Раздел 1
Базовые понятия языка С++

1.1. Введение в С++

В настоящее время С++ является языком, наиболее полно представляющим основные парадигмы современного программирования. С++ сочетает в себе три различных принципа программирования:

− процедурное программирование, представленное языком С и позволяющее создавать библиотеки функций;

− объектно-ориентированное программирование (ООП), представленное таким понятием как класс и позволяющее разрабатывать библиотеки классов;

− и обобщенное программирование, представленное шаблонами языка С++ и его стандартными библиотеками.

Язык С++ - замечательный язык, обладающий такими свойствами как эффективность, компактность, быстрота выполнения и переносимость программ. А три вышеперечисленные методики программирования С++ делают язык очень мощным, постоянно развивающимся в общем русле эволюции средств программирования и информатики.

Тот факт, что С++ включает в себя как ООП, так и обобщенное программирование наряду с традиционным процедурным подходом, свидетельствует о том, что в С++ упор делается на практичность, а не на идеологию. В этом и состоит одна из причин успеха данного языка. Вторая причина успеха – это двойственность характера языка С++. Элементы языка С обеспечивают доступ к аппаратным средствам на низком уровне. А ООП и обобщенное программирование обеспечивает высокий уровень абстракции данных. Именно сочетанием этих свойств обусловлена популярность С++.

История создания языка

В начале 70-х годов Денис Ритчи, сотрудник компании Bell Laboratories, занимался разработкой операционной системы UNIX. Операционная система – это совокупность программ, управляющих ресурсами компьютера и его взаимодействием с пользователем. Операционная система выводит на экран системные сообщения и управляет выполнением программ. Д. Ритчи нуждался в языке программирования, который мог обеспечить эффективное управление аппаратными средствами и создание компактных, быстро работающих программ. Такие потребности мог удовлетворить язык ассемблера, который тесно связан с внутренним машинным языком компьютера. Однако ассемблер – язык *низкого уровня*, то есть, привязан к определенному типу процессора. Операционная система UNIX предназначалась для работы на компьютерах различных типов (или платформах). А это предполагало использование языка высокого уровня. Язык *высокого уровня* ориентирован на решение задач, а не на конкретное аппаратное обеспечение. Специальные программы, которые называются *компиляторами*, транслируют программу, написанную на языке высокого уровня, в команды внутреннего языка конкретного компьютера.

Таким образом, используя, отдельный компилятор для каждой платформы, одну и ту же программу на языке высокого уровня можно выполнять на разных платформах. Нужен был язык, который сочетал бы в себе эффективность и возможность доступа к аппаратным средствам, обеспечиваемую языками низкого уровня, с более общим характером и переносимостью, присущими языкам высокого уровня. На базе имеющихся языков программирования Ритчи разработал язык С.

Язык С++ был создан в начале 80 годов также в компании Bell Laboratories Бьерном Страуструпом с целью повышения эффективности языка программирования. Страуструп выбрал в качестве основы язык С, так как он был кратким, хорошо подходил для системного программирования и был широко доступен. Страуструп добавил в язык С элементы ООП и обобщенного программирования, не меняя при этом существенно сам язык С. Таким образом, язык С++ был разработан как расширение языка С.

Язык С расширяется введением гибких и эффективных средств, предназначенных для построения новых типов. Программист структурирует свою задачу, определив новые типы, которые точно соответствуют предметной области задачи.

Такой метод построения программы обычно называют абстракцией данных. Ключевым понятием С++ является класс. Класс - это определяемый пользователем тип. Классы обеспечивают упрятывание данных, их инициализацию, неявное преобразование пользовательских типов, динамическое задание типов, контролируемое пользователем управление памятью и средства для перегрузки операций. Кроме того, С++ содержит усовершенствования, прямо с классами не связанные: символьные константы, функции-подстановки, стандартные значения параметров функций, перегрузка имен функций, операции управления свободной памятью и ссылочный тип. В С++ сохранены также все возможности языка С эффективной работы с основными объектами, отражающими аппаратную "реальность" (разряды, байты, слова, адреса и т.д.).

Различие между процедурными языками и объектно-ориентированными.

В общем случае язык программирования базируется на двух основных понятиях – это данные и алгоритмы. Данные представляют собой информацию, которую программа обрабатывает. А алгоритмы – это методы, которые программа использует для обработки данных. Язык С как и большинство языков программирования того времени является *процедурным* – это означает, что основной аспект в нем делается на алгоритмах.

Процедурное программирование заключается в том, что сначала определяется последовательность действий, которая должна быть выполнена компьютером, а затем эти действия реализуются с помощью языка программирования. Программа содержит набор процедур, которые компьютер должен выполнить, чтобы получить требуемый результат.

Первые процедурные языки, такие как FORTRAN и BASIC, столкнулись с проблемами организационного плана. Во многих старых программах благодаря операторам безусловного перехода алгоритм настолько запутан (их называют программы – «спагетти»), что его трудно читать. Для решения этой проблемы был разработан стиль программирования, который называется *структурным программированием.*

- Во-первых, структурное программирование ограничивает ветвление программы, в основном отказываясь от оператора безусловного перехода. Алгоритм строится, используя несколько основных структур: структура последовательного выполнения, структуры повторения с предусловием и с постусловием, структура - альтернатива и структура - ветвление.

Язык С поддерживает эти конструкции (операторы-выражения; циклы **for, while, do while**; оператор **if else** и оператор **switch**).

- Во-вторых, еще одним новым принципом структурного программирования является проектирование программы *сверху вниз.*

Идея заключается в разбиении большой программы на более мелкие, легче решаемые задачи. При этом используются описанные выше структуры. Если полученные задачи по-прежнему остаются слишком обширными, их также следует разделить на более мелкие задачи, используя для алгоритма все те же структурные схемы. Этот процесс продолжается до тех пор, пока программа не будет разделена на маленькие, легко программируемые модули. Разработанные таким образом (*сверху вниз)* алгоритмы обладают в некотором смысле свойством "правильности", встроенной в них шаг за шагом. Общая структура управления в структурированной программе является *деревом.* Такую программу легко читать сверху вниз, а, не прыгая по тексту из конца в начало. Методика структурного программирования отражает процедурный подход, при котором программа рассматривается с точки зрения выполняемых ею действий.

Хотя принципы структурного программирования позволили улучшить понятность и надежность программ, а также облегчить их сопровождение, создание программ больших размеров по-прежнему оставалось нелегкой задачей.

Объектно-ориентированное программирование (ООП) предлагает новый подход к решению этой задачи. В отличие от процедурного программирования, где главное внимание уделяется алгоритмам, в ООП основной акцент делается на данные. Идея заключается в создании таких форм данных, которые соответствовали специфике поставленной задачи.

Спецификацией, описывающей подобную уникальную форму данных в языке С++, является *класс,* а конкретной структурой данных, созданной в соответствии с этой спецификацией, - объект. В общем случае *класс* определяет, какие данные будут представлять объект и какие операции могут выполняться над этими данными. Используя объекты классов, можно приступать к разработке самой программы. Такой процесс продвижения от

более низкого уровня организации (классы) к более высокому уровню (программа), называется программированием *снизу вверх.*

Объектно-ориентированное программирование – это объединение данных и методов в описании класса. Сокрытие - *инкапсуляция* данных позволяет предохранить данные от нежелательного доступа. *Полиморфизм* дает возможность создавать множественные определения для операций и функций, а то, какое определение будет конкретно использоваться, зависит от контекста программы. *Наследование* позволяет создавать новые классы из старых и наследовать данные и методы от классов – родителей.

Таким образом, в ООП используется иной подход к созданию программ в сравнении с процедурным программированием. Основное внимание уделяется не алгоритмическому аспекту задач, а созданию новых нужных форм данных, представляющих абстракции общих понятий. Кроме того, объектно-ориентированные языки дают возможность включать в программы уже существующие библиотеки классов.

Понятие обобщенного программирования

Обобщенное программирование – это еще одна парадигма программирования, поддерживаемая языком С++. Назначение обобщенного программирования такое же как ООП – упростить повторное использование кодов программ и методов абстрагирования общих понятий.

Однако в то время как в ООП основное внимание уделяется данным, в обобщенном программировании упор делается на шаблоны алгоритмов и у него другая область применения. ООП – это инструмент для разработки больших программ, тогда как обобщенное программирование обеспечивает выполнение задач общего характера, таких как, например, сортировка данных или поиск. Обобщенное программирование – это создание кода программы независимого от типа данных. Б. Страуструп [1, с.377] определяет обобщенное программирование как "программирование с использованием типов в качестве параметров".

В С++ имеются данные различных типов – целые числа, вещественные, символы, строки символов. Кроме того типы определенные пользователем, например, сложные структуры, состоящие из данных нескольких типов. Если требуется, например, сортировать данные различных типов, то обычно требуется создавать отдельную функцию сортировки для каждого типа. Обобщенное программирование расширяет язык таким образом, что можно один раз написать функцию для обобщенного (то есть неопределенного) типа данных, а затем использовать ее для разнообразных реальных типов данных. Однако это можно обеспечить и с помощью шаблонов языка С++.

Шаблоны являются инструментами обобщенного программирования, но оно идет дальше по пути обобщения. Разнотипные данные могут быть объединены в обычный массив или составлять связанный список или быть элементами любого другого типа контейнеров. Цель обобщенного программирования - создать одну функцию, в данном случае - сортировки,

которая работала бы с массивами, связанными списками, или любым другим типом контейнеров. Иначе говоря, функция должна быть независимой не только от типов данных, содержащихся в контейнере, но и от самой структуры контейнера. Шаблоны обеспечивают обобщенное представление данных, сохраняемых в контейнере. В случае сортировки все, что нужно – это обобщенное представление процесса произвольного доступа, чтобы иметь возможность выполнения операции обмена между двумя не соседними элементами в контейнере. Именно такое обобщенное представление в *обобщенном программировании* реализуется с помощью итераторов. Назначение итератора – предоставить единый метод перебора элементов контейнера, не зависящий от вида контейнера и типа элементов в нем.

Международный стандарт языка С++.

Язык С++ с момента своего создания приобрел большую популярность у программистов и стал широко распространенным языком. Язык достиг определенного уровня зрелости и претерпел значительные изменения.

В течение многих лет велись работы по выработке стандартов языков С и С++. В 1983 году Национальным институтом стандартизации США (American National Standards Institute –ANSI) был утвержден первый стандарт языка С – ANSI C, который определил не только сам язык С, но и стандартную библиотеку С, которая должна быть включена во все реализации языка С.

В языке С++ тоже используется эта библиотека, обычно называемая стандартной библиотекой С или просто стандартной библиотекой. В дополнении к этому стандарт языка С++ должен представлять стандартную библиотеку классов языка С++.

Работа ANSI и Международной организацией по вопросам стандартизации (International Standards Organization – ISO) над стандартом языка С++ началась в 1990 году. В 1998 году был утвержден стандарт ANSI/ISO C++, который согласуется со стандартом ANSI C .

Новый стандарт С был принят ISO и ANSI в 1999 году. Эта версия называется С99. Она включает ряд усовершенствований и несколько новых средств.

Затем в течение многих лет велись работы по выработке новых стандартов языков С и С++, которые завершились в конце 2011г.

В октябре 2011г. вышел новый стандарт языка С++, обозначаемый как С++11. А в декабре 2011г. был принят новый стандарт и для языка С. Эту версию условно называют С11. Новые стандарты уже сейчас поддерживается свободно распространенным компилятором DJGPP.

Компиляторы языка С++ разработаны практически для всех аппаратно-программных платформ. Программы на С++ транслируются в исполняемые модули, работающие под управлением операционных систем UNIX (и ее разновидностей, таких как, LINUX, Solaris), Windows, Mac OS. В отличие от языков C# (Си-Шарп), Java и Visual Basic язык С++ позволяет создавать программы, для выполнения которых не требуется устанавливать на компьютер

специальное программное обеспечение, создающее среду исполнения программ.

Настоящее учебное пособие в основном будет ориентироваться на версию Borland C++ 3.11.

Структура языка С++.

Обобщенная структура языка С++ дана на рис. 1.

Рис. 1. Обобщенная структура языка С++

В левой части рисунка представлены средства языка, предназначенные для определения данных, объектов обработки программы. Типы данных определяют свойства данных, их внутреннее представление, возможные операции, которые можно производить с этими данными.

В следующей части рисунка представлены средства языка - операторы, предназначенные:

во-первых, для обработки данных путем, например, получения новых значений объектов программы в операторе присваивания;

во-вторых, для организации процесса обработки данных, например, организации повторяющейся обработки или организация разветвления процесса обработки.

Далее на рисунке представлены модули – относительно самостоятельные фрагменты программы для функционально законченной обработки данных, оформленные в виде функций.

В следующей части рисунка представлены средства и механизмы объектно-ориентированного программирования. Представлен новый тип данных, объединяющий данные и функции их обрабатывающие в единое целое – объект.

В последней части рисунка описаны средства обобщенного программирования, а именно множество контейнеров – структур данных, в которые можно помещать и извлекать данные любых типов, и набора обобщенных алгоритмов, позволяющих выполнять типовые операции над элементами контейнеров, не зависящие от вида контейнера. Абстракцию данных и алгоритмов обеспечивают шаблоны и итераторы.

Методика создания программ

Процесс разработки программ на С++ предполагает разбиение процесса решения задачи на ряд этапов, выполняющих функционально законченную обработку данных и формирование соответствующих функций. В результате программа представляет собой совокупность функций, одна из которых главная, называемая ***main***. Главная функция может располагаться в любом месте программы, но где бы она не находилась выполнение программы начинается и заканчивается именно в главной функции. Для главной функции можно использовать только имя ***main*.**

Определение любой функции, в том числе и главной в С++, состоит из заголовка и тела функции:

<тип возвращаемого функцией результата> <имя> (список параметров)

{ тело функции – последовательность действий функции }

Приведем пример простой программы [3]:

```
#include <iostream.h>
int main ( )
{cout << "Программа стартовала"<<endl;
return 0;
}
```

В результате выполнения программы в консольном окне экрана выведется фраза: ***Программа стартовала.***

В первой строке – команда (директива) препроцессора, обеспечивающая включение в программу средств работы со стандартными потоками ввода/вывода данных. Эти средства подключаются к программе при использовании заголовочного файла с именем ***iostream.*** Стандартным потоком вывода по умолчанию является вывод на экран дисплея. Стандартный поток ввода обеспечивает чтение данных с клавиатуры.

Вторая строка – это заголовок функции ***main.*** В общем случае функция С++ вызывается другой функцией, а заголовок функции описывает интерфейс между ней и той функцией, которая ее вызывает. Слово, стоящее перед именем описывает информацию, которую функция передает в вызывающую функцию. Заголовок главной функции описывает интерфейс между функцией ***main( )*** и операционной системой.

Стандарт языка С++ требует, чтобы определение функции ***main*** начиналось со следующего заголовка: ***int main ( ) -*** слово ***int*** указывает, что функция ***main()*** возвращает целое значение.

Возвращаемое функцией ***main()*** значение должно быть равно нулю, если выполнение программы прошло успешно. Круглые скобки после ***main*** требуются в соответствии с синтаксисом заголовка любой функции. В них помещается необязательный для главной функции список параметров. В данном примере список пуст. Некоторые программисты используют следующий также допустимый заголовок: ***void main (),*** означающий, что функция не возвращает результата.

Тело функции – это заключенная в фигурные скобки последовательность описаний, определений и операторов функции. В теле данной программы описаний и определений нет, а есть только два оператора.

Первый из них: ***cout << "Программа стартовала"<<endl;***
где ***cout*** - имя стандартного выходного потока. Данные для вывода передаются потоку с помощью операции <<. То, что нужно вывести, помещается от операции << справа. В данном случае это строка – ***"Программа стартовала",*** заключенная в кавычки последовательность символов. Вслед за строкой помещается еще одна операция вывода **<<,** а затем манипулятор ***endl*** (сокращение от "end of line" – "конец строки"). Его роль - очистить буфер выходного потока и поместить в выходной поток символ перехода на новую строку.

Второй оператор в программе ***return 0*** – оператор возврата. Он завершает выполнение программы и передает в точку ее вызова значение выражения, стоящего в операторе. Так как программа "запускается" на исполнение по команде операционной системы, то возврат будет выполнен к операционной системе.

Структура программы

Простая программа на С++ состоит из следующих элементов:

1) препроцессорные директивы, например:
   # include <имя файла>// - включение текстов стандартных файлов
   #define …   // - замены в тексте
2) объявление глобальных объектов программы (типов, переменных, констант);
3) объявление одной главной функции ***main ()*;**
4) объявление ряда неглавных функций;
5) комментарии.

Этапы создания исполняемого кода программы

Рассмотрим технологию подготовки программ.

Классическая схема подготовки исполняемой программы [2] приведена на рис. 2.

Подготовка программы начинается с редактирования файла, содержащего текст программы, который имеет расширение ".cpp".

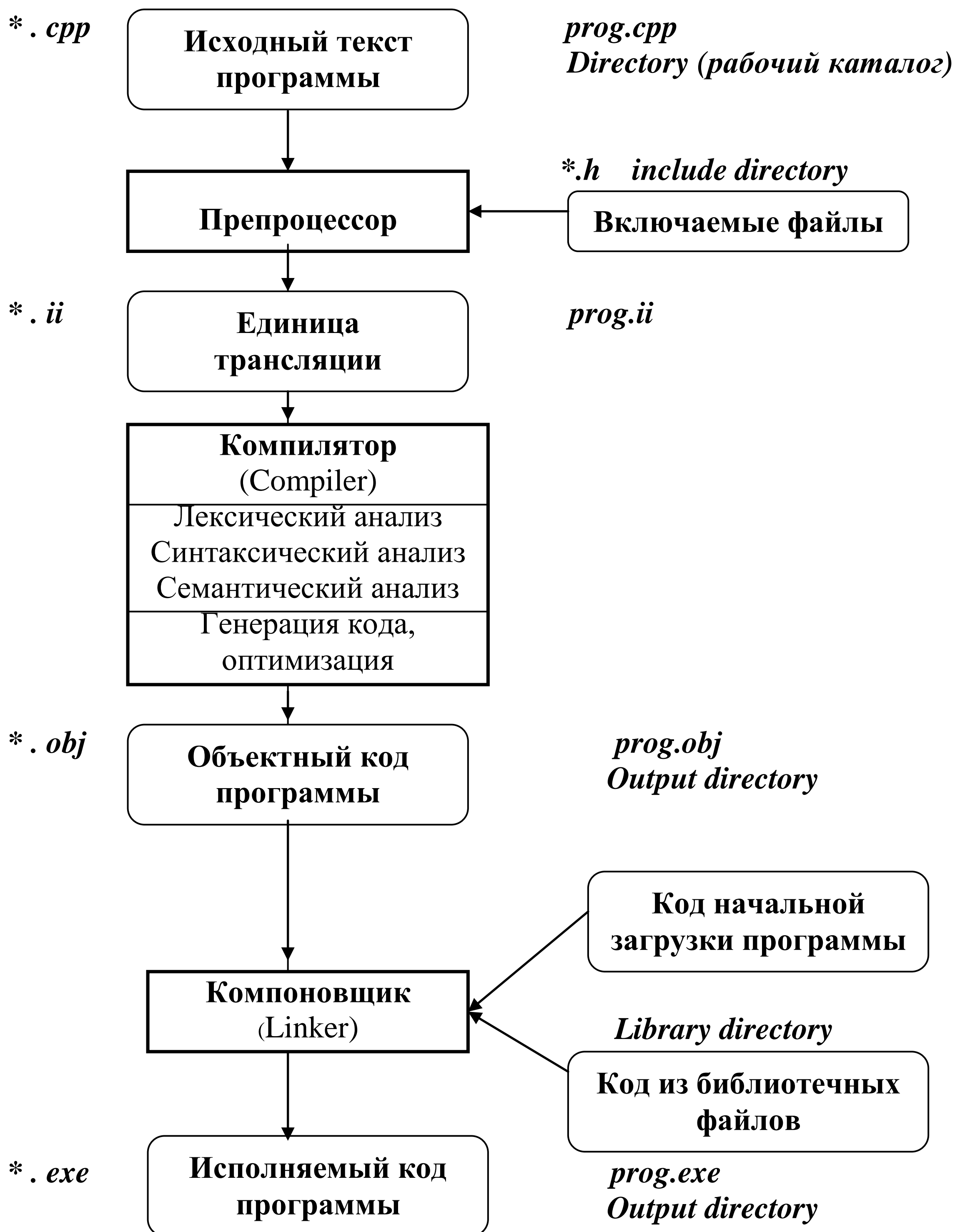

Рис. 2. Схема подготовки исполняемой программы

Перед шагом компиляции показан этап препроцессорной обработки текста программы. В нашем примере препроцессор обрабатывает директиву ***#include <iostream.h>*** и подключает к исходному тексту программы средства ввода/вывода. Результат препроцессорной подготовки при включении специальной опции компилятора помещают в файл с расширением ***".ii".***

Препроцессор сформирует полный текст программы – единицу трансляции (translation unit).

Затем выполняется компиляция программы, которая включает в себя несколько фаз: лексический, синтаксический, семантический анализ, генерация кода и его оптимизация. В результате компиляции получается объектный модуль - некий "полуфабрикат" готовой программы. Файл объектного модуля имеет стандартное расширение ".obj".

Компоновка (сборка) программы заключается в объединении одного или нескольких объектных модулей программы и объектных модулей, взятых из библиотечных файлов и содержащих стандартные функции. В результате получается исполняемая программа в виде отдельного файла (загрузочный модуль и программный файл) со стандартным расширением -".exe", который затем загружается в память и выполняется.

1.2. Лексические основы языка

Алфавит — это тот набор знаков (символов), который допустим в данном языке.

В алфавит языка С++ входят 96 символов.

- Из них 91 изображаемых символов:
  - прописные латинские буквы A..Z;
  - строчные латинские буквы a..z;
  - арабские цифры 0..9;
  - символ подчеркивания _ (рассматривается как буква).

Эти символы используются для образования ключевых слов и имён языка. В языке С++ прописные и строчные буквы различаются.

- В алфавит входят также 28 специальных символов:

, . ; : ? ' ! | / \ ~ * ( ) { } < > [ ] # % & ^ – = " +

- А также 5 – неизображаемых символов:
  - обобщенные пробельные символы (пробел, горизонтальная и вертикальная табуляция, перевод страницы, начало новой строки).

Комментарии

Комментарий — это последовательность любых знаков (символов) компьютера, которая используется в тексте программы для её пояснения. Обычно в тексте программы делают вводный комментарий к программе в целом (её назначение, автор, дата создание и т.д.), а далее дают комментарии к отдельным фрагментам текста программы, смысл которых не является очевидным. Компилятор языка программирования игнорирует комментарии, они нужны только для человека. В языке С++ имеется два вида комментариев: однострочные и многострочные.

Однострочный комментарий начинается с символов // (две косые черты). Всё, что записано после этих символов и до конца строки, считается комментарием. Например:

//Это текст однострочного комментария

Многострочный комментарий начинается парой символов /* (косая черта и звёздочка) и заканчивается символами */ (звёздочка и косая черта). Текст такого комментария может занимать одну или несколько строк. Всё, что находится между знаками /* и */, считается комментарием. Например:

/* *Это текст большого*
многострочного комментария */

В комментариях символы - это не только литеры из алфавита языка С++, но и любые возможные символы, включая русские буквы.

Лексемы языка

Из изображаемых символов алфавита формируются лексемы (tokens) языка. Лексемы - последовательности символов исходного кода программы, имеющие определенное смысловое значение.

В языке С++ имеются следующие категории лексем [1]:

- идентификаторы (identifier);
- ключевые (зарезервированные) слова (keyword);
- знаки операций (operator);
- константы – литералы (literal);
- разделители (знаки пунктуации – punctuator).

Рассмотрим эти лексические элементы языка подробнее.

Идентификаторы

Идентификатор — это имя программного объекта. В идентификаторе могут использоваться латинские буквы, цифры и знак подчеркивания. Прописные и строчные буквы различаются, например, sysop и SYSOP — два различных имени. Первым символом идентификатора может быть буква или знак подчеркивания, но не цифра. Пробелы внутри имен не допускаются. Длина идентификатора по Стандарту не ограничена, но некоторые компиляторы и компоновщики налагают на нее ограничения. Например, компиляторы фирмы Borland различают не более 32-х первых символов. При выборе идентификатора необходимо иметь в виду следующее:

- идентификатор не должен совпадать с ключевыми словами (см. следующий раздел) и с именами используемых стандартных объектов языка;
- не рекомендуется начинать идентификаторы с одного или двух символов подчеркивания, поскольку они могут совпасть с именами системных функций или переменных.

Ключевые (служебные) слова

Ключевые слова — это идентификаторы, зарезервированные в языке для специального использования. Эти идентификаторы имеют специальное значение для компилятора. Их можно использовать только в том смысле, в котором они определены. Список ключевых слов С++ [2] приведен в табл. 1.1.

Константы – литералы и перечисления

В языке С++ существует несколько видов констант: константы – литералы (неименованные константы), именованные константы, константы

перечислений и препроцессорные константы. Рассмотрим константы литералы – лексемы языка.

Таблица 1.1

Список ключевых слов С++

asm	else	new	this
auto	enum	operator	throw
bool	explicit	private	true
break	export	protected	try
case	extern	public	typedef
catch	false	register	typeid
char	float	reinterpret_cast	typename
class	for	return	union
const	friend	short	unsigned
const_cast	goto	signed	using
continue	if	sizeof	virtual
default	inline	static	void
delete	int	static_cast	volatile
do	long	struct	wchar_t
double	mutable	switch	while
dynamic_cast	namespace	template	

Константы – литералы и перечисления

В языке С++ существует несколько видов констант: константы – литералы (неименованные константы), именованные константы, константы перечислений и препроцессорные константы. Рассмотрим константы литералы – лексемы языка.

Константа – литерал – это лексема, представляющая изображение фиксированного числового, символьного или строкового значения. Константы – литералы делятся на пять групп [1]:

целые (integer literal);

вещественные (floating literal – с плавающей точкой);

логические (boolean literal - булевские);

символьные (character literal - литерные);

строковые (string literal - строки).

Компилятор, выделив константу в качестве лексемы, "по внешнему виду" относит ее к определенной группе, а внутри группы по форме записи и по числовому значению – к тому или иному типу данных.

Целые константы могут быть *десятичными, восьмеричными и шестнадцатеричными*.

Десятичная целая константа определена как последовательность десятичных цифр, начинающаяся не с нуля, если это не число нуль, например: 16, 897216, 0, 21. *Отрицательные константы* - это константы без знака, к которым применена операция изменения знака.

Восьмеричные целые константы начинаются всегда с нуля, например 016 имеет десятичное значение 14.

Шестнадцатеричная константа - это последовательность шестнадцатеричных цифр, которой предшествует 0x. В набор шестнадцатеричных цифр кроме десятичных входят латинские буквы от ***a*** (или ***A***) до ***f*** (или ***F***). Таким образом, 0x16 имеет десятичное значение 22, а 0xF - десятичное значение 15.

Диапазон допустимых целых положительных значений - от **0** до **4294967295**. Константы, превышающие указанное максимальное значение, вызывают ошибку на этапе компиляции. Абсолютные значения отрицательных констант не должны превышать **2147483648**.

В зависимости от значения целой константы, компилятор по-разному представляет её в памяти ЭВМ. О форме представления данных в памяти ЭВМ говорят, используя понятие *тип данных.*

Соответствие между значением целых констант и автоматически выбираемыми для них типами данных отражено в табл. 1.2 для компиляторов семейства IBM PC/XT/AT (16-разрядных компиляторов) [3].

Таблица 1.2

Целые константы и выбираемые для них типы
Диапазоны значений констант

Десятичные	Восьмеричные	Шестнадцатеричные	Тип данных
от 0 до 32767	от 00 до 077777	от 0x0000 до 0x7FFF	int
	от 0100000 до 0177777	от 0x8000 до 0xFFFF	unsigned int
от 32768 до 2147483647	от 0200000 до 017777777777	от 0x10000 до 0x7FFFFFFF	long
от 2147483647 до 4294967295	от 020000000000 до 037777777777	от 0x80000000 до 0xFFFFFFFF	unsigned long
> 4294967295	> 037777777777	> 0xFFFFFFFF	ошибка

В заголовке <limit.h> определены предельные значения целых величин различных типов. Младший тип целых констант – ***int*** имеет диапазон допустимых значений от – **32767** до **32767 .**

Для 32-разрядных компиляторов диапазон значений типа ***int*** обычно равен -**2147483647** и **2147483647.** Для типа ***unsigned int*** ("беззнаковый целый") минимальное значение равно 0, а максимальное - **4294967295.** Предельные значения целых констант даны в заголовке <climit.h>. И в реализациях 32-

разрядных компиляторов, в которых не различаются типы ***int*** **-** "целое" и ***long*** **-** "длинное целое" при использовании целых чисел, превышающих значение **4294967295**, возникает ошибка. Если программиста по каким-либо причинам не устаревает тот тип, который компилятор приписывает константе, то он может явным образом изменить тип. Можно явно указать тип, используя суффиксы ***L, l (long)*** и ***U, u (unsigned)***. Например, константа ***64L*** будет иметь тип ***long***, хотя значению 64 должен быть приписан тип ***int***, как это видно из табл. 1.2. Для одной константы можно использовать два суффикса в произвольном порядке. Например, константы ***0x22ul, 0x33Lu*** будут иметь тип ***unsigned long***.

Перечислимые константы. Стандарт относит к целочисленным константам и перечислимые. Перечислимые константы (или константы перечисления) определяются с помощью служебного слова ***enum***. Это целочисленные константы, которым даются уникальные имена по следующему правилу:

enum { список идентификаторов констант = значения констант }

Имена констант уникальны, но значения могут повторяться. Кроме того, значения в определении можно опускать. Константы автоматически получат значения, которые будут начинаться от нуля, а затем увеличиваться на 1 для следующей константы. Возможны определения, в которых присутствуют определения констант со значением и без значения. В этом случае константы без значения автоматически получают значения. Правило о последовательном увеличении на 1 значения констант действует и в этом случае. Например, определение:

enum { a, b =0, c, d, e=2, f };

вводит следующие константы:

a = =0, b = =0, c= =1, d = =2, e = =2, f = =3.

Вещественные константы

Для представления вещественных чисел используются константы, представляемые в памяти компьютера в форме с плавающей точкой. Даже не отличаясь от целой константы по значению, вещественные константы имеют другую форму внутреннего представления в ЭВМ. При операциях с такими константами требуется использование арифметики с плавающей точкой. Компилятор распознает вещественную константу по внешним признакам.

Вещественная константа может включать следующие шесть частей: целая часть (десятичная целая константа); десятичная точка; дробная часть (десятичная целая константа); признак экспоненты "**e**" или "**E**"; показатель десятичной степени (десятичная целая константа, возможно, со знаком); суффикс ***F*** (или ***f***), либо ***L*** (или ***l***).

При записи констант с плавающей точкой могут опускаться целая или дробная часть (но не одновременно); десятичная точка или символ экспоненты с показателем степени (но не одновременно). Примеры констант с плавающей точкой:

125.   .0   .17   3.141F   1.2e-5   .314159E25   2.77   2E+6L

Вещественная константа в экспоненциальном формате представляется в виде мантиссы и порядка. Мантисса записывается слева от знака экспоненты (E или e), порядок — справа от знака. Значение константы определяется как произведение мантиссы и возведенного в указанную в порядке степень числа 10. Обратите внимание, что пробелы внутри числа не допускаются, а для отделения целой части от дробной используется не запятая, а точка.

При отсутствии суффиксов вещественное число имеет форму внутреннего представления, которой соответствует тип данных ***double***. Добавив суффикс ***F*** или ***f***, константе придают тип ***float*** и соответственно тип ***long double,*** если в ее конце суффиксы ***L*** или ***l***. В табл. 1.3 приведены диапазоны возможных значений и длины внутреннего представления (размеры в битах) данных вещественного типа в конкретной реализации C++ (компилятор DJGPP).

Таблица 1.3

Данные вещественного типа

Тип данных	Размер в битах	Диапазон значений
float	32	от 3.4E-38 до 3.4E+38
double	64	от 1.7E-308 до 1.7E+308
long double	96	от 3.4E-4932 до 1.1E+4932

Булевские (логические) константы

Это два литерала типа ***bool: true*** (ИСТИНА) и ***false*** (ЛОЖЬ). Тип ***bool*** и литералы ***true*** и ***false*** были добавлены в последних версиях Стандарта языка. Наряду с логическими литералами продолжают действовать правила, унаследованные из ранних версий языка, в соответствии с которыми значению ЛОЖЬ соответствует число 0, а любое отличное от нуля значение воспринимается в логическом выражении как ИСТИНА.

Символьные (литерные) константы

Символьные константы – это один или два символа, заключенные в апострофы. Символьные константы, состоящие из одного символа, занимают в памяти один байт и имеют стандартный тип ***char***. Примеры: **'z', '*', '\012' , '\0', '\n'** – односимвольные константы. Двухсимвольные константы занимают два байта и имеют тип ***int,*** при этом первый символ размещается в байте с меньшим адресом (о типах данных рассказывается в следующем разделе) **'ab'**, **'\x07\x07', '\n\t'** – двухсимвольные константы.

Внутри апострофов может быть любой символ, имеющий изображение. Однако в ПК есть символы, не имеющие графического изображения. Это, как правило, управляющие символы, например символ перехода на новую строчку. Для изображения таких символов используется комбинация из нескольких символов, начинающаяся с обратной косой черты.

Последовательности символов, начинающиеся с обратной косой черты, называются управляющими эскейп - последовательностями (escape-sequence).

Эскейп - последовательности используются:

- для записи символов, не имеющих графического изображения (например, \a — звуковой сигнал);
- для записи символов**:** символа апострофа (**'**), обратной косой черты **(** \ ), знака вопроса (**?**) и кавычки (**"**);
- для записи любого символа с помощью его шестнадцатеричного или восьмеричного кода, например, \073, \0xF5. Числовое значение кода должно находиться в диапазоне от 0 до 255.

В табл. 1.4 приведены допустимые значения эскейп-последовательностей. Управляющая последовательность интерпретируется как одиночный символ. В таблице **000** – строка от одной до трех восьмеричных цифр, **hh** – строка из одной или двух шестнадцатеричных цифр. Последовательность **'\0'** обозначает пустую литеру.

Таблица 1.4

Допустимые ESC–последовательности в языке C++

Изображение	Внутренний код	Обозначаемый символ	Действие или смысл
'\a'	**0x07**	**bel (**audible bell)	звуковой сигнал
'\b'	**0x08**	**bs** (backspace)	возврат на шаг(забой)
'\f '	**0x0C**	**ff** (form feed)	перевод страницы
'\n'	**0x0A**	**lf** (line feed)	перевод строки (LF)
'\r'	**0x0D**	**cr** (carriage return)	возврат каретки(CR)
'\t	**0x09**	**ht** (horizontal tab)	горизонтальная табуляция
'\v'	**0x0B**	**vt** (vertical tab)	вертикальная табуляция
'\\'	**0x5C**	\ (backslash**)**	обратная черта
'\''	**0x27**	**'** (single quote)	апостроф
'\"'	**0x22**	**"** (double quote)	кавычка
'\?'	**0x3F**	**?** (question mart)	вопр. знак
'\000'	**000**	octal number	Восьмеричный код символа
'\0xhh'	**hh**	hex number	Шестнадцатеричный код

Для использования внутренних кодов символов нужна таблица, в которой каждому символу компьютера соответствует числовое значение его кода в десятичном, восьмеричном и шестнадцатеричном представлении. На IBM-совместимом ПЭВМ применяется таблица кодов ASCII [2].

Работу с символьными константами иллюстрирует программа:

```
#include<iostream.h>
void main()
{char c; int i;
c = 'ab'; cout << c << '\t';
```

i = 'ab'; cout << i <<'\t';
i = c; cout << i << '\t';
i = 'ba';
c = i; cout << c << '\t';}

В результате программы на экран будет выведено:

a 25185 97 b

здесь **25185** – двухбайтовое целое число, а **97** – код символа ***'a'*.**

Если выводить на экран односимвольную константу, то будет выведено изображение символа, но если эту же константу поместить, например, в арифметическое выражение, то значением константы будет ее десятичный внутренний код.

Для 32-разрядного компилятора допустимы константы – несколько символов, заключенных в апострофы, которые называются мультисимвольными (multicharacter literal) и имеют тип ***int.***

Строковые константы

Строка или строковая константа - это последовательность символов, заключенная в кавычки.

Внутреннее представление строки в памяти таково: все символы размещаются подряд, и каждый символ занимает 1 байт, в котором размещается внутренний код символа. А в конце строки компилятор помещает еще один символ, называемый байтовым нулем '\0'. Этот символ как любой другой занимает в памяти 1 байт, 8 двоичных разрядов, в которых находятся нули.

Среди символов строковой константы могут быть эскейп-последовательности, например:

"Монография \" Турбо – Паскаль\". "

Строки, записанные в программе подряд или через пробельные символы, при компиляции конкатенируются (склеиваются). Таким образом, в тексте программы последовательность из строк:

"Миру - " "мир!"

эквивалентна одной строке: ***"Миру – мир!"***

Длинную строковую константу можно размещать на нескольких строках в программе, используя еще и символ переноса строк - '\'

В строке может быть один символ, например, **"А",** которая в отличие от символьной константы **'А'** занимает в памяти 2 байта. Строковая константа может быть пустой **"",** при этом ее длина равна 1 байту. Символьная константа не может быть пустой, запись **'' -** не допустима.

Кроме непосредственного использования строк в выражениях, строку можно поместить в символьный массив, например, при его инициализации и обращаться к ней по имени массива (раздел 2).

Знаки операций

Знаки операций – это один из элементов выражений. Выражения есть правило получения значения. Результат операции - это всегда значение.

Знак операции — это один или более символов, определяющих действие над операндами. Внутри знака операции пробелы не допускаются. Операции делятся на унарные, бинарные, тернарные по количеству участвующих в них операндов. Один и тот же знак может интерпретироваться по-разному в зависимости от контекста.

В табл. 1.5 представлены операции, приоритеты (ранги) и ассоциативность операций [3].

Таблица 1.5

Операции языка C++ и их приоритеты

Ранг	**Операции**	**Ассоциативность**
1	**:: . -> ( ) [ ]**	→
2	**! ~ + - ++ -- & *** (тип) sizeof new delete тип ( ) typeid dynamic_cast static_cast reinterpret_cast const_cast	←
3	**.* ->***	→
4	*** / %** (мультипликативные)	→
5	+ – (аддитивные)	→
6	<< >> (сдвиги)	→
7	< <= >= > (сравнения)	→
8	== **!=** (сравнения)	→
9	**&** (поразрядная конъюнкция)	→
10	**^** (поразрядное исключающее ИЛИ)	→
11	**|** (поразрядная дизъюнкция)	→
12	**&&** (логическая конъюнкция)	→
13	**||** (логическая дизъюнкция)	→
14	**? :** (условная операция)	←
15	= *= /= %= += -= &= ^= |= <<= >>= операция присваивания	←
16	throw	
17	, (операция запятая)	→

Кроме стандартных режимов использования операций язык C++ допускает расширение (перегрузку) их действия, дает возможность распространения действия на объекты классов. Примером такой перегрузки являются операции поразрядных сдвигов >> и <<. Когда слева от них в выражениях находятся входные и выходные потоки, они трактуются как операции извлечения данных из потока >> и вывода данных в поток <<.

В табл. 1.6 дано краткое описание стандартных операций языка C++ [5].

Таблица 1.6

Сводка стандартных операций С++

Унарные операции	**Форма**
1. **&** - операция получения адреса некоторого объекта программы	& lvalue (lvalue – имя объекта программы)
2. ***** - разыменование, доступ по адресу к значению объекта	* a (a – указатель на объект)
3**. –** - унарный минус	**–** выражение
4. + - унарный плюс	+ выражение
5**. !** – логическое отрицание (НЕ) **false –** если операнд истинный и **true –** если операнд ложный	!выражение !1 ==0, !(-5)==0(ложь) !0 == 1 (истина)
6. ++ - инкремент, увеличение значения операнда на 1 − префиксная форма - до использования его значения; − постфиксная форма – после использования его значения;	++ lvalue lvalue ++
7. -- - декремент, уменьшение значения операнда на 1 − префиксная форма; − постфиксная форма	-- lvalue lvalue--
8. **sizeof** – размер в байтах внутреннего представления объекта	sizeof выражение **sizeof (тип)**
9. **new** – динамическое выделение памяти	new имя_ипа new имя_ипа инициализатор
10. **delete** – освобождение динамически выделенной памяти	delete указатель delete [] указатель
11. **()** - явное преобразование типа	(тип) выражение
12. **()** - функциональная форма преобразования типа	тип (выражение) тип имеет простое имя
13. **typeid** – операция определения типа операнда	typeid (выражение) typeid(имя_типа)
14. **dynamic_cast -** операция приведения типа с проверкой допустимости при выполнении программы	dynamic_cast <целевой тип> (выражение)
15. **static_cast –** операция приведения типов с проверкой допустимости приведения во время компиляции.	static_cast <целевой тип> (выражение)

Продолжение табл. 1.6

16. **reinterpret_cast** – операция приведения типов без проверки допустимости приведения.	reinterpret_cast <целевой тип> (выражение)
17. **const_cast** – операция приведения типов, которая аннулирует действие модификатора const	const_cast <целевой тип> (выражение)
18**. :: -** операция доступа из тела функции к внешнему объекту	:: имя_объекта
Бинарные операции	**Форма**
Аддитивные:	
19. + - сложение арифметических операндов, или сложение указателя с целочисленным операндом	выражение + выражение
20. – - вычитание арифметических операндов или указателей	выражение – выражение
Мультипликативные	
21. * -умножение арифметических операндов	выражение * выражение
22. / -деление операндов арифметических типов. При целых операндах дробная часть результата отбрасывается	выражение / выражение 20/6 равно 3, -20/6 равно -3, 20/(-6) равно -3
23. % -получение остатка от целочисленного деления	выражение % выражение 13%4 равно 1, (-13)%4 равно -1 13%(-4) равно 1,(-13)%(-4) равно -1
Операции сдвига	
24. << - сдвиг влево битового представления левого целочисленного операнда на количество разрядов, равное значению правого операнда	lvalue << выражение
25. >> - сдвиг вправо битового представления левого целочисленного операнда на количество разрядов, равное значению правого операнда	lvalue >> выражение
Поразрядные операции	
26. & поразрядная конъюнкция (И) битовых представлений целочисленных операндов	выражение& выражение

Продолжение табл. 1.6

<table>
<tr><td>27. | - поразрядная дизъюнкция (ИЛИ) битовых представлений целочисленных операндов</td><td>выражение | выражение</td></tr>
<tr><td>28. ^ - поразрядное исключающее ИЛИ битовых представлений целочисленных операндов</td><td>выражение ^ выражение</td></tr>
<tr><td>Операции отношений (сравнения)
Результат: true (1) , если сравнение истинно и false (0)– если ложно</td><td></td></tr>
<tr><td>29. < - меньше</td><td>выражение < выражение</td></tr>
<tr><td>30. < = - меньше или равно</td><td>выражение <= выражение</td></tr>
<tr><td>31. > - больше</td><td>выражение > выражение</td></tr>
<tr><td>32. > = - больше или равно</td><td>выражение >= выражение</td></tr>
<tr><td>33. = = - равно</td><td>выражение = = выражение</td></tr>
<tr><td>34. ! = - не равно</td><td>выражение != выражение</td></tr>
<tr><td>Логические бинарные операции
Результат: true (1) - истина
и false (0) – ложь</td><td></td></tr>
<tr><td>35. && - конъюнкция (логическое И) скалярных операндов и отношений</td><td>выражение && выражение</td></tr>
<tr><td>36. || - дизъюнкция (логическое ИЛИ) скалярных операндов и отношений</td><td>выражение || выражение</td></tr>
<tr><td>Операции присваивания</td><td></td></tr>
<tr><td>37. = простое присваивание левому операнду значения выражения - операнда из правой части</td><td>lvalue = выражение</td></tr>
<tr><td>38 операция = - составное присваивание левому операнду результат операции между левым и правым операндом;
операциb * / % + - << >> & | ^</td><td>lvalue операция = выражение эквивалентно
lvalue = lvalue операция (выражение)</td></tr>
<tr><td>Операции доступа к компонентам структурированного объекта</td><td></td></tr>
<tr><td>39. . (точка) – прямой доступ к компоненту</td><td>имя_объекта.имя_компонента</td></tr>
</table>

Продолжение табл. 1.6

40. –> косвенный доступ к компоненту структурированного объекта, адресуемого указателем	указатель_на_объект–>имя_компонента
Операции доступа к адресуемым компонентам класса	
41. **.*-** прямое обращение к компоненту класса по имени объекта и указателю на компонент	имя_объекта **.*** указатель_на_компонент
42. –>* косвенное обращение к компоненту класса через указатель на объект и указатель на компонент	указатель_на_объект –>* указатель_на_компонент
43. **:: -** бинарная операция расширения области видимости	имя_класса :: имя_компонента имя_пространства_имен :: имя
44. ( ) – операция круглые скобки – операция вызова функции	имя_функции (список_аргументов)
45. [ ] - операция квадратные скобки - индексация элементов массивов	имя_массива [индекс]
46. **? :** - условная (тернарная) операция;	выражение_1 **?** выражение_2 **:** выражение_3 если выражение_1 истинно, то значением операции является значение выражения_2, если выражение_1 ложно, то значением операции является значение выражения_3
47. **,** - операция запятая – это несколько выражений, разделенных запятыми, вычисляются последовательно слева направо; результатом операции является результат самого правого выражения	(список_выражений)

Операция и выражение присваивания

Операция присваивания обозначается символом '='. Простейший вид операции присвоения: ***l = v*** .

Здесь ***l*** - выражение, которое может принимать значение, ***v*** - произвольное выражение.

Операция присвоения выполняется справа налево, т.е. сначала вычисляется значение выражения ***v***, а затем это значение присваивается левому операнду ***l***. Левый операнд в операции присваивания должен быть так называемым **адресным выражением**, которое иначе называют ***lvalue***.

Примером адресного, или именующего выражения, является имя переменной. Адресным выражением никогда не являются константы. Не является ***lvalue*** и простое выражение, например, выражение ***a+b.*** Адресное выражение это объект, представляющий некоторый именованный участок памяти, в который можно поместить новое значение.

В языке С++ операция присваивания образует выражение присваивания, т.е. ***a = b*** означает не только засылку в ***a*** значения ***b***, но и то, что ***a = b*** является выражением, значением которого является левый операнд после присвоения. Отсюда следует, что возможна, например, такая запись:

a = b = c = d = e + 2;

Если тип правого операнда не совпадает с типом левого, то значение справа преобразуется к типу левого операнда (если это возможно). При этом может произойти потеря значения, например:

int i; char ch; i=3.14; ch=777;

Здесь ***i*** получает значение 3, а значение 777 слишком велико, чтобы быть представленным как ***char***, поэтому значение ***ch*** будет зависеть от способа, которым конкретная реализация производит преобразование из большего в меньший целый тип.

Существует так называемая комбинированная операция присваивания вида: ***a on =b,*** здесь ***on*** - знак одной из бинарных операций:
+ - * / % >> << & | ^ && ||.

Присваивание ***a on = b*** эквивалентно ***a = a on b***.

Разделители

Разделители или знаки пунктуаций входят в состав лексем.

- *Квадратные скобки* ***[]*** – ограничивают индексы массивов и номера индексированных элементов:

  int A[5] ={ …}; A[3]=5;

- *Круглые скобки ( ):*
  - выделяют условное выражение в условном операторе: ***if (x<0) x = -x;***
  - обязательный элемент в определении, описании и вызове функций:

    float F(float x, int n) //определение функции
    {тело функции}
    float F(float , int); // описании функции
    F(3.14, 10); // вызов функции
  - обязательны в определении указателя на функцию и в вызове функции через указатель:

    int (*pointer) ( ); //определение указателя
    (* pointer) ( ); //вызов функции через указатель на функцию
  - применяются для изменения приоритета операций в выражениях
  - элемент оператора цикла:

    for(int i =0, j = 3; i < j ; i+ =2, j++)
    {тело цикла};
  - используются при преобразовании типа:

(имя типа) операнд ; имя типа(операнд);

– в макроопределениях, обрабатываемых препроцессором:

#define имя(список параметров) (строка замещения)

- *Фигурные скобки { }:*

  – Обозначают начало и конец составного оператора или блока.

  Пример составного оператора в условном операторе:

  if (x<y) {x++; y--};

  Пример блока, являющегося телом функции:

  float s (float a, float b)
  {return a+b;}

  После закрывающей скобки '}' не ставится точка с запятой ';'.

  – Используются для выделения списка компонент структур, объединений и классов:

  struct st { char*b; int c;};
  union un {char s [2]; unsigned int i; };
  class m {int b; public : int c, d; m(int);};

  Точка с запятой ';' обязательна после определений каждого типа.

  – Используются для ограничения списков инициализации массивов, структур, объединений, объектов классов при их определении:

  int k[ ] = { 1,2,3,4,5,6,7};
  struct tt {int ii; char cc;} ss ={777 , '\n'};

- *Запятая ','*

  – разделяет элементы списков формальных и фактических параметров функций;

  – разделяет элементы списков инициализации структурированных объектов;

  – разделитель в заголовке цикла ***for*:**

  for(int x=p, y=q, i=2; i < 100; z=x+y, x=y, y=z, i++)

  Запятую - разделитель следует отделять от запятой- операции с помощью круглых скобок:

  int i=1; int m [ ] = { i, (i=2 , i * i) ,i };

- *Точка с запятой ';'* – завершает каждый оператор и пустой в том числе.
- *Двоеточие ':'*

  – служит для отделения метки от оператора: ***метка: оператор;***

  – при описании производного класса:

  class x : A, B {список компонентов};

- *Многоточие '...'* – используется для обозначения переменного числа параметров у функции
- *Звездочка '*'* используется как разделитель в определении указателя:

  int* r;

- *Знак '='* при определении объектов программы с инициализацией отделяет имя объекта от инициализирующего значения.

- *Символ* '**#**' используется для обозначения директив препроцессора.
- *Символ* '**&**' играет роль разделителя при определении ссылок:

  int d; int &c = d;

1.3. Представление данных

Данные - это формализованное представление информации. В компьютерной программе данные – это значение объектов программы. Данные С++ могут быть в виде констант и переменных.

Данные, которые не изменяются в программе, называются константами. Данные, зафиксированные в программе и изменяемые в ней в процессе обработки, являются переменными.

Переменные, перед тем как их использовать в программе, должны быть объявлены. Объявление переменной состоит в первую очередь из указания ее типа.

Понятие типа данных.

Тип данных характеризует:

- внутреннее представление данных в памяти компьютера;
- набор допустимых операций (действий);
- множество допустимых значений.

Классификация данных языка С++. Основные и производные типы

Все типы данных можно подразделить на простые (основные) — они предопределены стандартом языка, и сложные (или составные) — задаются пользователем. Данные простого типа нельзя разложить на более простые составляющие без потери сущности данного. Простые типы данных создают основу для построения более сложных типов: массивов, структур, классов. Простые типы в языке С++ — это целые, вещественные типы, символьный и логический тип и тип **void**.

Для определения и описания переменных основных типов используются следующие ключевые слова:

- **char** (символьный);
- **short** (короткий целый);
- **int** (целый);
- **long** (длинный целый);
- **float** (вещественный);
- **double** (вещественный с удвоенной точностью);
- **void** (отсутствие значения).

В табл. 1.7 приведены основные типы данных с диапазоном значений и назначением типов для компиляторов семейства IBM PC/XT/AT ([3]).

В таблице базовыми типами являются: ***char, int, float, double, void.*** Остальные получаются из них с использованием **модификаторов** типа: ***unsigned*** (беззнаковый), ***signed*** (знаковый), ***long*** (длинный), ***shot*** (короткий).

Целые типы

Целый тип данных предназначен для представления в памяти компьютера обычных целых чисел. Основным и наиболее употребительным целым типом является тип ***int***. Гораздо реже используют его разновидности: ***short*** (короткое целое) и ***long*** (длинное целое).

Таблица 1.7

ОСНОВНЫЕ ТИПЫ ДАННЫХ

Тип	**Размер (биты)**	**Диапазон значений**	**Назначение типа**
unsigned char	8	0…255	Небольшие целые числа и коды символов.
char	8	-128…127	Очень малые целые числа и ASCII-коды.
enum	16	-32768…32767	Упорядоченные наборы целых значений.
unsigned int	16	0…65535	Большие целые и счётчики циклов.
shot int	16	-32768…32767	Небольшие целые. Управление циклами.
int	16	-32768…32767	Небольшие целые. Управление циклами.
unsigned long	32	0… 2^{32}-1	Астрономические расстояния.
long	32	-2^{31} …2^{31}-1	Большие числа, популяции.
float	32	3.4E-38 …3.4E+38	Научные расчёты (7 значащих цифр)
double	64	1.7E-308 … 1.7E+308	Научные расчёты (15 значащих цифр)
long double	80	3.4E-4932 …1.7E+4932	Финансовые расчёты (19 значащих цифр)
void			

По умолчанию все целые типы являются *знаковыми*, то есть старший бит в таких числах определяет знак числа: 0 — число положительное, 1 — число отрицательное. Для представления отрицательного числа используется дополнительный код.

Кроме знаковых чисел на С++ можно использовать *беззнаковые*. В этом случае все разряды участвуют в формировании целого числа. При описании беззнаковых целых переменных добавляется слово ***unsigned*** (без знака).

Для 32-разрядных компиляторов, которые не делают различия между целыми типами ***int* и *long*** целые типы представлены для сравнения в табл.1.8.

Таблица 1.8

Целые типы для 32-разрядных компиляторов

Тип данных	***Размер, байт***	***Диапазон значений***
char	**1**	**-128 ... 127** **(-2^7 – 2^7-1)**
short	**2**	**-32768 ... 32767** **(-2^{15} – 2^{15}-1)**
int	**4**	**-2147483648 ... 2147483647** **(-2^{31} – 2^{31}-1)**
long	**4**	**-2147483648 ... 2147483647** **(-2^{31} – 2^{31}-1)**
unsigned char	**1**	**0 ... 255** **(0 – 2^8-1)**
unsigned short	**2**	**0 ... 65535** **(0 – 2^{16}-1)**
unsigned int	**4**	**0 ... 4294967295** **(0 – 2^{32}-1)**
unsigned long	**4**	**0 ... 4294967295** **(0 – 2^{32}-1)**

Символьный тип

В стандарте С++ для представления символьной информации есть два типа данных, пригодных для этой цели, — это типы ***char*** и ***wchar_t*.**

Тип ***char*** используется для представления символов в соответствии с системой кодировки *ASCII* (*American Standard Code for Information Interchange — Американский стандартный код обмена информации*) [2,3]. Это семибитовый код, его достаточно для кодировки 128 различных символов с кодами от 0 до 127. Символы с кодами от 128 до 255 используются для кодирования национальных алфавитов, символов псевдографики и др.

Тип ***wchar_t*** предназначен для работы с набором символов, для кодировки которых недостаточно 1 байта, например, Unicode. Размер типа ***wchar_t*** обычно равен 2 байтам. Если в программе необходимо использовать строковые константы типа ***wchar_t***, то их записывают с префиксом ***L***, например, ***L"Слово"*.**

Таким образом, значениями символьных данных являются целые числа – значения их внутреннего кода. Однако в операторах ввода/вывода фигурируют сами символы, что иллюстрирует следующая программа:

```
#include <iostream.h>
char  c,b ;
void main()
{ c='*';      b= 55;
cout<<c<<'\t'<<b<<'\t';
cout<< (int)c;}
```

Результатом будет:

```
*       7       42
```

Вывелись символы **'*'** и символ с кодом 55 – символ '7'. И код символа **'*'**, как результат приведения типа символа к целому типу.

Логический тип

Логический (булевый) тип обозначается словом ***bool***. Данные булевого типа могут принимать только два значения: ***true*** и ***false*** и занимают в памяти 1 байт. Значение **false** обычно равно числу 0, значение ***true*** — числу 1.

Вещественные типы

Особенностью вещественных (действительных) чисел является то, что в памяти компьютера они практически всегда хранятся приближенно.

Имеется три вещественных типа данных: ***float, double*** и ***long double***. Основным считается тип ***double.*** Так, все математические функции по умолчанию работают именно с типом ***double***. В табл. 1.9 приведены основные характеристики вещественных типов [2].

Таблица 1.9

Основные характеристики вещественных типов

Тип данных	***Размер, байт***	***Диапазон абсолютных величин***	***Точность, количество десятичных цифр***
float	4	от 3.4E—38 до 3.4E+38	7 — 8
double	8	от 1.7E—308 до 1.7E+308	15 — 16
long double	10 (12)	от 3.4E-4932 до 1.1E+4932	19 — 20

Рекомендуется везде использовать тип ***double***. Работа с ним всегда ведётся быстрее, меньше вероятность заметной потери точности при большом количестве вычислений.

Тип void

Тип ***void*** — самый необычный тип данных языка C++. Нельзя определить переменную этого типа. Тем не менее, это очень полезный тип данных. Он используется:

- для определения функций, которые не возвращают результата своей работы;
- для указания того, что список параметров функции пуст;
- а так же этот тип является базовым для работы с указателями. Указатель на тип ***void - у***казатель ни на что, но который может быть приведен к любому типу указателей. Так всё программирование с использованием Win32 API построено на применении указателей на тип **void [7]**.

Новое обозначение типа

Используя спецификатор ***typedef*** можно в программе вводить удобное обозначение для сложных обозначений типов по следующему правилу:

typedef имя типа новое имя типа.

В следующем примере:

typedef unsigned char cod;   cod symb;

объявлена переменная ***symb*** типа ***unsigned char.***

С помощью операций *, &, [ ], ( ) и механизмов определения структурированных типов можно создавать производные типы.

Представление некоторых форматов производных типов:

1. ***type имя [ ]*** – массив элементов типа type, например: ***long m [5];***
2. ***type1 имя (type2)*** - функция с аргументом ***type2*** и результатом типа ***type1***:
3. ***type * имя*** - указатель на объект типа ***type***, например: ***char*ptr***;
4. ***type* имя [ ]*** –массив указателей на объекты типа ***type***: ***int* arr[10];***
5. ***type (*имя) [ ]*** –указатель на массив объектов типа ***type***: ***int (*ptr) [10];***
6. ***type1* имя (type2)*** – функция, с аргументом типа ***type2*** и возвращающая указатель на объект типа ***type1***;
7. ***type1 (*имя) (type2)*** - указатель на функцию с аргументом типа **type2**, возвращающую значение типа ***type1***;
8. ***type1* (*имя) (type2)*** – указатель на функцию с аргументом ***type2***, возвращающую указатель на объект типа ***type1***;
9. ***type & имя = имя_объекта типа_type*** – определение ссылки;
10. ***type (&имя) (type2)*** – ссылка на функцию с аргументом ***type2***, возвращающую результат типа ***type1***;
11. ***struct имя{ type1 имя1; type2 имя2; };-*** объявление структуры;
12. ***union имя {type1 имя1; type2 имя2;};*** - объявление объединения;
13. ***class имя { type1 имя1; type2 имя2 (type3);};*** - определение класса.

Таким образом, типы можно разделить на ***скаляры, агрегаты и функции.***

Скаляры – это арифметические типы, перечисляемые типы, указатели и ссылки. ***Агрегаты*** (структурированные типы) – массивы, структуры, объединения и классы. ***Функции*** подробно рассмотрим в разделе 3.

Объявление переменных и констант в программе

Форма объявления переменных заданного типа:

имя_типа список имен переменных;

Например: **float x, X, cc2, pot_b;**

Переменные можно инициализировать, то есть задавать им начальные значения при их определении. Форма определения начальных значений проиллюстрирована на примере:

unsigned int year = 1999;

unsigned int year (1999);

Последнее определение с инициализацией разрешено только в функциях.

Допустимы следующие формы объявления именованных констант:

1) константы в объявлении перечисляемого типа;
2) с помощью спецификатора ***const*** :

const имя_типа имя_константы = значение

пример: ***const long M = 99999999***;

3) определение константы в препроцессорной директиве ***define***

пример: ***#define имя_константы значение_константы***

1.4. Объекты программы и их атрибуты

***Переменная* -** это именованная область памяти. Имя переменной – это ссылка на некоторую область памяти. Переменная - частный случай *леводопустимого выражения.*

Понятия леводопустимых и праводопустимых выражений

Леводопустимое выражение – это конструкция для обращения к некоторому участку памяти, куда можно поместить значение (***lvalue, l-значение*).**

Понятие *леводопустимого выражения* включает все возможные формы обращения к некоторому участку памяти с целью изменения его содержимого. Название сформировалось по причине, что *леводопустимые выражения* располагаются слева в операторе присваивания.

Примеры *леводопустимых выражений*:

1) имена скалярных переменных;
2) имена элементов массивов;
3) имена указателей;
4) ссылки на ***lvalue*** (синонимы ***lvalue***);
5) имена элементов структурированных данных:
   имя структуры . имя элемента;
   указатель на структуру -> имя элемента элемента;

6) выражения с операцией '*' - разыменования указателя:

int i, *p =&i;   *p=7;

здесь объявляется переменная и указатель на эту переменную, а затем с помощью операции разыменования указателя осуществляется доступ к ячейке памяти переменной.

7) вызовы функций, возвращающих ссылки на объекты программы.

Выражения, которые могут располагаться в правой части оператора присваивания, называются *праводопустимыми* выражениями.

Примеры *праводопустимых выражений*:

1) любое арифметическое, логическое выражение;
2) имя константы;
3) имя функции (указатель константа);
4) имя массива (указатель константа);
5) вызов функции, не возвращающей ссылки.

В дальнейшем рассмотрении, в качестве ***lvalue*** будет рассматриваться переменная как объект программы.

Кроме типов для переменных явно или по умолчанию определяются:

- класс памяти (задает размещение объекта);
- область действия, связанного с объектом идентификатора (имени);
- видимость объекта;
- продолжительность существования объекта;
- тип компоновки (связывания).

Все перечисленные атрибуты (свойства) взаимосвязаны и должны быть либо явно указаны, либо они выбираются по контексту неявно при определении переменной. Рассмотрим их подробнее.

Тип как уже указывалось выше, определяет размер памяти выделяемого для значения объекта, правила интерпретации двоичных кодов значений объектов.

Класс памяти определяет размещение объекта в памяти и продолжительность его существования**.**

Явно задать *класс памяти* можно с помощью спецификаторов: ***auto, register, static, extern.***

Ниже указаны спецификаторы *класса памяти* и соответствующее им место размещения объекта:

- ***auto* -** автоматически выделяемая, локальная оперативная память - ***сегмент стека*** (временная память). Спецификатор ***auto*** может быть задан только при определении переменной блока, например в теле функции. Локальная продолжительность существования. Этим объектам память выделяется при входе в блок и освобождается при выходе из него. Вне блока переменные класса ***auto*** не существуют.
- ***register* -** автоматически выделяемая по возможности регистровая память (***регистры процессора***). Спецификатор ***register*** аналогичен **auto,** но для размещения переменной используется не оперативная память, а регистры процессора. Если регистры заняты другими переменными, переменные класса ***register*** обрабатываются как объекты класса ***auto***.
- ***static*** - статическая продолжительность существования, память выделяется в ***сегменте данных,*** объект существует до конца программы. Внутренний тип компоновки – объект будет существовать в пределах того файла с исходным текстом программы, где он определен. Этот класс памяти может быть приписан как переменным, так и функциям.
- ***extern*** - внешняя, глобальная память, выделяется в ***сегменте данных***. Объект класса ***extern*** обладает статической продолжительностью существования и внешним типом компоновки. Он - глобален, то есть, доступен во всех файлах программы. Этот класс может быть приписан как переменным, так и функциям.

Класс памяти и соответственно размещение переменной (в стеке, в регистре, в динамически распределенной памяти, в сегменте данных) зависит как от синтаксиса определения, так и от размещения определения в программе.

Область действия (ОД) идентификатора (имени) - это часть программы, в которой можно обращаться к данному имени (сфера действия имени в программе).

Рассмотрим все случаи:

1) имя определено в блоке (локальные переменные)**: ОД** - от точки определения до конца блока;

2) формальные параметры в определении функции (формальные параметры – это те же локальные переменные функции): **ОД** параметров – блок тела функции;
3) метки операторов: **ОД** меток – блок тела функции;
4) глобальные объекты: **ОД** глобальных объектов - вся программа от точки их определения или от точки их описания;
5) формальные параметры в прототипе функции: **ОД** параметров - прототип функции.

Область видимости (ОВ)

Понятие области видимости понадобилось в связи с возможностью повторных определений идентификатора внутри вложенных блоков или функций. В этом случае разрывается связь имени с переменной, то есть с участком памяти данной переменной, она становится "невидимой" внутри вложенного блока, хотя сфера действия имени сохраняется.

ОВ – это часть программы, в которой обращение к имени переменной позволяет обратиться к участку памяти, связанному с данной переменной.

ОВ может быть только меньшей или равной ОД. Следующая программа проиллюстрирует данную ситуацию [3]**:**

```
#include<iostream.h>
int k=0;
void main()
{int k=1;
        {cout << k;
        char k = 'A';       cout << k;
        cout << ::k;        cout << k ;}
cout << k;}
```

Результат выполнения программы: **1A0A1**

Объявлена глобальная переменная ***int k***. Затем в главной функции определяется локальная переменная с тем же именем и того же типа. Глобальная переменная становится "невидимой" в теле функции. Обращаться к "невидимой" внутри функции переменной можно, используя операцию доступа к внешнему объекту ***::k***.

В главной функции определен внутренний блок, в котором определена локальная переменная внутреннего блока с тем же именем ***char k.*** От точки определения символьной переменной до конца внутреннего блока локальная целочисленная переменная ***int k*** становится недоступной. После выхода из блока видимость (доступность) данной переменной восстанавливается.

Продолжительность существования

Продолжительность существования **-** это период, в течение которого идентификаторам в программе соответствуют конкретные объекты в памяти.

Определены три вида продолжительности: ***статическая, локальная и динамическая.***

Объектам со ***статической продолжительностью существования*** память выделяется в начале выполнения программы и сохраняется до конца программы.

Статическую продолжительность имеют все функции и файлы.

Все глобальные переменные (т.е. объявленные вне всех функций) обладают статической продолжительностью существования.

Локальные переменные (объявленные в функциях), со спецификатором ***static*** также имеют статическую продолжительность**.** Статическая переменная, локализованная в функции, не теряет своего значения при отработке функции. Инициализируется статическая переменная только при первом вызове функции, при втором вызове и последующих вызовах – значением этой переменной будет то, что сохранилось в памяти при предыдущем вызове.

Глобальные и статические переменные по умолчанию инициализируются нулями.

Если в функции имеется описание переменной со спецификатором ***extern****,* это означает, что эта переменная глобальная и ее определение дано в другом месте вне функции. Такая переменная имеет статическую продолжительность существования.

Локальной продолжительностью существования обладают автоматические (локальные) переменные, объявленные в блоке. Такие переменные создаются при каждом входе в блок, где они определены и уничтожаются при выходе. Локальные переменные должны инициализироваться только явно, иначе их начальные значения не предсказуемы. Область действия локального объекта – блок. Спецификатор класса ***auto*** всегда избыточен, так как этот класс по умолчанию приписывается всем объектам, определенным в блоке.

Объекты с ***динамической продолжительность существования*** создаются и уничтожаются с помощью операторов в процессе выполнения программы по желанию программиста. Память таким переменным выделяется в области динамически распределяемой памяти, называемой **кучей.**

Для создания объекта используются операция ***new*** или функции *malloc()* и *calloc(),* а для уничтожения - операция ***delete*** или функция *free ().*

Операция: ***new имя_типа*** или ***new имя_типа инициализатор*** выделяет и делает доступным участок памяти для объекта данного типа и в выделенный участок заносит значение инициализатора, что не обязательно. Операция возвращает адрес первого байта выделенного участка, или нулевое значение в случае неудачи. Если надо определить динамическую переменную типа ***int*****,** следует объявить указатель на ***int*** и ему присвоить результат операции **new,** например: ***int*r = new (15);***

В дальнейшем доступ к выделенному участку памяти обеспечивается выражением ****r.***

Продолжительность существования выделенного участка – от точки создания до конца программы или до явного освобождения памяти операцией ***delete r;***

Тип компоновки

Если программа состоит из нескольких файлов, каждому имени, используемому в нескольких файлах, может соответствовать:

1) один объект, общий для всех файлов, или

2) один и более объектов в каждом файле.

Файлы программы могут транслироваться отдельно, и в этом случае возникает проблема установления связи между одним и тем же идентификатором и единственным объектом, которому он соответствует. Таким объектам компоновщик обеспечивает ***внешнее связывание*** при объединении отдельных модулей программы (первый случай).

Для объектов, локализованных в файлах, используется ***внутреннее связывание*** (второй случай)**;**

Тип компоновки компилятор устанавливает по контексту.

Определения и описания объектов программы.

Все описанные выше атрибуты (тип, класс памяти, ОД и так далее) приписываются объекту при его определении (объявлении) или при его описании, а также контекстом определения и описания.

В чем разница между определением или описанием.

При ***определении (definition)*** или ***объявлении*** объекта ему дается имя и устанавливаются атрибуты объекта, в соответствии с которыми выделяется нужный объем памяти и определяется формат внутреннего представления объекта. При ***определении*** происходит связывание имени объекта с участком памяти. ***Определение*** выполняет инициализацию объекта. ***Определение*** объекта может быть только одно в программе.

Описание или ***декларация (declaration)*** дает знать компилятору, что объект определен и напоминает свойства объекта (в основном компилятор интересуют типы).

Обычно, ***описание*** - это представление в конкретной функции уже объявленного где–то объекта. ***Описаний*** объекта может быть несколько в программе.

Нередко описание и определение по внешнему виду совпадают. Не совпадают они в следующих случаях:

1)описание – прототип функции;

2)описание содержит спецификатор ***extern;***

3)описывается класс или структурный тип;

4)описывается статический компонент класса;

5)описывается имя типа с помощью ***typedef.***

Определения (объявление) переменных заданного типа имеет следующий формат:

s m тип имя1 иниц.1, имя2 иниц.2, … ;

где *s* **-** спецификатор класса памяти **auto , static , extern , register,**
m **–**модификатор ***const*** или ***volatile***:
const **-** указывает на постоянство объекта;
valatile **–** указывает на изменчивость объекта без непосредственного обращения к нему.
Синтаксис ***инициализации*** переменной, определяющий на этапе компиляции начальное значение переменной**:**
***имя = инициализирующее выражение*;**
либо:
имя (инициализирующее выражение) – применяется только в функциях.

1.5. Выражения и преобразования типов

Выражение – это правило получения нового значения. Выражения формируются из операндов, операций и круглых скобок. Порядок применения операций к операндам определяется рангами операций и их ассоциативностью. Круглые скобки, как и в математических выражениях, позволяют изменить порядок выполнения операций.

При выполнении операций нужно учитывать особенности представления в программе данных разных типов, являющихся операндами выражений. В связи с этим рассмотрим преобразование и приведение типов, допустимые в С++.

Явное приведение типа

Синтаксис функционального приведения типа:

type*(*выражение)

Примеры: ***int*('*')*, float (5/3), char(65).***

Функциональное преобразование типа не может применяться для типов, не имеющих простого имени. Например, конструкции:

unsigned long (15/7) или ***char*(0777)***

вызовут ошибку при компиляции.

Кроме функционального преобразования типа может использовать каноническую операцию приведения к требуемому типу. Для ее представления обозначение типа заключается в круглые скобки. Такая операция может применяться и для типов, имеющих сложное обозначение. Допустима записи:
(unsigned long)15/7 или ***(char*) 0777***

Другую возможность явного преобразования типов со сложным обозначением дает введение собственных обозначений с помощью ***typedef***:

typedef unsigned long ul;   ul(15/7);
typedef char* pchar*;**   ***pchar(0777);

В последних версиях языка С++ [5] введены еще четыре операции явного преобразования типа, имеющих следующий формат:

название_ cast <целевой тип> операнд

dynamic_cast - операция приведения типа с проверкой допустимости на этапе выполнения программы;

static_cast – операция приведения типов с проверкой допустимости приведения во время компиляции;
reinterpret_cast – операция приведения типов без проверки допустимости приведения;
const_cast – операция приведения типов, которая аннулирует действие модификатора ***const***.

Приведенным выше примерам соответствуют следующие выражения:
static_cast <unsigned long>(15/2); и ***static_ cast <char*>(0777);***

При преобразовании типов существуют некоторые ограничения. Но прежде чем остановиться на них, рассмотрим стандартные преобразования типов, выполняемые по умолчанию.

Стандартные преобразования типов

При вычислении выражений операции требуют, чтобы операнды имели соответствующий тип, а если требования не выполнены, производится неявное приведение типа.

Неявное приведение типов происходит при инициализации, когда тип инициализирующего выражения приводится к типу определяемого объекта. То же относится ко всем формам операции присваивания, происходит неявное преобразование типа выражения к типу левого объекта. Реально типы могут преобразовываться один в другой многими способами. Рассмотрит на данном этапе лишь некоторые из них. Преобразования типов, выполняемые неявно:

- преобразования в логические значения: в значения типа ***bool*** преобразуются обобщенные целые числа (включая и символы), вещественные числа и указатели; ненулевые значения преобразуются в ***true***, а нулевые - в ***false***;
- преобразование указателей: любой указатель может быть неявно преобразован в ***void****; значение **0** преобразуется в любой указательный тип; неконстантный указатель преобразуется в константный указатель того же типа.
- преобразование операндов в арифметических выражениях.

Последнее преобразование рассмотрим подробнее. При преобразовании нужно различать преобразования, изменяющие внутреннее представление данных и преобразования, изменяющие только интерпретацию внутреннего представления. Например, при преобразовании ***unsigned int*** в ***int*** изменяется только интерпретация внутреннего представления. При преобразовании ***double*** в ***int*** изменяется как длина участка памяти для внутреннего представления, так и способ кодировки. При таком преобразовании возможен выход за диапазон допустимых значений типа ***int*** , потеря значимости, точности и так далее. Именно поэтому в программах рекомендуется с осторожностью применять преобразования типов.

В арифметических выражениях происходит автоматическое преобразование типа операндов к общему типу – наиболее высокому типу согласно иерархии типов. Преобразование происходит по схеме, изображенной на рис.3 [4]:

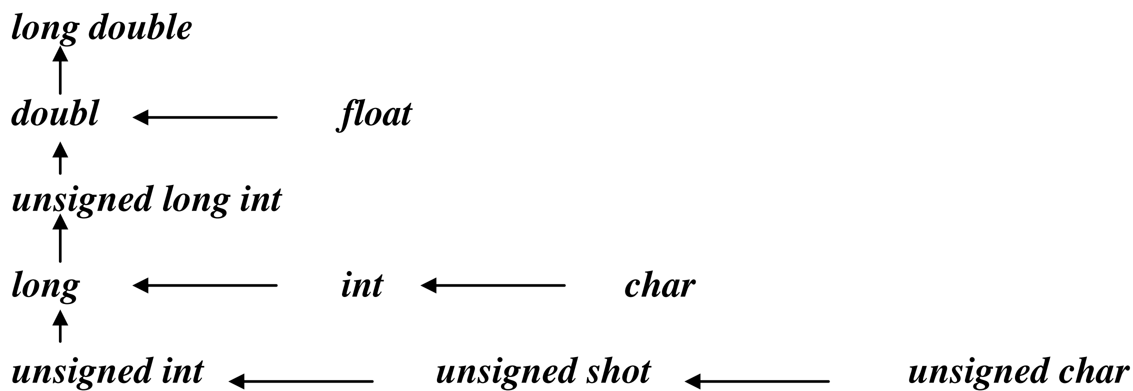

Рис.3. Схема последовательных преобразований типов операндов

Преобразования, гарантирующие сохранение значимости.

Используя в программе арифметические выражения, следует учитывать, что некоторые из них приводят к потерям информации и изменению числового значения. На рис.4 в соответствии со стандартом языка C++ [2] представлены стрелочками преобразования, гарантирующие сохранение точности и неизменность числового значения.

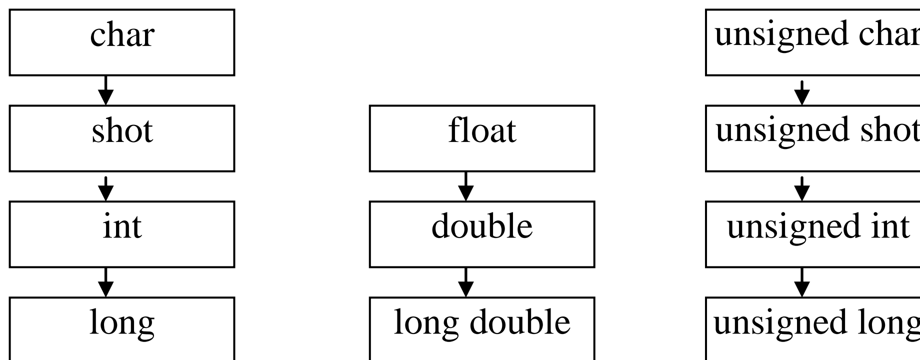

Рис.4. Последовательности преобразований типов, гарантирующие сохранение значимости

1.6. Операторы языка C++

Операторы - это конструкции языка, определяющие действия и логику выполнения действий в программе.

По характеру действий различают два типа операторов: операторы преобразования данных и операторы организации обработки данных.

Первая группа – это последовательно выполняемые операторы. Рассмотрим их.

Последовательно выполняемые операторы - это:

- операторы присваивания;
- операторы ввода и вывода данных;
- операторы вызова функций;
- операторы-выражения.

Все они являются типичными операторами преобразования данных, и именно эти операторы используются при программировании линейных процессов вычислений.

Каждый оператор языка C++ заканчивается разделителем *';'*. Любое выражение, после которого стоит *';'*, воспринимается компилятором как оператор (не считая выражений, входящих в заголовок оператора ***for***).

Оператор присваивания является также оператором выражения, так как оператор формируется из выражения присваивания, в конце которого поставлена "точкой с запятой".

Оператор простого присваивания: ***lvalue = выражение*** ;

Оператор составного присваивания: ***lvalue оп= выражение*** ;

Примеры: ***x*= i;*** ***i=x-4*i;***

Оператор – выражение – вызов функции, не возвращающей никакого значения.

Форма оператора: ***имя функции (список фактических параметров);***

Пример фрагмента программы:

```
void main ()
{ void cd (char);            // прототип функции
  cd ('*'); }          // оператор – выражение, вызов функции
```

Операторы – выражение – инкремент или декремент l-значения:

Формы операторов, с учетом префиксных и постфиксных форм операций инкремента и декремента:

++(lvalue); ***(lvalue)++;***

- -(lvalue); ***(lvalue)- -;***

Пример: ***i++; -*** возрастание значения переменной ***i*** на единицу.

Операторы ввода - вывода данных также являются операторами – выражениями.

Операторы используют непосредственно входные и выходные потоки из библиотеки потоковых классов, описания которых находятся в заголовочном файле **iostream.h.**

Препроцессорная директива: ***#include <iostream.h>***

подключает к программе библиотеку ввода/вывода, построенную на основе механизма классов. Ввод/вывод верхнего уровня в C++ построен на основе потоков, с которыми программа при исполнении обменивается данными.

Поток следует понимать как последовательность обмениваемых байтов между оперативной памятью и внешними устройствами (файл на диске, принтер, клавиатура, дисплей, стример и т.п.), или между различными участками оперативной памяти.

cin – имя стандартного входного потока по умолчанию, связанного с клавиатурой; ***cout*** – имя стандартного выходного потока по умолчанию, связанного с экраном дисплея;

>> - операция извлечение данных из потока или операция ввода;

<< - операция вставки данных в поток или операция вывода;

Операции возвращают ссылки на соответствующие потоки, то есть фактически сами потоки.

Операции извлечения данных из потока и вставки данных в поток являются основой для операторов ввода-вывода данных.

Форма оператора ввода (ввод данных с внешнего устройства в ОП):

cin >> *lvalue*;

lvalue –это именованный участок оперативной памяти, значение которого можно изменять, частный случай – имя переменной. В последнем случае формат оператора ввода таков: ***cin >> имя переменной ;***

Из потока ***cin*** извлекается значение и помещается в оперативную память, выделенную под переменную.

Внутри ЭВМ данные хранятся в виде двоичных кодов, которые регламентированы для каждого типа данных. При вводе выполняется преобразование символов визуального представления данных из потока в двоичные коды внутреннего представления данных, при этом происходит автоматическое распознавание типов вводимых данных.

Форма оператора вывода (вывод данных из ОП на внешнее устройство):

cout<< выражение ;

Из оперативной памяти извлекается значение выражения и помещается в выходной поток ***cout***. При этом происходит преобразование двоичных кодов типизированного значения выражения в последовательность символов, изображающих значение на внешнем устройстве. Интерпретация выводимого значения производится автоматически.

К этой группе операторов отнесем ***пустой оператор, составной и блок***.

Пустой оператор представляется символом "точкой с запятой", перед которым нет никакого выражения. Пустой оператор не предусматривает никаких действий. Используется там, где синтаксис языка требует присутствия оператора, а по смыслу программы никакие действия не должны выполняться.

Составной оператор - это последовательность операторов, заключенная в фигурные скобки.

Блок - определения, описания и последовательность операторов, заключенные в фигурные скобки.

Синтаксически и блок, и составной оператор являются единичными операторами. Внутри них после каждого оператора ставятся "точки с запятой", но после их закрывающей фигурной скобки символ **';'** не ставится.

Описания и определения объектов программы, после которых стоит **';'** также считаются операторами.

Перед любым оператором может стоять ***метка*** (включая и пустой оператор, описания и определения). *Метки* – это идентификаторы, локализованные в теле функции.

Иногда удобно поставить метку перед пустым оператором в конце функции.

К операторам управления работой программы относятся операторы выбора, циклы, операторы передачи управления.

***Операторы выбора* -** реализуют в программе алгоритмические схемы ветвления. И в первую очередь к ним относится условный оператор.

Условный оператор, реализующий алгоритмическую схему развилка, имеет следующую форму:

if (выражение) оператор 1 else оператор 2;

в качестве ***выражения*** может быть любое скалярное выражение, которое автоматически приводится к логическому типу; каждый **из *операторов (1 или 2)*** – это по синтаксису один оператор простой или составной.

Если ***выражение*** истинно (то есть его значение не равно нулю), то выполняется ***оператор 1***(прямая ветвь алгоритма*)*, в противном случае выражение – ложно и выполняется ***оператор 2*** (альтернативная ветвь).

Операторами 1,2 не могут быть описания и определения, но могут быть блоки с описаниями и определениями.

Перед ***else*** ставится ***';'***, если ***оператор 1*** – простой оператор.

Пример условного оператора, в котором в прямой и альтернативной ветвях находятся составной оператор и блок:

if(x>0) {x = -x; q (x);} else {int i=3; q(i*x);}

Условный оператор может иметь сокращенную форму:

if (выражение) оператор

Пример: ***if (a>b) a = -a;***

В случае ложности ***выражения*** никаких действий не выполняется.

Вложенные условные операторы

Если ***оператор 1*** или ***оператор 2*** в полной форме или ***оператор*** в сокращенной форме являются также условными операторами, то эти операторы называются вложенными условными операторами, которые также имеют прямую и альтернативную ветвь. При определении, к какому условному оператору какая относится альтернативная ветвь, существует правило: рассматриваются слева направо каждый ***else.*** Очередная альтернативная ветвь ***else*** принадлежит к ближайшему к ней, свободному (не связанному с другим ***else)*** оператору ***if.***

Рассмотрим пример:

if (x = =1) if (y = =1) cout << "x =1 и y =1"; else cout << "x != 1";

Условный оператор составлен неправильно. Действительно, при значениях ***x=1 и y!=1***, будет выведена неправильная фраза ***"x!=1"***

Ниже представлены два варианта правильно составленных операторов:

if (x = = 1) { if (y = = 1) cout << "x =1 и y =1";}else cout << "x != 1";

или

if (x= =1) if (y= =1) cout<<" x =1 и y =1 ";else ; else* cout *<< "x != 1";

<u>Оператор switch</u>, реализующий алгоритмическую схему мультиветвление имеет две формы – форма переключателя и форма выбора варианта.

Форма переключателя:

switch (переключающее выражение)
{ case константное варажение1: операторы1;
…
case константное выражение N: операторы N;
default: операторы;
};

здесь ***переключающее выражение*** может быть любого перечисляемого типа: целочисленного или символьного; ***константные выражения*** должны быть того же типа (или приводящимися к нему), что и переключающее выражение и различны по значению; для каждой ветви алгоритма возможно использовать несколько *константных выражений*, например: ***case 1 : case 5: операторы;***

Сначала вычисляется переключающее выражение. Полученное значение сравнивается со значениями константных выражений. Если совпадает значение, то выполняются операторы данного варианта и операторы всех последующих вариантов, включая и вариант с меткой ***default***. Если значение не совпало ни с одним значением константных выражений, выполняется вариант с меткой ***default.*** Вариант ***default*** может располагаться в любой части внутри фигурных скобок, а может просто отсутствовать и тогда при отсутствии совпадения не выполняется ничего.

Форма альтернативного выбора:

switch (выражение)
{ case константное варажение1 : операторы 1; break;
…
case константное выражение N: операторы N; break;
default: операторы; break;
};

Если значение ***выражения совпадет с одним из константных значений***, то выполняются операторы только данного варианта, после чего управление программой передается оператору, следующему за оператором ***switch.***

Обработка любого варианта может включать кроме операторов еще и описания и определения объектов. В этом случае все это нужно заключить в фигурные скобки, тем самым превратить в блок.

Операторы цикла. Циклы реализуют алгоритмическую схему повторения обработки данных. В C++ определены три разных цикла:

- *цикл с предусловием:*

  while (выражение-условие) тело_цикла

- *цикл с постусловием:*

  do тело_цикла

while (выражение-условие);

- *цикл с параметром:*

for (инициализация_цикла; выражение-условие; выражения коррекции) тело_ цикла

Тело_цикла - это отдельный оператор, который всегда завершается точкой с запятой, либо составной оператор, либо блок. Только описание или определение не может быть телом цикла. Операторы цикла задают многократное выполнение операторов тела цикла.

Выражение-условие **–** во всех операторах скалярное (чаще всего логическое или арифметическое) определяет условие продолжение повторения обработки, если оно истинно (отлично от нуля). Прекращение выполнения цикла происходит в случаях:

- ложное (нулевое) значение *выражения-условия;*
- выполнение в теле цикла оператора передачи управления за пределы тела цикла.

Последнюю возможность рассмотрим позже.

Оператор ***while*** - ***оператор цикла с предусловием***. При входе в цикл вычисляется условие, если оно отлично от нуля, выполняется *тело_цикла.* Затем вычисление *выражения-условия* и выполнение *тела_цикла* повторяются, пока условие не станет ложным.

Следующая функция подсчитывает длины строки, заданной с помощью адресующего ее указателя – параметра функции [6]:

```
int lt (char*stroka)
{ int ln =0;
while (*stroka++) ln++ ;
return ln;}
```

Ниже приведены некоторые выражения условия:

```
while (a<b),        while (point != NULL) (point-указатель),
while(point),        while (point ! = 0).
```

Если в теле цикла не изменяется выражение- условие, могут возникнуть бесконечные циклы или "зацикливание".

Пример бесконечного цикла, с пустым оператором в теле: ***while (1);*** Такой цикл может быть закончен за счет событий, явно не предусмотренных в программе. Например, событие – указание операционной системе "снять задачу".

Оператор ***do*** является ***оператором цикла с постусловием***. При входе в цикл выполняется *тело_цикла*. Затем проверяется *выражение-условие* и, если его значение истинно, вновь выполняется тело цикла и так далее. К выражению-условию требования те же: оно должно изменяться при итерациях за счет операторов тела циклов.

Цикл ***for*** – ***цикл с параметром***. Заголовок цикла после зарезервированного слова ***for*** в круглых скобках включает три части: *инициализация_цикла, выражение-условие и список выражений для коррекции*,

которые разделяются двумя знаками ';'. Каждая из этих частей может отсутствовать, но даже если все они отсутствуют, два разделителя - ';' должны присутствовать в заголовке.

Инициализация_цикла – это выражение, или определения объектов одного типа. Вычисляется один раз при входе в цикл. Заканчивается точкой с запятой. Как правило, определяются и инициализируются параметры цикла.

Выражение-условие - логическое выражение, проверяется на каждой итерации цикла. Если его нет, полагают, что оно истинно. В этом случае сохраняется следующая за условием точка с запятой.

Выражения коррекции – список выражений, разделенных запятыми, которые вычисляются на каждой итерации после выполнения *тела_цикла* до следующей проверки условия.

Тело_цикла – может быть отдельным простым оператором, составным оператором или блоком. При входе в цикл один раз вычисляются выражения из секции *инициализация_цикла*. Вычисляется значение *выражения-условия,* если оно истинно, выполняются операторы *тела_цикла* и вычисляются *выражения коррекции.* Далее цепочка повторяется, пока *выражение-условие* не станет ложным.

for – цикл с заданным числом повторений, для подсчета числа итераций используется управляющая переменная – параметр цикла.

Примеры показывают, что управляющая переменная может изменяться на любую требуемую величину:

for(c =0 ; c<=100 ; c +=10)…; ***for(b =100; b >= -100; b -= 25)…;***
for(let ='A ; let <= 'Z'; let ++)…; ***for(pr =0.0 ; pr <= 100.0 ; pr +=0.5)…;***

Область действия имени объекта, объявленного в заголовке цикла, определяется от точки объявления объекта до конца блока, в котором цикл присутствует.

Вложенные циклы

Разрешено вложение любых циклов в любые циклы.

Действует следующее правило: для каждой итерации внешнего цикла выполняются все итерации внутреннего цикла.

Операторы передачи управления

К операторам передачи управления относятся оператор безусловного перехода ***goto***, оператор возврата из функции ***return***, оператор выхода из цикла или переключателя ***break*** и оператор перехода к следующей итерации цикла ***continue***.

Оператор безусловного перехода имеет вид:

goto идентификатор метки;

Передача управления разрешена на любой идентификатор, помеченный меткой. Имя метки действует и уникально в теле функции, где расположен оператор. Существует запрет: нельзя передавать управление через определение,

содержащее инициализацию объекта, но можно обходить вложенные блоки, содержащие определение с инициализацией.

При использовании оператора ***goto*** рекомендуется:

1) не входить внутрь блока извне;
2) не входить внутрь условного оператора;
3) не входить извне внутрь переключателя;
4) не передавать управление внутрь цикла;

Есть случаи, когда использование оператора ***goto*** обеспечивает наиболее простое решение:

1) выход из нескольких вложенных циклов или переключателей;
2) к одному участку программы перейти из разных мест функции;

Оператор возврата из функции имеет вид***:***

return выражение; или ***return;***

Возвращает в точку вызова функции значение ***выражения***, которое, если присутствует, может быть только скалярным. Если ***выражение*** в операторе ***return***. отсутствует, то возвращаемое функцией значение имеет тип ***void.***

Оператор выхода из цикла или переключателя break прекращает выполнение операторов цикла или переключателя и осуществляет передачу управления к следующему за циклом или переключателем оператору.

При многократном вложении циклов и переключателей оператор ***break*** не может передать управление из внутреннего уровня на самый внешний. ***break*** позволяет выйти из внутреннего цикла в ближайший внешний цикл.

Оператор перехода к следующей итерации continue используется только в циклах. Завершает текущую итерацию цикла и начинается проверка условия продолжения цикла. Его действия эквивалентны оператору передаче управления в самом конце тела цикла.

Раздел 2
Адреса, указатели, ссылки, массивы

2.1. Указатели и адреса объектов

Понятие переменной определялось как имя ячейки памяти, в которой хранится значение указанного типа. Каждая ячейка памяти имеет свой уникальный **адрес**. Чтобы получить адрес переменной (простой или структурированной), используется унарная операция **&,** которая применима к объектам, размещенным в памяти и имеющим имена**.**

Указатель – это объект (переменная), значениями которого являются адреса участков памяти, выделенных для объектов конкретных типов.

Организация памяти в процессорах семейства 80x86

Основная память ПЭВМ – это память с непосредственным (прямым) доступом к участку памяти с любым адресом, не зависимо от того, к какому участку выполнялось предыдущее обращение.

Наименьшим адресуемым участком памяти является байт, содержащий 8 бит (двоичных разрядов). Оперативная память представляет собой ***последовательность пронумерованных байтов***, физические адреса (номера байтов), которых начинаются от 0 и возрастают.

Участки памяти, кратные 16 байт называются ***параграфами***, которые пронумерованы от нуля до 65535 . Всего 65536 параграфов, это соответствует объему оперативной памяти = 1 Мбайт.

Физический адрес параграфа (номер байта, с которого начинается параграф) равен произведению номера параграфа на 16. Начало любого параграфа может быть принято за начало ***сегмента памяти***. Длина сегмента памяти не может превышать 64Кбайт

Процессоры семейства 80x86 используют сегментированную организацию памяти.

В полном сегментированном адресе любого объекта два 16-разрядных числа: ***0xHHHH : 0xHHHH***, где ***H*** **–**любая шестнадцатеричная цифра. Первое число – это номер параграфа, с которого начинается сегмент. Это число называется ***сегментной частью адреса.*** Второе число – определяет смещение от начала сегмента интересующего нас первого байта объекта и называется ***смещением*** или ***относительной частью адреса.***

Обе части адреса – это четырехразрядные шестнадцатеричные числа, или 16-разрядные двоичные числа. Они могут принимать значения от 0 до 65535 (64Кбайт)

По полному сегментному адресу формируется 20-разрядные физические адреса: сегментная часть* 16 (номер первого байта сегмента) + относительная часть (смещение в сегменте) = ***0xHHHH * 0x10 + 0xHHHH = =0xHHHH0 + 0xHHHH = 0xHHHHH.***

Для работы с сегментированными адресами в процессорах семейства 80x86 имеются регистры сегментов:

- ***CS*** *(Code Segment)*-регистр сегмента кода программы, используется для формирования адресов выполняемых команд;
- ***DS*** *(Data Segment)*-регистр сегмента данных, используется для формирования адресов данных;
- ***SS*** *(Stack Segment)*- регистр сегмента стека, для формирования адресов данных из стека;
- ***ES*** *(Extra Segment)* - регистр сегмента расширения дополнительного сегмента данных.

Схема возможного размещения сегментов в памяти

Процессор может одновременно адресовать 4 различных сегмента памяти, каждый из которых может быть размером до 64 Кбайт. Они могут пересекаться и даже могут быть размещены все в одном участке размером в 64Кбайт. Схематично расположение сегментов представлено на рис.5.

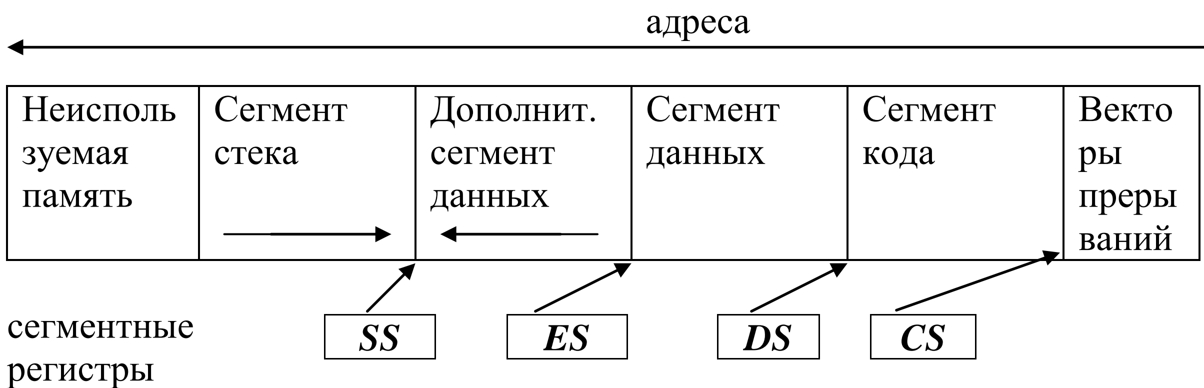

Рис. 5. Возможное размещение сегментов в памяти

При сегментной организации адресуется 2^{20} (~ 1 млн.) байтов. Но для формирования адреса использовались два 16-разрядных двоичных числа. С помощью этих чисел можно было бы адресовать на прямую без разбиения на сегменты $2^{16} * 2^{16} = 2^{32}$ (~ 4 млрд.) байтов.

Все возможности не используются из-за допустимости пересечения сегментов. Например, комбинации чисел **0x0000:0x0413, 0x0001:0x0403, 0x0021:0x0203** адресуют один и тот же физический адрес **0x413 (0x00000+0x0413==0x00010+0x0403==0x00210+0x0203==0x00413).**

Значениями указателей в общем случае (***far***) является число типа ***long*****,** старшие байты которого состоят из сегментной части адреса, а младшие – из смещения ***0xHHHHHHHH*****.**

Указатели в компиляторах TC++ и BC++ делятся на 4 группы, для описания которых введены модификаторы (служебные слова), позволяющие выбирать внутреннее представление указателей:

- **near** – ближние;
- **far** – дальние;
- **huge** – нормализованные;
- **_cs, _ds, _es, _ss** – сегментные.

Ближние указатели имеют длину 2 байта и представляют смещение в конкретном сегменте. Адресуют только 64Кбайта памяти.

Дальние **указатели** занимают 4 байта, содержат и номер сегмента и смещение. Адресуют только 1Мбайт памяти. Разные сочетания сегментной части и смещения могут адресовать один и тот же физический адрес, один и тот же байт могут адресовать несколько указателей.

Нормализованные указатели имеют длину 4 байта и воспринимаются, как одно 32-разрядное значение, но позволяют однозначно адресовать только 1Мбайт памяти. Любому физическому адресу представляется единственное сочетание сегмента и смещения. Дается номер только того сегмента, для которого смещение не больше 15 (от 0 до 15). Например, для физического адреса **0x413** полный адрес будет **0x0041:0x0003** и соответственно значение нормализованного указателя однозначно равно **0x00410003.**

Сегментные указатели – это ближние указатели, имеют длину 2 байта и хранят значение смещения в известных сегментах (данных, стеке, и так далее). Для доступа к этим сегментам введены регистровые переменные (_CS, _DS, _ES, _SS), в которых хранится сегментная часть адреса.

А сегментные указатели задают смещение в известных сегментах. Например, определим указатель**: nt _ss * pss;** значение указателя **pss** позволяет задавать смещение в сегменте стека.

2.2. Объявление указателей

В простейшем случае объявление указателя имеет вид:

type * имя указателя;

В определении указателя обязательно должен быть указан тип данных, на который "указывает" указатель, так как указатель несет информацию об адресе участка памяти и о размерах этого участка памяти.

type – тип объекта, на который "указывает" указатель;

type * – тип указателя, если рассматривать указатель как переменную.

Например, в объявлении:

int*A, *B, *C, D;

объявлено три указателя ***A , B , C*** на данные типа ***int*** и переменная ***D-*** типа ***int.***

Инициализация указателей

Инициализация указателей имеет две формы:

type * имя указателя = инициирующее выражение ;

type * имя указателя (инициирующее выражение);

В качестве ***инициирующего выражения*** может быть:

1) явно заданный адрес участка памяти: ***type * имя = (type *) 0x158e0ffc;***
2) выражение, возвращающее адрес объекта с помощью операции ***'&'***:
   type count= ;   type_1* iptr = (type_1*) & count;
3) указатель, имеющий значение:
   char ch = '+', *R = & ch;   char *ptr = R;
4) инициирующее выражение равно пустому указателю: ***NULL*** – специальное обозначение указателя, ни на что не указывающего. ***char * ch (0)*** эквивалентно ***char * ch (NULL).***

Доступ к значению объекта, ***адресуемому указателем,*** обеспечивает операция разыменования, выражение: ****имя указателя;*** позволяет получить значение самого объекта:

char cc = 'a', *pc = &cc; cout << *pc;

и изменить это значение: ****pc = '+'; cin >> *pc;***

Можно сказать, что **указатель –* обладает правами переменной.

Если определен указатель без инициализации**:** ***char *p,*** использовать выражение ****p*** в операциях присваивания или в операторе ввода не правомерно. Указатель ***p*** не обладает значением - адресом участка памяти, куда можно было занести значение. Указателю можно присвоить адрес участка памяти объекта типа ***char*****:**

1) ***p = new char ;*** //динамическое выделение, **delete (p) -** освобождение памяти
2) ***p = (char *) 0x157e0ffc*;** //значение адреса преобразуется к указателю ***char****
3) ***p =(char*) malloc (sizeof (char));***
// динамическое выделение памяти **free (p)** – освобождение памяти
4) ***p*** присвоить адрес переменной типа ***char***: ***char c; p = &c;***
5) ***p*** присвоить значение другого указателя на данные типа ***char****:

char s , *ptr=&s; p=ptr;

Теперь допустимы операции ****p ='+' ; cin >> *p;***

Указатели константы и на константы

Указатели бывают константами (то есть значение указателя нельзя поменять в программе) и указателями на константы (то есть указатель, адресующий неизменный участок памяти).

В общем случае определение константного указателя на константу имеет вид: ***type const * const имя указателя;***

в этом определении ***type const –*** тип константы, на которую "указывает" указатель. А в последующей части после разделителя ***'*'*** - ***const имя указателя*** определяется имя константного указателя**.** В этом случае нельзя изменить значения указателя и нельзя изменить значение участка памяти, на который "указывает" указатель.

Определение константного указателя имеет вид:

type* const имя указателя;

Например, в определении: ***char * const K = (char *) 1047;***
K – это указатель, значение которого невозможно изменить. Возможно менять значения по адресу ***K*,** то есть допустимы операции: ****K = '*'*** или ***cin >> *K;***

Определение указателя на константу имеет вид:

type const * имя указателя;

Например, после определений:
const float A = 56,1 float B = 1.1; float const *pA = &A ;
не допустимы операции изменения значения ****pA***, но допустимо разорвать связь указателя ***pA*** с константой ***A,*** присвоив указателю адрес другой переменной: ***pA = &B;*** и тогда допустимы, например**:** ****pA=0.1*** или ***cin >> *pA;***

Еще раз отметим свойства операции взятия адреса - ***'&'*** столь важной для инициализации указателей**.** Операция ***'&'*** применима только к объектам, имеющим ***имена*** и ***размещенным в ОП.***

2.3. Типы указателей и операции над ними

Типы указателей могут быть стандартными (указатели на объекты основных типов) и производными (указатели на массивы, указатели на указатели, указатели на функции, указатели на константы, на структуры, на объединения, на объекты классов и на данные типов определенных пользователем с помощью спецификатора ***typedef).***

Указатели на основные типы данных - на арифметические данные и на символьные.

Пример работы с указателем на данные арифметического типа ***int***:

int n = 6, *pn = &n;

cout << pn<<'\t' << *pn*; //будет выведено*: 0x1E10FFC 6

*//**адрес значение***

Пример работы с указателем на данные символьного типа ***char***:

char c = 65, *pc= &c;

cout << pc << '\t' << *p << '\t' << &c*; //будет выведено ***A A A,

то есть при выводе указателя, разыменованного указателя и адреса будет выводиться сам символ с кодом ***65*** – это прописная латинская буква ***A***

Если мы хотим вывести значение адреса символьной переменной и ее внутренний код надо поступить так:

cout << (void*)pc <<'\t'<< (int)*pc //результат: ***0x1e76a0c2 65***

явно привести указатель к типу ***void***, а разыменованный указатель к типу ***int.***

Операция ***- & имя*** дает однозначный результат – адрес данного объекта.

Результат операции разыменования ****имя_указателя –*** неоднозначен и зависит не только от значения указателя, но и от его типа, который указывает размер участка памяти, который будет доступен:

В следующем примере один и тот же адрес переменной рассматривается как значение указателей разных типов:

{long L = 0x12345678L;

char*ch=(char*)&L; int*I =(int*)&L; unsigned long*UN=(unsigned long*)&L;

cout << hex; //манипулятор вывода в шестнадцатеричном формате

cout << (int)*ch << '\t' << *I << '\t' << *UN << endl;

cout << (void*)ch << '\t' << I << '\t' << UN << endl; …}

Результат фрагмента программы:

0x78 0x5678 0x12345678

0x18efoffc 0x18efoffc 0x18efoffc

В первой строке выведены значения ячеек памяти в шестнадцатеричном формате, во второй строке значения указателей (адресов каждой ячейки).

Значения самих указателей совпали, а значения объектов зависят от размера участка памяти, на который "указывает" соответствующий указатель (1 байт, 2 байта или 4 байта).

Рассмотрим указатель типа ***void*.*** Такой указатель может иметь значение – адрес какого-либо объекта программы, однако он не содержит информации о размере этого объекта в памяти, и, следовательно, посредством такого указателя нельзя работать с объектом. Это можно исправить, используя явное преобразование типа указателя.

Указатель типа ***void**** приводится к любому указателю явным преобразованием типа, например:

void* p; int* ip, i= 10; p = &i ; ip = (int*)p;

Любой указатель приводится к типу ***void**** по умолчанию:

int i, j, *b = &i; void* a = &j;

a = b*; – допустимо, так как ***b автоматически приводится к типу ***void*,***

b = a; **–**недопустимо, надо явно приводить ***a*** к типу ***int****: ***b= (int*) a;***

Операции над указателями

1) разыменование (*)**;**
2) преобразование типа явное (только каноническая форма);
3) присваивание;
4) взятие адреса (**&);**
5) аддитивные операции (сложение и вычитание);
6) инкремент – автоувеличение (++)**;**
7) декремент – автоуменьшение **(--);**
8) операции отношения.

Первые три операции уже рассмотрены выше. Рассмотрим остальные.

Понятие указателя на указатель

Указатель как переменная, обладает адресуемой ячейкой в памяти (размером в общем случае – 4 байта) и, следовательно, к имени указателя применима операция взятия адреса. Адрес указателя можно присвоить указателю на указатель. Рассмотрим пример, в котором объявляется указатель на указатель.

```
int i =3, *a =&i, **b=&a;              //int**b;     указатель на указатель
cout << b << '\t' << *b << '\t' << **b;
0x1204fff0        0x1204fff4        3
//(&a)            (a, &i)           (*a, i)
```

Сначала будет выведено значение указателя ***b***, равное адресу указателя ***a***. Затем выведется значение ****b*** – результат разыменования указателя ***b***, которое равно значению указателя ***a*** – адресу переменной ***i***. И последним выведется значение *****b*** - результат разыменования указателя ****b,*** равное значению результата разыменования указателя ***a,*** равное значению самой переменной ***i.***

Аддитивные операции

I. Вычитание

Вычитание указателей на объекты одного и того же типа

Разность однотипных указателей – это «расстояние» между двумя участками памяти, адресуемыми указателями, выраженное в единицах кратных длине объекта того типа, который адресуется указателями:

```
type*p1, *p2… //далее указатели получают значения адресов объектов
p1 – p2 = ((long) p1 – (long) p2) / sizeof(type)
```

//здесь ***(long) p –*** значение указателя

Рассмотрим фрагмент программы:

```
…{int a = 1, b = 4, *aa = &a, *bb = &b;
cout << aa << '\t' << bb << endl;
cout << (aa – bb) << '\t' << ( (long) aa – (long) bb );}
```

Результат выполнения:

0x18e80ffe 0x18e80ffc **-** адреса переменной ***a*** и переменной ***b***

1   2 - значение разности указателей (1) и разность значений указателей (2).

Программа иллюстрирует, что:

1) разница двух указателей на соседние объекты одного типа равна ***1***
2) разница значений указателей на соседние объекты типа ***int*** равна ***2***
3) адрес первого объявленного объекта больше чем следующего**,** так как стек заполняется от больших адресов к малым адресам.

Вычитание из указателя целое число:

При вычитании из указателя целого числа: ***указатель – K;*** формируется значение указателя меньшее на величину ***K*sizeof(type),*** где ***K*** – вычитаемое число, ***type*** - тип объекта, к которому относится указатель.

II. Сложение указателя с целым значением

При сложении указателя с целым числом: ***указатель + K;*** формируется значение указателя большее на величину ***K* sizeof(type).*** Пример:

```
…{int a=0, b = 1, c = 3, d = 5, *p = &d;
cout << *p << '\t' << p << '\t' << *(p + 1) << '\t' << (p + 1) << '\t';
cout << *(p + 2) << '\t' << (p + 2) << '\t'<< *(p + 3) << '\t' << (p + 3);}…
```

Результат:

5 0x15e80ff4 3 0x15e80ff6 1 0x15e80ff8 0 0x15e80ffa

III. Инкремент ++ (декремент --) – увеличивает (уменьшает) значение указателя на ***sizeof(type).*** Указатель смещается к соседнему объекту с большим (меньшим) адресом.

Таким образом, обладая правами объекта (как именованного участка памяти), указатель имеет адрес, длину внутреннего представления и значение:

1) значения указателя – это адреса объектов, на которые "указывает" указатель;
2) адрес указателя получают операцией: ***&имя указателя***;
3) длина внутреннего представления
   sizeof (void*) == sizeof (char*) == sizeof (любой указатель) == 4
4) раз указатель это объект, то можно определять указатель на указатель и так сколько нужно раз: ***int i = 10, * p1 = &i, **p2 = &p1, ***p3 = &p2;***

type** – тип указателя на указатель на переменную типа ***type;***

type*** – тип указателя на указатель, который указывает на указатель на переменную типа ***type;***

При этом: ****p3 = = p2,*** ****p2 == p1,*** ****p1 == i,***

то есть к значению переменной можно добраться путем последовательных разыменований указателя ***p3***: ****(*(*p3)),*** что эквивалентно – ****** p3.***

2.4. Ссылки

Ссылка – это другое имя уже существующего объекта (именованного участка памяти).

При определении ссылки ей не выделяется память, ссылка дает новое имя уже существующему участку памяти.

При определении ссылок инициализирующее выражение обязательно!

Инициализирующим выражением может быть только ***l-value***, то есть объект, имеющий место в памяти и имя.

Объявление ссылки имеет вид:
type &имя_ссылки (инициирующее выражение)
type &имя_ссылки = инициирующее выражение

Пример:
int J = 5, &I = J; //можно было отдельно определить **int &I = J**
I++; //I = = 6 и J = = 6
I+ = 10; //I == 16 и J == 16

Все действия, которые происходят со *ссылкой,* происходят и с *переменной*, инициализирующей ссылку, и наоборот все, что происходит с *переменной*, происходит и со *ссылкой*.

При работе со ссылками действуют следующие правила:
1) ссылка не может иметь тип ***void*** (***void & a=b –*** не допустимо);
2) ссылку нельзя создавать динамически;
3) нет ссылок на др. ссылки (***int& &a = r –***не допустимо)
4) нет указателей на ссылки (***int & *p= &r –***не допустимо)
5) нет массивов ссылок
long a = 1, b = 2, c = 3; long& MR[ ] = {a, b, c} - не допустимо
5) ссылка навсегда связана с объектом инициализации;
6) объект может несколько ссылок

*Инициализация ссылок не обязательна**:***
1) в описаниях внешних объектов ***extern float & b;***
2) в описания ссылок на компоненты классов;
3) в спецификациях формальных параметров;
4) в описании типа, возвращаемого функцией;

Результатом применения операции ***sizeof (имя ссылки***) является размер в байтах участка памяти, выделенного для инициализатора ссылки. Операция **&** возвратит адрес инициализатора ссылки.

2.5. Массивы

Массив – это совокупность данных одного типа, рассматриваемых как единое целое. Все элементы массива пронумерованы. Массив в целом характеризуется ***именем***. Обращение к элементам массива выполняется по их номерам (***индексам*)**, которые всегда начинаются с **0**.

Массивы могут состоять из числовых данных, символов, строк, указателей и так далее. Символьные массивы, как правило, представляют в программе текстовые строки.

Если для обращения к какому-то элементу массива достаточно одного индекса, массив называется *одномерным*.

Если данные удобно представлять в форме таблицы, тогда для обращения к конкретному элементу надо задать два индекса: номер строки и номер столбца. Такие массивы называются *двумерными(или матрицами)*.

Массивы с числом индексов больше 1 называются *многомерными*.

Форма объявления одномерного массива (вектора):

type имя массива [K];

K – константное выражение, определяет *размер* массива (количество элементов в массиве); ***type –*** тип элементов массива.

Форма объявления многомерного массива:

type имя массива [K 1] [K2] …[K N];

type – тип элементов массива;

N -размерность массива- количество индексов, необходимое для обозначения конкретного элемента;

K1…KN* -** количество элементов в массиве по ***1…N- му измерению, так в двумерном массиве ***K1*** – количество строк, а ***K2*** – количество столбцов;

Значения индексов по ***i***-му измерению могут изменяться от ***0*** до ***Ki – 1*;**

K1*K2*…*KN – размер массива (количество элементов массива).

Например, ***float Z[13][6];*** определяет двумерный массив, первый индекс которого изменяется от 0 до 12, а второй индекс - от 0 до 5.

Внутреннее представление массивов в оперативной памяти

При определении массива для его элементов выделяется участок памяти, размеры которого определяются количеством элементов массива и их типом: ***sizeof (type)* количество элементов массива,*** где ***sizeof(type)*** – количество байтов, выделяемое для одного элемента массива данного типа.

Операция ***sizeof*** имеет две формы: ***sizeof(тип)*** и ***sizeof(объект).*** Учитывая это, а также то, что имя массива – это имя структурированной переменной, размер участка памяти, выделенного для всего массива, можно определить также из следующего выражения: ***sizeof (имя массива).***

В оперативной памяти все элементы массива располагаются подряд. Адреса элементов одномерных массивов увеличиваются от первого элемента к последнему. В многомерных массивах элементы следуют так, что при переходе от младших адресов к старшим наиболее быстро меняется крайний правый индекс массива. Так, при размещении двумерного массива в памяти сначала располагаются элементы первой строки, затем второй, третьей и т. д.

Например, элементы массива **int T [3][3]** будут располагаться так:

первая строка			**вторая строка**			**третья строка**		
T[0][0]	**T[0][1]**	**T[0][2]**	**T[1][0]**	**T[1][1]**	**T[1][2]**	**T[2][0]**	**T[2][1]**	**T[2][2]**

Возрастание адресов →

Инициализация числовых и символьных массивов.

Инициализация - это задание начальных значений объектов программы при их определении, проводится на этапе компиляции.

<u>Инициализация одномерных массивов</u> возможна двумя способами: либо с указанием размера массива в квадратных скобках, либо без явного указания:

int test [4] = { 10, 20 , 30 ,40 };

char ch [ ] = { ‘a’ , ‘b’ , ‘c’ , ‘d’ };

Список инициализации помещается в фигурные скобки при определении массива.

В первом случае инициализация могла быть неполной, если бы в фигурных скобках находилось меньше значений, чем четыре.

Во втором случае инициализация должна быть полной, и количество элементов массива компилятор определяет по числу значений в списке инициализации.

Рассмотрим инициализацию многомерных числовых массивов. Массив любой мерности в С++ рассматривается как одномерный. Например, массив:
int B [3] [2] [4] – трактуется как одномерный массив из трех элементов - двумерных массивов (матриц) ***int [2] [4]***, с именами ***B[0], B[1], B[2],*** каждый из этих массивов трактуется как одномерный массив, состоящий из двух одномерных массивов (векторов) ***int [4]***.

Располагаются элементы массива в ОП естественно одномерно: сначала располагаются все элементы первой матрицы, как было описано выше, затем второй и третьей. При определении, например, трехмерного массива инициализация его элементов может проводиться либо с учетом внутренней структуры массива, отделяя двумерные и одномерные массивы фигурными скобками, либо списком значений в соответствии с расположением элементов в оперативной памяти:

```
int B [ 3 ] [ 2 ] [ 4] = { { {1, 2, 3, 4}, {5, 6, 7, 8} },
                           { {9, 10, 11, 12}, {13, 14, 15, 16} },
                           { {17, 18, 19, 20}, {21, 22, 23, 24} } };
```

эквивалентно:

```
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 21, 23, 24 };
```

Для инициализации символьных массивов чаще всего используются строки. Инициация символьного массива может быть выполнена значением строковой константы, например, следующим образом:

char stroka [10] = "строка";

При такой инициализации компилятор запишет в память символы строковой константы и добавит в конце двоичный ноль '\0’, при этом памяти будет выделено с запасом (10 байт).

Можно объявлять символьный массив без указания его длины:

char stroka1 [ ] = "строка";

Компилятор, выделяя память для массива, определит его длину, которая в данном случае будет равна 7 байтов (6 байтов под собственно строку и один байт под завершающий нуль).

Инициализация двумерных символьных массивов проводится следующим образом:

char name [5] [18] = {"Иванов", "Петров", "Розенбаум", "Цой", "Григорьев"};

При объявлении символьного массива будет выделено памяти по 18 байтов на каждую строку, в которые будут записаны символы строки и в конце каждой строки добавлен байтовый ноль, всего 90 байтов. При таком объявлении в массиве остается много пустого места.

При определении многомерных массивов с инициализаций (в том числе и двумерных) значение первого индекса в квадратных скобках можно опускать.

Количество элементов массива по первому измерению компилятор определит из списка инициализации. Например, при определении:

char sh [ ] [40] = {"=======================================",
"| F | F | F | F | F |",
"======================================="};

компилятор определит три строки массива по 40 элементов каждая, причем **sh[0], sh[1], sh[2] –** адреса этих строк массива в оперативной памяти. В каждую из строк будет записана строковая константа из списка инициализации.

Форма обращения к элементам массивов

С помощью операции [] (квадратные скобки) обеспечивается доступ к произвольному элементу массива.

Обращение к элементу одномерного массива имеет вид:

имя массива [индекс],

где ***индекс –*** это не номер элемента, а его ***смещение*** относительно первого элемента с индексом **0.** Пример:

int A[10];   A[5] = 0;   //A[5] – обращение к шестому элементу массива.

Для обращения к элементам многомерного массива также используется имя массива, после которого в квадратных скобках стоит столько индексов, сколько измерений в массиве. Пример обращения к элементу двумерного массива: ***имя массива [i][j],*** это обращение к элементу ***i –й*** строки и ***j-го*** столбца двумерного массива.

2.6. Массивы и указатели

Имя массива воспринимается двояко на этапе определения массива и на этапе его использования.

С одной стороны ***имя массива*** следует рассматривать как имя структурированной переменной. И применение таких операций как ***sizeof*** и ***&*** к имени массива дают в качестве результата соответственно размер внутреннего представления в байтах всего массива и адрес первого элемента массива:

sizeof(имя массива) – размер массива;

&имя массива - адрес массива в целом, равный адресу первого элемента**.**

С другой стороны ***имя массива*** - это ***константный указатель,*** значением которого является адрес первого элемента массива и значение данного указателя нельзя изменять.

Рассмотрим одномерный массив. В соответствии с вышесказанным соблюдается равенство:

имя_массива = = &имя_массива = = &имя_массива[0],

то есть, имя массива отожествляется с адресом массива в целом и с адресом его первого элемента.

Данные соотношения позволяют сформулировать еще один способ обращения к элементам массива.

В соответствии с операцией сложения указателя с целым числом, если к имени массива (константному указателю) прибавить целое число ***i*:**

***имя_массива +i* ,**

то на машинном уровне сформируется следующее значение адреса:

имя_массива + i * sizeof(тип элемента),

которое равно адресу ***i - го*** элемента массива:

имя_массива + i = = &имя_массива[i].

Операция разыменования адреса объекта предоставляет доступ к самому объекту. Применяя операцию разыменования для левой и правой части представленного выше уравнения, получаем результат:

****(имя_массива + i) = = имя_массива [i],***

из которого следует, что обращаться к ***i-му*** элементу массива можно любым из этих эквивалентных способов**.**

Многомерные массивы рассмотрим на примере объявления двумерного массива: ***type T [m][n];*** ***m, n*** – целочисленные константы, ***type*** – тип элемента массива. В массиве ***m*** строк по ***n*** элементов в строке (***n*** столбцов**).** Имя двумерного массива ***T*** также отожествляется с его адресом, а также с адресом его первого элемента ***T[0][0].***

Каждая строка двумерного массива - это одномерный массив с именем ***T[i],*** где ***i*** – индекс строки, и имя этого массива является адресом первого элемента строки и адресом строки в целом.

Адреса строк массива равны: ***T[0], T[1],…,T[m-1],*** и, как показано выше, эквивалентны следующим выражениям: ***T, *(T+1),…,*(T+m-1).***

Выражения: ***T[i] = = *(T+i) = =&T[i][0]*** – представляют адрес ***i –***строки массива, и, следовательно, обращаться к элементу ***i –й*** строки ***j-го*** столбца массива можно следующими способами:

T[i][j] = = *(*(T+i)+j)==*(T[i]+j)==(*(T+i))[j]

Указатели и строки

Как и массивы типа ***char,*** указатели ***char**** могут инициализироваться строковыми константами:

char * имя_1 = "строка";
char * имя_2 = {"строка"};
char * имя_3("строка");

В операторе вывода по значению указателя выведется строка до байтового нуля:

cout << имя указателя на строку;

Если надо вывести адрес строковой константы, при выводе надо воспользоваться приведением типа указателя к ***void****:

cout << (void*) имя указателя на строку;

Если надо представлять в программе группу текстовых строк, целесообразно объявить массив указателей типа ***char**** по количеству строк:

char* name [5]= {"Иванов", "Петров", "Розен", "Цой", "Григорьев"};

Адреса отдельных строк равны значениям указателей из массива: ***name[0], name[1], name[2], name[3], name[4].***

По адресам выводим строки:

for (int i=0; i < 5; i++) cout << name[i] << endl;

Благодаря манипулятору ***endl*** фамилии выведутся в столбик.

Ввод/вывод элементов массивов

Ввод/вывод числовых массивов

Ввод/вывод значений арифметических массивов производится поэлементно.

Для одномерного массива следует организовать цикл (повторение обработки данных), в каждой итерации которого, изменять индекс элемента и производить ввод (вывод) значения соответствующего элемента. Пример иллюстрирует ввод элементов одномерного массива:

```
int test [100] ;
for (int i = 0; i < 100; i++) cin >> test [i];…
```

При вводе/выводе элементов двумерного массива для обращения к элементам необходимо устанавливать номера строк и столбцов элементов. При этом целесообразно учитывать, как элементы массива располагаются в оперативной памяти. Внешний цикл следует организовать по номерам строк. В теле этого цикла для каждого номера строки организовать внутренний цикл, в котором перебирать номера столбцов. Следуя такому алгоритму, обращение к элементам массива будет происходить в той последовательности, в которой они располагаются в оперативной памяти. Пример программы иллюстрирует вышесказанное:

```
#include < iostream.h>
#include< stdlib.h>
void main()
{ int T[3][4];
for( int i =0 ; i < 3; i++) { cout << '\n';
for( int j=0 ; j < 4; j++) { T[i][j] = rand( );  cout << T[i][j] << "   ";} }
```

При вводе/выводе элементов многомерных массивов должно быть организовано столько циклов перебора индексов сколь мерный массив. Причем, самый внешний цикл – цикл перебора самого левого индекса, а в самом вложенном цикле варьируется самый правый индекс.

Ввод/вывод символьных массивов

Ввод и вывод символьных массивов можно производить поэлементно, как и числовых массивов, то есть рассматривать символьный массив как набор отдельных символов.

Синтаксис языка С++ допускает также обращение к символьному массиву целиком по его имени, а именно по адресу этого массива в

оперативной памяти. При этом также допускается обращение к отдельным элементам – символам по их индексу в массиве.

Объявим некоторый символьный массив:

char text [80];

Следующие операторы позволяют произвести ввод символьных строк:

1) ***cin>>text*; - символы извлекаются из стандартного входного потока ***cin*,** и заносятся в оперативную память по адресу ***text*,** ввод начинается от первого символа, отличного от пробела до первого обобщенного пробела. В конце строки в память помещается двоичный ноль.

2) ***cin.getline(text, n);* -** извлекаются из стандартного входного потока ***cin*** любые символы, включая и пробелы, и заносятся в оперативную память по адресу ***text*.** Ввод происходит до наступления одного из событий: прочитан ***n-1*** символ или ранее появился символ перехода на новую строку ***'\n'***, (при вводе с клавиатуры это означает, что была нажата клавиша ***Enter*).** В последнем случае из потока символ ***'\n'*** извлекается, но в память не помещается, а помещается в память символ конца строки ***'\0'***.

3) ***gets(text);*** - читается строка из стандартного потока (по умолчанию связанного с клавиатуры) и помещается по адресу ***text***. Вводятся все символы до символа перехода на новую строку ***'\n' (Enter)*,** который в память не записывается, а в конце строки помещает двоичный ноль ***'\0'*.**

4) ***scanf("%s", text);*** – из стандартного потока читается строка символов до очередного пробела и вводит в массив ***text***. В конце помещается байтовый ноль. Если строка формата имеет вид ***"%ns"***, то считывается ***n*** непробельных символов.

5) ***scanf("%nc", text);*** – из стандартного потока вводятся **n** любых символов, включая и пробелы, и символ конца строки.

Если стандартный входной поток связан с клавиатурой, все приведенные выше операторы, в основе которых лежат вызовы функций, останавливают программу до ввода строки символов.

Вывод строки позволяют произвести следующие операторы:

1) ***cout << text;*** **-** выводит всю строку до байтового нуля в стандартный выходной поток ***cout*,** по умолчанию связанный с экраном дисплея.

2) ***puts(text);*** **-** выводит строку в стандартный поток и добавляет в завершении символ **'\n'** – перехода на новую строку.

3) ***printf("%s" ,text);*** **-** выводит в стандартный выходной поток всю строку;

printf("%ws" ,text); **-** выводит всю строку в поле ***w*,** где ***w*** – целое число, количество текстовых позиций на экране для вывода символов. Если **w** больше числа символов в строке, то слева (по умолчанию) или справа (формат **"%-ws"**) от строки выводятся пробелы. Если ***w*** меньше, чем количество выводимых символов, то выводится вся строка.

printf("%w.ns" ,text); **-** выводит **n** символов строки в поле ***w*;**

printf("%.ns" ,text); -выводит **n** символов строки в поле ***w = n;***

2.7. Создание динамических массивов

При определении массива ему выделяется память. При определении внешнего или статического массива память выделяется в ***сегменте данных,*** элементы массива по умолчанию инициализируются нулевыми значениями. При определении автоматического массива память выделяется в ***сегменте стека.***

В обоих случаях происходит ***статическое выделение памяти*** под массив либо в сегменте статических данных, либо в стеке. Это означает, что либо до конца программы, либо до конца блока массивам будут соответствовать участки памяти.

Можно выделять память под массив ***динамически*** в программе, с помощью соответствующих операторов и также программно освобождать ее по желанию программиста. Рассмотрим этот случай.

Объявим указатели (переменные) на объект типа ***type.*** Указатели можно связать с массивами - статическим (1) и динамическими (2, 3):

1) ***type * имя1 = имя уже определенного массива типа type;***
2) ***type* имя2 = new type [количество элементов массива];***
3) ***type * имя3 = (type *) malloc (количество элементов * sizeof(type));***

Проиллюстрируем это на примерах:

int n [5] = { 1, 2, 3 , 4 ,5}; //определен статический массив
int*pn = n; //присоединение указателя к массиву статической памяти
//значение ***pn*** - адрес первого элемента массива
float*p1 = new float [10]; //динамически выделено **40 байт** памяти
//значение ***p1*** **-** адрес первого байта этого участка
double*p2 = (double*) malloc (4 * sizeof (double));
//динамически выделено **32** байта памяти, значение ***p2-*** адрес первого байта

После того как указатель связан с массивом доступ к элементам массива осуществляется следующими способами:

1) ***имя указателя [индекс элемента]***
2) ****(имя указателя + индекс элемента)***
3) ****имя указателя ++***(операция допустима, т.к. указатель – переменная)

Пример вывода элементов арифметического массива (производится поэлементно):

int A [10] = {7.3, 5.5, 1, … };
for (int i=0, *pA=A; i<10 ; i++)
cout<<*pA++<<" "; //что эквивалентно ***cout << pA[i],*** или ***cout << *(pA+i)***

Массивы указателей

Как любые переменные указатели могут быть объединены в массивы.

Примеры массивов указателей статической памяти:

char*A [6] – одномерный массив указателей из 6 элементов – указателей на объекты типа ***char***, элементы массива **A** имеют тип ***char*.***

Выражение: ***(long)(A+1)- (long) A = 4 (байта)*** **–** дает внутренний размер указателя.

По общему правилу можно определять одномерные динамические массивы указателей, учитывая, что тип элемента массива равен ***type****:

1) ***type* * имя1 = new type* [количество указателей в массиве] ;***

2) ***type* * имя2 = (type* *) malloc (количество элементов * sizeof(type*));***

Создание двумерного динамического массива с помощью динамического массива указателей.

Двумерный массив составляется из одномерных массивов - строк массива, представляющих собой одномерные массивы, которые можно представить с помощью указателей.

Сколько строк в двумерном массиве (матрице) – столько надо указателей для их представления. Таким образом, надо объявить массив указателей на строки матрицы.

Массив указателей на строки матрицы тоже является одномерным массивом, который тоже можно определить динамически.

Элемент массива указателей имеет тип ***type*,*** где ***type*** **-** тип элементов исходной матрицы, поэтому массив указателей можно представить с помощью указателя на указатель ***type**.***

Пусть надо выделить память на динамическую матрицу элементов типа ***int***, размером: ***m***- строк по ***n*** элементов в строке (***m***-строк и ***n*** – столбцов).

```
int m, n;
cin >> m >> n;              //размеры матрицы m x n введем с клавиатуры
int ** W;             //определим указатель на указатель
                      //для представления динамического массива указателей
W = new int*[m];            // выделяем память для массива указателей из m
//элементов типа int*, W[i] – указатель на i-ю строку динамической матрицы
//В цикле выделяем память на одномерные массивы из n элементов
for (int i =0; i< m; i++) W[i] = new int [n];
```

Память на динамическую матрицу (***m x n)*** выделена, к элементам которой можно обращаться с помощью стандартных выражений:

W[i] [j] **или** ****(*(W+i)+j)*;**

После работы можно освободить память, причем сначала надо освободить память, выделенную на элементы строк матрицы, а затем освободить память, выделенную на массив указателей на строки матрицы:

```
for (int i =0; i< m; i++) delete W[i];
delete[ ] W;
```

Указатель на массив. Многомерные массивы динамической памяти.

Указатель на массив – это переменная, значением которой является адрес массива. Этому указателю доступен участок памяти, выделенный под массив. При прибавлении к такому указателю 1 получается значение адреса, большее на размер массива.

Определение указателя на массив рассмотрим на примере определения указателя ***ptr*** на массив из 6 элементов типа ***int:***

int (*ptr) [6];

Выражение**:**

(long)(ptr+1) - (long) ptr == 12 (байтов)

возвращает размер массива в байтах, на который "указывает" указатель ***ptr.***

Многомерный динамический массив можно определить следующим образом: ***new тип массива,*** где тип массива – это тип элементов массива и нужное количество квадратных скобок, с константными выражениями, определяющие размеры массива.

При таком определении существуют следующие правила:

1) при выделении памяти для динамического массива его размеры должны быть полностью определены.
2) нельзя явно инициализировать массив при динамическом выделении памяти.

Определим трехмерный динамический массив типа ***int [3][2][4].*** Этот массив состоит из трех подмассивов - матриц ***2x4.*** Определим указатель на подмассив – матрицу, типа ***int*** размером ***2x4***: ***int (*p) [2] [4];***

С помощью операции **new** выделяется участок памяти для трех матриц, причем операция возвращает адрес первой матрицы массива**,** который и присваивается указателю ***p***:

p = new int [3] [2] [4] – выделена память для массива ***3x2x4*** типа ***int***

Чтобы освободить память, выделенную на массив, используем оператор: ***delete [ ] p;***

Формы обращения к элементам массива:

p[i][j][k], либо ****(*(*(ptr1+i)+j)+k).***

В операторе:

cout << (long) (p+1) – (long) p**;** **//16**

- размер участка памяти, на который "указывает" указатель ***p.***

Определение типа массива имеет следующий вид:

typedef type имя_типа_массива [k1] [k2] ..[kn];

Пример: ***typedef float array [3][5][2];***

array – имя типа массива **3x5x2** с элементами типа ***float***

array Mas; **-** определен массив ***Mas*** типа ***array.***

Определение типа указателя на массив имеет следующий вид:

typedef type (*имя_типа_указателя_на_массив) [k1] [k2] ..[kn];

Пример: ***typedef float (*tpm) [5][2]*****;**

tpm – имя типа указателя на массив **5x2** с элементами типа ***float***

tpm pm ; - объявлен указатель такого типа

Раздел 3. Функции

3.1. Определение, описание и вызовы функций

Программа на языке С++ представляет собой совокупность произвольного количества функций, одна (и единственная) из которых -

главная функция с именем ***main***. Выполнение программы начинается и заканчивается выполнением функции ***main*.** Выполнение неглавных функций инициируется в главной функции непосредственно или в других функциях, которые сами инициируются в главной.

Функции – это относительно самостоятельные фрагменты программы, оформленные особым образом и снабженные именем.

Каждая функция существует в программе в единственном экземпляре, в то время как обращаться к ней можно многократно из разных точек программы.

Упоминание имени функции в тексте программы называется ***вызовом функции***. При вызове функции активизируется последовательность образующих ее операторов, а с помощью передаваемых функции параметров осуществляется обмен данными между функцией и вызывающей ее программой.

По умолчанию все функции внешние (класс памяти ***extern***), доступны во всех файлах программы. При определении функции допускается класс памяти ***static*,** если надо, чтобы функция использовалась только в данном файле программы.

Определение функции

Определение функции – это программный текст функции. Определение функции может располагаться в любой части программы, кроме как внутри других функций. В Языке С++ нет вложенных функций.

Определение состоит из заголовка и тела функции:

<тип > <имя функции > (<список формальных параметров>)

тело функции

1) ***тип –*** тип возвращаемого функцией значения, с помощью оператора ***return*,** если функция не возвращает никакого значения, на место типа следует поместить слово ***void***;
2) ***имя функции –*** идентификатор, уникальный в программе;
3) ***список формальных параметров (сигнатура параметров*) –** заключенный в круглые скобки список спецификаций отдельных формальных параметров, перечисляемых через запятую:

   <***тип параметра> <имя параметра>,***

   <тип параметра> <имя параметра> = <умалчиваемое значение>,

   если параметры отсутствуют, в заголовке после имени функции должны стоять, либо пустые скобки **(),** либо скобки **– (*void*);**

   для формального параметра может быть задано, а может и отсутствовать умалчиваемое значение – начальное значение параметра;

4) ***тело функции –*** это блок или составной оператор, т.е. последовательность определений, описаний и операторов, заключенная в фигурные скобки.

Вызов функции

Вызов функции передает ей управление, а также фактические параметры при наличии в определении функции формальных параметров.

Форма вызова функции:

имя функции (список фактических параметров);

Список фактических параметров может быть пустым, если функция без параметров: ***имя функции ();***

Фактические параметры должны соответствовать формальным параметрам по количеству, типу, и по расположению параметров.

Если функция возвращает результат, то ее вызов представляет собой выражение с операцией "круглые скобки". Операндами служат имя функции и список фактических параметров. Значением выражения является возвращаемое функцией значение.

Если функция не возвращает результата (тип – ***void),*** вызов функции представляет собой оператор.

При вызове функции происходит передача фактических параметров из вызывающей программы в функцию, и именно эти параметры обрабатываются в теле функции вместо соответствующих формальных параметров.

После завершения выполнения всех операторов функция возвращает управление программой в точку вызова.

Описание функции (прототип)

При вызове функции формальные параметры заменяются фактическими, причем соблюдается строгое соответствие параметров по типам. В связи с этой особенностью языка C++ проверка соответствия типов формальных и фактических параметров выполняется на этапе компиляции.

Строгое согласование по типам между параметрами требует, чтобы в модуле программы до первого обращения к функции было помещено либо ее определение, либо ее описание (прототип), содержащее сведения о типе результата и о типах всех параметров.

Прототип (описание) функции может внешне почти полностью совпадать с заголовком определения функции:

<тип функции> <имя функции>
(<спецификация формальных параметров>);

Отличие описания от заголовка определения функции состоит в следующем:

- наличие '**;**' в конце описания – это основное отличие и
- необязательность имен параметров, достаточно через запятые перечислить типы параметров.

Переменные, доступные функции

1) *локальные переменные:*

- объявлены в теле функции, доступны только в теле функции;
- при определении переменной ей выделяется память в *сегменте стека*, при завершении выполнения функции память освобождается;

2) *формальные параметры:*

- объявлены в заголовке функции и доступны только функции;
- формальные параметры за исключением параметров – ссылок являются локальными переменными, память им выделяется в *стеке*;

- параметр – ссылка доступен только функции, но он не является переменной, на него не выделяется память, это некоторая абстракция для обозначения внешней по отношению к функции переменной;

3) *глобальные переменные:*

- переменные объявлены в программе как внешние, т.е. вне всех функций, включая и главную функцию ***main*;**
- чтобы глобальная переменная была доступна функции, функция не должна содержать локальных переменных и формальных параметров с тем же именем; локальное имя "закрывает" глобальное и делает его не доступным;

Оператор return

Оператор ***return*** **-** оператор возврата ***управления программой и значения*** в точку вызова функции. С помощью этого оператора функция может вернуть одно скалярное значение любого типа. Форма оператора:

return (выражение);

- выражение определяет значение, возвращаемое функцией; выражение вычисляется, результат преобразуется к типу возвращаемого значения и передается в точку вызова функции;
- если функция не возвращает результата, оператор может, либо отсутствовать, либо присутствовать с пустым выражением: ***return*;**
- функция может иметь несколько операторов ***return*** с различными выражениями, если алгоритм функции предусматривает разветвление.

Функция завершается, как только встречается оператор ***return***. Если функция не возвращает результата, и не имеет оператора ***return*,** она завершается по окончанию тела функции.

Формальные и фактические параметры функции

Посредством формальных параметров осуществляется обмен данными между функцией и вызывающей ее программой. В функцию данные передаются для обработки. Функция, обработав эти данные, может вернуть в вызывающую функцию результат обработки.

В определении функции фигурируют формальные параметры, которые показывают, какие данные следует передавать в функцию при ее вызове, и как они будут обрабатываться операторами функции.

Список формальных параметров (список аргументов) функции указывает, с какими фактическими параметрами следует вызывать функцию.

Фактические параметры передаются в функцию при ее вызове, заменяя формальные параметры. Фактические параметры, по количеству, по типу (он должен быть идентичным или совместимым), по расположению должны соответствовать формальным параметрам.

Умалчиваемые значения параметров

Формальный параметр может содержать умалчиваемое значение. В этом случае при вызове функции соответствующий фактический параметр может быть опущен и умалчиваемое значение используется в качестве фактического параметра. При задании умалчиваемых значений должно соблюдаться правило.

Если параметр имеет умалчиваемое значение, то все параметры справа от него также должны иметь умалчиваемые значения.

Передача фактических параметров

В С++ передача параметров может осуществляться тремя способами:

− по значению, когда в функцию передается числовое значение параметра;

− по адресу, когда в функцию передается не значение параметра, а его адрес, что особенно удобно для передачи в качестве параметров массивов;

− по ссылке, когда в функцию передается не числовое значение параметра, а сам параметр и тем самым обеспечивается доступ из тела функции к объекту, являющемуся фактическим параметром.

Передача параметров по значению

Формальным параметром может быть только имя скалярной переменной стандартного типа или имя структуры, определенной пользователем.

При вызове функции формальному параметру выделяется память в стеке, в соответствии с его типом. Фактическим параметром является – выражение, *значение* которого копируется в стек, в область ОП, выделенную под формальный параметр. Фактическим параметром может быть просто неименованная константа нужного типа, или имя некоторой переменной, значение которой будет использовано как фактический параметр.

Все изменения, происходящие с формальным параметром в теле функции, не передаются переменной, *значение* которой являлось фактическим параметром функции.

Передача параметров по адресу - по указателю

Формальным параметром является указатель *type*p* на переменную типа *type*. При вызове функции формальному параметру-указателю выделяется память в стеке - 2 байта, если указатель ближний и 4 байта, если указатель дальний. Фактическим параметром может быть либо адрес в ОП переменной типа *type*, либо значение другого указателя, типа *type** из вызывающей программы. В область памяти (в стеке), выделенную для указателя *p* будет копироваться значение некоторого *адреса* из вызывающей функции.

В теле функции, используя операцию разыменования указателя **p*, можно получить доступ к участку памяти, адрес которого получил *p* при вызове функции, и изменить содержимое этого участка.

Если в теле функции изменяется значение **p*, при вызове функции эти изменения произойдут с тем объектом вызывающей программы, адрес которого использовался в качестве фактического параметра.

Передача параметров по ссылке

В языке С++ ссылка определена как другое имя уже существующей переменной. При определении ссылки оперативная память не выделяется Инициализатор, являющийся обязательным атрибутом определения ссылки, представляет собой имя переменной того же типа, имеющей место в памяти. Ссылка становится синонимом этой переменной.

*type & имя ссылки = имя переменной***;**

Основные достоинства ссылок проявляются при работе с функциями.

Если определить ссылку *type&a* и не инициализировать ее, то это равносильно созданию объекта, который имеет имя, но не связан ни с каким участком памяти. Это является ошибкой.

Однако такое определение допускается в спецификациях формальных параметров функций. Если в качестве формального параметра функции была определена неинициализированная ссылка - некоторая абстрактная переменная, которая имеет имя, но не имеет адреса, в качестве фактического параметра при вызове функции следует использовать имя переменной из вызывающей программы того же типа, что и ссылка. Эта переменная инициализирует ссылку, т. е. связывает ссылку с участком памяти.

Таким образом, ссылка обеспечивает доступ из функции непосредственно к внешнему участку памяти той переменной, которая при вызове функции будет использоваться в качестве фактического параметра.

Все изменения, происходящие в функции со ссылкой, будут происходить непосредственно с переменной, являющейся фактическим параметром.

Это наиболее перспективный метод передачи параметров, так как в этом случае вообще не происходит копирование фактического параметра в стек, будь то значение или адрес, функция непосредственно оперирует с внешними по отношению к ней переменными, используемыми в качестве фактических параметров.

3.2. Классификация формальных параметров

***Формальный параметр скаляр, указатель на скаляр, ссылка на скаляр*.**

В качестве скаляра будут рассматриваться переменные основных типов, элементы массивов или структур.

Если формальными параметрами функции являются:

а) скаляры;
б) указатели на скаляры;
в) ссылки на скаляры;
то фактическими параметрами должны быть:
а) значения скаляров;
б) адреса скаляров;
в) имена скаляров (имена участков памяти).

Формальные параметры – массивы

Массив в качестве фактического параметра может быть передан в функцию только по адресу, то есть с использованием указателя.

I) Формальный параметр- определение массива:

В качестве формальных параметров можно использовать:

1) определение массива с фиксированными границами, например:
   float A[5], int B[3][4][2], char S [25];
2) определение одномерного символьного массива с открытой границей:

char S1[];

При работе со строками, то есть с одномерными массивами данных типа ***char,*** последний элемент которых имеет известное значение - ***'\0',*** нет необходимости передавать размеры массива;

3) определение числового или многомерного символьного массива с открытой левой границей и параметр для передачи размера левой границы:

float A[ ], int B[ ][4][2], char S2 [ ][60], и **int n**

При всех этих определениях в стеке выделяется оперативная память на один указатель для передачи в функцию адреса нулевого элемента массива – фактического параметра.

Массив, адрес которого будет использован при вызове функции, как фактический параметр может быть изменен за счет выполнения операторов функции.

Если формальный параметр - массив с фиксированными границами <u>как в первом случае</u>, фактическим параметром будет имя массива из вызывающей программы с теми же фиксированными границами или же значение указателя на такой фиксированный массив.

При использовании <u>второго определения</u> фактическим параметром будет имя символьного массива, или указателя на первый элемент массива.

При использовании <u>третьего определения</u> фактическими параметрами будут имя массива из вызывающей программы, но уже с произвольным количеством элементов по первому измерению и размер массива по первому измерению.

Проиллюстрируем сказанное на примере определения функции, заполняющей трехмерный массив – параметр последовательными натуральными числами, начиная с единицы и возвращающей по ссылке сумму его элементов:

```
const int n=4, k =5;
float f ( float a [ ] [n][k] , int m, float & s )
{s = 0; int t =1;
for ( int i=0; i< m; i++) for( int j=0; i< n; j++) for ( int l=0; l< k; l++)
  { a[i][j][l]=t++;   s+= a[i][j][l];}}
void main ( )
{ float s, dat [3][4][5];
f(dat, 3, s);  //вызов функции заполняющей массив dat
cout << s;}
```

II) Формальный параметр - указатель:

1) В качестве формального параметра определяется указатель на первый элемент массива любой мерности и второй параметр – общее количество элементов в массиве:

type*p,   int n.

Фактическими параметрами в этом случае будут – указатель типа ***type****, значение которого - адрес первого элемента массива и второй параметр - значение общего количества элементов в массиве. При этом надо помнить, что имя массива любой мерности – это константный указатель, значением которого является адрес первого элемента массива, однако только для одномерного массива имя – есть указатель на элемент массива, для двумерного массива имя массива – это указатель на строку массива и так далее. Чтобы получить указатель на элемент массива для двумерного массива, надо разыменовать имя массива, для трехмерного массива – два раза разыменовать имя массива и так далее. Или решать проблему явным приведением типов указателей.

В качестве примера определим функцию, формирующую упорядоченный массив из элементов двух других массивов. Все три массива представим с помощью указателей на первый элемент массива. Используем еще два параметра для указания размеров исходных массивов. Размер результирующего массива является суммой количества элементов двух массивов.

```
#include<iostream.h>
void f (int*a, int*b, int*c, int n, int m)
{int i, j, p;
  for ( i=0; i < n+m; i++)          //формирование неупорядоченного массива
    if (i < n) c[i] = a[i];else c[i] = b[i-n];
  //упорядочивание массива методом пузырькового всплытия
  for (i = 0; i < n+m-1; i++)
    for (j = n+m - 1; j > i; j -- )
      if (c[j] < c[j-1]){p= c[j]; c[j] = c[j-1]; c[j-1] = p;}
}
int i, a [ ] = {7,9,5,4,0,2,89,33,73,11},
b [ ] = { 23,87,55,45,4,3,0,6,7,3},
n= sizeof( a ) / sizeof ( a[0]),
m= sizeof( b ) / sizeof ( b[0]);
//а)создаваемый массив – статический:
void main ( ){
int c [sizeof( a ) / sizeof ( a[0])+ sizeof( b ) / sizeof (b[0]) ];
// - выделена память на статический массив
f (a, b, c, n, m);
for(i = 0; i < n+m; i++)
cout << c[i] << " ";
}
//б) создается динамический массив:
void main ( )
{int*x = new int [n+m];// - выделена память на динамический массив
f (a , b , x , n , m ) ;
for ( i =0; i < n+m; i++)
cout << x[i] << " ";
```

delete [ ] x;
}

2) Формальные параметры - указатели на двумерный массив, размеры которого заранее не известны.

Параметрами являются указатель на массив указателей на строки матрицы: ***type**p*** и два значения целого типа: ***int m*** - количество этих строк и ***int n -*** количество элементов в каждой строке. В этом случае динамически выделяется память и на массив указателей на строки матрицы и на числа в этих строках, то есть производится полностью динамическое выделение памяти на двумерный массив.

В качестве примера определим функцию, заполняющую массив – параметр случайными числами:

```
void mas ( int** ptr, int m, int n)
{randomize( );
for (int i =0; i < m; i++)
for ( int j = 0; j < n; j++)
ptr[i][j] = rand( );
}
void main ( )
{float **ptr = new float* [m]; // динамическое выделение памяти
                                // на массив указателей на строки массива
for( i=0; i< m; i++ )
ptr[i] = new float [n]; //динамическое выделение памяти на строки массива
mas (ptr,5,6);         //вызов функции
for (i=0; i< m; i++) delete ptr[i]; //освобождение памяти на строки
delete [ ] ptr;        //освобождение памяти на массив указателей на строки
}
```

3) Формальные параметры указатели - для передачи многомерного массива. Параметрами являются указатель на подмассив и количество подмассивов в многомерном массиве. При этом размеры подмассива должны быть фиксированы.

<type> (* имя_указателя) [N1][N2] [N3]…[NN]; - определение указателя на массив размерностью ***[N1][N2]…[NN]*** с элементами ***type.***

Для иллюстрации определим функцию, заполняющую натуральными числами трехмерный массив, состоящий из произвольного числа матриц **3x4**:

```
void mas (int (*lp) [3][4], int n)
{ int i, j, k, t=1;
for (i=0; i < n; i++) {cout << "\n\n";
    for (j=0; j < 3; j++) {cout << "\n";
        for (k=0; k< 4; k++) { lp[i][j][k] = t++;cout <<lp [i][j][k] << " "; } } } }
void main ()
{ int i, j, k, n; cin >> n; //вводится количество матриц в трехмерном массиве
int (*tp) [3][4];          //определен указатель на матрицу
```

tp = new int [n] [3][4];
/* выделена динамическая память на ***n*** матриц, и адрес первой матрицы присвоен указателю ***tp*** */
mas (tp, n); //далее можно работать с элементами массива ***tp[i][j][k]***
delete [ ] tp; //освобождение памяти
}

3.3. Результат функции, возвращаемый с помощью оператора return

Оператор **return -** оператор возврата ***управления программой и значения*** в точку вызова функции. С помощью этого оператора функция может вернуть одно скалярное значение любого типа. Форма оператора:

return (выражение);

выражение вычисляется, результат преобразуется к типу возвращаемого значения и передается в точку вызова функции.

К скалярам относятся данные простых стандартных типов, указатели и ссылки. Рассмотрим некоторые возможные результаты работы функций, возвращаемые с помощью оператора ***return*****:**

1) скалярное значение любого простого типа, например ***return (5);***
2) указатель на скаляр, массив;
3) ссылка - возвращаемый результат функции

Результат указатель на скаляр, массив

Функция возвращает указатель на скаляр **-** возвращает адрес скаляра. Определим функцию поиска числа в произвольном массиве. Функция возвращает адрес числа или NULL, то есть указатель на элемент массива.

```
int * poisk ( int*c, int m, int n )
/* *c – указатель на первый элемент массива, m – количество элементов массива , n- искомое число*/
{for (int i =0; i < m; i++ ){ if (*c = = n) return ( c );c++;}
return (NULL);}
void main ( )
{ int a[5][4] = {…}, i, j;
int*p = poisk (*a, 20 ,7 ); //*a-адрес первого элемента, 20 - общее количество
                            //элементов, ищем число 7
if (p== NULL) cout << "число отсутствует";
else { cout<<(i= (p - *a ) / 4);   //номер строки
cout<<(j = p - *a – i * 4); };     //номер столбца
}
```

Результат функции – указатель на одномерный массив.

Используя указатель на элемент массива, можно в теле функции выделить память на одномерный динамический массив.

Этот указатель можно возвратить из функции с помощью оператора **return,** и тем самым возвратить "указатель на одномерный массив"**.**

Определим функцию, формирующую одномерный динамический массив, заполняя его значениями, вводимыми с клавиатуры. Формальный параметр функции – количество элементов в формируемом массиве. Функция возвращает указатель на первый элемент массива:

```
float* vvod ( int n)
{float*p = new int [n];            // динамическое выделение памяти под массив,
for( int i=0; i < n; i+ ) cin >> p[i];            //заполнение элементов массива
return p;}            //возвращение массива;
void main ( )
{float*dat, n;        cin >> n;     // вводится с клавиатуры количество элементов;
dat = vvod (n);            //указателю присваивается результат вызова функции;
for(int i=0; i < n; i++)    // поэлементный вывод на экран элементов массива
cout << dat[i] << "\t";
delete dat;}            // освобождение памяти;
```

Функция возвращает указатель на указатель на элемент массива

С помощью указателя на указатель на элемент массива в теле функции может быть сформирован двумерный динамический массив. Этот указатель на указатель можно быть возвращен как результат работы функции с помощью оператора ***return*****,** и тем самым возвращен "указатель на двумерный массив"**.**

В качестве иллюстрации определим функцию, возвращающую двумерный массив натуральных чисел, размеры массива передаются в функцию посредством параметров:

```
int** mas (int m, int n)
{int k=1;
int ** ptr= new int* [m ]; //выделение памяти на массив из m указателей на int;
for (int i=0; i < m; i++)
ptr [i] = new int [ n]; //выделение памяти, для каждой строки из n элементов
for (int i =0; i < m; i++) //заполнение массива и вывод его элементов;
{cout<<"\n"; for (int j = 0; j < n; j++) { ptr[ i] [j] = k++ ;cout<<ptr [i][j]<<" ";} }
return ptr;}            // возвращается "указатель на матрицу"
void main ( )
{int n , m ,i ,j;       cin>> n>>m;        // размеры массива вводятся с клавиатуры;
int ** Q = mas (m, n ); //указателю присваивается результат вызова функции;
…//Q[i][j] - обращение к элементам динамического двумерного массива
//освобождение памяти
for (i=0; i < m; i++)       delete Q[i];
delete [ ] Q; }
```

Результат функции - указатель на подмассив. Определим в качестве примера функцию, возвращающей значение указателя на подмассив **(4x5)** элементов типа **int**, которое является адресом первой матрицы трехмерного массива, сформированного в функции.

```
#include <iostream.h>
typedef int (*TPM)[4][5];//TPM - тип указателя на матрицу 4x5
```

```
TPM fun (int n)    //параметр функции – количество подмассивов в массиве
{int i, j, k, t=1;
TPM lp=new int [n][4][5]; // выделяем память на трехмерный массив
for (i=0; i < n; i++){cout << "\n\n";
    for (j=0; j < 4; j++){ cout << "\n";
        for (k=0; k< 5; k++){lp[i][j][k]=t++; cout <<lp [i][j][k] << " ";} } }
return lp;}
void main ()
{ int n, i, j , k; cin>>n;
TPM D=fun(n); //вызов функции
//D[i][j][k] - обращение к элементам трехмерного массива
delete []D; //освобождение памяти
}
```

Ссылка - возвращаемый результат функции

Ссылки не являются настоящими объектами, ссылка связана с участком памяти инициализирующего ее выражения.

Если функция возвращает ссылку, это означает, что функция должна возвращать не значение, а идентификатор для обращения к некоторому участку памяти, в простейшем случае имя переменной. Таким образом, в функции должен быть, например, такой оператор: ***return имя переменной;***

При этом следует помнить, что в операторе ***return*** не должно стоять имя локальной переменной, так как после вызова функции участки памяти, связанные в сегменте стека с локальными переменными становятся недоступными. Таким образом, в операторе должно стоять имя участка памяти из внешней программы.

Вызов такой функции представляет собой частный случай ***l-значения (l-value)***, которое представляет в программе некоторый участок памяти. Этот участок памяти может иметь некоторое значение и это значение можно изменять, например, в операторе присваивания.

В соответствие с этим, вызов такой функции может располагаться как в правой, так и в левой части оператора присваивания.

Следующая функция, возвращающая ссылку на элемент массива с максимальным значением, проиллюстрирует вышесказанное. Массив передается в функцию посредством параметра.

```
int& rmax (int d [ ], int n)
{int imax = 0;
for (int i =1; i< n; i++)
imax = (d[imax] > d[i] ? imax : i);
return (d[imax]);}
void main ()
{ int n =5, a [ ] = { 3, 7 , 21 , 33 , 6};
cout << rmax (n, a )<<endl;
rmax(n,a)=0;
```

```
for ( int i =0 ; i <n ; i++)
cout << a[i] << " " ;}
```

Результат программы:

33

3 7 21 0 6

Один из вызовов функции **rmax()** находится в левой части оператора присваивания, что позволяет занести в элемент новое значение.

3.4 Указатели на функции

Имя функции является указателем-константой на эту функцию. Значением этого указателя является адрес размещения операторов функции в оперативной памяти. Это значение адреса можно присвоить другому указателю-переменной на функцию с тем же типом результата и с той же сигнатурой параметров. И затем этот новый указатель можно применять для вызова функции.

Введем понятие указателя на функцию. Указатель на функцию – это некоторая переменная, значениями которой являются адреса функций (имена функций), характеристики которых (тип результата и сигнатура параметров) совпадают с характеристиками, указанными в определении указателя.

Определение указателя на функцию:

<тип_рез_ф> (* имя указателя) (спецификация_парам) =
< имя инициирующей функции>;

- при определении достаточно перечислить через запятую типы параметров, имена параметров можно опустить;
- ***тип_рез_ф*** – это тип результата, возвращаемого функцией;
- инициализация указателя не обязательна, но при ее наличии тип результата, и сигнатура параметров инициирующей функции должны быть такими же, как в определении указателя.

Например, определены два указателя:

```
int* (*fptr) (char*, int );          int (*ptr) (char*);
```

В примере указатели были определены без инициализации, но в дальнейшем этим указателям – переменным можно присвоить значения указателей – констант, а именно идентификаторы (имена) конкретных функций, спецификации которых должны полностью соответствовать спецификациям в определениях указателей.

Как только некоторому указателю присвоено имя функции, вызов этой функции можно производить, как используя имя функции, так и используя имя указателя на функцию.

Эквивалентные вызовы функции с помощью указателя на эту функцию:

имя указателя (список фактических параметров);

(*имя указателя) (список фактических параметров);

Рассмотрим на примере использование указателя на функцию. Определим функцию вычисления длины строки - параметр функции (количества символов в строке до байтового нуля, строка):

int len (char* e)
{int m=0;   while (e[m++]);return (m-1);}
void main ()
{int (*ptr) (char*);   //объявлен указатель на функцию без инициализации;
ptr = len;   //указателю присвоено значение – имя функции **len;**
char s [ ] = "rtgcerygw";
int n = ptr(s);}

Массивы указателей на функции

Указатели на функции могут быть объединены в массивы.

Ниже дано определение массива указателей на функции, возвращающие значение типа ***float*** и имеющие два параметра типа ***char*** и ***int***. В массиве с именем ***ptrArray*** четыре таких указателя:

float (*ptrArray) (char, int) [4];

Эквивалентное определение массива:

float (*ptrArray [4]) (char, int);

При объявлении массива можно провести инициализацию элементов массива указателей на функции именами соответствующих функций.

Пример определения массива указателей с инициализацией:

float v1 (char s, int n) {…}…
float v4 (char s, int n) {…}
float (*ptrr [4]) (char, int) = { v1, v2, v3, v4 };

В рассматриваемом примере даны определения четырех однотипных функций ***v1,…v4*** и определен массив ***ptrr*** из четырех указателей, которые инициализированы именами функций ***v1,…v4.***

Для того чтобы обратиться, например, к третьей из этих функции, можно использовать такие операторы:

float x = (*ptrr [2]) (‘a’ , 5);
float x = ptrr [2] (‘a’ , 5);

Для удобства последующих применений целесообразно вводить имя типа указателя на функцию с помощью спецификатора ***typedef:***

typedef <тип_функции> (*имя типа_указателя)
(спецификация_параметров);

Массивы указателей на функции удобно использовать при разработке программ, управление которыми выполняется с помощью меню, реализующих вызов различных функций обработки данных в интерактивном режиме.

Рассмотрим алгоритм программы простейшего меню:

- варианты обработки данных определяются в виде функций **art1() – act4();**
- объявляется тип **menu -** тип указателя на такие функции;

- объявляется массив **act** из четырех указателей на функции, инициированный именами функций **art1() – act4();**

***Интерактивная часть*:**

- на экран выводятся строки описания вариантов обработки данных и соответствующие вариантам целочисленные номера;
- пользователю предлагается выбрать из меню нужный ему пункт и ввести значение номера, соответствующее требуемому варианту обработки;
- пользователь вводит значение номера с клавиатуры;
- по номеру пункта, как по индексу, из массива указателей выбирается соответствующий элемент, инициированный адресом нужной функции обработки; производится вызов функции.

Использование массива указателей существенно упрощает программу, так как в данном случае отпадает необходимость использовать оператор **switch** – для выбора варианта.

Ниже приведена программа:

```
# include < iostream.h>
# include < stdlib.h>
// определение функций  обработки данных:
void act1 ( ) { cout << "чтение файла"; }
void act2 ( ) { cout << "модификация файла"; }
void act3 ( ) { cout << "дополнение  файла"; }
void act4( ) { cout << "удаление записей  файла"; }
typedef void ( * menu) ();          // дано описание типа указателя на функции;
menu act [4] = { act1, act2 , act3 , act 4};    //определен массив указателей;
void main ()
{int n;
cout << "\n1 - чтение файла";
cout << "\n2 - модификация файла";
cout << "\n3 - дополнение файла";
cout << "\n4 - удаление записей файла";
while (1) { cout << "\n введите номер"; cin >>n;
if ( n >= 1 && i<= 4) act [n-1] ( ); else exit(0);}}
```

Указатель на функцию - параметр функции

Указатели на функции удобно использовать в качестве параметров функций, когда объектами обработки функций должны служить другие функции.

В С++ все функции внешние, любую определенную функцию можно вызывать в теле любой другой функции непосредственно по имени, не передавая имя функции через механизм параметров.

Указатели на функции как параметры функций целесообразно использовать, когда в создаваемой функции должна быть заложена возможность обработки не конкретной, а произвольной функции. В этом случае адрес обрабатываемой функции целесообразно передавать в функцию

посредством параметра. В качестве формального параметра следует объявить указатель на функцию, а при вызове функции передавать в качестве фактического параметра идентификатор (адрес) нужной обрабатываемой функции.

Указатели на функции в качестве формальных параметров можно, например, использовать в функциях:

1) формирования таблиц результатов, получаемых с помощью различных функций (формул);
2) вычисления интегралов с различными подынтегральными функциями;
3) нахождения сумм рядов с различными общими членами и т. д.

Приведем пример: определить и вызвать функцию ***table()*** для построения таблицы значений различных функций. Функция ***table()*** использует в качестве параметров указатели на функции, которые определяют функции для вычислений значений в таблице.

Алгоритм задания:

- определяются три однотипных функции с одним вещественным параметром ***(a(x), b(x), c(x))*** для расчета значений, выводимых в таблицу;
- объявляется тип указателя ***func*** на такие функции;
- определяется массив ***S*** из трех указателей на функции инициированный именами функций ***a, b, c***;
- определяется функция ***table*,** выводящая в виде таблицы значения трех функций передаваемых **в *table*** посредством параметров; аргументами функции ***table*** являются:

  во-первых, массив ***ptrA*** указателей на функции с открытыми границами для передачи функций, вычисляющих значения и целочисленный параметр ***n*** для передачи количества указателей в массиве;

  и, во-вторых, параметры для аргумента функций – начальное значение - ***xn***, конечное значение - ***xk*** и шаг изменения аргумента - ***dx*;**
- в главной функции производится вызов функции ***table()*** и передаются фактические параметры – инициированный конкретными функциями массив ***S***, количество указателей в массиве - **3** и значения аргумента – начальное, конечное и шаг изменения аргумента.

Алгоритм функции table:

- устанавливается начальное значение аргумента функций ***x=xn*;**
- пока аргумент функций не достигнет своего конечного значения ***(x<= xk)*** выполняется повторяющаяся обработка: при каждом значении аргумента выводится строка значений трех функций, вызовы которых производятся с использованием указателей на функции из массива указателей и затем значение аргумента увеличивается на величину ***dx***.

Текст программы:

```
# include <iostream.h>
float a ( float x) { return x*x }
float b ( float x) { return (x*x +100) }
```

```
float c ( float x) { return sqrt ( fabs(x)) +x;}
typedef float (* func) ( float x) ;
func S [3] = {a, b, c}
//определение функции таблицы
void  table ( func ptrA [ ] , int n, float xn , float xk , float dx )
{float x = xn;
while ( x<= xk )
{cout << "\n";
for (int i=0; i< n; i++)
  { cout. width(10); cout << (* ptrA[i] ) (x);}
x+=dx ;} }
void main
{ table ( S, 3, 0., 2., 0.1 );}
```

Указатель на функцию может быть результатом работы функции, то есть функция может возвращать указатель на функцию с помощью оператора ***return*** и с помощью формального параметра (например, передача параметра по ссылке).

Рассмотрим это на примере:

```
typedef void ( * menu) ( )
menu act [4] = { act1, act2 , act3 , act 4};
menu F1 (int i) { return act [i] }
void F2 (int i , menu & r) { r = act [i] }
void main ( )
{ int i;         menu  p1 , p2;
while(1)      // бесконечный цикл;
{ cin >> i;
if ( i>=1 && i<=4 )
{p1 = F1 (i-1);      p1( );           // вызов функции;
F2(i-1, p2);p2 ( );           // вызов функции;
} else exit ( 0); } }
```

Ссылка на функцию

Подобно указателю на функцию определяется и ссылка на функцию:

<тип_функции> (& имя ссылки) (спецификация_параметров) <инициализирующее_выражение>;

здесь:

- ***<тип_функции>*** **-** тип возвращаемого функцией значения;
- ***спецификация_параметров*** – определяет сигнатуру функций, ***<инициализирующее_выражение>*** **-** обязательный элемент и является именем уже известной функции, имеющей тот же тип и ту же сигнатуру параметров, что и ссылка.

Ссылка на функцию является *синонимом (псевдонимом)* имени функции, обладает всеми правами основного имени функции.

3.5. Рекурсивные функции

Рекурсия – это способ организации обработки данных в функции, когда функция обращается сама к себе прямо или косвенно.

Функция называется косвенно рекурсивной, если она содержит обращение к другой функции, которая содержит прямой или косвенный вызов определяемой (первой) функции.

Если в теле функции явно используется вызов этой функции, то имеет место прямая рекурсия.

Рекурсивная форма алгоритма дает более компактный текст программы, но требует дополнительных затрат оперативной памяти для размещения данных и времени для рекурсивных вызовов функции.

Рекурсивный алгоритм позволяет повторять операторы тела функции многократно, каждый раз с новыми параметрами, конкурируя тем самым с итерационными методами (циклами). И также как и в итерационных методах, алгоритм должен включать условие для завершения повторения обработки данных, то есть иметь ветвь решения задачи без рекурсивного вызова функции.

Как было сказано выше, использование рекурсии не дает никакого практического выигрыша в программной реализации, и ее следует избегать, когда есть очевидное итерационное решение.

При выполнении рекурсивной функции происходит многократный ее вызов, при этом

- в стеке сохраняются значения всех локальных переменных и параметров функции для всех предыдущих вызовов, выделяется память для локальных переменных очередного вызова;
- переменным с классом памяти **extern** и **static** память выделяется один раз, которая сохраняется в течение всего времени программы;
- вызов рекурсивной функции происходит до тех пор, пока не будет получено конкретное значение без рекурсивного вызова функции.

Приведем примеры:

1) Определить функцию, возвращающую значение факториала целого числа. Факториал определятся только для положительных чисел формулой: $N!=1\cdot 2\cdot 3\cdot 4\cdot ...\cdot N$, и факториал нуля равен 1 (**0! =1).**

Определение факториала можно переписать в виде "математической рекурсии": $N!=N\cdot(N-1)!$**,** то есть факториал вычисляется через факториал. Соответственно определение функции имеет вид:

```
long fact ( int k)
{if ( k <0 ) return 0 ;
if ( k = = 0 ) return 1;              // здесь рекурсия прерывается;
return k * fact ( k- 1);}
```

2) Определить функцию, возвращающую целую степень вещественного числа. Определение **n**-й степени числа **X** можно представить в виде:

$$X^n = X * X^{n-1} \quad \text{при } n > 0,$$
$$X^n = X^{n+1} / X \quad \text{при } n < 0,$$

и в соответствии с этим определить рекурсивную функцию:

```
double step ( double X , int n )
{ if (n = = 0) return 1;     // здесь рекурсия прерывается;
if (X = = 0) return 0;
if (n > 0 ) return X * step(X , n-1 );
if (n < 0 ) return step(X , n+1 ) / X;}
```

3) Определить функцию, возвращающую сумму элементов массива. Запишем формулу для суммы **n** элементов в виде суммы последнего элемента и суммы первых **n-1** элементов**:**

$$S_n = a[n-1] + S_{n-1},$$

то есть сумма вычисляется через сумму. Соответственное определение рекурсивной функции:

```
int sum (int a[] , int n )
{ if (n = = 1) return ( a[0] );     // здесь рекурсия прерывается;
else return ( a[n-1] + sum (a, n-1) );}
```

3.6. Перегрузка функций. Шаблоны функций

Перегрузка функций

С++ позволяет определить в программе произвольное количество функций с одним именем, при условии, что все они имеют разный состав параметров, или *сигнатуру:*

```
void F (int);
void F (int, int);
void F (char*);
```

При вызове перегруженной функции компилятор анализирует ее сигнатуру и выбирает из списка одноименных функций ту функцию, сигнатура которой соответствует вызываемой.

При этом возвращаемый функцией тип значения не имеет, функции:

```
void F ( int),
int F (int)
```

не являются перегруженными, компилятор их не различает.

Для того чтобы различать одноименные функции компилятор использует кодирование имен функций (создает уточненные имена). Функциям даются имена, в которые входят имя класса, если эта функция компонентная, имя функции и список обозначений типов параметров.

```
class MyClass {…
void Func (int);          //@MyClass@Func$qi
void Func (int, int);     //@MyClass@Func$qii
void Func (char*);        //@ MyClass@Func$qpz
…};
```

Пример перегрузки глобальных функций:

```
#include<iostream.h>
void Print ( int i) { cout << "\n int =  " <<i;}
void Print ( int*pi ) { cout<<"\n pointer = " << pi << ", значение = " << *pi;}
void Print ( char*s) { cout << "\n string = " <<s;}
void main ()
{ int a=2;
int*b = &a;
char*c = "yyyy";
char d [] = "xxxx";
Print(a);    Print(b);    Print(c);    Print (d);}
```

Результат:

```
int=2
pointer =0x12340ffe , значение = 2
string = yyyy
string = xxxx
```

Перегрузка функций используется, как правило, когда отличаются в перегружаемых функциях и типы данных обрабатываемые функциями и отличается сам код обработки, но характер обработки имеет один и тот же смысл и целесообразно не придумывать разные названия функциям, а перегрузить функции для разных данных.

Если код функций совпадает, но отличаются обрабатываемые данные, то создается шаблон функций.

Шаблоны функций

Шаблон функций – это автоматизация создания функций, которые могут обрабатывать данные различных типов.

Шаблон семейства функций определяется один раз, но в это определение в качестве параметра, или параметров входят типы данных обрабатываемых функцией и возвращаемых ею. *Таким образом, шаблон обязан иметь параметры – типы данных.*

Шаблон состоит из двух частей – заголовка шаблона:

template <список параметров шаблона>
(<class имя_параметра1, class имя_параметра2, …>)
и определения функции, в котором используются параметры шаблона.

Параметры шаблона должны отвечать следующим требованиям:

1) имена параметров шаблона должны быть уникальными в определении шаблона;
2) список параметров не может быть пустым;
3) может быть несколько параметров

```
template<class T1,classT2,classT3>
```

все имена параметров должны быть разные, и обязательно перед каждым именем должно стоять слово – ***class;***
4) все параметры шаблона обязательно должны быть использованы в спецификации параметров функции.

Пример неправильного определения шаблона:

```
template<class T1,classT2,classT3>
T3 f ( T1 a, T2 b ) {T3 c;… } //ошибка!
```

Пример правильного определения шаблона функций обменивающих значения параметров:

```
template <class R>
void swap ( R&x, R&y)
{R t= x; x=y; y=t;}
```

Далее, если в программе встретится фрагмент :

```
double k= 3.7, n= 7.3;
swap(k, n);
```

компилятор на основе шаблона сформирует определение функции:

```
void swap ( double&x, double&y)
{double t=x; x=y; y=t;}
```

Затем будет выполнено обращение к ней и значения переменных ***k*** и ***n*** поменяются местами.

Шаблоны служат для автоматического формирования конкретных определений функций по тем вызовам, которые транслятор обнаруживает в тексте программы. В зависимости от вызовов создает определения функций обрабатывающих данные различных типов.

ЛИТЕРАТУРА

1. Страуструп Б. Язык программирования С++. – М.: Невский Диалект – БИНОМ, 1999 - 991 с.
2. Подбельский В.В. Стандартный Си++: учебное пособие – М.: Финансы и статистика, 2008, 688 с.
3. Подбельский В.В. Язык Си++. – М.: Финансы и статистика, 2000, 560 с.
4. Климова Л.М. Основы практического программирования на языке С++. –М.: Приор, 2000. - 454 с.
5. Страуструп Б. Дизайн и эволюция языка С++.- М.:ДМК Пресс, 2000 - 488 с.
6. Подбельский В.В Программирование на языке СИ: учебное пособие – М.: Финансы и статистика, 2003, 600 с.
7.Романов Е.Л. Практикум по программированию на языке С++: учебное пособие – СПб.: Петербург, 2004.- 432 с.

СОДЕРЖАНИЕ

1. Раздел 1. Базовые понятия языка С++ ………………………………………… 3
1.1. Введение в С++……………………………………………………….. 3
1.2. Лексические основы языка…………………………………………….. 12
1.3. Представление данных…………………………………………………. 27
1.4. Объекты программы и их атрибуты…………………………………… 32
1.5. Выражения и преобразования типов…………………………............. 37
1.6. Операторы языка С++…………………………………………............. 39
Раздел 2. Адреса, указатели, ссылки, массивы………………………………. 46
2.1. Указатели и адреса объектов…………………………………............. 46
2.2. Объявление указателей………………………………………………… 49
2.3. Типы указателей и операции над ними……………………………….. 50
2.4. Ссылки…………………………………………………………………... 53
2.5. Массивы…………………………………………………………………. 54
2.6. Массивы и указатели…………………………………………………… 57
2.7. Создание динамических массивов…………………………………….. 61
Раздел 3. Функции……………………………………………………………… 63
3.1. Определение, описание и вызовы функций…………………………... 63
3.2. Классификация формальных параметров…………………………….. 68
3.3.Результат функции, возвращаемый с помощью оператора return…............ 72
3.4. Указатели на функции………………………………………………… 75
3.5. Рекурсивные функции…………………………………………………. 80
3.6. Перегрузка функций. Шаблоны функций…………………………… 81
ЛИТЕРАТУРА………………………………………………………………… 83