сдвига его битового представления вправо на определенное количество разрядов, равное значению правого операнда: A >>= B эквивалентно A = A >> B;

– &=: присваивание после поразрядной конъюнкции. Присваивает левому операнду результат поразрядной

конъюнкции его битового представления с битовым представлением правого операнда: A &= B эквивалентно A = A & B;

– |=: присваивание после поразрядной дизъюнкции. Присваивает левому операнду результат поразрядной дизъюнкции его битового представления с битовым представлением правого операнда: A |= B эквивалентно A = A | B;

– ^=: присваивание после операции исключающего ИЛИ. Присваивает левому операнду результат операции исключающего ИЛИ его битового представления с битовым представлением правого операнда: A ^= B эквивалентно A = A ^ B;

– Управляющие символы (или как их ещё называют – escape-последовательность) – символы которые выталкиваются в поток вывода, с целью форматирования вывода или печати некоторых управляющих знаков C++. Основной список управляющих символов языка программирования C++ представлен ниже/

Таблица 2.5.1 – Управляющие символы C++

Символ	Описание
\r	возврат каретки в начало строки
\n	новая строка
\t	горизонтальная табуляция
\v	вертикальная табуляция
\»	двойные кавычки
\’	апостроф
\\	обратный слеш
\0	нулевой символ
\?	знак вопроса
\a	сигнал бипера (спикера) компьютера

Все управляющие символы, при использовании, обрамляются двойными кавычками, если необходимо вывести какое-то сообщение, то управляющие символы можно

записывать сразу в сообщении, в любом его месте. Ниже показан код программы, использующей управляющие символы.

Тема 2.6. Математические и логические операции

Рассмотрим арифметические операции на рисунке ниже.

Оператор	Операция, которая проводится с данными
+	сложение данных
-	вычитание данных
*	умножение данных
/	деление данных
%	деление данных по модулю

Рисунок 2.6.1 – Арифметические операции

Тут особое внимание следует уделить делению по модулю (%). Эта операция достаточно часто используется в решении определённых задач.

Пример её применения: если нам необходимо поделить по модулю 9 на 4 (9 % 4), результат будет равен 1 (это остаток – то, что на 4 уже не делится на цело). Еще примеры: 20 % 8 = 4 (8 помещается в 20-ти 2 раза: 8 * 2 = 16, 20 – 16 = 4 остаток от деления), 3 % 2 = 1, 99 % 10 = 9, 9 % 10 = 9. Важно: деление по модулю применяется только к целочисленным переменным; нельзя делить по модулю на 0.

В C++ определены в заголовочном файле <cmath> функции выполняющие некоторые часто используемые математические задачи. Например, нахождение корня, возведение в степень, sin(), cos() и многие другие. В таблице 2.6.2 показаны основные математические функции, прототипы которых содержатся в заголовочном файле <cmath>.

Таблица 2.6.1 – Математические операции

int abs(int x)	Абсолютное значение
double acos(double x)	Арккосинус
double sin(double x)	Арксинус
double atan(double x)	Арктангенс
double ceil(double x)	Округление до целого с избытком
double cos(double x)	Косинус
double cosh(double x)	Гиперболический косинус
double exp(double x)	Экспонента
double fabs(double x)	Абсолютное значение
double floor(double x)	Округление до целого с недостатком
double fmod(double x, double y)	Деление по модулю
double hypot(double x, double y)	Гипотенуза
long labs(long x)	Абсолютное значение
double ldexp(double x, int exponent)	Произведение x на 2 в степени exponent
double log(double x)	Натуральный логарифм
double log10(double x)	Десятичный логарифм
double pow(double x, double y)	x в степени y
double sine(double x)	Синус
double sinh(double x)	Гиперболический синус
double sqrt(double x)	Квадратный корень
double tan(double x)	Тангенс
double tanh(double x)	Гиперболический тангенс

Таблица 2.6.2 – Основные математические функций

Функция	Описание	Пример
abs(a)	модуль или абсолютное значение от a	abs(-3.0)= 3.0 abs(5.0)= 5.0
sqrt(a)	корень квадратный из a, причём a не отрицательно	sqrt(9.0)=3.0
pow(a, b)	возведение a в степень b	pow(2,3)=8
ceil(a)	округление a до наименьшего целого, но не меньше чем a	ceil(2.3)=3.0 ceil(-2.3)=-2.0
floor(a)	округление a до наибольшего целого, но не больше чем a	floor(12.4)=12 floor(-2.9)=-3
fmod(a, b)	вычисление остатка от a/b	fmod(4.4, 7.5) = 4.4 fmod(7.5, 4.4) = 3.1
exp(a)	вычисление экспоненты e^a	exp(0)=1
sin(a)	a задаётся в радианах	
cos(a)	a задаётся в радианах	
log(a)	натуральный логарифм a(основанием является экспонента)	log(1.0)=0.0
log10(a)	десятичный логарифм a	Log10(10)=1
asin(a)	арксинус a, где -1.0 < a < 1.0	asin(1)=1.5708

Необходимо запомнить то, что операнды данных функций всегда должны быть вещественными, то есть a и b числа с плавающей точкой. Это связано с тем, что существует несколько экземпляров перегруженных функций, соответствующих списку аргументов. Тему перегруженные функции рассмотрим немного позже, а пока надо запомнить, что a и b числа с плавающей точкой. Разработаем программу, которая будет использовать математические функции.

В C++ существует три логические операции:

1. Логическая операция И &&, нам уже известная;
2. Логическая операция ИЛИ ||;
3. Логическая операция НЕ ! или логическое отрицание.

Логические операции образуют сложное (составное) условие из нескольких простых (два или более) условий. Эти операции упрощают структуру программного кода в несколько раз. Да, можно обойтись и без них, но тогда количество ифов увеличивается в несколько раз, в зависимости от условия. В следующей таблице кратко охарактеризованы все логические операции в языке программирования C++, для построения логических условий.

Таблица 2.6.3 – Основные логические операции

Операции	Обозначение	Условие	Краткое описание
И	&&	a == 3 && b > 4	Составное условие истинно, если истинны оба простых условия
ИЛИ	||	a == 3 || b > 4	Составное условие истинно, если истинно, хотя бы одно из простых условий
НЕ	!	!(a == 3)	Условие истинно, если a не равно 3

Сейчас следует понять разницу между логической операцией И и логической операцией ИЛИ, чтобы в дальнейшем не путаться. Пришло время познакомиться с типом данных bool –логический тип данных. Данный тип данных может принимать два значения: true (истина) и false

(ложь). Проверяемое условие в операторах выбора имеет тип данных bool.

Тема 2.7. Приведение типов данных

В языке C++ различают неявное (автоматическое) и явное преобразование типов данных.

Неявное преобразование типов данных при выполнении операций, подобной рассмотренной выше (и в ряде других случаев), выполняется компилятором по определенным правилам автоматически. В чем же состоят эти правила?

Схема преобразования, используемая при выполнении арифметических операций, называется обычными арифметическими преобразованиями. Эта схема может быть описана следующими правилами:

1. Все данные типов char и short int преобразуются к типу int.

2. Если хотя бы один из операндов имеет тип double, то и другой операнд преобразуется к типу double (если он другого типа); результат вычисления имеет тип double.

3. Если хотя бы один из операндов имеет тип float, то и другой операнд преобразуется к типу float (если он другого типа); результат вычисления имеет тип float.

4. Если хотя бы один операнд имеет тип long, то и другой операнд преобразуется к типу long (если он другого типа); результат имеет тип long.

5. Если хотя бы один из операндов имеет тип unsigned, то и другой операнд преобразуется к типу unsigned (если его тип не unsigned); результат имеет тип unsigned.

Если ни один из случаев 1-5 не имеет места, то оба операнда должны иметь тип int; такой же тип будет и у результата.

Следует отметить, что компиляторы языка C++ достаточно свободно выполняют подобные преобразования, что может в ряде случаев привести к неожиданным результатам.

Таким образом, несмотря на то, что язык С++ достаточно «снисходителен» к действиям программиста, это требует от программиста еще большей дисциплины в его действиях и четких знаний нюансов языка программирования.

Здесь было использовано явное преобразование типов данных.

Явное преобразование типов данных осуществляется с помощью соответствующей операции преобразования типов данных, которая имеет один из двух следующих форматов:

(<тип данных>) <выражение> или <тип данных> (<выражение>)

Например:

(int) 3.14 int (3.14)

(double) a или double (a)

(long) (a + 1e5f) long (a + 1e5f)

Подобные преобразования имеют своим исходом три ситуации: преобразование без потерь; с потерей точности; с потерей данных.

Преобразование происходит без потерь, если преобразуемое значение принадлежит множеству значений типа, к которому осуществляется преобразование. Например:

```
short a = 100;
cout << (int) a << endl; // На экран выведено 100
cout << (char) a << endl; // Выведена буква d (ее десятичный эквивалент - 100)
cout << (float) a << endl; // На экран выведено 100
cout << (double) a << endl; // На экран выведено 100
float b = 3.14f;
cout << (double) b << endl; // На экран выведено 3.14
double d = 3.14;
cout << (float) d << endl; // На экран выведено 3.14
```

Преобразование любого вещественного типа к целому осуществляется путем отбрасывания дробной части вещественного значения, поэтому практически всегда такие

преобразования приводят к потере точности (осуществляются приближенно). Например:

double d = 3.74;

cout << (int) d << endl; // На экран выведено 3

А вот попытки преобразования значений, выходящих за пределы диапазона типа данных, к которому осуществляется преобразование, приводят к полному искажению данных. Например:

int a = -100;

cout << (unsigned) a << endl; // На экран выведено 4294967196

int a = 50000;

cout << (short) a << endl; // На экран выведено -15536

float b = 3e+9f;

cout << (int) b << endl; // На экран выведено -2147483648

double d = 3e+9;

cout << (int) d << endl; // На экран выведено -2147483648

double d = 3e+40;

cout << (float) d << endl; // На экран выведено 1.#INF - переполнение

double d = -3e+40;

cout << (float) d << endl; // На экран выведено -1.#INF - переполнение

Рассмотренная операция преобразования типов перешла в C++ из C. В C++ имеются свои операции преобразования типов данных. Например, рассмотренные выше преобразования в C++ можно было бы выполнить с помощью операции static_cast, имеющей следующий формат:

static_cast <тип данных> (выражение)

Например:

static_cast <double> (a + 2e+40f)

Пользоваться явными преобразованиями типов следует очень аккуратно и только там, где это действительно необходимо.

При явном преобразовании типов значения преобразуемых величин на самом деле не изменяются – изменяется только представление этих значений при выполнении действий над ними.

Раздел 3. Операторы управления
Тема 3.1. Операторы ветвления

Условные конструкции направляют ход программы по одному из возможных путей в зависимости от условия.

Конструкция if

Конструкция if проверяет истинность условия, и если оно истинно, выполняет блок инструкций. Этот оператор имеет следующую сокращенную форму:

```
if (условие)
{
 инструкции;
}
```

В качестве *условия* использоваться условное выражение, которое возвращает true или false. Если условие возвращает true, то выполняются последующие инструкции, которые входят в блок if. Если условие возвращает false, то последующие инструкции не выполняются. Блок инструкций заключается в фигурные скобки. Например:

```
#include <iostream>
 int main()
{
 int x = 60;
  if(x > 50)
 {
 std::cout << "x is greater than 50 \n";
 }
  if(x < 30)
 {
 std::cout << "x is less than 30 \n";
 }
  std::cout << "End of Program" << "\n";
 return 0;
}
```

Здесь определены две условных конструкции if. Они проверят больше или меньше значение переменной x, чем определенное значение. В качестве инструкции в обоих случаях выполняется вывод некоторой строки на консоль.

В первом случае x > 50 условие истинно, так как значение переменной x действительно больше 50, поэтому это условие возвратит true, и, следовательно, будут выполнятся инструкции, которые входят в блок if.

Во втором случае операция отношения x < 30 возвратит false, так как условие ложно, поэтому последующий блок инструкций выполняться не будет. В итоге при запуске программы вывод консоли будет выглядеть следующим образом:

```
x greater than 50
End of Program
```

Также мы можем использовать полную форму конструкции if, которая включает оператор else:

```
if(выражение_условия)
 инструкция_1
else
 инструкция_2
```

После оператора else мы можем определить набор инструкций, которые выполняются, если условие в операторе if возвращает false. То есть если *условие* истинно, выполняются инструкции после оператора if, а если это выражение ложно, то выполняются инструкции после оператора else.

```
int x = 50;
if(x > 60)
 std::cout << "x is greater than 60 \n";
else
 std::cout << "x is less or equal 60 \n";
```

В данном случае условие x > 60 ложно, то есть возвращает false, поэтому будет выполняться блок else. И в итоге на консоль будет выведена строка "x is less or equal 60 \n".

Однако нередко надо обработать не два возможных альтернативных варианта, а гораздо больше. Например, в случае выше можно насчитать три условия: переменная x может быть больше 60, меньше 60 и равна 60. Для проверки альтернативных условий мы можем вводить выражения else if:

```
int x = 60;
 if(x > 60)
{
 std::cout << "x is greater than 60 \n";
}
else if (x < 60)
{
 std::cout << "x is less than 60 \n";
}
else
{
 std::cout << "x is equal 60 \n";
}
```

То есть в данном случае мы получаем три ветки развития событий в программе.

Если в блоке if или else или else-if необходимо выполнить только одну инструкцию, то фигурные скобки можно опустить:

```
int x = 60;
 if(x > 60)
 std::cout << "x is greater than 60 \n";
else if (x < 60)
 std::cout << "x is less than 60 \n";
else
 std::cout << "x is equal 60 \n";
```

Тема 3.2. Операторы выбора

Конструкция switch

Другую форму организации ветвления программ представляет конструкция switch...case. Она имеет следующую форму:

```
switch(выражение)
{
 case константа_1: инструкции_1;
 case константа_2: инструкции_2;

 default: инструкции;
}
```

После ключевого слова switch в скобках идет сравниваемое выражение. Значение этого выражения последовательно сравнивается со значениями после оператора case. И если совпадение будет найдено, то будет выполняться определенный блок case.

В конце конструкции switch может стоять блок default. Он необязателен и выполняется в том случае, если значение после switch не соответствует ни одному из операторов case. Например:

```
#include <iostream>
 int main()
{
 int x = 2;
  switch(x)
 {
 case 1:
 std::cout << "x = 1" << "\n";
 break;
 case 2:
 std::cout << "x = 2" << "\n";
 break;
 case 3:
```

```
 std::cout << "x = 3" << "\n";
 break;
 default:
 std::cout << "x is undefined" << "\n";
 break;
 }
 return 0;
}
```

Чтобы избежать выполнения последующих блоков case/default, в конце каждого блока ставится оператор break. То есть в данном случае будет выполняться оператор

```
 case 2:
 std::cout << "x = 2" << "\n";
 break;
```

После выполнения оператора break произойдет выход из конструкции switch..case, и остальные операторы case будут проигнорированы. Поэтому на консоль будет выведена следующая строка

```
x = 2
```

Стоит отметить важность использования оператора break. Если мы его не укажем в блоке case, то после этого блока выполнение перейдет к следующему блоку case. Например, уберем из предыдущего примера все операторы break:

```
#include <iostream>
 int main()
{
 int x = 2;

 switch(x)
 {
 case 1:
 std::cout << "x = 1" << "\n";
 case 2:
 std::cout << "x = 2" << "\n";
```

```
 case 3:
 std::cout << "x = 3" << "\n";
 default:
 std::cout << "x is undefined" << "\n";
 }
 return 0;
}
```

В этом случае опять же будет выполняться оператор case 2:, так как переменная x=2. Однако так как этот блок case не завершается оператором break, то после его завершения будет выполняться набор инструкций после case 3: даже несмотря на то, что переменная x по прежнему равна 2. В итоге мы получим следующий консольный вывод:

```
x = 2
x = 3
x is undefined
```

Тернарный оператор

Тернарный оператор ?: позволяет сократить определение простейших условных конструкций if и имеет следующую форму:

```
[первый операнд - условие] ? [второй операнд] :
[третий операнд]
```

Оператор использует сразу три операнда. В зависимости от условия тернарный оператор возвращает второй или третий операнд: если условие равно true (то есть истинно), то возвращается второй операнд; если условие равно false (то есть ложно), то третий. Например:

```
#include <iostream>

int main()
{
 setlocale(LC_ALL, "");
 int x = 5;
 int y = 3;
```

```
char sign;
std::cout << "Введите знак операции: ";
std::cin >> sign;
int result = sign=='+'?x + y:x - y;
std::cout << "Результат: " << result << "\n";
return 0;
}
```

В данном случае производится ввод знака операции. Здесь результатом тернарной операции является переменная result. И если переменная sign содержит знак "+", то result будет равно второму операнду - (x+y). Иначе result будет равно третьему операнду.

Тема 3.3. Операторы циклов

Цикл while

Выполнение выражений в цикле while продолжается пока логическое выражение истинно.

Синтаксис:

Начальное значение;
While (условие) {
Инструкции…;
Приращение;
}
Например:
Int i=1;
While (i<20) {
Cout<<I;
++I;
}

Цикл do..While

Выполнение выражений в цикле do..while продолжается пока логическое выражение истинно, но в отличие от while условие проверяется в конце цикла:

Начальное значение;

Do {
Инструкции…;
Приращение;
} while (условие);
Например:
Int i=1;
Do {
Cout<<i;
++I;
} while (i<200);

Оператор continue

Переход на следующую итерацию цикла

Оператор continue позволяет исключить из цикла итерации.

Итерация – результат повторного выполнения математической операции

Например, выведем все числа от 1 до 100 кроме чисел от 5 до 10 включительно:

For (int i=1; i<=100; ++i) {
If (i>4 && i<11) continue;
Cout<<I;
}

Оператор break

Прерывание цикла досрочно

Int i=1;
While (1) {
If (i>100) break;
Cout<<I;
++I;
} // в результате выведутся все числа от 1 до 100

Оператор goto – передает управление в любое место программы

Формат: *goto метка;*

Например, вывод чисел от 1 до 1000:

Int i=1;

```
block_start: {
if(i>1000) goto block_end;
cout<<I;
++I;
goto block_start;
}block_end;
```

1. Какие особенности использования операторов инкремента и декремента в программах на С++?

В языке С++ определены два оператора, которые осуществляют увеличение или уменьшение целочисленной величины на 1:

- оператор **++** – инкремент;
- оператор **– –** – декремент.

Эти операторы являются унарными. Они требуют одного операнда. Эти операторы могут размещаться до и после операнда.

Оператор инкремента ++ увеличивает значение операнда на 1. Например, строка x = x + 1; есть аналогична строке x++;
или ++x;

Так же, оператор декремента **–** уменьшает значение операнда на 1. Например, строку

x = x – 1; можно записать x--; или --x;

2. Примеры применения операторов инкремента (++) и декремента (–)

Фрагмент кода, который объясняет работу
операторов **++** и **–.**

```
// операторы инкремента (++) и декремента (--)
int a, b;
a = 10;
b = a++; // b = 10; a = 11

a = 10;
b = ++a; // b = 11; a = 11
```

a = 10;
b = a--; // b = 10; a = 9

a = 10;
b = --a; // b = 9; a = 9

3. Какое отличие между выражением ++x (–x) и выражением x++ (x–)?

Отличие между префиксной и постфиксной формами операторов инкремента (**++**) и декремента (**–**) проявляется в случаях, когда эти операторы принимают участие в операторе присваивания.

Если префиксное выражение **++x** используется в операторе присваивания

y = ++x;

то оно работает по следующему принципу:

– сначала значение x увеличивается на 1, а затем результирующее значение присваивается переменной y.

Если выполнить постфиксную форму оператора инкремента

y = x++;

то в этом случае:

– сначала переменной y присваивается значение x, а потом значение x увеличивается на 1.

4. Какие составные операторы присваивания используются в C++?

В языке C++ можно использовать составные операторы присваивания. Эти операторы являются удобны, когда в программе используются длинные имена переменных. В этом случае отпадает необходимость лишний раз вводить длинное имя переменной.

Общий вид составного оператора присваивания следующий:

имя_переменной *operation*= выражение; где:

– *имя_переменной* – непосредственно имя переменной, которой присваивается значение;

– *operation* – одна из операций **+**, **–**, *****, /, %, &, |, ^, <<, >>.

Язык C++ поддерживает следующие составные операторы присваивания:

+=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=

5. Примеры использования составных операторов присваивания

```
// составные операторы присваивания
float x, y;
int a, b;

// +=, -=
a = 8;
b = 5;
a += b; // a = a + b = 13
b -= 4; // b = b - 4 = 1

// *=, /=
x = 4;
y = 5;
x *= y; // x = x * y = 4 * 5 = 20
y /= 2.5; // y = y / 2.5 = 2.0

// &=, |=
a = 8;
b = 3;
a &= b + 5; // a = a & b + 5 = a & (b+5) = 8
a |= b; // a = a | b = 11

// >>=, <<=
a = 34;
a >>= 1; // a = a >> 1 = 17
a = 6;
```

a <<= 3; // a = a << 3 = 48

// %=
b = 15;
b %= 6; // b = b % 6 = 3

6. Можно ли применять операции инкремента (++) и декремента (–) к типу bool?

В Visual C++ только операцию инкремента (++) можно применять к типу bool.

Если использовать операцию декремента (–) к типу bool, тогда компилятор выдает сообщение об ошибке:

Not allowed on operand of type 'bool'

Пример использования операции инкремента над переменной типа bool

bool b;
b = false;
b++; // b = true
b++; // b = true

7. Можно ли применять операции инкремента и декремента к вещественным типам (float, double, long double)?

С переменными вещественного типа операции инкремента и декремента работают точно так же, как и с переменными целого типа.

Пример:

float x;
x = 24.5;
x--; // x = 23.5

8. Можно ли применять операции инкремента и декремента к переменным символьного типа (char)?

Поскольку, символьный тип char неявно относится к целочисленным типам, то к переменным символьного типа можно применять операции инкремента и декремента.

Пример:

char c;

```
c = 'x';
c++; // c = 'y'

c = '6';
--c; // c = '5'
```

Раздел 4. Сложные типы данных
Тема 4.1. Массивы

Массив представляет набор однотипных данных. Формальное определение массива выглядит следующим образом:

тип_переменной название_массива [длина_массива]

После типа переменной идет название массива, а затем в квадратных скобках его размер. Например, определим массив из 4 чисел:

```
int numbers[4];
```

Данный массив имеет четыре числа, но все эти числа имеют неопределенное значение. Однако мы можем выполнить инициализацию и присвоить этим числам некоторые начальные значения через фигурные скобки:

```
int numbers[4] = {1,2,3,4};
```

Значения в фигурных скобках еще называют инициализаторами. Если инициализаторов меньше, чем элементов в массиве, то инициализаторы используются для первых элементов. Если в инициализаторов больше, чем элементов в массиве, то при компиляции возникнет ошибка:

```
int numbers[4] = {1, 2, 3, 4, 5, 6};
```

Здесь массив имеет размер 4, однако ему передается 6 значений.

Если размер массива не указан явно, то он выводится из количества инициализаторов:

```
int numbers[] = {1, 2, 3, 4, 5, 6};
```

В данном случае в массиве есть 6 элементов.

Свои особенности имеет инициализация символьных массивов. Мы можем передать символьному массиву как набор инициализаторов, так и строку:

```
char s1[] = {'h', 'e', 'l', 'l', 'o'};
char s2[] = "world";
```

Причем во втором случае массив s2 будет иметь не 5 элементов, а 6, поскольку при инициализации строкой в

символьный массив автоматически добавляется нулевой символ '\0'.

При этом не допускается присвоение одному массиву другого массива:

```
int nums1[] = {1,2,3,4,5};
int nums2[] = nums1; // ошибка
nums2 = nums1; // ошибка
```

После определения массива мы можем обратиться к его отдельным элементам по индексу. Индексы начинаются с нуля, поэтому для обращения к первому элементу необходимо использовать индекс 0. Обратившись к элементу по индексу, мы можем получить его значение, либо изменить его:

```
#include <iostream>

int main()
{
 int numbers[4] = {1,2,3,4};
 int first_number = numbers[0];
 std::cout << first_number << std::endl; // 1
 numbers[0] = 34; // изменяем элемент
 std::cout << numbers[0] << std::endl; // 34

 return 0;
}
```

Число элементов массива также можно определять через константу:

```
const int n = 4;
int numbers[n] = {1,2,3,4};
```

Многомерные массивы

Кроме одномерных массивов в C++ есть многомерные. Элементы таких массивов сами в свою очередь являются массивами, в которых также элементы могут быть массивами. Например, определим двухмерный массив чисел:

```
int numbers[3][2];
```

Такой массив состоит из трех элементов, при этом каждый элемент представляет массив из двух элементов. Инициализируем подобный массив:

```
int numbers[3][2] = { {1, 2}, {4, 5}, {7, 8} };
```

Также при инициализации можно опускать фигурные скобки:

```
int numbers[3][2] = { 1, 2, 4, 5, 7, 8 };
```

Возможна также инициализация не всех элементов, а только некоторых:

```
int numbers[3][2] = { {1, 2}, {}, {7} };
```

И чтобы обратиться к элементам вложенного массива, потребуется два индекса:

```
int numbers[3][2] = { {1, 2}, {3, 4}, {5, 6} };
std::cout << numbers[1][0] << std::endl; // 3
numbers[1][0] = 12; // изменение элемента
std::cout << numbers[1][0] << std::endl; // 12
```

Переберем двухмерный массив:

```
#include <iostream>

int main()
{
 const int rows = 3, columns = 2;
 int numbers[rows][columns] = { {1, 2}, {3, 4}, {5, 6} };
 for(int i=0; i < rows; i++)
 {
 for(int j=0; j < columns; j++)
 {
 std::cout << numbers[i] [j] << "\t";
 }
 std::cout << std::endl;
 }
 return 0;
}
```

Также для перебора элементов многомерного массива можно использовать другую форму цикла for:

```
#include <iostream>

int main()
{
 const int rows = 3, columns = 2;
 int numbers[rows][columns] = { {1, 2}, {3, 4}, {5, 6} };

 for(auto &subnumbers : numbers)
 {
 for(int number : subnumbers)
 {
 std::cout << number << "\t";
 }
 std::cout << std::endl;
 }
  return 0;
}
```

Для перебора массивов, которые входят в массив, применяются ссылки. То есть во внешнем цикле for(auto &subnumbers: numbers)&subnumbers представляет ссылку на подмассив в массиве. Во внутреннем цикле for(int number : subnumbers) из каждого подмассива в subnumbers получаем отдельные его элементы в переменную number и выводим ее значение на консоль.

Перебор элементов массива

```
Const size=20;
Int ar[size];
//нумеруем элементы массива
For (int i=0; j=1; i<size; ++I; ++j) {
Ar[i]=j;
}
// вывод значений в прямом порядке
```

```
For (int i=0; j=1; i<size; ++I) {
Cout<<Ar[i];
}
// вывод значений в обратном порядке
For (int j=size-1; j=1; j>=0; --j) {
Cout<<Ar[j];
}
```

Доступ к элементам массива

Может осуществляться как по индексу, так и с использованием адресной арифметики.

Инструкции выводящие одно и то же то ест цифру 2.

```
Int ar[3]={1,2,3}
Cout<<ar[1];
Cout<<*(ar+1);
Cout<<*(1+ar);
Cout<<1[ar];
```

Последние 3 инструкции – адресуют через указатель.

Поиск минимального и максимального значений

```
#include <iostream>
Int main() {
Const s=5;
Int a[s]={2,5,6,1,3};
Int min=a[0], max=a[0];
For (int i=1; i<s;++i){
If (min>a[i]) min=a[i];
If (max<a[i]) max=a[i];
}
Cout<<min<<max;
Return 0; }
```

Сортировка массивов

Это упорядочивание по возрастанию или убыванию.

Для этого есть стандартная функция qsort()

Пример листинга:

```
#include <iostream>
```

```
#include <cstdlib>

int mysort(const void *arg1, const void arg2);

int main() {
Const s=5;
int a[s]={2,7,5,1,4};
Qsort(a, s, sizeof(int), mysort);
For (int i=1; i<s;++i){
Cout<<a[i]; }
Return 0;
}
Int mysort(const void *arg1, const void *arg2) {
Return *(int *)arg1 - *(int *)arg2;
}
```

Чтобы произвести стртировку по убыванию поменять местами:

```
Return *(int *)arg2 - *(int *)arg1;
```

Проверка наличия значения в массиве:

Сводится к перебиранию всех элементов массива на вхождение.

```
int main() {
Const s=5;
int a[s]={2,7,5,1,4}, index=-1;
int key=5;

For (int i=1; i<s;++i){
if(a[i]==key) {
index=I;
break;
}}
Cout<<index;
Getch();
Return 0;}
```

Копирование элементов из одного массива в другой

Стандартные функции:

Memcpy()

Пример:

```
#include <cstring>

int main() {
Const s1=5, s2=3;
int a1[s1]={0}; a2[s2]={1, 2, 3};
int *p=0, i=0;
// копируем все эл-ты массива a2
P=(int*)std::memcpy(a1,a2,sizeof a2);
For (i=0; i<s1; ++i) {
Std::cout<<a1[i];} // 1,2,3,0,0
```

Тема 4.2. Перечисления

Перечисления (enum) представляют еще один способ определения своих типов. Их отличительной особенностью является то, что они содержат набор числовых констант.

Определим простейшее перечисление:

```
enum seasons
{
 spring,
 summer,
 autumn,
 winter
};
```

Для определения перечисления применяется ключевое слово enum, после которого идет название перечисления. Затем в фигурных скобках идет перечисление констант через запятую. Каждой константе по умолчанию будет присваиваться числовое значение, начиная с нуля. То есть в данном случае spring=0, а winter=3.

Используем перечисление:

```
#include <iostream>
enum seasons
{
 spring,
 summer,
 autumn,
 winter
};
int main()
{
 seasons currentSeason = autumn;
 // или так
 // seasons currentSeason = seasons::autumn;
 std::cout << "Season: " << currentSeason << std::endl;
 return 0; }
```

Мы можем определить переменную типа seasons и присвоить этой переменной значение одной из констант, объявленных в перечислении. Но фактически это будет числовое значение. В частности, консольный вывод данной программы:

```
Season: 2
```

В то же время перечисления - это отдельный тип, поэтому мы не можем присвоить переменной напрямую числовое значение:

```
seasons currentSeason = 2; // ошибка
```

Если нас не устраивают значения по умолчанию для констант, то мы можем явным образом задать значения. Например, установить начальное значение:

```
enum seasons
{
 spring = 1,
 summer, //2
 autumn, //3
 winter //4 };
```

В этом случае значения второй и последующих констант будет увеличиваться на единицу.

Также можно задать значение для каждой константы:

```
enum seasons
{
 spring = 1,
 summer = 2,
 autumn = 4,
 winter = 8
};
```

Когда необходимы перечисления? Перечисления могут использоваться, когда у нас есть ряд логически связанных констант, которые естественно лучше определить в одном общем типе данных. Например:

```
#include <iostream>
enum operations
{
 add = 1,
 subtract = 2,
 multiply = 4
};
int main()
{
 int operation;
 int x1;
 int x2;
 int result;

 std::cout << "Add: 1 \tSubtract: 2 \tMultiply: 4" << std::endl;
 std::cout << "Input x1: ";
 std::cin >> x1;
 std::cout << "Input x2: ";
 std::cin >> x2;
 std::cout << "Input operation number: ";
```

```
std::cin >> operation;

switch (operation)
{
case operations::add:
result = x1 + x2;
break;
case operations::subtract:
result = x1 - x2;
break;
case operations::multiply:
result = x1 * x2;
break;
}

std::cout << "Result: " << result << std::endl;
return 0;
}
```

В данном случае все арифметические операции хранятся в перечислении operations. В зависимости от выбранной операции в конструкции switch производятся определенные действия.

Тема 4.3. Структуры

Ранее для определения классов мы использовали ключевое слово class. Однако C++ предоставляет еще один способ для определения пользовательских типов, который заключается в использовании структур. Данный способ был унаследован языком C++ еще от языка Си.

Структура в языке C++ представляет собой производный тип данных, который представляет какую-то определенную сущность, также как и класс. Нередко структуры применительно к C++ также называют классами. И в реальности различия между ними не такие большие.

Для определения структуры применяется ключевое слово struct, а сам формат определения выглядит следующим образом:

```
struct имя_структуры
{
 компоненты_структуры
};
```

Имя_структуры представляет произвольный идентификатор, к которому применяются те же правила, что и при наименовании переменных.

После имени структуры в фигурных скобках помещаются *Компоненты_структуры*, которые представляют набор описаний объектов и функций, которые составляют структуру.

Например, определим простейшую структуру:

```
#include <iostream>
#include <string>

struct person
{
 int age;
 std::string name;
};
int main()
{
 person tom;
 tom.name = "Tom";
 tom.age = 34;
 std::cout << "Name: " << tom.name << "\tAge: " << tom.age << std::e
 return 0;
}
```

Здесь определена структура person, которая имеет два элемента: age (представляет тип int) и name (представляет тип string).

После определения структуры мы можем ее использовать. Для начала мы можем определить объект структуры - по сути обычную переменную, которая будет представлять выше созданный тип. Также после создания переменной структуры можно обращаться к ее элементам - получать их значения или, наоборот, присваивать им новые значения. Для обращения к элементам структуры используется операция «точка»:

имя_переменной_структуры.имя_элемента

По сути структура похожа на класс, то есть с помощью структур также можно определять сущности для использования в программе. В то же время все члены структуры, для которых не используется спецификатор доступа (public, private), по умолчанию являются открытыми (public). Тогда как в классе все его члены, для которых не указан спецификатор доступа, являются закрытыми (private).

Кроме того мы можем инициализировать структуру, присвоив ее переменным значения с помощью синтаксиса инициализации:

```
person tom = { 34, "Tom" };
```

Инициализация структур аналогична инициализации массивов: в фигурных скобках передаются значения для элементов структуры по порядку. Так как в структуре person первым определено свойство, которое представляет тип int - число, то в фигурных скобках вначале идет число. И так далее для всех элементов структуры по порядку.

При этом любой класс мы можем представить в виде структуры и наоборот. Возьмем, к примеру, следующий класс:

```
class Person
{
public:
 Person(std::string n, int a)
 {
 name = n; age = a;
 }
```

```
 void move()
 {
 std::cout << name << " is moving" << std::endl;
 }
 void setAge(int a)
 {
 if (a > 0 && a < 100) age = a;
 }
 std::string getName()
 {
 return name;
 }
 int getAge()
 {
 return age;
 }
private:
 std::string name;
 int age;
};
```

Данный класс определяет сущность человека и содержит ряд приватных и публичных переменных и функции. Вместо класса для определения той же сущности мы могли бы использовать структуру:

```
#include <iostream>
#include <string>

struct user
{
public:
 user(std::string n, int a)
 {
 name = n; age = a;
 }
```

```
    void move()
    {
    std::cout << name << " is moving" << std::endl;
    }
    void setAge(int a)
    {
    if (a > 0 && a < 100) age = a;
    }
    std::string getName()
    {
    return name;
    }
    int getAge()
    {
    return age;
    }
   private:
    std::string name;
    int age;

   };

   int main()
   {
    user tom("Tom", 22);
    std::cout << "Name: " << tom.getName() <<
"\tAge: " << tom.getAge() << std::endl;
    tom.setAge(31);
    std::cout << "Name: " << tom.getName() << "\tAge: " <<
tom.getAge() << std::endl;
    return 0;
   }
```

И в плане конечного результата программы мы не увидели бы никакой разницы.

Когда использовать структуры? Как правило, структуры используются для описания таких данных, которые имеют только набор публичных атрибутов - открытых переменных. Например, как та же структура person, которая была определена в начале статьи. Иногда подобные сущности еще называют аггрегатными классами (aggregate classes).

Тема 4.4. Объединения

Объединения – это объект, позволяющий нескольким переменным различных типов занимать один участок памяти. Объявление объединения похоже на объявление структуры:

```
union union_type {
int i; char ch;
};
```

Как и для структур, можно объявить переменную, поместив ее имя в конце определения или используя отдельный оператор объявления. Для объявления переменной cnvt объединения union_type следует написать:

```
union union_type cnvt;
```

В cnvt как целое число i, так и символ ch занимают один участок памяти. (Конечно, i занимает 2 или 4 байта, а ch – только 1.) Рисунок показывает, как i и ch разделяют один участок памяти (предполагается наличие 16-битных целых). Можно обратиться к данным, сохраненным в cnvt, как к целому числу, так и к символу.

Когда объявлено объединение, компилятор автоматически создает переменную достаточного размера для хранения наибольшей переменной, присутствующей в объединении.

Для доступа к членам объединения используется синтаксис, применяемый для доступа к структурам – с помощью операторов «точка» и «стрелка». Чтобы работать с объединением напрямую, надо использовать оператор «точка». Если к переменной объединения обращение происходит с помощью указателя, надо использовать

оператор «стрелка». Например, для присваивания целого числа 10 элементу i объединения cnvt следует написать:

cnvt.i = 10;

Использование объединений помогает создавать машинно-независимый (переносимый) код. Поскольку компилятор отслеживает настоящие размеры переменных, образующих объединение, уменьшается зависимость от компьютера. Не нужно беспокоиться о размере целых или вещественных чисел, символов или чего-либо еще.

Объединения часто используются при необходимости преобразования типов, поскольку можно обращаться к данным, хранящимся в объединении, совершенно различными способами. Рассмотрим проблему записи целого числа в файл. В то время как можно писать любой тип данных (включая целый) в файл с помощью fwrite(), для данной операции использование fwrite() слишком «жирно». Используя объединения, можно легко создать функцию, побайтно записывающую двоичное представление целого в файл. Хотя существует несколько способов создания данной функции, имеется один способ выполнения этого с помощью объединения. В данном примере предполагается использование 16-битных целых. Объединение состоит из одного целого и двухбайтного массива символов:

```
union pw {
int i;
char ch[2];
};
```

Объединение позволяет осуществить доступ к двум байтам, образующим целое, как к отдельным символам. Теперь можно использовать pw для создания функции write_int(), показанной в следующей программе:

```
#include <stdio.h>
#include <stdlib.h>
union pw {
```

```
int i;
char ch[2];
};

int write_int(int num, FILE *fp);

int main()
{
FILE *fp;
fp = fopen("test.tmp", "w+");

if(fp==NULL) {
printf("Cannot open file. \n");
exit(1);
}

write_int(1000, fp);
fclose(fp);
return 0;
}

/* вывод целого с помощью объединения */

int write_int (int num, FILE *fp) {
union pw wrd;
wrd.i = num;
putc(wrd.ch[0], fp); /* вывод первой половины */
return putc(wrd.ch[1], fp); /* вывод второй половины */
}
```

Хотя write_int() вызывается с целым, она использует объединение для записи обеих половинок целого в дисковый файл побайтно.

Тема 4.4. Битовые поля

Битовые поля – это своеобразная структура, которая позволяет работать с отдельными битами.

Битовые поля полезны по нескольким причинам. Ниже приведены три из них:

1. Если ограничено место для хранения информации, можно сохранить несколько логических (истина/ложь) переменных в одном байте.
2. Некоторые интерфейсы устройств передают информацию, закодировав биты в один байт.
3. Некоторым процедурам кодирования необходимо получить доступ к отдельным битам в байте.

Битовое поле, на самом деле, - это просто особый тип структуры, определяющей, какую длину имеет каждый член. Синтаксис объявления битовых полей следующий:

```
struct [имя_структуры] {
 тип [имя_битового_поля1]: длина;
 тип [имя_битового_поля2]: длина;
 . . . . .
 тип [имя_битового_поляN]: длина;
} [имя_объекта];
```

В языке С в качестве типа битовых полей обязательно нужно использовать int или unsigned или signed . В С++ разрешено использовать кроме перечисленных выше любой тип, интерпретируемый как целый: char, bool, short, long и перечисления. Длина битового поля задается целочисленным значением, которое определяет, сколько битов необходимо выделить указанному полю. Приведем пример битовых полей:

```
struct {
unsigned name1: 1;
unsigned name2: 3;
unsigned name3: 5;
} obj;
```

obj.name1 = 1;
obj.name3 = 30;

В приведенном примере мы создали три битовых поля. Для хранения элемента *name1* выделено 1 бит, для хранения элемента *name2* - 3 бита, а для хранения *name3* выделено 5 бит. После объявления битовых полей в нашем примере происходит присвоение значений битовым полям. При этом удостоверьтесь, что присваиваемые значения не превышают размер поля.

Размер одного битового поля не должен превышать размер типа данного поля. То есть, если битовое поле объявлено как unsigned или int, то размер такого поля не должен превышать 32 бита.

В следующем примере объявляется структура, которая содержит битовые поля:

```
struct Date {
 unsigned short nWeekDay : 3; // 0..7 (3 bits)
 unsigned short nMonthDay : 6; // 0..31 (6 bits)
 unsigned short nMonth : 5; // 0..12 (5 bits)
 unsigned short nYear : 8; // 0..100 (8 bits)
};
```

На следующем рисунке показана концептуальная структура памяти для объекта типа Date.

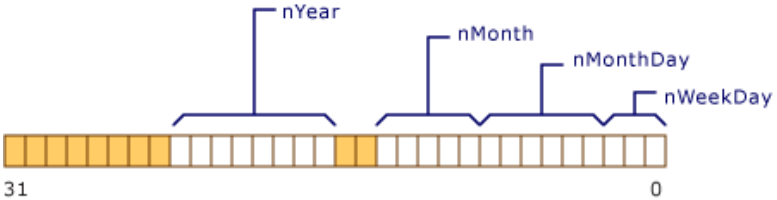


Рисунок 4.4.1 – Структура памяти объекта типа Date

Обратите внимание, что nYear на 8 бит длиннее и может переполнить границу слова для объявленного типа, unsigned short. Поэтому он находится в начале нового слова unsigned short. Совсем не обязательно, чтобы все битовые поля помещались в один объект базового типа; в зависимости от

количества бит, запрошенных в объявлении, выделяются новые единицы хранения.

Как Вы уже поняли, ссылка на битовое поле выполняется по *имени_битового_поля*. Если имя поля не указано, запрошенные биты все равно выделяются, но доступ к ним невозможен. Такое поле называется *неименованным битовым полем*. Существует множество ситуаций, в которых оправдано использование неименованных битовых полей. В конце концов, они ничем не хуже неименованных параметров функций объявление которых можно отделить от определения (инициализации) случаи: float sum (float a, float b) и float sum (float, float);.

```
struct Example1 {
 unsigned n1: 6;
 unsigned : 2;
 unsigned n2: 14;
 unsigned : 2;
 unsigned n3: 5;
};
```

Неименованные битовые поля с шириной поля 0 задают выравнивание следующего поля по границе максимального типа элементов структуры. Например, структура битовых полей

```
struct Example2 {
 unsigned n1: 12;
 unsigned : 0;
 unsigned n2: 8;
};
```

использует неименованное поле нулевой ширины, чтобы пропустить оставшиеся биты в том элементе памяти, в котором хранится n1, и выровнять n2 по границе следующего элемента памяти. Элементом памяти выступает в данном случае 4 байта, то есть размер unsigned int. В общей сложности данная структура будет занимать в памяти 8 байт.

К битовым полям не может быть применена операция получения адреса (&) и поэтому не существует указателей на поля.

Порядок размещения битовых полей в памяти в значительной степени зависит от компилятора и аппаратного обеспечения. Не нужно пытаться экономить память с помощью битовых полей, так как это чаще всего не удается. При использовании битовых полей обычно предполагается, что память для них будет выделена в виде последовательности битовых ячеек, следовательно, структура, состоящая из трех 2-битовых полей, заняла бы 6 бит, следующих один за одним. Но так ли это на самом деле - зависит, в конце концов, от компилятора. Кроме того, адресация битов в памяти является менее рациональной, чем адресация байтов, так как компилятор должен генерировать специальные коды.

Битовые поля можно использовать для создания всевозможных масок и флагов, а также при создании двоично-упакованных объектов.

Доступ к элементам битового поля осуществляется так же, как и доступ к обычным членам структуры. Например,

```
#include struct demo
{
unsigned a: 1;
 signed b: 3;
int c: 6; };
int main () {
struct demo s;
s.a = 1;
s.b = -1;
s.c = -4;
printf("s.a = %u\n", s.a); // печать: s.a = 1
printf("s.b = %d\n", s.b); // печать: s.b = -1
printf("s.c = %d\n", s.c); // печать: s.c = -4
return 1; }
```

Битовые операции

Битовые операции – это тестирование, установка или сдвиг битов в байте или слове, которые соответствуют стандартным типам языка C char и int. Битовые операторы не могут использоваться с float, double, long double, void и другими сложными типами.

Назначение битовых операций:

– разработчики Windows всегда старались сделать так, чтобы старшие поколения Windows поддерживали, по возможности, младшие версии. То есть приложения, работающие в 16-разрядной Windows, должны поддерживаться 32-разрядной Windows;

– к битовым операциям необходимо прибегать при работе с некоторыми системными функциями (особенно функциями для работы с оборудованием);

– битовые операции часто приходиться использовать при работе с графикой (имеется в виду программирование с графикой).

В C имеются следующие битовые операции:

& - битовое И,

| - битовое ИЛИ,

^ - битовое исключающее ИЛИ,

~ - инвертирует каждый разряд,

<< - поразрядный сдвиг влево,

>> - поразрядный сдвиг вправо.

*В операции исключающего ИЛИ сравниваются два разряда, если они одинаковы, то результат – ноль, в противном случае единица.

Примечание: Для сложения по модулю a ^ a = 0.

Пример:

Дано двоичное число 10000101, необходимо переключить его нечетные биты (1-й, 3-й, 5-й, 7-й). Для этого данное число необходимо сложить по модулю с числом, содержащим единицы в заданных разрядах.

10000101_2 ^ 10101010_2 = 00101111_2

Операция XOR также широко используется во многих алгоритмах шифрования информации, так как эта операция обладает свойством обратимости. Это означает, что если над результатом приведенного выше примера повторно провести ту же операцию, то результатом будет исходное число.

00101111_2 ^ 10101010_2 = 10000101_2

Число 10101010_2 в данном случае будет называться ключом шифрования.

При левом сдвиге разряды сдвигаются влево, самый правый разряд(младший) устанавливается в ноль, а самый левый теряется. Данная операция аналогична умножению на два.

Примечание: Для побитового сдвига влево a << N = a * 2^N, причем такой способ умножения работает во много раз быстрее операции арифметического умножения *.

Пример:

10011101_2 << 3 = 11101000_2

При правом сдвиге младшие биты отбрасываются, а старшие устанавливаются в ноль, что аналогично делению на 2.

Примечание: Для побитового сдвига вправо a >> N = a / 2^N, при N >= 0 (для N < 0 работает не всегда). Такой способ деления работает во много раз быстрее операции арифметического деления / (хотя в случае операции сдвига деление всегда целочисленное).

Пример:

01011101_2 >> 3 = 00001011_2

10011101_2 >> 3 = 11110011_2

Второй метод управления разрядами состоит в использовании битового (разрядного) поля, которое представляет собой последовательную цепочку разрядов в рамках значения типа signed int или unsigned int. Оно может быть только элементом структуры или объединения.

Создается путем объявления структуры (объединения), которая помечает каждое поле и определяет его разряд.

Приведем пример с использованием битовых полей в структуре:

```
struct packed_struct {
unsigned int : 3;
unsigned int f1 : 1;
unsigned int f2 : 1;
unsigned int f3 : 1;
unsigned int type : 8;
unsigned int index : 18; };
```

В созданном шаблоне-структуре с дескриптором packed_struct первый член не имеет имени. Символ 3 задает три безымянных бита. Второй, третий и четвертый члены структуры – f1, f2, f3 также имеют тип unsigned int. Символ 1 говорит о том, что в данном члене структуры будет храниться 1 бит. Член структуры с именем type в памяти занимает 8 бит, член структуры index рассчитан на хранение 18 бит.

Для заданного шаблона структуры можно определить структурную переменную, например

```
struct packed_struct packed_data;
```

После этого можно присваивать значения полям структуры:

```
packed_data.type = 7;
```

Если ранее была объявлена какая-то переменная, например n, то присвоение может быть таким:

```
packed_data.type = n;
```

При этом нет необходимости беспокоиться о том, что значение переменной n будет слишком большим. Только младшие 8 бит будут учитываться при присваивании значения для поля packed_data.type. Для извлечения битовых полей структуры можно использовать обычное утверждение:

```
n = packed_data.type;
```

После извлечения значения поля type будет произведен сдвиг в сторону младших бит.

Битовые поля могут быть объявлены только как тип int (в стандарте C99 также _Bool), от реализации которого зависит, будет он знаковым (signed) или беззнаковым (unsigned). Для исключения неоднозначности следует использовать явные объявления: signed int или unsigned int. Битовые поля нельзя объединять в массивы, нельзя использовать адрес битового поля, поэтому не может быть такого типа, как указатель на битовое поле. Компилятор языка программирования С не переупорядочивает битовые поля для получения оптимального распределения памяти. Но в некоторых случаях может производиться выравнивание за счет безымянного поля. Это может использоваться для выравнивания следующего поля структуры по границе блока.

С помощью битовых полей можно формировать объекты с длиной внутреннего представления, не кратной байту.

Использование аргументов командной строки в С

Аргумент командной строки – это информация, которая вводится в командной строке операционной системы вслед за именем программы.

В системных средах, поддерживающих язык программирования С, существует способ передавать в программу аргументы или параметры командной строки при запуске программы на выполнение. Для этого в главную функцию main() включают два аргумента, обычно argc и argv. Первый (от англ. *argument count* – «счетчик аргументов») содержит количество аргументов командной строки, с которыми была запущена программа. Второй (от англ. *argument vector* – «вектор аргументов») указывает на массив символьных строк, содержащих сами аргументы, – по одному в строке. В общем случае имена аргументов могут быть произвольными.

Формально можно определить следующий прототип функции main() с параметрами

*int main (int argc, char *argv[]);*

Второй параметр функции main() представляет собой многоуровневую систему указателей. В связи с этим можно применить другой способ задания параметров функции main(), а именно

*int main (int argc, char **argv);*

Каждый указатель значения типа char ссылается на одну из строк командной строки, при этом argv[0] указывает на имя команды (исполняемой программы), argv[1] – на первый аргумент командной строки, argv[2] – на второй аргумент и т. д. [3].

Аргументами командной строки могут быть исполняемые файлы. Из программы можно запустить на выполнение другую программу, новый процесс. Для этого существуют специальные функции библиотеки C Run–Time Library Reference системы Visual Studio (которую используем в качестве компилятора языка C).

Командная оболочка операционной системы Windows использует интерпретатор команд cmd.exe, который загружает приложения и направляет поток данных между приложениями, для перевода введенной команды в понятный системе код. Консоль командной строки присутствует во всех версиях операционных систем Windows.

Раздел 5. Указатели

Тема 5.1. Работа с указателем. Виды указателя

Определение переменных-указателей

В общем случае имеет вид:

*<тип> *<имя переменной>;*

Переменная-указатель предназначена для хранения адреса другой переменной. Для инициализации подобной переменной используется операция получения адреса &(амперсанд). Пример:

```
int n=5, *pn1;
int *pn2 = &n;
pn1 = &n;
```

В данном примере указатели pn1 и pn2 содержат адрес одной и той же переменной n.

Для того, чтобы получить доступ к ячейке памяти, адрес которой хранится в указателе, используется операция разыменования *.

```
int a=5; b=7;
int *pn;
 p=&a;
 b=*pn; //b=5
 *pn=10; //a=10
Printf("%d", *pn); // на экране 10
```

Примечание: Нельзя использовать операцию & для получения адреса какой-либо константы.

Явная инициализация указателей

Так же как и обычные переменные, указатели можно инициализировать явно во время объявления. К примеру:

```
int n=5;
int *p = &n;
```

Идентичная запись выглядит так:

```
int n=5;
int *p;
 p = &n;
```

Указатели на массивы

Как уже отмечалось, имя массива – это имя константы, содержащей адрес первого элемента массива.

Доступ к массивам в С осуществляется с помощью операции квадратные скобки формат которой выглядит следующим образом:

<адрес первого элемента>[<смещение относительно 1го элемента>].

Пример:

```
Int f[10];
Int *p;
 p=f;
a=f[5];
a=(&f[0])[5];
a=p[5]
```

Последние 3 строки являются абсолютно идентичными.

Все 3 оператора присваивают переменной а значение 5-го элемента массива f.

Непосредственно при объявлении массива можно также использовать указатели, однако при этом требуется явная инициализация массива.

Пример:

```
float *pf={2.0, 4.0, 5.0};
char *pstr=”\nHello”;
for (int i=0; i<3; i++){
        printf("%g,", pf[i]);
}
printf(pstr);
```

На экране:

2,4,5

Hello

Арифметические операции с указателями.

В языке С с адресом можно выполнять только 4 арифметические операции: целочисленное сложение и вычитание, инкремент и декремент.

В результате использования операции инкремент или декремент содержимое соответствующего указателя меняется на величину, равную размеру типа и байтах, на который данный указатель ссылается.

То же самое касается операций сложения и вычитания. Аргумент в правой части присваивания задает смещение, которое можно представить в виде

Аргумент*(размер_типа);

Пример:

```
Int n[4]={1,2,3,4};
Int *pn = n;
Int f[5]={6.0,7.0,8.0,9.0};
Int *pf = f;

Pn++;
Printf("%d", *pn); //на экране 2
Pf++;
Printf("%g", *pf); //на экране 7
Pn+=2;
Printf("%d", *pn); //на экране 4
Pf+=2;
Printf("%g", *pf); //на экране 9
```

Примечание: любые указатели можно сравнивать между собой.

Тема 5.2. Арифметика указателей

Указатели могут участвовать в арифметических операциях (сложение, вычитание, инкремент, декремент). Однако сами операции производятся немного иначе, чем с числами. И многое здесь зависит от типа указателя.

К указателю можно прибавлять целое число, и также можно вычитать из указателя целое число. Кроме того, можно вычитать из одного указателя другой указатель.

Рассмотрим вначале операции инкремента и декремента и для этого возьмем указатель на объект типа int:

```
#include <iostream>
 int main()
{
 int n = 10;

 int *ptr = &n;
 std::cout << "address=" << ptr << "\tvalue=" << *ptr << 
std::endl;

 ptr++;
 std::cout << "address=" << ptr << "\tvalue=" << *ptr << 
std::endl;

 ptr--;
 std::cout << "address=" << ptr << "\tvalue=" << *ptr << 
std::endl;
  return 0;
}
```

Операция инкремента ++ увеличивает значение на единицу. В случае с указателем увеличение на единицу будет означать увеличение адреса, который хранится в указателе, на размер типа указателя. То есть в данном случае указатель на тип int, а размер объектов int в большинстве архитектур равен 4 байтам. Поэтому увеличение указателя типа int на единицу означает увеличение значение указателя на 4.

И консольный вывод выглядит следующим образом:

```
address=0x60fe98        value=10
address=0x60fe9c        value=6356636
address=0x60fe98        value=10
```

Здесь видно, что после инкремента значение указателя увеличилось на 4: с 0x60fe98 до 0x60fe9c. А после декремента, то есть уменьшения на единицу, указатель получил предыдущий адрес в памяти.

Фактически увеличение на единицу означает, что мы хотим перейти к следующему объекту в памяти, который находится за текущим и на который указывает указатель. А уменьшение на единицу означает переход назад к предыдущему объекту в памяти.

После изменения адреса мы можем получить значение, которое находится по новому адресу, однако это значение может быть неопределенным, как показано в случае выше.

В случае с указателем типа int увеличение/уменьшение на единицу означает изменение адреса на 4. Аналогично, для указателя типа short эти операции изменяли бы адрес на 2, а для указателя типа char на 1.

```
#include <iostream>
 int main()
{
 double d = 10.6;
 double *pd = &d;
 std::cout << "Pointer pd: address:" << pd << std::endl;
 pd++;
 std::cout << "Pointer pd: address:" << pd << std::endl;

 char c = 'N';
 char *pc = &c;
 std::cout << "Pointer pc: address:" << (void*)pc << std::endl;
 pc++;
 std::cout << "Pointer pc: address:" << (void*)pc << std::endl;
  return 0;
}
```

Консольный вывод будет выглядеть следующим образом:

```
Pointer pd: address=0x60fe90
```

```
Pointer pd: address=0x60fe98
Pointer pc: address=0x60fe8f
Pointer pc: address=0x60fe90
```

Как видно из консольного вывода, увеличение на единицу указателя типа double дало увеличения хранимого в нем адреса на 8 единиц (размер объекта double - 8 байт), а увеличение на единицу указателя типа char дало увеличение хранимого в нем адреса на 1 (размер типа char - 1 байт).

Аналогично указатель будет изменяться при прибавлении/вычитании не единицы, а какого-то другого числа.

```
#include <iostream>

int main()
{
 double d = 10.6;
 double *pd = &d;
 std::cout << "Pointer pd: address:" << pd << std::endl;
 pd = pd + 2;
 std::cout << "Pointer pd: address:" << pd << std::endl;

 char c = 'N';
 char *pc = &c;
 std::cout << "Pointer pc: address:" << (void*)pc << std::endl;
 pc = pc - 3;
 std::cout << "Pointer pc: address:" << (void*)pc << std::endl;
 return 0;
}
```

Добавление к указателю типа double числа 2

```
pd = pd + 2;
```

означает, что мы хотим перейти на два объекта double вперед, что подразумевает изменение адреса на 2 * 8 = 16 байт.

Вычитание из указателя типа char числа 3

pc = pc - 3;

означает, что мы хотим перейти на три объекта char назад, что подразумевает изменение адреса на 3 * 1 = 3 байта.

И получается следующий консольный вывод:

```
Pointer pd: address=0x60fe90
Pointer pd: address=0x60fea0
Pointer pc: address=0x60fe8f
Pointer
```

В отличие от сложения операция вычитания может применять не только к указателю и целому числу, но и к двум указателям одного типа:

```
#include <iostream>

int main()
{
 int a = 10;
 int b = 23;
 int *pa = &a;
 int *pb = &b;
 int c = pa - pb;

 std::cout << "pa: " << pa << std::endl;
 std::cout << "pb: " << pb << std::endl;
 std::cout << "c: " << c << std::endl;

 return 0;
}
```

Консольный вывод:

```
pa: 0x60fe90
pb: 0x60fe8c
c: 1
```

Результатом разности двух указателей является «расстояние» между ними. Например, в случае выше адрес из первого указателя на 4 больше, чем адрес из второго указателя

(0x60fe8c + 4 = 0x60fe90). Так как размер одного объекта int равен 4 байтам, то расстояние между указателями будет равно (0x60fe90 - 0x60fe8c)/4 = 1.

Тема 5.3. Работа с динамической памятью

C++ поддерживает три основных типа выделения (распределения) памяти, с двумя из которых мы уже знакомы:

Статическое выделение памяти выполняется для статических и глобальных переменных. Память выделяется один раз, при запуске программы, и сохраняется на протяжении работы всей программы.

Автоматическое выделение памяти выполняется для параметров функции и локальных переменных. Память выделяется при входе в блок, в котором находятся эти переменные, и освобождается при выходе из него.

Как статическое, так и автоматическое распределение памяти имеют две общие черты:

– Размер переменной/массива должен быть известен во время компиляции.

– Выделение и освобождение памяти происходит автоматически (когда переменная создается или уничтожается).

В большинстве случаев с этим всё в порядке. Однако, когда дело доходит до работы с внешним вводом, то эти ограничения могут привести к проблемам.

Например, при использовании строки для хранения имени мы не знаем наперед насколько длинным оно будет, пока пользователь его не введет. Или когда нам нужно записать количество записей с диска в переменную, но мы не знаем заранее, сколько этих записей есть. Или мы можем создать игру с непостоянным количеством монстров (во время игры одни монстры умирают, другие рождаются), пытаясь, таким образом, убить игрока.

Если нам нужно объявить размер всех переменных во время компиляции, то самое лучшее, что мы можем сделать – это попытаться угадать их максимальный размер, надеясь, что этого будет достаточно:

char name[30]; // будем надеяться, что пользователь введет имя менее 30 символов!

Record record[400]; // будем надеяться, что количество записей будет не больше 400!

Monster monster[30]; // 30 монстров максимум

Polygon rendering[40000]; // этому 3d rendering лучше состоять из менее чем 40,000 полигонов!

Это плохое решение, по крайней мере, по трем причинам:

Во-первых, теряется память, если переменные фактически не используются или используются, но не на полную. Например, если мы выделим 30 символов для каждого имени, но имена в среднем будут занимать по 15 символов, то потребление памяти получится в два раза больше, чем нужно на самом деле. Или рассмотрим массив rendering: если он использует только 20 000 полигонов, то память с 20 000 полигонами фактически тратится впустую (то есть не используется)!

Во-вторых, память для большинства обычных переменных (включая фиксированные массивы) выделяется из специального резервуара памяти – стека. Объем памяти стека в программе, как правило, невелик – в Visual Studio он по умолчанию равен 1МБ. Если вы превысите это число, то произойдет *переполнение стека*, и операционная система автоматически завершит выполнение вашей программы.

В Visual Studio это можно проверить, запустив следующую программу:

int main()

{

int array[1000000]; // выделяем 1 миллион целочисленных значений }

Лимит в 1 МБ памяти может быть проблематичным для многих программ, особенно где используется графика.

В-третьих, и самое главное, это может привести к искусственным ограничениям и/или переполнению массива. Что произойдет, если пользователь попытается прочесть 500 записей с диска, но мы выделили память максимум для 400? Либо мы выведем пользователю ошибку, что максимальное количество записей – 400, либо (в худшем случае) выполнится переполнение массива и затем что-то очень нехорошее.

К счастью, эти проблемы легко устраняются с помощью динамического выделения памяти. Динамическое выделение памяти – это способ запроса памяти из операционной системы запущенными программами при необходимости. Эта память не выделяется из ограниченной памяти стека программы, а из гораздо большего хранилища, управляемого операционной системой – heap (кучи). На современных компьютерах размер кучи может составлять гигабайты памяти.

Динамическое выделение памяти

Для динамического выделения памяти для одной переменной используется оператор new:

new int; // динамически выделяем целочисленную переменную и сразу же отбрасываем результат (так как нигде его не сохраняем)

В примере выше мы запрашиваем выделение памяти для целочисленной переменной из операционной системы. Оператор new возвращает указатель, содержащий адрес выделенной памяти.

Для доступа к выделенной памяти создается указатель:

*int *ptr = new int; // динамически выделяем целочисленную переменную и присваиваем её адрес ptr, чтобы потом иметь возможность доступа к ней*

Затем мы можем разыменовать указатель для получения значения:

**ptr = 8; // присваиваем значение 8 только что выделенной памяти*

Вот один из случаев, когда указатели полезны. Без указателя с адресом на только что выделенную память, у нас бы не было способа получить доступ к ней.

Как работает динамическое выделение памяти?

На вашем компьютере имеется память (возможно, большая ее часть), которая доступна для использования приложениями. При запуске приложения ваша операционная система загружает это приложение в некоторую часть этой памяти. И эта память, используемая вашим приложением, разделена на несколько частей, каждая из которых выполняет определенную задачу. Одна часть содержит ваш код, другая используется для выполнения обычных операций (отслеживание вызываемых функций, создание и уничтожение глобальных и локальных переменных и т.д.). Мы поговорим об этом позже. Тем не менее, большая часть доступной памяти просто находится там, ожидая запросов на выделение от программ.

Когда вы динамически выделяете память, вы просите операционную систему зарезервировать часть этой памяти для использования вашей программой. Если ОС может выполнить этот запрос, то возвращается адрес этой памяти обратно, в ваше приложение. С этого момента и в дальнейшем ваше приложение может использовать эту память, как только пожелает. Когда вы уже выполнили всё, что было необходимо, с этой памятью, то её нужно вернуть обратно в операционную систему, для распределения между другими запросами.

В отличие от статического или автоматического выделения памяти, сама программа отвечает за запрос и обратный возврат динамически выделенной памяти.

Инициализация динамически выделенных переменных

Когда вы динамически выделяете переменную, то вы также можете её инициализировать посредством прямой инициализации или uniform инициализации (в C++):

*int *ptr1 = new int (7); // используем прямую инициализацию*

*int *ptr2 = new int { 8 }; // используем uniform инициализацию*

Удаление переменных

Когда уже всё, что нужно было, выполнено с динамически выделенной переменной – нужно явно указать C++ освободить эту память. Для отдельных переменных это выполняется с помощью оператора delete:

// предположим, что ptr ранее уже был выделен с помощью оператора new

delete ptr; // возвращаем память, на которую указывал ptr, обратно в операционную систему

ptr = 0; // делаем ptr нулевым указателем (используйте nullptr вместо 0 в C++11)

Что означает «удаление памяти»?

Оператор delete на самом деле ничего не удаляет. Он просто возвращает память, которая была выделена ранее, обратно в операционную систему. Затем операционная система может переназначить эту память другому приложению (или этому же снова).

Хотя может показаться, что мы удаляем *переменную*, но это не так! Переменная-указатель по-прежнему имеет ту же область видимости, что и раньше, и ей можно присвоить новое значение, как и любой другой переменной.

Обратите внимание, удаление указателя, не указывающего на динамически выделенную память, может привести к проблемам.

Висячие указатели

C++ не дает никаких гарантий относительно того, что произойдет с содержимым освобожденной памяти, или со значением удаляемого указателя. В большинстве случаев

память, возвращаемая операционной системе, будет содержать те же значения, которые были у нее до *освобождения*, а указатель так и останется указывать на память, только уже освобожденную (удаленную).

Указатель, указывающий на освобожденную память, называется висячим указателем. Разыменование или удаление висячего указателя приведет к неожиданным результатам. Рассмотрим следующую программу:

```
#include <iostream>
 int main()
{
 int *ptr = new int; // динамически выделяем целочисленную переменную
 *ptr = 8; // помещаем значение в выделенную ячейку памяти

 delete ptr; // возвращаем память обратно в операционную систему. ptr теперь уже висячий указатель

 std::cout << *ptr; // разыменование висячого указателя приведет к неожиданным результатам
 delete ptr; // попытка освободить память снова приведет к неожиданным результатам также
 return 0;
}
```

В программе выше значение 8, которое ранее было присвоено выделенной памяти, после освобождения может и далее находиться там, а может и нет. Также возможно, что освобожденная память уже могла быть выделена другому приложению (или для собственного использования операционной системы), и попытка доступа к ней приведет к тому, что операционная система автоматически прекратит выполнение вашей программы.

Процесс освобождения памяти может также привести к созданию *нескольких* висячих указателей. Рассмотрим следующий пример:

```
#include <iostream>
 int main()
{
 int *ptr = new int; // динамически выделяем целочисленную переменную
 int *otherPtr = ptr; // otherPtr теперь указывает на ту же самую выделенную память, что и ptr
 delete ptr; // возвращаем память обратно в операционную систему. ptr и otherPtr теперь висячие указатели
 ptr = 0; // ptr теперь уже nullptr
 // однако otherPtr по-прежнему является висячим указателем!
 return 0;
}
```

Есть несколько рекомендаций, которые могут здесь помочь:

Во-первых, старайтесь избегать ситуаций, когда несколько указателей указывают на одну и ту же часть выделенной памяти. Если это невозможно, то проясните, какой указатель из всех «владеет» памятью (и отвечает за ее удаление), а какие указатели просто получают доступ к ней.

Во-вторых, когда вы удаляете указатель, и если он не выходит из области видимости сразу же после удаления, то его нужно сделать нулевым, т.е. задать значение 0 (или nullptr в C++11). Под «выходом из области видимости сразу же после удаления» имеется в виду, что вы удаляете указатель в самом конце блока, в котором он объявлен.

Правило: Присваивайте удаленным указателям значение 0, если они не выходят из области видимости сразу же после удаления.

Работа оператора new

При запросе памяти из операционной системы в редких случаях она может быть не доступной (т.е. её может и не быть в наличии).

По умолчанию, если new не сработал, память не выделилась, то генерируется исключение *bad_alloc* . Если это исключение будет неправильно обрабатываться (а именно так и будет, поскольку мы еще не рассмотрели исключения и их обработку), то программа просто прекратит своё выполнение (произойдет сбой) с необработанной ошибкой исключения.

Во многих случаях процесс генерации исключения оператором new (как и сбой программы) нежелателен, поэтому есть альтернативная форма new, которая возвращает нулевой указатель, если память не может быть выделена. Нужно просто добавить константу std::nothrow между ключевым словом new и типом выделения данных:

*int *value = new (std::nothrow) int; // указатель value станет нулевым, если динамическое выделение целочисленной переменной не выполнится*

В примере выше, если new не возвратит указатель с динамически выделенной памятью, то возвратится нулевой указатель.

Разыменовывать его также не рекомендуется, так как это приведет к неожиданным результатам (скорее всего, к сбою в программе). Поэтому, наилучшей практикой является проверка всех запросов на выделение памяти, для обеспечения того, что эти запросы выполняться успешно и память будет выделена.

*int *value = new (std::nothrow) int; // запрос на выделение динамической памяти для целочисленного значения*

if (!value) // обрабатываем случай, когда new возвращает null (т.е. память не выделяется)

{

// обработка этого случая

std::cout << "Could not allocate memory";
}

Поскольку не выделение памяти оператором new происходит крайне редко, то обычно программисты забывают выполнять эту проверку!

Нулевые указатели и динамическое выделение памяти

Нулевые указатели (указатели со значением 0 или nullptr) особенно полезны в процессе выделения динамической памяти. Их наличие как бы говорит: «этому указателю не выделено никакой памяти». А это в свою очередь можно использовать для выполнения условного выделения памяти:

// если ptr-у до сих пор не выделено памяти, выделяем её
if (!ptr)
ptr = new int;

Удаление нулевого указателя ни на что не влияет. Таким образом, в следующем нет необходимости:

if (ptr)
delete ptr;

Вместо этого вы можете просто написать:

delete ptr;

Если ptr не является нулевым, то динамически выделенная переменная будет удалена. Если значением указателя является нуль, то ничего не произойдет.

Утечка памяти

Динамически выделенная память не имеет области видимости. То есть она остается выделенной до тех пор, пока не будет явно освобождена или пока ваша программа не завершится (и операционная система очистит все буфера памяти самостоятельно). Однако указатели, используемые для хранения динамически выделенных адресов памяти, следуют правилам области видимости нормальных переменных. Это несоответствие может вызвать интересное поведение.

Рассмотрим следующую функцию:

void doSomething()

```
{
 int *ptr = new int;
}
```

Здесь мы динамически выделяем целочисленную переменную, но никогда не освобождаем память через delete. Поскольку указатели следуют всем тем же правилам, что и обычные переменные, то когда функция завершится, ptr выйдет из области видимости. Поскольку ptr – это единственная переменная, хранящая адрес динамически выделенной целочисленной переменной, то когда ptr уничтожиться – больше не останется ссылок (адресов) на динамически выделенную память. Это означает, что программа «потеряет» адрес динамически выделенной памяти. И в результате эту динамически выделенную целочисленную переменную нельзя будет удалить.

Это называется утечкой памяти. Утечка памяти происходит, когда ваша программа теряет адрес некоторой динамически выделенной части памяти (например, переменной или массива), прежде чем вернуть её обратно в операционную систему. Когда это происходит, то программа уже не может удалить эту динамически выделенную память, поскольку она больше не знает, где та находится. Операционная система также не может использовать эту память, поскольку считается, что та по-прежнему используется вашей программой.

Утечки памяти съедают свободную память во время выполнения программы, уменьшая количество доступной памяти не только для этой программы, но и для других программ также. Программы с серьезными проблемами с утечкой памяти могут съесть всю доступную память, в результате чего весь ваш компьютер будет медленнее работать или даже произойдет сбой. Только после того, как выполнение вашей программы завершится, операционная

система сможет очистить и *вернуть* всю просочившуюся память.

Хотя утечка памяти может возникнуть и из-за того, что указатель выходит из области видимости, возможны и другие способы, которые могут привести к утечкам памяти. Например, если указателю, хранящему адрес динамически выделенной памяти, присвоить другое значение:

```
int value = 7;
int *ptr = new int; // выделяем память
ptr = &value; // старый адрес утерян - произойдет утечка памяти
```

Это легко решается удалением указателя перед операцией переприсваивания:

```
int value = 7;
int *ptr = new int; // выделяем память
delete ptr; // возвращаем память обратно в операционную систему
ptr = &value; // переприсваиваем указателю адрес value
```

Кроме того, утечка памяти также может произойти и через двойное выделение памяти:

```
int *ptr = new int;
ptr = new int; // старый адрес утерян - произойдет утечка памяти
```

Адрес, возвращаемый из второго выделения памяти, перезаписывает адрес из первого выделения. Следовательно, первое динамическое выделение становится утечкой памяти!

Точно так же этого можно избежать удалением указателя перед операцией переприсваивания.

Раздел 6. Функции и функциональное программирование

Тема 6.1 Функция. Виды функций

Использование функций дает много преимуществ, в том числе:

– код внутри функции может быть повторно использован;

– гораздо проще изменить или обновить код в функции (что делается один раз), нежели искать и изменять все части кода в main() на месте. Дублированный код – хороший рецепт для ошибок и уменьшения производительности;

– упрощается чтение и понимание кода, так как вам не нужно знать, как реализована функция, чтобы понять, что она делает (предполагается наличие информативного названия функции и комментариев);

– в функциях поддерживается проверка типов данных для гарантии того, что передаваемые аргументы соответствуют параметрам функции;

– функции упрощают отладку вашей программы.

Однако одним из главных недостатков использования функций является то, что каждый раз, когда она вызывается, происходит расход ресурсов, что влияет на производительность программы. Это связано с тем, что ЦП должен хранить адрес текущей команды (инструкции или стейтмента), которую он выполняет (чтобы знать, куда нужно будет вернуться позже) вместе с другими данными. Затем точка выполнения перемещается в другое место программы. Дальше все параметры функции должны быть созданы и им должны быть присвоены значения. И только потом, после выполнения функции, точка выполнения возвращается обратно. Код, написанный на месте, выполняется значительно быстрее.

Для функций, которые являются большими и/или выполняют сложные задачи, расходы на вызов обычно незначительны по сравнению с количеством времени, которое отводится на выполнение кода этой функции. Однако для небольших, часто используемых функций, время, необходимое для выполнения вызова, часто превышает время, необходимое для фактического выполнения кода этой функции. А это в свою очередь может привести к существенному снижению производительности.

C++ предлагает способ сочетания преимуществ функций со скоростью кода, написанного на месте: встроенные функции. Ключевое слово inline используется для запроса, чтобы компилятор рассматривал вашу функцию как встроенную. При компиляции вашего кода, все встроенные функции раскрываются на месте, то есть вызов функции заменяется копией содержимого самой функции, и ресурсы, которые могли бы быть потрачены на вызов этой функции, сохраняются! Минусом является лишь увеличение компилируемого кода за счет того, что встроенная функция раскрывается в коде при каждом вызове (особенно если она длинная и/или её вызывают много раз).

Рассмотрим следующий фрагмент кода:

```
int max(int a, int b)
{
 return a < b ? b : a;
}
 int main()
{
 std::cout << max(7, 8) << '\n';
 std::cout << max(5, 4) << '\n';
 return 0;
}
int max(int a, int b)
{
```

```
 return a < b ? b : a;
}
 int main()
{
 std::cout << max(7, 8) << '\n';
 std::cout << max(5, 4) << '\n';
 return 0;
}
```

Эта программа дважды вызывает функцию max(), т.е. дважды расходуются ресурсы на вызов функции. Поскольку max() является довольно таки короткой, то это идеальный кандидат для конвертации во встроенную функцию:

```
inline int max(int a, int b)
{
 return a < b ? b : a;
}
inline int max(int a, int b)
{
 return a < b ? b : a;
}
```

Теперь, при компиляции main(), ЦП будет читать этот код так, как если бы он был бы следующим:

```
int main()
{
 std::cout << (7 < 8 ? 8 : 7) << '\n';
 std::cout << (5 < 4 ? 4 : 5) << '\n';
 return 0;
}
int main()
{
 std::cout << (7 < 8 ? 8 : 7) << '\n';
 std::cout << (5 < 4 ? 4 : 5) << '\n';
 return 0;
}
```

Такой код выполниться быстрее, но цена – несколько увеличенный объем.

Из-за возможности подобного «раздувания», встроенные функции лучше всего использовать только для коротких функций (не более нескольких строк), которые обычно вызываются внутри циклов и не имеют ветвлений. Также обратите внимание, ключевое слово inline является только рекомендацией – компилятор может игнорировать ваш запрос на встроенную функцию. Подобное произойдет, если вы попытаетесь сделать встроенной длинную функцию!

Наконец, современные компиляторы автоматически конвертируют соответствующие функции во встроенные – этот процесс автоматизирован настолько, что даже лучше ручной выборочной конвертации программистом. Даже если вы не отметите функцию как встроенную, компилятор автоматически выполнит её как таковую, если посчитает, что это способствует производительности. Таким образом, в большинстве случаев нет особой необходимости использовать ключевое слово inline. Компилятор всё сделает сам.

Правило: Если вы используете современный компилятор, то нет необходимости использовать ключевое слово inline.

Встроенные функции освобождаются от правила одного определения

В предыдущих главах мы не раз говорили, что вы не должны определять функции в заголовочных файлах, так как если вы подключите один заголовок с определением функции в несколько файлов .cpp, то определение функции также будет скопировано несколько раз. Затем при соединении файлов линкер выдаст ошибку, что вы определяете одну и ту же функцию больше одного раза.

Однако встроенные функции освобождаются от этого правила, так как дублирования в исходном коде нет – определение функции одно, и никакого конфликта при соединении линкером файлов .cpp возникнуть не должно.

Сейчас это всё может показаться неинтересными пустяками, но в следующей главе, когда мы будем рассматривать новый тип функций (дружественные функции), эти пустяки пригодятся.

И помните, что, даже с использованием встроенных функций, вы не должны определять глобальные функции в заголовочных файлах.

Создание пользовательских функций

Вам, конечно, для начала интересно, а зачем нужно вообще изучать создание пользовательских функций, когда есть множество функций, встроенных в нашу среду разработки. Ведь первые уроки мы пользовались только ими.

Но у вас чаще всего будут стоять задачи, которые потребуют нетривиальных решений. Т.е. обычными функциями поставленную цель будет решить очень сложно. Т.е. вывод таков: для решения общих задач вполне можно обойтись и функциями, созданными другими программистами, а вот для решения более узконаправленных задач уже нужны пользовательские функции.

Мы забыли сказать, что означает пользовательская функция. Это функция, которая создана пользователем. Вот вам парочку примеров, из задания, где вам нужно было дать названия функциям:

```
void swap (int x, int y){
 int z = x;
 x = y;
 y = z;
}
```

Как вы, наверное, догадались - это пользовательская функция, которая меняет местами значения у элементов

```
void GetError (){
 pritnf ("Ошибка\n");
}
```

Пользовательская функция, выбрасывающая сообщение об ошибке:

```
int max (int a, int b, int c){
 if (a > b && a > c)
 return a;
 if (b > a && b > c)
 return b;
 if (c > a && c > b)
 return c;
}
```

Эта созданная пользовательская функция определяет наибольшее число из трех.

Но есть одно важное замечание. В нашей пользовательской функции swap, которая меняет местами значения у элементов. На самом деле значения, после выполнении функции останутся на своем месте, так как в саму функцию передаются копии объектов. Как передавать в функцию сам объект мы рассмотрим позднее

Рекурсия – это вид функции, которая вызывает саму себя. Давайте для начала рассмотрим пример рекурсии:

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

int sum (int temp){
 int i=0;
 if (temp == 0)
 return temp;
 else
 return temp+sum(temp-1);
}
int main()
{
 printf ("%d", sum(5));
 getch();
```

return 0;
}

Этот пример рекурсивной функции (рекурсии) выводит сумму чисел до введенной в аргументе. Суть рекурсивной функции (рекурсии) тут состоит в том, что пока наше число не равно нулю, мы будем накапливать полученное число в аргументе и передавать в эту же функцию число, но на единицу меньшее. Т.е. на первом шаге мы в накоплении оставим 5, а передадим 4, далее мы к 5 прибавляем 4, а передаем 3. Вот как действует наша рекурсия.

Тема 6.2 Возвращаемые значения и параметры функции. Передача значений

Указатель на функцию (function pointer) хранит адрес функции. По сути указатель на функцию содержит адрес первого байта в памяти, по которому располагается выполняемый код функции.

Самым распространенным указателем на функцию является ее имя. С помощью имени функции можно вызывать ее и получать результат ее работы.

Но также указатель на функцию мы можем определять в виде отдельной переменной с помощью следующего синтаксиса:

*тип (*имя_указателя) (параметры);*

Здесь *тип* представляет тип возвращаемого функцией значения.

имя_указателя представляет произвольно выбранный идентификатор в соответствии с правилами о наименовании переменных.

И *параметры* определяют тип и название параметров через запятую при их наличии.

Например, определим указатель на функцию:

*void (*message) ();>*

В данном случае определен указатель, который имеет имя message. Он может указывать на функции без параметров, которые возвращают тип void (то есть ничего не возвращают).

Используем указатель на функцию:

```
#include <iostream>

void hello();
void goodbye();

int main()
{
 void (*message)();

 message=hello;
 message();
 message = goodbye;
 message();

 return 0;
}
void hello()
{
 std::cout << "Hello, World" << std::endl;
}
void goodbye()
{
 std::cout << "Good Bye, World" << std::endl;
}
```

Указателю на функцию можно присвоить функцию, которая соответствует указателю по возвращаемому типу и спецификации параметров:

```
message=hello;
```

То есть в данном случае указатель message теперь хранит адрес функции hello. И посредством обращения к указателю мы можем вызвать эту функцию:

message();

В качестве альтернативы мы можем обращаться к указателю на функцию следующим образом:

*(*message)();*

Впоследствии мы можем присвоит указателю адрес другой функции, как в данном случае. В итоге результатом данной программы будет следующий вывод:

```
Hello, World
Good Bye, World
```

При определении указателя стоит обратить внимание на скобки вокруг имени. Так, использованное выше определение

*void (*message) ();*

НЕ будет аналогично следующему определению:

*void *message ();*

Во втором случае определен не указатель на функцию, а прототип функции message, которая возвращает указатель типа void*****.

Рассмотрим еще один указатель на функцию:

```
#include <iostream>

int add(int, int);
int subtract(int, int);

int main()
{
 int a = 10;
 int b = 5;
 int result;
 int (*operation)(int a, int b);

 operation=add;
```

```
    result = operation(a, b);
    // result = (*operation)(a, b); // альтернативный вариант
    std::cout << "result=" << result << std::endl; // result=15

    operation = subtract;
    result = operation(a, b);
    std::cout << "result=" << result << std::endl; // result=5

    return 0;
}
int add(int x, int y)
{
    return x+y;
}
int subtract(int x, int y)
{
    return x-y;
}
```

Здесь определен указатель operation, который может указывать на функцию с двумя параметрами типа int, возвращающую также значение типа int. Соответственно мы можем присвоить указателю адреса функций add и subtract и вызвать их, передав при вызове указателю некоторые значения для параметров.

Массивы указателей на функции

Кроме одиночных указателей на функции мы можем определять их массивы. Для этого используется следующий формальный синтаксис:

*тип (*имя_массива[размер]) (параметры)*

Например:

*double (*actions[]) (int, int)*

Здесь actions представляет массив указателей на функции, каждая из которых обязательно должна принимать два параметра типа int и возвращать значение типа double.

Посмотрим применение массива указателей на функции на примере:

```
#include <iostream>

void add(int, int);
void subtract(int, int);
void multiply(int, int);

int main()
{
 int a = 10;
 int b = 5;
 void (*operations[3])(int, int) = {add, subtract, multiply};

 // получаем длину массива
 int length = sizeof(operations)/sizeof(operations[0]);

 for(int i=0; i < length;i++)
 {
 operations[i](a, b); // вызов функции по указателю
 }

 return 0;
}
void add(int x, int y)
{
 std::cout << "x + y = " << x + y << std::endl;
}
void subtract(int x, int y)
{
 int result = x - y;
 std::cout << "x - y = " << x - y << std::endl;
}
void multiply(int x, int y)
```

```
{
 std::cout << "x * y = " << x * y << std::endl;
}
```

Здесь массив operations содержит три функции add, subtract и multiply, которые последовательно вызываются в цикле через перебор массива в функции main.

Консольный вывод программы:

```
x + y = 15
x - y = 5
x * y = 50
```

Формальные параметры

Если функция использует аргументы, то в ней должны объявляться переменные, которые будут принимать значения аргументов. Данные переменные называются формальными параметрами функции. Они ведут себя, как любые другие локальные переменные в функции. Как показано в следующем фрагменте программы, они объявляются в круглых скобках, следующих за именем функции.

```
/* возвращает 1, если с является частью строки s; в противном случае - 0 */

int is_in (char *s, char c) {
while(*s)
if(*s==c) return 1;
else s++;
return 0;
}
```

Функция is_in() имеет два параметра: s и c. Необходимо сообщить компилятору тип этих переменных, как показано выше. Когда это сделано, они могут использоваться в функции как обычные локальные переменные. Надо помнить, что локальные переменные также являются динамическими и уничтожаются при выходе из функции.

Как и с локальными переменными, формальным параметрам можно присваивать значения или использовать их в любых допустимых выражениях С. Даже если эти переменные играют особую роль для некоторых задач по получению значений аргументов, переданных в функцию, то они могут использоваться, как и остальные локальные переменные.

Тема 6.4 Локальные и глобальные переменные

В этой главе обсуждаются два важных вопроса, касающиеся объявлений в C++. Где употребляется объявленное имя? Когда можно безопасно использовать объект или вызывать функцию, т.е. каково время жизни сущности в программе? Для ответа на первый вопрос мы введем понятие областей видимости и покажем, как они ограничивают применение имен в исходном файле программы. Мы рассмотрим разные типы таких областей: глобальную и локальную, а также более сложное понятие областей видимости пространств имен, которое появится в конце главы. Отвечая на второй вопрос, мы опишем, как объявления вводят глобальные объекты и функции (сущности, «живущие» в течение всего времени работы программы), локальные («живущие» на определенном отрезке выполнения) и динамически размещаемые объекты (временем жизни которых управляет программист). Мы также исследуем свойства времени выполнения, характерные для этих объектов и функций.

Область видимости

Каждое имя в C++ программе должно относиться к уникальной сущности (объекту, функции, типу или шаблону). Это не значит, что оно встречается только один раз во всей программе: его можно повторно использовать для обозначения другой сущности, если только есть некоторый *контекст*, помогающий различить разные значения одного и

того же имени. Контекстом, служащим для такого различения, служит *область видимости*. В C++ поддерживается три их типа: *локальная* область видимости, область видимости *пространства имен* и область видимости *класса*.

Локальная область – это часть исходного текста программы, содержащаяся в определении функции (или в блоке). Любая функция имеет собственную такую часть, и каждая составная инструкция (или блок) внутри функции также представляет собой отдельную локальную область.

Область видимости пространства имен – часть исходного текста программы, не содержащаяся внутри объявления или определения функции или определения класса. Самая внешняя часть называется глобальной областью видимости или глобальной областью видимости пространства имен.

Объекты, функции, типы и шаблоны могут быть определены в глобальной области видимости. Программисту разрешено задать *пользовательские* пространства имен, заключенные внутри глобальной области с помощью *определения пространства имен*. Каждое такое пространство является отдельной областью видимости. Пользовательское пространство, как и глобальное, может содержать объявления и определения объектов, функций, типов и шаблонов, а также вложенные пользовательские пространства имен.

Каждое определение класса представляет собой *отдельную область видимости класса*.

Имя может обозначать различные сущности в зависимости от области видимости. В следующем фрагменте программы имя s1 относится к четырем разным сущностям:

```
#include <iostream>
#include <string>
// сравниваем s1 и s2 лексикографически
int lexicoCompare( const string &sl, const string &s2 ) { ... }

// сравниваем длины s1 и s2
```

```
int sizeCompare( const string &sl, const string &s2 ) { ... }

typedef int ( PFI)( const string &, const string & );
// сортируем массив строк
void sort( string *s1, string *s2, PFI compare =lexicoCompare
)
 { ... }

string sl[10] = { "a", "light", "drizzle", "was", "falling",
 "when", "they", "left", "the", "school" };
int main()
{
 // вызов sort() со значением по умолчанию параметра
compare
 // s1 - глобальный массив
 sort( s1, s1 + sizeof(s1)/sizeof(s1[0]) - 1 );

 // выводим результат сортировки
 for ( int i = 0; i < sizeof(s1) / sizeof(s1[0]); ++i )
 cout << s1[ i ].c_str() << "\n\t";
}
```

Поскольку определения функций lexicoCompare(), sizeCompare() и sort() представляют собой различные области видимости и все они отличны от глобальной, в каждой из этих областей можно завести переменную с именем s1.

Имя, введенное с помощью объявления, можно использовать от точки объявления до конца области видимости (включая вложенные области). Так, имя s1 параметра функции lexicoCompare() разрешается употреблять до конца ее области видимости, то есть до конца ее определения.

Имя глобального массива s1 видимо с точки его объявления до конца исходного файла, включая вложенные области, такие, как определение функции main().

В общем случае имя должно обозначать одну сущность внутри одной области видимости. Если в предыдущем примере после объявления массива s1 добавить следующую строку, компилятор выдаст сообщение об ошибке:

void s1(); // ошибка: повторное объявление s1

Перегруженные функции являются исключением из правила: можно завести несколько одноименных функций в одной области видимости, если они отличаются списком параметров.

В C++ имя должно быть объявлено до момента его первого использования в выражении. В противном случае компилятор выдаст сообщение об ошибке. Процесс сопоставления имени, используемого в выражении, с его объявлением называется разрешением. С помощью этого процесса имя получает конкретный смысл. Разрешение имени зависит от способа его употребления и от его области видимости. Мы рассмотрим этот процесс в различных контекстах.

Области видимости и разрешение имен – понятия времени компиляции. Они применимы к отдельным частям текста программы. Компилятор интерпретирует текст программы согласно правилам областей видимости и правилам разрешения имен.

Локальная область видимости

Локальная область видимости – это часть исходного текста программы, содержащаяся в определении функции (или блоке внутри тела функции). Все функции имеют свои локальные области видимости. Каждая составная инструкция (или блок) внутри функции также представляет собой отдельную локальную область. Такие области могут быть вложенными. Например, следующее определение функции содержит два их уровня (функция выполняет двоичный поиск в отсортированном векторе целых чисел):

const int notFound = -1; // глобальная область видимости

```
    int binSearch( const vector<int> &vec, int val )
{ // локальная область видимости: уровень #1
 int low = 0;
 int high = vec.size() - 1;
 while ( low <= high )
 { // локальная область видимости: уровень #2
 int mid = ( low + high ) / 2;
 if ( val < vec[ mid ] )
 high = mid - 1;
 else low = mid + 1;
 }
 return notFound; // локальная область видимости: уровень #1
}
```

Первая локальная область видимости – тело функции binSearch(). В ней объявлены параметры функции vec и val, а также переменные low и high. Цикл while внутри функции задает вложенную локальную область, в которой определена одна переменная mid. Параметры vec и val и переменные low и high видны во вложенной области. Глобальная область видимости включает в себя обе локальных. В ней определена одна целая константа notFound.

Имена параметров функции vec и val принадлежат к первой локальной области видимости тела функции, и в ней использовать те же имена для других сущностей нельзя. Например:

```
    int binSearch( const vector<int> &vec, int val )
    { // локальная область видимости: уровень #1
     int val; // ошибка: неверное переопределение val
     // ...
```

Имена параметров употребляются как внутри тела функции binSearch(), так и внутри вложенной области видимости цикла while. Параметры vec и val недоступны вне тела функции binSearch().

Разрешение имени в локальной области видимости происходит следующим образом: просматривается та область, где оно встретилось. Если объявление найдено, имя разрешено. Если нет, просматривается область видимости, включающая текущую. Этот процесс продолжается до тех пор, пока объявление не будет найдено либо не будет достигнута глобальная область видимости. Если и там имени нет, оно будет считаться ошибочным.

Из-за порядка просмотра областей видимости в процессе разрешения имен объявление из внешней области может быть скрыто объявлением того же имени во вложенной области. Если бы в предыдущем примере переменная low была объявлена в глобальной области видимости перед определением функции binSearch(), то использование low в локальной области видимости цикла while все равно относилось бы к локальному объявлению, скрывающему глобальное:

```
int low;
int binSearch( const vector<int> &vec, int val )
{
 // локальное объявление low
 // скрывает глобальное объявление
 int low = 0;
 // ...
 // low - локальная переменная
 while ( low <= high )
 {//...
 }
 // ...
}
```

Для некоторых инструкций языка C++ разрешено объявлять переменные внутри управляющей части. Например, в цикле for переменную можно определить внутри инструкции инициализации:

```
for ( int index = 0; index < vecSize; ++index )
{
 // переменная index видна только здесь
 if ( vec[ index ] == someValue )
 break;
}
// ошибка: переменная index не видна
if ( index != vecSize ) // элемент найден
```

Подобные переменные видны только в локальной области самого цикла for и вложенных в него (это верно для стандарта C++, в предыдущих версиях языка поведение было иным). Компилятор рассматривает это объявление так же, как если бы оно было записано в виде:

```
// представление компилятора
{ // невидимый блок
 int index = 0;
 for ( ; index < vecSize; ++index )
 {
 // ...
 }
}
```

Тем самым программисту запрещается применять управляющую переменную вне локальной области видимости цикла. Если нужно проверить index, чтобы определить, было ли найдено значение, то данный фрагмент кода следует переписать так:

```
int index = 0;
for ( ; index < vecSize; ++index )
{
 // ...
}
// правильно: переменная index видна
if ( index != vecSize ) // элемент найден
```

Поскольку переменная, объявленная в инструкции инициализации цикла for, является локальной для цикла, то же самое имя допустимо использовать аналогичным образом и в других циклах, расположенных в данной локальной области видимости:

```
void fooBar( int *ia, int sz )
{
 for (int i=0; i<sz; ++i) ... // правильно
 for (int i=0; i<sz; ++i) ... // правильно, другое i
 for (int i=0; i<sz; ++i) ... // правильно, другое i
}
```

Аналогично переменная может быть объявлена внутри условия инструкций if и switch, а также внутри условия циклов while и for. Например:

```
if ( int *pi = getValue() )
{
 // pi != 0 -- *pi можно использовать здесь
 int result = calc(*pi);
 // ...
}
else
{
 // здесь pi тоже видна
 // pi == 0
 cout << "ошибка: getValue() завершилась неудачно" << endl;
}
```

Переменные, определенные в условии инструкции if, как переменная pi, видны только внутри if и соответствующей части else, а также во вложенных областях. Значением условия является значение этой переменной, которое она получает в результате инициализации. Если pi равна 0 (нулевой указатель), условие ложно и выполняется ветвь else.

Если pi инициализируется любым другим значением, условие истинно и выполняется ветвь if.

Тема 6.5 Время жизни и видимость переменных, классы памяти

В языках C и C++ определенны правила видимости, которые устанавливают такие понятия как область видимости и время жизни объектов. Различают глобальную область видимости и локальную.

Глобальная область видимости существует вне всех других областей. Имя, объявленное в глобальной области, известно всей программе. Например, глобальная переменная доступна для использования всеми функциями программы. Глобальные переменные существуют на протяжении всего жизненного цикла программы.

Локальная область видимости определяется границами блока. Имя объявленное внутри локальной области,известно только внутри этой области. Локальные переменные создаются при входе в блок и разрушаются при выходе из него. Это означает, что локальные переменные не хранят своих значений между вызовами функций. Чтобы сохранить значения переменных между вызовами, можно использовать модификатор static.

Тема 6.6 Функция main: параметры и возвращаемое значение

Объявление функций: прототипы функций
returnDataType functionName(dataType argName1, dataType argName2, ..., dataType argNameN);
где,

- returnDataType – возвращаемый тип данных;
- functionName – имя функции;
- dataType – тип данных;

– argName1...N – имена параметров функции (количество параметров неограниченно).

Смотрим пример объявления функции:

// Объявление прототипа функции с двумя целыми параметрами

// функция принимает два аргумента и возвращает их сумму

int sum(int num1, int num2);

В языках C и C++, функции должны быть объявлены до момента их вызова. Вы можете объявить функцию, при этом функция может возвращать значение или – нет, имя функции присваивает программист, типы данных параметров указываются в соответствии с передаваемыми в функцию значениями. Имена аргументов, при объявления прототипов являются необязательными:

int sum(int , int); // тот же прототип функции

Иногда, объявление функции называют определением функции, хотя это не одно и то же.

Определение функций

returnDataType functionName(dataType argName1, dataType argName2, ..., dataType argNameN)

{

// тело функции

}

Рассмотрим определение функции на примере функции sum.

// определение функции, которая суммирует два целых числа и возвращает их сумму

int sum(int num1, int num2)

{

return (num1 + num2);

}

В языках C и C++, функции не должны быть определены до момента их использования, но они должны быть ранее

объявлены. Но даже после всего этого, в конце концов, эта функция должна быть определена. После этого прототип функции и ее определение связываются, и эта функция может быть использована.

Если функция ранее была объявлена, она должна быть определена с тем же возвращаемым значением и типами данных, в противном случае, будет создана новая, перегруженная функция. Заметьте, что имена параметров функции не должны быть одинаковыми.

```
// объявление функции суммирования
int sum(int, int);

// определение функции суммирования
int sum(int num1, int num2)
{
   return (num1 + num2);
}
```

Вызов функций

После того, как функция была объявлена и определена, её можно использовать, для этого её нужно вызвать. Вызов функции выполняется следующим образом:

funcName(arg1, arg2, ...);

где, funcName – имя функции; arg1..2 – аргументы функции (значения или переменные).

Тема 6.7 Указатели. Рекурсия

В языках C и C++ функции могут вызывать сами себя. Этот процесс называют рекурсией, а функцию, которая сама себя вызывает – рекурсивной. В качестве примера приведём функцию fact(), вычисляющую факториал целого числа.

```
int fact (int n)
{
   int ans;
   if (n == 1) return 1;
```

```
    ans = fact (n-1) * n;
    return ans;
}
```

Ключевое слово this представляет указатель на текущий объект данного класса. Соответственно через this мы можем обращаться внутри класса к любым его членам.

```
#include <iostream>
class Point
{
public:
    Point(int x, int y)
    {
        this->x = x;
        this->y = y;
    }
    void showCoords()
    {
        std::cout << "Coords x: " << this->x << "\t y: " << y << std::endl;
    }
private:
    int x;
    int y;
};

int main()
{
    Point p1(20, 50);
    p1.showCoords();

    return 0;
}
```

В данном случае определен класс Point, который представляет точку на плоскости. И для хранения координат точки в классе определены переменные x и y.

Для обращения к переменным используется указатель this. Причем после this ставится не точка, а стрелка ->.

В большинстве случаев для обращения к членам класса вряд ли поднадобится ключевое слово this. Но оно может быть необходимо, если параметры функции или переменные, которые определяются внутри функции, называются также как и переменные класса. К примеру, чтобы в конструкторе разграничить параметры и переменные класса как раз и используется указатель this.

С помощью this можно возвращать текущий объект класса:

```
#include <iostream>

class Point
{
public:
    Point(int x, int y)
    {
        this->x = x;
        this->y = y;
    }
    void showCoords()
    {
        std::cout << "Coords x: " << x << "\t y: " << y << std::endl;
    }
    Point &move(int x, int y)
    {
        this->x += x;
        this->y += y;
        return *this;
    }
private:
```

```
    int x;
    int y;
};

int main()
{
    Point p1(20, 50);
    p1.move(10, 5).move(10, 10);
    p1.showCoords(); // x: 40  y: 65
     return 0; }
```

Здесь метод move с помощью указателя this возвращает ссылку на объект текущего класса, осуществляя условное перемещение точки. Таким образом, мы можем по цепочке для одного и того же объекта вызывать метод move:

```
p1.move(10, 5).move(10);
```

Здесь также важно отметить возвращение не просто объекта Point, а ссылки на этот объект. Так, в данном случае выге определенная строка фактически будет аналогично следующему коду:

```
p1.move(10, 5);
p1.move(10, 10);
```

Но если бы метод move возвращал бы не ссылку, а посто объект:

```
Point move(int x, int y)
{
    this->x += x;
    this->y += y;
    return *this;
}
```

То вызов p1.move(10, 5).move(10) был бы фактически эквивалентен следующему коду:

```
Point temp = p1.move(10, 5);
temp.move(10, 10);
```

Где второй вызов метода move вызывался бы для временной копии и никак бы не затрагивал переменную p1.

Раздел VII. Файлы

Тема 7.1 Понятие файла. Виды файлов

Двоичный файл отличается от текстового тем, что данные в нем представлены во внутренней форме. А поскольку при внутреннем представлении используется двоичная система счисления, то «в честь ее» файлы и называются двоичными. По существу, двоичный файл является аналогом внутренней (оперативной, физической) памяти – неограниченным массивом байтов с возможностью непосредственного обращения (произвольного доступа) к любой его части.

Такая модель файла полностью совпадает с системой представлений, принятой в Си для работы с памятью на низком (физическом уровне):

– физическая память имеет байтную структуру – единицей адресации является байт;

– любая переменная занимает фиксированное количество байтов, определяемое ее типом. Операция sizeof возвращает эту размерность;

– указатель на переменную интерпретируется как ее адрес в памяти. Преобразование типа указателя к void* позволяет интерпретировать его как «чистый» адрес, а преобразование к char* - как указатель на массив байтов (физическое представление памяти).

Исходя из этих принципов, функции двоичного ввода-вывода fread и fwrite переносят содержимое памяти в двоичный файл «прозрачно», т.е. байт в байт без каких либо преобразований. Функции используются для перенесения данных из файла в память программы (чтение) и обратно (запись).

*int fread (void *buf, int size, int nrec, FILE *fd);*
*int fwrite (void *buf, int size, int nrec, FILE *fd);*

Особенностью этих функций является то, что для них безразличен (неизвестен) характер структуры данных в той области памяти, в которую осуществляется ввод-вывод (указатель void* buf).Функци fread читает, а функция fwrite пишет в файл, начиная с текущей позиции, массив из nrec элементов размерностью size байтов каждый, возвращая количество успешно прочитанных (записанных) элементов.

Чтобы воспользоваться этими функциями, необходимо обеспечить преобразования переменных к «массиву байтов», используя указатели для задания адресов и операцию sizeof для вычисления размерности:

```
// Прочитать  целую переменную и следующий за ней
// динамический массив из n переменных типа double
int n;                                  // в целой переменной – размерность массива
fread(&n, sizeof(int),1,fd);            // указатель на переменную int
double   *pd = new double[n];
fread(pd, sizeof(double),n,fd);         // преобразование к void* - неявное
```

Дальнейшее изложение приходится начинать с банальности: при использовании исключительно функций fread/fwrite данные, записанные в определенной последовательности в файл, хранятся в нем и читаются в том же самом порядке. Этот неизменный порядок извлечения данных называется последовательным доступом, а файл - последовательным двоичным файлом. Естественно, что нас при этом не интересуют адреса размещения данных в файле. Однако существует и другой способ, позволяющий извлекать данные в любом произвольном порядке – прямой (или произвольный) доступ.

Произвольный доступ базируется на понятии адреса в двоичном файле. Поскольку на физическом уровне двоичный файл представляется как «неограниченно растущий» массив

байтов, то под адресом понимается порядковый номер байта, начиная с 0.

В Си для представления адресов используются переменные типа long, на которые можно распространить известное понятие указатель - указатель в файле. Такая образная аналогия вполне уместна и позволяет с общих позиций рассматривать структуры данных, размещенные в файлах. Указатель в файле не является типизированным, его тип никак не связан с типом адресуемых данных и он рассматривается как «чистый» физический адрес в файле.

Но для начала обсудим, как произвольный доступ и система адресации поддерживается библиотекой ввода-вывода. С каждым открытым файлом связывается такой параметр как текущая позиция (текущий адрес) - номер байта, начиная с которого будет выполняться очередная операция чтения-записи. При открытии файла текущая позиция устанавливается на начало файла, после чтения-записи порции данных перемещается вперед на размерность этих данных. Для дополнения файла новыми данными необходимо установить текущую позицию на конец файла и выполнить операцию записи.

При отладке программ, работающих с двоичными файлами, иногда сложно установить, какой фрагмент – запись или чтение – содержит ошибку. Аналогично, при чтении уже известного формата необходимо проверять, насколько правильно читаются данные. Здесь не обойтись без навыков чтения дампа – двоичного содержимого файла. Для этого нам придется вспомнить основы представления базовых типов данных в памяти. Естественно, что все данные и адреса присутствуют в шестнадцатеричной системе счисления.

Тема 7.2 Функции для работы с файлами

Файлом называют способ хранения информации на физическом устройстве. Файл – это понятие, которое применимо ко всему – от файла на диске до терминала.

В C++ 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.

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.

Организация работы с файлами средствами С

Объявление файла

*FILE *идентификатор;*

Пример

*FILE *f;*

Открытие файла:

fopen(имя физического файла, режим доступа)

Режим доступа – строка, указывающая режим открытия файла файла и тип файла

Типы файла: бинарный (b); текстовый (t).

Например

```
f = fopen(s, "wb");
k = fopen("h:\ex.dat", "rb");
```

Таблица 7.2.1 – Режимы доступа к файлам

Значение	Описание
r	Файл открывается только для чтения
w	Файл открывается только для записи. Если соответствующий физический файл существует, он будет перезаписан
a	Файл открывается для записи в конец (для дозаписи) или создается, если не существует
r+	Файл открывается для чтения и записи.
w+	Файл открывается для записи и чтения. Если соответствующий физический файл существует, он будет перезаписан
a+	Файл открывается для записи в конец (для дозаписи) или создается, если не существует

Неформатированные файловый ввод-вывод

Запись в файл

fwrite(адрес записываемой величины, размер одного экземпляра, количество записываемых величин, имя логического файла);

Например,

fwrite(&dat, sizeof(int), 1, f);

Чтение из файла

fread(адрес величины, размер одного экземпляра, количество считываемых величин, имя логического файла);

Например,

fread(&dat, sizeof(int), 1, f);

Закрытие файла

fclose(имя логического файла);

Пример 1. Заполнить файл некоторым количеством целых случайных чисел.

```
/* Заполнить файл некоторым количеством целых случайных чисел. */

/* Dev-C++ */
#include <cstdlib>
#include <iostream>

using namespace std;
```

```
int main()
{
 FILE *f; int dat;
 srand(time(0));
 int n=rand()%30 + 1;
 cout << "File name? ";
 char s[20];
 cin.getline(s, 20);
 f=fopen(s, "wb");
 for (int i=1; i<=n; i++)
 { dat = rand()%101 - 50;
   cout << dat << " ";
   fwrite(&dat, sizeof(int), 1, f);
 }
 cout << endl;
 fclose(f);
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Пример 2. Найти сумму и количество целых чисел, записанных в бинарный файл.

```
/* Найти сумму и количество целых чисел, записанных в бинарный файл. */
/* Dev-C++ */
#include <cstdlib>
#include <iostream>

using namespace std;

int main()
{
 FILE *f;
 int dat, n=0, sum=0;
```

```
 cout << "File name? ";
 char s[20];
 cin.getline(s, 20);
 f=fopen(s, "rb");
 while (fread(&dat, sizeof(int), 1, f))
      {n++;
       cout << dat << " ";
       sum+=dat;
      }
 cout << endl;
 cout << "sum: " << sum << "; number: " << n << endl;
 fclose(f);
 system("PAUSE");
 return EXIT_SUCCESS;
}
```

Пример 3. Поместить в файл *n* записей, содержащих сведения о кроликах, содержащихся в хозяйстве: пол (m/f), возраст (в мес.), масса.

```
/* Поместить в файл n записей, содержащих сведения о
кроликах, содержащихся в хозяйстве:
   пол (m/f), возраст (в мес.), масса. */

/* Dev-C++ */
#include <cstdlib>
#include <iostream>

using namespace std;

struct krolik {char pol; int vozrast; double massa;};

int main()
{
 FILE *f; krolik dat; int n;
 cout << "File name? ";
```

```
    char s[20];
    cin.getline(s, 20);
    f=fopen(s, "wb");
    cout << "How many rabbits? "; cin >> n;
    for (int i=1; i<=n; i++)
    { cout << "What sex " << i << "th rabbit? "; cin >> dat.pol;
      cout << "How old " << i << "th rabbit? "; cin >>
dat.vozrast;
      cout << "What is the mass of the " << i << "th rabbit? "; cin
>> dat.massa;
      fwrite(&dat, sizeof(krolik), 1, f);
    }
    fclose(f);
      system("PAUSE");
      return EXIT_SUCCESS;
   }
```

Пример 3 (продолжение). В бинарном файле хранятся сведения о кроликах, содержащихся в хозяйстве: пол (m/f), возраст (в мес.), масса. Найти наиболее старого кролика. Если таких несколько, то вывести информацию о том из них, масса которого больше.

```
   /* В бинарном файле хранятся сведения о  кроликах,
содержащихся в хозяйстве: пол (m/f),   возраст (в мес.),
масса.
      Найти наиболее старого кролика. Если таких несколько,
то вывести информацию о том из них, масса которого
больше. */

   /* Dev-C++ */
   #include <cstdlib>
   #include <iostream>

   using namespace std;
```

```
struct krolik {char pol; int vozrast; double massa;};

int main()
{
 FILE *f; krolik dat, max; int n;
 cout << "File name? ";
 char s[20];
 cin.getline(s, 20);
 f=fopen(s, "rb");
 fread(&dat, sizeof(krolik), 1, f);
 max=dat;
 while (fread(&dat, sizeof(krolik), 1, f))
 {if (dat.vozrast>max.vozrast) max=dat;
  else if (dat.vozrast==max.vozrast&&dat.massa>max.massa)
max=dat;}
 cout << "The oldest rabbit has a sex " << max.pol << ", age "
<< max.vozrast << " and mass " << max.massa << endl;
   system("PAUSE");
   return EXIT_SUCCESS;
}
```

Форматированный файловый ввод-вывод

1) Функции fgetc() и fputc() позволяют соттветственно осуществить ввод-вывод символа.

2) Функции fgets() и fputs() позволяют соттветственно осуществить ввод-вывод строки.

3) Функции fscanf() и fprintf() позволяют соттветственно осуществить форматированный ввод-вывод и аналогичный соответствующим функиям форматированного ввода-вывода, только делают это применительно к файлу.

Организация работы с файлами средствами C++

Файловый ввод-вывод с использованием потоков

Библиотека потокового ввода-вывода

fstream

Связь файла с потоком вывода

ofstream имя логического файла;

Связь файла с потоком ввода

ifstream имя логического файла;

Открытие файла

имя логического файла.open(имя физического файла);

Закрытие файла

имя логического файла.close();

Пример 4. Заполнить файл значениями функции y = x * cos x.

```
/* Заполнить файл значениями функции y = x * cos x. */

/* Dev-C++ */
#include <cstdlib>
#include <iostream>
#include <fstream>
#include <cmath>

using namespace std;

double fun(double x);

int main()
{double a, b, h, x; char s[20];
 cout << "Enter the beginning and end of the segment, step-
tabulation: ";
 cin >> a >> b >> h;
 cout << "File name? "; cin >> s;
 ofstream f;
 f.open(s);
 for (x=a; x<=b; x+=h)
  {f.width(10);   f << x;
   f.width(15);   f << fun(x) << endl;  }
 f.close();
```

```
    system("PAUSE");
    return EXIT_SUCCESS;
}
double fun(double x)
{ return x*cos(x); }
```

Пример 5. Файл содержит несколько строк, в каждой из которых записано единственное выражение вида a#b (без ошибок), где a, b - целочисленные величины, # - операция +, -, /, *. Вывести каждое из выражений и их значения.

```
/* Dev-C++ */
#include <cstdlib>
#include <iostream>
#include <fstream>

using namespace std;
int main()
{
long a, b;  char s[256], c;  int i;
  cout << "File name? "; cin >> s;
  ifstream f;  f.open(s);
  while (!f.eof())
   {    f.getline(s, 256);
    i=0; a=0;
    while (s[i]>='0'&&s[i]<='9')
     {
         a=a*10+s[i]-'0';
          i++;
     }
    c=s[i++];    b=0;
    while (s[i]>='0'&&s[i]<='9')
     {
         b=b*10+s[i]-'0';
          i++;
     }
```

```
    switch (c){
    case '+': a+=b; break;
    case '-': a-=b; break;
    case '/': a/=b; break;
    case '*': a*=b; break;}
    cout << s << " = " << a << endl;   }
    f.close();
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Пример 6. В заданном файле целых чисел посчитать количество компонент, кратных 3.

```
/* В заданном файле целых чисел посчитать количество
компонент, кратных 3. */
/* Dev-C++ */
#include <cstdlib>
#include <iostream>
#include <fstream>

using namespace std;
int main()
{int r,ch;
  ifstream f;
  f.open("CH_Z.TXT");
  ch=0;
  for (;f.peek()!=EOF;)
    {f>>r;
     cout << r << " ";
     if (r%3==0) ch++ ;
     }
  f.close();
  cout << endl << "Answer: " << ch;
    system("PAUSE");
    return EXIT_SUCCESS;}
```

Раздел 8. Алгоритмы сортировки и поиска

Тема 8.1 Алгоритмы сортировки и их классификация

Сортировка массива в C++ встречается довольно часто. Существует много функций, позволяющих массив отсортировать, то есть расположить все элементы массива по возрастанию или убыванию. Остановимся на встроенных в C++ функциях qsort и std::sort. Важно заметить, что можно реализовать и использовать и другие самые разнообразные функции для сортировки массива.

Первая из интересующих нас функций qsort библиотеки stdlib.h:

qsort (<сортируемый массив>, <количество элементов>, sizeof(<тип элемента>), <функция для сравнения>);

qsort (<сортируемый массив>, <количество элементов>, sizeof(<тип элемента>), <функция для сравнения>);

Функция для сравнения - наиболее своеобразная часть кода. Она выглядит примерно таким образом:

```
//функция для сортировки по возрастанию
int comp1 (const void * a, const void * b)
{
  return ( *(int*)a - *(int*)b );
}
//функция для сортировки по возрастанию
int comp1 (const void * a, const void * b)
{
  return ( *(int*)a - *(int*)b );
}
```

Тип данных int для массива другого типа можно заменить на другой.

Тоже самое можно проделать и с помощью другой функции std::sort, которая находится в библиотеке algorithm из stl:

sort (<итератор начала сортировки>, <итератор конца сортировки>, <функция для сравнения>);

sort (<итератор начала сортировки>, <итератор конца сортировки>, <функция для сравнения>);

можно и без последнего параметра, тогда сравнение будет по возрастанию:

sort (<итератор начала сортировки>, <итератор конца сортировки>);

sort (<итератор начала сортировки>, <итератор конца сортировки>);

Для использования этой функции массив нужно предварительно преобразовать в vector.

Затем уместно использовать итераторы начала и конца полученного вектора

sort_int_vector.begin(), sort_int_vector.end()

sort_int_vector.begin(), sort_int_vector.end()

для указания начала и конца сортируемой части массива.

```
//пример использования функции qsort
#include <iostream>
#include <cstdlib>

int vector[] = { 14, 10, 11, 19, 2, 25 };

int compare(const void * x1, const void * x2)   // функция сравнения элементов массива
{
  return ( *(int*)x1 - *(int*)x2 );              // если результат вычитания равен 0, то числа равны, < 0: x1 < x2; > 0: x1 > x2
}
int main ()
{
  qsort(vector, 6, sizeof(int), compare);      // сортируем массив чисел
  for ( int ix = 0; ix < 6; ix++)
     std::cout << vector[ix] << " ";
  return 0; }
```

Тема 8.2 Алгоритмы поиска и их классификация

В классе ios_base определен ти seekdir:

Typedef T4 seekdir;

Тип является синонимом вложенного типа T4, определяющего перечисление и используемого для определения направления поиска позиции в потоке. В классе ios_base определены следующие флаги, которые управляют направление поиска позиции:

Static const seekdir beg, cur, end;

Эти флаги имеют следующее значение: beg – поиск позиции, начиная от начала потока; cur – поиск позиции, начиная от текущей позиции; end – поиск позиции, начиная от конца потока.

Раздел 9. Динамические структуры данных
Тема 9.1 Понятия и классификация динамических структур

Структура в языке C++ представляет собой производный тип данных, который представляет какую-то определенную сущность, также как и класс. Нередко структуры применительно к C++ также называют классами. И в реальности различия между ними не такие большие.

Для определения структуры применяется ключевое слово struct, а сам формат определения выглядит следующим образом:

```
struct имя_структуры
{
   компоненты_структуры
};
```

Имя_структуры представляет произвольный идентификатор, к которому применяются те же правила, что и при наименовании переменных.

После имени структуры в фигурных скобках помещаются *Компоненты_структуры*, которые представляют набор описаний объектов и функций, которые составляют структуру.

Например, определим простейшую структуру:

```
#include <iostream>
#include <string>

struct person
{
   int age;
   std::string name;
};

int main()
{
```

```
    person tom;
    tom.name = "Tom";
    tom.age = 34;
    std::cout << "Name: " << tom.name << "\tAge: " <<
tom.age << std::endl;
    return 0;
}
```

Здесь определена структура person, которая имеет два элемента: age (представляет тип int) и name (представляет тип string).

После определения структуры мы можем ее использовать. Для начала мы можем определить объект структуры – по сути обычную переменную, которая будет представлять выше созданный тип. Также после создания переменной структуры можно обращаться к ее элементам – получать их значения или, наоборот, присваивать им новые значения. Для обращения к элементам структуры используется операция «точка»:

имя_переменной_структуры.имя_элемента

По сути структура похожа на класс, то есть с помощью структур также можно определять сущности для использования в программе. В то же время все члены структуры, для которых не используется спецификатор доступа (public, private), по умолчанию являются открытыми (public). Тогда как в классе все его члены, для которых не указан спецификатор доступа, являются закрытыми (private).

Кроме того мы можем инициализировать структуру, присвоив ее переменным значения с помощью синтаксиса инициализации:

```
person tom = { 34, "Tom" };
```

Инициализация структур аналогична инициализации массивов: в фигурных скобках передаются значения для элементов структуры по порядку. Так как в структуре person первым определено свойство, которое представляет тип int -

число, то в фигурных скобках вначале идет число. И так далее для всех элементов структуры по порядку.

Когда использовать структуры? Как правило, структуры используются для описания таких данных, которые имеют только набор публичных атрибутов – открытых переменных. Например, как та же структура person, которая была определена в начале статьи. Иногда подобные сущности еще называют аггрегатными классами (aggregate classes).

Часто в серьезных программах надо использовать данные, размер и структура которых должны меняться в процессе работы. Динамические массивы здесь не выручают, поскольку заранее нельзя сказать, сколько памяти надо выделить – это выясняется только в процессе работы. Например, надо проанализировать текст и определить, какие слова и в каком количество в нем встречаются, причем эти слова нужно расставить по алфавиту.

В таких случаях применяют данные особой структуры, которые представляют собой отдельные элементы, связанные с помощью ссылок.

Каждый элемент (узел) состоит из двух областей памяти: поля данных и ссылок. Ссылки – это адреса других узлов этого же типа, с которыми данный элемент логически связан. В языке Си для организации ссылок используются переменныеуказатели. При добавлении нового узла в такую структуру выделяется новый блок памяти и (с помощью ссылок) устанавливаются связи этого элемента с уже существующими. Для обозначения конечного элемента в цепи используются нулевые ссылки (NULL).

Тема 9.2 Списки

Массив – не единственный тип, способный играть роль контейнера для данных. Другим таким типом является связанный список. В отличие от массива, он реализуется не языком, а программистом.

Пример. Определить связанный список и операции над ним.

```
// Структура – элемент списка

struct Item {
int info;
Item* next;
};
void main ( )
{
Item *first = 0; //Указатель на начало списка
Item *p;
int i;
// Создать список
for (;;) {
// Вводить числа, пока не введем 0
cin » i;
if (!i) break;
// Создать новый элемент списка
p = new Item;
p->info = i;
// Присоединить новый элемент к началу списка
p->next = first ;
first = p;
}
// Пройти список и вывести элементы
p = first;
while (p) {
cout « p->info « » » ;
p = p->next;
}
// Пройти список и удалить элементы
while (first) {
p = first;
```

```
first = first->next ;
delete p;
}
}
```

Другие виды списков:

– список с указателями на первый и последний элементы – позволяет добавлять и удалять элементы с обеих сторон списка;

– двунаправленный список – позволяет перемещаться по списку в обоих направлениях;

– кольцевой список – позволяет достичь любой элемент списка,

начав движение с любого места в списке.

Тема 9.3 Очереди

Очередь – это структура данных, которая построена по принципу LILO (last in – last out: последним пришел – последним вышел). В C++ уже есть готовый STL контейнер – queue.

В очереди, если вы добавите элемент, который вошел самый первый, то он выйдет тоже самым первым. Получается, если вы добавите 4 элемента, то первый добавленный элемент выйдет первым.

Чтобы понять принцип работы очереди вы можете представить себе магазинную очередь. И вы стоите посреди нее, чтобы вы оказались напротив кассы, сначала понадобится всех впереди стоящих людей обслужить. А вот для последнего человека в очереди нужно, чтобы кассир обслужил всех людей кроме него самого.

Рисунок 9.3.1 – Пример очереди

На рисунке 9.3.1 находятся 7 чисел: 2, 4, 7, 1, 4, 9, 10. Если нам понадобится их извлечь, то извлекать мы будем в таком же порядке как они находятся на рисунке!

Хотя в стеке присутствует функция peek() (она позволяет обратится к элементу по индексу), в шаблоне очереди невозможно обратится к определенному элементу.

Если вы хотите использовать шаблон очереди в C++, то вам сначала нужно подключить библиотеку – <queue>.

Дальше для объявления очереди нужно воспользоваться конструкцией ниже.

queue <тип данных> <имя>;

- Сначала нам нужно написать слова queue.
- Дальше в <тип данных> мы должны указать тот тип, которым будем заполнять нашу очередь.
- И в конце нам остается только указать название очереди.

Пример правильного объявления:

queue <int> q;

Метод – это та же самая функция, но она работает только с контейнерами STL. Для работы с очередью вам понадобится знать функции: push(), pop(), front(), back(), empty().

1. Для добавления в очередь нового элемента нужно воспользоваться функцией – push(). В круглых скобках должно находится значение, которое мы хотим добавить.
2. Если нам понадобилось удалить первый элемент нужно оперировать функцией pop(). В круглых скобках уже не чего не нужно указывать, но по правилам они в обязательном порядке должны присутствовать! Эти функции тоже не нуждаются в указании аргумента: empty(), back() и front().
3. Если вам понадобилось обратиться к первому элементу очереди, то вам понадобится функция front().
4. Чтобы обратиться к последнему элементу в очереди вам поможет функция back().

5. Чтобы узнать пуста ли очередь нужно воспользоваться функцией empty().

- Если ваша очередь пуста – возвратит true.
- Если же в ней что-то есть – возвратит false.

Ниже мы использовали все выше перечисленные методы:

```
#include <iostream>
#include <queue>  // подключили библиотеку queue

using namespace std;

int main() {
  setlocale(LC_ALL,"rus");
  queue <int> q;  // создали очередь q

  cout << "Пользователь, пожалуйста введите 7 чисел: "
<< endl;

  for (int h = 0; h < 7; h++) {
    int a;

    cin >> a;

    q.push(a);  // добавляем в очередь элементы
  }

  cout << endl;
  cout << "Самый первый элемент в очереди: " << q.front()
<< endl;  // выводим первый
                                          // элемент очереди
  q.pop();  // удаляем элемент из очереди

  cout << "Новый первый элемент (после удаления): " <<
q.front() << endl;
```

```
if (!q.empty()) cout << "Очередь не пуста!";  // проверяем пуста ли очередь (нет)

system("pause");
return 0; }
```

Очередь можно реализовать через массив. Обычно, если кто-то создает такую очередь, то массив называют queue.

Для реализации нам понадобится создать две дополнительные переменные start и ends. start будет указывать на первый элемент очереди, а ends на последний элемент.

Чтобы обратится к последнему элементу нам придется использовать эту конструкцию – queue[ends]. Обращение к первому элементу будет выглядеть аналогично queue[start].

Если понадобится удалить элемент из очереди, то придется всего лишь уменьшить переменную start на один.

«А как же проверить пуста ли очередь?» – спросите вы. Для этого мы просто проверим условие start == ends:

1. Если результатом условия будет true, то очередь пуста.
2. Если же результат условия false, значит очередь чем-то заполнена.

Ниже находится пример создания такой очереди:

```
setlocale(LC_ALL,"rus");
int q[7];  // создали массив q
int start = 0, ends = 0;  // создали переменные начала и конца очереди

cout << "Пользователь, пожалуйста введите 7 чисел: " << endl;

for (int h = 0; h < 7; h++) { int a; cin >> a;
 int a;

 cin >> a;
  q[start++] = a;  // добавляем элементы в
```

```
очередь(массив)
    }
    cout << "Самый первый элемент в очереди: " << q[start - 1] << endl;
    start--;  // удаляем первый элемент(уменьшаем start на 1)
    cout << "Новый первый элемент (после удаления): " << q[start - 1] << endl;
    cout << "Самый последний элемент в очереди: " << q[ends];  // выводим последний элемент очереди
    if (start != ends) cout << "Очередь не пуста!";  // проверяем пуста ли очередь
```

Очередь с приоритетом (priority_queue) – это обычная очередь, но в ней новый элемент добавляется в то место, чтобы очередь была отсортирована по убыванию.

Так самый большой элемент в приоритетной очереди будет стоять на первом месте.

Для объявления шаблона очереди с приоритетом нужно использовать конструкцию ниже:

priority_queue <тип данных> <имя>;

- В начале нужно написать priority_queue.
- Потом в скобках указать тип данных, который будет находится в очереди.
- И конечно же в самом конце мы должны дать ей имя.

Для добавления элемента в очередь с приоритетом мы должны использовать функцию push(). Но чтобы обратится к первому элементу должны использоваться именно функция – top() (как и в стеке). А не функция – front().

Также нельзя использовать функцию back() для обращения к последнему элементу. Для приоритетной очереди она также не работает, как функция front().

Вот пример использования очереди с приоритетом в программе:

```
    setlocale(LC_ALL,"rus");

    priority_queue <int> priority_q;  // объявляем
приоритетную очередь

    cout << "Введите 7 чисел: " << endl;

    for (int j = 0; j < 7; j++) { int a; cin >> a;

      priority_q.push(a);  // добавляем элементы в очередь
    }
  // выводим первый
    cout << "Первый элемент очереди: " << priority_q.top();
// элемент
```

Тема 9.4 Стеки

Загрузка в стек, получение из стека. Порядок следования значений.

Стек – это структура данных, которая работает по принципу FILO (first in – last out; первый пришел – последний ушел). В C++ уже есть готовый шаблон – stack.

В стеке элемент, который вошел самый первый – выйдет самым последним. Получается, если вы добавили три элемента в стек первым будет удален последний добавленный элемент.

На рисунке 9.4.1 вы можете увидеть 6 чисел: 6, 5, 1, 2, 5, 9. Извлекать их будем в таком же порядке. В стеке нет индексов как в массиве, а значит, вы не можете обратиться к определенному элементу. Все потому что, стек построен на связных списках.

Это значит что каждый элемент (кроме последнего – он показывает на NULL) имеет указатель на следующий элемент. Но есть элемент, на который нет указателя – первый (или как его еще называют *головной*).

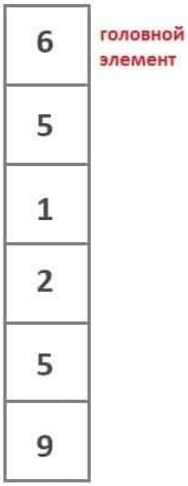


Рисунок 9.4.1 – Пример стека

Но все достоинство шаблонного стека заключается в добавлении и удалении элементов. Эти операции происходят за константное время (это хороший плюс).

Для использования шаблона стека в начале нашей программе мы должны подключить библиотеку – <stack>.

Чтобы создать стек нам понадобится оперировать схемой ниже:

stack <тип данных> <имя>;

Давайте тщательнее ее разберем:

– С новой строки мы должны написать слова stack.

– <тип данных> – здесь нам понадобиться написать тот тип данных, который будет храниться в стеке.

– <имя> – здесь вам все должно быть понятно.

Методы – это функции, которые используются для контейнеров типа очереди и стека:

```
#include <iostream>
#include <stack>  // подключаем библиотеку для
                  // использования стека
using namespace std;
```

```
int main() {
  setlocale(LC_ALL,"rus");
  stack <int> steck;  // создаем стек

  int i = 0;

  cout << "Введите шесть любых целых чисел: " << endl; // предлагаем пользователю
                                                       // ввести 6 чисел
  while (i != 10) {
    int a;
    cin >> a;

    steck.push(a);  // добавляем введенные числа
    i++;
  }

  if (steck.empty()) cout << "Стек не пуст";  // проверяем пуст ли стек (нет)

  cout << "Верхний элемент стека: " << steck.top() << endl; // выводим верхний элемент
  cout << "Давайте удалим верхний элемент " << endl;

  steck.pop();  // удаляем верхний элемент

  cout << "А это новый верхний элемент: " << steck.top(); // выводим уже новый
                                                          // верхний элемент
  system("pause");
  return 0; }
```

В строке 18: мы добавляем в стек элемент, с помощью функции push(). В скобках должно находиться значение, которое мы хотим добавить.

В строке 22: чтобы проверить пуст ли стек мы воспользовались функцией empty().

– Если результатом этой функции будет true, то стек чист.

– Если же результатом будет false, то в стеке что-то есть.

В строке 27: была использована функция pop().Ее используют для удаления *верхнего* элемента стека.

В функции pop() в отличии от функции push() в скобках не нужно не чего указывать, но сами скобки обязательно должны присутствовать. Так же для функций: empty() и top()!

В строках 24 и 29: мы решили обратиться к верхнему элементу стека, для этого была использована функция – top().

В библиотеку stack добавили новую функцию peek(), с помощью которой вы можете обратиться к N элементу стека (от вершины).

Так с помощью этой функции стек начинает напоминать массив.

Ниже вы мы использовали функцию peek(), чтобы вывести третий элемент:

```
setlocale(LC_ALL,"rus");
stack <int> steck;

steck.push(5);   // Добавляем
steck.push(1);   // элементы
steck.push(9);   // в
steck.push(10);  // стек
cout << "Третий элемент стека: " << steck.peek(3); // выведет 1
```

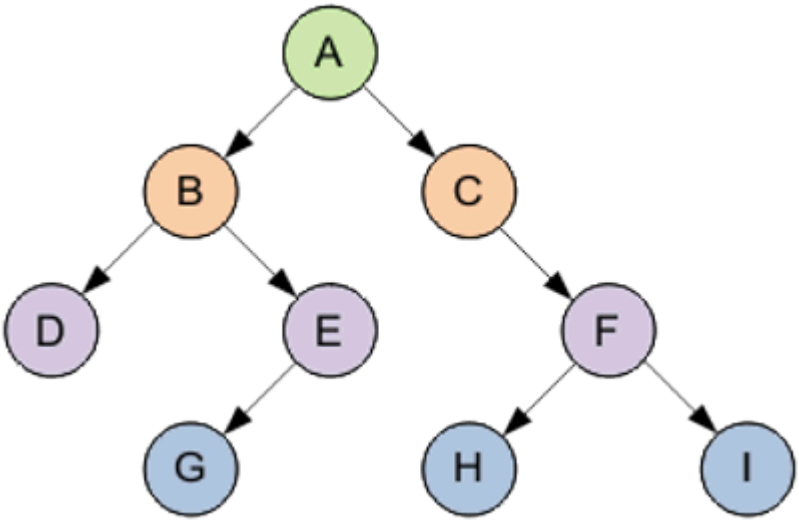
Этой функцией мы можем воспользоваться только в версиях C++ 11 и выше.

Тема 9.5 Деревья

Дерево – структура данных, представляющая собой древовидную структуру в виде набора связанных узлов.

Бинарное дерево – это конечное множество элементов, которое либо пусто, либо содержит элемент (корень), связанный с двумя различными бинарными деревьями, называемыми левым и правым поддеревьями. Каждый элемент бинарного дерева называется узлом. Связи между узлами дерева называются его ветвями.

Способ представления бинарного дерева изображен на рисунке 9.5.1.



A – корень дерева; B – корень левого поддерева;
C – корень правого поддерева.

Рисунок 9.5.1 – Бинарное дерево

Корень дерева расположен на уровне с минимальным значением.

Узел D, который находится непосредственно под узлом B, называется потомком B. Если D находится на уровне i, то B – на уровне i-1. Узел B называется предком D.

Максимальный уровень какого-либо элемента дерева называется его глубиной или высотой.

Если элемент не имеет потомков, он называется листом или терминальным узлом дерева.

Остальные элементы – внутренние узлы (узлы ветвления).

Число потомков внутреннего узла называется его степенью. Максимальная степень всех узлов есть степень дерева.

Число ветвей, которое нужно пройти от корня к узлу x, называется длиной пути к x. Корень имеет длину пути равную 0; узел на уровне i имеет длину пути равную i.

Бинарное дерево применяется в тех случаях, когда в каждой точке вычислительного процесса должно быть принято одно из двух возможных решений.

Имеется много задач, которые можно выполнять на дереве.

Распространенная задача – выполнение заданной операции p с каждым элементом дерева. Здесь рассматривается как параметр более общей задачи посещения всех узлов или задачи обхода дерева.

Если рассматривать задачу как единый последовательный процесс, то отдельные узлы посещаются в определенном порядке и могут считаться расположенными линейно.

Способы обхода дерева

Пусть имеем дерево, где A – корень, B и C – левое и правое поддеревья.

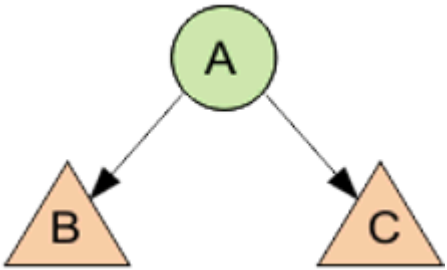


Рисунок 9.5.2 – Обход дерева

Существует три способа обхода дерева:

– Обход дерева сверху вниз (в прямом порядке): A, B, C – префиксная форма.

– Обход дерева в симметричном порядке (слева направо): B, A, C – инфиксная форма.

– Обход дерева в обратном порядке (снизу вверх): B, C, A – постфиксная форма.

Реализация дерева

Узел дерева можно описать как структуру:

```
struct tnode {
  int field;          // поле данных
  struct tnode *left;  // левый потомок
  struct tnode *right; // правый потомок
};
```

При этом обход дерева в префиксной форме будет иметь вид

```
void treeprint(tnode *tree) {
  if (tree!=NULL) { //Пока не встретится пустой узел
    cout << tree->field; //Отображаем корень дерева
    treeprint(tree->left); //Рекурсивная функция для левого поддерева
    treeprint(tree->right); //Рекурсивная функция для правого поддерева
  }
}
```

Обход дерева в инфиксной форме будет иметь вид

```
void treeprint(tnode *tree) {
  if (tree!=NULL) { //Пока не встретится пустой узел
    treeprint(tree->left); //Рекурсивная функция для левого поддерева
    cout << tree->field; //Отображаем корень дерева
    treeprint(tree->right); //Рекурсивная функция для правого поддерева
  }
}
```

Обход дерева в постфиксной форме будет иметь вид

```
void treeprint(tnode *tree) {
  if (tree!=NULL) { //Пока не встретится пустой узел
```

```
    treeprint(tree->left); //Рекурсивная функция для левого поддерева
    treeprint(tree->right); //Рекурсивная функция для правого поддерева
    cout << tree->field; //Отображаем корень дерева
   }
  }
```

Бинарное (двоичное) дерево поиска – это бинарное дерево, для которого выполняются следующие дополнительные условия (свойства дерева поиска):

- оба поддерева – левое и правое, являются двоичными деревьями поиска;
- у всех узлов левого поддерева произвольного узла X значения ключей данных меньше, чем значение ключа данных самого узла X;
- у всех узлов правого поддерева произвольного узла X значения ключей данных не меньше, чем значение ключа данных узла X.

Данные в каждом узле должны обладать ключами, на которых определена операция сравнения меньше.

Как правило, информация, представляющая каждый узел, является записью, а не единственным полем данных.

Для составления бинарного дерева поиска рассмотрим функцию добавления узла в дерево.

Добавление узлов в дерево

```
struct tnode * addnode(int x, tnode *tree) {
  if (tree == NULL) { // Если дерева нет, то формируем корень
    tree =new tnode; // память под узел
    tree->field = x;  // поле данных
    tree->left =  NULL;
    tree->right = NULL; // ветви инициализируем пустотой
  }else  if (x < tree->field)   // условие добавление левого потомка
```

```
    tree->left = addnode(x,tree->left);
  else    // условие добавление правого потомка
    tree->right = addnode(x,tree->right);
  return(tree);
}
Удаление поддерева
void freemem(tnode *tree) {
  if(tree!=NULL) {
    freemem(tree->left);
    freemem(tree->right);
    delete tree;
  }
}
```

Раздел 11. Модульное программирование

Тема 11.1 Понятие модуля, его составные части

Метод модульного программирования предполагает разделение текста программы на несколько файлов, в каждом из которых сосредоточены независимые части программы (сгруппированные по смыслу функции).

В программах на Си++ часто применяются библиотечные функции (например, "sqrt(...)"). Для использования большинства функций, в том числе и библиотечных, необходимы два файла:

– *Заголовочный файл* ("math.h") с прототипом функции ("sqrt(...)" и многих других математических функций). Поэтому в программах, вызывающих "sqrt(...)", есть строка "#include <math.h>", а не явное объявление этой функции.

– *Файл реализации* (для пользовательских функций это файлы с исходным текстом на Си++, а библиотечные функции обычно хранятся в скомпилированном виде в специальных библиотечных файлах, например, "libcmtd.lib"). Файлы реализации пользовательских функций (обычно с расширением ".cpp") содержат определения этих функций.

Обратите внимание, что имя файла стандартной библиотеки "iostream.h" в директиве препроцессора "include" заключено в угловые скобки ("<>"). Файлы с именами в угловых скобках препроцессор ищет в библиотечных каталогах, указанных в настройках компилятора. Имена пользовательских заголовочных файлов обычно заключаются в двойные кавычки, и препроцессор ищет их в текущем каталоге программы.

Далее приведено содержимое файла "averages.h". В нем есть идентификатор препроцессора "AVERAGES_H" и служебные слова препроцессора "ifndef" ("если не определено"), "define" ("определить") и "endif" ("конец директивы if"). Идентификатор "AVERAGES_H" является глобальным символическим именем заголовочного файла.

Первые две строки файла служат защитой от повторной обработки текста заголовочного файла препроцессором, на случай, если в исходном тексте программы строка "#include "averages.h"" встречается несколько раз.

В заголовочных файлах, кроме прототипов функций, часто размещаются описания глобальных констант и *пользовательских типов.*

```
#ifndef AVERAGES_H
#define AVERAGES_H
// (Определения констант и пользовательских типов)

// ПРОТОТИП ФУНКЦИИ ДЛЯ ВЫЧИСЛЕНИЯ ЦЕЛОЧИСЛЕННОГО СРЕДНЕГО
// ЗНАЧЕНИЯ 3-Х ЦЕЛЫХ ЧИСЕЛ:
int average( int first_number, int second_number, int third_number );
// ПРОТОТИП ФУНКЦИИ ДЛЯ ВЫЧИСЛЕНИЯ ЦЕЛОЧИСЛЕННОГО СРЕДНЕГО
// ЗНАЧЕНИЯ 2-Х ЦЕЛЫХ ЧИСЕЛ:
int average( int first_number, int second_number );

#endif
```

Тема 11.2 Модульная схема программы

Проект представляет собой совокупность файлов, которые компилятор использует для создания выполняемого файла.

Основными элементам проекта являются: главный модуль приложения (cpp-файл); модуль формы (h-файл).

В главном модуле находится функция main, с которой начинается выполнение программы. Функция main обеспечивает запуск приложения – создает главную форму.

Процесс преобразования исходной программы в выполняемую называется компиляцией или построением. Укрупненно процесс построения программы можно

представить как последовательность двух этапов: компиляция и компоновка. На этапе компиляции выполняется перевод исходной программы (модулей) в некоторое представление. На этапе компоновки выполняется объединение модулей в единую программу.

Процесс построения программы активизируется в результате выбора в меня Build команды Build project (где project – имя проекта), а также в результате запуска программы из среды разработки, если с момента последней компиляции в программу были внесены изменения.

Процесс и результат компиляции отражаются в оке Output. Если в программе нет ошибок, то по завершения процесса компиляции в оке Output отображается сообщение *Build succeeded*.

Компилятор генерирует выполняемую программу только в случае, если в исходной программе нет синтаксических ошибок. Если в программе есть ошибки, то программист должен их устранить. Процесс устранения ошибок носит итерационный характер. Обычно сначала устраняются наиболее очевидные ошибки. После очередного внесения изменений в текст программы выполняется повторная компиляция. Следует обратить внимание, что компилятор не всегда может точно локализовать ошибку. Поэтому, анализируя фрагмент кода, который помечен как содержащий ошибку, надо обращать внимание и на текст, который находится в предыдущих строках.

После того как приложение будет отлажен, можно выполнить его сборку в режиме Release – создать выполняемый файл, предназначенный для передачи пользователям программы. Режим создания сборки задается путем выбора в списке Solution Configuration.

Введение в практический раздел

Лабораторные работы посвящены изучению синтаксических и семантических конструкций языка программирования высокого уровня С++.

В пособии рассматриваются принципы программирования разветвляющихся и циклических вычислительных процессов, обработки массивов и файлов, использования структурированных типов данных, подпрограмм с особенностями передачи в них параметров.

Приводятся различные приемы программирования, в том числе использование указателей и ссылок, динамически распределяемой памяти.

В каждой лабораторной работе приводится пример выполнения типового задания с учетом предъявляемых требований. Для получения глубоких знаний по теме необходимо внимательно ознакомиться с порядком выполнения работы и выполнить все указанные требования – от изучения теоретического материала до требований, предъявляемых к алгоритму решения задачи.

Основная цель лабораторных работ – сформировать у студентов фундаментальные основы знаний, необходимые при проектировании программ для вычислительных систем, привить навыки системного подхода к решению поставленной задачи, на практике познакомить с этапами решения практических задач на ЭВМ, научить оформлять сопроводительную документацию.

Содержание отчетов п олабораторным работам

Каждая лабораторная работа включает в себя оформление соответствующей документации. В документации обязательно должны быть представлены следующие пункты:

1) текст индивидуального задания по варианту;

2) описания всех разработанных процедур и/или функций;

3) листинг программы решения задачи на языке высокого уровня С++;

4) ответы на контрольные вопросы;

5) выводы по работе.

Все представленные пункты должны быть отражены в отчете по каждой лабораторной работе.

Подготовка к лабораторным работам

Для выполнения лабораторных работ потребуется среда разработки, поддерживающая язык C++. В данном пособии будем ориентироваться на Microsoft Visual Studio, версии не ниже 8.0.

Все лабораторные работы данного цикла ориентированы на создание консольных приложений.

Для создания проекта в среде Microsoft Visual Studio выполните следующее:

1. Запустите Microsoft Visual Studio.

2. Выберите в меню создание нового проекта: File-> New -> Project...

3. В открывшемся окне:

а) выберите Visual C++;

б) выберите *Пустой проект*;

в) в строке ввода Name введите имя проекта, в строке Location укажите место расположения будущего проекта на диске (рабочая директория проекта), в строке Solution name будет то же самое имя проекта, что и в Name.

4. Нажмите кнопку ОК.

В результате открывается совокупность окон редактора текста, дерева файлов проекта, отладчика и других, в которых располагается шаблон приложения.

В редакторе видим текст созданного файла с именем, как у проекта, и расширением .cpp. Для компиляции приложения выберите в меню *Build->Build Solution* (или нажмите F7).

Если компиляция выполнилась без ошибок, то в рабочей директории проекта будет создан исполняемый файл с именем проекта и расширением .exe. Если компилятор обнаружил

ошибки, то их список будет располагаться в окне Output, там же появляется сообщение об успешной («Build succeeded»), либо неуспешной («Build FAILED») компиляции. Для отладки программы можно выполнять ее по шагам, для этого выберите в меню *Debug-> Step Over* (или Step Into) или нажмите F10 (F11).

ЛАБОРАТОРНАЯ РАБОТА № 1.
Основы программирования на языке С++

Цель работы: получить базовые умения работы в среде MS Visual Studio, написания простых консольных приложений.

Порядок выполнения работы: изучить теоретическую часть, выполнить практические задания, оформить отчет, осуществить защиту лабораторной работы.

Теоретическая часть

Структура программы на С++ состоящих из следующих элементов:

#include <заголовочный файл>
int main(){
операторы;
return 0;}

где, *#include <заголовочный файл>* – это директива препроцессора, которая представляет собой инструкцию, записанную в тексте программы на С++ и выполняемую до трансляции программы.

Директива *#include* включает в текст программы содержимое указанного файла.

Имеет две формы:

#include "имя_файла"
#include <имя_файла>

Если имя файла задано в угловых кавычках, то поиск файла производится в стандартных каталогах ОС, задаваемых командой PATH. Если же имя указано в кавычках, то поиск файла осуществляется в соответствии с заданным маршрутом, а при его отсутствии – в текущем каталоге.

int main(){
return 0;}

Это означает, что перед нами функция. Вся совокупность приведенных строк называется определением функции. Это определение состоит из двух частей: первой строки *int main (),* которая называется заголовком функции и остальной части,

заключенной в фигурные скобки, которая является телом функции.

Заголовок функции определяет интерфейс между функцией и остальной частью программы, а тело функции содержит инструкции для компьютера, т.е. определяет то, что собственно делает функция.

Заключительный оператор *return 0; (оператор возврата)*.

Синтаксис языка С++ требует, что определение функции main начиналось с int main(). Информация в круглых скобках называется списком аргументов или списком параметров.

Алфавит языка С++

В С++ используются следующие символы:

Знаки препинания: . , : ;

Скобки: () {} []

Знаки: | ^ ? _

Двойные и одинарные кавычки: “ ‘

Комментарии

Переменные

Каждая переменная имеет имя, тип, размер и значение.

Имя переменной в прямом смысле является ее названием. Имя переменной, или идентификатор, может состоять из латинских букв, цифр и символа подчеркивания.

Тип определяет, какие символы или числа записаны в ячейку памяти под этим именем.

Размер указывает максимальную величину или точность задания числа.

Описание переменных имеет вид:

имя_типа список_переменных

Примеры описаний:

char symbol, cc;

int number, row;

Типы данных

Данные в языке С++ разделяются на две категории: простые (скалярные), будем их называть базовыми, и сложные (составные) типы данных.

Основные типы базовых данных: целый *int* (integer), вещественный с одинарной точностью *float* и символьный *-char* (character).

В свою очередь, данные целого типа могут быть короткими *short*, длинными *long* и беззнаковыми *unsigned*, а вещественные с удвоенной точностью *double*.

Арифметические операции. К ним относятся:

– вычитание или унарный минус;

+ сложение или унарный плюс;

* умножение;

/ деление;

% деление по модулю;

++ унарная операция увеличения на единицу (инкремент);

-- унарная операция уменьшения на единицу (декремент).

Таблица 1. – Типы данных языка С++

Тип данных	Объем памяти (байт)	Диапазон значений
char	1	-128 …127
int	2	-32768…32767
short	2(1)	-32768…32767(-128…127)
long	4	-2147483648…2147483647
unsigned int	4	0…65535
unsigned long	4	0…4294967295
float	4	3,14*10-38…3,14*1038
double	8	1,7 *10-308 1,7 *10308

Пример 1. Программа расчета среднего арифметического трех введенных значений.

```
#include <iostream>
using namespace std;
int main ()
{
double a, b, c, sr;
cout<< “Input a, b, c”;
```

```
cin>> a>>b>>c;
sr=(a+b+c)/3;
cout<<sr;
return 0;
}
```

Программы линейной структуры представляют собой одну из разновидностей базовых конструкций структурного программирования, так называемое следование.

Следованием называется конструкция, представляющая собой последовательное выполнение двух или более операторов (простых или составных).

Целью использования базовых конструкция является получение программы простой структуры. Такую программу легко читать, отлаживать и при необходимости вносить в нее изменения.

Пример 2.

$$\sin\sqrt{x+1} - y^3$$

```
#include <iostream>
#include <math.h>
using namespace std;
int main ()
{
double x, y, F
cout<<"x=";
cin>>x;
cout<<"y=";
F=sin(sqrt(x+1++-pow(y,3);
cout<<"F="<<F<<endl;
}
```

Для вычисления корней, степени, синуса, косинуса и т.д. в C++ используется библиотека math.h.

Функция вычисления степени

pow(число_возводимое_в_степень, степень)

Функция вычисления корня

sqrt(выражение)

Функция вычисления синуса

sin(число)

Условные операторы

Ветвление в простейшем случае описывается в языке С++ с помощью условного оператора, имеющего вид:

if (выражение) оператор_1;

else оператор_2;

где часть *else оператор_2* может и отсутствовать.

Сначала вычисляется «выражение» в скобках; если оно истинно то выполняется оператор_1. Если «выражение» ложно (равно нулю – NULL), то оператор_1 пропускается, а выполняется оператор_2. Если на месте условно выполняемых операторов должна располагаться группа из нескольких операторов языка, то они заключаются в фигурные скобки – { }.

Часто «выражение» в скобках представляет условие, заданное с помощью операций отношений и логических операций.

Операции отношения обозначаются в С++ следующим образом:

== равно;

!= не равно;

< меньше;

> больше;

<= меньше или равно;

>= больше или равно.

Символ ! в языке С++ обозначает логическое отрицание.

Есть еще две логические операции: || означает или, а && – логическое И. Операции отношения имеют приоритет ниже арифметических операций, так что выражение вида k>n%i вычисляется как k>(n%i).

Приоритет && выше, чем у ||, но обе логические операции выполняются после операций отношения и арифметических.

В сомнительных случаях лучше расставлять скобки.

Пример 3.

```
if (num < 10)
{ // если введенное число меньше 10
cout << "Это число меньше 10." << endl;
}
else
{ // иначе
cout << "Это число больше либо равно 10" << endl;
}
```

Условный оператор с одной ветвью:

```
if (num!= 10) // если введенное число не равно 10
cout << "Это число не равно 10." << endl;
```

Распространенные ошибки:

1. Использование в выражении при проверке равенства вместо оператора (==) простое присваивание (=).

2. Неверная запись проверки на принадлежность диапазону. Условие $0 < x < 1$ необходимо записать следующим образом: *if (0 <x&&x< 1).*

Оператор множественного выбора switch

Общая форма оператора следующая:

```
switch(переменная выбора)
{
case const 1: операторы 1; break;
…
case const N: операторы N; break;
default: операторы N+1;
}
```

При использовании оператора switch сначала анализируется переменная выбора и проверяется, совпадает ли её значение со значением одной из констант. При совпадении выполняются операторы этого case. Конструкция default (может отсутствовать) выполняется, если результат выражения не совпал ни с одной из констант.

Пример 4.

```
int a=1;
```

```
switch(a)
{
   case 1:
      a++;
   case 2:
      a++;
   case 3:
      a++;
}
cout<<"a= "<<a;
```

Данная программа выведет а = 4.

Практическая часть

Задание 1. Создайте программу решения уравнения (x и y вводятся пользователем во время выполнения программы), учтите возможные условия арифметических операций.

Варианты заданий:

1. $\sqrt{x-y}/3x^2$
2. $\left(x^2-4x+y\right)/3$
3. $\sqrt{3x+y}/3x^2$
4. $\left(x^2-4y+y\right)/3$
5. $\sqrt{|x-y|}/3x$
6. $\left(x\ +4y-6\right)/3$
7. $\sqrt{10x-5}/3y$
8. $\left(x^2-7x+y\right)/y$
9. $\left(8x-4y-90\right)/x$
10. $x+\frac{3y}{x^2}$
11. $x^2+\frac{3y}{x}$
12. $\left(y^2-56\right)x\cdot y$

13. $x \cdot y - y^2 + 78/x$

14. $x(y-2) + \frac{3y}{x}$

15. $y(y-2) + \frac{3y}{x}$

Задание 2. Создайте программу решения уравнения вида $x^2 - a \cdot x + b - c$, где:

1. A=5, b=7, c=7 – Это константы, x вводит пользователь с клавиатуры;
2. A=5, b=7 – Это константы, c и x вводит пользователь с клавиатуры;
3. A=5, c=7 – Это константы, b и x вводит пользователь с клавиатуры;
4. b=7, c=7 – Это константы, a и x вводит пользователь с клавиатуры;
5. A=15, b=7, c=27 – Это константы, x вводит пользователь с клавиатуры;
6. A=3, b=6 – Это константы, c и x вводит пользователь с клавиатуры;
7. A=1, c=2 – Это константы, b и x вводит пользователь с клавиатуры;
8. b=3, c=3 – Это константы, a и x вводит пользователь с клавиатуры;
9. A=11, b=22, c=33 – Это константы, x вводит пользователь с клавиатуры
10. A=7, b=8 – Это константы, c и x вводит пользователь с клавиатуры;
11. A=8, c=6 – Это константы, b и x вводит пользователь с клавиатуры;
12. b=4, c=4 – Это константы, a и x вводит пользователь с клавиатуры;

13. A=2, b=2, c=4 – Это константы, x вводит пользователь с клавиатуры;

14. A=4, b=3 – Это константы, c и x вводит пользователь с клавиатуры;

15. A=2, c=2 – Это константы, b и x вводит пользователь с клавиатуры.

Задание 3. Решите задачу (*Номер варианта определяется по списку группы кратно количеству вариантов в задании*).

1. Пользователь вводит порядковый номер пальца руки. Необходимо показать его название на экран

2. Дано значение n = 1..7, которое есть номером дня недели. По значению n определить, выходной этот день или рабочий. Результат вывести на экран.

3. Даны три целых числа *a*, *b*, *c*. Разработать программу, которая находит минимальное значение между этими числами. Результат вывести на экран.

4. Дан номер месяца года *n*. По номеру месяца определить, сколько дней в этом месяце. Фрагмент кода, который решает данную задачу. Принять, что в феврале 28 дней. Результат вывести на экран.

5. Даны три целых числа: A, B, C. Проверить истинность высказывания: «Справедливо двойное неравенство A < B < C». Результат вывести на экран.

6. Даны три целых числа. Найти количество положительных и количество отрицательных чисел в исходном наборе. Результат вывести на экран.

7. Даны три целых числа. Определите, сколько среди них совпадающих. Программа должна вывести одно из чисел: 3 (если все совпадают), 2 (если два совпадает) или 0 (если все числа различны).

Контрольные вопросы:

1. Простые типы данных языка C++.
2. Структура программы на языке C++.

3. Стандартные библиотеки и их подключение.
4. Что такое идентификатор, переменная, константа?
5. Что такое совместимость типов?
6. Состав языка С++.
7. Синтаксис условного оператора if.

ЛАБОРАТОРНАЯ РАБОТА № 2.
Массивы

Цель работы: получить навыки создания и обработки одномерных и двухмерных массивов.

Порядок выполнения работы: изучить теоретическую часть, выполнить практические задания, оформить отчет, осуществить защиту лабораторной работы.

Теоретическая часть

Наиболее часто работа с массивами организуется посредством
циклов.

Оператор цикла for

Общий вид оператора:

for (инициализирующее выражение; условие; инкрементирующее выражение)

{

тело цикла

}

Инициализирующее выражение выполняется только один раз в начале выполнения цикла и, как правило, инициализирует счетчик цикла.

Условие содержит операцию отношения, которая выполняется в начале каждого цикла. Если условие равно 1 (true), то цикл повторяется, иначе выполняется следующий за телом цикла оператор. Инкрементирующее выражение, как правило, предназначено для изменения значения счетчика цикла.

Модификация счетчика происходит после каждого выполнения тела цикла.

Оператор цикла while

Оператор цикла с предусловием. Общий вид оператора:

while (условие)

{

тело цикла

}

Организует повторение операторов тела цикла до тех пор, пока условие истинно.

Оператор цикла do

Оператор цикла с постусловием. Общий вид оператора:

do {

тело цикла

}

while (условие);

Организует повторение операторов тела цикла до тех пор, пока условие истинно.

Массивы

Массив представляет нечто целое, что содержит целый набор однотипных пронумерованных элементов.

Объявляется создаваемый одномерный массив так:

int A[100]; //Массив A под 100 элементов тип массива int (т.е. это 100 целых чисел)

char S[256]; //Массив S типа char В массиве всего 256 элементов

В квадратных скобках указывается число элементов в массиве.

Присвоение первому элемента массива значения 333: *A[0]=333*;

Особенности работы с массивами:

Для создания массива компилятору необходимо знать тип данных и количество элементов в массиве.

Массивы могут иметь те же типы данных, что и простые переменные.

Квадратные скобки это своеобразный индикатор того, что происходит работа с массивом.

При объявлении массива, внутри квадратных скобок указывается число элементов для массива.

При использовании массива, внутри квадратных скобок указывается номер элемента из массива.

Номер элемента массива называется индексом массива.

Внешние и статические массивы можно инициализировать.

Автоматические и регистровые массивы инициализировать нельзя.

Любой массив требует такой же инициализации как и переменные, иначе в него может попасть информационный мусор.

Пример инициализации массива:

int A[10]={10,222,3,444,5,55,34,4,43,4};

Если у вас большой массив, но надо установить все элементы в нулевое значение, то не обязательно приписывать в фигурных скобках нули столько раз, сколько элементов в массиве, можно использовать простую конструкцию int A[1000]={0,};

Синтаксис C++ позволяет писать пустые квадратные скобки при объявлении массива. В этом случае компилятор сам определяет, сколько памяти нужно выделить массиву.

*int A[]={0,}; // Выделилось (1 байт * sizeof(int) = 2 байта) – под 2 элемента*

*int A[]={0,0,0,};//Выделилось (3 байта * sizeof(int) =12 байт) – под 4 элемента*

Узнать сколько байт съедает один элемент из массива, можно командой ***sizeof(A[0])*.** Узнать сколько элементов может поместиться в массив командой ***sizeof(A)/sizeof(A[0])*;** – применимо именно к массиву, у указателя узнать сколько вмещает элементов указатель не выйдет.

Пример 1. Создать массив из 10 чисел и вывести его на экран

#include <iostream>

using namespace std;

```
void main ()
{
const int N=10; //размер обычного массива можно определить константой
int A[N]={0,]; //массив из 10 элементов, каждый элемент равен 0
A[5]=100; // шестому элементу присвоили значение 100
for (int i=0;i<N; i++) cout<<A[i]<<" "; // вывод массива на экран поэлементно
cin.get();
}
```

Пример 2. Удалить из одномерного массива все отрицательные элементы

```
for (i=0; i<N; i++)
if (a[i]<0
{
for (j=i+1; j<N; j++) a[j-1]=a[j];
n--; i--;
}
```

C++ позволяет создавать многомерные массивы. Простейшим видом многомерного массива является двумерный массив. Двумерный массив - это массив одномерных массивов. Двумерный массив объявляется следующим образом:

тип имя_массива[размер второго измерения][размер первого измерения];

Следовательно, для объявления двумерного массива целых с размером 10 на 20 следует написать:

```
int d[10] [20];
```

Посмотрим внимательно на это объявление. В противоположность другим компьютерным языкам, где размерности массива отделяются запятой, C++ помещает каждую размерность в отдельные скобки.

Для доступа к элементу с индексами 3, 5 массива d следует использовать:

d[3] [5]

Двумерные массивы сохраняются в виде матрицы, где первый индекс отвечает за строку, а второй – за столбец. Это означает, что правый индекс изменяется быстрее левого, если двигаться по массиву в порядке расположения элементов в памяти.

Число байт в памяти, требуемых для размещения двумерного массива, вычисляется следующим образом:

*число байт = размер второго измерения * размер первого измерения * sizeof (базовый тип)*

Предполагая наличие в системе 2-байтных целых, целочисленный массив с размерностями 10 на 5 будет занимать 10×5×2, то есть 100 байт.

Пример 3. В двумерный массив заносятся числа от 1 до 12, после чего массив выводится на экран.

```
#include <stdio.h>
int main(void)
{
int t,i, num[3][4];
/* загрузка чисел */
for(t=0; t<3; ++t)
for (i=0; i<4; ++i)
num[t][i] = (t*4)+i+1;
/* вывод чисел */
for (t=0; t<3; ++t)
{
for (i=0; i<4; ++i)
printf("%d  ",num[t][i]);
printf ("\n");
}
return 0;
}
```

Практическая часть

Задание 1. Выполните задание согласно варианта.

1. Задан массив из k символов. Преобразовать массив следующим образом: сначала должны стоять цифры, входящие в массив, а затем все остальные символы. Взаимное расположение символов в каждой группе не должно изменяться.

2. Задан массив из k символов. Преобразовать массив следующим образом: расположить символы в обратном порядке.

3. Задан массив из k чисел. Найти число, наиболее часто встречающееся в этом массиве.

4. Задан массив из k чисел. Отсортировать элементы массива по возрастанию.

5. Задан массив из k чисел. Найти числа, входящие в массив только один раз.

6. Задан массив из k чисел. Сдвинуть элементы массива циклически на n позиций влево.

7. Задан массив из k чисел. Сдвинуть элементы массива циклически на n позиций вправо.

8. Задан массив из k чисел. Преобразовать массив следующим образом: все отрицательные элементы массива перенести в начало, а все остальные – в конец, сохранив исходное взаимное расположение, как среди отрицательных, так и среди положительных элементов.

9. Задан массив из k символов. Создать два новых массива: в первый перенести все цифры из исходного массива, во второй – все остальные символы.

10. Задан массив из k символов. Определить, симметричен ли он, т. е. читается ли он одинаково слева направо и справа налево.

11. Задано два массива. Найти наименьшие среди элементов первого массива, которые не входят во второй массив.

12. Задан массив из k чисел. Определить количество инверсий в массиве (т. е. таких пар элементов, в которых большее число находится слева от меньшего).

13. Задан массив из k символов. Удалить из него повторные вхождения каждого символа.

14. Задан массив из k символов. Определить количество различных элементов в массиве.

15. Задан массив из k символов латинского алфавита. Вывести на экран в алфавитном порядке все символы, которые входят в этот массив по одному разу.

Задание 2. Создайте приложение в котором пользователь вводит двумерный массив размером N×M, выводит его на экран, а так же выполняет дополнительное действие, описанное ниже:

1. N=3, M=4, выводит сумму элементов второй строки.
2. N=4, M=3, выводит сумму элементов первой строки.
3. N=3, M=4, выводит сумму элементов первого столбца.
4. N=3, M=4, выводит сумму элементов второго столбца.
5. N=5, M=2, выводит сумму элементов третьей строки.
6. N=4, M=2, выводит сумму элементов четвертой строки.
7. N=5, M=5, выводит сумму элементов третьего столбца.
8. N=5, M=5, выводит сумму элементов четвертого столбца.
9. N=4, M=3, выводит сумму элементов второй строки.
10. N=3, M=4, выводит сумму элементов первой строки.
11. N=4, M=3, выводит сумму элементов первого столбца.
12. N=4, M=3, выводит сумму элементов второго столбца.
13. N=5, M=3, выводит сумму элементов третьей строки.
14. N=5, M=2, выводит сумму элементов четвертой строки.
15. N=5, M=4, выводит сумму элементов третьего столбца.

Задание 3. Выполните задание согласно варианта.

1. Дана прямоугольная таблица, которая содержит не более 10 строк и не более 10 столбцов. Найти сумму элементов, которые расположены в строках с нечетными номерами.

2. Дан двумерный массив, который содержит не более 10 строк и не более 10 столбцов. Найти максимальный по абсолютной величине элемент и поменять его местами с последним элементом массива.

3. Даны оценки, полученные на 4 экзаменах во время сессии студентами одной группы, по 10 бальной системе. Определить сколько студентов не сдали сессию.

4. Дана прямоугольная таблица, которая содержит не более 10 строк и не более 10 столбцов. Найти сумму элементов, которые делятся на данное число X.

5. Дан двумерный массив, который содержит не более 10 строк и не более 10 столбцов. Найти минимальный элемент и поменять его местами с тем элементом, который стоит в конце массива.

6. Даны оценки, полученные на 4 экзаменах во время сессии студентами одной группы, по 10 бальной системе. Определить сколько студентов сдали сессию на 10 и 9 баллов.

7. Дана прямоугольная таблица, которая содержит не более 10 строк и не более 10 столбцов. Найти сумму элементов, у которых сумма делителей меньше данного числа X.

8. Дан двумерный массив, который содержит не более 10 строк и не более 10 столбцов. Создать новый массив, элементами которого являются суммы цифр каждого числа старого массива.

9. Даны оценки, полученные на 4 экзаменах во время сессии студентами одной группы, по 10 бальной системе. Определить сколько студентов сдали сессию только на 7.

10. Дана прямоугольная таблица, которая содержит не более 10 строк и не более 10 столбцов. Найти сумму отрицательных и сумму положительных элементов и сравнить их по модулю.

11. Дан двумерный массив, который содержит не более 10 строк и не более 10 столбцов. Создать новый массив, элементами которого являются суммы делителей каждого числа старого массива.

12. Даны оценки, полученные на 4 экзаменах во время сессии студентами одной группы, по 10 бальной системе. Определить сколько студентов сдали сессию на балл не ниже 6.

13. Дана прямоугольная таблица, которая содержит не более 10 строк и не более 10 столбцов. Найти и вывести те элементы, которые больше предыдущего, стоящего с ним в одной строке.

14. Дан двумерный массив, который содержит не более 10 строк и не более 10 столбцов. Создать новый массив, элементами которого являются суммы первой и последней цифры каждого числа старого массива.

15. Есть группа спортсменов из 7 человек. Для каждого спортсмена приводится его рост и вес. Вес спортсмена считается нормальным, если от роста отнять 100 и полученное число отличается от веса не более чем на 3. Вывести номера тех спортсменов, чей вес превышает норму.

Задание 4. Выполните задание согласно варианта.

1. Сформируйте массив L(I, J) с помощью датчика случайных чисел. Увеличить каждый элемент массива в 3 раза и поменяйте знак на противоположный. Массив выведите на экран в виде таблицы.

2. Дана матрица целых чисел размера N×M. Вывести номер строки, содержащей минимальное число одинаковых элементов.

3. Дана целочисленная прямоугольная матрица размера M×N. Сформировать одномерный массив, состоящий из элементов, лежащих в интервале [1, 10]. Найти произведение элементов полученного одномерного массива.

4. Дана целочисленная матрица размера 8×5. Определить: а) сумму всех элементов второго столбца массива; б) сумму всех элементов 3-й строки массива.

5. Дан двумерный массив. Вставьте первую строку после строки, в которой находится первый встреченный минимальный элемент.

6. Вычислить средние арифметические значения неотрицательных элементов каждой строки матрицы [N; M].

7. Преобразовать исходную матрицу A(M×N) так, чтобы последний элемент каждой строки был заменен суммой предыдущих элементов той же строки.

8. Задана матрица A (n, n) действительных чисел. «Перевернуть» в ней главную и побочную диагонали (переписать цифры в обратном порядке). Для того, чтобы

«перевернуть» диагонали, не надо перебирать всю матрицу. Достаточно перебрать половину

9. В матрице найти элементы (их позицию), которые являются одновременно минимальными в строке и столбце.

10. В заданной матрице найти произведение ненулевых элементов.

11. Задан двухмерный массив чисел. Значение элементов матрицы вводятся с клавиатуры. Вычислить сумму элементов матрицы, индексы которых составляют в сумме заданное число K (это число вводится пользователем). Вывести результат.

12. Задан двухмерный массив чисел. Значение элементов матрицы вводятся с клавиатуры. Определить сумму одинаковых элементов матрицы и вывести те из них, которые находятся на нечетных столбцах. Вывести результат.

13. Посчитать суммы каждой строки и каждого столбца матрицы. Вывести суммы строк в конце каждой строки, а суммы столбцов под соответствующими столбцами.

14. Матрицу 10×20 заполнить случайными числами от 0 до 15. Вывести на экран саму матрицу и номера строк, в которых число 5 встречается три и более раз.

15. Заполнить матрицу случайными целыми неотрицательными числами. Вывести на экран матрицу в табличном виде так, чтобы сначала шли нечетные числа, затем четные. Также вывести количество четных и нечетных чисел в матрице.

Контрольные вопросы:

1. Что такое массив?
2. Опишите, как объявляется массив
3. Опишите, что такое инициализация массива и как она выполняется.
4. Какие типы данных может содержать массив?
5. Какие отличия имеет двухмерный массив от одномерного?

6. Как вычисляется число байт, требуемых для хранения в памяти массива?

ЛАБОРАТОРНАЯ РАБОТА № 3.
Перечисления. Структуры. Объединения

Цель работы: получить навыки работы с перечислениями, структурами и объединениями.

Порядок выполнения работы: изучить теоретическую часть, выполнить практические задания, оформить отчет, осуществить защиту лабораторной работы.

Теоретическая часть

Перечисления

Перечисления (enum) представляют еще один способ определения своих типов. Их отличительной особенностью является то, что они содержат набор числовых констант.

Определим простейшее перечисление:

```
enum seasons
{
spring,
summer,
autumn,
winter
};
```

Для определения перечисления применяется ключевое слово enum, после которого идет название перечисления. Затем в фигурных скобках идет перечисление констант через запятую. Каждой константе по умолчанию будет присваиваться числовое значение, начиная с нуля. То есть в данном случае spring=0, а winter=3.

Пример 1. Используем перечисление

```
#include <iostream>
enum seasons
{
spring,
```

```
summer,
autumn,
winter
};
int main()
{
seasons currentSeason = autumn;
std::cout << "Season: " << currentSeason << std::endl;
return 0;
}
```

Мы можем определить переменную типа seasons и присвоить этой переменной значение одной из констант, объявленных в перечислении. Но фактически это будет числовое значение. В частности, консольный вывод данной программы: *Season: 2*.

В то же время перечисления – это отдельный тип, поэтому мы не можем присвоить переменной напрямую числовое значение:

```
seasons currentSeason = 2;  // ошибка
```

Если нас не устраивают значения по умолчанию для констант, то мы можем явным образом задать значения. Например, установить начальное значение:

```
enum seasons
{
spring = 1,
summer, //2
autumn, //3
winter  //4
};
```

В этом случае значения второй и последующих констант будет увеличиваться на единицу.

Также можно задать значение для каждой константы:

```
enum seasons
{
spring = 1,
summer = 2,
```

autumn = 4,
winter = 8
};

Перечисления могут использоваться, когда у нас есть ряд логически связанных констант, которые естественно лучше определить в одном общем типе данных.

Структуры

Структура представляет собой переменную, группирующую связанные части информации, называемые элементами, типы которых могут различаться. Группируя данные в одну переменную подобным образом, вы упрощаете ваши программы, снижая количество переменных, которыми необходимо управлять, передавать в функции и т. д.

Концепции структур:

– Структуры позволяют вашим программам группировать в одной переменной связанные данные, типы которых могут различаться.

– Структура состоит из одной или нескольких частей данных, называемых элементами.

– Для определения структуры внутри программы следует указать имя структуры и ее элементы.

– Каждый элемент структуры имеет тип, например char, int и float, и имя каждого элемента должно быть уникальным.

– После того как ваша программа определит структуру, она может объявить переменные типа этой структуры.

– Для изменения элементов структуры внутри функции ваши программы должны передать структуру в функцию с помощью адреса.

Объявление структуры

Структура определяет шаблон, с помощью которого ваша программа может позднее объявить одну или несколько переменных. Другими словами, ваша программа сначала определяет структуру, а затем объявляет переменные типа этой структуры. Для определения структуры ваши программы

используют ключевое слово struct, за которым обычно следует имя и левая фигурная скобка. Следом за открывающей фигурной скобкой вы указываете тип и имя одного или нескольких элементов. За последним элементом вы размещаете правую закрывающую фигурную скобку. В этот момент вы можете (необязательно) объявить переменные данной структуры:

```
struct name
{
int member_name_1 ;// Объявления элементов структуры
float member_name_2;
 } variable; //Объявление переменной
```

Пример 1. Объявление и заполнение структуры.

```
#include <iostream>
using namespace std;
struct Struct{
int a;
};
void main(){
Struct a;
a.a=4;
cout<<a.a<<endl;
}
```

Пример 2. Объявление и заполнение структуры.

```
struct database {
int id_number;
int age;
float salary;
};
int main()
{
database employee;
employee.age = 22;
employee.id_number = 1;
```

employee.salary = 12000.21;
}

Структура database содержит три переменные доступ к которым осуществляется с помощью оператора «.».

Указатели

Если вы работаете с указателем на структуру, то для доступа к переменным надо использовать оператор «->» вместо точки. Все свойства указателей не изменяются.

Пример 3

```
#include <iostream>
using namespace std;
struct xampl {
int x;
};
int main()
{
xampl structure;
xampl *ptr;
structure.x = 12;
ptr = &structure;
cout<< ptr->x;
cin.get();
}
```

Объединения

Объединение – это объект, позволяющий нескольким переменным различных типов занимать один участок памяти. Объявление объединения похоже на объявление структуры:

```
union union_type {
int i; char ch;
};
```

Когда объявлено объединение, компилятор автоматически создает переменную достаточного размера для хранения наибольшей переменной, присутствующей в объединении.

Для доступа к членам объединения используется синтаксис, применяемый для доступа к структурам – с помощью операторов «точка» и «стрелка». Чтобы работать с объединением напрямую, надо использовать оператор «точка». Если к переменной объединения обращение происходит с помощью указателя, надо использовать оператор «стрелка».

Например, для присваивания целого числа 10 элементу i объединения *cnvt* следует написать: *cnvt.i = 10*;

Использование объединений помогает создавать машинно-независимый (переносимый) код. Поскольку компилятор отслеживает настоящие размеры переменных, образующих объединение, уменьшается зависимость от компьютера. Не нужно беспокоиться о размере целых или вещественных чисел, символов или чего-либо еще.

Объединения часто используются при необходимости преобразования типов, поскольку можно обращаться к данным, хранящимся в объединении, совершенно различными способами.

Практическая часть

Задание 1. Создайте программу, реализующую работу со структурой (ввод-вывод данных) согласно варианта.

1. Структура «Класс»: Номер по списку, Фамилия ученика, Имя ученика, Отчество ученика, адрес жительства, домашний телефон.
2. Структура «Кладовка»: Код товара, Наименование товара, количество штук, вес одной штуки.
3. Структура «Прокат DVD»: код диска, Название диска, количество штук, стоимость проката, год выпуска.
4. Структура «Расписание занятий»: Код дня недели, День недели, 1-я пара, 2-я пара, 3-я пара.
5. Структура «Библиотека»: Код книги, название книги, автор, год издания.

6. Структура «Учебная группа»: Номер по списку, Фамилия студента, Имя студента, Отчество студента, адрес жительства, мобильный телефон.

7. Структура «Класс»: Номер по списку, Фамилия ученика, Имя ученика, Отчество ученика, адрес жительства, домашний телефон.

8. Структура «Кладовка»: Код товара, Наименование товара, количество штук, вес одной штуки.

9. Структура «Прокат DVD»: код диска, Название диска, количество штук, стоимость проката, год выпуска.

10. Структура «Расписание занятий»: Код дня недели, День недели, 1-я пара, 2-я пара, 3-я пара.

11. Структура «Библиотека»: Код книги, название книги, автор, год издания.

12. Структура «Учебная группа»: Номер по списку, Фамилия студента, Имя студента, Отчество студента, адрес жительства, мобильный телефон.

13. Структура «Класс»: Номер по списку, Фамилия ученика, Имя ученика, Отчество ученика, адрес жительства, домашний телефон.

14. Структура «Кладовка»: Код товара, Наименование товара, количество штук, вес одной штуки.

15. Структура «Прокат DVD»: код диска, Название диска, количество штук, стоимость проката, год выпуска.

Контрольные вопросы:

1. Понятия «структура», «перечисление» и «объединение».
2. Опишите концепции структур.
3. Перечислите операторы для доступа к структурам, объединениям.

ЛАБОРАТОРНАЯ РАБОТА № 4.
Битовые поля. Указатели

Цель работы: получить навыки работы с битовыми полями.

Порядок выполнения работы: изучить теоретическую часть, выполнить практические задания, оформить отчет, осуществить защиту лабораторной работы.

Теоретическая часть

Битовое поле – это своеобразная структура, которая позволяет работать с отдельными битами. Причины использования битовых полей:

Если ограничено место для хранения информации, можно сохранить несколько логических (истина/ложь) переменных в одном байте.

Некоторые интерфейсы устройств передают информацию, закодировав биты в один байт.

Некоторым процедурам кодирования необходимо получить доступ к отдельным битам в байте.

Синтаксис объявления битовых полей следующий:

struct [имя_структуры] {
тип [имя_битового_поля1]: длина;
тип [имя_битового_поля2]: длина;
. . . . .
тип [имя_битового_поляN]: длина;
} [имя_объекта];

В языке С в качестве типа битовых полей обязательно нужно использовать *int* или *unsigned* или *signed* . В С++ разрешено использовать кроме перечисленных выше любой тип, интерпретируемый как целый: *char*, *bool*, *short*, *long* и перечисления. Длина битового поля задается целочисленным значением, которое определяет, сколько битов необходимо выделить указанному полю.

Пример 1. Битовое поле:

struct {
unsigned name1: 1;

unsigned name2: 3;
unsigned name3: 5;
} obj;
obj.name1 = 1;
obj.name3 = 30;

В приведенном примере мы создали три битовых поля. Для хранения элемента *name1* выделено 1 бит, для хранения элемента *name2* - 3 бита, а для хранения *name3* выделено 5 бит. После объявления битовых полей в нашем примере происходит присвоение значений битовым полям. При этом удостоверьтесь, что присваиваемые значения не превышают размер поля.

Размер одного битового поля не должен превышать размер типа данного поля. То есть, если битовое поле объявлено как *unsigned* или *int*, то размер такого поля не должен превышать 32 бита.

В следующем примере объявляется структура, которая содержит битовые поля:

struct Date {
unsigned short nWeekDay : 3; // 0..7 (3 bits)
unsigned short nMonthDay : 6; // 0..31 (6 bits)
unsigned short nMonth : 5; // 0..12 (5 bits)
unsigned short nYear : 8; // 0..100 (8 bits)
};

Ссылка на битовое поле выполняется по *имени_битового_поля*. Если имя поля не указано, запрошенные биты все равно выделяются, но доступ к ним невозможен. Такое поле называется неименованным битовым полем. Существует множество ситуаций, в которых оправдано использование неименованных битовых полей: они ничем не хуже неименованных параметров функций объявление которых можно отделить от определения (инициализации) случаи:

float sum (float a, float b) и *float sum (float, float);*.
struct Example1 {
nsigned n1: 6;

unsigned : 2;
unsigned n2: 14;
unsigned : 2;
unsigned n3: 5;
};

Неименованные битовые поля с шириной поля 0 задают выравнивание следующего поля по границе максимального типа элементов структуры.

Пример 2. Структура битовых полей.

struct Example2 {
unsigned n1: 12;
unsigned : 0;
unsigned n2: 8;
};

использует неименованное поле нулевой ширины, чтобы пропустить оставшиеся биты в том элементе памяти, в котором хранится *n1*, и выровнять *n2* по границе следующего элемента памяти. Элементом памяти выступает в данном случае 4 байта, то есть размер *unsigned int*. В общей сложности данная структура будет занимать в памяти 8 байт.

К битовым полям не может быть применена операция получения адреса (&) и поэтому не существует указателей на поля.

Доступ к элементам битового поля осуществляется так же, как и доступ к обычным членам структуры. Например:

#include struct demo
{
unsigned a: 1;
signed b: 3;
int c: 6; };
int main () {
struct demo s;
s.a = 1;
s.b = -1;

```
s.c = -4;
printf("s.a = %u\n", s.a); // печать: s.a = 1
printf("s.b = %d\n", s.b); // печать: s.b = -1
printf("s.c = %d\n", s.c); // печать: s.c = -4
return 1;
}
```

Указатели

Указатели представляют собой объекты, значением которых служат адреса других объектов (переменных, констант, указателей) или функций. Как и ссылки, указатели применяются для косвенного доступа к объекту. Однако в отличие от ссылок указатели обладают большими возможностями.

Для определения указателя надо указать тип объекта, на который указывает указатель, и символ звездочки *. Например, определим указатель на объект типа *int*: *int *p;*

Пока указатель не ссылается ни на какой объект. При этом в отличие от ссылки указатель необязательно инициализировать каким-либо значением. Теперь присвоим указателю адрес переменной:

```
int x = 10;    // определяем переменную
int *p;        // определяем указатель
p = &x;        // указатель получает адрес переменной
```

Для получения адреса переменной применяется операция *&*. Что важно, переменная x имеет тип *int*, и указатель, который указывает на ее адрес, тоже имеет тип *int*. То есть должно быть соответствие по типу.

Если мы попробуем вывести адрес переменной на консоль, то увидим, что он представляет шестнадцатиричное значение:

```
#include <iostream>
int main()
{
int x = 10;    // определяем переменную
int *p;        // определяем указатель
p = &x;        // указатель получает адрес переменной
```

```
std::cout << "p = " << p << std::endl;
return 0;
}
```

Практическая часть

Задание 1. Создайте структуру с битовыми полями, описывающую текущий момент времени от секунды до года. Обеспечьте ввод/вывод информации.

Задание 2. Решите задачу.

1. Ввести значение 2-х целых переменных a и b. Направить два указателя на эти переменные. С помощью указателя увеличить значение переменной a в 2 раза. Затем поменять местами значения переменных a и b через их указатели.

2. Ввести значение 2-х целых переменных a и b. Направить два указателя на эти переменные. С помощью указателя увеличить значение переменной a в 2 раза если a>b иначе b уменьшить в 2 раза.

3. Ввести значение 2-х вещественных переменных a и b. Направить два указателя на эти переменные. С помощью указателя увеличить значение переменной a в 3 раза. Затем поменять местами значения переменных a и b через их указатели.

4. Ввести значение 2-х вещественных переменных a и b. Направить два указателя на эти переменные. Если a>b, то с помощью указателя увеличить значение переменной a на 3 и b уменьшить в 3 раза, в противном случае a уменьшить в 2 раза и b увеличить на 3.

5. Ввести значение 2-х символьных переменных a и b. Направить два указателя на эти переменные. С помощью указателя изменить значение переменной a. Затем поменять местами значения переменных a и b через их указатели.

6. Ввести значение 2-х целых переменных a и b. Направить два указателя на эти переменные. Большее из них с помощью указателя увеличить в 5 раз и меньшее уменьшить на 5.

7. Ввести значение 3-х целых переменных a и b и c. Направить указатели на эти переменные. С помощью указателя увеличить значение переменной a в 2 раза. Затем поменять местами значения переменных c и b через их указатели.

8. Ввести значение 3-х вещественных переменных a и b и c. Направить указатели на эти переменные. С помощью указателя увеличить значение переменной c в 3 раза. Затем поменять местами значения переменных a и c через их указатели.

9. Ввести значение 2-х вещественных переменных a и b. Направить два указателя на эти переменные. Большее из них с помощью указателя увеличить на 7 и меньшее уменьшить на 3.

10. Ввести значение 2-х символьных переменных a и b. Направить два указателя на эти переменные. Затем поменять местами значения переменных a и b через их указатели.

11. Ввести значение 2-х целых переменных a и b. Направить два указателя на эти переменные. Затем поменять местами значения переменных a и b через их указатели.

12. Ввести значение 2-х вещественных переменных a и b. Направить два указателя на эти переменные. Затем поменять местами значения переменных a и b через их указатели.

13. Ввести значение 2-х целых переменных a и b. Направить два указателя на эти переменные. С помощью указателя увеличить значение переменной a в 2 раза, a b уменьшить в 2 раза.

14. Ввести значение 2-х вещественных переменных a и b. Направить два указателя на эти переменные. С помощью указателя увеличить значение переменной a в 3 раза, a b уменьшить в 3 раза

15. Ввести значение 2-х вещественных переменных a и b. Направить два указателя на эти переменные. С помощью указателя увеличить значение переменной a в 3 раза, a b уменьшить в 3 раза.

ЛАБОРАТОРНАЯ РАБОТА № 5.

Функция

Цель работы: получить навыки использования пользовательских функций.

Порядок выполнения работы: изучить теоретическую часть, выполнить практические задания, оформить отчет, осуществить защиту лабораторной работы.

Теоретическая часть

Функцией называется выделенная последовательность инструкций, предназначенных для решения определенной задачи.

Есть несколько причин для использования пользовательских функций, во-первых, программа приобретает некоторую структуру и, тем самым, становится более понятной и упорядоченной, во-вторых, исключаются повторы похожих участков текста, то есть текст программы оптимизируется.

Функция может многократно вызываться из различных частей программы, в общем случае она выполняет следующие действия:

– получает параметры;

– выполняет инструкции, согласно заложенному алгоритму;

– может возвращать результат в вызывающую программу.

С использованием функций в языке C++ связаны понятия, которые условно можно разделить на две группы.

В первую группу входят определение, прототип и вызов функции – все три понятия связанны с подготовкой функции к работе.

Вторая группа, параметры и возвращаемое значение, обеспечивает связь функции с «внешней средой». Функция может многократно вызываться из различных частей программы, при этом необходимо обеспечить её связь с вызывающей программой, из вызывающей программы в

функцию передать необходимые для работы данные, а по окончанию работы принять результат.

Определение функции – это описание действий, выполняемых функцией согласно требованиям алгоритма. Именно эта часть программы будет впоследствии многократно вызываться из других частей программы.

Прототип функции (объявление) используется в том случае, если вызов функции предшествует её определению или если определение и вызовы функции находятся в разных файлах.

Вызов функции обеспечивает связь с вызывающей программой. При вызове:

– передаются параметры из вызывающей программы в функцию

– управление передается первой инструкции в теле функции,

– после завершения работы функции в вызывающую программу передается возвращаемое значение, управление возвращается в точку вызова.

Определение функции состоит из заголовка и тела, например:

double f1 (int a, int f) //заголовок

{ ... } // тело

В данном примере определена функция *f1* с двумя параметрами *int a* и *int f* , возвращающая значение типа double.

Тип функции (в нашем примере double) определяет тип значения, которое возвращает функция. Если тип не указан, то предполагается, что функция возвращает целое значение, типа *int*. Если функция не должна возвращать значение, то используется тип *void*, который в данном случае означает отсутствие значения.

В заголовке функции параметры называются формальными, и служат для её связи с вызывающей программой. Формальные параметры создаются в начале работы функции – это локальные переменные, которые инициализируются значениями, полученными из вызывающей программы при вызове функции.

Параметры при вызове функции получают конкретные значения и называются фактическими параметрами, например, вызов функции может выглядеть так:

```
…
double z;
int s1=10;
…
z = f1(s1, 5); //вызов функции f1, s1 и 5 фактические параметры
```

Передача параметров в функцию и возврат значений

Параметры позволяют передать информацию из вызывающей программы в функцию. В теле функции параметрами можно пользоваться так же, как и локальными переменными. При вызове функции:

– для каждого формального параметра создаётся локальная переменная;

– начальными значениями созданных переменных являются фактические параметры, определяемые при вызове функции.

В языке С функция может возвращать только одно значение, для этого её выполнение следует завершить оператором return, содержащим некоторое выражение. Следует отметить, что тип функции в определении должен соответствовать типу выражения оператора return в её теле.

Пример 1. Функция нахождения суммы двух чисел

```
…
int sum(int a, int b)
{ return a+b;}
int _main()
{ int x =5,y=10,z;
z = sum(x,y);
…
}
```

При вызове функции создаются две локальные *a* и *b* переменные, которые инициализируются фактическими параметрами *x* и *y* :

a=x

b=y

Функция возвращает значение типа *int*, которое записывается в переменную *z*.

Использование прототипа функции

В языке Си определения функций могут следовать за определением функции main, перед ним, или находится в другом файле.

Положение определения функции :

– за функцией main;
– перед функцией main;
– в другом модуле (файле).

Однако во всех случаях к моменту вызова функция должна быть определена или объявлена. Это требование обусловлено тем, что компилятор должен осуществить проверку корректности вызова функции (проверку соответствия количества и типов фактических параметров, количеству и типам формальных параметров).

Когда вызов функции предшествует её определению, эта проверка выполняется по прототипу.

Прототип напоминает заголовок в определении функции:

– Тело функции отсутствует;
– Имена формальных параметров могут быть опущены (типы параметров опускать нельзя!).

Прототипы:

int power (int base, int n);

или

int power (int , int);

Если функция определена до функции main() – прототип не обязателен.

Практическая часть

Задание 1. Решите задачу.

1. Определить функцию для вычисления расстояния между двумя точками, заданными своими координатами (вычисление длины отрезка). Создать программу ввода информации о нескольких отрезках и вычислении их длин.
2. Определить функцию возможности построения треугольника по его сторонам. Определить функцию вычисления площади треугольника по его сторонам.
3. Создать программу многократного ввода сторон треугольника и вычисление его площади предусмотреть вывод сообщения о невозможности построения треугольника.
4. Определить функцию вычисления объема куба, заданного стороной. Создать программу многократного ввода стороны куба и вычисление его объема.
5. Определить функцию вычисления площади прямоугольника по его сторонам.
6. Создать программу многократного ввода сторон прямоугольника и вычисления его площади.
7. Определить функцию вычисления площади поверхности цилиндра, заданного радиусом и высотой.
8. Создать программу ввода радиуса и высоты и вычисления площади поверхности цилиндра.
9. Определить функцию вычисления периметра квадрата, заданного диагональю.
10. Создать программу многократного ввода диагонали квадрата и вычисления его периметра.
11. Определить функцию вычисления объема конуса, заданного радиусом высотой.
12. Создать программу многократного ввода радиуса и высоты и вычисления объема конуса.
13. Определить функцию вычисления площади прямоугольника по его сторонам.

14. Создать программу многократного ввода информации о прямоугольниках и определить их площадь.

15. Определить функцию вычисления объема куба, заданного стороной. Создать программу многократного ввода стороны куба и вычисление его объема.

16. Определить функцию вычисления периметра прямоугольника по его сторонам.

17. Создать программу многократного ввода сторон прямоугольника и вычисления его периметра.

18. Определить функцию вычисления площади круга, заданного радиусом.

19. Создать программу многократного ввода двух радиусов и вычисления площади «бублика» между двумя кругами.

20. Определить функцию возможности построения треугольника по его сторонам. Определить процедуру определения типа треугольника (равносторонний, равнобедренный, разносторонний)

21. Создать программу многократного ввода сторон треугольника и определения его типа, предусмотреть вывод сообщения о невозможности построения треугольника.

22. Определить функцию вычисления площади круга, заданного радиусом.

23. Создать программу многократного ввода радиуса и вычисления площади круга.

24. Определить функцию вычисления площади квадрата, заданного диагональю. Создать программу многократного ввода диагонали квадрата и вычисления его площади.

25. Определить функцию возможности построения треугольника по его сторонам. Определить процедуру вычисления периметра треугольника.

26. Создать программу многократного ввода сторон треугольника и определения его периметра, предусмотреть вывод сообщения о невозможности построения треугольника.

Задание 2. Решите задачу.

1. Создать и распечатать четыре исходных массива целых двухзначных чисел X(N) , Y(M), Z(K), Q(L). Сформировать и распечатать результирующий массив R(4), содержащий максимальные элементы исходных массивов. В массиве с максимальным значением максимума заменить все нули максимумом, результат вывести на экран. Использовать функции: инициализации, вывода, поиска максимального элемента в произвольном массиве целых чисел.
2. Создать и распечатать исходную матрицу вещественных двухзначных чисел N×M (S знаков после запятой). Вычислить и отобразить суммы строк. Определить номер строки с максимальной суммой, вывести сообщение. Использовать функции: инициализации, вывода, вычисления суммы заданной строки матрицы.
3. Создать и распечатать два исходных массива вещественных чисел X(N) , Y(M).
4. (S знаков после запятой). Определить массив, в котором количество отрицательных элементов меньше. Использовать функции: инициализации, вывода, определения количества отрицательных элементов в произвольных массивах целых чисел.
5. Создать и распечатать две исходных матриц вещественных трехзначных чисел N×M (S знаков после запятой). Определить матрицу с большей суммой выше главной диагонали. Использовать функции: инициализации, вывода, определения суммы выше главной диагонали в матрице вещественных чисел.
6. Создать и распечатать исходную матрицу вещественных трехзначных чисел N×M (S знаков после запятой). Вычислить и отобразить суммы столбцов. Определить номер столбца с минимальной суммой, вывести сообщение. Использовать функции: инициализации, вывода, вычисления суммы заданного столбца произвольной матрицы.

7. Создать два исходных массива целых чисел X(N) , Y(M). Определить массив с меньшим значением минимума, вывести сообщение. В массиве с большим значением минимума подсчитать количество нулей. Использовать функции: инициализации, вывода, поиска минимального элемента в произвольном массиве вещественных чисел.

8. Создать и распечатать исходную матрицу вещественных двухзначных чисел N×M (S знаков после запятой). Вычислить и отобразить максимумы столбцов. Определить номер столбца с минимальным максимумом, вывести сообщение. Использовать функции: инициализации, вывода, вычисления максимума заданного столбца произвольной матрицы.

9. Создать и распечатать две исходных матрицы целых двухзначных чисел N×N. Определить матрицу с большей суммой побочной диагонали, вывести сообщение. Использовать функции: инициализации, вывода, определения суммы побочной диагонали произвольной матрицы.

10. Создать и распечатать исходную матрицу целых двузначных чисел N×M. Определить столбец с максимальным числом элементов, кратных K, (K вводится с клавиатуры). Использовать функции: инициализации, вывода, вычисления числа элементов, кратных K в заданном столбце матрицы.

11. Создать и распечатать два исходных массива вещественных чисел X(N) , Y(M) (S знаков после запятой). Определить массив с большим числом отрицательных элементов, вывести сообщение. В массиве с меньшим числом отрицательных элементов определить минимальный отрицательный элемент. Использовать функции: инициализации, вывода, поиска минимального отрицательного элемента, поиска количества отрицательных элементов в произвольном массиве вещественных чисел.

12. Создать и распечатать две исходных матрицы целых двухзначных чисел N×M. Вычислить и отобразить суммы

столбцов. Определить номер столбца с минимальной суммой, вывести сообщение.

13. Определить матрицу с максимальной суммой столбца, вывести сообщение (указать матрицу и номер столбца). Использовать функции: инициализации, вывода, определения суммы заданного столбца произвольной матрицы.

14. Создать и распечатать три исходных массива целых чисел X(N) , Y(M), Z(K). Сформировать и распечатать результирующий массив R(3), содержащий максимальные положительные элементы исходных массивов. Вывести на печать массив с минимальным количеством положительных элементов.

15. Использовать функции: инициализации, вывода, поиска максимального положительного элемента и функцию подсчета количества положительных элементов в произвольных массивах целых чисел.

16. Создать и распечатать две исходных матрицы целых трехзначных чисел N×N. Определить и распечатать матрицу с меньшей суммой главной диагонали. Использовать функции: инициализации, вывода, определения суммы главной диагонали произвольной матрицы целых чисел.

17. Создать и распечатать два исходных массива целых двузначных чисел X(N) , Y(M) Определить массив, в котором количество элементов кратных K больше (K ввести с клавиатуры). Использовать функции: инициализации, вывода, определения количества элементов, кратных K в массиве целых чисел.

18. Создать и распечатать исходную матрицу целых двухзначных чисел N×M. Вычислить и отобразить число нулей в каждой строке. Определить строку с максимальным числом нулей. Использовать функции: инициализации, вывода, определения числа нулей в заданной строке произвольной матрицы целых чисел.

Контрольные вопросы:

1. Поясните общие понятия, связанные с использованием функции: определение, вызов, параметры функции.
2. Что такое прототип функции, когда он используется.
3. Что такое тип функции?
4. Какую роль выполняют параметры в функции? Расскажите о формальных и фактических параметрах функции.
5. Расскажите об использовании переменных в функциях, какая разница между локальной и глобальной переменной?
6. Как передать массив в функцию?

ЛАБОРАТОРНАЯ РАБОТА № 6.

Переменные и классы

Цель работы: получить навыки работы с классами.

Порядок выполнения работы: изучить теоретическую часть, выполнить практические задания, оформить отчет, осуществить защиту лабораторной работы.

Теоретическая часть

В стандарте C++ под классом (class) подразумевается пользовательский тип, объявленный с использованием одного из ключевых слов class, struct или union, под структурой (structure) подразумевается класс, определённый через ключевое слово struct, и под объединением (union) подразумевается класс, определённый через ключевое слово union.

Особенности классов:

1. Класс является собственным типом данных;
2. Класс – это некоторая идея еще не существующего объекта, в которой воедино собраны все детали, все свойства и все нужные действия, необходимые для этого объекта. Например:

```
void main()
{
  int   //Вы еще не прописали переменной, но она задумана и тип уже прописан.
```

return;
}

3. Создается класс с помощью слова class, за которым следует открывающаяся и закрывающаяся фигурные скобки. После закрывающейся фигурной скобки ставится точка с запятой. Внутри фигурных скобок пишется вся информация для класса. После создания пустого класса внутри этого класса прописывается вся информация для выполнения поставленной задачи. Важные моменты то, что внутри класса используются слова public, private и protected;

4. По сути, класс есть структура. Отличается класс от структуры только модификатором доступа по умолчанию (private);

5. Функции внутри класса называют методами класса или полями класса;

6. Объект есть воплощение вашей идеи, описанной в классе во что-то реально существующее.

Пример 1.

Объявляем класс

```
#include <conio.h>
#include <iostream>
using namespace std;
class sum
{
};
void main ()
{
}
```

Описываем класс

```
#include <conio.h>
#include <iostream>
using namespace std;
class sum
{
```

```
int x, y;
public:
void get_xy()
{
cout<<”Input x”;
cin>>x;
cout<<”Input y”;
cin>>y;

}
int sum_xy() return x+y;
};
void main ()
{
}
```
Описываем работу с классом
```
#include <conio.h>
#include <iostream>
using namespace std;
class sum
{
int x, y;
public:
void get_xy()
{
cout<<”Input x”;
cin>>x;
cout<<”Input y”;
cin>>y;
}
int sum_xy() return x+y;
} s1;
void main ()
{
```

s1.get_xy();
cout<< s1.get_xy();
}

Практическая часть

Задание 1. Реализовать через класс расчет уравнения, X и Y вводятся программно.

1. X^2+Y^2, X=5, Y=6
2. $3X-Y^3$, X=10, Y=2
3. X^2+Y-2, X=7, Y=6
4. $2X^2+Y-2$, X=7, Y=6
5. $X^2+4Y-12$, X=7, Y=6
6. X+Y+, X=7, Y=6
7. $3X-Y^3$, X=11, Y=3
8. X^2+Y-2, X=17, Y=4
9. $2X^2+Y-2$, X=9, Y=7
10. $X^2+4Y-12$, X=10, Y=4
11. X^2+Y^2, X=2, Y=3
12. $3X-Y^3$, X=3, Y=2
13. X^2+Y-2, X=6, Y=8
14. $2X^2+Y-2$, X=8, Y=8
15. $X^2+4Y-12$, X=6, Y=9

Задание 2. Реализовать программу работы с классом, программа должна обеспечивать ввод вывод данных из объектов класса на примере одного объекта.

1. Класс Vizitka. Содержит: Фамилию, Имя, Номер телефона.
2. Класс Vid_zanatiy. Содержит: Номер, Название вида, Описание вида.
3. Класс Времена года. Содержит: Название времени года, Описание времени года.
4. Класс Zametki. Содержит: Номер заметки, дата заметки, текст заметки.
5. Класс Gruppa. Содержит: Номер группы, номер в группе, ФИО.

6. Класс Raspisanie. Содержит: Номер пары, Время начала, время окончания.

7. Класс Moi_mesta. Содержит: Номер, Название места, Мои комментарии.

8. Класс Zveri_lesa. Содержит: Номер, Название зверя, описание зверя.

9. Класс Riby. Содержит: Номер, Название рыбы, описание рыбы.

10. Класс Pamyatniki_Minska. Содержит: Название памятника, адрес, описание.

11. Класс Reki. Содержит: Название реки, длина, описание реки.

12. Класс Ozera. Содержит: Название озера, площадь, описание озера.

13. Класс Goroda. Содержит: Название города, количество жителей, год основания.

14. Класс Frukti. Содержит: Название фрукта, Описание, страна импортер.

15. Класс Ovoschi. Содержит: название овоща, описание, вес плода (средний).

Контрольные вопросы:

1. Понятие «Класс».
2. Особенности классов.

Литература практического раздела

1. Дьюхарст Программирование на C++ / Дьюхарст, Старк Стефан , Кэти. – М.: ДиаСофт, 2015. – 272 с.

2. Мейерс, С. Эффективный и современный C++. 42 рекомендации по использованию C++11 и C++14 / С. Мейерс. – М.: Вильямс, 2015. – 304 с.

3. Секунов, Н.Ю. Самоучитель Visual C++ 6.0 / Н.Ю. Секунов. – М.: СПб: BHV, 2014. – 960 с.

4. Ашарина, И.В. Основы программирования на языках С и С++: Курс лекций для высших учебных заведений / И.В. Ашарина. – М.: Гор. линия-Телеком, 2018. – 208 с.

5. Дорогов, В.Г. Основы программирования на языке С: Учебное пособие / В.Г. Дорогов, Е.Г. Дорогова; Под общ. ред. проф. Л.Г. Гагарина. – М.: ИД ФОРУМ, НИЦ ИНФРА-М, 2017. – 224 с.

6. Страуструп, Б. Язык программирования С++: Специальное издание / Б. Страуструп; Пер. с англ. Н.Н. Мартынов. – М.: БИНОМ, 2017. – 1136 с.

7. Фридман, А.Л. Основы объектно-ориентированного программирования на языке Си++ / А.Л. Фридман. – М.: Гор. линия-Телеком, 2016. – 234 с.

8. Павловская, Т.А. С/ С++. Программирование на языке высокого уровня / Т.А. Павловская. – СПб.: Питер, 2011. – 461 с.

9. Шилдт, Г. С++: базовый курс / Г. Шилдт – М.: Издательский дом «Вильямс», 2008. – 624 с.

10. Васильев, А.Н. Самоучитель С++ с примерами и задачами/ А.Н. Васильев. – СПб.: Наука и Техника, 2010. – 480с.

Контроль знаний

(вопросы для подготовки к экзамену)

1. Типы данных языка С++
2. Переменные и константы С++: инициализация и объявление
3. Пространства имен в С++, их примеры
4. Массивы в С++, объявление массива
5. Динамическое выделение памяти
6. Перечисления в С++: понятие, объявление и присвоение значения
7. Приведение типов данных
8. Операторы GOTO, GONTINIE, BREAK
9. Доступ к символами внутри строки, получение количества символов строки
10. Работа со строками в С++. Конкатенация, Методы строки
11. Функции/даты времени в С++
12. Перегрузка функции в С++, шаблонная функция, статическая переменная
13. Работа с файлами в С++: шаги (этапы работы)
14. функции работы с файлами средствами С++
15. Понятия «Класс» и «Объект», Понятие «Член класса-данное», Понятие «Член класса-функция» в С++
16. Объявление класса. Реализация класса
17. Комментарии в программе. Виды
18. Функции ввода/вывода данных в С++
19. Макрос в С++
20. Спецификаторы хранения переменных в С++, их описание
21. Строки, объявление и наполнение строк
22. Указатели в С++
23. Структуры в С++: понятие, объявление и присвоение значения

24. Объединения в C++: понятие, объявление и присвоение значения

25. Операторы C++: математические, присваивания, сравнения, логических выражений

26. Операторы ветвления, выбора и организации циклов в C++

27. Класс string: объявление без инициализации, объявление с инициализацией, присвоение значения после объявления, вывод и ввод строк в C++

28. Пользовательские функции: понятие, объявление и определение в C++

29. Способы чтения из файла, режимы открытия файлов, виды файлов в C++

30. Функции работы с файлами средствами C

31. Методы в классах. Спецификаторы доступа в классах в C++

32. Соглашения по выбору идентификаторов классов в C++

33. Конструкторы объектов класса. Деструктор объектов класса в C++

34. Качественный код, критерии качества

35. Консольное приложение на C#: структура модуля, достоинства и недостатки консольного приложения

36. Вектор (vector), синтаксис работы с векторами

37. Основные методы вектора

38. Итераторы, категории итераторов. Описание категорий итераторов.

Глоссарий

asctime – возвращает указатель на строку, которая содержит информацию о дате и времени

Base Class Library (BCL) – общая для всех языков программирования платформы Microsoft .NET Framework библиотека классов, в состав которой входят компоненты работы с различными структурами данных, файловыми структурами, базами данных, ресурсами в интернете и т.д.

break – используется внутри циклов и оператора switch, чтобы перейти в конец блока кода

C++ – объектно-ориентированный язык программирования, разработанный Бьерном Страуструпом.

char – символьный тип для определения одного символа

clear() – удалить все элементы вектора

clock – возвращает значение, которое приблизительно соответствует продолжительности времени работы вызывающей программы

continue – оставшийся код внутри цикла, но после continue, должен быть пропущен, потом переходит к началу цикла и это повторяется до тех пор, пока цикл не закончится.

ctime – озвращает указатель на строку, которая содержит информацию о дате и времени

delete – используется для освобождения памяти.

difftime – возвращает значение разности в секундах между значениями заданных параметров

DLL, Dynamic Link Library – динамически подключаемая библиотека

do оператор while (выражение) – в начале выполняется оператор, затем вычисляется значение выражения. Если оно истинно, оператор тела цикла выполняется еще раз.

double – вещественное число двойной точности с плавающей точкой;

empty() – проверить вектор на пустоту

Extern – сообщает компилятору, что переменная объявлена в другом месте

Float – вещественное число одинарной точности с плавающей точкой;

gmtime – возвращает указатель на поэлементную форму параметра time в виде структуры

goto – это оператор управления потоком выполнения программ, который заставляет центральный процессор выполнить переход из одного участка кода в другой (осуществить прыжок). Другой участок кода идентифицируется с помощью лейбла

int – целочисленный тип, целое число;

localeconv – возвращает указатель на структуру типа, которая содержит различную информацию о геополитической среде

localtime – возвращает указатель на поэлементную форму параметра time в виде структуры tm

Microsoft .NET Framework – одна из последних программных технологий компании Microsoft, созданная для разработки платформонезависимых приложений, исполняемых виртуальной машиной CLR.

Microsoft Visual Studio – интегрированная среда разработки программных продуктов компании Microsoft, которая, в том числе, поддерживает языки программирования для платформы Microsoft .NET Framework.

mktime – возвращает эквивалент календарного времени на основе времени, заданного в поэлементном виде и хранимого в структуре, которая адресуется параметром

new – используется для выделения памяти

pop_back() – удалить последний элемент вектора

sizeof() – позволяет определить сколько памяти занимает та или иная переменная

strcat – функция библиотеки <cstring>б, позволяет объединять две строки в одну.

strftime – помещает информацию о времени и дате(вместе с другой информацией) в строку

time – возвращает текущее календарное время системы

time setlocale – позволяет запросить или установить определенные параметры, которые зависят от геополитической среды выполнения программы

void – тип без значения

Абстрактный класс – класс, который невозможно непосредственно использовать для создания объектов; часто используется для определения ин-

Абстракция – описание сущности, которая вольно или невольно игнорирует (скрывает) детали (например, детали реализации); селективное незнание.

Адрес – значение, позволяющее найти объект в памяти компьютера

Алгоритм – порядок действий, которые необходимо выполнить для решения определенной задачи. В программировании алгоритмы описывают средствами псевдокода, блок-схем и UML диаграмм.

Аргумент – значение, передаваемое функции или шаблону, в которых доступ осуществляется через параметр

Баг – ошибка в программе/коде, из-за которой результаты выполнения программы неправильные.

Базовый класс – класс, используемый как база иерархии классов.

Библиотека – совокупность типов, функций, классов и т.п., реализованных в виде набора средств (абстракций), которые могут использовать многие программы.

Буфер – рабочая область памяти при пересылке данных

Вектор в C++ – это замена стандартному динамическому массиву, память для которого выделяется вручную, с помощью оператора new.

Виртуальная функция – функция-член, которую можно заместить в производном классе.

Время жизни – время, прошедшее между моментом инициализации и моментом, в который объект становится неиспользуемым (выходя из области видимости, уничтожается или прекращает существовать из-за прекращения работы программы)

Двунаправленные итераторы – способны перемещаться в обоих направлениях благодаря инкременту (++) и декременту (--)

Деструктор – – так же особый метод класса, который срабатывает во время уничтожения объектов класса

Динамическое выделение памяти – способ выделения оперативной памяти компьютера для объектов в программе, при котором выделение памяти под объект осуществляется во время выполнения программы.

Идентификатор – литерная цепочка, выступающая в определенном контексте в роли символа

Инициализация – – приведение областей памяти в состояние, исходное для последующей обработки или размещения данных

Инициирование – создание условий для запуска процесса обработки данных

Инкапсуляция – механизм, который объединяет данные и код, манипулирующий зтими данными, а также защищает и то, и другое от внешнего вмешательства или неправильного использования.

Интерпретатор – программа или техническое средство, выполняющее интерпретацию

Интерпретация – реализация смысла некоторого синтаксически законченного текста, представленного на конкретном языке

Интерфейс – объявление или набор объявлений, определяющих способ вызова фрагмента кода (например, функции или класса).

Итератор ввода – это такой итератор, который перемещается только вперед и поддерживает только чтение

Итераторы произвольного доступа – это такие итераторы, которые не только дают легко обращаться к произвольному элементу. Несмотря на то, что мы изменяем итератор переприсваиванием в него значения, на контейнер это никак не влияет.

Классы в C++ – это абстракция, описывающая методы, свойства, ещё не существующих объектов.

Комментарий – это строка (или несколько строк) текста, которая вставляется в исходный код для объяснения того, что делает этот код. В C++ есть два типа комментариев: однострочные и многострочные.

Компилятор – программа или техническое средство, выполняющее компиляцию

Компиляция – трансляция программы с языка высокого уровня в форму, близкую к программе на машинном языке

Компонент – составная часть распределенного приложения

Константы – их значение устанавливается один раз и впоследствии мы его не можем изменить.

Конструктор – – это особый метод класса, который выполняется автоматически в момент создания объекта класса

Контейнер в программировании – структура, позволяющая инкапсулировать в себя объекты разных типов.

Литерал – обозначение, которое непосредственно задает число

Макросы – это препроцессорные "функции", т.е. лексемы, созданные с помощью директивы #define, которые принимают параметры подобно функциям.

Массив – "представляет набор однотипных данных. Формальное определение массива выглядит следующим образом: тип_переменной название_массива [длина_массива]"

Методы – это функции, которые могут выполнять какие-либо действия над данными (свойствами) класса

Многострочные комментарии – это комментарии, которые пишутся между символами /* */. Всё, что находится между звёздочками – игнорируется компилятором

Наследование – это процесс, посредством которого один объект может приобретать свойства другого

Область памяти – память, выделенная для размещения одной или нескольких порций данных

Объединение – область памяти, используемая для хранения данных разных типов.

Объект – 1) Инициализированная область памяти известного типа, в которой записано какое-то значение данного типа; 2) область памятиж; 3) конкретное представление абстракции, имеющее свои свойства и методы.

Объектно-ориентированное программирование (ООП) – суть объектно-ориентированного программирования в представлении обрабатываемой информации в виде объектов – экземпляров классов.

Однонаправленный итератор – это итератор, который поддерживает чтение и запись адресуемого элемента.

Однострочные комментарии – это комментарии, которые пишутся после символов //. Они размещаются в отдельных строках и всё, что находится после этих символов комментирования – игнорируется компилятором

Оператор for – здесь начальные_присваивания — оператор или группа операторов, применяемые для присвоения начальных значений величинам используемым в цикле; выражение — определяет условие выполнения цикла, если оно истинно, то цикл выполняется; приращение — оператор, который выполняется после каждого шага (прохода) по телу цикла; оператор — любой оператор.

Оператор if – служит для того, чтобы выполнить какую-либо операцию в том случае, когда условие является верным. Внутри фигурных скобок указывается тело условия. Если

условие выполнится, то начнется выполнение всех команд, которые находятся между фигурными скобками.

Оператор выбора – оператор множественного выбора switch где осуществляется переход к той ветви программы, для которой значение переменной или выражения совпадает с указанным константным выражением. Далее выполняется оператор или группа операторов пока не встретится зарезервированное слово break или закрывающая фигурная скобочка

Оператор с предусловием (while) – вычисляется значение выражения. Если оно истинно, то выполняется оператор. В противном случае цикл заканчивается. Если состоит более чем из одного оператора, необходимо использовать составной оператор.

Операция – нечто, выполняющее какое-то действие, например функция или оператор

Отладка – поиск и удаление ошибок из программы; обычно имеет менее систематичный характер, чем тестирование

очистка и оптимизация кода – сюда можно отнести лишние импорты, переменные и методы которые уже не используются, но по какой-то причине их оставили в наследство.

Перегрузка функции – это возможность использовать названия для нескольких ф-ий с различными типами параметров и их количеством.

Переменная – это «ячейка» оперативной памяти компьютера, в которой может храниться какая-либо информация. Имя переменной содержит только лат. буквы, цифры, _, и начинается с буквы.

Перечисления (enum) – содержат набор числовых констант

Полиморфизм – это свойство, которое позволяет одно и то же имя использовать для решения двух или более схожих, но технически разных задач.

Правила поддержки кода – правила, которые должны сигнализировать что код слишком сложный и его будет трудно сопровождать.

Прикладное программирование – процесс разработки программного обеспечения, предназначенного для решения прикладных задач в определенной сфере деятельности.

Программа – данные, предназначенные для управления конкретными компонентами системы обработки информации в целях реализации определенного алгоритма

Программное обеспечение – совокупность программ системы обработки информации и программных документов, необходимых для эксплуатации этих программ

Проект – общее описание того, как должно работать программное обеспечение, чтобы соответствовать своей спецификации.

Производный класс – класс, являющийся наследником одного или нескольких базовых классов

Пространство имён – определяет область кода, в которой гарантируется уникальность всех идентификаторов. Пример: namespace Boo{}

Реализация класса – это способ осуществления работоспособности класса

Регулярные выражения – язык шаблонов или язык масок для поиска в тексте фрагментов, удовлетворяющих определенному набору критериев, разбиения найденных фрагментов на группы с целью дальнейшего их анализа и обработки.

Рекурсия – вызов функции самой себя

Синтаксические правила – к ним можно отнести стиль именования переменных, констант (uppercase), методов, стиль написания фигурных скобок и нужны ли они если в блоке только одна строка кода.

Спецификация программы – формализованное представление требований, предъявляемых к программе,

которые должны быть удовлетворены при ее разработке, а также описание задачи, условия и эффекта действии без указания способа его достижения ссылка – 1) Значение, описывающее место в памяти значения, имеющего тип; 2) переменная, содержащая такое значение.

Статические переменные – являются долговременными переменными, существующими на протяжении функции или файла.

Строки – это массив символов, последний эл-т которого содержит нулевой символ

Структура – это совокупность элементов, объединённых одним именем

Структурное программирование – метод построения программ, использующий только иерархически вложенные конструкции, каждая из которых имеет единственную точку входа и единственную точку выхода

Тип – это любой поддерживаемый тип данных

Транслятор – программа или техническое средство, выполняющие трансляцию программы

Указатели – представляют собой объекты, значением которых служат адреса других объектов (переменных, констант, указателей) или функций. Для определения указателя надо указать тип объекта, на который указывает указатель, и символ звездочки *.

Указатель области памяти – адрес области памяти, размещенный в пространстве памяти, в котором расположена эта область

Файл – Контейнер, содержащий информацию в постоянной памяти компьютера.

Фреймворк – вид программного обеспечения, являющегося основой (каркасом) различных прикладных программных продуктов.

Функция – Именованная единица кода, которую можно активизировать (вызвать) из разных частей программы; логическая единица вычислений.

Функция fwrite – записывает блок данных в поток

Функция fclose – принимает один аргумент: указатель на структуру FILE потока для закрытия.

Цикл – последовательность команд в программе, которая должна исполняться неоднократно в результате перехода от начала последовательности к концу

Член класса (данное) – именованная характеристика или свойство объектов класса, которое имеет собственное значение для каждого объекта

Член класса (функция) – это функция, принадлежащая классу, при помощи которой объект класса взаимодействует с объектами других классов или того же самого класса.

Шаблоны функций – это инструкции, согласно которым создаются локальные версии шаблонированной функции для определенного набора параметров и типов данных

Язык высокого уровня – язык программирования, понятия и структура которого удобны для восприятия человеком

Язык С – основе своей был создан в 1972 г. как язык для операционной системы UNIX. Его автором считается Денис М. Ритчи (Dennis M. Ritchie).

Литература

1. Павловская, Т.А. C/C++. Процедурное и объектно-ориентированное программирование: учебник для студентов вузов, обучающихся по направлению подготовки дипломированных специалистов «Информатика и вычислительная техника»: [для бакалавров и специалистов] / Т.А. Павловская. – Санкт-Петербург, Санкт-Петербург [и др.]: Питер, 2018. – 495 с.

2. Лафоре, Р.Объектно-ориентированное программирование в C++: пер. с англ. / Роберт Лафоре; пер. А. Кузнецов, М. Назаров, В. Шрага. – 4-е изд. – Санкт-Петербург [и др.]: Питер, 2017. – 923 с.

3. Стандарт C++: перевод, комментарии, примеры: пер. с англ. / ред. перевода Е.А. Зуев, А.А. Чупринов. – Москва: ВАШ ФОРМАТ, 2016. – 886 с.

4. Долгов, В.В. Основы программирования на языке C++: учебное пособие / В.В. Долгов, А.А. Скляренко, Н.Н. Венцов; кол. авт. Донской государственный технический университет. – Ростов-на-Дону: ДГТУ, 2016. – 171 с.

5. Навроцкий, А.А. Основы алгоритмизации и программирования в среде Visual C++: учебно-методическое пособие / А.А. Навроцкий; кол. авт. Белорусский государственный университет информатики и радиоэлектроники. – Минск: БГУИР, 2014. – 159 с.