

Оглавление

1.Основные принципы алгоритмизации и программирования. Алгоритмы и программы. Данные. Понятие типа данных. Логические основы алгоритмизации.....	5
2.Основные принципы алгоритмизации и программирования. Языки программирования: эволюция, классификация. Системы программирования. Файлы данных.	7
3.Основные принципы алгоритмизации и программирования. Объектно-ориентированный подход к программированию. Разработка программного обеспечения (ПО).....	9
4.Пакет компиляторов Visual C++. Рекомендуемое оборудование. Минимальные требования к аппаратному и программному обеспечению. Рекомендуемое аппаратное и программное обеспечение.	10
5. Пакет компиляторов Visual C++. Выбор правильных параметров установки. Какую конфигурацию выбрать?. Обычная установка под Windows. Каталоги.	11
6.Пакет компиляторов Visual C++. Система разработки. Новый встроенный отладчик. Новые встроенные редакторы ресурсов. Дополнительное средство TestContainer.	15
7.Пакет компиляторов Visual C++. Инструменты, не вошедшие в интегрированную среду. ProcessViewer (PView). WinDiff.	15
8.Пакет компиляторов Visual C++. Важные возможности компилятора. Р-код. Предварительно откомпилированные заголовки и типы. Библиотека MicrosoftFoundationClass. Встраивание функций..	16
9.Пакет компиляторов Visual C++. Параметры компилятора. General. Debug. CustomBuild.	18
10.Пакет компиляторов Visual C++. C/C++. C++ Language. CodeGeneration. Customization. ListingFiles. Optimizations. PrecompiledHeaders. Preprocessor.	18
11.Пакет компиляторов Visual C++. Link. General. Customization. Debug. Input. Output.....	19
12.Пакет компиляторов Visual C++. Resources. OLE Types. BrowseInfo.	20
13.Меню File (файл). New... Open... Close. Open Workspace. Close Workspace. Save. Save As.....	20
14.Меню File (файл). Save All. Find in Files... Page Setup... Print... Список последних проектов. Exit.....	21
15.Меню Edit. Undo. Redo. Cut. Copy. Paste. Delete. Select All.	22
16.Меню Edit. Find... Replace... Go To... InfoViewer Bookmarks... Bookmark. Breakpoints... Properties... ..	23
17.Меню View. ClassWizard... Resource Symbols... Resource Includes... Full Screen. Toolbars... InfoViewer Query Results. InfoViewer History List. Project Workspace.	23
18. Меню View. Info Viewer Topic. Output. Watch. Variables. Registers. Memory. Call Stack. Disassembly. Меню Insert.....	24
19.Меню Build. Compile. Build. Rebuild All. Batch Build... Stop Build. Update All Dependencies.	25
20.Меню Build. Debug. Execute. Settings... Configurations... Subprojects... Set Default Configuration... ..	26
21.Меню Tools. Browse... Close Browse Info File. OLE Control Test Container. OLE Object View.	26
22.Меню Window. New Window. Split. Hide. Cascade.	27
23.Меню Window. Tile Horizontally, Tile Vertically. Close All... Windows. Меню Help.	27
24.История языка C. Взаимоотношения с другими языками. Достоинства языка C. Малый размер. Набор команд языка. Быстродействие. Язык со слабой типизацией. Структурированный язык. Поддержка модульного программирования.	28

26.История языка C. Недостатки языка C. Слабая типизация. Отсутствие проверок на этапе исполнения. Использование языка Си. Будущее языка Си.	29
27.Исходные файлы и выполняемые файлы. Принципы программирования. Стандарт ANSI C. Эволюция языка C++ и объектно-ориентированное программирование. История C++. Использование объектов C++ для быстрого создания программы.	30
28.Исходные файлы и выполняемые файлы. Некоторые усовершенствования по сравнению с языком C. Комментарии. Имена перечисляемых типов. Имена структуры или класса. Блочные объявления. Операция уточнения области действия (scope). Описатель const. Анонимные объединения	31
29.Исходные файлы и выполняемые файлы. Явное преобразование типов. Объявления функций. Перегруженные функции. Значения параметров функций по умолчанию. Функции с неуказанным числом параметров. Ссылочные параметры функций. Операторы new и delete. Указатели void	32
30.Исходные файлы и выполняемые файлы. Основные усовершенствования по сравнению с языком C.(часто повторяется). Конструкторы классов и инкапсуляция данных. Класс struct. Конструкторы и деструкторы. Сообщения. "Дружественные" классы.	33
31. Исходные файлы и выполняемые файлы. Перегрузка операций. Производные классы. Полиморфизм при использовании виртуальных функций. Библиотеки потоков. Базовые элементы программы на C. Пять основных компонентов программы.	34
32.Написание и компиляция простых программ на C/C++. Написание вашей первой программы. Пример простой программы на C. Структура простой программы. Как сделать программу читаемой. Подготовка и компиляция простых программ на C/C++.	35
33.Написание и компиляция простых программ на C/C++. Редактирование текста программы. Сохранение программ. Построение программы.	37
34.Написание и компиляция простых программ на C/C++. Использование утилиты Project Workspace. Создание нового проекта. Добавление файлов к проекту. Запуск команд Build или Rebuild All.	38
35.Написание и компиляция простых программ на C/C++. Отладка программы. Понимание сообщений об ошибках и предупреждений. Распространенная ошибка при использовании нового языка. Переключение между окном вывода сообщений и окном редактирования. Использование функций замены или быстрого поиска. Выбор опций замены	40
36.Написание и компиляция простых программ на C/C++. Переключение между окном вывода сообщений и окном редактирования. Быстрый способ. Значение сообщений об ошибках и предупреждений. Повторная сборка программы ERROR.C. Запуск программы.	42
37.Написание и компиляция простых программ на C/C++. Использование встроенного отладчика. Использование команд пошагового выполнения (Step Into и Step Over). Определение точек останова (breakpoints). Запуск программы с точками останова. Использование быстрого просмотра	42
38.Данные. Идентификаторы. Ключевые слова. Символы. Данные: переменные и константы. Данные: типы данных.	44
39.Три целочисленных типа. Описание данных целого типа. Целые константы. Инициализация переменных целого типа. Модификатор unsigned.	45
40.Числа с плавающей точкой. Описание переменных с плавающей точкой. Перечисляемый тип данных (enum).	46
41.Модификаторы доступа. Модификатор const. Определение констант через #define. Модификатор volatile. Совместное использование const и volatile.	47
42. Модификаторы pascal, cdecl, near, far и huge. Модификатор pascal. Модификатор cdecl. Модификаторы near, far и huge.	48

43. Тип данных char Другие типы и размеры данных. Преобразование типов данных. Явные преобразования типов при помощи операции приведения типа.....	49
44. Символьные строки. Массивы символов в C++ Функции работы со строками символов.	51
45. Символьные строки. Определение длины строк Копирование и конкатенация строк Сравнение строк Преобразование строк	51
46.Символьные строки. Обращение строк Поиск символов Поиск подстрок.....	52
47.Функции преобразования типа Функции printf() и scanf(). Использование функции printf(). Модификаторы спецификации преобразования, используемые в функции printf(). Использование функции printf() для преобразования данных. Применение функции scanf().	53
48.Основные операции. Операция присваивания: =. Операция сложения: +. Операция вычитания: -. Операция изменения знака: -. Операция умножения: *. Операция деления: /.	55
49.Основные операции. Поразрядные операции. Поразрядное И (AND). Поразрядное ИЛИ (OR). Поразрядное исключающее ИЛИ (OR).	57
50. Основные операции. Поразрядный сдвиг влево и вправо. Операции отношения и логические операции. Условная операция (? :). Операция запятая (,). Порядок выполнения операций.	59
51.Дополнительные операции. Операция деления по модулю: %. Операции увеличения и уменьшения: ++. Операция уменьшения: --. Старшинство операций.	63
52. Выражения и операторы. Выражения. Операторы. Составные операторы (блоки).	64
53.Ввод и вывод одного символа: функции getchar() и putchar(). Буферы. Чтение данных. Чтение строки. Чтение файла.	65
54.Переключение и работа с файлами. Операционная система UNIX. Переключение вывода. Переключение ввода. Комбинированное переключение.	69
55. Выбор вариантов. Операции отношения и выражения. Понятие "истина".	70
56.Условные операторы. Оператор if. Оператор if-else. Вложенные операторы if-else.....	72
57.Условные операторы. Операторы if-else-if. Условный оператор ?. Оператор switch. Совместное использование операторов if-else-if и switch.	74
58.Оператор цикла. Цикл while. Завершение цикла while.	79
59.Оператор цикла. Цикл do-while. Цикл for.	80
60.Оператор цикла. Операция "запятая" в цикле for. Гибкость конструкции for. Философ Зенон и цикл for.	81
61.Оператор цикла. Вложенные циклы. Алгоритмы и псевдокод.	84
62.Оператор цикла. Оператор break. Оператор continue. Совместное использование операторов break и continue.	86
63.Оператор goto. Использование goto.	88
64.Оператор exit(). Оператор atexit(). Сравнение циклов.	90
65.Массивы. Понятие массив. Массивы в C. Объявление массивов. Доступ к элементам массива. Размещение массивов в памяти. Проблема ввода.....	93
66.Массивы. Инициализация массивов. Инициализация по умолчанию. Явная инициализация. Инициализация безразмерных массивов.....	94
67.Массивы. Инициализация массивов и классы памяти. Вычисление размера массива (sizeof()). Выход индекса за пределы массива.	95

68. Массивы. Многомерные массивы. Инициализация двумерного массива.....	96
69. Массивы. Массивы в качестве аргументов функций. Передача массивов функциям С. Передача массивов функциям С++.	98
70. Ввод и вывод строк. Строковые функции и символьные массивы. Динамическое выделение памяти. Функции gets(), puts(), fgets(), fputs() и sprintf(). Функции strcpy(), strcat(), strcmp() и strlen().	102
71. Указатели. Определение переменных-указателей. Разыменование указателей. Объявление переменных-указателей. Простые операторы с указателями. Инициализация указателей. Неправильное использование операции определения адреса.	104
72. Указатели. Указатели на массивы. Указатели и многомерные массивы. Указатели на указатели. Указатели на строки.	104
73. Указатели. Арифметические операции с указателями. Арифметические операции с указателями и массивы. Операции с указателями.	108
74. Указатели. Применение к указателям оператора sizeof. Сложности при использовании операций ++ и --. Сравнение указателей. Переносимость указателей. Использование функции sizeof() с указателями в среде DOS.	112
75. Указатели. Указатели на функции. Динамическая память. Использование указателей типа void. ..	117
76. Указатели. Указатели и массивы. Функции, массивы и указатели. Использование указателей при работе с массивами. Строки (массивы типа char).	121
77. Указатели. Массивы указателей. Дополнительная информация об указателях на указатели. Массивы указателей на строки.	127
78. Указатели. Ссылочный тип в С++ (reference type). Адрес в качестве возвращаемого значения функции. Передача параметров по ссылке и по значению. Использование встроенного отладчика. Использование ссылочного типа. Использование указателей и ссылок с ключевым.....	129
79. Использование функций. Создание и использование простой функции. Прототипы функций. Вызов по значению и вызов по ссылке. Использование указателей для связи между функциями.....	133
80. Использование функций. Рекурсия. Равноправность функций в языке Си. Параметры и аргументы функций. Формальные и фактические параметры.	142
81. Использование функций. Аргумент типа void. Символьные параметры. Целочисленные параметры.	158
82. Использование функций. Массивы в качестве параметров. Аргументы по умолчанию. Возвращение значения функцией: оператор return.	165
83. Использование функций. Типы функций. Функции типа void. Функции типа char. Функции типа int.	171
84. Использование функций. Аргументы функции main(). Строки. Целые числа. Числа с плавающей точкой.	177
85. Использование функций. Важные возможности С++. Встраивание (inline). Перегрузка (overloading). Многоточие (..).	186
86. Использование функций. Области видимости. Локальные и глобальные переменные Сложности в правилах области действия (scope rules). Неопределенные символы в программе на С. Использование переменной с файловой областью действия.	193
Глобальная переменная уступает локальной	197
87. Использование функций. Приоритет переменных с файловой и локальной областями действия. Проблемы области действия в С++. Операция уточнения области действия в С++.	201

1. Основные принципы алгоритмизации и программирования. Алгоритмы и программы. Данные. Понятие типа данных. Логические основы алгоритмизации.

Под **алгоритмом** понимали конечную последовательность точно сформулированных правил, которые позволяют решать те или иные классы задач. Такое определение алгоритма не является строго математическим, так как в нем не содержится точной характеристики того, что следует понимать под классом задач и под правилами их решения.

Свойства алгоритмов.

Каждое указание алгоритма предписывает исполнителю выполнить одно конкретное законченное действие. Исполнитель не может перейти к выполнению следующей операции, не закончив полностью выполнения предыдущей. Предписания алгоритма надо выполнять последовательно одно за другим, в соответствии с указанным порядком их записи. Выполнение всех предписаний гарантирует правильное решение задачи.

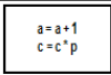

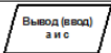
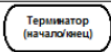
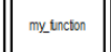



достижению цели. Разделение выполнения решения задачи на отдельные операции (выполняемые исполнителем по определенным командам) — важное свойство алгоритмов, называемое **дискретностью**.

Алгоритм, составленный для конкретного исполнителя, должен включать только те команды, которые входят в его систему команд. Это свойство алгоритма называется **понятностью**.

Еще одно важное требование, предъявляемое к алгоритмам, — **результативность (или конечность)** алгоритма. Оно означает, что исполнение алгоритма должно закончиться за конечное число шагов.

Формы записи алгоритмов

- записан на естественном языке (примеры записи алгоритма на естественном языке приведены при определении понятия алгоритма);
- изображен в виде блок-схемы;
- записан на алгоритмическом языке.

№	Символ	Наименование	Содержание
1		Блок вычислений	Вычислительные действия или последовательность действий
2		Логический блок	Выбор направления выполнения алгоритма в зависимости от некоторого условия
3		Блоки ввода-вывода данных	1. Общие обозначения ввода (вывода) данных (вне зависимости от физического носителя) 2. Вывод данных, носителем которых является документ
4		Начало (конец)	Начало или конец алгоритма, вход или выход в программу
5		Процесс пользователя (подпрограмма)	Вычисление по стандартной программе или подпрограмме
6		Блок модификации	Функция выполняет действия, изменяющие пункты (например, заголовок цикла) алгоритма
7		Соединитель	Указание связи прерванными линиями между потоками информации в пределах одного листа.
8		Межстраничные соединения	Указание связи между информацией на разных листах

Базовые структуры алгоритмов — это определенный набор блоков и стандартных способов их соединения для выполнения типичных последовательностей действий. К основным структурам относятся следующие: **линейные**, **разветвляющиеся**, **циклические**.

Линейными называются алгоритмы, в которых действия осуществляются последовательно друг за другом.

Разветвляющимся называется алгоритм, в котором действие выполняется по одной из возможных ветвей решения задачи, в зависимости от выполнения условий.

Циклическим называется алгоритм, в котором некоторая часть операций (тело цикла — последовательность команд) выполняется многократно.

Данные — это любая информация, представленная в формализованном виде и пригодная для обработки алгоритмом.

Данные делятся на **переменные** и **константы**.

Переменные — это такие данные, значения которых могут изменяться в процессе выполнения алгоритма.

Константы — это данные, значения которых не меняются в процессе выполнения алгоритма.

Типы данных принято делить на **простые (базовые)** и **структурированные**.

К основным базовым типам относятся:

- **целый (INTEGER)** — определяет подмножество допустимых значений из множества целых чисел;
- **вещественный (REAL, FLOAT)** — определяет подмножество допустимых значений из множества вещественных чисел;
- **логический (BOOLEAN)** — множество допустимых значений — истина и ложь;
- **символьный (CHAR)** — цифры, буквы, знаки препинания и пр.

Структурированные типы описывают наборы однотипных или разнотипных данных, с которыми алгоритм должен работать как с одной именованной переменной.

Наиболее широко известная структура данных — **Массив**. **Массив** -представляет собой упорядоченную структуру однотипных данных, которые называются элементами массива.

Логические основы алгоритмизации.

Важной составляющей алгоритмов являются логические условия. Вычисление значений логических условий происходит в соответствии с аксиомами алгебры логики. Алгебра логики используется при построении основных узлов ЭВМ — шифратора, дешифратора, сумматора. Высказывания принято обозначать большими буквами латинского алфавита: A, B, C ... X, Y и т.д. Если высказывание C истинно, то пишут $C = 1$ ($C = t$, true), а если оно ложно, то $C = 0$ ($C = f$, false).

Для образования новых высказываний наиболее часто используются логические операции, выражаемые словами «не», «и», «или».

Конъюнкция (логическое умножение). Соединение двух (или нескольких) высказываний в одно с помощью союза И (OR) называется операцией, логического умножения, или конъюнкцией.

Объединение двух (или нескольких) высказываний с помощью союза ИЛИ (OR) называется операцией **логического сложения, или дизъюнкцией**. Эту операцию обозначают знаками « \vee », « \vee » или знаком

сложения «+». Сложное высказывание $A \vee B$ истинно, если истинно хотя бы одно из входящих в него высказываний.

Присоединение частицы НЕ (NOT) к данному высказыванию называется **операцией отрицания (инверсии)**. Она обозначается \bar{A} (или $\neg A$) и читается не А. Если высказывание А истинно, то В ложно, и наоборот.

2. Основные принципы алгоритмизации и программирования. Языки программирования: эволюция, классификация. Системы программирования. Файлы данных.

В развитии инструментального программного обеспечения (т.е. программного обеспечения, служащего для создания программных средств в любой проблемной области) рассматривают пять поколений языков программирования (ЯП). Языки программирования как средство общения человека с ЭВМ от поколения к поколению улучшали свои характеристики, становясь все более доступными в освоении непрофессионалам.

Первое	Машинные	Ориентированы на использование в конкретной ЭВМ, сложны в освоении, требуют хорошего знания архитектуры ЭВМ
Второе	Ассемблеры, Макроассемблеры	Более удобны для использования, но по-прежнему машинно-зависимы
Третье	Языки высокого уровня	Мобильные, человеко-ориентированные, проще в освоении
Четвертое	Непроцедурные, объектно-ориентированные, языки запросов, параллельные	Ориентированы на непрофессионального пользователя и на ЭВМ с параллельной архитектурой
Пятое	Языки искусственного интеллекта, экспертных систем и баз знаний, естественные языки	Ориентированы на повышение интеллектуального уровня ЭВМ и интерфейса с языками

Вычислительная машина (система), независимо от типа и поколения, состоит из двух основных типов устройств:

- центральное устройство (ЦУ), включающее в себя центральный процессор (ЦП) и оперативную память (ОП);
- периферийные (внешние) устройства (ВУ).

Классификация ЯП. Изучение ЯП часто начинают с их классификации. Определяющие факторы классификации обычно жестко не фиксируются..

Фактор	Характеристика	Группы	Примеры ЯП
Уровень ЯП	Степень близости ЯП к архитектуре компьютера	Низкий	Автокод, ассемблер
		Высокий	Fortran, Pascal, ADA, Basic, C и др. ЯВУ
		Сверхвысокий	Сетл
Специализация ЯП	Потенциальная или реальная область применения	Общего назначения (универсальные)	Algol, PL/I, Simula, Basic, Pascal
		Специализированные	Fortran (инженерные расчеты), Cobol (коммерческие задачи), Refal, Lisp (символьная обработка), Modula, Ada (программирование в реальном времени)
Алгоритмичность (процедурность)	Возможность абстрагироваться от деталей алгоритма решения задачи. Алгоритмичность тем выше, чем точнее приходится планировать порядок выполняемых действий	Непроцедурные	Prolog, Langin
		Процедурные	Ассемблер, Fortran, Basic, Pascal, Ada

Системы программирования – это комплекс инструментальных программных средств, предназначенных для работы с программами на одном из языков программирования. Системы

программирования представляют сервисные возможности программистам для разработки их собственных компьютерных программ.

Системы программирования классифицируются по признакам, приведенным в таблице.

Признак классификации	Типы
Набор исходных языков	Одноязыковые
	Многоязыковые
Возможности расширения	Замкнутые
	Открытые
Трансляция	Компиляция
	Интерпретация

Следует отметить, что:

- отличительной особенностью многоязыковых систем является то, что отдельные части (секции, модули или сегменты) программы могут быть подготовлены на различных языках и объединены во время или перед выполнением в единый модуль;
- в открытую систему можно ввести новый входной язык с транслятором, не требуя изменений в системе;
- в интерпретирующей системе осуществляется покомандная расшифровка и выполнение инструкций входного языка (в среде данной системы программирования); в компилирующей — подготовка результирующего модуля, который может выполняться на ЭВМ практически независимо от среды.

Файлы данных.

Ввод-вывод данных в языках программирования осуществляется путем взаимодействия программы с **внешними файлами**. **Внешний файл** — это поименованный файл на диске или устройство ввода-вывода (например, клавиатура или дисплей). Во внешних файлах сохраняются результаты работы программы и располагаются данные, служащие источником информации, необходимой для ее функционирования. **Файл** можно определить как логически непрерывный именованный набор данных на внешнем носителе.

Могут быть рассмотрены следующие типы файлов:

по типу записей:

- файлы с записями фиксированной длины;
- файлы с записями переменной или неопределенной длины;
- файлы, образующие байтовый или битовый поток;

по способу выборки информации:

- файлы последовательного доступа;
- файлы прямого доступа.

Файлы последовательного доступа. Рассмотрим вначале файлы последовательного доступа. Для таких файлов характерны операции последовательного чтения и записи в конец файла

Файлы прямого доступа. Для файлов прямого доступа характерны операции чтения и записи по произвольному адресу.

3. Основные принципы алгоритмизации и программирования. Объектно-ориентированный подход к программированию. Разработка программного обеспечения (ПО).

Каждое новое достижение в аппаратном либо в программном обеспечении приводит к попыткам расширить сферу применения ЭВМ, что влечет за собой постановку новых задач, для решения которых, в свою очередь, нужны новые вычислительные средства. Одной из первых и наиболее широко применяемых технологий программирования стало **структурное программирование**.

Структурный подход базируется на двух основополагающих принципах:

- использование процедурного стиля программирования;
- последовательная декомпозиция алгоритма решения задачи сверху вниз.

Объектно-ориентированное программирование родилось и получило широкое распространение именно благодаря попыткам разрешения следующих проблем, возникавших в процессе проектирования и разработки программных комплексов.

1. Развитие языков и методов программирования не успевало за все более растущими потребностями в прикладных программах. Единственным реальным способом снизить временные затраты на разработку был метод многократного использования разработанного программного обеспечения, т.е. проектирование новой программной системы на базе разработанных и отлаженных ранее модулей, которые выступают в роли своеобразных «кирпичиков», лежащих в Фундамент новой разработки.
2. Ускорение разработки программного обеспечения требовало решения проблемы упрощения их сопровождения и модификации.
3. Не все задачи поддаются алгоритмическому описанию по требованиям структурного программирования, поэтому в целях упрощения процесса проектирования необходимо было решить проблему приближения структуры программы к структуре решаемой задачи.

ПО — сложная, достаточно уникальная система. Однако существуют некоторые общие принципы, которые следует использовать при разработке ПО.

Частотный принцип. Основан на выделении в алгоритмах и в обрабатываемых структурах действий и данных по частоте использования. Для действий, которые часто встречаются при работе ПО, обеспечиваются условия их быстрого выполнения. К данным, которым происходит частое обращение, обеспечивается наиболее быстрый доступ, а подобные операции стараются сделать более короткими.

Принцип модульности. Под модулем в общем случае понимают функциональный элемент рассматриваемой системы, имеющий оформление, законченное и выполненное в пределах требований системы, и средства сопряжения с подобными элементами или элементами более высокого уровня данной или другой системы.

Принцип функциональной избирательности. Этот принцип является логическим продолжением частотного и модульного принципов и, используется при проектировании ПО, объем которого существенно превосходит имеющийся объем оперативной памяти.

Принцип генерируемости. Данный принцип определяет такой способ исходного представления ПО, который бы позволял осуществлять настройку на конкретную конфигурацию технических средств, круг решаемых проблем, условия работы пользователя.

Принцип функциональной избыточности. Этот принцип учитывает возможность проведения одной и той же работы (функции) различными средствами. Особенно важен учет этого принципа при разработке пользовательского интерфейса для выдачи данных из-за психологических различий в восприятии информации.

Принцип «по умолчанию». Применяется для облегчения организации связей с системой, как на стадии генерации, так и при работе с уже готовым ПО. Принцип основан на хранении в системе некоторых базовых описаний структур, модулей, конфигураций оборудования и данных, определяющих условия работы с ПО. Эту информацию ПО использует в качестве заданной, если пользователь забудет или сознательно не конкретизирует ее. В данном случае ПО само установит соответствующие значения.

Общесистемные принципы. При создании и развитии ПО рекомендуется применять следующие общесистемные принципы:

- принцип включения, который предусматривает, что требования к созданию, функционированию и развитию ПО определяются со стороны более сложной, включающей его в себя системы;
- принцип системного единства, который состоит в том, что на всех стадиях создания, функционирования и развития ПО его целостность будет обеспечиваться связями между подсистемами, а также функционированием подсистемы управления;
- принцип развития, который предусматривает в ПО возможность его наращивания и совершенствования компонентов и связей между ними;
- принцип комплексности, который заключается в том, что ПО обеспечивает связность обработки информации, как отдельных элементов, так и для всего объема данных в целом на всех стадиях обработки;
- принцип информационного единства, определяющий, что во всех подсистемах, средствах обеспечения и компонентах ПО используются единые термины, символы, условные обозначения и способы представления;
- принцип совместимости, который состоит в том, что язык, символы, коды и средства обеспечения ПО согласованы, обеспечивают совместное функционирование всех его подсистем и сохраняют открытой структуру системы в целом;
- принцип инвариантности, который предопределяет, что подсистемы и компоненты ПО инвариантны к обрабатываемой информации, т.е. являются универсальными или типовыми.

4.Пакет компиляторов Visual C++. Рекомендуемое оборудование. Минимальные требования к аппаратному и программному обеспечению. Рекомендуемое аппаратное и программное обеспечение.

Пакет Microsoft Visual C++ включает в себя средства для построения программ для Windows. Кроме этого, ваш код может быть адаптирован для таких аппаратных платформ, как Apple Macintosh и машин с RISC-процессорами. В пакете C++ имеются все необходимые заголовочные файлы, библиотеки, редакторы окон диалога и ресурсов, необходимые для создания действительно надежного приложения

Windows. Microsoft также включила непосредственно в среду разработки редакторы ресурсов для пиктограмм, растровых изображений, курсоров, меню и окон диалога.

Минимальные требования к аппаратному и программному обеспечению

Стандартный пакет компиляторов Microsoft Visual C++ работает на обширном семействе компьютеров с процессорами Intel. Существуют специальные версии Visual C++ для компьютеров Macintosh, MIPS и DEC Alpha AXP.

Ниже перечислены минимальные требования Microsoft к оборудованию и программному обеспечению для работы 16- и 32-разрядной версии Microsoft Visual C++:

- ПК с процессором 80386
- 16MB ОЗУ (минимум)
- 10MB дискового пространства для минимальной установки
- Microsoft Windows 95 или Windows NT для разработки приложений Win32
- Дисплей VGA
- Дисковод для дискет высокой плотности
- Привод CD-ROM (для электронной документации)
- Мышь Microsoft

Рекомендуемое аппаратное и программное обеспечение.

Для оптимизации цикла разработки программ на C и C++ мы рекомендуем следующую конфигурацию системы:

- ПК с процессором Pentium, тактовая частота 90 МГц (или выше)
- 20MB ОЗУ
- 100MB свободного дискового пространства
- Microsoft Windows 95 или Windows NT для разработки приложений Win32
- Дисплей Super VGA
- Дисковод для 3.5-дюймовых дискет высокой плотности
- Привод CD-ROM (для электронной документации)
- Мышь Microsoft

- Процессор с частотой от 1,6 ГГц.
- ОЗУ 1 Гб (1,5 Гб для работы на виртуальной машине)
- 10 Гб доступного пространства на жестком диске (частота вращения жёсткого диска от 5400 об/мин.
- Видеокарта с поддержкой DirectX9
- Разрешение экрана от 1024x768

Требования к системе:

- В Windows 8.1 и Windows Server 2012 R2 для Visual Studio 2015 нужно обновление 291935 (также доступно через центр обновления Windows)

5. Пакет компиляторов Visual C++. Выбор правильных параметров установки. Какую конфигурацию выбрать?. Обычная установка под Windows. Каталоги.

Пакет Microsoft Visual C++ включает в себя средства для построения программ для Windows. Кроме этого, ваш код может быть адаптирован для таких аппаратных платформ, как Apple Macintosh и машин с RISC-процессорами. В пакете C++ имеются все необходимые заголовочные файлы, библиотеки, редакторы окон диалога и ресурсов, необходимые для создания действительно надежного приложения Windows. Microsoft также включила непосредственно в среду разработки редакторы ресурсов для пиктограмм, растровых изображений, курсоров, меню и окон диалога.

Установите Visual Studio 2017

Добро пожаловать на новый способ установки Visual Studio! В нашей новейшей версии мы упростили вам выбор и установку только тех функций, которые вам нужны. Мы также уменьшили минимальный размер Visual Studio, чтобы он устанавливался быстрее и с меньшим воздействием системы, чем когда-либо прежде.

Хотите узнать больше о том, что нового в этой версии? См. Наши примечания к выпуску .
Готов к установке? Мы пройдем через это, шаг за шагом.

Шаг 1 - Убедитесь, что ваш компьютер готов для Visual Studio

Прежде чем приступать к установке Visual Studio:

1. Проверьте системные требования . Эти требования помогут вам узнать, поддерживает ли ваш компьютер Visual Studio 2017.
2. Примените последние обновления Windows. Эти обновления гарантируют, что на вашем компьютере установлены последние обновления безопасности и необходимые системные компоненты для Visual Studio.
3. Перезагружать. Перезагрузка гарантирует, что любые ожидающие установки или обновления не будут препятствовать установке Visual Studio.
4. Освободите место. Удалите ненужные файлы и приложения из вашего % SystemDrive%, например, запустив приложение «Очистка диска».

Вопросы о работе предыдущих версий Visual Studio рядом с Visual Studio 2017 см. В сведениях о совместимости Visual Studio .

Шаг 2 - Загрузить Visual Studio

Затем загрузите файл начальной загрузки Visual Studio. Для этого щелкните следующую кнопку, выберите версию Visual Studio 2017, которую вы хотите, нажмите « **Сохранить** », а затем « **Открыть папку** » .

Шаг 3 - Установите установщик Visual Studio

Затем запустите файл bootstrapper, чтобы установить установщик Visual Studio. Этот новый легкий установщик включает все необходимое для установки и настройки Visual Studio 2017.

1. В папке « **Загрузка** » дважды щелкните загрузочный файл, соответствующий или похожий на один из следующих файлов:
 - **vs_enterprise.exe** для Visual Studio Enterprise

- **vs_professional.exe** для Visual Studio Professional
- **vs_community.exe** для сообщества Visual Studio

Если вы получили уведомление об управлении учетными записями пользователей, нажмите « Да » .

2. Мы попросим вас подтвердить условия лицензии Microsoft и заявление о конфиденциальности Microsoft . Нажмите « **Продолжить** » .

Шаг 4 - Выбор рабочих нагрузок

После установки установщика вы можете использовать его для настройки своей установки, выбрав нужные вам наборы функций или рабочие нагрузки. Вот как.

1. Найдите требуемую рабочую нагрузку на экране « **Установка Visual Studio** » .

Например, выберите рабочую нагрузку «.NET desktop development». Он поставляется с основным редактором ядра, который включает в себя базовую поддержку редактирования кода на более чем 20 языках, возможность открывать и редактировать код из любой папки без необходимости проекта и интегрированного управления исходным кодом.

2. После того, как вы выберете требуемую рабочую нагрузку (ы), нажмите « **Установить** » .

Затем появятся экраны состояния, показывающие ход установки Visual Studio.

3. После установки новых рабочих нагрузок и компонентов нажмите « **Запустить** » .

Наконечник

В любое время после установки вы можете установить рабочие нагрузки или компоненты, которые вы не установили изначально. Если вы открыли Visual Studio, откройте « **Инструменты** » > « **Получить инструменты и функции ...** », который открывает установщик Visual Studio. Или откройте **установщик Visual Studio** в меню «Пуск». Оттуда вы можете выбрать рабочие нагрузки или компоненты, которые вы хотите установить, затем нажмите « **Изменить** » .

Шаг 5 - Выберите отдельные компоненты (необязательно)

Если вы не хотите использовать функцию Workloads для настройки вашей Visual Studio, вы можете сделать это, установив вместо этого отдельные компоненты. Чтобы выбрать отдельные компоненты, выберите параметр « **Отдельные компоненты** » в установщике Visual Studio, выберите нужное, а затем следуйте инструкциям.

Шаг 6 - Установите языковые пакеты (необязательно)

По умолчанию программа установки пытается совместить язык операционной системы, когда она запускается в первый раз. Чтобы установить Visual Studio 2017 на выбранном вами языке, выберите параметр « **Языковые пакеты** » из установщика Visual Studio и следуйте инструкциям.

Измените язык установщика из командной строки

Другой способ, с помощью которого вы можете изменить язык по умолчанию, - запустить установщик из командной строки. Например, вы можете заставить программу установки работать на английском языке, используя следующую команду: `vs_installer.exe --locale en-US`. Установщик помнит этот

параметр, когда он будет запущен в следующий раз. Установщик поддерживает следующие языковые токены: zh-cn, zh-tw, cs-cz, en-us, es-es, fr-fr, de-de, it-it, ja-jp, ko-kr, pl-pl, pt-br, ru-ru и tr-tr.

Шаг 7 - Начните разработку

1. После завершения установки Visual Studio нажмите кнопку « **Запустить** » , чтобы начать работу с Visual Studio .
2. Нажмите « **Файл** » , а затем « **Новый проект** » .
3. Выберите тип проекта.

Например, чтобы создать приложение C++ , нажмите « **Установлено** » , разверните **Visual C++** и выберите тип проекта C++ , который вы хотите создать.

Чтобы создать приложение C# , нажмите « **Установить** » , разверните **Visual C#** и выберите тип проекта C# , который вы хотите создать.

Получать поддержку:

Иногда все может пойти не так. Если ваша установка Visual Studio не удалась, см. Страницу « Устранение неполадок с Visual Studio 2017 » и « Проблемы с обновлением » . Если ни один из шагов по устранению неполадок не помогает, вы можете связаться с нами через чат для помощи по установке (только на английском языке). Подробнее см. На странице поддержки Visual Studio .

Вот еще несколько вариантов поддержки:

- Вы можете сообщить нам о проблемах с продуктом через инструмент « Сообщить о проблеме » , который появляется как в установщике Visual Studio, так и в среде Visual Studio.
- Вы можете поделиться предложением продукта с нами в UserVoice .
- Вы можете отслеживать проблемы продукта в сообществе разработчиков Visual Studio, задавать вопросы и находить ответы.
- Вы также можете общаться с нами и другими разработчиками Visual Studio через нашу беседу в Visual Studio в сообществе Gitter . (Для этой опции требуется учетная запись GitHub .)

Каталоги.

В таблице показана типичная группа подкаталогов для Visual C++ 4.0, установленного в каталог MSdev.

Название	Назначение
BIN	Исполняемые файлы и средства для построения 32-разрядных приложений
HELP	Файлы справки
INCLUDE	Заголовочные файлы C++
LIB	Библиотеки языка C++ и Win32 SDK
MFC	Файлы библиотеки Microsoft Foundation Classes (MFC)
OLE	Файлы для построения приложений OLE
SAMPLES	Примеры программ

Документация по Visual C++ состоит из Quick Reference (Быстрой справки) и Books Online (Электронных книг). Быстрая справка позволит вам быстро отыскать информацию в процессе программирования. Электронные книги представляют собой полный набор документации по Visual C++ в компьютерном формате.

В документации содержатся, в частности, следующие разделы:

- Как пользоваться Электронными книгами
- Различные Руководства пользователя
- Microsoft Foundation Classes (MFC)
- Программирование с использованием библиотеки MFC
- Справочник по библиотеке MFC
- Примеры по библиотеке MFC
- Технические указания по библиотеке MFC/C++

6.Пакет компиляторов Visual C++. Система разработки. Новый встроенный отладчик. Новые встроенные редакторы ресурсов. Дополнительное средство TestContainer.

Пакет Microsoft Visual C++ включает в себя средства для построения программ для Windows. Кроме этого, ваш код может быть адаптирован для таких аппаратных платформ, как Apple Macintosh и машин с RISC-процессорами. В пакете C++ имеются все необходимые заголовочные файлы, библиотеки, редакторы окон диалога и ресурсов, необходимые для создания действительно надежного приложения Windows. Microsoft также включила непосредственно в среду разработки редакторы ресурсов для пиктограмм, растровых изображений, курсоров, меню и окон диалога.

Система разработки.

32-разрядный компилятор Microsoft Visual C++ для Windows NT и Windows 95 вобрал в себя новейшие средства разработки для Windows, тесно связанные друг с другом и снабженные наглядным интерфейсом. Microsoft использовала всю мощь своего отладчика CodeView непосредственно в среде Visual C++. Отладчик доступен из меню Build (сборка). Встроенный отладчик позволит вам выполнять программу в пошаговом режиме, считывать и изменять содержимое переменных, и даже двигаться назад по коду. Доступ к этим редакторам осуществляется из меню Insert (вставить). Редакторы ресурсов позволяют проектировать и создавать ресурсы Windows: растровые изображения, курсоры, значки, меню и окна диалога.

Test Container (тестовый контейнер) — это спроектированное Microsoft приложение, которое позволяет быстро тестировать ваши управляющие элементы. Можно изменять свойства и возможности элемента, находящегося в тестовом контейнере.

Отдельные инструменты расположены вне интегрированной среды разработки (Integrated Development Environment, IDE). Некоторые из них, как например Spy++ и MFC Tracer, доступны и внутри интегрированной среды, и вне ее.

7.Пакет компиляторов Visual C++. Инструменты, не вошедшие в интегрированную среду. ProcessViewer (PView). WinDiff.

Пакет Microsoft Visual C++ включает в себя средства для построения программ для Windows. Кроме этого, ваш код может быть адаптирован для таких аппаратных платформ, как Apple Macintosh и машин с RISC-процессорами. В пакете C++ имеются все необходимые заголовочные файлы, библиотеки, редакторы окон диалога и ресурсов, необходимые для создания действительно надежного приложения Windows. Microsoft также включила непосредственно в среду разработки редакторы ресурсов для пиктограмм, растровых изображений, курсоров, меню и окон диалога.

Инструменты, не вошедшие в интегрированную среду

Process Viewer (наблюдение за процессами) позволяет быстро устанавливать параметры, необходимые для отслеживания выполняемых процессов, потоков и квантования процессорного времени. Чтобы запустить Process Viewer, просто дважды щелкните на значке PView в группе Visual C++.

WinDiff.

Утилита WinDiff также находится в группе Visual C++. Это средство позволит вам в графическом виде сравнивать и изменять два файла или два каталога. Все возможности WinDiff работают очень сходно с соответствующими командами Windows 95 Explorer или Windows NT File Manager.

8. Пакет компиляторов Visual C++. Важные возможности компилятора. Р-код. Предварительно откомпилированные заголовки и типы. Библиотека MicrosoftFoundationClass. Встраивание функций.

Пакет Microsoft Visual C++ включает в себя средства для построения программ для Windows. Кроме этого, ваш код может быть адаптирован для таких аппаратных платформ, как Apple Macintosh и машин с RISC-процессорами. В пакете C++ имеются все необходимые заголовочные файлы, библиотеки, редакторы окон диалога и ресурсов, необходимые для создания действительно надежного приложения Windows. Microsoft также включила непосредственно в среду разработки редакторы ресурсов для пиктограмм, растровых изображений, курсоров, меню и окон диалога.

Важные возможности компилятора

Пакет компилятора Visual C++ включает множество усовершенствований, новых возможностей и дополнений. Следующие разделы представят вам эти улучшения и кратко пояснят их применение.

Р-код (сокращение от packed code — упакованный код) нацелен на оптимизацию размера и быстродействия кода. Р-код может существенно уменьшить размер программы и повысить скорость ее выполнения на величину до 60 процентов. Более того, все это достигается простым включением определенного режима компиляции. Это означает, что любая программа на C или C++ может компилироваться как обычным способом, так и с применением р-кода.

Visual C++ помещает родовые типы, прототипы функций, внешние ссылки и объявления функций-членов класса в специальные файлы, называемые заголовочными. Эти файлы содержат много важных определений, необходимых многочисленным исходным файлам, которые связываются воедино для создания исполняемой программы. Части заголовочных файлов, как правило, компилируются заново для каждого модуля, в который он включается. К сожалению, необходимость повторной компиляции участков текста приводит к снижению общей производительности компилятора. Visual C++ ускоряет процесс компиляции, позволяя вам заранее откомпилировать ваши заголовочные файлы. Хотя **принцип предварительной компиляции** не нов, Microsoft использует действительно новый подход. Предварительная компиляция сохраняет на определенном этапе состояние компилируемой программы и вводит соотношение между файлами с текстом программы и заранее откомпилированным заголовком. Можно создать и больше одного откомпилированного заголовочного файла на каждый из файлов с текстом программы.

В отличие от большинства распространенных компиляторов C++, компилятор фирмы Microsoft не ограничивает предварительную компиляцию только заголовочными файлами. Поскольку он позволяет предварительно откомпилировать программу до заданной точки, вы можете также иметь и заранее откомпилированный исходный текст. Это чрезвычайно существенно для тех программ на C++, которые содержат в заголовочных файлах большую часть определений функций-членов класса. Вообще, предварительная компиляция предназначена для тех участков текста программы, которые можно считать неизменными; она служит для уменьшения времени компиляции тех частей программы, которые находятся в процессе разработки.

Библиотека **MFC** предоставляет классы для управления объектами Windows и обладает рядом классов общего назначения, которые могут быть использованы и в приложениях MS-DOS, и

Windows. Например, есть классы для управления файлами, строками, временем, памятью и обработки исключительных ситуаций.

Компилятор Microsoft Visual C++ обеспечивает полную поддержку **встраиваемых функций**. Это означает, что функция, содержащая комбинацию команд любого типа, может быть встроена в программу в месте ее вызова. Многие распространенные компиляторы C++ не допускают встраивание определенных операторов или выражений, — например, встраивание не применяется к функциям, содержащим оператор `switch`, `while` или `for`.

Кроме обычных функций в C++, о которых вы уже знаете, есть еще встроенные функции. Встроенные функции не так значимы, но желательно в них разбираться. Основная идея в том, чтобы ускорить программу ценой занимаемого места. Встроенные функции во многом похожи на заполнитель. После того как вы определите встроенную функцию с помощью ключевого слова `inline`, всякий раз когда вы будете вызывать эту функцию, компилятор будет заменять вызов функции фактическим кодом из функции.

Как это делает программу быстрее? Легко, вызовы функций занимают больше времени, чем написание всего кода без функции. Просмотр вашей программы и замена функции, которую вы использовали 100 раз с кодом из функции, займет очень много времени. Конечно, используя встроенные функции для замены обычных вызовов функций, вы также значительно увеличите размер вашей программы.

Использовать ключевое слово `inline` легко, просто поставьте его перед именем функции. Затем, используйте её как обычную функцию.

Пример встроенной функции

```
1      #include <iostream>
2
3      using namespace std;
4
5      inline void hello()
6      {
7          cout<<"hello";
8      }
9      int main()
10     {
11         hello(); //Call it like a normal function...
12         cin.get();
13     }
```

Однако, как только программа будет скомпилирована, вызов `hello();` будет заменен на код функции.

Встроенные функции очень хороши для ускорения программы, но если вы используете их слишком часто или с большими функциями, у вас будет чрезвычайно большая программа. Иногда большие программы менее эффективны, и поэтому они будут работать медленнее, чем раньше. Встроенные функции лучше всего подходят для небольших функций, которые часто вызываются. Наконец, обратите внимание, что компилятор может, по своему желанию, игнорировать ваши попытки сделать функцию встроенной. Так что если вы ошибетесь и сделаете встроенной чудовищную функцию в пятьдесят строк, которая вызывается тысячи раз, компилятор может игнорировать вас.

9.Пакет компиляторов Visual C++. Параметры компилятора. General. Debug. CustomBuild.

Пакет Microsoft Visual C++ включает в себя средства для построения программ для Windows. Кроме этого, ваш код может быть адаптирован для таких аппаратных платформ, как Apple Macintosh и машин с RISC-процессорами. В пакете C++ имеются все необходимые заголовочные файлы, библиотеки, редакторы окон диалога и ресурсов, необходимые для создания действительно надежного приложения Windows. Microsoft также включила непосредственно в среду разработки редакторы ресурсов для пиктограмм, растровых изображений, курсоров, меню и окон диалога.

Параметры компилятора

Рассматриваемые ниже параметры позволяют оптимизировать скорость, размер исполняемого модуля или время компиляции и сборки. Если вы не наблюдаете заметного скачка производительности, то, возможно, ваше приложение содержит недостаточно кода. Все параметры устанавливаются путем выбора пункта Settings (параметры) из меню Build (сборка).

Закладка **General** (общие параметры) позволяет указать, следует или нет использовать библиотеку **Microsoft Foundation Class**. Можно также задать каталоги, куда будут помещены промежуточные и окончательные результаты компиляции.

Закладка **Debug** (отладка) позволяет указать местоположение исполняемого файла и рабочий каталог, дополнительные параметры для программы, а также путь и имя файла для дистанционной отладки.

При помощи закладки **Custom Build** (дополнительные средства) вы можете задать дополнительные инструменты для построения программы. Сюда входят, например, программы, обрабатывающие выходной файл конфигурации проекта.

10.Пакет компиляторов Visual C++. C/C++. C++ Language. CodeGeneration. Customization. ListingFiles. Optimizations. PrecompiledHeaders. Preprocessor.

Пакет Microsoft Visual C++ включает в себя средства для построения программ для Windows. Кроме этого, ваш код может быть адаптирован для таких аппаратных платформ, как Apple Macintosh и машин с RISC-процессорами. В пакете C++ имеются все необходимые заголовочные файлы, библиотеки, редакторы окон диалога и ресурсов, необходимые для создания действительно надежного приложения Windows. Microsoft также включила непосредственно в среду разработки редакторы ресурсов для пиктограмм, растровых изображений, курсоров, меню и окон диалога.

C/C++

Закладка C/C++ позволяет выбрать одну из следующих категорий:

- General
- C++ Language
- Code Generation
- Customization
- Listing Files
- Optimizations
- Precompiled Headers
- Preprocessor

C++ Language.

Категория **Code Generation** (генерация кода) позволяет задать целевой микропроцессор (от 80386 до Pentium), выбрать соглашение о вызовах, указать стандартную библиотеку и задать метод выравнивания элементов структуры.

Категория **Customization** (адаптация) позволяет включить или отключить следующие возможности:

- расширения языка
- компоновка на уровне функций
- идентичные строки
- минимальная перекompиляция
- инкрементная компиляция
- отмена заставки и информационных сообщений

Категория **Listing Files** (файлы листинга) позволяет включить генерацию информации для браузера. Кроме того, можно указать местоположение файла для браузера. Можно разрешить включение локальных переменных в информацию браузера. Дополнительно можно задать типы файлов. Перечислены параметры проекта.

Категория **Optimizations** (параметры оптимизации) позволяет установить различные варианты оптимизации, например, по скорости или по размеру.

Категория **Precompiled Headers** (предварительно откомпилированные заголовки) позволяет использовать заранее откомпилированные заголовочные файлы. Это файлы с расширением PCH. Предварительная компиляция заголовочных файлов ускоряет процесс компиляции и сборки, но по завершении проекта их следует удалить, так как они занимают много места.

Категория **Preprocessor** (препроцессор) позволяет задать макроопределения для препроцессора. Можно также указать дополнительные каталоги и отменить стандартные пути.

11.Пакет компиляторов Visual C++. Link. General. Customization. Debug. Input. Output.

Пакет Microsoft Visual C++ включает в себя средства для построения программ для Windows. Кроме этого, ваш код может быть адаптирован для таких аппаратных платформ, как Apple Macintosh и машин с RISC-процессорами. В пакете C++ имеются все необходимые заголовочные файлы, библиотеки, редакторы окон диалога и ресурсов, необходимые для создания действительно надежного приложения Windows. Microsoft также включила непосредственно в среду разработки редакторы ресурсов для пиктограмм, растровых изображений, курсоров, меню и окон диалога.

C++ Language

Закладка **Link** (компоновка) позволяет выбрать одну из следующих категорий: - General -Customization - Debug - Input - Output

Категория **General** (общие параметры) позволяет установить порог выдачи предупреждающих сообщений, указать отладочную информацию, установить оптимизацию компиляции и перечислить дополнительные параметры проекта.

Категория **Customization** (адаптация) позволяет включить или отключить следующие возможности:

- расширения языка
- компоновка на уровне функций
- идентичные строки
- минимальная перекompиляция
- инкрементная компиляция

- отмена заставки и информационных сообщений

Категория **Debug** (отладка) позволяет задать различные форматы для создания карты компоновки и отладочной информации.

Категория **Input** (ввод) позволяет указать объектные модули и библиотеки. Кроме того, можно задать имена файла перекрестных ссылок и файла, где находится DOS-заголовок программы.

Категория **Output** (вывод) позволяет установить базовый адрес, точку входа, размер стека и информацию о версии проекта.

12.Пакет компиляторов Visual C++. Resources. OLE Types. BrowseInfo.

Пакет Microsoft Visual C++ включает в себя средства для построения программ для Windows. Кроме этого, ваш код может быть адаптирован для таких аппаратных платформ, как Apple Macintosh и машин с RISC-процессорами. В пакете C++ имеются все необходимые заголовочные файлы, библиотеки, редакторы окон диалога и ресурсов, необходимые для создания действительно надежного приложения Windows. Microsoft также включила непосредственно в среду разработки редакторы ресурсов для пиктограмм, растровых изображений, курсоров, меню и окон диалога.

C++ Language

Закладка **Resources** (ресурсы) позволяет задать имя файла ресурсов (обычно он имеет расширение .RES). Можно дополнительно указать язык ресурсов, каталоги для ресурсов и задать препроцессорные определения.

Закладка **OLE Types** (типы OLE) позволяет задать имя выходного файла, имя выходного заголовочного файла, определения для препроцессора и заставку.

Закладка **Browse Info** (информация браузера) дает возможность указать имя файла с информацией браузера. Кроме того, можно включить возможность генерации информации браузера и заставки.

13.Меню File (файл). New... Open... Close. Open Workspace. Close Workspace. Save. Save As...

В меню File среды Visual C++ собраны стандартные средства работы с файлами, встречающиеся во многих приложениях Windows.

New...

Пункт **New...** (создать) открывает новое окно редактирования файла. Обычно с этого начинается создание программы. Среда автоматически называет и нумерует открытые вами окна. Нумерация начинается с 1, поэтому заголовок первого окна всегда будет xxx/, второго окна — xxx2, и так Далее, xxx соответствует типу файла, с которым вы работаете (текст, проект, ресурс, рисунок, код, значок или курсор).

Open...

В отличие от **New...**, который открывает окно редактирования не существовавшего ранее файла, пункт **Open...** (открыть) вызывает появление окна диалога с запросом информации о файле, который уже сохранен на диске. Это стандартное окно диалога **Open File** (открыть файл), в котором показаны текущее устройство, каталог и шаблон поиска файлов, и предлагается ввести нужные параметры.

Close.

Команда **Close** (закрыть) используется, чтобы закрыть открытый файл. Если у вас открыто несколько файлов, будет закрыто активное (текущее) окно. Вы можете отличить активное окно от неактивного, посмотрев на рамку окна. Активное (текущее) окно имеет фокус ввода и изображается цветом, установленным в системе для отображения активных окон. Обычно для активных окон установлены выделенный цветом заголовок и темная рамка. Неактивные окна имеют обычно серые заголовок и рамку.

Open Workspace.

Команда **Open Workspace** (открыть рабочее пространство) используется для активизации ранее сохраненного рабочего пространства. Рабочие пространства соответствуют приложениям, которые вы можете создавать. Проект состоит из одного набора исходных файлов и набора из одной или более конфигураций. Каждая конфигурация проекта вместе с набором файлов однозначно определяют двоичный файл, который в результате будет генерироваться.

Close Workspace.

Команда **Close Workspace** (закрыть рабочее пространство) закрывает активное рабочее пространство. Это позволяет открыть другое рабочее пространство и перейти к работе над другим приложением.

Save.

Команда **Save** (сохранить) записывает содержимое активного окна в соответствующий файл. Вы можете определить, существует ли такой файл, просто взглянув на заголовок окна. Если вы увидите автоматически созданный заголовок (вроде xxx1), значит, содержимое окна еще ни разу не было сохранено, и ему не назначено имени файла. Вы также можете сохранить файлы, нажав соответствующую кнопку панели инструментов.

Save As...

Команда **Save As...** (сохранить как) позволяет создать копию содержимого активного окна под другим именем. Например, вы закончили работу над проектом. У вас есть работающая программа, но вы все же хотите попробовать внести пару изменений. Ради собственного спокойствия вы не хотите трогать работающую версию. Воспользовавшись командой **Save As**, вы можете сохранить файл под другим именем, и затем работать с его копией. Если новая версия работать не будет, можно вернуться к первоначальной версии файла.

14. Меню File (файл). Save All. Find in Files... Page Setup... Print... Список последних проектов. Exit.

В меню **File** среды Visual C++ собраны стандартные средства работы с файлами, встречающиеся во многих приложениях Windows.

Save All.

Команда **Save All** (сохранить все) записывает в соответствующие файл, содержимое всех открытых окон. Если вы никогда не писали приложения Windows на C или C++, вас, возможно, ошеломит общее количество файлов, требуемых для создания работающей программы. Недостаток команды **Save** заключается в том, что она сохранит лишь содержимое активного окна.

Find in Files...

Команда **Find in Files** (поиск в файлах) позволяет найти последовательность символов в одном или нескольких файлах. Файлы, в которых следует искать, задаются указанием типа файла и каталога, где они находятся. Для задания искомой последовательности вы можете пользоваться регулярными выражениями. Результаты поиска отображаются в окне **Output** (вывод). Когда поиск закончен, можно открыть нужный файл, дважды щелкнув на его имени в окне **Output**.

Page Setup...

Наиболее часто команда **Page Setup** (настройка параметров страницы) используется при оформлении распечаток текстов программ. Эта команда позволяет выбрать для каждой страницы верхний и нижний колонтитул, а также установить размеры полей.

Print...

Чтобы получить на бумаге копию содержимого активного окна, достаточно выполнить команду **Print** (печать). Окно диалога **Print** предоставляет вам несколько возможностей. Во-первых, вы можете решить, печатать ли все содержимое окна, или только выделенный текст. Если в окне есть выделенный текст, кнопка

опции **Print Range Selection** (печатать только выделенный) будет изображена обычным цветом (не серым), указывая на то, что данная возможность доступна.

Список последних проектов.

Список последних использовавшихся проектов расположен в меню **File** непосредственно под хронологическим списком файлов. Список проектов схож со списком файлов, но содержит только имена проектов. Чтобы открыть любой файл из этих списков, щелкните левой кнопкой мыши на нужном имени.

Exit.

Команда **Exit** (выход) позволяет завершить работу среды Visual C++. Не волнуйтесь, если вы забыли сохранить какие-либо файлы; среда автоматически выдаст предупреждение и даст возможность сохранить все измененные файлы.

15. Меню Edit. Undo. Redo. Cut. Copy. Paste. Delete. Select All.

Команды меню Edit (правка) позволяют быстро находить и исправлять текст в активном окне примерно так же, как это делается любым из распространенных текстовых процессоров.

Undo.

Команда **Undo** (отменить) позволяет отменить последнее изменение, которое было внесено в процессе редактирования. Вы можете также выполнить эту команду с помощью панели инструментов. На кнопке **Undo** изображена стрелочка влево; эта кнопка расположена седьмой слева.

Redo.

Команда **Redo** (восстановить) позволяет отменить действие последней команды **Undo**. Пользуйтесь этой командой, чтобы вернуть те изменения, которые вы по ошибке отменили. Команда **Redo** доступна также посредством панели инструментов. Это восьмая кнопка слева.

Cut.

Команда **Cut** (вырезать) переносит в буфер обмена текст, выделенный в активном окне. При этом выделенный текст будет удален из окна. Для того чтобы выделить текст, поместите указатель мыши на первый из символов выделяемого текста и, удерживая левую кнопку нажатой, переместите мышь вправо/вниз до конца выделяемого участка. После этого текст, который вы выделили, будет отображаться инверсным цветом (негативно).

Copy.

Как и **Cut**, команда **Copy** (копировать) помещает выделенный текст в буфер обмена, но, в отличие от **Cut**, **Copy** оставляет исходный текст на месте. Эта команда пригодится при копировании сложных участков текста программы или комментариев, которые требуются в нескольких местах программы.

Paste.

Команда **Paste** (вставить) используется, чтобы вставить содержимое буфера обмена туда, где в данный момент находится курсор. Вставлять из буфера обмена можно только то, что было ранее помещено туда командой **Cut** или **Copy**.

Delete.

Команда **Delete** (удалить) уничтожает выделенный текст, не помещая его в буфер обмена. Для того чтобы выделить текст, поместите указатель мыши на первый из символов выделяемого текста и, удерживая левую кнопку нажатой, переместите мышь вправо/вниз до конца выделяемого участка. После этого текст, который вы выделили, будет отображаться инверсным цветом (негативно).

Select All.

Команда **Select All** (выделить все) используется, чтобы выделить все содержимое активного окна для последующей вырезки, копирования или удаления.

16. Меню Edit. Find... Replace... Go To... InfoViewer Bookmarks... Bookmark. Breakpoints... Properties...

Команды меню Edit (правка) позволяют быстро находить и исправлять текст в активном окне примерно так же, как это делается любым из распространенных текстовых процессоров.

Find...

Команда **Find...** (поиск) работает аналогично команде поиска в текстовом процессоре. Поскольку в языках C и C++ различаются прописные и строчные буквы, команду **Find...** можно настроить для поиска с учетом регистра, без учета регистра, а также поиска только полного слова. Команда **Find** также позволяет указать направление поиска (вперед или назад от текущей позиции курсора)

Replace...

Команда **Replace...** (заменить) вызывает окно диалога **Replace**, которое позволяет найти текст по образцу и заменить на заданный. Введите образец, который следует найти, затем укажите текст, на который его следует заменить, и наконец, выберите один из типов соответствия образцу. Можно задать соответствие с учетом регистра, без учета регистра, только полное слово или регулярное выражение.

Go To...

Поскольку приложение Windows обычно строится из десятков файлов, иногда не так уж просто найти место, где находится описание интересующего вас объекта. Это сделает для вас команда **Go To... (перейти к...)**. Чтобы найти строку с описанием константы, переменной или функции, установите курсор на имя объекта (или непосредственно слева от него) и выполните эту команду. Интегрированная среда автоматически проследит, где находится описание данного имени.

InfoViewer Bookmarks...

На панели **InfoViewer** показано оглавление электронных книг. Вы можете вызвать любой раздел из этой иерархии.

Bookmark.

Команда **Bookmark** (закладка) устанавливает или удаляет закладку. Закладки используются для того, чтобы отмечать некоторые строки в тексте программы, к которым придется обратиться позже. Закладки можно установить как этой командой, так и при помощи параметра **Set Bookmark** (установить закладку) команды Find...

Breakpoints...

Команда **Breakpoints...** (точки останова) открывает окно диалога **Breakpoints**, позволяющее добавлять, удалять, включать отдельные точки останова, а также удалить все точки сразу. Точки останова затем могут быть использованы командой **Go (запустить)**.

Properties...

Команда **Properties...** (свойства) вызывает окно диалога, посредством которого задаются характеристики текущего окна. К примеру, файл в активном окне можно пометить "только для чтения" или указать, что он может содержать конструкции C, но не C++.

17. Меню View. ClassWizard... Resource Symbols... Resource Includes... Full Screen. Toolbars... InfoViewer Query Results. InfoViewer History List. Project Workspace.

Меню **View (просмотр)** обеспечивает доступ к командам, применяемым для отображения текущего проекта в разных видах. Сюда входят средства просмотра файлов справки, электронных книг, а также другие средства, облегчающие отладку приложений.

ClassWizard...

Команда **ClassWizard...** (мастер классов) вызывает средство высокоуровневого программирования, которое позволяет объявлять новые классы на основе компонент **Microsoft Foundation Class (MFC)**, либо добавлять новые методы обработки сообщений в существующий объект, построенный на основе MFC.

Resource Symbols...

Вы можете использовать команду **Resource Symbols...** (символы ресурсов) для отображения списка символов, используемых в данном файле ресурсов.

Resource Includes...

Обычно **Microsoft Developer Studio** (мастерская разработчика) хранит все ресурсы в файле с расширением RC, а определения символов — в файле RESOURCE.H. При помощи команды **Resource Includes...** (включаемые файлы ресурсов) вы можете изменить распределение ресурсов в файлах

Full Screen.

При помощи команды **Full Screen** (полный экран) вы можете развернуть текстовый редактор (или другие редакторы ресурсов) на полный экран.

Toolbars...

Команда **Toolbars...** (панели инструментов) вызывает появление окна диалога Toolbars, позволяющее выбрать панели инструментов, которые должны отображаться в среде Visual C++.

InfoViewer Query Results.

Команда **InfoViewer Query Results** (результаты запросов **InfoViewer**) показывает список предыдущих запросов на поиск информации в справочной системе.

InfoViewer History List.

Команда **InfoViewer History List** (хронологический список **InfoViewer**) показывает список разделов справки, к которым вы недавно обращались.

Project Workspace.

Команда **Project Workspace** (рабочее пространство проекта) выдает окно с закладками, соответствующими активным окнам текущего проекта

18. Меню View. Info Viewer Topic. Output. Watch. Variables. Registers. Memory. Call Stack. Disassembly. Меню Insert.

Info Viewer Topic.

Команда **InfoViewer Topic** (раздел **InfoViewer**) показывает содержание текущего раздела справочной системы.

Output.

Команда **Output (вывод)** активизирует окно **Output** для данного проекта.

Watch.

Команда **Watch (наблюдение)** отображает окно отладчика **Watch**. Это окно используется для наблюдения за значениями переменных по ходу выполнения программы.

Variables.

Команда **Variables (переменные)** открывает окно **Variables**, позволяющее увидеть активные в данный момент переменные, локальные переменные, или указатель **this**.

Registers.

Команда **Registers** вызывает окно регистров. Это окно позволяет считывать (но не модифицировать) состояние процессора, в том числе содержимое регистров и состояние флагов.

Memory.

Команда **Memory (память)** открывает окно, отображающее содержимое памяти с заданного адреса. Окно дает возможность просматривать состояние памяти в пределах адресного пространства программы.

Call Stack.

Команда **Call Stack (стек вызовов)** вызывает соответствующее окно диалога. В нем показана последовательность вложенных вызовов функций до уровня, на котором находится текущая строка текста.

Disassembly.

Команда **Disassembly (дизассемблер)** отображает окно с ассемблерным кодом, который был сгенерирован из ваших исходных текстов.

Меню Insert.

Посредством меню **Insert (вставить)** вы сможете вставить файлы (Files...), ресурсы (Resources), копии ресурсов (Resource Copies), файлы в проект (Files into a Project), проекты в проект (Project) и компоненты рабочего пространства (Components...).

19. Меню Build. Compile. Build. Rebuild All. Batch Build... Stop Build. Update All Dependencies.

Меню **Build (сборка)** содержит много пунктов, необходимых для генерации исполняемого файла приложения (*.EXE).

Compile.

Команда **Compile (компиляция)** заставляет среду компилировать текст в текущем окне. Компиляция — важный этап разработки, поскольку именно здесь вы узнаете, содержит ли данный файл синтаксические ошибки. По этой причине существует возможность компилировать заголовочные файлы (*.H), несмотря на то, что они не могут исполняться. Если при компиляции обнаружатся ошибки (или предупреждения о возможных ошибках), сообщения о них будут помещены в окно Output.

Build.

Обычно программы на C/C++ состоят из множества файлов. Эти файлы могут поставляться с компилятором, с операционной системой, создаваться программистом или даже приобретаться у сторонних производителей. Картина еще более усложняется, если файлы, составляющие проект, создаются несколькими группами программистов. Поскольку файлов очень много, а компиляция может занимать много времени, команда **Build (сборка)** становится чрезвычайно полезной. В процессе сборки анализируются все файлы проекта и затем компилируются и компонуются лишь те из них, которые были изменены после создания исполняемого файла проекта.

Rebuild All.

Единственная разница между командами **Build** и **Rebuild All (полная сборка)** состоит в том, что **Rebuild All** не обращает внимания на даты изменения файлов и терпеливо компилирует и компоует все.

Batch Build...

Команда **Batch Build... (пакетная сборка)** работает аналогично обычной сборке, но может создать в одном проекте сразу несколько целевых файлов.

Stop Build.

Используйте команду **Stop Build (прервать сборку)**, если вы хотите остановить процесс сборки.

Update All Dependencies.

Команда **Update All Dependencies (обновить список зависимостей)** может быть использована, если вы хотите, чтобы система заново построила диски включаемых файлов для каждого из исходных файлов проекта.

20. Меню Build. Debug. Execute. Settings... Configurations... Subprojects... Set Default Configuration...

Меню **Build (сборка)** содержит много пунктов, необходимых для генерации исполняемого файла приложения (*.EXE).

Debug.

Команда **Debug (отладка)** запускает интегрированный отладчик.

Execute.

Команда **Execute (выполнить)** используется для запуска вашей программы. В зависимости от типа целевого файла Visual C++ автоматически вызовет среду MS-DOS, Windows 95 или Windows NT для тестирования вашей программы.

Settings...

Команда **Settings... (параметры)** используется для выбора конфигурации проекта.

Configurations...

Команда **Configurations...(конфигурации)** служит для управления конфигурациями проекта.

Subprojects...

Microsoft Developer Studio позволяет вам включать в проект другие проекты. Доступ к ним осуществляется при помощи команды **Subprojects... (подпроекты)**.

Set Default Configuration...

В большинстве случаев достаточно настроить конфигурацию для всего проекта в целом, но при необходимости вы можете задать для конкретных файлов проекта другие значения каких-либо параметров. Таким образом, конфигурация имеет иерархическую структуру: параметр, заданный на уровне проекта, относится ко всем файлам, кроме тех, для которых он переопределен на уровне файла. Например, если при помощи команды **Set Default Configuration (установить основную конфигурацию)** вы установили параметр **Default optimization (стандартная оптимизация)**, то эта установка будет действовать на все файлы в данной конфигурации. Тем не менее, вы можете установить конкретные настройки оптимизации (или отключить ее вообще) для каких-либо файлов в отдельности. Значения параметров, заданные на уровне файла, будут использованы вместо тех, которые заданы на уровне проекта. Некоторые виды параметров (например, параметры компоновки) можно задавать только на уровне проекта.

21. Меню Tools. Browse... Close Browse Info File. OLE Control Test Container. OLE Object View.

Меню Tools (инструменты) обеспечивает доступ к множеству полезных вспомогательных средств интегрированной среды. Эти средства призваны облегчить, насколько это возможно, процесс разработки и модификации различных объектно-ориентированных приложений Windows.

Browse...

Команда **Browse... (найти)** выдаст хронологический список описаний объектов или ссылок на них, которые вы искали ранее. Список устроен наподобие стека, то есть при добавлении новый элемент становится первым в списке.

Close Browse Info File.

Команда **Close Browse Info File (закрыть файл информации браузера)** закрывает файл **Browse Info (информация браузера)**

OLE Control Test Container.

Эта команда вызывает утилиту **OLE Control Test Container** (тестовый контейнер для управляющих элементов OLE). Это приложение, которое полностью поддерживает управляющие элементы OLE и позволяет встраивать их в свои окна и окна диалога.

OLE Object View.

Команда **OLE Object View** (показать объект OLE) вызывает **OLE Object Viewer** (средство просмотра объектов OLE). **OLE Object Viewer** — это вспомогательная программа для того, чтобы помочь разработчикам OLE-приложений лучше понять, что происходит в их системах.

22. Меню Window. New Window. Split. Hide. Cascade.

Меню **Window** позволяет управлять отображением различных окон, используемых в процессе разработки приложения. Это меню также позволяет устанавливать фокус ввода, указывая, какое окно сделать активным.

New Window.

Команда **New Window** (новое окно) дает еще одну возможность создать в текущем проекте новое окно

Split

Команда **Split** (разделить) делит текущее окно на две части.

Hide.

Команда **Hide** (спрятать) делает текущее окно скрытым.

Cascade.

Команда **Cascade** (каскад) располагает все открытые на экране окна одно за другим со сдвигом вправо и вниз, наподобие колоды карт.

23. Меню Window. Tile Horizontally, Tile Vertically. Close All... Windows. Меню Help.

Меню **Window** позволяет управлять отображением различных окон, используемых в процессе разработки приложения. Это меню также позволяет устанавливать фокус ввода, указывая, какое окно сделать активным.

Tile Horizontally, Tile Vertically.

Команды **Tile Horizontally** (сверху вниз) и **Tile Vertically** (справа налево) заставляют среду поровну разделить экран между всеми окнами. Преимущество такого расположения состоит в том, что вы можете одновременно видеть содержимое всех окон. Недостатки проявляются при большом количестве окон: тогда каждому из них достается место размером с почтовую марку.

Close All...

Команда **Close All...** (закрыть все) закрывает все открытые окна.

Windows.

Команда **Windows** (окна) открывает список всех имеющихся в среде окон, позволяя быстро найти нужное.

Меню Help.

Возможности такой разнообразной, насыщенной и интеллектуальной среды, как Visual C++, могут остаться невостребованными без последнего, но самого важного меню **Help** (справка). Пытаетесь ли вы разобраться в конструкции языка C/C++, узнать о средстве из среды Visual C++ или научиться использовать их совместно, вам помогут команды, доступные посредством меню **Help**.

24. История языка С. Взаимоотношения с другими языками. Достоинства языка С. Малый размер. Набор команд языка. Быстродействие. Язык со слабой типизацией. Структурированный язык. Поддержка модульного программирования.

Язык программирования Си был разработан и реализован в 1972 году сотрудником фирмы AT&T Bell Laboratories Денисом Ритчи. Прообразом языка Си для Д. Ритчи послужил язык Би, разработанный Кеном Томпсоном. Он является результатом эволюционного развития языков BCPL (Richards, M., "BCPL: A. Tool for Compiler Writing and System Programming", Proc. AFIPS SJCC, 34, 557-566, 1969) и Би (Johnson, S. C., and B. W. Kernighan, "The Programming Language B", Comp. Sci. Tech. Rep. No. 8, Bell Laboratories. 1973). Основным достоинством языка Си по сравнению с языками BCPL и Би является введение в него типов данных. Язык Си был разработан во время создания операционной системы UNIX (ОС UNIX). Развитие языка Си продолжалось и после окончания его разработки и касалось, в частности, проверки типов данных и средств, облегчающих перенос программ в другую среду.

Достоинства языка С

Оптимальный размер программы

В основу С положено значительно меньше синтаксических правил, чем у других языков программирования. В результате для эффективной работы компилятора языка достаточно всего 256 Кб оперативной памяти. Действительно, список операторов и их комбинаций в языке С обширнее, чем список ключевых слов.

Сокращенный набор ключевых слов

Первоначально, в том виде, в каком его создал Деннис Ритчи, язык С содержал всего 27 ключевых слов. В ANSI С было добавлено несколько новых зарезервированных слов. В Microsoft С набор ключевых слов был еще доработан, и общее их число превысило 50. Многие функции, представленные в большинстве других языков программирования, не включены в язык С. Например, в С нет встроенных функций ввода/вывода, отсутствуют математические функции (за исключением базовых арифметических операций) и функции работы со строками. Но если для большинства языков отсутствие таких функций было бы признаком слабости, то С взамен этого предоставляет доступ к самостоятельным библиотекам, включающим все перечисленные функции и многие другие. Обращения к библиотечным функциям в программах на языке С происходят столь часто, что эти функции можно считать составной частью языка. Но в то же время их легко можно переписать без ущерба для структуры программы — это безусловное преимущество С.

Быстрое выполнение программ

Программы, написанные на С, отличаются высокой эффективностью. Благодаря небольшому размеру исполняемых модулей, а также тому, что С является языком достаточно низкого уровня, скорость выполнения программ на языке С соизмерима со скоростью работы их ассемблерных аналогов.

Упрощенный контроль за типами данных

В отличие от языка Pascal, в котором ведется строгий контроль типов данных, в С понятие типа данных трактуется несколько шире. Это унаследовано от языка В, который так же свободно обращался с данными разных типов. Язык С позволяет в одном месте программы рассматривать переменную как символ, а в другом месте — как ASCII-код этого символа, от которого можно отнять 32, чтобы перевести символ в верхний регистр.

Реализация принципа проектирования "сверху вниз"

Язык С содержит все управляющие конструкции, характерные для современных языков программирования, в том числе инструкции for, if/else, switch/case, while и другие. На момент появления языка это было очень большим достижением. Язык С также позволяет создавать изолированные программные блоки, в пределах которых переменные имеют собственную область видимости. Разрешается создавать локальные переменные и передавать в подпрограммы значения параметров, а не сами параметры, чтобы защитить их от модификации.

Модульная структура

Язык C поддерживает модульное программирование, суть которого состоит в возможности отдельной компиляции и компоновки разных частей программы. Например, вы можете выполнить компиляцию только той части программы, которая была изменена в ходе последнего сеанса редактирования. Это значительно ускоряет процесс разработки больших и даже среднего размера проектов, особенно если приходится работать на медленных машинах. Если бы язык C не поддерживал модульное программирование, то после внесения небольших изменений в программный код пришлось бы компилировать всю программу целиком, что могло бы занять слишком много времени.

26. История языка C. Недостатки языка C. Слабая типизация. Отсутствие проверок на этапе исполнения. Использование языка Си. Будущее языка Си.

Язык программирования Си был разработан и реализован в 1972 году сотрудником фирмы AT&T Bell Laboratories Денисом Ритчи. Прототипом языка Си для Д. Ритчи послужил язык Би, разработанный Кеном Томпсоном. Он является результатом эволюционного развития языков BCPL (Richards, M., "BCPL: A. Tool for Compiler Writing and System Programming", Proc. AFIPS SJCC, 34, 557-566, 1969) и Би (Johnson, S. C., and B. W. Kernighan, "The Programming Language B", Comp. Sci. Tech. Rep. No. 8, Bell Laboratories. 1973). Основным достоинством языка Си по сравнению с языками BCPL и Би является введение в него типов данных. Язык Си был разработан во время создания операционной системы UNIX (OS UNIX). Развитие языка Си продолжалось и после окончания его разработки и касалось, в частности, проверки типов данных и средств, облегчающих перенос программ в другую среду.

Недостатки языка C

Упрощенный контроль за типами данных!

То, что язык C не осуществляет строгого контроля за типами данных, можно считать как достоинством, так и недостатком. В некоторых языках программирования присвоение переменной одного типа значения другого типа воспринимается как ошибка, если при этом явно не указана функция преобразования. Благодаря этому исключается появление ошибок, связанных с неконтролируемым округлением значений. Как было сказано выше, язык C позволяет присваивать целое число символьной переменной и наоборот. Это не проблема для опытных программистов, но для новичков данная особенность может оказаться одним из источников побочных эффектов. Побочными эффектами называются неконтролируемые изменения значений переменных. Поскольку C не осуществляет строгого контроля за типами данных, это дает большую гибкость при манипулировании переменными. Например, в одном выражении оператор присваивания (=) может использоваться несколько раз. Но это также означает, что в программе могут встречаться выражения, тип результата которых неясен или трудно определить. Если бы C требовал однозначного определения типа данных, то это устранило бы появление побочных эффектов и неожиданных результатов, но в то же время сильно уменьшило бы мощь языка, сведя его к обычному языку высокого уровня.

Ограниченные средства управления ходом выполнения программы

Вследствие того, что выполнение программ на языке C слабо контролируется, многие их недостатки могут остаться незамеченными. Например, во время выполнения программы не поступит никаких предупреждающих сообщений, если осуществлен выход за границы массива. Это та цена, которую пришлось заплатить за упрощение компилятора ради его скорости и эффективности использования памяти.

Язык Си быстро становится одним из наиболее важных и популярных языков программирования. Его использование все более расширяется, поскольку часто программисты предпочитают язык Си всем другим языкам после первого знакомства с ним.

27. Исходные файлы и выполняемые файлы. Принципы программирования. Стандарт ANSI C. Эволюция языка C++ и объектно-ориентированное программирование. История C++. Использование объектов C++ для быстрого создания программы.

Принципы программирования

ANSI C — стандарт языка C, опубликованный Американским национальным институтом стандартов (ANSI). Следование этому стандарту помогает создавать легко портируемые программы.

Первый стандарт языка C был опубликован американским институтом ANSI. Через некоторое время он был принят международной организацией по стандартизации ISO, продолжившей выпускать следующие версии стандарта, которые стали приниматься как стандарт и институтом ANSI. Несмотря на это стандарт до сих пор чаще называют ANSI C, а не ISO C.

Истоки C++

История создания C++ начинается с языка C. И немудрено: C++ построен на фундаменте C. C++ и в самом деле представляет собой супермножество языка C. (Все компиляторы C++ можно использовать для компиляции C-программ.) C++ можно назвать расширенной и улучшенной версией языка C, в которой реализованы принципы объектно-ориентированного программирования

Рождение C++

Итак, C++ появился как ответ на необходимость преодолеть еще большую сложность программ. Он был создан Бьерном Страуструпом (Bjarne Stroustrup) в 1979 году в компании Bell Laboratories (г. Муррей-Хилл, шт. Нью-Джерси). Сначала новый язык получил имя "C с классами" (C with Classes), но в 1983 году он стал называться C++.

Эволюция C++

С момента изобретения C++ претерпел три крупных переработки, причем каждый раз язык как дополнялся новыми средствами, так и в чем-то изменялся. Первой ревизии он был подвергнут в 1985 году, а второй — в 1990. Третья ревизия имела место в процессе стандартизации, который активизировался в начале 1990-х. Специально для этого был сформирован объединенный ANSI/ISO-комитет, который 25 января 1994 года принял первый проект предложенного на рассмотрение стандарта. В этот проект были включены все средства, впервые определенные Страуструпом, и добавлены новые. Но в целом он отражал состояние C++ на тот момент времени. Вскоре после завершения работы над первым проектом стандарта C++ произошло событие, которое заставило значительно расширить существующий стандарт. Речь идет о создании Александром Степановым стандартной библиотеки шаблонов (Standard Template Library — STL). Как вы узнаете позже, STL — это набор обобщенных функций, которые можно использовать для обработки данных. Он довольно большой по размеру. Комитет ANSI/ISO проголосовал за включение STL в спецификацию C++. Добавление STL расширило сферу рассмотрения средств C++ далеко за пределы исходного определения языка. Однако включение STL, помимо прочего, замедлило процесс стандартизации C++, причем довольно существенно. Помимо STL, в сам язык было добавлено несколько новых средств и внесено множество мелких изменений. Поэтому версия C++ после рассмотрения комитетом по стандартизации стала намного больше и сложнее по сравнению с исходным вариантом Страуструпа. Конечный результат работы комитета датируется 14 ноября 1997 года, а реально ANSI/ISO-стандарт языка C++ увидел свет в 1998 году. Именно эта спецификация C++ обычно называется Standard C++. И именно она описана в этой книге. Эта версия C++ поддерживается всеми основными C++-компиляторами, включая Visual C++ (Microsoft) и C++ Builder (Borland). Поэтому код программ, приведенных в этой книге, полностью применим ко всем современным C++-средам.

28. Исходные файлы и выполняемые файлы. Некоторые усовершенствования по сравнению с языком C. Комментарии. Имена перечисляемых типов. Имена структур или класса. Блочные объявления. Операция уточнения области действия (scope).

Описатель const. Анонимные объединения

Комментарии

Символ комментария `//`, действующий до конца строки, является теперь составной частью языка, а не только расширением Microsoft, как это было для языка C. Комментарии предназначены для упрощения чтения исходного кода программы путем добавления пояснений на литературном языке (Как правило, английском) в исходный код

`//` Это строка-комментарий, открывается `//`, закрывания не требует `/*`

Это блок-комментарий, `/*` - открывает комментарий, `*/` - закрывает `*/`

Имена перечислений, структур и объединений

В C++ имя перечисления, структуры или объединения является именем типа. Это упрощает нотацию, поскольку не нужно использовать ключевые слова `enum`, `struct` и `union` перед именем соответствующего типа. Таким образом, в C++ выражения вида

```
struct Fruits{  
    // Компоненты структуры  
};  
Fruits stApple; // Определение переменной типа Fruits .  
являются допустимыми.
```

Блочные объявления

В C++ допускаются объявления внутри блоков и после программных операторов, что позволяет определять (объявлять) объект в том месте программы, где он используется первый раз — во многих случаях это улучшает читаемость программы.

```
void myFunc()  
{  
    int nFirstVar;  
    nFirstVar++;  
    // Другие операторы функции  
    ...  
    float fSecondVar = 7.0;  
    ...  
    // Остальные операторы функции  
}
```

Кроме того, можно определить и инициализировать переменную непосредственно внутри формального описания управляющей структуры:

```
for(int iCount = 0; iCount < MAX_COUNT; iCount++) { ... }
```

Оператор **scope (::)** уточняет область действия идентификатора переменной/функции/т.п.; Его необходимость связана с тем, что каждый применяемый в программе идентификатор должен быть уникален (Программа не может решить сама, какую из трех переменных X применять) в каждой области видимости — что не всегда удобно при подключении большого кол-ва внешних библиотек. В таком случае создаются пространства имен, которые позволяют группировать идентификаторы, иначе видимые глобально, в группы, объединенные общими именами. Идентификатор, объявленный в пространстве какого-то имени, в нем может применяться без ограничений, но вне этого пространства, чтобы обратиться к нему, нужно воспользоваться оператором **scope (::)** и обозначить, что вызываемый идентификатор принадлежит к этому пространству имен, например:

```
namespace first
```

```
{ int x = 5; }
```

```
Int main () {
```

```
int y = first::x; //Правильно, программа видит переменную x, принадлежащую first
```

```
x= y-1; //Неправильно, программа не видит переменную x
```

```
}
```

Описатель **const**: Может быть добавлен при объявлении & инициализации переменной, значение которой программа не будет изменять. Например: `const double pi = 3.141529`; Назначение? Удобнее всякий раз, когда нам нужно значение числа π , писать `pi`, чем многозначное число. Альтернатива – директива препроцессора `#define`, будет выглядеть как `#define pi 3.141529`. Разница – при использовании `const` программа видит `pi`, читает как имя и обращается к значению, которая лежит по адресу `pi`; При использовании `#define` программа видит `pi`, читает 3.141529 Соответственно, `#define` быстрее, `const` надежнее (Если не взлетит – найти ошибку проще)

29. Исходные файлы и выполняемые файлы. Явное преобразование типов. Объявления функций. Перегруженные функции. Значения параметров функций по умолчанию. Функции с неуказанным числом параметров. Ссылочные параметры функции. Операторы `new` и `delete`. Указатели `void`

Явное преобразование типов: Это преобразование, при котором мы явно выразили желание изменить тип данных. Как и любое преобразование, может привести к потере данных – в данном случае подразумевается, что это учтено разработчиком. Пример:

```
double a = 3.5;
```

```
int (a); //Производится явное преобразование типа переменной a
```

Объявление функций: В C++ в выражениях могут быть использованы только объявленные идентификаторы, поэтому любая функция должна быть объявлена до её вызова.

При объявлении не обязательно полностью описывать функцию – достаточно необходимой для вызова информации – идентификатора функции, типа возвращаемых ею данных и типов передаваемых функции параметров.

Перегрузка функций: Несколько функций может иметь одинаковый идентификатор (имя), если параметры их вызова различаются в числе и/или типе; Программа при вызове функции под таким именем выберет ту из функций, которой соответствуют переданные аргументы.

Значение параметров функций по умолчанию: При описании параметров функции можно задать опциональный параметр, что позволяет, к примеру, при вызове функции с тремя параметрами задать только два аргумента. Для этого необходимо для последнего параметра задать значение по умолчанию. Например:

```
int divide (int a, int b=2){return a/b;}
```

Функции с неуказанным числом параметров: Если в функцию может быть передано неопределенное число параметров одного типа (Например, передан массив чисел неопределенной длины), то вместо перегрузки функции с её описанием для каждого возможного числа параметров можно описать функцию с неуказанным числом параметров, например:

```
int average (int n, ..) {~~~~}
```


Это позволяет значительно сократить в объеме код программы.

Ссылочные параметры функции: Параметр, передаваемый в функцию, может быть передан двумя путями – может быть передано значение - тогда функция работает с локальной копией параметра, или может быть передана ссылка на значение (= Адрес этого значения), и тогда функция работает непосредственно с оригинальным параметром.

Операторы New & Delete служат для динамического выделения памяти в ходе выполнения программы в C++; При помощи оператора **New** программа выделяет память и возвращает адрес выделенной памяти, при помощи оператора **Delete** программа освобождает выделенную память.

```
int *a = new int(2); // Оператор new выделит память, 2 – значение,
                    // которым эта память будет проинициализирована
cout << a; // Выведет адрес в шестнадцатеричном формате, например
0x102AF2
cout << *a; // Выведет значение, которое располагается по адресу
0x102AF2. Т.е., 2
delete a; // Пометит занимаемую область памяти, как освобождённую
```

В C++ существует специальный тип указателя, который называется указателем на неопределённый тип. Для определения такого указателя вместо имени типа используется ключевое слово **void** в сочетании с описателем, перед которым располагается символ ptrОперации *.

```
void *UndefPoint;
```

30.Исходные файлы и выполняемые файлы. Основные усовершенствования по сравнению с языком C.(часто повторяется).

Конструкторы классов и инкапсуляция данных. Класс struct.

Конструкторы и деструкторы. Сообщения. "Дружественные" классы.

Конструкторы и деструкторы:

Это функции класса. Конструктор при вызове создает и инициализирует объект класса (Что позволяет избежать обращения к объекту до его инициализации), деструктор при вызове очищает память, выделенную под объект (Сам объект удаляется в последнюю очередь). Конструктор класса имеет имя класса и вызывается подобно функции (Класс идентификатор_объекта = конструктор(аргументы)); Деструктор имеет имя, состоящее из ~ и имени класса. Деструктор необходимо вызывать для объектов, на которые память выделена динамически (delete имя_объекта;), для остальных объектов он вызывается автоматически по завершении выполнения программы; Если деструктор в классе не описан, компилятор его автоматически создаст.

Инкапсуляция:

Для любого поля класса (Переменная, функция, т.д.) можно задать параметры доступа к нему. Возможны 2 варианта – Public (Общедоступен) и Private (Доступен только для объектов этого класса). Инкапсуляция поля есть запрещение доступа к нему извне.

Дружественные классы и функции:

Иногда нам надо дать доступ к инкапсулированным данным какой-то функции или классу извне (Например, чтобы не описывать эту же функцию еще раз.) В таком случае в описании класса мы записываем `friend class имя_класса;` или `friend прототип_функции;`, и тогда этот класс или функция получают доступ к инкапсулированным данным.

Класс `struct`:

Класс, созданный с помощью **`struct`**, практически не отличается от обычного класса (Объявленного через `class`). Единственное функциональное отличие – в объекте вида `class` по умолчанию любой член может быть вызван только другим членом класса (`private`), а в объекте вида `struct` любой член по умолчанию доступен кому угодно (`public`).

31. Исходные файлы и выполняемые файлы. Перегрузка операций. Производные классы. Полиморфизм при использовании виртуальных функций. Библиотеки потоков. Базовые элементы программы на C. Пять основных компонентов программы.

Перегрузка операций

Так же, как и для функций, язык C++ позволяет переопределить действие большинства операций так, чтобы при использовании с объектами конкретного класса они выполняли заданные действия. Такая перегрузка желательна с точки зрения непротиворечивости использования. Классическим примером, приводимым практически во всех учебниках по C++, является выполнение арифметических операций над классами, представляющими комплексные числа. В C++ механизм перегрузки операций реализован путем определения функции с ключевым словом **`operator`**, стоящим перед перегружаемой операцией.

Переопределить операцию для объектов класса можно используя либо соответствующий метод, либо дружественную функцию класса. В случае переопределения операции с помощью метода класса, последний в качестве неявного параметра принимает ключевое слово **`this`**, являющееся указателем на данный объект класса. Поэтому если переопределяется бинарная операция, то переопределяющий ее метод класса должен принимать только один параметр, а если унарная, то метод класса вообще не имеет параметров.

Если операция переопределяется при помощи дружественной функции, то она должна принимать соответственно два и один параметр.

При переопределении операций необходимо учитывать некоторые имеющиеся в языке ограничения:

- нельзя изменять количество параметров операции;
- нельзя изменить приоритет операций;
- нельзя определить новые операции;
- нельзя использовать параметры по умолчанию;

Полиморфизм

Прямым результатом **наследования** явилось включение в язык **виртуальных функций**.

Наследование — некоторый класс может передавать свои компоненты другому классу. При этом первый класс называется базовым, а второй — производным классом (или подклассом).

Чтобы разобраться, о чем идет речь, рассмотрим пример:

```
class BaseClass
{
public:
void Method();
};
```

```

class DrivedClass : public BaseClass
{
public:
void Method();
};
void SimpleFunc(BaseClass *ptrBase)
{
ptrBase->Method();
}

```

А теперь попробуйте ответить на вопрос: "К какому методу **Method** относится вызов в функции **SimpleFunc**?" Правильно, к методу базового класса. И для того, чтобы вызвать метод производного класса, необходимо использование операции разрешения контекста. Что же делать, чтобы иметь возможность вызывать оба метода? Ответ на этот вопрос дает **полиморфизм** — использование одного и того же вызова для ссылки на разные методы в зависимости от типа объекта. Для поддержки полиморфизма язык программирования должен поддерживать механизм позднего связывания, при котором решение о том, какой именно метод с одним и тем же именем должен быть вызван, принимается в зависимости от типа объекта уже во время выполнения программы.

Библиотека потоков

Библиотека потоков данных TPL модель потоков данных поддерживающая программирование на основе субъектов путем обеспечения в процессе передачи сообщений для не детализированного потока данных и задач по конвейеризации. Компоненты потока данных основываются на типах и инфраструктуре планирования TPL и интегрировать с поддержкой языка C#, Visual Basic для асинхронного программирования. Эти компоненты потоков данных полезны при наличии нескольких операций, которые должны асинхронно взаимодействовать друг с другом, или при необходимости обрабатывать данные по мере того, как они становятся доступными.

Базовые элементы программы на C

- Main Menu
- ToolBars
- Symbol Window
- Symbol Panel
- File Tab Bar
- Code Editor
- File Tree Window
- Messange Window

32. Написание и компиляция простых программ на C/C++. Написание вашей первой программы. Пример простой программы на C. Структура простой программы. Как сделать программу читаемой. Подготовка и компиляция простых программ на C/C++.

Компиляция — трансляция программы, составленной на исходном языке высокого уровня, в эквивалентную программу на низкоуровневом языке, близком машинному коду (абсолютный код, объектный модуль, иногда на язык ассемблера), выполняемая компилятором

Пример простой программы на C

```
#include <stdio.h>

int main (void)
{
    puts ("Hello, World!");
    return 0;
}
```

Структура программы на языке С Программа может состоять из одной или нескольких, связанных между собой, функций, главная из которых называется **main** – именно с нее начинается выполнение программы. Поэтому, наличие функции с таким именем в любой программе. Описание функции состоит из заголовка и тела. Заголовков в свою очередь состоит из директив препроцессора типа **#include** и т. д. и имени функции.

НЕСКОЛЬКО СОВЕТОВ, КАК СДЕЛАТЬ ПРОГРАММУ ЧИТАЕМОЙ

Создание читаемой программы служит признаком хорошего стиля программирования. Это приводит к облегчению понимания смысла программы, поиска ошибок и в случае необходимости ее модификации. Действия, связанные с улучшением читаемости программы, кроме того, помогут более четко понять, что программа делает.

- выбор осмысленных обозначений для переменных
- использование комментариев
- помещать каждый оператор на отдельной строке
- использовании пустых строк (для того, чтобы отделить одну часть функции, соответствующую некоторому семантическому понятию, от другой. Например, в нашей простой программе одна пустая строка отделяет описательную часть от выполняемой (присваивание значения и вывод на печать). Синтаксические правила языка Си не требуют наличия пустой строки в данном месте, но поскольку это стало уже традицией, то и мы делаем также.)

Компиляция простых программ на C/C++

Исполнить исходные файлы нельзя, их необходимо **скомпилировать**, т.е. создать исполняемый файл, содержащий в себе инструкции процессора и пригодный для запуска на компьютере.

Процесс преобразования исходных файлов в исполняемый называется *компиляцией*. Если ваша программа состоит из одного исходного файла `hello.c`, то для его компиляции достаточно выполнить команду:

```
bash$ gcc hello.c -o hello
```

В результате получится файл `hello`, имя которого мы указали в опции `-o`. Этот файл является исполняемым и его можно запускать (**execute**) при помощи команды:

```
bash$ ./hello
```

Пара символов `./` перед `hello` означает "искать исполняемый файл `hello` в текущей директории".

Строчка

```
bash$ gcc xxx.c yyy.c -o zzz -I./common -I.. -lm
```

соответствует команде: "скомпилировать файлы xxx.c ууу.c в программу zzz; заголовочные файлы находятся в директориях ./common и ..; подключить библиотеку libm"

Библиотека libm (подключаемая с помощью опции `-lm`) содержит откомпилированные математические функции, которые объявляются в заголовочном файле `math.h`. Если вы используете функции из этой библиотеки (такие как `log`, `sin`, `cos`, `exp`), то не забывайте подключать её при компиляции.

33. Написание и компиляция простых программ на C/C++. Редактирование текста программы. Сохранение программ. Построение программы.

Компиляция — трансляция программы, составленной на исходном языке высокого уровня, в эквивалентную программу на низкоуровневом языке, близком машинному коду (абсолютный код, объектный модуль, иногда на язык ассемблера), выполняемая компилятором

Сохранение файла

Желательно сохранить файл до того, как вы приступите к его компиляции и компоновке, а тем более до того, как попытаетесь запустить программу на выполнение. Чтобы сохранить введенный только что код, вы можете либо щелкнуть на третьей кнопке слева на стандартной панели инструментов, либо выбрать в меню File команду Save, либо нажать [Ctrl+S]. Когда вы в первый раз выбираете команду Save, открывается диалоговое окно Save.

Построить проекты Visual C++ можно двумя способами:

- с помощью Visual Studio;
- с помощью командной строки.

При построении приложения Visual C++ в Visual Studio в диалоговом окне "Окна свойств" проекта можно изменить множество параметров построения.

Для программистов, которые предпочитают выполнять построение приложений из командной строки, в Visual C++ представлены средства командной строки. Для построения проекта Visual C++ можно использовать следующие средства командной строки:

- DEVENV.EXE (Команда Devenv предоставляет возможность установки из командной строки различных параметров для интегрированной среды разработки (IDE), а также для компиляции, построения и отладки проектов. Используйте эти переключатели для запуска IDE из файла сценария или из BAT-файла, например сценария построения программы в ночное время, либо для запуска IDE в особой конфигурации.)
- NMAKE.EXE (Утилита построения программ (Майкрософт) NMAKE.EXE — это средство, предназначенное для построения проектов на основании команд, содержащихся в файле описания.)
- Справочник по программе VCBUILD (Программа VCBUILD.exe может использоваться для построения проектов Visual C++ и решений Visual Studio из командной строки. Использование этого средства аналогично выполнению команд Построить проект или Построить решение в интерфейсе интегрированной среды разработки Visual Studio.)

Чтобы получить справку по предупреждениям, ошибкам и сообщениям, отображаемым в процессе построения из командной строки, запустите среду разработки и выберите в меню **Справка** команды **Указатель** или **Поиск**.

34. Написание и компиляция простых программ на C/C++. Использование утилиты Project Workspace. Создание нового проекта. Добавление файлов к проекту. Запуск команд Build или Rebuild All.

Компиляция — трансляция программы, составленной на исходном языке высокого уровня, в эквивалентную программу на низкоуровневом языке, близком машинному коду (абсолютный код, объектный модуль, иногда на язык ассемблера), выполняемая компилятором

Workspace

Каталог Workspace содержит такие файлы проекта, как источники данных, страницы, утилиты и загрузки.

Обзор

Каталог Workspace содержит все файлы проекта Symphony. По умолчанию там сохраняются источники данных, события, страницы, утилиты и файлы загружаемые пользователями. Часто разработчики используют данный каталог для хранения различных вспомогательных файлов, таких как CSS и JavaScript или картинок шаблонов.

Использование

Пользователи могут создавать любую структуру подкаталогов в каталоге workspace.

URL адрес каталога workspace включён в системные параметры Symphony.

Детали

По умолчанию система создаёт и использует четыре подкаталога в каталоге **workspace**:

/data-sources

/events

/pages

/utilities

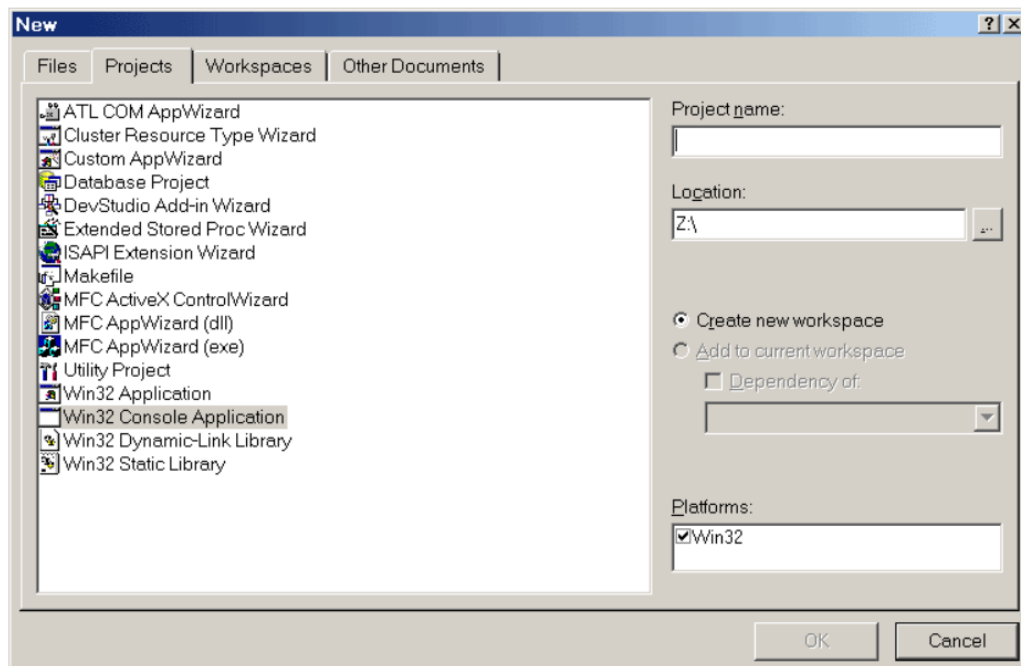
Файлы физически расположенные в данных подкаталогах могут быть отредактированы непосредственно в панели управления администратора.

При необходимости могут быть созданы дополнительные подкаталоги. На практике довольно часто создаются вспомогательные каталоги (такие, как /styles, /images и /scripts) и /uploads для хранения загружаемого пользовательского контента.

Создание нового проекта

При программировании на C++ существует понятие **проекта**. Проект может объединять в себя несколько файлов с текстом программы, но после компиляции проекта получается один exe-модуль. В консольном приложении работа проекта начинается с вызова функции main, а уже из нее идет вызов всех остальных функций, которые могут находиться в разных файлах.

Для создания нового проекта нужно выбрать пункт меню File->New, затем на закладке Projects выбрать тип проекта - Win32 Console Application, в поле Location написать путь к проекту (на своем диске!), в поле Project Name - имя проекта (например, test).

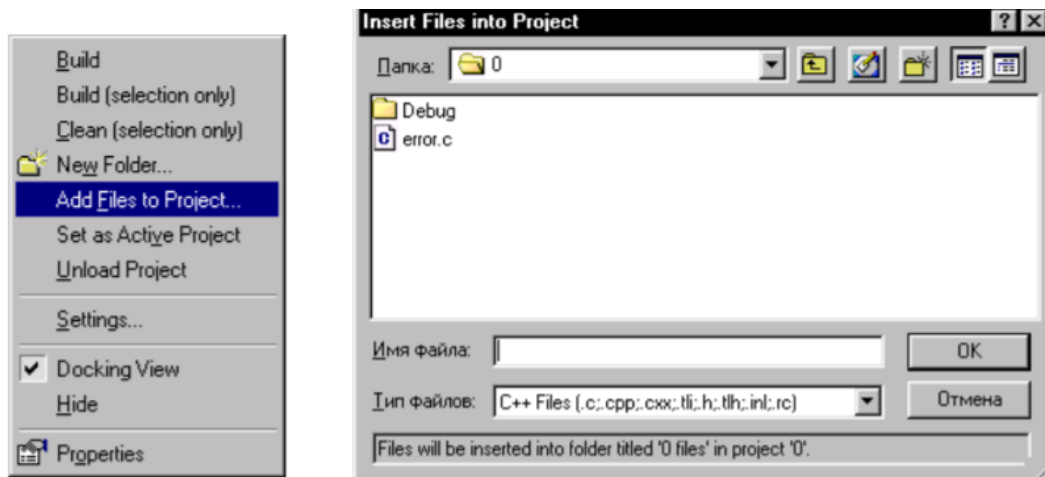


После этого запустится мастер создания приложений, который для консольного приложения состоит из одного шага. Можно сгенерировать:

- An empty project - пустой проект, который не будет содержать файлов с текстом программы;
- A simple application - проект содержит сгенерированные автоматически файлы с минимальным текстом программы и комментариями;
- A "Hello, World" application - генерируется проект, выводящий на экран надпись "Hello, World".

Добавление файлов в проект

После того как проект создан, в него можно сразу же добавлять новые файлы. Перейдите на вкладку FileView и щелкните правой кнопкой мыши на элементе ERRORfiles. Контекстное меню с выделенной командой Add Files to Project.... При выборе данной команды появляется окно InsertFilesintoProject, где вы можете отметить файлы, подлежащие включению в проект. Одно замечание по поводу типа файлов: файлы заголовков (с расширением H) не включаются в список файлов проекта, а добавляются в проект непосредственно во время построения программы с помощью директив препроцессора #include. В нашем случае нужно найти созданный ранее файл ERROR.C и выполнить на нем двойной щелчок, в результате чего файл автоматически будет добавлен в проект.



Запуск команд **Build** или **Rebuild All**.

Команда **Build** (ей соответствует комбинация клавиш <Alt>+<F9>) строит проект, проверяя метки времени всех исходных файлов в проекте, поэтому если исходные файлы (или файлы, которые в них включены) являются более новыми, чем зависимые **OBJ**-файлы, то соответствующие модули проекта будут перекомпилированы.

Команда **Rebuild All** (ей соответствует комбинация клавиш <Ctrl>+<Alt>+<F9>) выполняет то же действие, что и **Build**, причем все файлы будут повторно сгенерированы или откомпилированы и скомпонованы независимо от их меток времени.

35. Написание и компиляция простых программ на C/C++. Отладка программы. Понимание сообщений об ошибках и предупреждений. Распространенная ошибка при использовании нового языка. Переключение между окном вывода сообщений и окном редактирования.

Использование функций замены или быстрого поиска. Выбор опций замены

Компиляция — трансляция программы, составленной на исходном языке высокого уровня, в эквивалентную программу на низкоуровневом языке, близком машинному коду (абсолютный код, объектный модуль, иногда на язык ассемблера), выполняемая компилятором

Отладка — этап разработки компьютерной программы, на котором обнаруживают, локализуют и устраняют ошибки. Чтобы понять, где возникла ошибка, приходится:

-узнавать текущие значения переменных;

-выяснять, по какому пути выполнялась программа.

Типы ошибок при отладке:

- 1) предупреждения компилятора
- 2) ошибки компилятора
- 3) ошибки компоновщика

Предупреждения компилятора – несерьёзные ошибки, которые не препятствуют компиляции программы. Предупреждения компилятора — это признак того, что что-то может пойти не так во время выполнения. Например, в условии =, а не == .

Ошибки — это условия, которые препятствуют завершению компиляции ваших файлов. Ошибки компилятора ограничены отдельными файлами исходного кода и являются результатом “синтаксических ошибок”. Например, выражение for(;;). Ошибки компилятора всегда будут включать номер строки, в которой была обнаружена ошибка.

Ошибки компоновщика — это, проблемы с поиском определения функций, структур, классов или глобальных переменных, которые были объявлены, но не определены, в файле исходного кода.

Существуют две взаимодополняющие технологии отладки:

-**Использование отладчиков** — программ, которые включают в себя пользовательский интерфейс для пошагового выполнения программы: оператор за оператором, функция за функцией, с остановками на некоторых строках исходного кода или при достижении определённого условия.

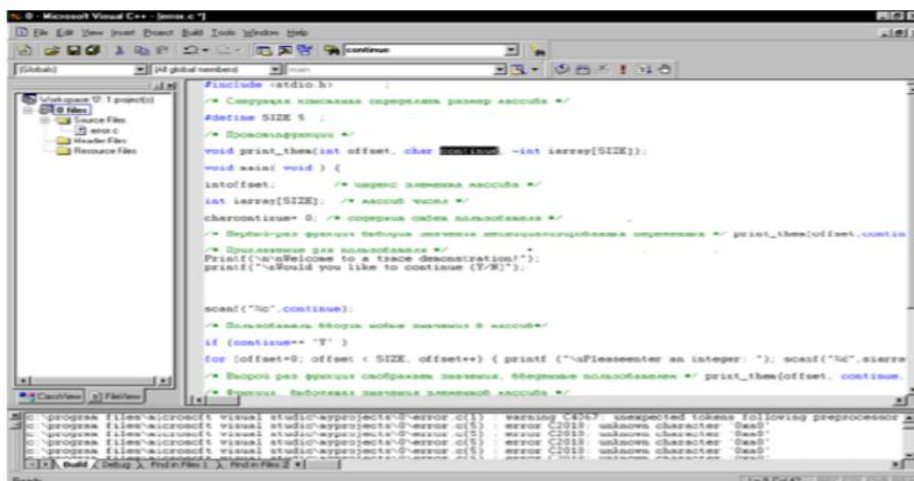
-**Вывод текущего состояния программы** с помощью расположенных в критических точках программы операторов вывода — на экран, принтер, громкоговоритель или в файл. Вывод отладочных сведений в файл называется журналированием.

Распространенная ошибка при использовании нового языка: незнание плюсов и минусов языка, его особенностей.

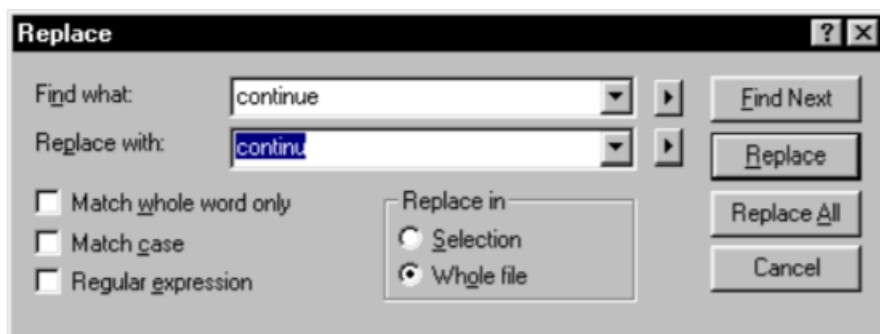
Переключение между окном вывода и окном редактирования происходит либо мышкой, либо с помощью комбинации клавиш Alt + Tab.

Использование команд Find и Replace

Довольно часто в процессе программирования возникают ситуации, когда вам нужно найти и заменить какое-то слово в тексте программы. Вы, конечно же, можете сделать это с помощью диалогового окна, открываемого командой Replace... из меню Edit, но имеется и более быстрый способ. Рассмотрите внимательно панель инструментов, и найдите в поле списка Find слово continue. Чтобы воспользоваться этим средством поиска, щелкните мышью в поле и введите слово, которое хотите найти, после чего нажмите [Enter]. На рис. показан результат такого поиска. В тексте программы выделено слово continue, обнаруженное первым.



Данный метод достаточно удобен для поиска нужного слова. Но наша задача этим не ограничивается, поскольку имя переменной continue нам необходимо заменить во всей программе другим именем. В таком случае целесообразнее воспользоваться командой Replace... из меню Edit



Наша цель состоит в том, чтобы заменить имя переменной continue словом, также указывало бы на назначение этой переменной, но отличалось бы от ервированных имен. С этой целью введем в поле Replacewith слово continu. Но осталась маленькая проблема. В программе имеется строка "\nWould you like to continue (Y/N)". Если вы выполните автоматическую замену во всем файле, щелкнув на кнопке ReplaceAll, то сообщение, выдаваемое программой, будет содержать, грамматическую ошибку. Поэтому замену следует проводить последовательно, переходя от слова к слову, а в указанном месте щелкнуть на кнопке FindNext.

36. Написание и компиляция простых программ на C/C++. Переключение между окном вывода сообщений и окном редактирования. Быстрый способ. Значение сообщений об ошибках и предупреждений. Повторная сборка программы ERROR.C. Запуск программы.

Компиляция — трансляция программы, составленной на исходном языке высокого уровня, в эквивалентную программу на низкоуровневом языке, близком машинному коду (абсолютный код, объектный модуль, иногда на язык ассемблера), выполняемая компилятором

Переключение между окном вывода и окном редактирования происходит либо мышкой, либо с помощью комбинации клавиш Alt + Tab.

Типы ошибок при отладке:

- 1) предупреждения компилятора
- 2) ошибки компилятора
- 3) ошибки компоновщика

Предупреждения компилятора – несерьёзные ошибки, которые не препятствуют компиляции программы. Предупреждения компилятора — это признак того, что что-то может пойти не так во время выполнения. Например, в условии =, а не == .

Ошибки — это условия, которые препятствуют завершению компиляции ваших файлов. Ошибки компилятора ограничены отдельными файлами исходного кода и являются результатом “синтаксических ошибок”. Например, выражение for(;;). Ошибки компилятора всегда будут включать номер строки, в которой была обнаружена ошибка.

Ошибки компоновщика — это, проблемы с поиском определения функций, структур, классов или глобальных переменных, которые были объявлены, но не определены, в файле исходного кода.

Запуск программы

Чтобы запустить программу, просто выберите в меню Project команду Execute.

Для Visual Studio: запуск программы с отладкой – F5, без отладки – Ctrl + F5.

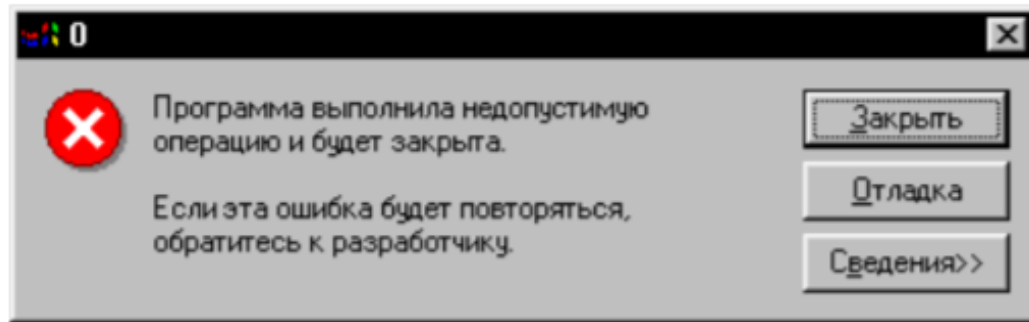
37. Написание и компиляция простых программ на C/C++. Использование встроенного отладчика. Использование команд пошагового выполнения (Step Into и Step Over). Определение точек останова (breakpoints). Запуск программы с точками останова. Использование быстрого просмотра

Компиляция — трансляция программы, составленной на исходном языке высокого уровня, в эквивалентную программу на низкоуровневом языке, близком машинному коду (абсолютный код, объектный модуль, иногда на язык ассемблера), выполняемая компилятором.

Использование встроенного отладчика

Созданная нами программа в начале своей работы отображает на экране исходное содержимое массива данных, после чего спрашивает, хотите ли вы продолжить работу. Ответ Y (yes — да) сигнализирует о том, что вы хотите заполнить массив собственными данными и отобразить их на экране. Из рис. можно сделать вывод о том, что хотя программный код набран совершенно правильно, т.е. в нем нет синтаксических ошибок, программа работает не так, как нам бы хотелось. Ошибки такого рода называются логическими. К счастью, встроенный в VisualC++ отладчик содержит ряд средств, которые послужат для вас спасательным кругом в подобной ситуации. Во-первых, вы можете выполнять

программу пошагово, строка за строкой. Во-вторых, вам предоставляется возможность анализировать значения переменных в любой момент выполнения программы



Разница между командами Step Into и Step Over

Когда вы начинаете процесс отладки, появляется панель инструментов Debug. Из множества представленных на ней кнопок наиболее часто задействуются Step Into (четвертая справа в верхнем ряду) и Step Over (третья справа). В обоих случаях программа будет запущена на выполнение в пошаговом режиме, а в тексте программы выделяется та строка, которая сейчас будет выполнена. Различия между командами Step Into и Step Over проявляются только тогда, когда в программе встречается вызов функции. Если выбрать команду Step Into, то отладчик войдет в функцию и начнет выполнять шаг за шагом все ее операторы. При выборе команды Step Over отладчик выполнит функцию как единое целое и перейдет к строке, следующей за вызовом функции. Эту команду удобно применять в тех случаях, когда в программе делается обращение к стандартной функции или созданной вами подпрограмме, которая уже была протестирована. Давайте выполним пошаговую отладку нашей программы.

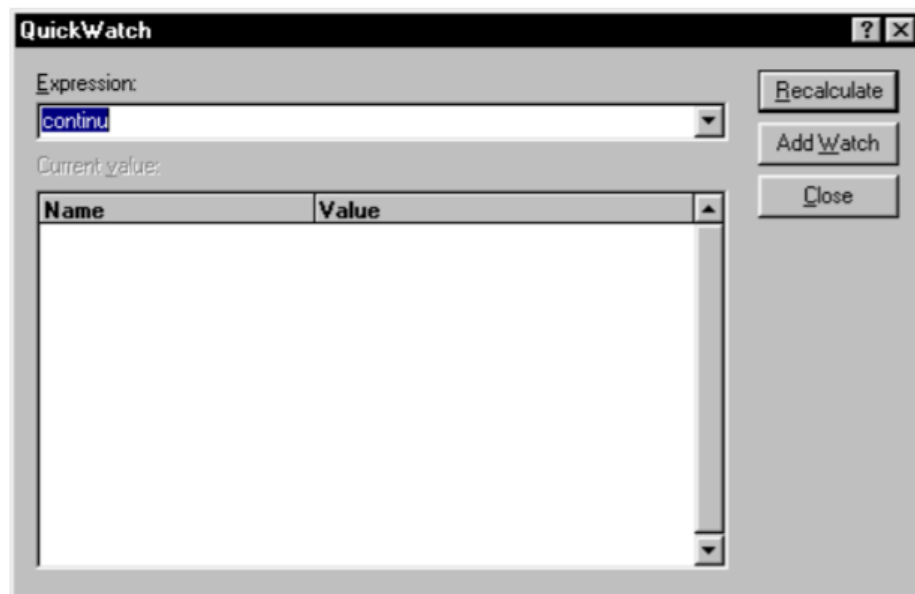
Точка останова — это преднамеренное прерывание выполнения программы, при котором выполняется вызов отладчика

Работа с точками останова

Предположим, что вы поставили точку останова в строке программы, содержащей вызов функции `scanf()`. Теперь выберите команду Go — либо из меню, либо нажав клавишу [F5]. Обратите внимание, что выполнение программы прерывается не на первой строке программы, а на строке, содержащей точку останова. Далее можно продолжить выполнение программы в пошаговом режиме либо проанализировать текущие значения переменных. Нас интересует, будет ли функция `scanf()` работать корректно после того, как в программный код были внесены изменения. Выберите команду Step Over, перейдите к окну программы, введите букву Y в верхнем регистре и нажмите клавишу [Enter]. (Мы применили команду Step Over для того, чтобы избежать пошагового анализа отладчиком всех операторов функции `scanf()`). При выборе команды Step In появляется предложение указать местонахождение файла SCANF.C) Все отлично! Отладчик не выдал окна с сообщением об ошибке. Но означает ли это, что все проблемы разрешены? Чтобы ответить на этот вопрос, достаточно будет проанализировать текущее значение переменной `contin`.

Окно QuickWatch,

Команда **QuickWatch...** открывает диалоговое окно **QuickWatch**, которое позволяет по ходу выполнения программы анализировать значения переменных. Простейший способ определить значение переменной с помощью данного окна состоит в том, что курсор помещается на имени переменной в окне редактирования, а затем нажимается комбинация клавиш [Shift+F9].



38. Данные. Идентификаторы. Ключевые слова. Символы. Данные: переменные и константы. Данные: типы данных.

Данные — это любая информация, представленная в формализованном виде и пригодная для обработки алгоритмом.

Данные делятся на **переменные** и **константы**.

Переменные — это такие данные, значения которых могут изменяться в процессе выполнения алгоритма.

Константы — это данные, значения которых не меняются в процессе выполнения алгоритма.

Типы данных принято делить на **простые (базовые)** и **структурированные**.

К основным базовым типам относятся:

- **целый (INTEGER)** — определяет подмножество допустимых значений из множества целых чисел;
- **вещественный (REAL, FLOAT)** — определяет подмножество допустимых значений из множества вещественных чисел;
- **логический (BOOLEAN)** — множество допустимых значений — истина и ложь;
- **символьный (CHAR)** — цифры, буквы, знаки препинания и пр.

Структурированные типы описывают наборы однотипных или разнотипных данных, с которыми алгоритм должен работать как с одной именованной переменной.

Наиболее широко известная структура данных — **Массив**. **Массив** -представляет собой упорядоченную структуру однотипных данных, которые называются элементами массива.

Идентификаторы

Идентификаторами называются имена, присваиваемые : переменным, константам, типам данных и функциям, используемым в программах. После описания идентификатора можно ссылаться на обозначаемую им сущность в любом месте программы. Идентификатор представляет собой последовательность символов произвольной длины, содержащую буквы, цифры и символы подчеркивания, но начинающуюся обязательно с буквы или символа подчеркивания. Компилятор распознает только первые 31 символ.

Ключевые слова

Ключевые слова являются встроенными идентификаторами, каждому из которых соответствует определенное действие. Изменить назначение ключевого слова нельзя. (С помощью директивы препроцессора **#define** можно создать "псевдоним" ключевого слова, который будет дублировать его действия, возможно, с некоторыми изменениями.) Помните, что имена идентификаторов, создаваемых в программе, не могут совпадать с ключевыми словами языков C/C++

39. Три целочисленных типа. Описание данных целого типа. Целые константы. Инициализация переменных целого типа. Модификатор unsigned.

Три типа целых чисел в C/C++ поддерживаются три типа целых чисел. Наравне со стандартным типом `int` существуют типы `shortint` (короткое целое) и `longint` (длинное целое). Допускается сокращенная запись `short` и `long`. Хотя синтаксис самого языка не зависит от используемой платформы, размерность типов данных `short`, `int` и `long` может варьироваться. Гарантируется лишь, что соотношение размерностей таково: `short` ≤ `int` ≤ `long`. В Microsoft Visual C/C++ для переменных типа `short` резервируется 2 байта, для типов `int` и `long` — 4 байта

Описание данных целого типа.

При описании данных необходимо ввести только тип, за которым должен следовать список имен переменных. Ниже приведены некоторые возможные примеры описаний:

```
int erns;  
short stops;  
long Johns;  
int hogs, cows, goats;
```

В качестве разделителя между именами переменных необходимо использовать запятую; весь список должен оканчиваться символом «точка с запятой». Вы можете собрать в один оператор описания переменных с одним и тем же типом или, наоборот, разбить одно описание на несколько операторов. Например, описание

```
int erns, hogs, cows, goats;
```

будет давать тот же самый эффект, что и два отдельных описания типа `int` в предшествующем примере. При желании вы даже могли бы использовать четыре различных описания данных типа `int` — по одному для каждой переменной. Иногда вам могут встретиться сочетания ключевых слов, как, например, `long int` или `short int`. Эти комбинации являются просто более длинной записью ключевых слов `long` и `short`.

Целая константа

Может быть записана в десятичной,восмеричной или шестнадцатиричной системе счисления.

В десятичной записывается как обычное десятичное число,при условии,что первая цифра не является нулём.

В восмеричной целая константа записывается восмеричными цифрами и должна начинаться с нуля.

В шестнадцатиричной целая константа записывается шестнадцатиричными цифрами и должна начинаться с символов 0x или 0X.При этом для обозначения цифр от 10 до 15 используются буквы A,B,C,D,F

Модификаторы unsigned

Компиляторы языков C/C++ позволяют при описании переменных некоторых типов указывать модификатор unsigned. В настоящее время он применяется с четырьмя типами данных: char, short, int и long. Наличие данного модификатора указывает на то, что значение переменной должно интерпретироваться как беззнаковое число, т.е. самый старший бит является битом данных, а не битом знака. Предположим, мы создали новый тип данных, *my_octal*, в котором для записи каждого числа выделяется 3 бита. По умолчанию считается, что значения этого типа могут быть как положительными, так и отрицательными. Поскольку из 3-х доступных битов старший будет указывать на наличие знака, то диапазон возможных значений получится от -4 до 3. Но если при описании переменной типа *my_octal* указать модификатор unsigned, то тем самым мы освободим первый бит для хранения полезной информации и получим диапазон возможных значений от 0 до 7.

40.Числа с плавающей точкой. Описание переменных с плавающей точкой. Перечисляемый тип данных (enum).

В большинстве проектов разработки программного обеспечения оказывается вполне достаточным использовать данные целых типов. Однако в программах вычислительного характера часто применяются числа с плавающей точкой. В языке Си такие данные описываются типом **float** - с плавающей точкой одинарной точности; **double** – с плавающей точкой двойной точности;**long double** - тип с плавающей точкой повышенной точности;

Описание переменных с плавающей точкой.

Переменные с плавающей точкой описываются и инициализируются точно таким же образом, что и переменные целого типа. Ниже приведено несколько примеров:

```
float noah, jonah;  
  
double trouble;
```

Перечисляемый тип данных (enum)

При объявлении переменной перечисляемого типа с ней связывается набор именованных целочисленных констант, называемых перечисляемыми константами. В каждый момент времени переменная может иметь значение одной из констант, и к ней можно обращаться по имени. К примеру, следующее описание создает перечисляемый тип *air_supply*, перечисляемые константы *EMPTY*, *USEABLE* и *FULL*, и перечисляемую переменную *instructor_tank*:

```
enum air_supply {EMPTY,  
USEABLE,
```

```
FULL=5 } instructor_tank;
```

Все константы и переменные имеют тип `int`, каждая константа автоматически получает по умолчанию некоторое значение, если не указывается какое-либо другое. В предыдущем примере константа `EMPTY` имеет по умолчанию целое значение ноль, так как она — первая в списке и не переопределена особо. Значение константы `USEABLE` равно 1, поскольку она следует непосредственно за константой с нулевым значением. Константа `FULL` инициализирована отдельно и имеет значение 5; если бы вслед за ней в списке была еще одна константа, эта новая константа была бы равна 6.

После создания типа `air_supply` вы можете в дальнейшем описать другую переменную, `student_tank`, следующим образом:

```
enum air_supply student_tank;
```

После этого оператора можно выполнить следующие:

```
instructor_tank = FULL;
```

```
student_tank = EMPTY;
```

При этом переменная `instructor_tank` получит значение 5, а переменная `student_tank` — значение 0.

41. Модификаторы доступа. Модификатор `const`. Определение констант через `#define`. Модификатор `volatile`. Совместное использование `const` и `volatile`.

Модификаторы доступа.

Модификаторы `const` и `volatile` являются новыми в C и C++. Они были добавлены в стандарте ANSI C для того, чтобы различать переменные, которые никогда не изменяются (**`const`**), и переменные, значение которых может измениться непредсказуемо (**`volatile`**).

Модификатор `const`.

Иногда необходимо использовать некоторую величину, которая не изменяется на протяжении всей программы. Такая величина называется константой. Например, если в программе вычисляется площадь круга и длина окружности, часто будет использоваться константа `pi=3.14159`. При финансовых расчетах константой может быть процентная ставка. В подобных случаях можно улучшить читаемость программы, если дать константам осмысленные имена.

В C и C++ для объявления константы перед ключевым словом (`int`, `float` или `double`) пишется `const`. К примеру:

```
const int iMIN=1, iSALE_PERCENTAGE=25;
```

```
const float fbase_change=32.157;
```

```
int irow_index=1, itotal=100, lobject;
```

```
double ddistance=0, dvelocity;
```

Определение констант через `#define`

В C и C++ существует другой способ описания констант: директива препроцессора **`#define`**.

Предположим, в начале программы введена такая строка: **`#define SALES_TEAM 10`**

Общий формат данной строки следующий: директива `#define`, литерал `sales_team`(имя константы), литерал `10` (значение константы). Встретив такую команду, препроцессор осуществит глобальную замену в программном коде имени `sales_team` числом `10`. Причем никакие другие значения идентификатору `sales_team` не могут быть присвоены, поскольку переменная с таким именем в программе формально не описана. В результате идентификатор `sales_team` может использоваться в программе в качестве константы. Обратите внимание, что строка с директивой `#define` не заканчивается точкой с запятой. После числа `10` этот символ будет воспринят как часть идентификатора, в результате чего все вхождения `SALES_TEAM` будут заменены литералом `10`;

Модификатор `volatile`

Ключевое слово **`volatile`** указывает на то, что данная переменная в любой момент может быть изменена в результате выполнения внешних действий, не контролируемых программой. Например, следующая строка описывает переменную `event_time`, значение которой может измениться без ведома программы:

`volatile int event_time;`

Подобное описание переменной может понадобиться в том случае, когда переменная `event_time` обновляется системными устройствами, например таймером. При получении сигнала от таймера выполнение программы прерывается и значение переменной изменяется.

Одновременное использование `const` и `volatile`

Допускается одновременное употребление ключевых слов **`const`** и **`volatile`** при объявлении переменных. Так, в следующей строке создается переменная, обновляемая извне, но значение которой не может быть изменено самой программой:

`const volatile constant_event_time;`

Таким способом реализуются два важных момента. Во-первых, в случае обнаружения компилятором выражения, присваивающего переменной `constant_event_time` какое-нибудь значение, будет выдано сообщение об ошибке. Во-вторых, компилятор не будет выполнять оптимизацию, связанную с подстановкой вместо адреса переменной `constant_event_time` ее действительного значения, поскольку во время выполнения программы это значение в любой момент может быть изменено.

42. Модификаторы `pascal`, `cdecl`, `near`, `far` и `huge`. Модификатор `pascal`. Модификатор `cdecl`. Модификаторы `near`, `far` и `huge`.

Первые два модификатора, `pascal` и `cdecl`, чаще всего используются в сложных приложениях. Microsoft Visual C/C++ позволяет создавать программы, которые с легкостью могут вызывать процедуры, написанные на разных языках. Также возможна связь и в обратном направлении. Например, можно написать программу на Паскале, вызывающую процедуру C++. При подобном смешении языков необходимо учитывать два момента: имена идентификаторов и способ передачи параметров.

Когда Microsoft Visual C/C++ компилирует программу, все глобальные идентификаторы программы (функции и переменные) помещаются в выходной файл с объектным кодом для последующей компоновки. По умолчанию компилятор сохраняет эти идентификаторы, используя те же буквы, которые использовались в описаниях (заглавные, строчные или и те, и другие). Кроме этого, в начале идентификатора добавляется символ подчеркивания (`_`). Так как встроенный компоновщик Microsoft Visual C/C++ различает (по умолчанию) заглавные и строчные буквы, предполагается, что все внешние идентификаторы, объявленные в программе, сохраняют символ подчеркивания и тот вид, который они имели при объявлении (название и регистр букв).

Pascal

Применение модификатора **`pascal`** к идентификатору приводит к тому, что идентификатор преобразуется к верхнему регистру и к нему не добавляется символ подчеркивания. Этот идентификатор может использоваться для именования в программе на языке Си глобального объекта, который используется также в программе на языке Паскаль. В объектном коде, сгенерированном компилятором

языка Си, и в объектном коде, сгенерированном компилятором языка Паскаль, идентификатор будет представлен идентично. Функции типа **pascal** не могут иметь переменное число аргументов, как, например, функция **printf**. Поэтому нельзя использовать завершающее многоточие в списке параметров функции типа **pascal**.

Cdecl

Существует опция компиляции, которая присваивает всем функциям и указателям на функции тип **pascal**. Это значит, что они будут использовать вызывающую последовательность, принятую в языке Паскаль, а их идентификаторы будут приемлемы для вызова из программы на Паскале. При этом можно указать, что некоторые функции и указатели на функции используют вызывающую последовательность, принятую в языке Си, а их идентификаторы имеют традиционный вид для идентификаторов языка Си. Для этого их объявления должны содержать модификатор **cdecl**.

Near, Far, Huge

Эти модификаторы оказывают воздействие на работу с адресами объектов.

Указатель типа **near** — 16-битовый

Указатель типа **far** — 32-битовый

Указатель типа **huge** — 32-битовый

—операции отношения **==**, **!=**, **<**, **>**, **<=**, **>=** выполняются корректно и предсказуемо над указателями типа **huge**, но не над указателями типа **far**;

—при использовании указателей типа **huge** требуется дополнительное время, т. к. программы нормализации должны вызываться при выполнении любой арифметической операции над этими указателями. Объем кода программы также возрастает.

В СП MSC модификатор **huge** применяется только к массивам, размер которых превышает 64 К. В СП TC недопустимы массивы больше 64 К, а модификатор **huge** применяется к функциям и указателям для спецификации того, что адрес функции или указуемого объекта имеет тип **huge**.

Для вызова функции типа **near** используются машинные инструкции ближнего вызова, для типов **far** и **huge** — дальнего.

43. Тип данных char Другие типы и размеры данных. Преобразование типов данных. Явные преобразования типов при помощи операции приведения типа.

Тип данных char

Этот тип определяет целые числа без знака в диапазоне от 0 до 255. Обычно такое целое размещается в одном байте памяти.

```
char response;
```

```
char intable, latan;
```

```
char isma = 'S';
```

Тип	Типичный размер в битах
char	8
unsigned char	8
signed char	8
int	16 или 32
unsigned int	16 или 32
signed int	16 или 32
short int	16
unsigned short int	16
signed short int	16
long int	32
long long int	64
signed long int	32
unsigned long int	32
unsigned long long int	64
float	32
double	64
long double	80

Преобразование типов данных.

Предположим, что выполняется следующая операция, в которой переменные **fresult** и **fvalue** имеют тип float, а переменная **ivalue** — тип int:

fresult = fvalue * ivalue;

Следовательно, это операция смешанного типа. При выполнении оператора значение **ivalue** будет преобразовано в число с плавающей точкой, а затем выполнится умножение. Компилятор распознает смешанные операции и генерирует код для выполнения следующих операций.

Давайте посмотрим, что происходит при преобразовании типа **float** в тип int. Предположим, что описаны переменные **ivalue1** и **ivalue2** типа int и переменные **fvalue** и **fresult** типа float. Рассмотрим следующую последовательность операторов:

Ivalue1 = 3;

ivalue2 = 4;

fvalue = 7.0;

fresult = fvalue + ivalue1/ivalue2;

Операция **ivalue1/ivalue2** — не смешанная; она представляет собой деление двух целых чисел, и ее результат 0, так как при выполнении целочисленного деления дробная часть (в этом примере 0.75) отбрасывается. Поэтому значение **fresult** равно 7.0.

Что будет, если переменную **ivalue2** описать как float? В этом случае переменная **fresult** получит значение с плавающей точкой 7.75, поскольку операция **ivalue1/ivalue2** будет иметь смешанный тип. Тогда значение **ivalue1** временно преобразуется в число с плавающей точкой 3.0 и результат деления будет равен 0.75. После сложения с **fvalue** получим 7.75.

Явные преобразования типов при помощи операции приведения типа.

Как вы видели, компилятор C и C++ автоматически меняет формат переменной при выполнении смешанных операций, в которых задействуются различные типы данных. Однако, в некоторых случаях, несмотря на то, что автоматическое преобразование не выполняется, преобразование типов было бы желательным. В таких случаях необходимо особо указать, что требуется изменение типа. Кроме того, подобные явные указания облегчают другим программистам понимание используемых операторов. В языке C имеется несколько процедур, позволяющих указывать необходимость преобразования типа.

Одна из таких процедур называется приведением типа. Если нужно временно изменить формат некоторой переменной, достаточно перед ее именем указать в скобках имя типа, который необходимо получить. Например: если переменные **ivalue1** и **ivalue2** объявлены как int, а переменные **fvalue** и **fresult** как float, то три следующих оператора будут эквивалентны:

fresult = fvalue + (float)ivalue1/ivalue2;

fresult = fvalue + ivalue1/(float)ivalue2;

fresult = fvalue + (float)ivalue1/(float)ivalue2;

Все три оператора выполняют преобразование в тип с плавающей точкой и деление переменных **ivalue1** и **ivalue2**. Если любая из этих переменных приведена в тип float, то — согласно описанным выше правилам стандартного преобразования типов в смешанных арифметических операциях — выполняется деление чисел с плавающей точкой. Третий оператор явно указывает, какие операции должны выполняться.

44. Символьные строки. Массивы символов в С++ Функции работы со строками СИМВОЛОВ.

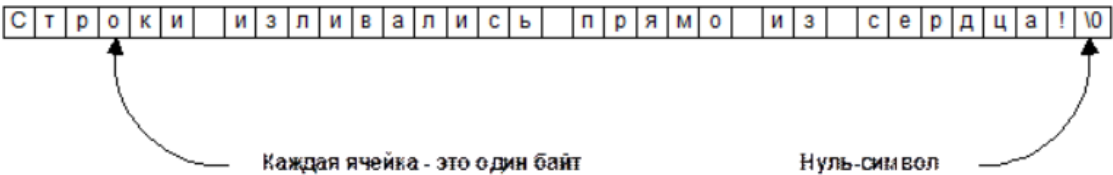
Символьные строки.

«Символьная строка» — это последовательность, состоящая из одного или более символов. В качестве примера рассмотрим следующую строку:

"Строки изливались прямо из сердца!"

Кавычки не являются частью строки. Они вводятся только для того, чтобы отметить ее начало и конец, т. е. играют ту же роль, что и апострофы в случае одиночного символа.

В языке Си нет специального типа, который можно было бы использовать для описания строк. Вместо этого строки представляются в виде «массива» элементов типа char.



Необходимо отметить, что на рисунке последним элементом массива является символ \0. Это «нуль-символ», и в языке Си он используется для того, чтобы отмечать конец строки.

Ну, а теперь спросим, что такое массив? Массив можно представить себе как совокупность нескольких ячеек памяти, объединенных в одну строку. Если вы предпочитаете более формальные и строгие определения, то массив — это упорядоченная последовательность элементов данных одного типа.

```
char name [40];
```

Квадратные скобки указывают, что переменная name — массив, 40 — число его элементов, а char задает тип каждого элемента.

Библиотека С++ содержит функции копирования строк (strcpy, strncpy), сравнения (strcmp, strncmp), объединения строк (strcat, strncat), поиска подстроки (strstr), поиска вхождения символа (strchr, strnchr, strbrk) определения длины строки strlen и другие.

45. Символьные строки. Определение длины строк Копирование и конкатенация строк Сравнение строк Преобразование строк

Функция	Пояснение
strlen (имя_строки)	определяет длину указанной строки, без учёта нуль-символа
Копирование строк	
strcpy (s1,s2)	выполняет побайтное копирование символов из строки s2 в строку s1
strncpy (s1,s2, n)	выполняет побайтное копирование n символов из строки s2 в строку s1. возвращает значения s1

Функция	Пояснение
Объединение строк	
strcat(s1,s2)	объединяет строку s2 со строкой s1. Результат сохраняется в s1
strncat(s1,s2,n)	объединяет n символов строки s2 со строкой s1. Результат сохраняется в s1
Сравнение строк	
strcmp(s1,s2)	сравнивает строку s1 со строкой s2 и возвращает результат типа int: 0 –если строки эквивалентны, >0 – если s1<s2, <0 — если s1>s2 С учётом регистра
strncmp(s1,s2)	сравнивает n символов строки s1 со строкой s2 и возвращает результат типа int: 0 –если строки эквивалентны, >0 – если s1<s2, <0 — если s1>s2 С учётом регистра
stricmp(s1,s2)	сравнивает строку s1 со строкой s2 и возвращает результат типа int : 0 –если строки эквивалентны, >0 – если s1<s2, <0 — если s1>s2 Без учёта регистра
strnicmp(s1,s2)	сравнивает n символов строки s1 со строкой s2 и возвращает результат типа int: 0 –если строки эквивалентны, >0 – если s1<s2, <0 — если s1>s2 Без учёта регистра
Функции преобразования	
atof(s1)	преобразует строку s1 в тип double
atoi(s1)	преобразует строку s1 в тип int
atol(s1)	преобразует строку s1 в тип long int

46.Символьные строки. Обращение строк Поиск символов Поиск подстрок

Функция обращения строки **strrev()** меняет порядок следования символов на обратный (реверс строки). Данная функция имеет прототип:

char* strrev(char* str)

Следующий пример демонстрирует работу функции **strrev()**.

char str[]—"Привет"; cout « strrev(str);

В результате на экране будет выведена строка "тевирП". Эта функция также преобразует строку-оригинал.

Поиск символа в строке.

Для поиска символа в строке используются функции **strchr**, **strrchr**, **strspn**, **strcspn** и **strpbrk**.

Функция char* strchr (const char *str, int c);

ищет первое вхождение символа, заданного параметром c, в строку str. В случае успеха функция **возвращает указатель на первый найденный символ, а в случае неудачи – NULL.**

Функция * strrchr (const char *str, int c);

ищет последнее вхождение символа, заданного параметром c, в строку str. В случае успеха функция **возвращает указатель на последний найденный символ, а в случае неудачи – NULL.**

Функция size_t strspn (const char *str1, const char *str2);

возвращает индекс первого символа из строки str1, который не входит в строку str2.

Функция t strcspn (const char *str1, const char *str2);

возвращает индекс первого символа из строки str1, который входит в строку str2.

Функция char* strpbrk (const char *str1, const char *str2);

находит первый символ в строке str1, который равен одному из символов в строке str2. В случае успеха функция **возвращает указатель на этот символ, а в случае неудачи – NULL.**

Поиск подстрок.

Стандартная библиотека предлагает воспользоваться функцией **strstr ()**.

Функция **strstr ()** описана следующим образом:

char* strstr(const char* str, const char* substr)

47. Функции преобразования типа Функции printf() и scanf(). Использование функции printf(). Модификаторы спецификации преобразования, используемые в функции printf(). Использование функции printf() для преобразования данных. Применение функции scanf().

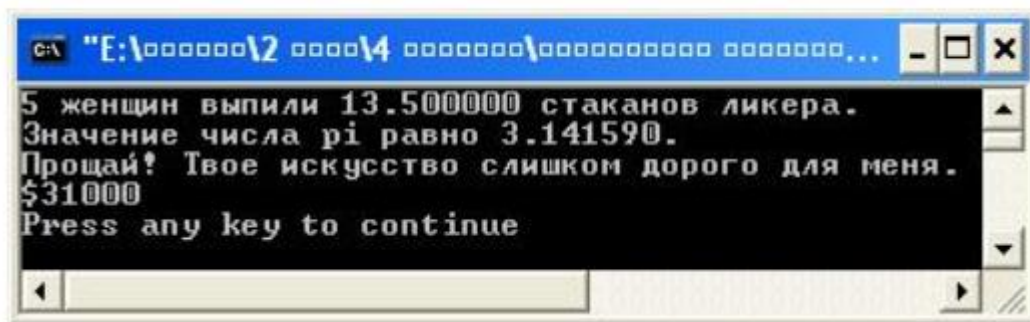
Функции printf() и scanf().

Функции **printf()** и **scanf()** дают нам возможность взаимодействовать с программой. Мы называем их функциями ввода-вывода. Это не единственные функции, которыми мы можем воспользоваться для ввода и вывода данных с помощью программ на языке Си, но они наиболее универсальны. Указанные функции не входят в описание языка Си. И действительно, при работе с языком Си реализация функций ввода-вывода возлагается на создателей компилятора; это дает возможность более эффективно организовать ввод-вывод на конкретных машинах.

Использование функции printf().

Приведем программу, иллюстрирующую обсуждаемые вопросы.

```
#define PI 3.14159
main()
{
    int number = 5;
    float ouzo = 13.5;
    int cost = 31000;
    printf("%d женщин выпили %f стаканов ликера.\n", number, ouzo);
    printf("Значение числа pi равно %f.\n", PI);
    printf("Прощай! Твое искусство слишком дорого для меня. \n");
}
```



Модификаторы спецификации преобразования, используемые в функции printf().

Мы можем несколько расширить основное определение спецификации преобразования, поместив **модификаторы** между знаком **%** и **символом, определяющим тип преобразования**. В приводимой ниже таблице дан список тех символов, которые вы имеете право туда поместить.

Модификатор	Значение
—	Аргумент будет печататься с левой позиции поля заданной ширины (как объяснено ниже). Обычно печать аргумента оканчивается в самой правой позиции поля Пример: %—10d
строка цифр	Задаёт минимальную ширину поля. Большее поле будет использоваться, если печатаемое число или строка не помещаются в исходном поле Пример: %4d
строка цифр	Определяет точность: для типов данных с плавающей точкой — число печатаемых цифр справа от десятичной точки; для символьных строк — максимальное число печатаемых символов Пример: %4.2f (две десятичные цифры для поля шириной в четыре символа)
l	Соответствующий элемент данных имеет тип long, а не int. Пример: %ld

```
int main()
{
    printf("/%ld\n", 336);

    printf("/%5.3d\n", 336);

    printf("/% 10d\n", 336);

    printf("/%-10d\n", 336);
}
```

```
C:\WINDOWS\system32\cmd.exe
/336/
/ 336/
/ 336/
/336 /
Для продолжения нажмите любую клавишу . . .
```

Использование функции printf() для преобразования данных.

```
main()
{
    printf(" %d\n", 336);
    printf(" %o\n", 336);
    printf(" %x\n", 336);
    printf(" %d\n", -336);
}
```

```
C:\WINDOWS\system32\cmd.exe
336
520
150
-336
Для продолжения нажмите любую клавишу . . .
```

Как вы, по-видимому, и ожидали, при использовании спецификации %d будет получено число 336 точно так же, как в примере, обсуждавшемся чуть выше. Но давайте посмотрим, что произойдет, когда вы «попросите» программу напечатать это десятичное целое число в восьмеричном коде. Она напечатает число 520, являющееся восьмеричным эквивалентом 336 ($5 \cdot 64 + 2 \cdot 8 + 0 \cdot 1 = 336$). Аналогично при печати этого числа в шестнадцатеричном коде мы получим 150.

Таким образом, мы можем использовать спецификации, применяемые для функции **printf()** с целью преобразования десятичных чисел в восьмеричные или шестнадцатеричные и наоборот. Или же

если вы захотите напечатать данные в желаемом для вас виде, то необходимо указать спецификацию **%d для получения десятичных чисел, %o — для восьмеричных, а %x— для шестнадцатеричных.** При этом не имеет ни малейшего значения, в какой форме число первоначально появилось в программе.

Применение функции scanf().

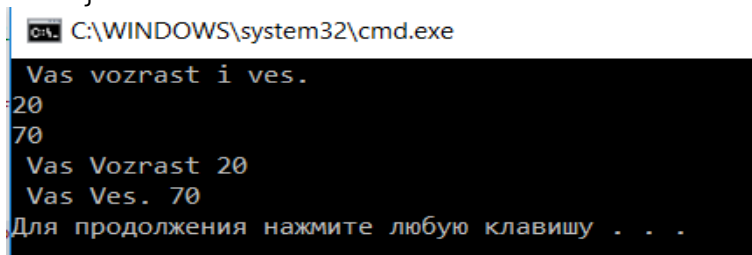
Так же как для функции **printf()**, для функции **scanf()** указываются управляющая строка и следующий за ней список аргументов. Основное различие двух этих функций заключается в особенностях данного списка. Функция **printf()** использует имена переменных, константы и выражения, в то время как функция **scanf()** — только указатели на переменные. К счастью, при применении этой функции мы ничего не должны знать о таких указателях. Необходимо помнить только два правила:

1. Если вам нужно ввести некоторое значение и присвоить его переменной одного из основных типов, то перед именем переменной требуется писать символ **&**.
2. Если вы хотите ввести значение строковой переменной, использовать символ **&** не нужно.

Приведем правильную программу:

```
int main()
{
    int age, ves;

    printf(" Vas vozrast i ves.\n");
    scanf_s(" %d ", &age);
    scanf_s("%d", &ves);
    printf(" Vas Vozrast %d\n", age);
    printf(" Vas Ves. %d\n", ves);
}
```



48.Основные операции. Операция присваивания: =. Операция сложения: +. Операция вычитания: -. Операция изменения знака: -. Операция умножения: *. Операция деления: /.

Рассмотрим способы обработки данных - для этого язык Си имеет широкий набор возможностей. Основные арифметические операции: сложения, вычитания, умножения, деления. Операции в языке Си применяются для представления арифметических действий. Например, выполнение операции **+** приводит к сложению двух величин, стоящих слева и справа от этого знака. Рассмотрим операции **=, +, -, *, /**

Операция присваивания "="

В языке Си знак равенства не означает "равно". Он означает *операцию присваивания* некоторого значения. С помощью оператора

year=2004;

переменной с именем **year** присваивается значение **2004**, т.е. элемент слева от знака **=** - это имя переменной, а элемент справа - ее значение. Мы называем символ **=** **операцией присваивания**

Операция сложения: +

Выполнение операции + приводит к сложению двух величин, стоящих слева и справа от этого знака. Например, в результате работы оператора

printf("%d", 100 + 65);

на печать будет выведено число 165, а не выражение 100+65. *Операнды* могут быть как переменными, так и константами. Операция + называется "бинарной", или "диадической". Эти названия отражают тот факт, что она имеет дело с двумя *операндами*.

Операция вычитания: -

Выполнение операции вычитания приводит к вычитанию числа, расположенного справа от знака -, из числа, стоящего слева от этого знака. Оператор

n = 163.00 - 100.00;

присваивает переменной n значение 63.00.

Операция изменения знака: -

Знак минус используется для указания или изменения алгебраического знака некоторой величины. Например, в результате выполнения последовательности операторов

teg = -15;

get = -teg;

переменной get будет присвоено значение 15. Когда знак используется подобным образом, данная операция называется "унарной". Такое название указывает на то, что она имеет дело только с одним *операндом*.

Операция умножения: *

Операция умножения обозначается знаком *. При выполнении оператора

z = 3 * x

значение переменной x умножается на 3, и результат присваивается переменной z.

Операция деления: /

В языке Си символ / указывает на операцию деления. Величина, стоящая слева от этого знака, делится на величину, расположенную справа от этого знака. Например, в результате выполнения оператора

l = 126.0 / 2.0;

переменной l будет присвоено значение 63.0. Над данными целого типа операция деления производится не так, как над данными с плавающей точкой: в первом случае результат будет целым числом, а во втором - числом с плавающей точкой. В языке Си принято правило, согласно которому дробная часть у результата деления целых чисел отбрасывается. Это действие называется "усечением".

Рассмотрим пример, как выполняется усечение и чем деление целых чисел отличается от деления чисел с плавающей точкой:

```
int main()
{
    setlocale(LC_CTYPE, "Russian");
    printf("деление целых: 5/4 это %d \n", 5 / 4);

    printf("деление целых: 6/3 это %d \n", 6 / 3);

    printf("деление целых: 7/4 это %d \n", 7 / 4);

    printf("деление чисел с плавающей точкой: 7/4 это %2.2f \n", 7. / 4.);

    printf("смешанное деление: 7./4 это %2.2f \n", 7. / 4);

}
```

```
C:\WINDOWS\system32\cmd.exe
деление целых: 5/4 это 1
деление целых: 6/3 это 2
деление целых: 7/4 это 1
деление чисел с плавающей точкой: 7/4 это 1.75
смешанное деление: 7./4 это 1.75
Для продолжения нажмите любую клавишу . . .
```

49. Основные операции. Поразрядные операции. Поразрядное И (AND). Поразрядное ИЛИ (OR). Поразрядное исключающее ИЛИ (XOR).

В C++ существуют три поразрядные логические операции:

1. поразрядное **И**, обозначение: **&**
2. поразрядное исключающее **ИЛИ**, обозначение: **^**
3. поразрядное включающее **ИЛИ**, обозначение: **|**

Так же в C++ существуют логические операции: **или** — **||**; **и** — **&&**. У многих возникает вопрос, “Чем отличаются операции: **&** и **&&**; **|** и **||**?”. Ответ на этот вопрос можно получить, если понять принцип работы поразрядных логических операций. Сразу могу сказать одно, что логические операции **&&** и **||** используются только для построения логических условий. Тогда как поразрядные логические операции применяются в бинарной арифметике. Данные операции работают с битами ячеек памяти, причём операнды и результат могут быть заданы в другой форме, например, в десятичной. Далее рассмотрим каждую из операций подробно.

Поразрядная логическая операция **И**.

Обозначения: X, Y – операнды; F – результат выполнения логической операции

Таблица 1 — Таблица истинности поразрядного И

X	Y	F
0	0	0
0	1	0
1	0	0
1	1	1

Из таблицы истинности видно, что результат будет нулевым, если хотя бы один из битов 0. Если оба бита равны 1, то результат равен 1.

Пример с числами:

Для упрощения вычислений возьмем четырёхразрядные(4-х разрядный двоичный код) положительные операнды. Сначала переводим числа в двоичный код, а потом выполняем операцию.

1111 = 15	1000 = 8
<u>& 1001 = 9</u>	<u>& 1010 = 10</u>
1001 = 9	1000 = 8
Результат равен 9.	Результат равен 8.

Поразрядное исключающее ИЛИ.

Таблица 2 — Таблица истинности поразрядного исключающего ИЛИ

X	Y	F
0	0	0
0	1	1
1	0	1
1	1	0

Из таблицы истинности видно, что результат будет нулевым, если оба бита будут равны, во всех остальных случаях результат равен 1.

Пример с числами:

1111 = 15	1000 = 8
<u>^ 1001 = 9</u>	<u>^ 1010 = 10</u>
0110 = 6	0010 = 2
Результат равен 6.	Результат равен 2.

Поразрядное включающее ИЛИ.

Таблица 3 — Таблица истинности поразрядного включающего ИЛИ

X	Y	F
0	0	0
0	1	1
1	0	1
1	1	1

Из таблицы истинности видно, что результат будет нулевым, если оба бита будут равны 0, во всех остальных случаях результат равен 1.

Пример с числами:

1111 = 15	1000 = 8
1001 = 9	1010 = 10
1111 = 15	1010 = 10
Результат равен 15.	Результат равен 10.

50. Основные операции. Поразрядный сдвиг влево и вправо. Операции отношения и логические операции. Условная операция (? :). Операция запятая (,). Порядок выполнения операций.

Операции сдвига влево и вправо

Сдвиг влево (<<)

Задача данного оператора сдвинуть биты своего операнда на какое-то количество влево, начиная с младшего правого бита. Пустые места после сдвига заполняются нулями. Если размера числа будет недостаточно чтобы сохранить сдвинутые биты в левую сторону, они будут утеряны.

Мы собираемся взять число 0101 и сдвинуть его влево на 2 разряда.

Шаг 1:

Оператор перемещает все значения на один разряд влево и получается следующие (0)101X

В скобках число которое вышло за пределы ячейки отведенной для нашего 4 битного значения и было утеряно, а X вакантное место и это вакантное место оператор заполнит нулем и получится: (0)101[0] И так мы видим, в круглых скобках что вышло и утеряно а в квадратных то что оператор вставил.

Шаг2:

На момент шага два мы имеем число 1010.

Оператор производит второй сдвиг и мы получаем вот такую ситуацию (01)010X

Как мы видим единица теперь оказалась в круглых скобках и ушла в никуда и опять образовалась вакансия в правой стороне, оператор в очередной раз радостно заполнит ее нулем и получим в итоге: (01)01[00] В круглых скобках та часть исходного числа что была утеряна так как вышла за пределы нашей 4-х битной ячейки а в квадратных скобках то что добавил оператор.

Операция : 0101 << 2	Разряд 4	Разряд 3	Разряд 2	Разряд 1
Операнд	0	1	0	1
Результат	0	1	0	0

Как видно из таблицы, все биты сместились в лево на 2 позиции, и как вы можете заметить мы потеряли одну единицу, так как в нашем примере наши числа имеют длину лишь 4 бита. И так в результате применения операции 0101 << 2 будет 0100.

Сдвиг вправо (>>)

Задача данного оператора сдвинуть биты своего операнда на какое-то количество вправо, начиная со старшего левого бита. Сдвиг вправо числа типа unsigned всегда заполняет освободившиеся биты нулями. Сдвиг вправо числа со знаком в некоторых системах приводит к заполнению этих битов значениями знакового бита, а в других нулями. Все младшие биты будут утеряны.

Операция : 0101 >> 2	Разряд 4	Разряд 3	Разряд 2	Разряд 1
Операнд	0	1	0	1
Результат	0	0	0	1

Как видно из таблицы, все биты сместились в право на 2 позиции, единица которая изначально была в 1 разряде вышла за границы и была утеряна.

И так в результате применения операции 0101 >> 2 будет 0001.

Операции отношений и логические операции.

Логические операции

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

! - отрицание (логическое НЕТ);

&& - конъюнкция (логическое И);

|| - дизъюнкция (логическое ИЛИ).

Операции сравнения

== - равно или эквивалентно;

!= - не равно;

< - меньше;

<= - меньше либо равно;

> - больше;

>= - больше либо равно.

Операция «,» (запятая)

Данная операция используется для организации заданной последовательности вычисления выражений (обычно используется там, где по синтаксису допустима только одна операция, а необходимо разместить две и более, например, в операторе *for*). Форма записи:

выражение_1, ..., выражение_N;

выражения 1, 2, ..., N вычисляются последовательно друг за другом и результатом операции становится значение последнего выражения *N*, например:

m = (i = 1, j = i ++, k = 6, n = i + j + k);

получим последовательность вычислений: *i = 1, j = i = 1, i = 2, k = 6, n = 2 + 1 + 6*, и в результате ***m = n = 9***.

Условная операция «? :»

Если одно и то же выражение (или переменная) вычисляется по-разному в зависимости от некоторого условия, вместо оператора *if* можно использовать более короткую запись - *условную операцию*. Эта операция – ***тернарная***, т.е. в ней участвуют три операнда. Формат написания условной операции следующий:

Выражение 1 ? выражение 2 : выражение 3;

если *выражение 1* (условие) отлично от нуля (истинно), то результатом операции является значение *выражения 2*, в противном случае – значение *выражения 3*. Каждый раз вычисляется только одно из выражений 2 или 3.

Таблица 1 — Приоритет операций в C++

Приоритет	Операция	Ассоциативность	Описание
1	::	слева направо	унарная операция разрешения области действия
	[]		операция индексирования
	()		круглые скобки
	.		обращение к члену структуры или класса
	->		обращение к члену структуры или класса через указатель
2	++	слева направо	постфиксный инкремент
	—		постфиксный декремент
3	++	справа налево	префиксный инкремент
	—		префиксный декремент
4	*	слева направо	умножение
	/		деление
	%		остаток от деления
5	+	слева направо	сложение
	—		вычитание
6	>>	слева направо	сдвиг вправо
	<<		сдвиг влево
7	<	слева направо	меньше
	<=		меньше либо равно
	>		больше
	>=		больше либо равно
8	==	слева направо	равно
	!=		не равно
9	&&	слева направо	логическое и
10		слева направо	логическое или
11	?:	справа налево	условная операция (тернарная операция)
12	=	справа налево	присваивание
	*=		умножение с присваиванием
	/=		деление с присваиванием
	%=		остаток от деления с присваиванием
	+=		сложение с присваиванием
	-=		вычитание с присваиванием
13	,	слева направо	запятая

51.Дополнительные операции. Операция деления по модулю: %. Операции увеличения и уменьшения: ++. Операция уменьшения: --. Старшинство операций.

В языке Си имеется около сорока операций. Те операции, которые мы рассмотрели, являются наиболее общеупотребительными. Рассмотрим еще три операции, наиболее используемые программистами.

Операция деления по модулю: %

Эта операция используется в целочисленной арифметике. Ее результатом является остаток от деления целого числа, стоящего слева от знака операции, на число, расположенное справа от нее. Например, **63%5**, читается как 63 по модулю 5, имеет значение 3, т.к. $63=12*5+3$.

Операция увеличения: ++

Операция увеличения осуществляет следующее простое действие: она увеличивает значение своего *операнда* на единицу. Существуют две возможности использования данной операции: первая, когда символы ++ находятся слева от переменной (*операнда*), - "**префиксная**" форма, и вторая, когда символы ++ стоят справа от переменной, - "**постфиксная**" форма. Эти две формы указанной операции различаются между собой только тем, в какой момент осуществляется увеличение *операнда*. Префиксная форма изменяет значение *операнда* перед тем, как *операнд* используется. Постфиксная форма изменяет значение после того как *операнд* использовался. **В примере**,

j=i++;

переменной j сначала присваивается значение i, затем значение переменной i увеличивается на 1.

Операция уменьшения: --

Каждой операции увеличения соответствует некоторая операция уменьшения, при этом вместо символов ++ мы используем --. Когда символы -- находятся слева от *операнда* - "**префиксная**" форма операции уменьшения. Если символы -- стоят справа от *операнда* - это "**постфиксная**" форма операции уменьшения.

! Не применяйте операции увеличения или уменьшения к переменной, присутствующей в более чем одном аргументе функции. Не применяйте операции увеличения или уменьшения к переменной, которая входит в выражение более одного раза.

Старшинство операций

Операция		Описание
1	++	инкремент (увеличение на единицу)
2	--	декремент (уменьшение на единицу)
3	*	умножение
4	/	деление
5	%	остаток от деления
6	+	сложение
7	-	вычитание
8	<<	сдвиг влево
9	>>	сдвиг вправо
10	~	инверсия
11	^	исключающее ИЛИ
12	&	логическое И
13		логическое ИЛИ
14	!	логическое НЕ
15	?:	операция условного выполнения
16	=	присваивание
17	<=	меньше или равно
18	>=	больше или равно
19	<	меньше
20	>	больше
21	==	равно
22	!=	не равно
23	<=	меньше или равно
24	>=	больше или равно
25	<	меньше
26	>	больше
27	==	равно
28	!=	не равно
29	<=	меньше или равно
30	>=	больше или равно
31	<	меньше
32	>	больше
33	==	равно
34	!=	не равно
35	<=	меньше или равно
36	>=	больше или равно
37	<	меньше
38	>	больше
39	==	равно
40	!=	не равно
41	<=	меньше или равно
42	>=	больше или равно
43	<	меньше
44	>	больше
45	==	равно
46	!=	не равно
47	<=	меньше или равно
48	>=	больше или равно
49	<	меньше
50	>	больше
51	==	равно
52	!=	не равно
53	<=	меньше или равно
54	>=	больше или равно
55	<	меньше
56	>	больше
57	==	равно
58	!=	не равно
59	<=	меньше или равно
60	>=	больше или равно
61	<	меньше
62	>	больше
63	==	равно
64	!=	не равно
65	<=	меньше или равно
66	>=	больше или равно
67	<	меньше
68	>	больше
69	==	равно
70	!=	не равно
71	<=	меньше или равно
72	>=	больше или равно
73	<	меньше
74	>	больше
75	==	равно
76	!=	не равно
77	<=	меньше или равно
78	>=	больше или равно
79	<	меньше
80	>	больше
81	==	равно
82	!=	не равно
83	<=	меньше или равно
84	>=	больше или равно
85	<	меньше
86	>	больше
87	==	равно
88	!=	не равно
89	<=	меньше или равно
90	>=	больше или равно
91	<	меньше
92	>	больше
93	==	равно
94	!=	не равно
95	<=	меньше или равно
96	>=	больше или равно
97	<	меньше
98	>	больше
99	==	равно
100	!=	не равно

52. Выражения и операторы. Выражения. Операторы. Составные операторы (блоки).

Выражения

Выражения представляют собой множество данных связанных между собой операциями - особыми операторами языка, возвращающих некоторое значение. Аргументы операций называются операндами. Большинство операций либо унарные (с одним операндом) или бинарные (с двумя операндами). Также операции характеризуются приоритетом (старшинством) выполнения в выражении. Например, результат выражения $4+5*2$ будет 14, а не 18, так как операция умножения имеет больший приоритет, чем сложение. Операции одинакового приоритета выполняются последовательно слева направо.

Операторы

Операторы программы на языке C управляют процессом ее выполнения. В языке C, как и в других языках программирования, имеется ряд операторов, с помощью которых можно выполнять циклы, указывать другие операторы для выполнения и передавать управление на другой участок программы

оператор <code>break</code>	оператор <code>goto</code> и операторы с метками
составной оператор	оператор <code>if</code>
оператор <code>continue</code>	пустой оператор
оператор <code>do</code>	оператор <code>return</code>
оператор <code>expression</code>	оператор <code>switch</code>
оператор <code>for</code>	оператор <code>while</code>

Операторы языка C состоят из ключевых слов, выражений и других операторов. В операторах языка C появляются следующие ключевые слова:

<code>break</code>	<code>default</code>	<code>for</code>	<code>return</code>
<code>case</code>	<code>do</code>	<code>goto</code>	<code>switch</code>
<code>continue</code>	<code>else</code>	<code>if</code>	<code>while</code>

Составной оператор

К составным операторам относят собственно составные операторы и блоки. В обоих случаях это последовательность операторов, заключенная в фигурные скобки. Блок отличается от составного оператора наличием определений в теле блока. Например:

1 {	
2 n++;	
3 summa+=n;	
4 }	
5 {	
6 int n=0;	
7 n++;	
8 summa+=n;	
9 }	

это составной оператор

это блок

53. Ввод и вывод одного символа: функции `getchar()` и `putchar()`. Буферы. Чтение данных. Чтение строки. Чтение файла.

Ввод и вывод одного символа: функции `getchar()` и `putchar()`.

Функция `getchar()` получает один символ, поступающий с пульта терминала (и поэтому имеющий название), и передает его выполняющейся в данный момент программе. Функция `putchar()` получает один символ, поступающий из программы, и пересылает его для вывода на экран. Ниже приводится пример очень простой программы. Единственное, что она делает, это принимает один символ с клавиатуры и выводит его на экран.

```
#include <stdio.h>

void main()
{
    char ch;
    ch = getchar();
    putchar (ch);
}
```



Буферы.

При выполнении данной программы вводимый символ в одних вычислительных системах немедленно появляется на экране («эхо-печать»), в других же ничего не происходит до тех пор, пока вы не нажмете клавишу [ввод]. Первый случай относится к так называемому «небуферизованному» (прямому) вводу, означающему, что вводимый символ оказывается немедленно доступным ожидающей программе. Например, работа программы в системе, использующей буферизованный ввод, будет выглядеть следующим образом:

Вот длинная входная строка. [ввод]

В

В системе с небуферизованным вводом отображение на экране символа В произойдет сразу, как только вы нажмете соответствующую клавишу. Результат ввода-вывода при этом может выглядеть, например, так:

В Вот длинная входная строка.

Символ В, появившийся на второй позиции данной строки, — это непосредственный результат. Зачем нужны буферы? Во-первых, оказывается, что передачу нескольких символов в виде одного блока можно осуществить гораздо быстрее, чем передавать их последовательно по одному. Во-вторых, если при вводе символов допущена ошибка, вы можете воспользоваться корректирующими средствами терминала, чтобы ее исправить. И когда в конце концов вы нажмете клавишу [ввод], будет произведена передача откорректированной строки.

Чтение данных.

Теперь возьмемся за что-нибудь несколько более сложное, чем чтение и вывод на печать одного символа — например, за вывод на печать групп символов. Желательно также, чтобы в любой момент можно было остановить работу программы; для этого спроектируем ее так, чтобы она прекращала работу при получении какого-нибудь специального символа, скажем *. Поставленную задачу можно решить, используя цикл **while**:

```
#include <stdio.h>

#define STOP '*'

void main(void)
{
    char ch;

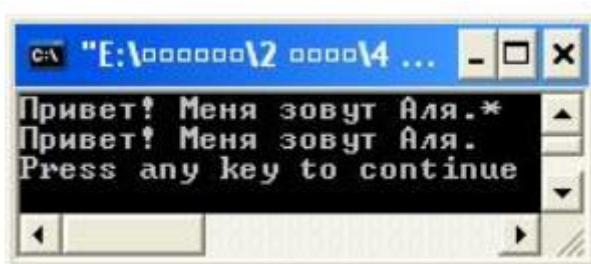
    ch = getchar(); // строка 9

    while (ch != STOP) { // строка 10

        putchar(ch); // строка 11

        ch = getchar(); //строка 12

    }
}
```



При первом прохождении тела цикла функция `putchar()` получает значение своего аргумента в результате выполнения оператора, расположенного в строке 9; в дальнейшем, вплоть до завершения работы цикла, значением этого аргумента является символ, передаваемый программе функцией `getchar()`, расположенной в строке 12. Мы ввели новую операцию отношения `!=`, смысл которой выражается словами «не равно». В результате всего этого цикл `while` будет осуществлять чтение и печать символов до тех пор, пока не поступит признак `STOP`. Мы могли бы опустить в программе директиву `#define` и использовать лишь символ `*` в операторе `while`, но наш способ делает смысл данного знака более очевидным.

Чтение строки.

Одной из возможностей является использование символа «новая строка» (`\n`). Для этого нужно лишь переопределить признак `STOP`:

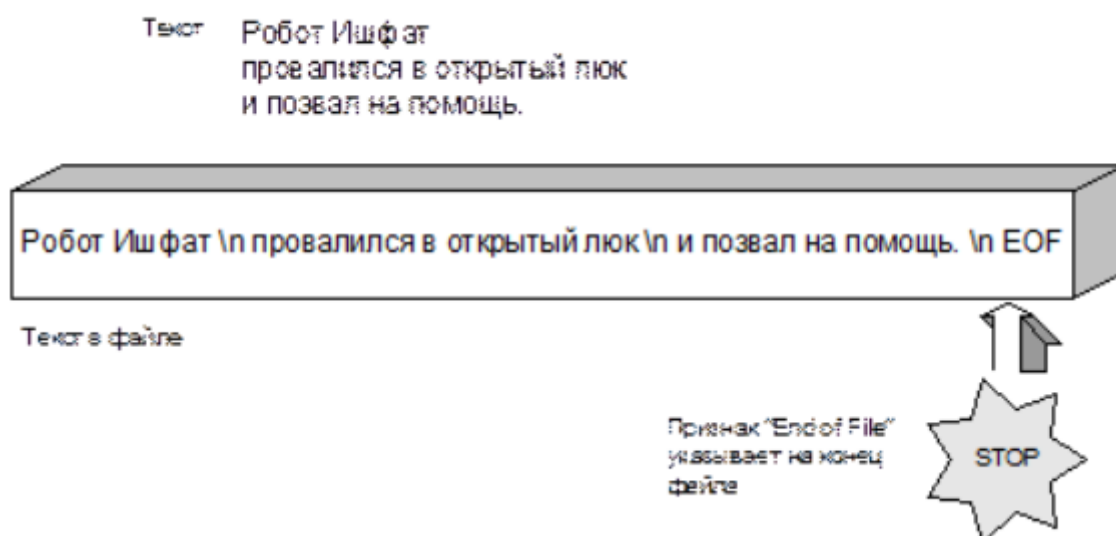
#define STOP '\n'

Какой это даст эффект? Очень большой: ведь символ «новая строка» пересылается при нажатии клавиши [ввод].

Чтение файла.

Каким может быть идеальный признак STOP? Это должен быть такой символ, который обычно не используется в тексте и, следовательно, не приводит к ситуации, когда он случайно встретится при вводе, и работа программы будет остановлена раньше, чем мы хотели бы.

Проблема подобного сорта не нова, и, к счастью для нас, она уже была успешно решена проектировщиками вычислительных систем. На самом деле задача, которую они рассматривали, была несколько отличной от нашей, но мы вполне можем воспользоваться их решением. Занимавшая их проблема касалась «файлов». Файлом можно назвать участок памяти, в который помещена некоторая информация. Обычно файл хранится в некоторой долговременной памяти, например, на гибких или жестких дисках, или на магнитной ленте. Чтобы отмечать, где кончается один файл и начинается другой, полезно иметь специальный символ, указывающий на конец файла. Это должен быть символ, который не может появиться где-нибудь в середине файла, точно так же как выше нам требовался символ, обычно не встречающийся во вводимом тексте. Решением указанной проблемы служит введение специального признака, называемого «End-of-File» (конец файла), или EOF, для краткости. Выбор конкретного признака EOF зависит от типа системы: он может состоять даже из нескольких символов. Но такой признак всегда существует, и компилятор с языка Си, которым вы пользуетесь, конечно же знает, как такой признак действует в вашей системе.



Каким образом можно воспользоваться символом EOF? Обычно его определение содержится в файле `<stdio.h>`. Общеупотребительным является определение:

#define EOF (- 1)

Это дает возможность использовать в программах выражения, подобные, например, такому:

while ((ch=getchar()) != EOF

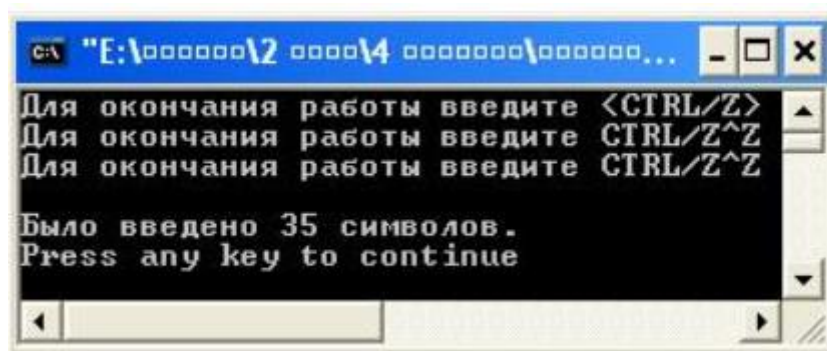
`#include <stdio.h>`

`void main(void)`

```

{
int ch;
int count=0;
printf("Для окончания работы введите <CTRL/Z>\n");
while ((ch=getchar()) != EOF)
{
putchar(ch);
count++;
}
printf("\n Было введено %d символов.\n",count);
}

```



Отметим следующие моменты:

1. Нам не нужно самим определять признак EOF, поскольку заботу об этом берет на себя файл `<stdio.h>`.
2. Мы можем не интересоваться фактическим значением символа EOF, поскольку директива `#define`, имеющаяся в файле `<stdio.h>`, позволяет нам использовать его символическое представление.
3. Мы изменили тип переменной `ch` с `char` на `int`. Мы поступили так потому, что значениями переменных типа `char` являются целые числа без знака в диапазоне от 0 до 255, а признак EOF может иметь числовое значение -1. Эта величина недопустима для переменной типа `char`, но вполне подходит для переменной типа `int`. К счастью, функция `getchar()` фактически возвращает значение типа `int`, поэтому она в состоянии прочесть символ EOF.
4. Переменная `ch` целого типа никак не может повлиять на работу функции `putchar()`. Она просто выводит на печать символьный эквивалент значения аргумента.
5. При работе с данной программой, когда символы вводятся с клавиатуры, необходимо уметь вводить признак EOF. Не думайте, что вы можете просто указать буквы E-O-F или число -1. (Число -1 служит эквивалентом кода ASCII данного символа, а не самим этим символом.) Вместо этого вам необходимо узнать, какое представление используется в вашей системе. В большинстве реализаций операционной системы UNIX, например, ввод знака [CTRL/D] (нажать на клавишу [D], держа нажатой клавишу [CTRL]) интерпретируется как признак EOF. Во многих микрокомпьютерах для той же цели используется знак [CTRL/Z].

54.Переключение и работа с файлами. Операционная система UNIX.

Переключение вывода. Переключение ввода. Комбинированное переключение.

Переключение и работа с файлами.

Операция переключения — это средство ОС UNIX, а не самого языка Си. Но она оказалась настолько полезной, что при переносе компилятора с языка Си на другие вычислительные системы часто вместе с ним переносится в какой-то форме и эта операция. Более того, многие из вновь созданных операционных систем, таких, как MS-DOS 2, включают в себя данное средство. Поэтому, даже если вы не работаете в среде ОС UNIX существует большая вероятность того, что вы в той или иной форме сможете воспользоваться операцией переключения. Мы обсудим сначала возможности этой операции в ОС UNIX, а затем и в других системах.

Операционная система UNIX.

Переключение вывода.

Символ **<** служит обозначением операции переключения, используемой в ОС UNIX. Выполнение указанной операции приводит к тому, что содержимое файла **words** будет направлено в файл с именем **getput4**. Сама программа ввод-вывод4 не знает (и не должна знать), что входные данные поступают из некоторого файла, а не с терминала; на ее вход просто поступает поток символов, она читает их и последовательно, по одному выводит на печать до тех пор, пока не встретит признак EOF. В операционной системе UNIX файлы и устройства ввода-вывода в логическом смысле представляют собой одно и то же, поэтому теперь файл для данной программы является устройством ввода-вывода. Если вы попытаетесь ввести команду

getput4 < words

Переключение ввода.

Теперь предположим, вам хочется, чтобы слова, вводимые с клавиатуры, попадали в файл с именем **mywords**. Для этого вы должны ввести команду

getput4 > mywords

и начать ввод символов. Символ **>** служит обозначением еще одной операции переключения, используемой в ОС UNIX. Ее выполнение приводит к тому, что создается новый файл с именем **mywords**, а затем результат работы программы ввод-вывод4, представляющий собой копию вводимых символов, направляется в данный файл. Если файл с именем **mywords** уже существует, он обычно уничтожается, и вместо него создается новый.

Комбинированное переключение.

Предположим теперь, что вы хотите создать копию файла **mywords** и назвать ее **savewords**. Введите для этого команду

getput4 < mywords >savewords

и требуемое задание будет выполнено. Команда

getput4 >savewords < mywords

приведет к такому же результату, поскольку порядок указания операций переключения не имеет значения.

Нельзя использовать в одной команде один и тот же файл и для ввода и для вывода одновременно.

getput4 <mywords >mywords (НЕПРАВИЛЬНО)

Причина этого заключается в том, что указание операции **>mywords** приводит к стиранию исходного файла перед его использованием в качестве входного.

55. Выбор вариантов. Операции отношения и выражения. Понятие "истина".

Выбор вариантов.

Хотите научиться создавать мощные, «интеллектуальные», универсальные и полезные программы? Тогда вам потребуется язык, обеспечивающий три основные формы управления процессом выполнения программ. Согласно теории вычислительных систем, хороший язык должен обеспечивать реализацию следующих трех форм управления процессом выполнения программ:

- Выполнение последовательности операторов.
- Выполнение определенной последовательности операторов до тех пор, пока некоторое условие истинно.
- Использование проверки истинности условия для выбора между различными возможными способами действия.

Операции отношения и выражения.

Операции отношения используются для сравнений. Мы уже использовали ранее некоторые из них, а сейчас приведем полный список операций отношения, применяемых при программировании на языке Си.

Операции сравнения(отношения)

== - равно или эквивалентно;

!= - не равно;

< - меньше;

<= - меньше либо равно;

> - больше;

>= - больше либо равно.

При программировании требуется аккуратность, потому что в ряде случаев компилятор не сможет обнаружить ошибки, связанной с неправильным использованием знаков этих операций, что приведет к результатам, отличным от тех, которые вы должны были получить.

Язык	Операция присваивания	Операция отношения «равенство»
Бейсик	=	=
Фортран	=	.EQ.
Си	=	= =
Паскаль	:=	=
ПЛ/1	=	=
Лого	make	=

Операции отношения применяются при формировании условных выражений, используемых в операторах if и while. Указанные операторы проверяют, истинно или ложно данное выражение. Ниже приводятся четыре не связанные между собой оператора

f (number < 6)

printf(" Ваше число слишком мало.\n");

while (ch != '\$')

count++;

if (total == 100)

printf(" Вы набрали максимум очков! \n");

if (ch > 'M')

printf(" Отправьте этого человека по другому маршруту.\n");

Сравнение

```
canoes == 5
```

= = проверяет, равно ли значение canoes пяти

Присваивание

```
canoes = 3
```

= дает canoes значение 3

Понятие «истина».

Мы можем ответить на этот вечный вопрос, по крайней мере, так, как он решен в языке Си. Напомним, во-первых, что выражение в Си всегда имеет значение. Это утверждение остается верным даже для условных выражений, как показывает пример, приведенный ниже. В нем определяются значения двух условных выражений, одно из которых оказывается истинным, а второе — ложным.

```
Int main()
{
int tr, fa;
tr= (10 > 2); /* отношение истинно */
fa=(10 == 2); /* отношение ложно */
printf("true = %d; false = %d\n", tr, fa);
}
```

В данном примере значения двух условных выражений присваиваются двум переменным. Чтобы не запутать читателя, мы присвоили переменной true значение выражения, которое оказывается истинным, а переменной false — значение выражения, которое оказывается ложным. При выполнении программы получим следующий простой результат:

true = 1; false = 0

Вот как! Оказывается, в языке Си значение «истина» — это 1, а «ложь» — 0.

56. Условные операторы. Оператор if. Оператор if-else. Вложенные операторы if-else.

Условные операторы.

В языке C имеется четыре основных условных оператора: **if**, **if-else**, **?** и **switch**. Перед тем, как обсудить их отдельно, необходимо отметить одно общее правило.

Большинство условных операторов можно использовать для выборочного выполнения или одной программной строки, или нескольких логически связанных строк (называемых блоком). Если условный оператор связан только с одной строкой исполняемого кода, эту строку заключать в фигурные скобки ({ }) не нужно. Если же условный оператор относится к блоку выполняемых операторов, то для связи блока с проверяемым условием фигурные скобки необходимы. По этой причине в операторе switch необходимы открывающая и закрывающая фигурные скобки.

Оператор if.

Оператор if используется для условного выполнения фрагмента кода. Простейший вид оператора if следующий:

if (выражение)

действие_если_истина;

Обратите внимание на то, что выражение должно заключаться в круглые скобки. Для выполнения оператора if необходимо вычислить выражение и определить его значение: истина или ложь. Если выражение истинно, выполняется действие_если_истина и программа продолжается, начиная с оператора, следующего за этим действием. Если выражение ложно, действие_если_истина не выполняется, а выполняется оператор, следующий за невыполненным действием. Например: следующий фрагмент программы напечатает сообщение "Have a great day!" только в случае, если переменная ioutside_temp больше или равна 72:

if(ioutside_temp >= 72)

printf("Have a great day!");

Оператор if, связанный с блоком выполняемых операторов, выглядит следующим образом:

if (выражение)

{

действие_если_истина1;

действие_если_истина2;

действие_если_истина3;

действие_если_истина4;

}

Оператор if-else.

Оператор if-else нужен для того, чтобы программа выполнила два разных действия в зависимости от истинности некоторого выражения. В простейшем случае оператор if-else выглядит следующим образом:

```
if (выражение)  
действие_если_истина;  
else  
действие_если_ложь;
```

В этом операторе, если выражение истинно, выполняется действие_если_истина; если же выражение ложно, выполняется действие_если_ложь. Рассмотрим пример:

```
if(ckeypressed == UP)  
iy_pixel_coord++;  
else  
iy_pixel_coord--;
```

Любое из выражений — **выражение_если_истина**, **выражение_если_ложь**, или **оба вместе** — могут быть составным оператором или блоком, заключенным в фигурные скобки. Синтаксис этих трех вариантов очень прост:

```
if (выражение) {  
действие_если_истина1;  
действие_если_истина2;  
действие_если_истина3;  
}  
else  
действие_если_ложь;
```

```
if (выражение) {  
действие_если_истина;  
else {  
действие_если_ложь1;  
действие_если_ложь2;  
действие_если_ложь3;  
}
```

```
if (выражение) {  
действие_если_истина1;  
действие_если_истина2;
```

действие если истина3;

}

else {

действие если ложь1;

действие если ложь2;

действие если ложь3;

}

Вложенные операторы if-else.

Когда используются вложенные операторы if, необходимо следить за тем, какому if соответствует конкретное действие else. Взгляните на следующий пример — сможете ли вы предсказать результат:

```
if(iout_side_temp < 50) /* Если температура на улице ... */  
if(iout_side_temp < 30) printf("Wear the down jacket!"); /*Надеть куртку! */  
else printf("Parka will do."); /* Достаточно одеть парку. */
```

Листинг намеренно не выровнен, чтобы не было зрительных подсказок на вопрос: какой оператор к какому if относится. Что получится, если переменная iout_side_temp равна 55? Будет ли напечатано сообщение "Parka will do."?

Ответ — нет. В этом примере действие else связано со вторым выражением if. Это вызвано тем, что в C else связывается с ближайшим несвязанным if. Для облегчения отладки подобных фрагментов компилятор C написан так, что каждый else связывается с ближайшим if, еще не имеющим части else.

57.Условные операторы. Операторы if-else-if. Условный оператор ?. Оператор switch.

Совместное использование операторов if-else-if и switch.

Операторы if-else-if.

Комбинация операторов **if-else-if** часто используется для выполнения многочисленных последовательных сравнений. В общем виде они выглядят следующим образом:

if(выражение 1)

действие если условие1 истина;

else if(выражение2)

действие если условие2 истина;

else if(выражение3)

действие если условие3 истина;

Каждое действие может быть составным блоком в фигурных скобках (причем после закрывающей фигурной скобки точка с запятой не ставится). Такая управляющая логическая структура вычисляет каждое выражение до тех пор, пока не найдет истинное. Когда это происходит, все оставшиеся проверочные условия опускаются. В предыдущем примере, если ни одно из выражений не дает значения "истина", никаких действий не выполняется.

Когда у вас в программе имеется несколько конструкций **if и else**, каким образом компилятор решает, какому оператору if соответствует какой оператор else? Рассмотрим, например, фрагмент программы

```

if (number > 6)
{
if (number < 12)
printf(" Вы закончили!\n" );
}
else
printf(" Простите, вы потеряли ход!\n" );

```

В каком случае фраза «Простите, вы потеряли ход!» будет напечатана? Когда значение переменной `number` меньше или равно 6, или когда оно больше 12? Другими словами, чему соответствует `else`: первому `if` или второму? `else` соответствует второму `if`

Существует правило, которое гласит, что `else` соответствует ближайшему `if`, кроме тех случаев, когда имеются фигурные скобки. Мы сознательно записали этот фрагмент так, как будто `else` соответствует первому `if`, но вспомните, что компилятор не обращает внимания на отступы в строках.

Условный оператор ?.

Условный оператор `?` позволяет кратко записать условие проверки. Соответствующие действия выполняются в зависимости от вычисленного значения выражение_условие: истина или ложь. Этот оператор можно использовать вместо эквивалентного оператора `if-else`. Условный оператор имеет следующий синтаксис:

Выражение_условие ? действие_если_истина : действие_если_ложь;

Оператор `?` иногда называют троичным оператором, так как он требует трех операндов. Рассмотрим такой оператор:

if(fvalue >= 0.0)

fvalue = fvalue;

else

fvalue = -fvalue;

Этот оператор можно записать при помощи условного оператора `?:`:

```
fvalue = (fvalue >= 0.0) ? fvalue : -fvalue;
```

Оба оператора определяют абсолютное значение `fvalue`. Приоритет условного оператора ниже любого другого, используемого в выражении; поэтому в данном примере круглые скобки не нужны. Тем не менее, скобки часто используются для лучшего восприятия программы.

Оператор switch.

Часто необходимо сравнить некоторую переменную или выражение с несколькими значениями. Для этого можно использовать либо вложенные операторы `if-else-if`, либо — оператор `switch`.

switch (общее_выражение) {

case константа1:

операторы1;

break;

case константа2:

операторы2;

break;

...

...

...

case константа N:

операторыN;

break;

default: операторы;

Простая программа на switch

```
int main()
{
    setlocale(LC_CTYPE, "Russian");
    cout << "Выберите 1 или 2" << endl;
    int t;
    cin >> t;
    switch (t)
    {
        case 1: {
            cout << "Привет" << endl;
            break;
        }
        case 2: {
            cout << "Helloy" << endl;
            break;
        }
    }
}
```

C:\WINDOWS\system32\cmd.exe

Выберите 1 или 2

1

Привет

Для продолжения нажмите любую клавишу . . .

C:\WINDOWS\system32\cmd.exe

Выберите 1 или 2

2

Helloy

Для продолжения нажмите любую клавишу . . .

Совместное использование операторов if-else-if и switch.

В следующем программном примере перечисляемый тип (enum) используется для выполнения необходимых преобразований единиц измерения длины:

Программа на C, иллюстрирующая использование оператора if-else-if

в сочетании с различными операторами switch

```

#include "stdafx.h"

#include <iostream>

#include <conio.h>

#include <stdio.h>

// #include <iostream.h>

using namespace std;

typedef enum conversion_type { YARDS, INCHES, CENTIMETERS, METERS } C_TYPE;

main()

{

    int iuser_response;

    C_TYPE C_Tconversion;

    int ilength=30;

    float fmeasurement; /* Укажите единицы измерения для преобразования : */

    printf("\nPlease enter the measurement to be converted : ");

    scanf("%f",&fmeasurement);

    /* Введите :

    0— ярды

    1— дюймы

    2— сантиметры

    3—метры*/

    printf("\nPlease enter : \

\n\t\t 0 for YARDS \

\n\t\t 1 for INCHES \

\n\t\t 2 for CENTIMETERS \

\n\t\t 3 for METERS \

\n\n\t\tYour response -->> ");

    /* Ваш ответ --> */

```

```
scanf("%d",&iuser_response);

switch(iuser_response) {

case 0 : C_Tconversion = YARDS;

break;

case 1 : C_Tconversion = INCHES;

break;

case 2 : C_Tconversion = CENTIMETERS;

break;

default : C_Tconversion = METERS;

}

if(C_Tconversion == YARDS)

fmeasurement = ilength / 3;

else if(C_Tconversion == INCHES)

fmeasurement = ilength * 12;

else if(C_Tconversion == CENTIMETERS)

fmeasurement = ilength * 12 * 2.54;

else if(C_Tconversion = METERS)

fmeasurement = (ilength * 12 * 2.54)/100;

else

printf("No conversion required"); /* Преобразование не требуется */

switch(C_Tconversion) {

case YARDS : printf("\n\t\t %4.2f yards", fmeasurement);

break;

case INCHES : printf("\n\t\t %4.2f inches", fmeasurement);

break;

case CENTIMETERS : printf("\n\t\t %4.2f centimeters", fmeasurement);

break;
```

```

default : printf("\n\t\t %4.2f meters", fmeasurement);

}

printf ("\n\nPress any key to finish\n");/* Конец работы */

_getch();

return(0);

```

} В этом примере для указанного пересчета единиц длины используется перечисляемый тип данных. В стандартном С перечисляемые типы существуют только внутри самой программы (для обеспечения читабельности программы), и значения входящих в них переменных нельзя задавать или выводить непосредственно. Первый оператор switch в программе предназначен для преобразования введенного кода в соответствующий тип С `Tconversion`. Нужно преобразование выполняется при помощи вложенных операторов if-else-if. В последнем операторе switch печатается преобразованное значение и название единицы измерения. Конечно же, вложенные операторы if-else-if можно было бы записать при помощи оператора switch.

58. Оператор цикла. Цикл while. Завершение цикла while.

Оператор цикла.

В языке С имеется стандартный набор операторов цикла: for, while и do-while (называемый в некоторых других языках высокого уровня циклом repeat-until). Вас может, однако, удивить то, каким способом программа выходит из цикла. В С можно изменить порядок выполнения цикла четырьмя способами. Естественно, все циклы заканчиваются при выполнении заданного проверочного условия. Однако, в С цикл может также закончиться по некоторому заданному условию ошибки при помощи операторов break или exit. Кроме этого, в циклах может быть собственная управляющая логика, изменяемая при помощи оператора break или оператора continue.

Цикл while.

Так же как и цикл for, в С цикл while является циклом с предусловием. Это означает, что в программе проверка условия осуществляется до выполнения оператора или операторов, входящих в тело цикла. Благодаря этому, циклы с предусловием могут либо не выполняться вообще, либо выполняться множество раз. В С синтаксис цикла while следующий:

while(проверка условия)

оператор;

В циклах while с несколькими операторами необходимы фигурные скобки:

while(проверка условия) {

оператор1;

оператор2;

оператор3;

операторn;

```

int main()
{
    int i=0, x=1, z=1,k;
    while (i < 10)
    {
        k = i*(x + z);
    }
}

```

```

        i++;
    }
    cout << k;
}

```

Завершение цикла while.

Мы подошли к самому существенному моменту рассмотрения циклов while. При построении цикла while вы должны включить в него какие-то конструкции, изменяющие величину проверяемого выражения так, чтобы в конце концов оно стало ложным. В противном случае выполнение цикла никогда не завершится. Рассмотрим следующий пример:

```

index = 1;
while( index < 5)
printf("Доброе утро!\n");

```

Данный фрагмент программы печатает это радостное сообщение бесконечное число раз, поскольку в цикле отсутствуют конструкции, изменяющие величину переменной index, которой было присвоено значение 1.

```

index = 1;
while( index < 5)
{
printf("Доброе утро!\n");
index++;
}

```

Если добавить строчку в программу index++ то программа становится рабочей.

59.Оператор цикла. Цикл do-while. Цикл for.

Оператор цикла.

В языке C имеется стандартный набор операторов цикла: for, while и do-while (называемый в некоторых других языках высоким уровнем циклом repeat-until). Вас может, однако, удивить то, каким способом программа выходит из цикла. В C можно изменить порядок выполнения цикла четырьмя способами. Естественно, все циклы заканчиваются при выполнении заданного проверочного условия. Однако, в C цикл может также закончиться по некоторому заданному условию ошибки при помощи операторов break или exit. Кроме этого, в циклах может быть собственная управляющая логика, изменяемая при помощи оператора break или оператора continue.

Цикл do-while.

Цикл do-while отличается от циклов for и while тем, что это — цикл с постусловием. Другими словами: цикл всегда выполняется хотя бы один раз, после чего в конце первого прохода проверяется условие продолжения цикла. В отличие от этого циклы for и while могут не выполняться вообще, или выполняться множество раз в зависимости от значения переменной управления циклом. Поскольку циклы do-while выполняются, по меньшей мере, один раз, их лучше использовать тогда, когда нет сомнений о вхождении в определенный цикл.

Синтаксис цикла do-while следующий:

do

действие;

while(проверка условия);

```

do {
    действие1;
    действие2;
    действие3;
    действиеn;
} while(проверка условия) ;

```

Цикл for.

В цикле for распространен для «математических повторений». Вот пример его записи:

```

for(count = 1; count <= NUMBER; count ++ )
printf(" Будь моим Валентином!\n");

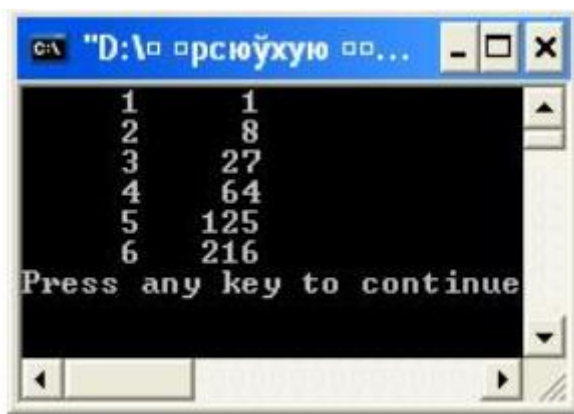
```

В круглых скобках содержатся три выражения, разделенные символом «точка с запятой». Первое из них служит для инициализации счетчика. Она осуществляется только один раз — когда цикл for начинает выполняться. Второе выражение — для проверки условия; она производится перед каждым возможным выполнением тела цикла. Когда выражение становится ложным (или в общем случае равным нулю), цикл завершается. Третье выражение вычисляется в конце каждого выполнения тела цикла.

```

main()
{
    int num;
    for(num = 1; num <= 6; num++)
        printf(" %5d %5d \n" , num, num*num*num);
}

```



60. Оператор цикла. Операция "запятая" в цикле for. Гибкость конструкции for. Философ Зенон и цикл for.

Оператор цикла.

В языке C имеется стандартный набор операторов цикла: for, while и do-while (называемый в некоторых других языках высоким уровнем циклом repeat-until). Вас может, однако, удивить то, каким способом программа выходит из цикла. В C можно изменить порядок выполнения цикла четырьмя способами. Естественно, все циклы заканчиваются при выполнении заданного проверочного условия. Однако, в C цикл

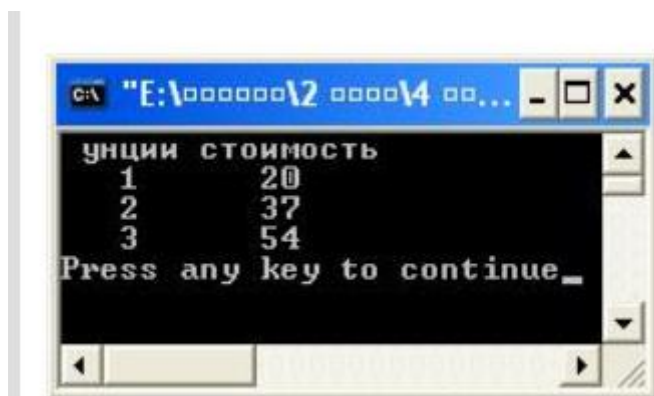
может также закончиться по некоторому заданному условию ошибки при помощи операторов `break` или `exit`. Кроме этого, в циклах может быть собственная управляющая логика, изменяемая при помощи оператора `break` или оператора `continue`.

Операция «запятая» в цикле `for`.

Операция «запятая» увеличивает гибкость использования цикла `for`, позволяя включать в его спецификацию несколько инициализирующих или корректирующих выражений. Например, ниже приводится программа, которая выводит на печать величины почтовых тарифов первого класса обслуживания. (Предположим, что почтовые тарифы: 20 центов за первую унцию и по 17 центов за каждую следующую.)

```
#define FIRST 20
#define NEXT 17

main()
{
    int ounces, cost;
    printf(" Функции стоимость\n");
    for(ounces = 1, cost = FIRST; ounces < 16; ounces++, cost += NEXT)
        printf(" %3d %7d\n", ounces, cost);
}
```



Гибкость конструкции `for`.

Хотя цикл `for` на первый взгляд очень похож на цикл `DO` в Фортране, цикл `FOR` в Паскале и цикл `FOR ... NEXT` в Бейсике, `for` в Си является гораздо более гибким средством, чем любой из упомянутых. Эта гибкость — следствие способа использования упомянутых выше трех выражений в спецификации цикла `for`. До сих пор первое выражение применялось для инициализации счетчика, второе — для задания его граничного значения, а третье — для увеличения его текущего значения на 1. Используемый таким образом, оператор `for` в языке Си совершенно аналогичен упомянутым выше соответствующим операторам в других языках. Но, кроме описанной, существует еще и много других возможностей его применения, девять из которых мы приводим ниже.

1. Можно применять операцию уменьшения для счета в порядке убывания вместо счета в порядке возрастания.

```
for(n = 10; n > 0; n--)
```

2. При желании вы можете вести счет двойками, десятками и т. д.

```
for(n = 2; n < 60; n = n + 13)
```


3. Можно вести подсчет с помощью символов, а не только чисел:

```
for(ch = 'a' ; ch <= 'z'; ch++ )
```

4. Можно проверить выполнение некоторого произвольного условия, отличного от условия, налагаемого на число итераций. В нашей программе таблица кубов вы могли бы заменить спецификацию

```
for(num = 1; num <= 6; num++ )
```

на

```
for(num = 1; num*num*num <= 216; num ++ )
```

Это было бы целесообразно в случае, если бы нас больше занимало ограничение максимального значения диапазона кубов чисел, а не количества итераций.

5. Можно сделать так, чтобы значение некоторой величины возрастало в геометрической, а не в арифметической прогрессии, т. е. вместо прибавления фиксированного значения на каждом шаге цикла, выполнялось бы умножение:

```
for(debt = 100.0; debt < 150.0; debt = debt* 1.1)
```

6. В качестве третьего выражения можно использовать любое правильно составленное выражение. Какое бы выражение вы ни указали, его значение будет меняться при каждой итерации.

```
for(x = 1; y <= 75; y = 5*x+++ 10);
```

7. Можно даже опустить одно или более выражений (но при этом нельзя опустить символы «точка с запятой»). Необходимо только включить в тело цикла несколько операторов, которые в конце концов приведут к завершению его работы.

```
ans = 2;
```

```
for(n = 3; ans <= 25;)
```

```
ans = ans*n;
```

8. Первое выражение не обязательно должно инициализировать переменную. Вместо этого, например, там мог бы стоять оператор printf() некоторого специального вида. Необходимо помнить только, что первое выражение вычисляется только один раз перед тем, как остальные части цикла начнут выполняться.

```
for( printf(" Запоминайте введенные числа!\n"); num == 6; )
```

```
scanf(" %d", &num);
```

```
printf("Это как раз то, что я хочу!\n");
```

Философ Зенон известен своими знаменитыми парадоксами (апориями). Посмотрим, как с помощью операции «запятая» можно разрешить старый парадокс. Греческий философ Зенон утверждал, что пущенная стрела никогда не достигнет цели. Сначала, говорил он, стрела пролетит половину расстояния до цели. После этого ей останется пролететь половину всего расстояния, но сначала она должна будет пролететь половину того, что ей осталось пролететь, и т. д. до бесконечности. Поскольку расстояние полета разбито на бесконечное число частей, для достижения цели стреле может потребоваться бесконечное время. Мы сомневаемся, однако, что Зенон вызвался бы стать мишенью для стрелы, полагаясь только на убедительность своего аргумента.

Применим количественный подход и предположим, что за одну секунду полета стрела пролетает первую половину расстояния. Тогда за последующую $1/2$ секунды она пролетит половину того, что осталось от половины, за $1/4$ — половину того, что осталось после этого, и т. д. Полное время полета представляется в виде суммы бесконечного ряда $1 + 1/2 + 1/4 + 1/8 + 1/16 + \dots$. Мы можем написать короткую программу для нахождения суммы первых нескольких членов.

```
#define LIMIT 15

main ()
{
    int count;
    float sum, x;
    for (sum = 0.0, x = 1.0, count = 1; count <= LIMIT; count++, x *= 2.0)
    {
        sum += 1.0/x;
        printf("sum = %f когда count = %d.\n", sum, count);
    }
}
```

```
sum = 1.000000 когда count = 1.
sum = 1.500000 когда count = 2.
sum = 1.750000 когда count = 3.
sum = 1.875000 когда count = 4.
sum = 1.937500 когда count = 5.
sum = 1.968750 когда count = 6.
sum = 1.984375 когда count = 7.
sum = 1.992188 когда count = 8.
sum = 1.996094 когда count = 9.
sum = 1.998047 когда count = 10.
sum = 1.999023 когда count = 11.
sum = 1.999512 когда count = 12.
sum = 1.999756 когда count = 13.
sum = 1.999878 когда count = 14.
sum = 1.999939 когда count = 15.
Press any key to continue
```

Можно видеть, что, хотя мы и добавляем новые члены, сумма, по-видимому, стремится к какому-то пределу. И действительно, математики показали, что при стремлении числа членов к бесконечности сумма ряда сходится к 2,0, что и демонстрируется нашей программой.

61.Оператор цикла. Вложенные циклы. Алгоритмы и псевдокод.

Оператор цикла.

В языке С имеется стандартный набор операторов цикла: for, while и do-while (называемый в некоторых других языках высокого уровня циклом repeat-until). Вас может, однако, удивить то, каким способом программа выходит из цикла. В С можно изменить порядок выполнения цикла четырьмя способами. Естественно, все циклы заканчиваются при выполнении заданного проверочного условия. Однако, в С цикл может также закончиться по некоторому заданному условию ошибки при помощи операторов break или exit. Кроме этого, в циклах может быть собственная управляющая логика, изменяемая при помощи оператора break или оператора continue.

Вложенные циклы.

Вложенным называется цикл, находящийся внутри другого цикла.

```
int main()
{
    int row, column;
    for (row = 1; row <= 10; row++)
    {
        for (column = 1; column <= 10; column++)
            printf("*");
        puts("\n");
    }
}
```

```

    }
}

```

Под **алгоритмом** понимали конечную последовательность точно сформулированных правил, которые позволяют решать те или иные классы задач. Такое определение алгоритма не является строго математическим, так как в нем не содержится точной характеристики того, что следует понимать под классом задач и под правилами их решения.

Свойства алгоритмов.

Каждое указание алгоритма предписывает исполнителю выполнить одно конкретное законченное действие. Исполнитель не может перейти к выполнению следующей операции, не закончив полностью выполнения предыдущей. Предписания алгоритма надо выполнять последовательно одно за другим, в соответствии с указанным порядком их записи. Выполнение всех предписаний гарантирует правильное решение задачи.

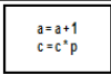


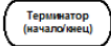
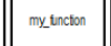
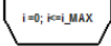
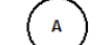

достижению цели. Разделение выполнения решения задачи на отдельные операции (выполняемые исполнителем по определенным командам) — важное свойство алгоритмов, называемое **дискретностью**.

Алгоритм, составленный для конкретного исполнителя, должен включать только те команды, которые входят в его систему команд. Это свойство алгоритма называется **понятностью**.

Еще одно важное требование, предъявляемое к алгоритмам, — **результативность (или конечность)** алгоритма. Оно означает, что исполнение алгоритма должно закончиться за конечное число шагов.

Формы записи алгоритмов

- записан на естественном языке (примеры записи алгоритма на естественном языке приведены при определении понятия алгоритма);
- изображен в виде блок-схемы;
- записан на алгоритмическом языке.

№	Символ	Наименование	Содержание
1		Блок вычислений	Вычислительные действия или последовательность действий
2		Логический блок	Выбор направления выполнения алгоритма в зависимости от некоторого условия
3		Блоки ввода-вывода данных	1. Общие обозначения ввода (вывода) данных (вне зависимости от физического носителя) 2. Вывод данных, носителем которых является документ
4		Начало (конец)	Начало или конец алгоритма, вход или выход в программу
5		Процесс пользователя (подпрограмма)	Вычисление по стандартной программе или подпрограмме
6		Блок модификации	Функция выполняет действия, изменяющие пункты (например, заголовок цикла) алгоритма
7		Соединитель	Указание связи прерванными линиями между потоками информации в пределах одного листа.
8		Межстраничные соединения	Указание связи между информацией на разных листах

Базовые структуры алгоритмов — это определенный набор блоков и стандартных способов их соединения для выполнения типичных последовательностей действий. К основным структурам относятся следующие: **линейные**, **разветвляющиеся**, **циклические**.

Линейными называются алгоритмы, в которых действия осуществляются последовательно друг за другом.

Разветвляющимся называется алгоритм, в котором действие выполняется по одной из возможных ветвей решения задачи, в зависимости от выполнения условий.

Циклическим называется алгоритм, в котором некоторая часть операций (тело цикла — последовательность команд) выполняется многократно.

Псевдокод

Компактный язык описания алгоритмов, использующий ключевые слова императивных языков программирования, но опускающий несущественные подробности и специфический синтаксис. Псевдокод обычно опускает детали, несущественные для понимания алгоритма человеком. Такими несущественными деталями могут быть описания переменных, системно-зависимый код и подпрограммы. Главная цель использования псевдокода - обеспечить понимание алгоритма человеком, сделать описание более воспринимаемым, чем исходный код на языке программирования. Псевдокод широко используется в учебниках и научно-технических публикациях, а также на начальных стадиях разработки компьютерных программ.

62.Оператор цикла. Оператор break. Оператор continue. Совместное использование операторов break и continue.

Оператор цикла.

В языке С имеется стандартный набор операторов цикла: for, while и do-while (называемый в некоторых других языках высоким уровнем цикла repeat-until). Вас может, однако, удивить то, каким способом программа выходит из цикла. В С можно изменить порядок выполнения цикла четырьмя способами. Естественно, все циклы заканчиваются при выполнении заданного проверочного условия. Однако, в С цикл может также закончиться по некоторому заданному условию ошибки при помощи операторов break или exit. Кроме этого, в циклах может быть собственная управляющая логика, изменяемая при помощи оператора break или оператора continue.

В теле любого цикла можно использовать **оператор break**, который позволяет выйти из цикла, не завершая его.

while(<условие>)

break;

или

switch(<целое выражение>)

{ case <константное выражение1>: <оператор1>; break;

...

}

Оператор continue

В С существует небольшое различие между операторами **break** и **continue**. В отличие от break, оператор continue приводит к игнорированию всех следующих за ним операторов, однако не препятствует инкременту переменной управления циклом или выполнению проверки условия продолжения цикла. Другими словами: если переменная управления циклом продолжает отвечать условию выполнения цикла, то цикл повторяется.

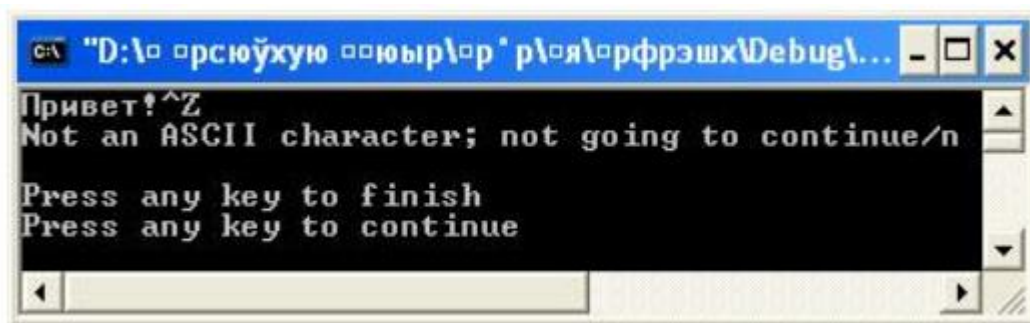
while(<условие>)

continue;

Совместное использование операторов break и continue.

Операторы break и continue можно использовать вместе для решения некоторых интересных задач программирования. Посмотрите на следующий пример на C++

```
using namespace std;
#define NEWLINE '\n'
main ()
{
    int c;
    while((c=getchar())!= EOF)
    {
        if (isascii(c) == 0)
        {
            cout << "Not an ASCII character; "; // Не ASCII-символ;
            cout << "not going to continue/n"; // продолжения нет
            break;
        }
        if(ispunct(c) || isspace(c)) {
            putchar(NEWLINE) ;
            continue;
        }
        if (isprint(c) == 0) {
            c = getchar();
            continue;
        }
        putchar(c);
    }
    printf ("\n\nPress any key to finish\n");
    _getch();
    return(0);
}
```



63. Оператор goto. Использование goto.

Оператор goto.

goto - оператор условного перехода. Оператор **goto** — одно из важнейших средств Бейсика и Фортрана — также реализован и в Си. Однако на этом языке в отличие от двух других можно программировать, совершенно не используя указанное средство. Керниган и Ритчи считают оператор **goto** «чрезвычайно плохим» средством и предлагают «применять его как можно реже или не применять совсем».

Оператор **goto** состоит из двух частей — ключевого слова **goto** и имени метки. Имена меток образуются по тем же правилам, что и имена переменных. Приведем пример записи оператора

goto part2;

Чтобы этот оператор выполнялся правильно, необходимо наличие другого оператора, имеющего метку **part2**; в этом случае запись оператора начинается с метки, за которой следует двоеточие.

part2: printf(" Уточненный анализ:\n");

Использование goto.

В принципе вы никогда не обязаны пользоваться оператором **goto** при программировании на Си. Но если ваш предыдущий опыт связан с работой на Фортране или Бейсике, в каждом из которых требуется его использовать, то у вас могли выработаться навыки программирования, основанные на применении данного оператора. Чтобы помочь вам преодолеть эту привычку, ниже вкратце приводятся несколько знакомых вам ситуаций, реализуемых с помощью **goto**, а затем показывается, как это можно осуществить другими средствами, в большей степени соответствующими духу языка Си.

1. Работа в ситуации, когда в операторе **if** требуется выполнить более одного оператора:

```
if (size > 12)
goto a;
goto b;
a: cost = cost * 1.05;
flag = 2;
b: bill = cost * flag;
```

Обычный подход, применяемый в языке Си и заключающийся в использовании составного оператора, или блока, упрощает понимание смысла программы:

```
if (size > 12);
```

```
{  
cost = cost * 1.05;  
flag = 2;  
}  
bill = cost * flag;
```

2. Осуществление выбора из двух вариантов:

```
if(size > 14)  
goto a;  
sheds = 2;  
goto b;  
a: sheds = 3;  
b: help = 2 * sheds;
```

Наличие в языке Си структуры if-else позволяет реализовать такой выбор более наглядно:

```
if(ibex > 14)  
sheds = 3;  
else  
sheds = 2;  
help = 2 * sheds;
```

3. Реализация бесконечного цикла:

```
readin: scanf("%d", &score);  
if(score < 0)  
goto stage2;  
большое количество операторов;  
goto readin;  
stage2: дополнительная чепуха;
```

Эквивалентный фрагмент, в котором используется цикл while, выглядит так:

```
scanf("%d", &score);
```



```
while(score >= 0)
{
    большое количество операторов;
    scanf(" %d", &score);
    дополнительная чепуха;
```

64.Оператор exit(). Оператор atexit(). Сравнение циклов.

Оператор exit().

В некоторых случаях программу необходимо закончить до того, как выполнялись все ее операторы или условия. Для таких особых случаев в С имеется библиотечная функция exit(). Эта функция может иметь один целочисленный аргумент, называемый статусом. Операционные системы UNIX и MS-DOS интерпретируют значение статуса, равное нулю, как успешное завершение программы, а все ненулевые значения статуса говорят о различного вида ошибках.

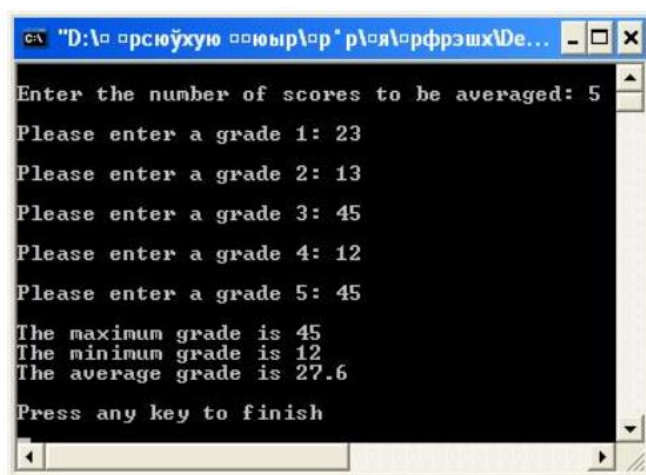
В следующей программе на С++ выполняется поиск среднего значения из списка, имеющего до 30 чисел. Выход из программы происходит, если пользователь просит усреднить большее, чем LIMIT, количество цифр.

```
using namespace std;
#define LIMIT 30
main()
{
    int irow,irequested_qty,iscORES[LIMIT];
    float fsum=0,imax_score=0,imin_score=100, faverage;
    // Введите количество усредняемых величин:
    cout << "\nEnter the number of scores to be averaged: ";
    cin >> irequested_qty;
    if(irequested_qty > LIMIT) {
        // Вы можете задать только ... чисел для усреднения,
        cout << "\nYou can only enter up to " << LIMIT << \
            " scores" << " to be averaged.\n";
        // Программа окончена.
        cout << "\n >>> Program was exited. <<<\n";
        exit;
    }
    for(irow = 0; irow < irequested_qty; irow++) {
        // Введите число
        cout << "\nPlease enter a grade " << irow+1 << ": ";
        cin >> iscores[irow];
    }
```

```

for(irow = 0; irow < irequested_qty; irow++)
fsum = fsum + iscores[irow];
faverage = fsum/(float)irequested_qty;
for(irow = 0; irow < irequested_qty; irow++) {
if(iscores[irow] > imax_score)
imax_score = iscores[irow];
if(iscores[irow] < imin_score)
imin_score = iscores[irow];
}
cout << "\nThe maximum grade is " << imax_score; // Максимум
cout << "\nThe minimum grade is " << imin_score; // Минимум
cout << "\nThe average grade is " << faverage; // Среднее значение
printf ("\n\nPress any key to finish\n");
_getch();
return(0);
}

```



Оператор atexit().

Всякий раз, когда программа вызывает функцию `exit()` или происходит нормальное завершение программы, можно также вызывать любую зарегистрированную "функцию выхода", занесенную в `atexit()`. Следующая программа на C иллюстрирует эту возможность:

```

void atexit_fn1(void);
void atexit_fn2(void);
void atexit_fn3(void);
main()
{
atexit(atexit_fn1);
atexit(atexit_fn2);

```

```

atexit(atexit_fn3) ;
printf("Atexit program entered.\n"); /* Вход в программу Atexit.*/
printf("Atexit program exited.\n\n"); /* Выход из программы Atexit.*/
printf(">>>>>>>>> <<<<<<<<, \n\n") ;
printf("\n\nPress any key to finish\n");
_getch();
return(0);
}

```

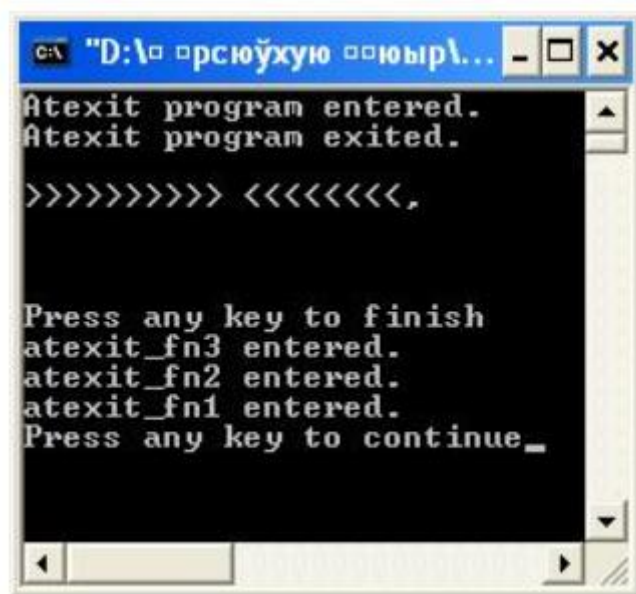
```

void atexit_fn1(void)
{
printf("atexit_fn1 entered.\n"); /* Вход в функцию atexit_fn1.*/
}

void atexit_fn2(void)
{
printf("atexit_fn2 entered.\n"); /* Вход в функцию atexit_fn2.*/
}

void atexit_fn3(void)
{
printf("atexit_fn3 entered.\n"); /* Вход в функцию atexit_fn3.*/
}

```



65. Массивы. Понятие массив. Массивы в С. Объявление массивов. Доступ к элементам массива. Размещение массивов в памяти. Проблема ввода.

Массивы.

Массив — это набор переменных, имеющих одно и то же базовое имя и отличающихся одна от другой числовым признаком. Например, с помощью описания

float debts [20];

объявляется, что **debts** — массив, состоящий из двадцати членов, или «элементов». Первый элемент массива называется `debts[0]`, второй — `debts[1]`, и т. д. вплоть до `debts[19]`. Заметим, что перечисление элементов массива начинается с 0, а не с 1. Поскольку мы объявили, что массив имеет тип **float**, каждому его элементу можно присвоить величину типа `float`.

Понятие массив.

Можно рассматривать **массивы** как переменные, содержащие несколько элементов одного типа. Доступ к каждому отдельному элементу данных осуществляется при помощи индекса этой переменной. В языке С массивы не являются стандартным типом данных; они представляют собой составной тип, созданный на основе других типов данных. В С возможно создавать массивы из любых типов переменных: символов, целых, чисел двойной длины, массивов, указателей, структур и так далее. В общих чертах концепции массивов и способы их использования в С и С++ совпадают.

Массивы в С.

Массивы имеют четыре основных характеристики:

- Отдельные объекты данных в массиве называются элементами.
- Все элементы массива должны иметь одинаковый тип данных.
- Все элементы располагаются в памяти компьютера последовательно, и индекс первого элемента равен нулю.
- Имя массива является постоянным значением, представляющим собой адрес первого элемента массив

Объявление массивов.

Ниже приведены примеры объявлений массивов:

```
int iarray[12]; /* массив из 12 целых чисел */
```

```
char carray[20]; /* массив из 20 символов */
```

Проблема ввода.

Язык Си предоставляет много средств для структурирования программ. С помощью операторов `while` и `for` реализуются циклы с предусловием. Второй оператор особенно подходит для циклов, включающих в себя инициализацию и коррекцию переменной. Использование операции «запятая» в цикле `for` позволяет инициализировать и корректировать более одной переменной. Для тех редких случаев, когда требуется использовать цикл с постусловием, язык Си предоставляет оператор `do while`. Операторы `break`, `continue` и `goto` обеспечивают дополнительные возможности управления ходом выполнения программы.

Массивы в программе описываются так же, как обычные переменные, но при этом в квадратных скобках указывается число элементов. Первому элементу массива присваивается номер 0, второму — номер 1 и т. д. Индексы, используемые для нумерации элементов массива, могут обрабатываться обычным образом при помощи циклов.

66.Массивы. Инициализация массивов. Инициализация по умолчанию. Явная инициализация. Инициализация безразмерных массивов.

Инициализация массивов.

Имеется три способа инициализации массивов:

- По умолчанию во время их создания. Применимо только к глобальным и статическим (static) массивам.
- Явно во время создания при помощи констант инициализации.
- Во время выполнения программы при присваивании или копировании данных в массив.

Для инициализации массива во время создания можно использовать только константы. Если элементы массива должны получать значения из переменных, то в программном коде должны быть явные операторы инициализации массива.

Инициализация по умолчанию.

В стандарте ANSI C оговорено, что массивы бывают либо глобальные (описанные вне main() и любых других функций), либо автоматические static (статические, но описанные после какой-либо открывающей скобки), и при отсутствии инициализирующей информации они всегда получают значения двоичных нулей. В C числовые массивы инициализируются значением ноль. (Массивы указателей получают начальные значения null)

Явная инициализация.

Аналогично описаниям и инициализации переменных типов **int**, **char**, **float**, **double** и других вы можете инициализировать массивы. Стандарт ANSI C позволяет задавать и инициализирующие значения любого массива, глобального или иного, описанного в любой части программы. Следующий фрагмент кода иллюстрирует описание и инициализацию массивов:

```
int iarray[3] = {-1,0,1};  
  
static float fpercent[4] = {1.141579,0.75,55E0,-.33E1} ;  
  
static int idecimal[3] = {0,1,2,3,4,5,6,7,8,9};
```

Безразмерные массивы

Для большинства компиляторов безразлично — указываете ли вы размер массива, или задаете список фактических значений; важно указать либо то, либо другое. Например: в программах часто описывается набор собственных сообщений об ошибках. Это можно сделать двумя способами. Вот первый из них:

```
char e1[12] = "Ошибка чтения\n";  
char e2[13] = "Ошибка записи\n";  
char e3[18] = "Нельзя открыть файл\n";
```

При таком подходе необходимо считать количество символов в строке, не забывая добавить 1, чтобы учесть невидимый null-символ окончания строки \0. Это, в лучшем случае, очень утомительный для глаз метод, который чреват многими ошибками. При втором способе, показанном ниже, используются безразмерные массивы, и язык C автоматически определяет их размер:

```
char e1[] = "Ошибка чтения\n";  
char e2[] = "Ошибка записи\n";  
char e3[] = "Нельзя открыть файл\n";
```

Всякий раз, когда встречается оператор инициализации массива и размер массива не указан, компилятор автоматически выделяет достаточно места для всех указанных данных.

67. Массивы. Инициализация массивов и классы памяти. Вычисление размера массива (sizeof()). Выход индекса за пределы массива.

Инициализация массивов и классы памяти.

Для хранения данных, необходимых программе, часто используют массивы. Например, в массиве из 12 элементов можно хранить информацию о количестве дней каждого месяца. В подобных случаях желательно иметь удобный способ инициализации массива перед началом работы программы. Такая возможность, вообще говоря, существует, но только для статической и внешней памяти. Давайте посмотрим, как она используется.

Мы знаем, что скалярные переменные можно инициализировать в описании типа при помощи таких выражений, как, например:

int fix = 1;

Можем ли мы делать что-либо подобное с массивом? Ответ не однозначен: и да, и нет.

Внешние и статические массивы можно инициализировать. Автоматические и регистровые массивы инициализировать нельзя.

Прежде чем попытаться инициализировать массив, давайте посмотрим, что там находится, если мы в него ничего не записали.

```
main ()
{
int fuzzy[2]; /* автоматический массив */
static int wuzzy[2]; /* статический массив */
printf(" %d %d\n" , fuzzy[1], wuzzy[1]);
}
```



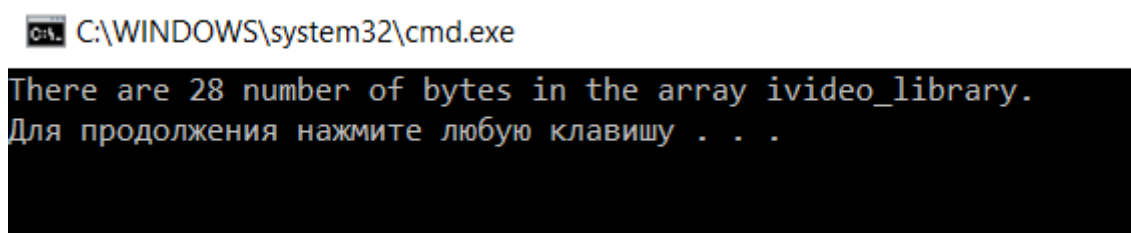
Если ничего не засылать в массив перед началом работы с ним, то внешние, статические и автоматические массивы инициализируются для числовых типов нулем и '\0' (null) для символьных типов, а регистровые массивы содержат какой-то мусор, оставшийся в этой части памяти

Вычисление размера массива (sizeof()).

Как вы уже знаете, операция **sizeof()** возвращает физический размер в байтах того объекта, к которому она применяется. Ее можно использовать с объектами любых типов, за исключением

битовых полей. Часто операция `sizeof()` применяется для определения физического размера переменной в тех случаях, когда размер переменных этого типа может меняться от одного компьютера к другому. Вы уже видели, что в зависимости от используемой системы целые числа могут иметь длину 2 или 4 байта. Если у операционной системы запрашивается дополнительная память для размещения семи целых чисел, то необходим способ для определения количества памяти: либо 14 байт (7x2 байта на число), либо 28 байт (7x4 байта на число). В следующей программе этот вопрос решается автоматически (и печатается значение 14 для тех систем, в которых для целого числа отводится 2 байта):

```
using namespace std;
#define IDAYS_OF_WEEK 7
int main()
{
    int ivideo_library[IDAYS_OF_WEEK] = { 1,2,3,4,5,6,7 };
    printf("There are %d number of bytes in the array"
        " ivideo_library.\n", (int) sizeof(ivideo_library));
    return(0);
}
```



The screenshot shows a Windows command prompt window titled "C:\WINDOWS\system32\cmd.exe". The output of the program is displayed in two lines: "There are 28 number of bytes in the array ivideo_library." and "Для продолжения нажмите любую клавишу . . .".

Выход индекса за пределы массива.

Вам, наверное, знакомо выражение: "За все нужно платить". Это справедливо и для массивов в С. "Все" — это быстрое выполнение кода программы, а "плата" — отсутствие проверки границ массива. Напомним: поскольку язык С был разработан для замены ассемблера, в целях обеспечения компактности кода, ошибки такого рода компилятор не проверяет. Поскольку компилятор никак не определяет ошибочные ситуации, вы должны быть очень осторожны при работе с граничными значениями индекса массива.

68. Массивы. Многомерные массивы. Инициализация двумерного массива.

Массив - это структура данных, представленная в виде группы ячеек одного типа, объединенных под одним единым именем. Массивы используются для обработки большого количества однотипных данных. Имя массива является указателем.

Отдельная ячейка данных массива называется *элементом массива*. Элементами массива могут быть данные любого типа. Массивы могут иметь как одно, так и более одного измерений. В зависимости от количества измерений массивы делятся на одномерные массивы, двумерные массивы, трёхмерные массивы и так далее до *n*-мерного массива. Чаще всего в программировании используются одномерные и двумерные массивы

Одномерный массив — массив, с одним параметром, характеризующим количество элементов одномерного массива. Фактически одномерный массив — это массив, у которого может быть только одна строка, и n-е количество столбцов. Столбцы в одномерном массиве — это элементы массива. На рисунке 1 показана структура целочисленного одномерного массива а. Размер этого массива — 16 ячеек.

5	-12	-12	9	10	0	-9	-12	-1	23	65	64	11	43	39	-15
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]	a[11]	a[12]	a[13]	a[14]	a[15]

Рисунок 1 — Массивы в C++

Инициализация одномерного массива выполняется в фигурных скобках после знака **равно**, каждый элемент массива отделяется от предыдущего запятой.

```
int a[] = { 4,8,9,5,-2,7,2,1,1,0,13,-1 };
           //объявление и инициализация
```

Заметьте, что максимальный индекс одномерного массива а равен 15, но размер массива 16 ячеек, потому что нумерация ячеек массива всегда начинается с 0. Индекс ячейки – это целое неотрицательное число, по которому можно обращаться к каждой ячейке массива и выполнять какие-либо действия над ней (ячейкой).

C позволяет создавать многомерные массивы. Простейшим видом многомерного массива является двумерный массив. Двумерный массив - это массив одномерных массивов. Двумерный массив объявляется следующим образом:

тип имя_массива[размер второго измерения][размер первого измерения];

Следовательно, для объявления двумерного массива целых с размером 10 на 20 следует написать:

```
int d[10][20] ; // объявление
int a[4][3] = { {4,8,9}, {5,-2,7}, {2,1,1}, {0,13,-1} };
               //объявление и инициализация
```

Посмотрим внимательно на это объявление. В противоположность другим компьютерным языкам, где размерности массива отделяются запятой, C помещает каждую размерность в отдельные скобки.

Для доступа к элементу с индексами 3, 5 массива d следует использовать

```
d[3][5]
```

В следующем примере в двумерный массив заносятся числа от 1 до 12, после чего массив выводится на экран.

```
#include <stdio.h>
int main(void)
{
    int t,i, num[3][4];
    /* загрузка чисел */
    for(t=0; t<3; ++t)
        for (i=0; i<4; ++i)
            num[t][i] = (t*4)+i+1;

    /* вывод чисел */
    for (t=0; t<3; ++t)
    {
        for (i=0; i<4; ++i)
            printf("%d ",num[t][i]);
        printf ("\n");
    }
    return 0;
}
```

В данном примере num[0][0] имеет значение 1, num[0][1] имеет значение 2, num[0][2] - 3 и так далее. num[2][3] имеет значение 12.

Двумерные массивы сохраняются в виде матрицы, где первый индекс отвечает за строку, а второй -за столбец. Это означает, что правый индекс изменяется быстрее левого, если двигаться по массиву в порядке расположения элементов в памяти. На рис. показано графическое представление двумерного массива в памяти. Левый индекс можно рассматривать как указатель на строку.

Число байт в памяти, требуемых для размещения двумерного массива, вычисляется следующим образом:

*число байт = размер второго измерения * размер первого измерения * sizeof (базовый тип)*

Предполагая наличие в системе 2-байтных целых, целочисленный массив с размерностями 10 на 5 будет занимать $10 * 5 * 2$, то есть 100 байт.

69. Массивы. Массивы в качестве аргументов функций. Передача массивов функциям C. Передача массивов функциям C++.

Массив — это набор переменных одного типа, имеющих одно и то же имя. Доступ к конкретному элементу массива осуществляется с помощью индекса. В языке C все массивы располагаются в отдельной непрерывной области памяти. Первый элемент массива располагается по самому меньшему адресу, а последний — по самому большому. Массивы могут быть одномерными и многомерными. *Строка* представляет собой массив символьных переменных, заканчивающийся специальным нулевым символом, это наиболее распространенный тип массива.

Одномерный массив — массив, с одним параметром, характеризующим количество элементов одномерного массива. Фактически одномерный массив — это массив, у которого может быть только одна строка, и n-е количество столбцов. Столбцы в одномерном массиве — это элементы массива.

```
int a[11]; //объявление
```

```
int a[11] = { 4,8,9,5,-2,7,2,1,1,0,13,-1 }; //объявление и инициализация
```

```
int a[] = { 4,8,9,5,-2,7,2,1,1,0,13,-1 };  
//объявление и инициализация
```

С позволяет создавать многомерные массивы. Простейшим видом многомерного массива является двумерный массив. Двумерный массив - это массив одномерных массивов. Двумерный массив объявляется следующим образом:

тип имя_массива[размер второго измерения][размер первого измерения];

Следовательно, для объявления двумерного массива целых с размером 10 на 20 следует написать:

```
int d[10][20]; // объявление  
int a[4][3] = { {4,8,9}, {5,-2,7}, {2,1,1}, {0,13,-1} };  
//объявление и инициализация
```

Вызов функций с помощью массивов

В настоящем же разделе рассказывается о передаче массивов функциям в качестве аргументов. Этот вопрос рассматривается потому, что эта операция является исключением по отношению к обычной передаче параметров, выполняемой путем вызова по значению (Ведь при вызове по значению пришлось бы копировать весь массив!).

Когда в качестве аргумента функции используется массив, то функции передается его адрес. В этом и состоит исключение по отношению к правилу, которое гласит, что при передаче параметров используется вызов по значению. В случае передачи массива функции ее внутренний код работает с реальным содержимым этого массива и вполне может изменить это содержимое. Проанализируйте, например, функцию `print_upper()`, которая печатает свой строковый аргумент на верхнем регистре:

```
#include <stdio.h>  
#include <ctype.h>  
  
void print_upper(char *string);  
  
int main(void)  
{  
    char s[80];  
  
    printf("Введите строку символов: ");  
    gets(s);  
    print_upper(s);  
    printf("\ns теперь на верхнем регистре: %s", s);  
    return 0;  
}  
  
/* Печатать строку на верхнем регистре. */
```

```

void print_upper(char *string)
{
    register int t;

    for(t=0; string[t]; ++t) {
        string[t] = toupper(string[t]);
        putchar(string[t]);
    }
}

```

Вот что будет выведено в случае фразы "This is a test." (это тест):

```

Введите строку символов: This
is a test.
THIS IS A TEST.
s теперь в верхнем регистре:
THIS IS A TEST.

```

Правда, эта программа не работает с символами кириллицы.

После вызова `print_upper()` содержимое массива `s` в `main()` переводится в символы верхнего регистра. Если вам это не нужно, программу можно написать следующим образом:

```

#include <stdio.h>
#include <ctype.h>

void print_upper(char *string);

int main(void)
{
    char s[80];

    printf("Введите строку символов: ");
    gets(s);
    print_upper(s);
    printf("\ns не изменялась: %s", s);

    return 0;
}

void print_upper(char *string)
{
    register int t;

    for(t=0; string[t]; ++t)
        putchar(toupper(string[t]));
}

```

Вот какой на этот раз получится фраза "This is a test.":

```

Введите строку символов:
This is a test.
THIS IS A TEST.
s не изменилась: This is a test.

```

На этот раз содержимое массива не изменилось, потому что внутри `print_upper()` не изменялись его значения.

Классическим примером передачи массивов в функции является стандартная библиотечная функция `gets()`. Хотя `gets()`, которая находится в вашей стандартной

библиотеке, и более сложная, чем предлагаемая вам версия `xgets()`, но с помощью функции `xgets()` вы сможете получить представление о том, как работает `gets()`.

```
/* Упрощенная версия стандартной библиотечной функции gets().
*/
char *xgets(char *s)
{
    char ch, *p;
    int t;

    p = s; /* xgets() возвращает указатель s */

    for(t=0; t<80; ++t){
        ch = getchar();

        switch(ch) {
            case '\n':
                s[t] = '\0'; /* завершает строку */
                return p;
            case '\b':
                if(t>0) t--;
                break;
            default:
                s[t] = ch;
        }
    }
    s[79] = '\0';
    return p;
}
```

Функцию `xgets()` следует вызывать с указателем `char *`.

Им, конечно же, может быть имя символьного массива, которое по определению является указателем `char *`. В самом начале программы `xgets()` выполняется цикл `for` от 0 до 80. Это не даст вводить с клавиатуры строки, содержащие более 80 символов. При попытке ввода большего количества символов происходит возврат из функции. (В настоящей функции `gets()` такого ограничения нет.) Так как в языке С нет встроенной проверки границ, программист должен сам позаботиться, чтобы в любом массиве, используемом при вызове `xgets()`, помещалось не менее 80 символов. Когда символы вводятся с клавиатуры, они сразу записываются в строку. Если пользователь нажимает клавишу `<Backspace>`, то счетчик `t` уменьшается на 1, а из массива удаляется последний символ, введенный перед нажатием этой клавиши. Когда пользователь нажмет `<ENTER>`, в конец строки запишется ноль, т.е. признак конца строки. Так как массив, использованный для вызова `xgets()`, модифицируется, то при возврате из функции в нем будут находиться введенные пользователем символы.

70. Ввод и вывод строк. Строковые функции и символьные массивы. Динамическое выделение памяти. Функции `gets()`, `puts()`, `fgets()`, `fputs()` и `sprintf()`. Функции `strcpy()`, `strcat()`, `strncmp()` и `strlen()`.

Функция	Пояснение
<u>strlen</u> (имя_строки)	определяет длину указанной строки, без учёта нуля-символа
Копирование строк	
<u>strcpy</u> (s1,s2)	выполняет побайтное копирование символов из строки s2 в строку s1
<u>strncpy</u> (s1,s2, n)	выполняет побайтное копирование n символов из строки s2 в строку s1. возвращает значения s1
Объединение строк	
<u>strcat</u> (s1,s2)	объединяет строку s2 со строкой s1. Результат сохраняется в s1
<u>strncat</u> (s1,s2,n)	объединяет n символов строки s2 со строкой s1. Результат сохраняется в s1
Сравнение строк	
<u>strcmp</u> (s1,s2)	сравнивает строку s1 со строкой s2 и возвращает результат типа int : 0 –если строки эквивалентны, >0 – если s1<s2, <0 — если s1>s2 С учётом регистра
<u>strncmp</u> (s1,s2)	сравнивает n символов строки s1 со строкой s2 и возвращает результат типа int : 0 –если строки эквивалентны, >0 – если s1<s2, <0 — если s1>s2 С учётом регистра
<u>stricmp</u> (s1,s2)	сравнивает строку s1 со строкой s2 и возвращает результат типа int : 0 –если строки эквивалентны, >0 – если s1<s2, <0 — если s1>s2 Без учёта регистра
Функции преобразования	
<u>atof</u> (s1)	преобразует строку s1 в тип double
<u>atoi</u> (s1)	преобразует строку s1 в тип int
<u>atol</u> (s1)	преобразует строку s1 в тип long int

Символ – элементарная единица, некоторый набор которых несет определенный смысл. В языке программирования C++ предусмотрено использование символьных констант. Символьная константа – это целочисленное значение (типа **int**) представленное в виде символа, заключённого в одинарные кавычки, например **'a'**. В таблице ASCII представлены символы и их целочисленные значения.

```

1 // объявления символьной переменной
2 char symbol = 'a';
3 // где symbol - имя переменной типа char
4 // char - тип данных для хранения символов
```

Строки в C++ представляются как массивы элементов типа **char**, заканчивающиеся нуль-терминатором **\0** называются C строками или строками в стиле C.

\0 — символ нуль-терминатора.

```

1 //пример объявления строки
2 char string[10];
3 //где string - имя строковой переменной
4 //10 - размер массива, то есть в данной строке может поместиться 9 символов,
    последнее место отводится под нуль-терминатор.

```

Символьные строки состоят из набора символьных констант заключённых в двойные кавычки. При объявлении строкового массива необходимо учитывать наличие в конце строки нуль-терминатора, и отводить дополнительный байт под него.

Строка при объявлении может быть инициализирована начальным значением, например, так:

```
1 char string[10] = "abcdefghf";
```

Если подсчитать кол-во символов в двойных кавычках после символа равно их окажется 9, а размер строки 10 символов, последнее место отводится под нуль-терминатор, причём компилятор сам добавит его в конец строки.

```

1 // посимвольная инициализация строки:
2 char string[10] = {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'f', '\0'};
3 // десятый символ это нуль-терминатор.

```

При объявлении строки не обязательно указывать её размер, но при этом обязательно нужно её инициализировать начальным значением. Тогда размер строки определится автоматически и в конец строки добавится нуль-терминатор.

Строка может содержать символы, цифры и специальные знаки. В C++ строки заключаются в двойные кавычки. Имя строки является константным указателем на первый символ. Разработаем программу, с использованием строк.

```

1 // symbols.cpp: определяет точку входа для консольного приложения.
2
3 #include "stdafx.h"
4 #include <iostream>
5 using namespace std;
6
7 int main(int argc, char* argv[])
8 {
9     char string[] = "this is string - "; // объявление и инициализация строки
10    cout << "Enter the string: ";
11    char in_string[500]; // строковый массив для ввода
12    gets(in_string); // функция gets() считывает все введённые символы
13    // с пробелами до тех пор, пока не будет нажата клавиша Enter
14    cout << string << in_string << endl; // вывод строкового значения
15    system("pause");
16    return 0;
17 }

```

В строке 12 с помощью функции `gets()` считаются все введенные символы с пробелами до тех пор, пока во вводимом потоке не встретится код клавиши `enter`. Если использовать операцию `cin` то из всего введенного считается последовательность символов до первого пробела.

71. Указатели. Определение переменных-указателей. Разыменование указателей. Объявление переменных-указателей. Простые операторы с указателями. Инициализация указателей. Неправильное использование операции определения адреса.

Указатель – переменная, значением которой является адрес ячейки памяти. То есть указатель ссылается на блок данных из области памяти, причём на самое его начало. Указатель может ссылаться на переменную или функцию. Для этого нужно знать адрес переменной или функции. Так вот, чтобы узнать адрес конкретной переменной в C++ существует унарная операция взятия адреса `&`. Такая операция извлекает адрес объявленных переменных, для того, чтобы его присвоить указателю.

```
//объявление указателя
/*тип данных*/ * /*имя указателя*/;
```

Основной операцией при работе с указателями является получение доступа к значению, адрес которого хранится в указателе.

Например:

```
int *pn, n;
*pn = 5;
n = *pn;
```

Выражение `*pn` имеет такой же смысл, как имя целой переменной. Операция `<<*>>` называется разыменованием.

Действие, обратное к разыменованию, позволяет получить адрес переменной по ее имени. Например,

```
pn = &n;
```

эта операция называется взятие адреса.

72. Указатели. Указатели на массивы. Указатели и многомерные массивы. Указатели на указатели. Указатели на строки.

Указатель — переменная, содержащая адрес объекта. Указатель не несет информации о содержимом объекта, а содержит сведения о том, где размещен объект.

Память компьютера можно представить в виде последовательности пронумерованных однобайтовых ячеек, с которыми можно работать по отдельности или блоками.

Каждая переменная в памяти имеет свой адрес - номер первой ячейки, где она расположена,

а также свое значение. Указатель — это тоже переменная, которая размещается в памяти. Она тоже имеет адрес, а ее значение является адресом некоторой другой переменной. Переменная, объявленная как указатель, занимает 4 байта в оперативной памяти (в случае 32-битной версии компилятора).

Указатель, как и любая переменная, должен быть объявлен.
Общая форма объявления указателя

тип *ИмяОбъекта;

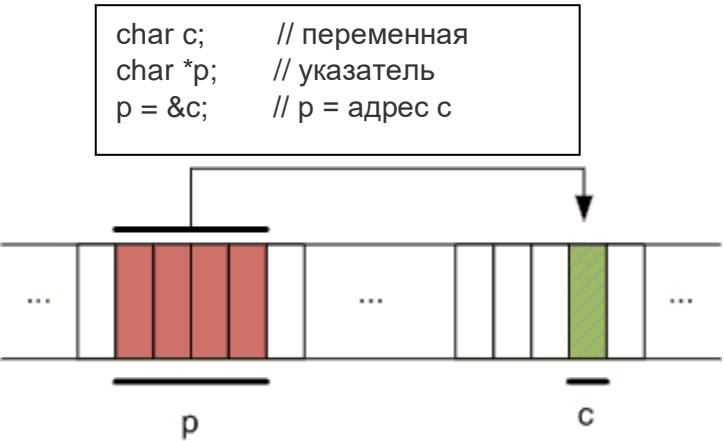
Тип указателя— это тип переменной, адрес которой он содержит.

Для работы с указателями в Си определены две операции:

операция * (звездочка) — позволяет получить значение объекта по его адресу - определяет значение переменной, которое содержится по адресу, содержащемуся в указателе;

операция & (амперсанд) — позволяет определить адрес переменной.

Например,



Для указанного примера обращение к одним и тем же значениям переменной и адреса представлено в таблице

	Переменная	Указатель
Адрес	&c	p
Значение	c	*p

Пример на Си

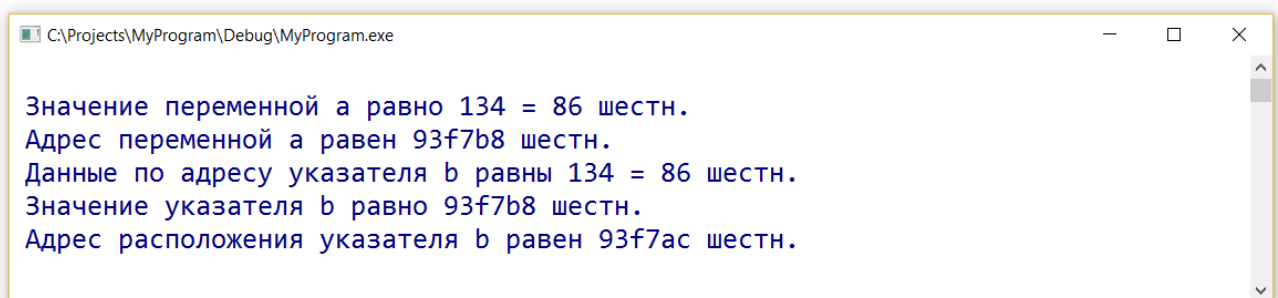
```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int a, *b;
    system("chcp 1251");
```

```

system("cls");
a = 134;
b = &a;
// %x = вывод числа в шестнадцатеричной форме
printf("\n Значение переменной a равно %d = %x шестн.", a,a);
printf("\n Адрес переменной a равен %x шестн.", &a);
printf("\n Данные по адресу указателя b равны %d = %x шестн.", *b,*b);
printf("\n Значение указателя b равно %x шестн.", b);
printf("\n Адрес расположения указателя b равен %x шестн.", &b);
getchar();
return 0;
}

```

Результат выполнения программы:



Расположение в памяти переменной a и указателя b:

Адрес		0093F7AC	0093F7AD	0093F7AE	0093F7AF	...	0093F7B8	0093F7B9	0093F7BA	0093F7BB		
		B8	F7	93	00	...	86	00	00	00		
Значение		B8	F7	93	00	...	86	00	00	00		
		b					a					

Необходимо помнить, что компиляторы высокого уровня поддерживают прямой способ адресации: младший байт хранится в ячейке, имеющей младший адрес.

Указатель на массив:

Код:

```

int* myM;
myM = new int[10];

```

Возможно объявление переменной, которая содержит адрес другой переменной, которая, в свою очередь, также является указателем. Такая переменная может быть необходима, если в функции нужно изменить адрес какого-либо объекта. Однако наличие более двух звёздочек в объявлении переменной говорит, скорее всего, о плохом проектировании.

```

int **ppi;           // Объявляем указатель на указатель на целое

void f(int **ppi)
{
    // Указателю на целое присваивается значение, хранящееся по адресу,
    int *pi = *ppi;
    ...              // содержащемуся в указателе на указатель на целое
}

```

Строка – это последовательность (массив) символов (типа `char`), которая заканчивается специальным символом – признаком конца строки. Это символ записывается как `'\0'` (не путайте с символом переноса строки `'\n'`) и равен 0. При вводе строки символ конца строки добавляется автоматически. Все функции работы со строками – и стандартные, и создаваемые программистом – должны ориентироваться на этот символ. Если требуется сформировать новую строку, то обязательно надо добавлять признак конца строки. Если этого не сделать, то при дальнейшей работе возникнут ошибки.

Строковым литералом называется последовательность символов, заключённых в двойные кавычки. В строковом литерале на один символ больше, чем используется при его записи – добавляется символ `'\0'`.

Заголовки стандартных функций работы со строками хранятся в файле `<string.h>`. Основными из этих функций являются:

- Определение длины строки – `int strlen(const char *str);`
- Сравнение строк – `int strcmp(const char *str1, const char *str2);`
- Копирование – `char *strcpy(char *str1, const char *str2);`
- Конкатенация строк – `char *strcat(char *str1, const char *str2);`
- Поиск символа в строке – `char *strchr(const char *str, char c);`
- Поиск подстроки – `char *strstr(const char *str1, const char *str2);`

Ввод/вывод строки:

- Ввод строки до пробела или другого разделителя – функция `scanf` с форматом `%s`;
- Ввод строки, содержащей пробелы – `char *gets (char *buffer);`
- Ввод строки из файла, `n` задаёт максимальное количество символов для ввода – `char *fgets(char *string, int n, FILE *stream);`
- Вывод строки с форматированием – функция `printf` с форматом `%s`;
- Вывод строки – `int puts(const char *string);`

- Вывод строки в файл – `int fputs(const char *string, FILE *stream);`

Пример 1. Функция, которая меняет все вхождения буквы «я» на «а», «а» — на «б», «б» — на «в» и т.д. Остальные символы остаются без изменения

```
char* Change(char *str)
{ char *p;

  for (p = str; *p; p++)
    if (*p == 'я')
      *p = 'а';
    else if ('а' <= *p && *p <= 'ю')
      (*p)++;
  return str;
}
```

73. Указатели. Арифметические операции с указателями.

Арифметические операции с указателями и массивы. Операции с указателями.

Указатель — переменная, содержащая адрес объекта. Указатель не несет информации о содержимом объекта, а содержит сведения о том, где размещен объект.

Память компьютера можно представить в виде последовательности пронумерованных однобайтовых ячеек, с которыми можно работать по отдельности или блоками.

Каждая переменная в памяти имеет свой адрес - номер первой ячейки, где она расположена, а также свое значение. Указатель — это тоже переменная, которая размещается в памяти. Она тоже имеет адрес, а ее значение является адресом некоторой другой переменной. Переменная, объявленная как указатель, занимает 4 байта в оперативной памяти (в случае 32-битной версии компилятора).

Указатель, как и любая переменная, должен быть объявлен.
Общая форма объявления указателя

тип *ИмяОбъекта;

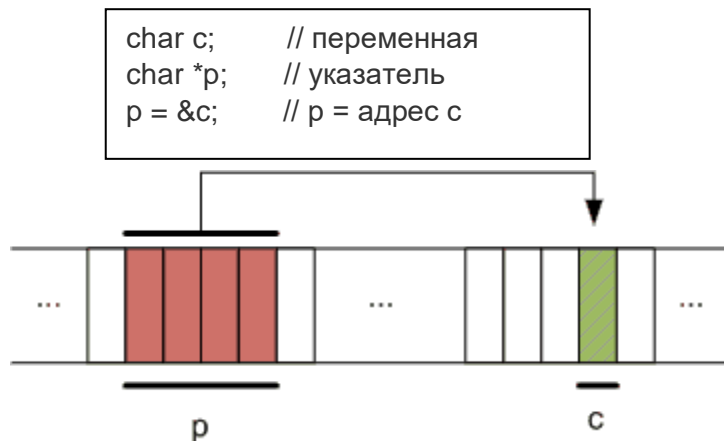
Тип указателя— это тип переменной, адрес которой он содержит.

Для работы с указателями в Си определены две операции:

операция * (звездочка) — позволяет получить значение объекта по его адресу - определяет значение переменной, которое содержится по адресу, содержащемуся в указателе;

операция & (амперсанд) — позволяет определить адрес переменной.

Например,



Для указанного примера обращение к одним и тем же значениям переменной и адреса представлено в таблице

	Переменная	Указатель
Адрес	&c	p
Значение	c	*p

Пример на Си

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int a, *b;
    system("chcp 1251");
    system("cls");
    a = 134;
    b = &a;
    // %x = вывод числа в шестнадцатеричной форме
    printf("\n Значение переменной a равно %d = %x шестн.", a,a);
    printf("\n Адрес переменной a равен %x шестн.", &a);
    printf("\n Данные по адресу указателя b равны %d = %x шестн.", *b,*b);
    printf("\n Значение указателя b равно %x шестн.", b);
    printf("\n Адрес расположения указателя b равен %x шестн.", &b);
    getchar();
    return 0;
}
```

Результат выполнения программы:

```
C:\Projects\MyProgram\Debug\MyProgram.exe

Значение переменной a равно 134 = 86 шестн.
Адрес переменной a равен 93f7b8 шестн.
Данные по адресу указателя b равны 134 = 86 шестн.
Значение указателя b равно 93f7b8 шестн.
Адрес расположения указателя b равен 93f7ac шестн.
```

Расположение в памяти переменной a и указателя b:

Адрес		0093F7AC	0093F7AD	0093F7AE	0093F7AF	...	0093F7B8	0093F7B9	0093F7BA	0093F7BB	
	Значение	B8	F7	93	00	...	86	00	00	00	
		b					a				

Необходимо помнить, что компиляторы высокого уровня поддерживают прямой способ адресации: младший байт хранится в ячейке, имеющей младший адрес.

В следующих двух программах с помощью индекса выполняется обработка массива из десяти символов. Обе программы считывают десять символов и затем печатают их в обратном порядке. В первой программе используется более традиционный для языков высокого уровня подход: обращение к элементам массива при помощи индексов. Вторая программа работает аналогично за исключением того, что обращение к элементам массива выполняется по адресу с использованием арифметических операций с указателями. Вот первая программа:

```
/*10ARYSUB.C
Программа на C, использующая обычные индексы массива*/

#include "stdafx.h"
#include "E:\LECTURE\AlgorithmProgramming 02\Universal_HederFile.h"
#define ISIZE 10
void StopWait(void);
main()
{
    char string10[ISIZE];
    int i;
    fputs("Enter 11 symbols\n",stdout);
    for(i=0; i < ISIZE; i++)
        string10[i]=getchar();
    for(i=ISIZE-1; i >= 0; i--)
        putchar(string10[i]);
    fputs("\n",stdout);
}
```

Вот второй пример:

```
/*10ARYPTR.C
Программа на С, в которой для доступа к элементам массива используются арифметические
операции с указателями.*/
#include "stdafx.h"
#include "E:\LECTURE\AlgorithmProgramming 02\Universal_HederFile.h"
#define ISIZE 10
void StopWait(void);
main()
{
char string10[ISIZE], *pc;
int icount;
pc=&string10[0];
fputs("Enter 11 symbols\n",stdout);
for(icount=0; icount < ISIZE; icount++)
{
*pc=getchar();
pc++;
}
fputs("\n",stdout);
pc=&string10[ISIZE-1];
for(icount=0; icount < ISIZE; icount++)
{
putchar(*pc);
pc--;
}
fputs("\n",stdout);
StopWait(); /* Wait a little */
return (0);
}
```

Над адресами в С++ определены следующие арифметические операции:

- сложение и вычитание указателей с константой;
- вычитание одного указателя из другого;
- инкремент;
- декремент.

Инкремент перемещает указатель к следующему элементу массива, а **декремент** – к предыдущему;

К указателям могут применяться только две арифметические операции: сложение и вычитание. Для понимания арифметических действий с указателями предположим, что p1 - это указатель на целое, содержащий значение 2000, и будем считать, что целые имеют длину 2 байта. После выражения

```
p1 ++;
```

содержимое p1 станет 2002, а не 2001! Каждый раз при увеличении p1 указатель будет указывать на следующее целое. Это справедливо и для уменьшения. Например:

p1 --;

приведет к тому, что p1 получит значение 1998, если считать, что раньше было 2000.

Каждый раз, когда указатель увеличивается, он указывает на следующий элемент базового типа. Каждый раз, когда уменьшается - на предыдущий элемент. В случае указателей на символы это приводит к «обычной» арифметике. Все остальные указатели увеличиваются или уменьшаются на длину базового типа. Рис. демонстрирует данную концепцию.

Естественно, все не ограничивается только уменьшением или увеличением. Можно добавлять или вычитать из указателей целые числа. Выражение

p1 = p1 + 9;

приводит к тому, что указатель p1 указывает на девятый элемент по сравнению с элементом, на который он указывал до присваивания.

Помимо добавления или вычитания указателей и целых чисел, единственную операцию, которую можно выполнять с указателями, - это вычитание одного указателя из другого.

В большинстве случаев вычитание одного указателя из другого имеет смысл только тогда, когда оба указателя указывают на один объект, - как правило, массив. В результате вычитания получается число элементов базового типа, находящихся между указателями. Помимо этих операций не существует других арифметических операций, применимых к указателям. Нельзя умножать или делить указатели, нельзя складывать указатели, нельзя применять битовый сдвиг или маски к указателям, нельзя добавлять или вычитать типы float или double.

К указателям так же применимы *операции отношения* ==, !=, <, >, <=, >=. Иными словами, указатели можно сравнивать. Например, если i указывает на пятый элемент массива, а j — на первый, то отношение i > j истинно. Кроме того, любой указатель можно сравнивать на равенство с нулем. Однако, все эти утверждения верны, если речь идет об указателях, ссылающихся на один массив. В противном случае результат арифметических операций и операций отношения будет не определен.

74. Указатели. Применение к указателям оператора sizeof. Сложности при использовании операций ++ и --. Сравнение указателей. Переносимость указателей. Использование функции sizeof() с указателями в среде DOS.

Указатель — переменная, содержащая адрес объекта. Указатель не несет информации о содержимом объекта, а содержит сведения о том, где размещен объект.

Память компьютера можно представить в виде последовательности пронумерованных однобайтовых ячеек, с которыми можно работать по отдельности или блоками.

Каждая переменная в памяти имеет свой адрес - номер первой ячейки, где она расположена, а также свое значение. Указатель — это тоже переменная, которая размещается в памяти. Она тоже имеет адрес, а ее значение является адресом некоторой другой переменной. Переменная, объявленная как указатель, занимает 4 байта в оперативной памяти (в случае 32-битной версии компилятора).

Указатель, как и любая переменная, должен быть объявлен.
Общая форма объявления указателя

тип *ИмяОбъекта;

Тип указателя— это тип переменной, адрес которой он содержит.

Для работы с указателями в Си определены две операции:

операция * (звездочка) — позволяет получить значение объекта по его адресу - определяет значение переменной, которое содержится по адресу, содержащемуся в указателе;

операция & (амперсанд) — позволяет определить адрес переменной.

Например,

```
char c;      // переменная
char *p;     // указатель
p = &c;      // p = адрес c
```

Оператор **sizeof** возвращает размер в байтах объекта или типа данных. Синтаксис его таков:

```
sizeof ( type name );
sizeof ( object );
sizeof object;
```

Результат имеет специальный тип `size_t`, который определен как `typedef` в заголовочном файле `cstdint`.

Применение `sizeof` к указателю дает размер самого указателя, а не объекта, на который он указывает:

```
int *pi = new int[ 3 ];
size_t pointer_size = sizeof ( pi );
```

Здесь значением `pointer_size` будет память под указатель в байтах (4 в 32-битных системах), а не массива `ia`.

Вот пример программы, использующей оператор `sizeof`:

```
#include string
#include iostream
#include cstdint
int main() {
    size_t ia;
    ia = sizeof( ia ); // правильно
    ia = sizeof ia;   // правильно
```

```

// ia = sizeof int; // ошибка
ia = sizeof( int ); // правильно
int *pi = new int[ 12 ];
cout << "pi: " << sizeof( pi )
    << " *pi: " << sizeof( *pi )
    << endl;
// sizeof строки не зависит от ее реальной длины

string st1( "foobar" );
string st2( "a mighty oak" );
string *ps = st1;
cout << " st1: " << sizeof( st1 )
    << " st2: " << sizeof( st2 )
    << " ps: sizeof( ps )
    << " *ps: " << sizeof( *ps )
    << endl;
cout << "short : " << sizeof(short) << endl;
cout << "shortf" : " << sizeof(short*) << endl;
cout << "short : " << sizeof(short) << endl;
cout << "short[3] : " << sizeof(short[3]) << endl;
}

```

Результатом работы программы будет:

```

pi: 4 *pi: 4
st1: 12 st2: 12 ps: 4 *ps:12
short : 2
short* : 4
short : 2
short[3] : 6

```

Из данного примера видно, что применение `sizeof` к указателю позволяет узнать размер памяти, необходимой для хранения адреса.

Над адресами в C++ определены следующие арифметические операции:

- сложение и вычитание указателей с константой;
- вычитание одного указателя из другого;
- инкремент;
- декремент.

Инкремент перемещает указатель к следующему элементу массива, а **декремент** – к предыдущему;

К указателям могут применяться только две арифметические операции: сложение и вычитание. Для понимания арифметических действий с указателями предположим, что `p1` - это указатель на целое, содержащий значение 2000, и будем считать, что целые имеют длину 2 байта. После выражения

`p1 ++;`

содержимое `p1` станет 2002, а не 2001! Каждый раз при увеличении `p1` указатель будет указывать на следующее целое. Это справедливо и для уменьшения. Например:

`p1 --;`

приведет к тому, что `p1` получит значение 1998, если считать, что раньше было 2000.

Каждый раз, когда указатель увеличивается, он указывает на следующий элемент базового типа. Каждый раз, когда уменьшается - на предыдущий элемент. В случае указателей на символы это приводит к «обычной» арифметике. Все остальные указатели увеличиваются или уменьшаются на длину базового типа. Рис. демонстрирует данную концепцию.

Естественно, все не ограничивается только уменьшением или увеличением. Можно добавлять или вычитать из указателей целые числа. Выражение

`p1 = p1 + 9;`

приводит к тому, что указатель `p1` указывает на девятый элемент по сравнению с элементом, на который он указывал до присваивания.

Помимо добавления или вычитания указателей и целых чисел, единственную операцию, которую можно выполнять с указателями, - это вычитание одного указателя из другого.

В большинстве случаев вычитание одного указателя из другого имеет смысл только тогда, когда оба указателя указывают на один объект, - как правило, массив. В результате вычитания получается число элементов базового типа, находящихся между указателями. Помимо этих операций не существует других арифметических операций, применимых к указателям. Нельзя умножать или делить указатели, нельзя складывать указатели, нельзя применять битовый сдвиг или маски к указателям, нельзя добавлять или вычитать типы `float` или `double`.

К указателям так же применимы *операции отношения* `==`, `!=`, `<`, `>`, `<=`, `>=`. Иными словами, указатели можно сравнивать. Например, если `i` указывает на пятый элемент массива, а `j` — на первый, то отношение `i > j` истинно. Кроме того, любой указатель можно сравнивать на равенство с нулем. Однако, все эти утверждения верны, если речь идет об указателях, ссылающихся на один массив. В противном случае результат арифметических операций и операций отношения будет не определен.

Переносимость указателей.

В приведенных примерах адреса представлялись целыми числами. Вы можете предположить, что в C указатели имеют тип `int`. Это не так. Указатель содержит адрес переменной некоторого типа, однако сам указатель не относится ни какому простому типу данных вроде `int`, `float` и им подобному. В конкретной системе C указатель может копироваться в переменную типа `int`, а переменная `int` может копироваться в указатель; однако, язык C не гарантирует, что указатели могут храниться в переменных типа `int`. Для обеспечения переносимости программного кода таких операций следует избегать.

Кроме того, разрешены не все арифметические операции с указателями. Например, запрещается складывать или перемножать два указателя, или делить один указатель на другой.

В этой программе на C++ печатается размер указателей по умолчанию, и их размер с моделями памяти `__far` и `near`. Также используется директива препроцессора (`#`), определяющая `PRINTF_SIZEOF` с аргументом `A_POINTER`, поэтому печатается не только размер указателя, но и его название.

```

//10STRIZE.CPP
//sizeof с указателями
#include <stdio.h>
#define PRINT_SIZEOF(A_POINTER) \
printf("sizeof\t("#A_POINTER")\t= %d\n", \
sizeof(A_POINTER))
main()
{
char *reg_pc;
long double *reg_pldbl;
char __far *far_pc;
long double __far *far_pldbl;
char __near *near_pc;
long double __near *near_pldbl;
PRINT_SIZEOF(reg_pc);
PRINT_SIZEOF(reg_pldbl);
PRINT_SIZEOF(far_pc);
PRINT_SIZEOF(far_pldbl);
PRINT_SIZEOF(near_pc);
PRINT_SIZEOF(near_pldbl);
return (0);
}

```

Результат выполнения программы:

Sizeof	(reg_pc)	= 2
Sizeof	(reg_pldbl)	= 2
Sizeof	(far_pc)	= 4
Sizeof	(far_pldbl)	= 4
Sizeof	(near_pc)	= 2
Sizeof	(near_pldbl)	= 2

75. Указатели. Указатели на функции. Динамическая память. Использование указателей типа void.

Указатель — переменная, содержащая адрес объекта. Указатель не несет информации о содержимом объекта, а содержит сведения о том, где размещен объект.

Память компьютера можно представить в виде последовательности пронумерованных однобайтовых ячеек, с которыми можно работать по отдельности или блоками.

Каждая переменная в памяти имеет свой адрес - номер первой ячейки, где она расположена, а также свое значение. Указатель — это тоже переменная, которая размещается в памяти. Она тоже имеет адрес, а ее значение является адресом некоторой другой переменной. Переменная, объявленная как указатель, занимает 4 байта в оперативной памяти (в случае 32-битной версии компилятора).

Указатель, как и любая переменная, должен быть объявлен.
Общая форма объявления указателя

тип *ИмяОбъекта;

Тип указателя— это тип переменной, адрес которой он содержит.

Для работы с указателями в Си определены две операции:

операция * (звездочка) — позволяет получить значение объекта по его адресу - определяет значение переменной, которое содержится по адресу, содержащемуся в указателе;

операция & (амперсанд) — позволяет определить адрес переменной.

Например,

<code>char c;</code>	<code>// переменная</code>
<code>char *p;</code>	<code>// указатель</code>
<code>p = &c;</code>	<code>// p = адрес c</code>

Указатель на функцию можно использовать в нескольких важных случаях. Рассмотрим, например, функцию `qsort()`. Одним из ее параметров является указатель на функцию. Адресуемая функция осуществляет необходимое сравнение, которое должно выполняться для элементов сортируемого массива. Применение в `qsort()` указателя на функцию вызвано тем, что процесс сравнения двух элементов может быть весьма сложным, и при помощи одного управляющего флага его представить нельзя. Функцию нельзя передать по значению — то есть, передать ее код. Однако, в С можно передать указатель на код или указатель на функцию.

Следующая ниже программа на С показывает, как описать указатель на функцию и как передать пользовательскую функцию в функцию `qsort()`, объявленную в файле `stdlib.h`.

Вот программа на С:

```
/*10FNCPTR.C
```

```
Программа на C, иллюстрирующая объявление пользовательской  
функции*/
```

```
#include "stdafx.h"
```

```
#include "E:\LECTURE\AlgorithmProgramming 02\Universal_HederFile.h"
```

```
#define IMAXVALUES 10
```

```
int icompare_func(const void *iresult_a, const void *iresult_b);
```

```
int (*ifunct_ptr)(const void *,const void *);
```

```
void StopWait(void);
```

```
main()
```

```
{
```

```
int i;
```

```
int iarray[IMAXVALUES]={0,5,3,2,8,7,9,1,4,6};
```

```
ifunct_ptr=icompare_func;
```

```
qsort(iarray,IMAXVALUES, sizeof(int), ifunct_ptr);
```

```
for(i=0; i < IMAXVALUES; i++)
```

```
printf("%d ",iarray[i]);
```

```
printf("\n");
```

```
StopWait(); /* Wait a little */
```

```
return (0);
```

```
}
```

```
int icompare_func(const void *iresult_a, const void *iresult_b)
```

```
{
```

```
return((*(int *)iresult_a)-(*(int *) iresult_b));
```

```
}
```

Динамическая память.

При компиляции программы на C память компьютера разбивается на четыре области, содержащие код программы, все глобальные данные, стек и динамически распределяемую область памяти (иногда ее называют heap-куча). Динамическая память (heap) — это просто

свободная область памяти, с которой работают при помощи функций динамического выделения памяти `malloc()` и `free()`.

При вызове функции `malloc()` выделяется непрерывный блок памяти для указанного объекта и возвращается указатель на начало этого блока. Функция `free()` возвращает выделенный блок обратно в динамическую область памяти для повторного использования.

Аргумент, передаваемый функции `malloc()`, представляет собой объем необходимой памяти в байтах. В следующем фрагменте кода выделяется память для 300 чисел типа `float`:

```
float *pf;  
int inum_floats = 300;  
pf = (float *) malloc(inum_floats *  
sizeof(float));
```

Когда блок становится ненужным, его можно вернуть операционной системе, используя, следующий оператор:

```
free((void *) pf);
```

Программы на С и С++ могут в любой момент выделять и освобождать динамическую память. Важно помнить, что на переменные, память для которых выделена в динамической области, не распространяются правила области действия, которые справедливы для других переменных. Эти переменные действуют всегда; поэтому, если вы выделили динамическую область памяти, то не должны забывать о ее освобождении. Если повторно выделить динамическую память, предварительно не освободив ее, то выполнение программы, скорее всего закончится крахом.

Для распределения динамической памяти в большинстве компиляторов С используются описанные выше библиотечные функции `malloc()` и `free()`, однако, в С++ эти средства считаются настолько важными, что они включены в ядро языка. В С++ для выделения и освобождения динамической памяти используются операторы *new* и *delete*.

Использование указателей типа void.

В Си существует особый тип указателей – указатели типа `void` или пустые указатели. Эти указатели используются в том случае, когда тип переменной не известен. Так как `void` не имеет типа, то к нему не применима операция разадресации (взятие содержимого) и адресная арифметика, так как неизвестно представление данных. Тем не менее, если мы работаем с указателем типа `void`, то нам доступны операции сравнения.

Если необходимо работать с пустым указателем, то сначала нужно явно привести тип. Например

```

1  #include <conio.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <string.h>
5
6  void swap(void *a, void *b, size_t size) {
7      char* tmp;
8      //создаём временную переменную для обмена
9      tmp = (char*) malloc(size);
10     memcpy(tmp, a, size);
11     memcpy(a, b, size);
12     memcpy(b, tmp, size);
13     free(tmp);
14 }
15
16 int main() {
17     float a = 10.f;
18     float b = 20.f;
19     double c = 555;
20     double d = 777;
21     unsigned long e = 211;
22     unsigned long f = 311;
23
24     printf("a = %.3f, b = %.3f\n", a, b);
25     swap(&a, &b, sizeof(float));
26     printf("a = %.3f, b = %.3f\n", a, b);
27
28     printf("c = %.3f, d = %.3f\n", c, d);
29     swap(&c, &d, sizeof(double));
30     printf("c = %.3f, d = %.3f\n", c, d);
31
32     printf("e = %ld, f = %ld \n", e, f);
33     swap(&e, &f, sizeof(unsigned long));
34     printf("e = %ld, f = %ld \n", e, f);
35
36     getch();
37 }

```

Переменная не может иметь типа void, этот тип определён только для указателей. Пустые указатели нашли широкое применение при вызове функций. Можно написать функцию общего назначения, которая будет работать с любым типом. Например, напомним функцию swap, которая обменивает местами содержимое двух переменных. У этой функции будет три аргумента – указатели на переменные, которые необходимо обменять местами и их размер.

Наша функция может выглядеть и по-другому. Обойдёмся без дорогостоящего выделения памяти и будем копировать побайтно.

```

6  void swap(void *a, void *b, size_t size) {
7      char tmp;
8      size_t i;
9      for (i = 0; i < size; i++) {
10         tmp = *((char*) b + i);
11         *((char*) b + i) = *((char*) a + i);
12         *((char*) a + i) = tmp;
13     }
14 }

```


Пустые указатели позволяют создавать функции, которые возвращают и принимают одинаковые параметры, но имеют разное название. Это пригодится в дальнейшем, при изучении указателей на функции. Например

```
1  int cmpInt(void* a, void* b) {
2      return *((int*) a) - *((int*) b);
3  }
4
5  int cmpString(void* a, void* b) {
6      return strcmp((char*) a, (char*) b);
7  }
8
9  int cmpFloat(void* a, void* b) {
10     float fdiff = *((float*) a) - *((float*) b);
11     if (fabs(fdiff) < 0.000001f) {
12         return 0;
13     }
14     if (fdiff < 0) {
15         return -1;
16     } else {
17         return 1;
18     }
19 }
```

Объявление указателя на неопределенный тип:

`void *ptr;`

Такому указателю может быть присвоен указатель на любой тип, но не наоборот

```
void *ptr;      // Указатель на void
int i, *ptri;   // Целая переменная, указатель на int
ptr = &i;       // Допустимо
ptr = ptri;     // Допустимо
ptri = (int)ptr; // Допустимо
               // ptri=ptr;    // Недопустимо
```

Для последней операции необходимо явное приведение типа.

Над указателем неопределенного типа нельзя выполнять операцию разыменования без явного приведения типа.

76. Указатели. Указатели и массивы. Функции, массивы и указатели. Использование указателей при работе с массивами. Строки (массивы типа `char`).

Указатель — переменная, содержащая адрес объекта. Указатель не несет информации о содержимом объекта, а содержит сведения о том, где размещен объект.

Память компьютера можно представить в виде последовательности пронумерованных однобайтовых ячеек, с которыми можно работать по отдельности или блоками.

Каждая переменная в памяти имеет свой адрес - номер первой ячейки, где она расположена, а также свое значение. Указатель — это тоже переменная, которая размещается в памяти. Она тоже имеет адрес, а ее значение является адресом некоторой другой переменной. Переменная, объявленная как указатель, занимает 4 байта в оперативной памяти (в случае 32-битной версии компилятора).

Указатель, как и любая переменная, должен быть объявлен.
Общая форма объявления указателя

тип *ИмяОбъекта;

Тип указателя— это тип переменной, адрес которой он содержит.

Для работы с указателями в Си определены две операции:

операция * (звездочка) — позволяет получить значение объекта по его адресу - определяет значение переменной, которое содержится по адресу, содержащемуся в указателе;

операция & (амперсанд) — позволяет определить адрес переменной.

Например,

```
char c;      // переменная
char *p;     // указатель
p = &c;      // p = адрес c
```

Указатели и массивы.

В следующих разделах приводится множество примеров, в которых затрагиваются концепция массивов и их связь с указателями.

Указатели позволяют нам работать с символическими адресами. Поскольку в реализуемых аппаратно командах вычислительной машины интенсивно используются адреса, указатели предоставляют возможность применять адреса примерно так, как это делается в самой машине, и тем самым повышать эффективность программ. В частности, указатели позволяют эффективно организовать работу с массивами. Действительно, как мы могли убедиться, наше обозначение массива представляет собой просто скрытую форму использования указателей.

Посмотрите, что происходит со значением указателя, если к нему прибавить число.

```

/* прибавление к указателю */
#include<stdio.h>
main ()
{
int dates[4], *pti, index;
float bills [4], *ptf;
pti = dates; /* присваивает адрес указателю массива */
ptf = bills;
for (index = 0; index < 4; index++ )
printf(" ukazatel + %d: %10u %10u\n", index, pti + index, ptf + index);
}

```

Результат работы программы

```

C:\Z:\000 0000000000000000\1\Debug\1.exe
ukazatel + 0: u 1245040
ukazatel + 1: u 1245044
ukazatel + 2: u 1245048
ukazatel + 3: u 1245052
Press any key to continue

```

Вот результат

указатели + 0: 56014 56026

указатели + 1: 56016 56030

указатели + 2: 56018 56034

указатели + 3: 56020 56038

Первая напечатанная строка содержит начальные адреса двух массивов, а следующая строка — результат прибавления единицы к адресу и т. д. Почему так получается?

56014 + 1 = 56016?

56026 + 1 = 56030?

В нашей системе единицей адресации является байт, но тип `int` использует два байта, а тип `float` — четыре. Что произойдет, если вы скажете: «прибавить единицу к указателю?»
Компилятор языка Си добавит единицу памяти. Для массивов это означает, что мы перейдем к адресу следующего элемента, а не следующего байта. Вот почему мы должны специально

оговаривать тип объекта, на который ссылается указатель; одного адреса здесь недостаточно, так как машина должна знать, сколько байтов потребуется для запоминания объекта. (Это справедливо также для указателей на скалярные переменные; иными словами, при помощи операции `*pt` нельзя получить значение.)

Функции, массивы и указатели.

Массивы можно использовать в программе двояко. Во-первых, их можно описать в теле функции. Во-вторых, они могут быть аргументами функции. Все, что было сказано в этой главе о массивах, относится к первому их применению; теперь рассмотрим массивы в качестве аргументов.

Об этом уже говорилось в гл. 10. Сейчас, когда мы познакомились с указателями, можно заняться более глубоким изучением массивов-аргументов. Давайте проанализируем скелет программы, обращая внимание на описания:

```
/* массив-аргумент */
main ()
{
    int ages [50]; /* массив из 50
    элементов */
    convert(ages);
    ...
}

convert (years)
int years []; /* каков размер
массива? */
{
    ...
}
```

Очевидно, что массив `ages` состоит из 50 элементов. А что можно сказать о массиве `years`? Оказывается, в программе нет такого массива. Описатель `int years[];` создает не массив, а указатель на него.

Вот вызов нашей функции:

```
convert(ages);
```

`ages` — аргумент функции `convert`. Вы помните, что имя `ages` является указателем на первый элемент массива, состоящего из 50 элементов. Таким образом, оператор вызова функции передает ей указатель, т. е. адрес функции `convert ()`. Это значит, что аргумент функции является указателем, и мы можем написать функцию `convert ()` следующим образом:

```
convert (years)
```

Действительно, операторы

```
int years [];
int *years;
```

синонимы. Оба они объявляют переменную years указателем массива целых чисел. Однако главное их отличие состоит в том, что первый из них напоминает нам, что указатель years ссылается на массив.

Использование указателей при работе с массивами.

Попробуем написать функцию, использующую массивы, а затем перепишем ее, применяя указатели.

Рассмотрим простую функцию, которая находит (или пытается найти) среднее значение массива целых чисел. На входе функции мы имеем имя массива и количество элементов. На выходе получаем среднее значение, которое передается при помощи оператора return. Оператор вызова функции может выглядеть следующим образом:

```
/* X54.C */
#include <stdio.h>
void main(void)
{
    int n1=9,i, *pa1,*pa,n;
    int array[] = {9,5,3,1,0,-9,-5,-3,-1};
    pa1=array;
    int mean(int *pa,int n);
    i = mean(pa1,n1);
    printf("Average is %d\n",i);
}
int mean(int *pa,int n)
{
    int index;
    long sum;
    if(n > 0)
    {
        for(index = 0, sum = 0; index < n; index++)
            sum += *(pa+index);
        return((int) (sum/n));
    }
    else
    {
        printf("Net massiva\n");
        return(0);
    }
}
```

Результат работы программы



Эту программу легко переделать, применяя указатели. Объявим pa указателем на тип int. Затем заменим элемент массива array[index] на соответствующее значение: *(pa + index).

```

/* Использование указателей для нахождения среднего значения
массива n целых чисел */
int mean(pa, n)
int *pa, n;
{
    int index;
    long sum; /*Если целых слишком много, их можно суммировать в формате
long int */
    if (n > 0)
    {
        for (index = 0, sum = 0; index < n; index++)
            sum += *(pa + index);
        return((int) (sum/n) ); /* Возвращает целое */
    }
    else
    {
        printf(" Нет массива. \n");
        return(0);
    }
}

```

Строки (массивы типа char).

Многие строковые операции в С обычно выполняются с использованием указателей и арифметических операций с указателями для доступа к элементам символьного массива. Это обусловлено тем, что символьные массивы или строки, как правило, обрабатываются строго последовательно. Напоминаем, что все строки в С заканчиваются символом null (\0). Следующая программа на С++ иллюстрирует использование указателей с символьными массивами:

```

// 10CHRRARY.CPP
// Программа на С++, печатающая массив символов в обратном порядке
// и использующая указатель на символ и оператор декремента

#include "stdafx.h"
#include "E:\LECTURE\AlgorithmProgramming 02\Universal_HederFile.h"

void StopWait(void);

void main()
{
    char pszpalindrome[]="POOR DAN IN A DROOP";
    char *pc;
    pc=&pszpalindrome[0]+(strlen(pszpalindrome)-1);
    do
    {
        cout << *pc;
        pc--;
    } while (pc >= pszpalindrome);
    cout << endl;

    StopWait(); /* Wait a little */
}

```

ПРИМЕЧАНИЕ.

Функция `strlen()` считает только символы. Она не включает в результат `null` - символ `\0`

77. Указатели. Массивы указателей. Дополнительная информация об указателях на указатели. Массивы указателей на строки.

Указатель — переменная, содержащая адрес объекта. Указатель не несет информации о содержимом объекта, а содержит сведения о том, где размещен объект.

Память компьютера можно представить в виде последовательности пронумерованных однобайтовых ячеек, с которыми можно работать по отдельности или блоками.

Каждая переменная в памяти имеет свой адрес - номер первой ячейки, где она расположена, а также свое значение. Указатель — это тоже переменная, которая размещается в памяти. Она тоже имеет адрес, а ее значение является адресом некоторой другой переменной. Переменная, объявленная как указатель, занимает 4 байта в оперативной памяти (в случае 32-битной версии компилятора).

Указатель, как и любая переменная, должен быть объявлен.
Общая форма объявления указателя

тип *ИмяОбъекта;

Тип указателя— это тип переменной, адрес которой он содержит.

Для работы с указателями в Си определены две операции:

операция `*` (звездочка) — позволяет получить значение объекта по его адресу - определяет значение переменной, которое содержится по адресу, содержащемуся в указателе;

операция `&` (амперсанд) — позволяет определить адрес переменной.

Например,

```
char c;      // переменная
char *p;     // указатель
p = &c;      // p = адрес c
```

Массивы указателей.

В языках C и C++ можно создавать не только простые массивы и указатели. Их можно объединить в очень полезную конструкцию — массив указателей. Массив указателей — это такой массив, элементы которого являются указателями на другие объекты. Эти объекты, в свою очередь, могут быть указателями. Это означает, что можно иметь массив указателей, которые ссылаются на другие указатели.

Можно создавать массивы указателей. Для объявления массива целочисленных указателей из десяти элементов следует написать:

```
int *x[10];
```

Для присвоения адреса целочисленной переменной var третьему элементу массива следует написать:

```
x[2] = &var;
```

Для получения значения var следует написать:

```
*x [2]
```

Если необходимо передать массив указателей в функцию, можно использовать метод, аналогичный передаче обычных массивов. Просто надо вызвать функцию с именем массива без индексов. Например, функция, получающая массив x, должна выглядеть следующим образом:

```
void display_array(int *q[])
{
    int t;
    for(t=0; t<10; t++)
        printf ("%d ", *q[t]);
}
```

Надо помнить, что q - это не указатель на целое, а массив указателей на целые. Следовательно, необходимо объявить параметр q как массив целых указателей. Он не может объявиться как простой целочисленный указатель, поскольку он не является им.

Типичным использованием массивов указателей является хранение сообщений об ошибках. Можно создать функцию, выводящую сообщение по полученному номеру, как показано ниже:

```
void serror(int num)
{
    static char *err[] = {
        "Cannot Open File\n",
        "Read Error\n",
        "Write Error\n",
        "Media Failure\n"
    };
    printf ("%s", err[num]);
}
```

Как можно видеть, printf() в serror() вызывается с указателем на символ, указывающим на одно из сообщений, номер которого передается в функцию. Например, если num приняла значение 2, будет выведено сообщение «Write Error».

Интересно заметить, что аргумент командной строки argv является массивом указателей на символы.

78. Указатели. Ссылочный тип в C++ (reference type). Адрес в качестве возвращаемого значения функции. Передача параметров по ссылке и по значению. Использование встроенного отладчика. Использование ссылочного типа. Использование указателей и ссылок с ключевым

Указатель — переменная, содержащая адрес объекта. Указатель не несет информации о содержимом объекта, а содержит сведения о том, где размещен объект.

Память компьютера можно представить в виде последовательности пронумерованных однобайтовых ячеек, с которыми можно работать по отдельности или блоками.

Каждая переменная в памяти имеет свой адрес - номер первой ячейки, где она расположена, а также свое значение. Указатель — это тоже переменная, которая размещается в памяти. Она тоже имеет адрес, а ее значение является адресом некоторой другой переменной. Переменная, объявленная как указатель, занимает 4 байта в оперативной памяти (в случае 32-битной версии компилятора).

Указатель, как и любая переменная, должен быть объявлен.
Общая форма объявления указателя

тип *ИмяОбъекта;

Тип указателя— это тип переменной, адрес которой он содержит.

Для работы с указателями в Си определены две операции:

операция * (звездочка) — позволяет получить значение объекта по его адресу - определяет значение переменной, которое содержится по адресу, содержащемуся в указателе;

операция & (амперсанд) — позволяет определить адрес переменной.

Например,

```
char c;      // переменная
char *p;     // указатель
p = &c;      // p = адрес c
```

Ссылочный тип в C++ (reference type).

В языке C++ имеется способ вызова по ссылке, который использовать еще проще, чем указатели. Для начала рассмотрим использование ссылочных переменных в C++. Так же, как и в C, в языке C++ можно объявить обычные переменные или переменные-указатели. В первом случае для данных действительно выделяется память; во втором случае память резервируется для адреса объекта, который будет создан в другое время. В C++ имеется третий тип объявлений — ссылки. Так же, как указатель, ссылочная переменная указывает на положение другой переменной, однако, так же как и в случае обычной переменной, для ссылки не требуется специальной операции разыменования. Синтаксис ссылочной переменной понятен:

```
int ireult_a=5;
```

```
int& rresult_a=iresult_a; // правильно
```

```
int& rresult b; // неправильно: нет начального значения
```

В этом примере определяется ссылочная переменная `rresult_a` и присваивается существующей переменной `iresult_a`. После этого адресуемое значение имеет два имени — `iresult_a` и `rresult_a`. Поскольку обе переменные указывают на одну и ту же ячейку памяти, они представляют собой, по сути, одну переменную. Любое присваивание, сделанное по отношению к `rresult_a`, отражается на `iresult_a`; справедливо и обратное утверждение. Следовательно, при помощи ссылки можно создать нечто подобное псевдониму (alias) переменной.

Ссылки имеют ограничение, позволяющее отличать их от указателей, которые ведут себя очень похоже. Значение ссылочного типа должно быть задано при объявлении и не может меняться в процессе выполнения программы. После инициализации типа при объявлении, ссылка всегда указывает на одну и ту же ячейку памяти. Следовательно, при любом присваивании значения ссылочной переменной изменяются только данные в памяти, а не адрес самой переменной. Другими словами, ссылку можно считать указателем на ячейку памяти с постоянным адресом.

Адрес в качестве возвращаемого значения функции.

Когда функция возвращает адрес либо в виде переменной-указателя, либо в виде ссылки, пользователь получает некоторый адрес в памяти. Он может считать значение, находящееся по этому адресу, и, если тип указателя не объявлен как `const`, всегда может что-нибудь туда записать. Если функция возвращает адрес, то пользователь получает возможность читать и — в случае указателей, имеющих тип, отличный от `const` — обновлять локальные данные. Это важно для проектирования приложений.

Передача параметров функции по ссылке и по значению

В функцию параметры могут передаваться как по значению, так и по ссылке.

При передаче параметров по значению они при выходе из функции не изменятся. Например, следующий фрагмент напечатает на экране 1 и 3:

```
...
int func(int k) {
    k* = 2;
    return k;
}
void main(){
    int z = 1, y = 3, k;
    k = func(z) + func(y);
    cout << z <<" " << y;
    ...
}
```

При передаче же параметров по ссылке при выходе из функции их значения могут измениться. Как, например, в следующем фрагменте:

```
...
int func(int &k) {
    k* = 2;
    return k;
}
void main(){
    int z = 1, y = 3, k;
    k = func(z) + func(y);
    ...
}
```

```
cout<<z<<" "<<y;  
...
```

После его выполнения на экране напечатается 2 и 6.

Разница между двумя этими фрагментами весьма невелика - всего 1 символ амперсанда (&) в строке

```
...  
int func(int &k ){  
...
```

Еще один классический пример на эту тему - это функция, которая меняет значения своих параметров:

```
void func(int &n, int &m) {  
    int tmp = n;  
    n = m;  
    m = tmp;  
}
```

После выполнения такой функции значения её параметров поменяются местами.

Использование встроенного отладчика.

Чтобы увидеть работу этой программы на C++ в реальных условиях, можно воспользоваться встроенным отладчиком. Проследим при помощи окна Trace изменения переменной `pi`.

Что происходит? При вызове функции `ifirst_function()` для переменной `ilocal_to_first` в стеке выделяется локальная область памяти, в которую записывается число 11. После этого функция `ifirst_function()` возвращает адрес этой локальной переменной (очень плохая новость!). Во второй строке функции `mata()` вызывается функция `isecond_function()`, которая в свою очередь выделяет локальную область памяти для переменной `ilocal_to_second` и записывает туда число 44. Почему же оператор `printf` печатает значение 44, хотя ему при вызове функции `inrst_fiinction()` был передан адрес переменной `ilocaljojirsf`?

Фактически происходит следующее. Когда при вызове функции `ifirst_function()` адрес временной локальной переменной `ilocal_to_first` был присвоен переменной `pi`, то этот адрес был сохранен, несмотря на завершение области действия `ilocal_to_first`. При вызове функции `isecond_function()`, ей также понадобилась локальная память. Поскольку переменной `ilocal_to_first` не стало, переменная `ilocal_to_second` получила ту же локальную область, что и ее предшественница. Так как `pi` указывает на ту же занятую ячейку памяти, то становится понятным, почему при печати адресуемой ячейки появляется число 44. Следует быть чрезвычайно внимательным и стараться не возвращать адреса локальных переменных.

Использование ссылочного типа.

Подводя черту, можно выделить четыре основных причины использования ссылок C++:

- Ссылки упрощают восприятие программы, поскольку можно игнорировать детали процесса передачи параметра.
- Ссылки передают ответственность за передачу аргумента тому программисту, который пишет функцию, а не тому, который ею пользуется.
- Ссылки являются необходимой составной частью перегрузки операций.

- Ссылки используются при передаче функциям классов, при этом конструкторы и деструкторы не вызываются.

Ссылка представляет собой синоним имени, указанного при инициализации ссылки. Ссылку можно рассматривать как указатель, который всегда разыменовывается. Формат объявления ссылки:

тип & имя;

где тип — это тип величины, на которую указывает ссылка, & — оператор ссылки, означающий, что следующее за ним имя является именем переменной ссылочного типа, например:

```
int kol;  
int& pal = kol; // ссылка pal - альтернативное имя для kol  
const char& CR = '\n '; // ссылка на константу
```

Запомните следующие правила.

- Переменная-ссылка должна явно инициализироваться при ее описании, кроме случаев, когда она является параметром функции, описана как extern или ссылается на поле данных класса.
- После инициализации ссылке не может быть присвоена другая переменная.
- Тип ссылки должен совпадать с типом величины, на которую она ссылается.
- Не разрешается определять указатели на ссылки, создавать массивы ссылок и ссылки на ссылки.

Ссылки применяются чаще всего в качестве параметров функций и типов возвращаемых функциями значений. Ссылки позволяют использовать в функциях переменные, передаваемые по адресу, без операции разадресации, что улучшает читаемость программы.

Ссылка, в отличие от указателя, не занимает дополнительного пространства в памяти и является просто другим именем величины. Операция над ссылкой приводит к изменению величины, на которую она ссылается.

Использование указателей и ссылок с ключевым словом const

Некоторые конструкции языка C++ являются источником путаницы. Одной из таких конструкций является использование ключевого слова const с указателями и ссылками. Следующие примеры помогут вам прояснить ситуацию.

// Объявление данных

```
int number;
```

```
const int count=0;
```

// Указатель является константой

```
int* const n1=unumber;
```

// Указатель указывает на константу

// (указываемое значение есть const)

```
const int* n2=&count;
```

```

// И указатель, и указываемое значение

// являются константами

const int* const n3=&count;

// Указатели на строки

// строка является константной

const char* str1="text";

// Указатель на строку является константой

char* const str2="text";

// Указатель и сама строка – константы

const char* const str3="text";

/* Массивы указателей на символы */

// Символы являются константами

const char* text1[]={ "lne1", "lne2", "lne3" };

// Указатели являются константами

char* const text2[]={ "lne1", "lne2", "lne3" };

// Указатели и символы являются константами

const char* const text3={ "st1", "st2", "st3" };

```

79. Использование функций. Создание и использование простой функции. Прототипы функций. Вызов по значению и вызов по ссылке. Использование указателей для связи между функциями

Объявление функций: прототипы функций

```
1    returnDataType functionName( dataType argName1, dataType argName2, ..., dataType argNameN);
```

где,

returnDataType — возвращаемый тип данных

functionName — имя функции

dataType — тип данных

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

Смотрим пример объявления функции:

```
1 // Объявление прототипа функции с двумя целыми параметрами
2 // функция принимает два аргумента и возвращает их сумму
3 int sum(int num1, int num2);
```

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

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

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

Определение функций

```
1 returnDataType functionName( dataType argName1, dataType argName2, ..., dataType argNameN)
2 {
3     // тело функции
4 }
```

Рассмотрим определение функции на примере функции sum.

```
1 // определение функции, которая суммирует два целых числа и возвращает их сумму
2 int sum(int num1, int num2)
3 {
4     return (num1 + num2);
5 }
```

В языках C и C++, функции не должны быть определены до момента их использования, но они должны быть ранее объявлены. Но даже после всего этого, в конце концов, эта функция должна быть определена. После этого прототип функции и ее определение связываются, и эта функция может быть использована.

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

```
1 // объявление функции суммирования
2 int sum(int, int);
3
4 // определение функции суммирования
5 int sum(int num1, int num2)
6 {
7     return (num1 + num2);
8 }
```

Согласно стандарту ANSI C все функции должны иметь прототипы. Прототипы могут располагаться либо в самой программе на C или C++, либо в заголовочном файле. Большинство прототипов функций находятся в самих программах. Объявление функции в C и C++ начинается с ее прототипа. Прототип функции достаточно прост; обычно он включается в начало программы для того, чтобы сообщить компилятору тип и количество аргументов, используемых некоторой функцией. Использование прототипов обеспечивает более строгую проверку типов по сравнению с той, которая была при прежних стандартах C.

Хотя допустимы и другие стили записи прототипов функций, все же рекомендуется использовать, по возможности, следующий стиль: повторение строки объявления функции с добавлением в конце точки с запятой. Например:

возвращаемый_тип имя_функции(тип_аргумента(-ов)) (имя_аргумента(-ов));

Функция может иметь тип void, int, float и так далее и определяется возвращаемым типом. Имя_функции() — это любое значимое наименование, выбранное вами для определения этой функции.

Вызов функций

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

```
1 funcName( arg1, arg2, ... );
```

где,

funcName — имя функции;

arg1..2 — аргументы функции (значения или переменные)

Примечание: функции могут не иметь параметров, тогда в круглых скобках ничего писать не надо.

Смотрим пример:

```
1 // вызов функции синуса
2 sin( 60 );
```

Вызов функции выполняется записью её имени, а затем круглых скобочек (). Если функция принимает аргументы, то в круглых скобках передаются аргументы, в порядке, указанном в объявлении функции. Подробно про функции, читайте статью: [Функции в C++](#).

Передача параметров функции

Локальные и глобальные переменные

Внутри функции могут быть объявлены переменные, как, например, переменные `r` и `i` в функции `power`. Такие переменные называются *локальными* и они определены только внутри этой функции. Переменные, определенные вне тела любой функции (ранее мы такие переменные вообще не рассматривали) называются *глобальными* и они определены всюду начиная с места определения и до конца файла. Глобальными переменными можно пользоваться во всех функциях, но с современной точки зрения использовать глобальные переменные рекомендуется как можно реже.

Локальные переменные создаются каждый раз при входе в функцию и уничтожаются при выходе из нее. Таким образом, значения, сохраненные в локальных переменных, пропадут после завершения работы функции.

В различных функциях можно определять переменные с одним и тем же именем. Это будут различные переменные. Также можно объявлять в функциях переменные, имена которых совпадают с именами глобальных переменных. Тогда будет создана новая локальная переменная, а изменить значение глобальной переменной с таким именем будет невозможно. Пример:

```
#include<iostream>
using namespace std;
int i;          // i - глобальная переменная
void f1() {
    int i;      // Определена локальная переменная i
    i=5;        // Изменяется значение локальной переменной
    cout<<i<<endl; // Будет напечатано 5
}
void f2() {
    i=3;        // Изменяется значение глобальной переменной
}
int main(){
    i=1;        // Изменяется значение глобальной переменной
    f1();       // f1() не меняет значение глобальной переменной
    cout<<i<<endl; // Будет напечатано 1
    f2();       // f2() меняет значение глобальной переменной
    cout<<i<<endl; // Будет напечатано 3
    return 0;   }
```

Вопрос: как будет работать программа, если добавить в функцию `main` определение переменной `i`, как локальной? А если при этом убрать объявление глобальной переменной `i`?

Передача параметров по значению и по ссылке

Переменные, в которых сохраняются параметры, передаваемые функции, также являются локальными для этой функции. Эти переменные создаются при вызове функции и в них копируются значения, передаваемые функции в качестве параметров. Эти переменные можно изменять, но все изменения этих переменных будут "забыты" после выхода из функции.

Рассмотрим это на примере следующей функции, "меняющей" значения двух переданных ей переменных:

```
#include<iostream>
using namespace std;
void swap(int a, int b)
{
    int t;
    t=b;
    b=a;
    a=t;
}
int main()
{
    int p=3,q=5;
    swap(p,q);
    cout<<p<<" "<<q<<endl;
    return 0;
}
```

При вызове функции swap создаются

новые переменные *a* и *b*, им присваиваются значения 3 и 5. Эти переменные никак не связаны с переменными *p* и *q* и их изменение не изменяет значения *p* и *q*. Такой способ передачи параметров называется *передачей параметров по значению*.

Чтобы функция могла изменять значения переменных, объявленных в других функциях, необходимо указать, что передаваемый параметр является не просто константной величиной, а переменной, необходимо передавать значения *по ссылке*. Для этого функцию swap следовало бы объявить следующим образом:

void swap(int & a, int & b)

Амперсанды перед именем переменной означают, что эта переменная является не локальной переменной, а ссылкой на переменную, указанную в качестве параметра при вызове функции. Теперь при вызове swap(*p,q*) переменные *a* и *b* являются синонимами для переменных *p* и *q*, и изменение их значений влечет изменение значений *p* и *q*. А вот вызывать функцию в виде swap(3,5) уже нельзя, поскольку 3 и 5 — это константы, и сделать переменные синонимами констант нельзя.

Однако в языке C (не C++) вообще не было такого понятия, как передача параметров по ссылке. Для того, чтобы реализовать функцию, аналогичную swap в рассмотренном примере, необходимо было передавать адреса переменных *p* и *q*, а сама функция при этом должна быть объявлена, как

*void swap(int * a, int * b)*

Поскольку в этом случае функция swap знает физические адреса в оперативной памяти переменных *p* и *q*, то разыменовав эти адреса функция swap сможет изменить значения самих переменных *p* и *q*.

Теперь вспомним, что в C++ массивы и указатели — это почти одно и то же. А поскольку передача массива по значению, то есть создание копии всего массива является трудоемкой операцией (особенно для больших массивов), то вместо массива всегда передается указатель на его начало. То есть два объявления функции void f(int A[10]) и void f(int * A) абсолютно идентичны. В любом случае функция f получит указатель на начало массива, а, значит, будет способна изменять значения его элементов.

Вызов по значению и вызов по ссылке.

В предыдущих примерах аргументы передаются функциям по значению. Когда переменные передаются по значению, в функцию передается копия текущего значения этой переменной. Поскольку передается копия переменной, сама эта переменная внутри вызывающей функции не изменяется. Вызов по значению — наиболее распространенный способ передачи информации (параметров) в функцию, и этот метод в С и С++ задан по умолчанию. Главным ограничением вызова по значению является то, что функция обычно возвращает только одно значение.

При вызове по ссылке в функцию передается не текущее значение, а адрес аргумента. Программе требуется меньше памяти, чем при вызове по значению. Кроме того, переменные в вызывающей функции могут быть изменены. Еще одним достоинством этого метода является то, что функция может возвращать более одного значения.

В результате выполнения операции & определяется адрес ячейки памяти, которая соответствует переменной. Если pooh — имя переменной, то &pooh — ее адрес. Можно представить себе адрес как ячейку памяти, но можно рассматривать его и как метку, которая используется компьютером для идентификации переменной. Предположим, мы имеем оператор

```
pooh = 24;
```

Пусть также адрес ячейки, где размещается переменная pooh, — 12126. Тогда в результате выполнения оператора

```
printf( "%d %d\n" , pooh, &pooh);
```

получим

```
24 12126
```

Более того, машинный код, соответствующий первому оператору, словами можно выразить приблизительно так: «Поместить число 24 в ячейку с адресом 12126».

Воспользуемся указанной выше операцией для проверки того, в каких ячейках хранятся значения переменных, принадлежащих разным функциям, но имеющих одно и то же имя.

```
/* контроль адресов */
#include<stdio.h>
void mikado(int);
void main()
{
    int pooh = 2, bah = 5;
    printf(" В main(), pooh = %d u &pooh = %u\n" , pooh, &pooh);
    printf(" В main(), bah = %d u &bah = %u\n", bah, &bah);
    mikado(pooh);
```

```
}
```

```
void mikado(int bah)
```

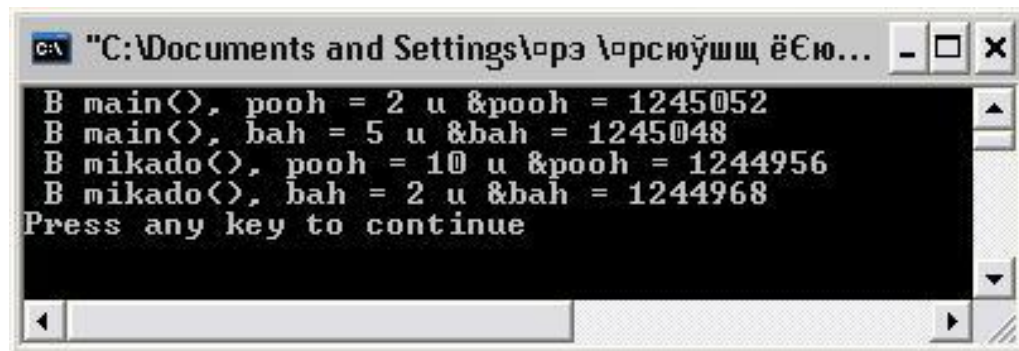
```
{
```

```
int pooh = 10;
```

```
printf(" B mikado(), pooh = %d u &pooh = %u\n", pooh, &pooh);
```

```
printf(" B mikado(), bah = %d u &bah = %u\n" , bah, &bah);
```

```
}
```



```
C:\Documents and Settings\user\Рабочий стол\ёёё...
B main(), pooh = 2 u &pooh = 1245052
B main(), bah = 5 u &bah = 1245048
B mikado(), pooh = 10 u &pooh = 1244956
B mikado(), bah = 2 u &bah = 1244968
Press any key to continue
```

Мы воспользовались форматом %u (целое без знака) для вывода на печать адресов на тот случай, если их величины превысят максимально возможное значение числа типа int. В нашей вычислительной системе результат работы этой маленькой программы выглядит так:

B main(), pooh = 2 и &pooh = 56002

B main(), bah = 5 и &bah = 50004

B mikado(), pooh = 10 и &pooh = 55994

B mikado(), bah = 2 и &bah = 56000

Использование указателей для связи между функциями

Мы только прикоснулись к обширному и увлекательному миру указателей. Сейчас нашей целью является использование указателей для решения задачи об установлении связи между функциями. Ниже приводится программа, в которой указатели служат средством, обеспечивающим правильную работу функции, которая осуществляет обмен значениями переменных.

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

```
void interchange(int *,int * );
```

```
void main()
```

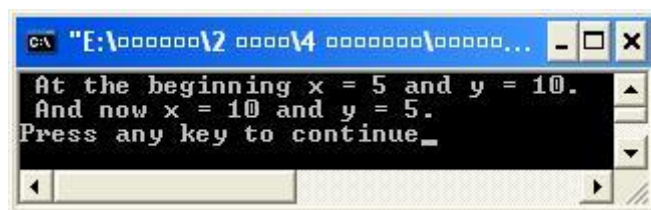
```
{
```

```
int x = 5, y = 10;
```

```
printf(" At the beginning x = %d and y = %d.\n" , x, y);
interchange(&x,&y); /* передача адресов функции */
printf(" And now x = %d and y = %d.\n" , x, y);
}
```

```
void interchange(int *u, int *v) /* u и v являются указателями */
{
int temp;
temp = *u; /* temp присваивается значение, на которое указывает u */
*u = *v;
*v = temp;
}
```

После всех встретившихся трудностей, проверим, работает ли этот вариант!



Посмотрим, как она работает. Во-первых, теперь вызов функции выглядит следующим образом:

```
interchange(&x, &y);
```

Вместо передачи значений *x* и *y* мы передаем их адреса. Это означает, что формальные аргументы *u* и *v*, имеющиеся в спецификации

```
interchange(u,v)
```

при обращении будут заменены адресами *u*, следовательно, они должны быть описаны как указатели. Поскольку *x* и *y* — целого типа, а *u* и *v* являются указателями на переменные целого типа, и мы вводим следующее описание:

```
int *u, *v;
```

Далее в теле функции оператор описания

```
int temp
```

используется с целью резервирования памяти. Мы хотим поместить значение переменной `x` в переменную `temp`, поэтому пишем

```
temp = *u;
```

Вспомните, что значение переменной `u` — это `&x`, поэтому переменная `u` ссылается на `x`. Это означает, что операция `*u` дает значение `x`, которое как раз нам и требуется. Мы не должны писать, например, так:

```
temp = u; /* неправильно */
```

поскольку при этом происходит запоминание адреса переменной `x`, а не ее значения; мы же пытаемся осуществить обмен значениями, а не адресами.

Точно так же, желая присвоить переменной `u` значение переменной `x`, мы пользуемся оператором

```
*u = *v;
```

который соответствует оператору

```
x = y;
```

Подведем итоги. Нам требовалась функция, которая могла бы изменять значения переменных `x` и `y`. Путем передачи функции адресов переменных `x` и `y` мы предоставили ей возможность доступа к ним. Используя указатели и операцию `*`, функция смогла извлечь величины, помещенные в соответствующие ячейки памяти, и поменять их местами.

Вообще говоря, при вызове функции информация о переменной может передаваться функции в двух видах. Если мы используем форму обращения

```
function1(x);
```

происходит передача значения переменной `x`. Если же мы используем форму обращения

```
function2(&x);
```

происходит передача адреса переменной `x`. Первая форма обращения требует, чтобы определение функции включало в себя формальный аргумент того же типа, что и `x`:

```
function1(num)
int num;
```

Вторая форма обращения требует, чтобы определение функции включало в себя формальный аргумент, являющийся указателем на объект соответствующего типа:

```
function2(ptr)
int *ptr;
```

80. Использование функций. Рекурсия. Равноправность функций в языке Си. Параметры и аргументы функций. Формальные и фактические параметры.

Функция — это самостоятельная единица программы, которая спроектирована для реализации конкретной подзадачи.

Функция является подпрограммой, которая может содержаться в основной программе, а может быть создана отдельно (в библиотеке). Каждая функция выполняет в программе определенные действия.

Сигнатура функции определяет правила использования функции. Обычно сигнатура представляет собой описание функции, включающее имя функции, перечень формальных параметров с их типами и тип возвращаемого значения.

Семантика функции определяет способ реализации функции. Обычно представляет собой тело функции.

Определение функции

Каждая функция в языке Си должна быть определена, то есть должны быть указаны:

- тип возвращаемого значения;
- имя функции;
- информация о формальных аргументах;
- тело функции.

Определение функции имеет следующий синтаксис:

```
ТипВозвращаемогоЗначения ИмяФункции(СписокФормальныхАргументов)
{
    ТелоФункции;
    ...
    return(ВозвращаемоеЗначение);
}
```

Пример: Функция сложения двух вещественных чисел

```
float function(float x, float z)
{
    float y;
    y=x+z;
    return(y);
}
```

В указанном примере возвращаемое значение имеет тип float. В качестве возвращаемого значения в вызывающую функцию передается значение переменной y. Формальными аргументами являются значения переменных x и z.

Если функция не возвращает значения, то тип возвращаемого значения для нее указывается как void. При этом операция return может быть опущена. Если функция не принимает аргументов, в круглых скобках также указывается void.

Различают *системные* (в составе систем программирования) и *собственные функции*.

Системные функции хранятся в стандартных библиотеках, и пользователю не нужно вдаваться в подробности их реализации. Достаточно знать лишь их сигнатуру. Примером системных функций, используемых ранее, являются функции printf() и scanf().

Собственные функции - это функции, написанные пользователем для решения конкретной подзадачи.

Разбиение программ на функции дает следующие преимущества:

- Функцию можно вызвать из различных мест программы, что позволяет избежать повторения программного кода.
- Одну и ту же функцию можно использовать в разных программах.
- Функции повышают уровень модульности программы и облегчают ее проектирование.
- Использование функций облегчает чтение и понимание программы и ускоряет поиск и исправление ошибок.

С точки зрения вызывающей программы функцию можно представить как некий "черный ящик", у которого есть несколько входов и один выход. С точки зрения вызывающей программы неважно, каким образом производится обработка информации внутри функции. Для корректного использования функции достаточно знать лишь ее сигнатуру.

Вызов функции

Общий вид вызова функции

```
Переменная = ИмяФункции(СписокФактическихАргументов);
```

Фактический аргумент — это величина, которая присваивается формальному аргументу при вызове функции. Таким образом, **формальный аргумент** — это переменная в вызываемой функции, а фактический аргумент — это конкретное значение, присвоенное этой переменной вызывающей функцией. Фактический аргумент может быть константой, переменной или выражением. Если фактический аргумент представлен в виде выражения, то его значение сначала вычисляется, а затем передается в вызываемую функцию. Если в функцию требуется передать

несколько значений, то они записываются через запятую. При этом формальные параметры заменяются значениями фактических параметров в порядке их следования в сигнатуре функции.

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

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

```
int is_in (char *s, char c) {
```

```
while(*s)
```

```
if(*s==c) return 1;
```

```
else s++;
```

```
return 0;
```

```
}
```

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

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

Возврат в вызывающую функцию

По окончании выполнения вызываемой функции осуществляется возврат значения в точку ее вызова. Это значение присваивается переменной, тип которой должен соответствовать типу возвращаемого значения функции. Функция может передать в вызывающую программу только одно значение. Для передачи возвращаемого значения в вызывающую функцию используется оператор `return` в одной из форм:

```
return(ВозвращаемоеЗначение);
```

```
return ВозвращаемоеЗначение;
```

Действие оператора следующее: значение выражения, заключенного в скобки, вычисляется и передается в вызывающую функцию. Возвращаемое значение может использоваться в вызывающей программе как часть некоторого выражения.

Оператор `return` также завершает выполнение функции и передает управление следующему оператору в вызывающей функции. Оператор `return` не обязательно должен находиться в конце тела функции.

Функции могут и не возвращать значения, а просто выполнять некоторые вычисления. В этом случае указывается пустой тип возвращаемого значения `void`, а оператор `return` может либо

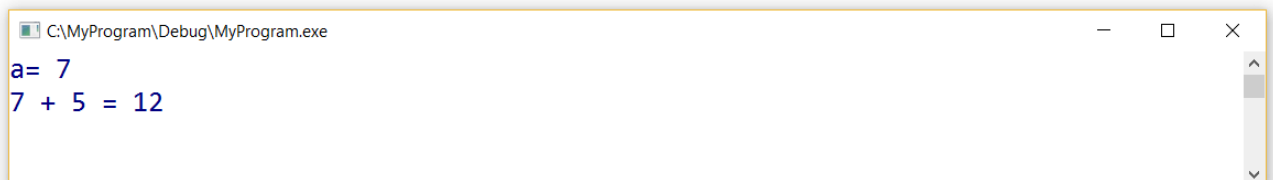
отсутствовать, либо не возвращать никакого значения:

```
return;
```

Пример: Посчитать сумму двух чисел.

```
#define _CRT_SECURE_NO_WARNINGS // для возможности использования scanf
#include <stdio.h>
// Функция вычисления суммы двух чисел
int sum(int x, int y) // в функцию передаются два целых числа
{
    int k = x + y; // вычисляем сумму чисел и сохраняем в k
    return k;      // возвращаем значение k
}
int main()
{
    int a, r;      // описание двух целых переменных
    printf("a= ");
    scanf("%d", &a); // вводим a
    r = sum(a, 5);  // вызов функции: x=a, y=5
    printf("%d + 5 = %d", a, r); // вывод: a + 5 = r
    getchar(); getchar(); // мы использовали scanf(),
    return 0; // поэтому getchar() вызываем дважды
}
```

Результат выполнения



В языке Си нельзя определять одну функцию внутри другой.

В языке Си нет требования, чтобы семантика функции обязательно предшествовало её вызову. Функции могут определяться как до вызывающей функции, так и после нее. Однако если семантика вызываемой функции описывается ниже ее вызова, необходимо до вызова функции определить прототип этой функции, содержащий:

- тип возвращаемого значения;
- имя функции;
- типы формальных аргументов в порядке их следования.

Прототип необходим для того, чтобы компилятор мог осуществить проверку соответствия типов передаваемых фактических аргументов типам формальных аргументов. Имена формальных аргументов в прототипе функции могут отсутствовать.

Если в примере выше тело функции сложения чисел разместить после тела функции main, то код будет выглядеть следующим образом:

```

#define _CRT_SECURE_NO_WARNINGS // для возможности использования scanf
#include <stdio.h>
int sum(int, int); // сигнатура
int main()
{
    int a, r;
    printf("a= ");
    scanf("%d", &a);
    r = sum(a, 5); // вызов функции: x=a, y=5
    printf("%d + 5 = %d", a, r);
    getchar(); getchar();
    return 0;
}
int sum(int x, int y) // семантика
{
    int k;
    k = x + y;
    return(k);
}

```

Рекурсивные функции

Функция, которая вызывает сама себя, называется **рекурсивной функцией**.

Рекурсия - вызов функции из самой функции.

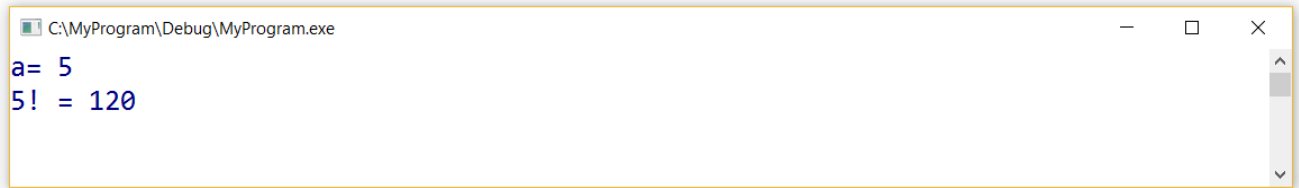
Пример рекурсивной функции - функция вычисления факториала.

```

#define _CRT_SECURE_NO_WARNINGS // для возможности использования scanf
#include <stdio.h>
int fact(int num) // вычисление факториала числа num
{
    if (num <= 1) return 1; // если число не больше 1, возвращаем 1
    else return num*fact(num - 1); // рекурсивный вызов для числа на 1
    меньше
}
// Главная функция
int main()
{
    int a, r;
    printf("a= ");
    scanf("%d", &a);
    r = fact(a); // вызов функции: num=a
    printf("%d! = %d", a, r);
    getchar(); getchar();
    return 0;
}

```

Результат выполнения



```
C:\MyProgram\Debug\MyProgram.exe
a= 5
5! = 120
```

Математические функции

Математические функции хранятся в стандартной библиотеке `math.h`. Аргументы большинства математических функций имеют тип `double`. Возвращаемое значение также имеет тип `double`. Углы в тригонометрических функциях задаются в радианах.

Основные математические функции стандартной библиотеки.

Функция	Описание
<code>int abs(int x)</code>	Модуль целого числа x
<code>double acos(double x)</code>	Арккосинус x
<code>double asin(double x)</code>	Арсинус x
<code>double atan(double x)</code>	Арктангенс x
<code>double cos(double x)</code>	Косинус x
<code>double cosh(double x)</code>	Косинус гиперболический x
<code>double exp(double x)</code>	Экспонента x
<code>double fabs(double x)</code>	Модуль вещественного числа
<code>double fmod(double x, double y)</code>	Остаток от деления x/y
<code>double log(double x)</code>	Натуральный логарифм x
<code>double log10(double x)</code>	Десятичный логарифм x
<code>double pow(double x, double y)</code>	x в степени y
<code>double sin(double x)</code>	Синус x
<code>double sinh(double x)</code>	Синус гиперболический x
<code>double sqrt(double x)</code>	Квадратный корень x

Функция	Описание
double tan(double x)	Тангенс x
double tanh(double x)	Тангенс гиперболический x

/*08FACTR.C

Программа на C, использующая рекурсивные вызовы функции.

Вычисляет факториал числа.

Пример: $7! = 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1 = 5040$ */

```
#include "stdafx.h"
```

```
#include <iostream>
```

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

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

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

```
#include <ctype.h>
```

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

```
using namespace std;
```

```
double dfactorial(double danswer);
```

```
main()
```

```
{
```

```
double dnumber=15.0;
```

```
double dresult;
```

```
dresult=dfactorial(dnumber);
```

```
printf("The factorial of %.0lf is: %.0lf\n",dnumber,dresult);
```

```
printf ("\n\nPress any key to finish\n");
```

```
_getch();
```

```
return(0);
```

```
}
```

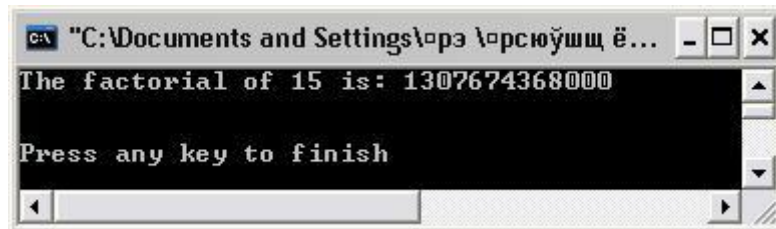
```
double dfactorial(double danswer)
```

```
{
```

```

if(danswer <= 1.0)
return(1.0);
else
return(danswer*dfactorial(danswer-1.0));
}

```



Возникает рекурсия, так как внутри функции `dfactorial()` имеется вызов ее самой.

В `printf("%d", (int)sqrt(...))` приведении в целое число усекает цифры в целые числа, поэтому 2.9 становится 2 - и `%d` печатает это.

Но в `printf("%.1f", sqrt(...))` число с плавающей запятой округляется до нуля десятичных цифр, поэтому 2.9 записывается как 3.

Таким образом, если ответ не является целым числом, результаты могут отличаться.

Равноправность функций в языке Си.

Все функции в программе, написанной на языке Си, равноправны: каждая из них может вызывать любую другую функцию и в свою очередь каждая может быть вызвана любой другой функцией. Это делает функции языка Си несколько отличными от процедур Паскаля, поскольку процедуры в Паскале могут быть вложены в другие процедуры (причем, процедуры, содержащиеся в одном гнезде, являются недоступными для процедур, расположенных в другом).

Нет ли у функции `main()` какой-то специфики? Безусловно, есть. Она заключается в том, что после «сборки» программы, состоящей из нескольких функций, ее выполнение начинается с первого оператора функции `main()`. Но этим ее исключительность и ограничивается. Даже функция `main()` может быть вызвана другими функциями.

Аргумент типа `void`.

В соответствии с ANSI C, отсутствие списка аргументов функции должно быть указано явно при помощи ключевого слова `void`. В C++ использование `void` пока не обязательно, но считается целесообразным. В следующей программе имеется простая функция `voutput()`, не имеющая параметров и не возвращающая никакого значения. Функция `main()` вызывает `voutput()`. При выходе из `voutput()` управление возвращается функции `main()`. Трудно придумать более простую функцию

```
/*08FVOID.C
```

Программа на C печатает сообщение при помощи функции.

В функции используются параметр типа void и стандартная библиотечная функция C sqrt()*/

```
#include "stdafx.h"
```

```
#include <iostream>
```

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

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

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

```
#include <ctype.h>
```

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

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

```
using namespace std;
```

```
void voutput(void);
```

```
main()
```

```
{
```

```
/* Программа определяет квадратный корень */
```

```
printf("This programm will find the square root\n\n");
```

```
voutput();
```

```
printf ("\n\nPress any key to finish\n");
```

```
_getch();
```

```
return(0);
```

```
}
```

```
void voutput(void)
```

```
{
```

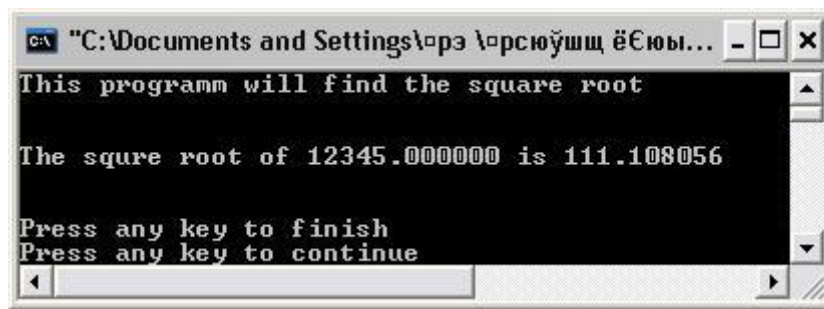
```
double dt=12345.0;
```

```
double du;
```

```
du=sqrt(dt);
```

```
printf("The squire root of %lf is %lf\n",dt,du);
```

```
}
```



Обратите внимание, что функция `voutput()` вызывает библиотечную функцию C, называемую `sqrt()`. Прототип `sqrt()` находится в файле `math.h`. У функции один параметр в формате числа двойной длины, и возвращает она результат извлечения квадратного корня тоже в виде числа двойной длины.

Символьные параметры.

Функции можно передавать символьные значения. В следующем примере в функции `main()` одиночный символ считывается с клавиатуры и передается функции `voutput()`. Символ считывается функцией `getch()`. В стандартной библиотеке C имеются другие функции, тесно связанные с функцией `getch()`: `getc()`, `getcharQ` и `getcheQ`. Эти функции можно использовать и в C++, однако во многих случаях предпочтительнее пользоваться `std`. Функция `getch()` получает символ от стандартного устройства ввода (клавиатуры) и возвращает символьное значение, не отображая его на экране:

```
//08FCHAR.C
```

```
/*Программа на C считывает символ с клавиатуры, передает его функции  
и печатает сообщение, использующее этот символ*/
```

```
#include "stdafx.h"
```

```
#include <iostream>
```

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

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

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

```
#include <ctype.h>
```

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

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

```
using namespace std;
```

```
void voutput(char c);
```

```
main()
```

```
{
```

```

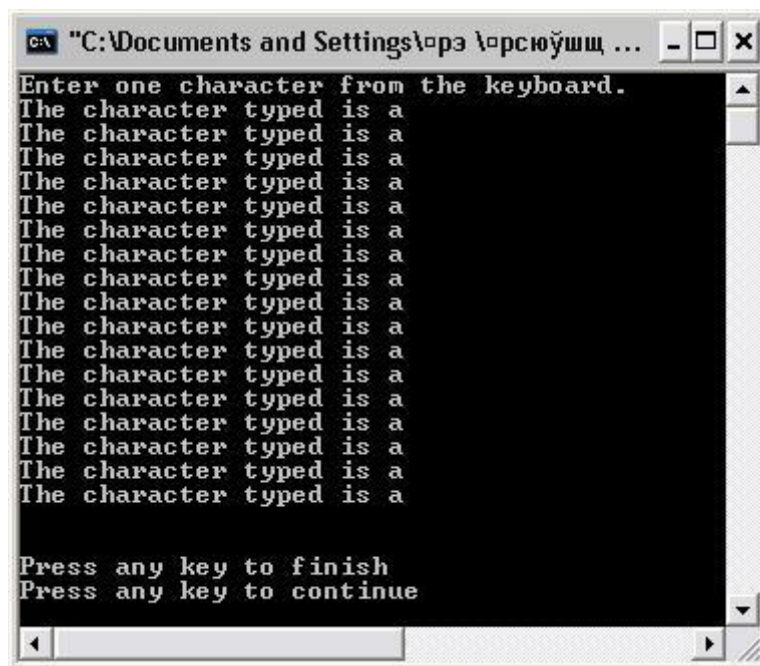
char cyourchar;

/* Введите один символ с клавиатуры */
printf("Enter one character from the keyboard. \n");
cyourchar=getch();
voutput(cyourchar);

printf ("\n\nPress any key to finish\n");
_getch();
return(0);
}

void voutput(char c)
{
int j;
for(j=0;j<=16;j++)
/* Введен символ ... */
printf("The character typed is %c \n",c);
}

```



Целочисленные параметры.

В следующем примере одно целое число вводится с клавиатуры при помощи функции C scanf() и передается функции vside(), в которой на основе полученного значения, означающего длину стороны, вычисляются и печатаются площадь квадрата, объем куба и площадь поверхности куба.

//08FINT.C

/*Программа на С вычисляет значения на основании введенной длины.

Функция получает параметр типа int, введенный с

клавиатуры при помощи функции scanf()*/

```
#include "stdafx.h"
```

```
#include <iostream>
```

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

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

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

```
#include <ctype.h>
```

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

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

```
using namespace std;
```

```
void vside(int is);
```

```
main()
```

```
{
```

```
int iyourlength=0;
```

```
/* Введите с клавиатуры длину как целое число */
```

```
printf("Enter the length, as an integer from the keyboard. \n");
```

```
scanf("%d",&iyourlength);
```

```
vside(iyourlength);
```

```
printf ("\n\nPress any key to finish\n");
```

```
_getch();
```

```
return(0);
```

```
}
```

```
void vside(int is)
```

```
{
```

```
int iarea, ivolume,isarea;
```

```
iarea=is*is;
```

```
ivolume=is*is*is;
```

```
isarea=6*iarea;
```

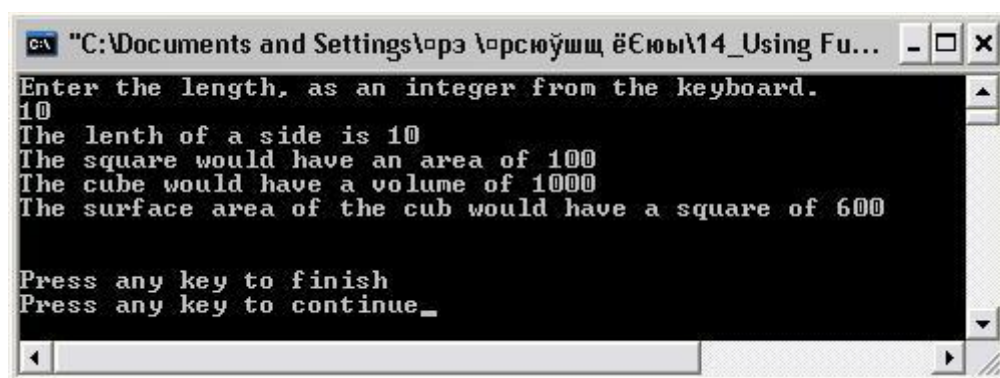
```
/* Длина стороны равна */
```

```
printf("The lenth of a side is %d \n",is);
```

```

/* Квадрат будет иметь площадь */
printf("The square would have an area of %d \n",iarea);
/* Куб будет иметь объем */
printf("The cube would have a volume of %d \n",ivolume);
/* Площадь поверхности куба */
printf("The surface area of the cub would have a square of %d \n",isarea);
}

```



Параметры в формате чисел с плавающей точкой.

Числа с плавающей точкой так же легко передавать в качестве параметров функции, как и целые значения.

Параметры в формате чисел двойной длины.

Тип чисел двойной длины `double` обеспечивает очень большую точность чисел с плавающей точкой. Все функции, описанные в заголовочном файле `math.h`, получают и возвращают значения типа `double`. В своих программах вы можете использовать и другие математические функции, перечисленные в табл. 8.1. Для получения более подробной информации вы можете также просмотреть содержимое файла `math.h`.

Таблица. Математические функции, описанные в заголовочном файле Microsoft `math.h`

<code>acos, acosl</code>	Арккосинус
<code>asin, asinl</code>	Арсинус
<code>atan, atanl</code>	Арктангенс
<code>atan2, atan2l</code>	Арктангенс
<code>bessel</code>	Функции Бесселя
<code>_cabs, _cabsl</code>	Абсолютное значение комплексного числа

ceil, ceill	Целочисленное максимальное значение
_chgsign	Инвертирование знака
_clear87, clearfp	Чтение и сброс слова состояния числа с плавающей точкой
_control87, _controlfp	Чтение старого управляющего слова числа с плавающей точкой и установка нового
_copysign	Возвращает число x со знаком числа y
cos, cosl	Косинус
cosh, coshl	Гиперболический косинус
_dieetombsbin	Преобразование IEEE-числа двойной точности в двоичный формат Microsoft
Div	Деление одного целого на другое, возвращается частное и остаток
_dmsbintoieee	Преобразование Microsoft-числа двойной точности в формат IEEE
exp, expl	Степенная функция
fabs, fabsl	Абсолютное значение
_fieeeetombsbin	Преобразование IEEE-числа одинарной точности в двоичный формат Microsoft
_finite	Проверка числа с плавающей точкой на бесконечность
floor, floorl	Нахождение наибольшего целого, меньшего или равного аргументу
fmod, fmodl	Нахождение остатка
_fmsbintoieee	Преобразование Microsoft-числа одинарной точности в формат IEEE
_fpclass	Возвращает слово состояния с информацией о классе чисел с плавающей точкой
_fpieee_ftl	Вызов описанного пользователем обработчика исключительных ситуаций для чисел с плавающей точкой IEEE-стандарта
_fpreset	Повторная инициализация пакета математических функций

frexp, frexpl	Вычисление экспоненциального значения
_hypot, _hypotl	Вычисление гипотенузы правильного треугольника
_isnan	Проверка числа с плавающей точкой на значение "не число" (NaN)
ldexp, ldexpl	Произведение от аргумента
Ldiv	Деление одного целого long на другое, возвращается частное и остаток
log, logl	Натуральный логарифм
log10, log10l	Десятичный логарифм
_logb	Выделение показателя числа с плавающей точкой
_lrotl, _lrotr	Сдвиг числа unsigned long int влево или вправо
_matherr, _matherrl	Обработка математических ошибок
_max, _min	Определение большего или меньшего из двух значение
modf, modfl	Деление аргумента на целую и дробную части
_nextafter	Определение следующего значения
pow, powl	Вычисление значения, возведенного в степень
Rand	Получение псевдослучайного числа
_rotl, _rotr	Сдвиг числа unsigned int влево или вправо
_scalb	Степень числа 2, определяемая аргументом
sin, sinl	Синус
sinh, sinhl	Гиперболический синус
sqrt, sqrtl	Квадратный корень
Srand	Инициализация датчика псевдослучайных чисел

<code>_status87,</code> <code>_statusfp</code>	Получение слов состояния числа с плавающей точкой
<code>tan, tanl</code>	Тангенс
<code>tanh, tanhl</code>	Гиперболический тангенс

Массивы в качестве параметров.

В следующем примере содержимое некоторого массива передается в функцию в качестве параметра, вызываемого по ссылке. В этом случае адрес первого элемента массива передается через указатель.

```
//08FPNTR.C
```

```
/*Программа на С передает функции массив в качестве параметра.
```

```
Для передачи информации о массиве используется указатель*/
```

```
#include "stdafx.h"
```

```
#include <iostream>
```

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

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

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

```
#include <ctype.h>
```

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

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

```
using namespace std;
```

```
void voutput(int *pinums);
```

```
main()
```

```
{
```

```
int iyourarray[7]={2,7,15,32,45,3,1};
```

```
/* Передать информацию о массиве в функцию */
```

```
printf("Send array information to function. \n");
```

```
voutput(iyourarray);
```

```
printf ("\n\nPress any key to finish\n");
```

```
_getch();
```

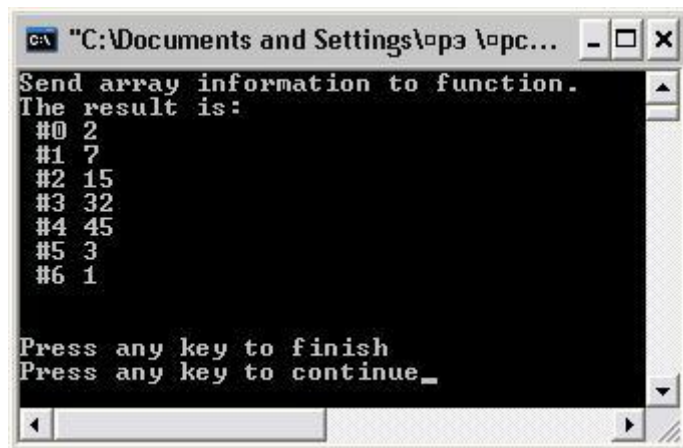
```
return(0);
```

```
}
```

```

void voutput(int *pinums)
{
    int t;
    /* Результат равен */
    printf("The result is:\n");
    for(t=0;t<7;t++)
        printf(" #%d %d\n",t,pinums[t]);
}

```



```

C:\Documents and Settings\user\pc...
Send array information to function.
The result is:
#0 2
#1 7
#2 15
#3 32
#4 45
#5 3
#6 1
Press any key to finish
Press any key to continue_

```

Обратите внимание на то, что при вызове функции указывается только имя `yourarray`.

Также допустимо передать информацию по адресу и для безразмерного массива. В следующем примере показано, как это можно сделать на C++. (Такой же подход возможен и в C.) Информация, содержащаяся в массиве `yourarray`, передается путем указания адреса его первого элемента.

81. Использование функций. Аргумент типа `void`. Символьные параметры. Целочисленные параметры.

Функция — это самостоятельная единица программы, которая спроектирована для реализации конкретной подзадачи.

Функция является подпрограммой, которая может содержаться в основной программе, а может быть создана отдельно (в библиотеке). Каждая функция выполняет в программе определенные действия.

Сигнатура функции определяет правила использования функции. Обычно сигнатура представляет собой описание функции, включающее имя функции, перечень формальных параметров с их типами и тип возвращаемого значения.

Семантика функции определяет способ реализации функции. Обычно представляет собой тело функции.

Определение функции

Каждая функция в языке Си должна быть определена, то есть должны быть указаны:

- тип возвращаемого значения;
- имя функции;
- информация о формальных аргументах;
- тело функции.

Определение функции имеет следующий синтаксис:

```
ТипВозвращаемогоЗначения ИмяФункции(СписокФормальныхАргументов)
{
    ТелоФункции;
    ...
    return(ВозвращаемоеЗначение);
}
```

Пример: Функция сложения двух вещественных чисел

```
float function(float x, float z)
{
    float y;
    y=x+z;
    return(y);
}
```

Объявление функций: прототипы функций

```
1    returnDataType functionName( dataType argName1, dataType argName2, ..., dataType argNameN);
```

где,

returnDataType — возвращаемый тип данных

functionName — имя функции

dataType — тип данных

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

Смотрим пример объявления функции:

```
1    // Объявление прототипа функции с двумя целыми параметрами
2    // функция принимает два аргумента и возвращает их сумму
3    int sum(int num1, int num2);
```

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

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

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

Определение функций

```
1    returnType functionName( dataType argName1, dataType argName2, ..., dataType argNameN)
2    {
3       // тело функции
4    }
```

Рассмотрим определение функции на примере функции sum.

```
1    // определение функции, которая суммирует два целых числа и возвращает их сумму
2    int sum(int num1, int num2)
3    {
4       return (num1 + num2);
5    }
```

Аргумент типа void.

В соответствии с ANSI C, отсутствие списка аргументов функции должно быть указано явно при помощи ключевого слова void. В C++ использование void пока не обязательно, но считается целесообразным. В следующей программе имеется простая функция voutput(), не имеющая параметров и не возвращающая никакого значения. Функция main() вызывает voutput(). При выходе из voutput() управление возвращается функции main(). Трудно придумать более простую функцию

/*08FVOID.C

Программа на C печатает сообщение при помощи функции.

В функции используются параметр типа void и стандартная библиотечная функция C sqrt()*/

```
#include "stdafx.h"
```

```
#include <iostream>
```

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

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

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

```
#include <ctype.h>
```

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



```

#include <math.h>

using namespace std;

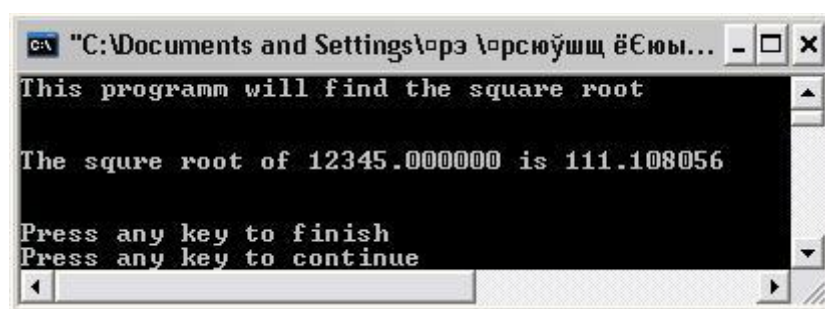
void voutput(void);

main()
{
/* Программа определяет квадратный корень */
printf("This programm will find the square root\n\n");
voutput();

printf ("\n\nPress any key to finish\n");
_getch();
return(0);
}

void voutput(void)
{
double dt=12345.0;
double du;
du=sqrt(dt);
printf("The squire root of %lf is %lf\n",dt,du);
}

```



Обратите внимание, что функция `voutput()` вызывает библиотечную функцию C, называемую `sqrt()`. Прототип `sqrt()` находится в файле `math.h`. У функции один параметр в формате числа двойной длины, и возвращает она результат извлечения квадратного корня тоже в виде числа двойной длины.

Символьные параметры.

Функции можно передавать символьные значения. В следующем примере в функции `main()` одиночный символ считывается с клавиатуры и передается функции `voutput()`. Символ считывается функцией `getch()`. В стандартной библиотеке C имеются другие функции, тесно

связанные с функцией getch(): getch(), getcharQ и getcheQ Эти функции можно использовать и в C++, однако во многих случаях предпочтительнее пользоваться cm. Функция getch() получает символ от стандартного устройства ввода (клавиатуры) и возвращает символьное значение, не отображая его на экране:

```
//08FCHAR.C
```

```
/*Программа на C считывает символ с клавиатуры, передает его функции
```

```
и печатает сообщение, использующее этот символ*/
```

```
#include "stdafx.h"
```

```
#include <iostream>
```

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

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

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

```
#include <ctype.h>
```

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

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

```
using namespace std;
```

```
void voutput(char c);
```

```
main()
```

```
{
```

```
char cyourchar;
```

```
/* Введите один символ с клавиатуры */
```

```
printf("Enter one character from the keyboard. \n");
```

```
cyourchar=getch();
```

```
voutput(cyourchar);
```

```
printf ("\n\nPress any key to finish\n");
```

```
_getch();
```

```
return(0);
```

```
}
```

```
void voutput(char c)
```

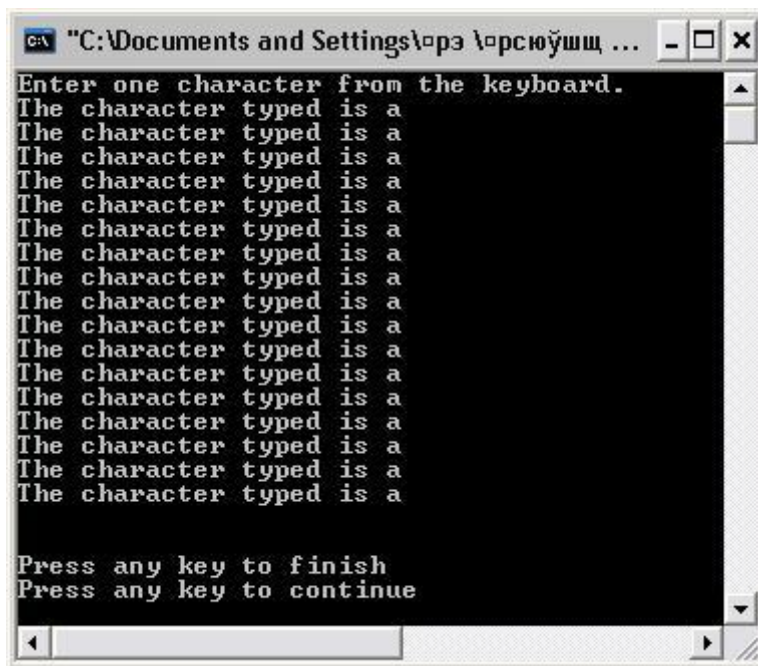
```
{
```

```
int j;
```

```
for(j=0;j<=16;j++)
```

```
/* Введен символ ... */
```

```
printf("The character typed is %c \n",c);  
}
```



Целочисленные параметры.

В следующем примере одно целое число вводится с клавиатуры при помощи функции C `scanf()` и передается функции `vside()`, в которой на основе полученного значения, означающего длину стороны, вычисляются и печатаются площадь квадрата, объем куба и площадь поверхности куба.

```
//08FINT.C
```

```
/*Программа на C вычисляет значения на основании введенной длины.
```

```
Функция получает параметр типа int, введенный с
```

```
клавиатуры при помощи функции scanf()*/
```

```
#include "stdafx.h"
```

```
#include <iostream>
```

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

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

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

```
#include <ctype.h>
```

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

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

```
using namespace std;
```

```
void vside(int is);
```

```
main()
```

```

{
int iyourlength=0;
/* Введите с клавиатуры длину как целое число */
printf("Enter the length, as an integer from the keyboard. \n");
scanf("%d",&iyourlength);
vside(iyourlength);

printf ("\n\nPress any key to finish\n");
_getch();
return(0);
}

void vside(int is)
{
int iarea, ivolume,isarea;
iarea=is*is;
ivolume=is*is*is;
isarea=6*iarea;
/* Длина стороны равна */
printf("The lenth of a side is %d \n",is);
/* Квадрат будет иметь площадь */
printf("The square would have an area of %d \n",iarea);
/* Куб будет иметь объем */
printf("The cube would have a volume of %d \n",ivolume);
/* Площадь поверхности куба */
printf("The surface area of the cub would have a square of %d \n",isarea);
}

```

```

C:\Documents and Settings\user\14_Using Fu...
Enter the length, as an integer from the keyboard.
10
The lenth of a side is 10
The square would have an area of 100
The cube would have a volume of 1000
The surface area of the cub would have a square of 600

Press any key to finish
Press any key to continue_

```

82. Использование функций. Массивы в качестве параметров.

Аргументы по умолчанию. Возвращение значения функцией: оператор `return`.

Функция — это самостоятельная единица программы, которая спроектирована для реализации конкретной подзадачи.

Функция является подпрограммой, которая может содержаться в основной программе, а может быть создана отдельно (в библиотеке). Каждая функция выполняет в программе определенные действия.

Сигнатура функции определяет правила использования функции. Обычно сигнатура представляет собой описание функции, включающее имя функции, перечень формальных параметров с их типами и тип возвращаемого значения.

Семантика функции определяет способ реализации функции. Обычно представляет собой тело функции.

Определение функции

Каждая функция в языке Си должна быть определена, то есть должны быть указаны:

- тип возвращаемого значения;
- имя функции;
- информация о формальных аргументах;
- тело функции.

Определение функции имеет следующий синтаксис:

```
ТипВозвращаемогоЗначения ИмяФункции(СписокФормальныхАргументов)
{
    ТелоФункции;
    ...
    return(ВозвращаемоеЗначение);
}
```

Пример: Функция сложения двух вещественных чисел

```
float function(float x, float z)
{
    float y;
    y=x+z;
    return(y);
}
```

Объявление функций: прототипы функций

```
1    returnType functionName( dataType argName1, dataType argName2, ..., dataType argNameN);
```

где,

returnDataType — возвращаемый тип данных
functionName — имя функции
dataType — тип данных
argName1...N — имена параметров функции (количество параметров неограниченно)

Смотрим пример объявления функции:

```
1 // Объявление прототипа функции с двумя целыми параметрами
2 // функция принимает два аргумента и возвращает их сумму
3 int sum(int num1, int num2);
```

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

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

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

Определение функций

```
1 returnType functionName( dataType argName1, dataType argName2, ..., dataType argNameN)
2 {
3     // тело функции
4 }
```

Рассмотрим определение функции на примере функции sum.

```
1 // определение функции, которая суммирует два целых числа и возвращает их сумму
2 int sum(int num1, int num2)
3 {
4     return (num1 + num2);
5 }
```

Массивы в качестве параметров.

В следующем примере содержимое некоторого массива передается в функцию в качестве параметра, вызываемого по ссылке. В этом случае адрес первого элемента массива передается через указатель.

//08FPNTR.C

/*Программа на С передает функции массив в качестве параметра.

Для передачи информации о массиве используется указатель*/

```
#include "stdafx.h"
```

```
#include <iostream>
```

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

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

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

```
#include <ctype.h>
```

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

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

```
using namespace std;
```

```
void voutput(int *pinums);
```

```
main()
```

```
{
```

```
int iyourarray[7]={2,7,15,32,45,3,1};
```

```
/* Передать информацию о массиве в функцию */
```

```
printf("Send array information to function. \n");
```

```
voutput(iyourarray);
```

```
printf ("\n\nPress any key to finish\n");
```

```
_getch();
```

```
return(0);
```

```
}
```

```
void voutput(int *pinums)
```

```
{
```

```
int t;
```

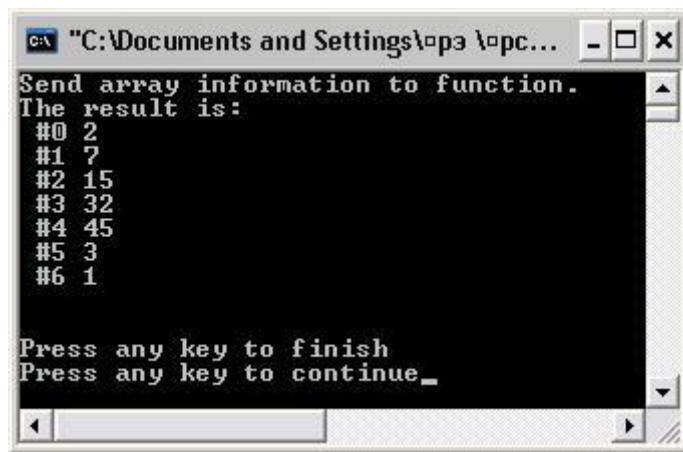
```
/* Результат равен */
```

```
printf("The result is:\n");
```

```
for(t=0;t<7;t++)
```

```
printf(" #%d %d\n",t,pinums[t]);
```

```
}
```



Обратите внимание на то, что при вызове функции указывается только имя `yourarray`.

Также допустимо передать информацию по адресу и для безразмерного массива. В следующем примере показано, как это можно сделать на C++. (Такой же подход возможен и в C.) Информация, содержащаяся в массиве `yourarray`, передается путем указания адреса его первого элемента.

Аргументы функций по умолчанию.

При обращении к функции, можно опускать некоторые её аргументы, но для этого необходимо при объявлении прототипа данной функции проинициализировать её параметры какими-то значениями, эти значения и будут использоваться в функции по умолчанию. Аргументы по умолчанию должны быть заданы в прототипе функции. Если в функции несколько параметров, то параметры, которые опускаются должны находиться правее остальных. Таким образом, если опускается самый первый параметр функции, то все остальные параметры тоже должны быть опущены. Если опускается какой-то другой параметр, то все параметры, расположенные перед ним могут не опускаться, но после него они должны быть опущены. Разработаем программу, в которой объявим функцию с аргументами по умолчанию.

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

```
#include "stdafx.h"
#include <iostream>
#include <cmath> // описывает работу математической функции sqrt() - квадратный корень
using namespace std;

double heron_space(const double a = 5, const double b = 6.5, const double c = 10.7); //параметры
функции инициализированы по умолчанию

int main(int argc, char* argv[])
{
    cout << "S = " << heron_space() << endl << endl; // все параметры используются по умолчанию
    cout << "S = " << heron_space(10,5) << endl << endl; // только последний параметр используется
по умолчанию
    cout << "S = " << heron_space(7) << endl << endl; // два последних параметра берутся по
умолчанию, а первый равен 7
    system("pause");
    return 0;
}

double heron_space(const double a, const double b, const double c) // функция вычисления площади
треугольника по формуле Герона
{
    const double p = (a + b + c) / 2; // полупериметр
    cout << "a = " << a << "\nb = " << b << "\nc = " << c << endl;
    return (sqrt(p * (p - a) * (p - b) * (p - c))); // формула Герона для нахождения площади
треугольника
}
```


Параметры функции инициализированы по умолчанию в прототипе функции. Если при запуске функции не передавать ей значения, то по умолчанию будут использоваться аргументы **5, 6.5, 10.7**. В строках **12, 13, 14** показаны различные способы использования функции `heron_space()` с аргументами по умолчанию. Данная функция `heron_space` вычисляет площадь треугольников по формуле Герона. Формула Герона позволяет вычислить площадь треугольника (S) по его сторонам a, b, c:

$$S = \sqrt{p(p-a)(p-b)(p-c)},$$

где p — полупериметр треугольника:

$$p = \frac{a + b + c}{2}$$

В строке **5** подключен стандартный заголовочный файл `<cmath>` для того, чтобы воспользоваться функцией вычисления корня квадратного `sqrt()` в строке **23**. Результат работы программы показан на рисунке.

```
a = 5
b = 6.5
c = 10.7
S = 11.1618

a = 10
b = 5
c = 10.7
S = 24.8615

a = 7
b = 6.5
c = 10.7
S = 21.9956
```

Иногда при многократном использовании одной функции необходимо менять не все её параметры, в таком случае использование параметров по умолчанию очень удобно.

Возвращение значения функцией: оператор `return`.

Приведем наш драйвер и функцию, вычисляющую абсолютную величину числа:

```
/* abs.драйвер */
#include<stdio.h>

int abs(int );

void main()
{
    int a= 10, b= 0, c= -22;
    int d, e, f;

    d = abs(a);
    e = abs(b);
    f = abs(c);
    printf(" %d %d %d\n", d, e, f);
}
```

```
}
```

```
/* функция, вычисляющая величину числа */
```

```
int abs(int x)
```

```
{
```

```
int y;
```

```
y = (x < 0) ? -x : x; /*вспомните операцию ? : */
```

```
return (y); /* возвращает значение y вызывающей программе */
```

```
}
```

Результат работы программы выглядит так:

```
10 0 22
```



Ключевое слово `return` указывает на то, что значение выражения, заключенного в круглые скобки, будет присвоено функции, содержащей это ключевое слово. Поэтому, когда функция `abs()` впервые вызывается нашим драйвером, значением `abs(a)` будет число 10, которое затем присваивается переменной `d`.

Переменная `y` является внутренним объектом функции `abs()`, но значение `y` передается в вызывающую программу с помощью оператора `return`. Действие, оказываемое оператором

```
d = abs(a);
```

по-другому можно выразить так:

```
abs(a);
```

```
d = y;
```

Оператор `return` оказывает и другое действие. Он завершает выполнение функции и передает управление следующему оператору в вызывающей функции. Это происходит даже в том случае, если оператор `return` является не последним оператором тела функции. Следовательно, функцию `abs()` мы могли бы записать следующим образом:

```
/* функция, вычисляющая абсолютную величину числа, вторая
версия */

abs(x)
int x;
{
if (x < 0)
return(-x);
else
return(x);
}
```

Наличие оператора return препятствует тому, чтобы оператор печати printf() когда-нибудь выполнялся в программе. Вы можете также использовать просто оператор

```
return;
```

Его применение приводит к тому, что функция, в которой он содержится, завершает свое выполнение и управление возвращается в вызывающую функцию. Поскольку у данного оператора отсутствует выражение в скобках, какое значение при этом не передается функции.

83. Использование функций. Типы функций. Функции типа void. Функции типа char. Функции типа int.

Функция — это самостоятельная единица программы, которая спроектирована для реализации конкретной подзадачи.

Функция является подпрограммой, которая может содержаться в основной программе, а может быть создана отдельно (в библиотеке). Каждая функция выполняет в программе определенные действия.

Сигнатура функции определяет правила использования функции. Обычно сигнатура представляет собой описание функции, включающее имя функции, перечень формальных параметров с их типами и тип возвращаемого значения.

Семантика функции определяет способ реализации функции. Обычно представляет собой тело функции.

Определение функции

Каждая функция в языке Си должна быть определена, то есть должны быть указаны:

- тип возвращаемого значения;
- имя функции;
- информация о формальных аргументах;
- тело функции.

Определение функции имеет следующий синтаксис:

```
ТипВозвращаемогоЗначения ИмяФункции(СписокФормальныхАргументов)
{
    ТелоФункции;
    ...
    return(ВозвращаемоеЗначение);
}
```

Пример: Функция сложения двух вещественных чисел

```
float function(float x, float z)
{
    float y;
    y=x+z;
    return(y);
}
```

Объявление функций: прототипы функций

```
1    returnDataType functionName( dataType argName1, dataType argName2, ..., dataType argNameN);
```

где,

returnDataType — возвращаемый тип данных

functionName — имя функции

dataType — тип данных

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

Смотрим пример объявления функции:

```
1    // Объявление прототипа функции с двумя целыми параметрами
2    // функция принимает два аргумента и возвращает их сумму
3    int sum(int num1, int num2);
```

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

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

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

Определение функций

```
1  returnType functionName( dataType argName1, dataType argName2, ..., dataType argNameN)
2  {
3      // тело функции
4  }
```

Рассмотрим определение функции на примере функции sum.

```
1  // определение функции, которая суммирует два целых числа и возвращает их сумму
2  int sum(int num1, int num2)
3  {
4      return (num1 + num2);
5  }
```

Типы функций.

Тип функции определяется типом возвращаемого ею значения, а не типом ее аргументов. Если указание типа отсутствует, то по умолчанию считается, что функция имеет тип int. Если значения функции не принадлежат типу int, то необходимо указать ее тип в двух местах.

1. Описать тип функции в ее определении:

```
char pun(ch, n) /* функция возвращает символ */
int n;
char ch;
```

```
float raft(num) /* функция возвращает величину типа float */
int num;
```

2. Описать тип функции также в вызывающей программе.

Описание функции должно быть приведено наряду с описаниями переменных программы; необходимо только указать скобки (но не аргументы) для идентификации данного объекта как функции.

```
main()
{
char, rch, pun();
float raft;
```

Запомните! Если функция возвращает величину не типа int, указывайте тип функции там, где она определяется, и там, где она используется.

Функции типа void.

```
//08VOIDF.C
```

```
/*Программа на C иллюстрирует использование функции типа void.
```

```
Программа печатает двоичное представление числа*/
```

```
#include <iostream>
```

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

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

```
using namespace std;
```

```
void vbinary(int ivalue);
```

```
main()
```

```
{
```

```
int ivalue;
```

```
/* Введите число с основанием 10 для преобразования в двоичное */
```

```
printf("Enter a decimal number to conversion to binary.\n");
```

```
scanf("%d",&ivalue);
```

```
vbinary(ivalue);
```

```
printf ("\n\nPress any key to finish\n");
```

```
_getch();
```

```
return(0);
```

```
}
```

```
void vbinary(int idata)
```

```
{
```

```
int t=0;
```

```
int iyourarray[50];
```

```
while(idata != 0)
```

```
{
```

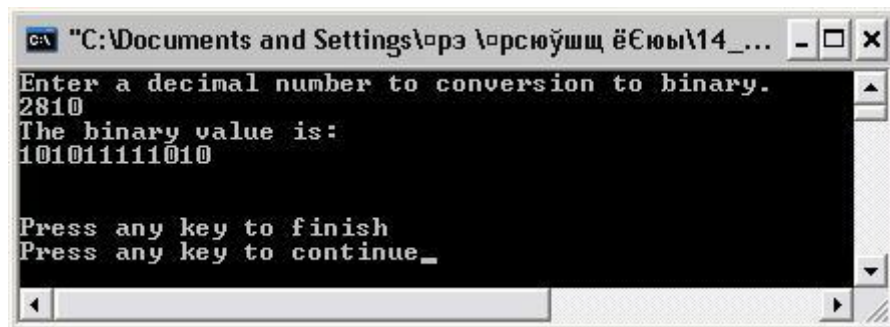
```
iyourarray[t]=(idata %2);
```

```
idata/=2;
```

```

t++;
}
t--;
printf("The binary value is:\n");
for(;t>=0;t--)
printf("%1d",iyourarray[t]);
printf("\n");
}

```



Функции типа char.

Функция `C lowercase()` имеет символьный параметр и возвращает также символьное значение. В данном примере некоторая заглавная буква вводится с клавиатуры и передается в функцию, в которой для преобразования символа в строчную букву используется библиотечная функция `tolower()` (из стандартной библиотеки с прототипом в файле `ctype.h`). Близкие `tolower()` функции: `tolower()` и `toupper()`.

//08CHARF.C

/*Программа на C иллюстрирует использование функции типа char.

Функция получает символ в верхнем регистре и преобразует его в нижний регистр*/

```
#include <iostream>
```

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

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

```
using namespace std;
```

```
char lowercase(char c);
```

```
main()
```

```
{
```

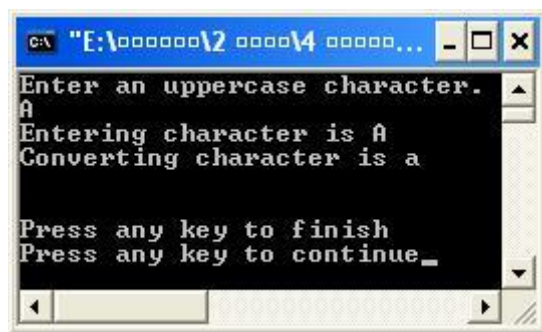
```
char clowchar,chichar;
```

```
/* Введите символ в верхнем регистре */
```

```
printf("Enter an uppercase character.\n");
chichar=getchar();
clowchar=lowercase(chichar);
printf("Entering character is %c\nConverting character is %c\n",chichar,clowchar);
```

```
printf ("\n\nPress any key to finish\n");
_getch();
return(0);
}
```

```
char lowercase(char c)
{
return(tolower(c));
}
```



Функции типа int.

Следующая функция имеет целочисленные параметры и возвращает целые значения. Функция icube() получает из main() некоторое число (0, 2, 4, 6, 8, 10 и так далее), возводит его в куб и возвращает целое значение в main(). Исходное число и его куб выводятся на экран.

```
//08INTF.C
```

```
/*Программа на С иллюстрирует использование функции типа int.
```

```
Функция считывает по очереди целые числа и возвращает
их значения, возведенные в куб*/
```

```
#include <iostream>
```

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

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

```
using namespace std;
```

```
int icube(int ivalue);
```



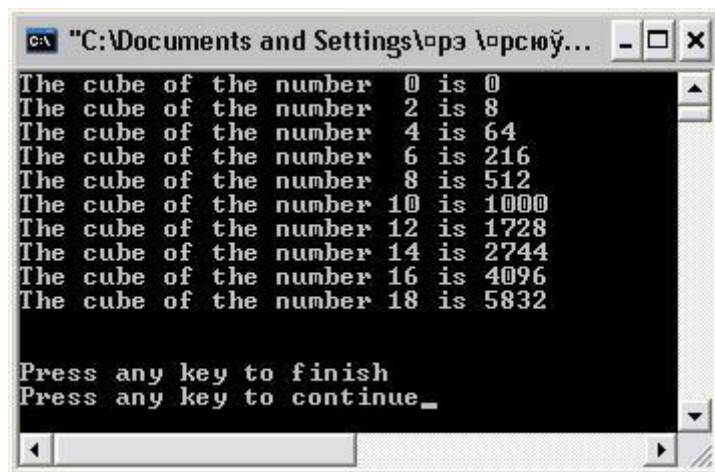
```

main()
{
    int k, inumbercube;
    for(k=0; k<20; k+=2)
    {
        inumbercube=icube(k);
        /* Куб числа ... равен ... */
        printf("The cube of the number %2d is %d\n", k, inumbercube);
    }

    printf ("\n\nPress any key to finish\n");
    _getch();
    return(0);
}

int icube(int ivalue)
{
    return(ivalue*ivalue*ivalue);
}

```



84. Использование функций. Аргументы функции main(). Строки. Целые числа. Числа с плавающей точкой.

Функция — это самостоятельная единица программы, которая спроектирована для реализации конкретной подзадачи.

Функция является подпрограммой, которая может содержаться в основной программе, а может

быть создана отдельно (в библиотеке). Каждая функция выполняет в программе определенные действия.

Сигнатура функции определяет правила использования функции. Обычно сигнатура представляет собой описание функции, включающее имя функции, перечень формальных параметров с их типами и тип возвращаемого значения.

Семантика функции определяет способ реализации функции. Обычно представляет собой тело функции.

Определение функции

Каждая функция в языке Си должна быть определена, то есть должны быть указаны:

- тип возвращаемого значения;
- имя функции;
- информация о формальных аргументах;
- тело функции.

Определение функции имеет следующий синтаксис:

```
ТипВозвращаемогоЗначения ИмяФункции(СписокФормальныхАргументов)
{
    ТелоФункции;
    ...
    return(ВозвращаемоеЗначение);
}
```

Пример: Функция сложения двух вещественных чисел

```
float function(float x, float z)
{
    float y;
    y=x+z;
    return(y);
}
```

Объявление функций: прототипы функций

```
1    returnType functionName( dataType argName1, dataType argName2, ..., dataType argNameN);
```

где,

returnDataType — возвращаемый тип данных

functionName — имя функции

dataType — тип данных

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

Смотрим пример объявления функции:

```
1 // Объявление прототипа функции с двумя целыми параметрами
2 // функция принимает два аргумента и возвращает их сумму
3 int sum(int num1, int num2);
```

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

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

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

Определение функций

```
1 returnDataType functionName( dataType argName1, dataType argName2, ..., dataType argNameN)
2 {
3     // тело функции
4 }
```

Рассмотрим определение функции на примере функции sum.

```
1 // определение функции, которая суммирует два целых числа и возвращает их сумму
2 int sum(int num1, int num2)
3 {
4     return (num1 + num2);
5 }
```

Аргументы функции main().

В обоих языках, C и C++, имеется возможность обработки аргументов командной строки, которые представляют собой параметры, вводимые вместе с именем программы при ее вызове из командной строки операционной системы. Эта возможность позволяет передавать аргументы непосредственно вашей программе без дополнительных запросов из этой программы. Например, пусть некоторая программа получает из командной строки четыре аргумента:

```
YOURPROGRAM Sneakers, Dumbdog, Shadow, Wonderdog
```

Здесь четыре значения передаются из командной строки программе YOURPROGRAM. На самом деле эта информация передается функции main(). Один аргумент, получаемый main(), argc,

представляет собой целое число, определяющее количество элементов командной строки плюс 1.

Начиная с DOS 3.0, название программы считается первым элементом, передаваемым из командной строки. Вторым аргументом — это указатель на массив указателей на строки, называемый `argv`. Так как все элементы являются символьными строками, `argv` имеет тип `char*[argc]`. Поскольку все программы имеют название, `argc` всегда на единицу больше числа аргументов командной строки. В дальнейшем вы познакомитесь с различными способами извлечения разных типов данных из командной строки. Названия аргументов `argc` и `argv` являются общепринятыми именами переменных, используемых во всех программах на C и C++.

При создании консольного приложения в языке программирования C++, автоматически создается строка очень похожая на эту:

```
1 int main(int argc, char* argv[]) // параметры функции main()
```

Эта строка — заголовок главной функции `main()`, в скобках объявлены параметры `argc` и `argv`. Так вот, если программу запускать через командную строку, то существует возможность передать какую-либо информацию этой программе, для этого и существуют параметры `argc` и `argv[]`. Параметр `argc` имеет тип данных `int`, и содержит количество параметров, передаваемых в функцию `main`. Причем `argc` всегда не меньше 1, даже когда мы не передаем никакой информации, так как первым параметром считается имя функции. Параметр `argv[]` это массив указателей на строки. Через командную строку можно передать только данные строкового типа. Указатели и строки — это две большие темы, под которые созданы отдельные разделы. Так вот именно через параметр `argv[]` и передается какая-либо информация. Разработаем программу, которую будем запускать через командную строку Windows, и передавать ей некоторую информацию.

```
1 // argc_argv.cpp: определяет точку входа для консольного приложения.
2
3 #include "stdafx.h"
4 #include <iostream>
5 using namespace std;
6
7 int main(int argc, char* argv[])
8 {
9     if (argc > 1) // если передаем аргументы,
10         то argc будет больше 1 (в зависимости от кол-ва аргументов)
11     {
12         cout << argv[1] << endl; // вывод второй строки из массива указателей
13         на строки (нумерация в строках начинается с 0)
14     } else
15     {
16         cout << "Not arguments" << endl;
17     }
18     system("pause");
19     return 0;
20 }
```

После того как отладили программу, открываем командную строку Windows и перетаскиваем в окно командной строки значок нашей программы, в командной строке отобразится полный путь к программе (но можно прописать путь к программе вручную), после этого можно нажимать **ENTER** и программа запустится (см. Рисунок 1).



Рисунок 1 — Параметры функции main

Так как мы просто запустили программу и не передавали ей никаких аргументов, появилось сообщение **Not arguments**. На рисунке 2 изображён запуск этой же программы через командную строку, но уже с передачей ей аргумента **Open**.



Рисунок 2 — Параметры функции main

Аргументом является слово **Open**, как видно из рисунка, это слово появилось на экране. Передавать можно несколько параметров сразу, отделяя их между собой запятой. Если необходимо передать параметр состоящий из нескольких слов, то их необходимо взять в двойные кавычки, и тогда эти слова будут считаться как один параметр. Например, на рисунке изображён запуск программы, с передачей ей аргумента, состоящего из двух слов — **It work**.

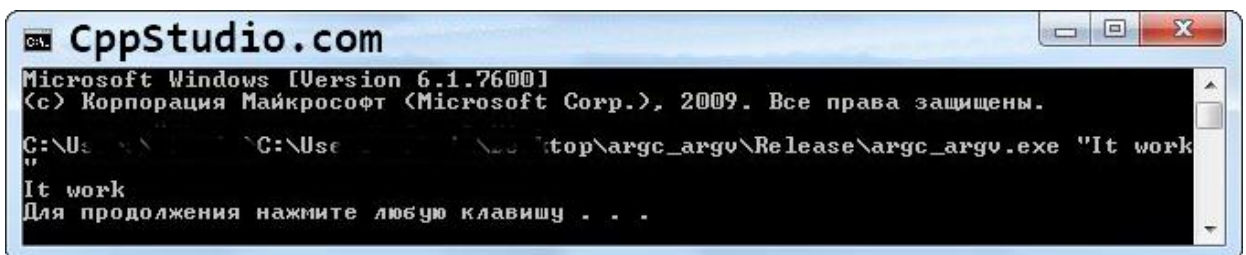


Рисунок 3 — Параметры функции main

А если убрать кавычки. То увидим только слово **It**. Если не планируется передавать какую-либо информацию при запуске программы, то можно удалить аргументы в функции **main()**, также можно менять имена данных аргументов. Иногда встречается модификации параметров **argc** и **argv[]**, но это все зависит от типа создаваемого приложения или от среды разработки.

Строки.

Аргументы командной строки передаются как символьные строки, что облегчает работу с ними.

```
/*08SARGV.C
```

Программа на C иллюстрирует ввод в программу строковых данных

с аргументом командной строки

```
C:\> 08SARGV.EXE 07 Nik Helen*/
```

```
#include "stdafx.h"
```

```
#include <iostream>
```

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

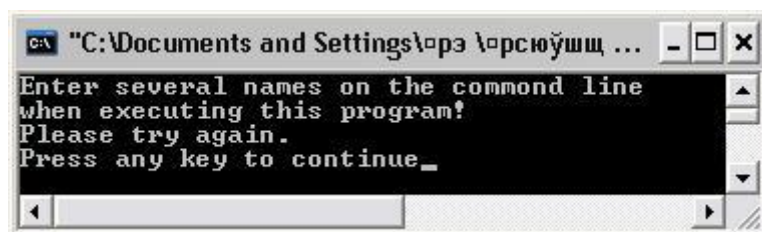
```

#include <stdio.h>
#include <process.h>
#include <ctype.h>
#include <stdlib.h>
#include <math.h>
using namespace std;

int main(int argc, char *argv[])
{
    int t;
    if(argc < 2)
    {
        /* Введите несколько имен в командной строке */
        /* при запуске этой программы! */
        printf("Enter several names on the command line\n");
        printf("when executing this program!\n");
        /* Попробуйте еще раз. */
        printf("Please try again.\n");
        exit(0);
    }
    for(t = 1; t < argc; t++)
        printf("Entry #%d is %s\n", t, argv[t]);

    printf("\n\nPress any key to finish\n");
    _getch();
    return(0);
}

```



Эта программа целиком находится в функции main() и не требует дополнительных функций. Имена, введенные с командной строки, печатаются на экране в том же порядке. Если из командной строки вводятся цифровые значения, то они интерпретируются как ASCII-строки отдельных символов и должны печататься такими как есть.

Целые числа.

Во многих программах желательно иметь возможность ввода из командной строки целых чисел; это может быть, к примеру, программа для вычисления средних оценок студентов. В таких случаях символьная информация в кодах ASCII должна быть преобразована в целые значения. В приведенном ниже примере на C++ из командной строки вводится одно целое число. Поскольку на самом деле это число является символьной строкой, оно преобразуется в целое при помощи библиотечной функции `atoi()`. Значение из командной строки `ivalue` передается в использованную в предыдущем примере функцию `vbinary()`, которая преобразует значение `ivalue` в строку двоичных цифр и печатает ее на экране. Когда управление возвращается функции `main()`, значение `ivalue` печатается в восьмеричном и шестнадцатеричном форматах.

```
// 08IARGV.CPP

// Программа на C++ иллюстрирует ввод в программу целых чисел
// из командной строки
// C:\> 08IARGV.EXE 97


#include <iostream>
#include <conio.h>
#include <stdio.h>
#include <process.h>
#include <ctype.h>
#include <stdlib.h>
#include <math.h>
using namespace std;

void vbinary(int idigits);

main(int argc, char *argv[])
{
    int ivalue;
    if(argc!=2)
    {
        // Введите десятичное число в командной строке.
        // Оно будет преобразовано в двоичное, восьмеричное и
        // шестнадцатеричное.
        cout<<"Enter a decimal number on the command line.\n";
        cout<<"It will be converted to binary, octal and hexadecimal.\n";
        exit(1);
    }
    ivalue=atoi(argv[1]);
```

```

// Десятичное значение равно
cout << "The decimal value is: " << ivalue << endl;
vbinary(ivalue);

// Восьмеричное значение равно
cout << "The octal value is: " << oct << ivalue << endl;

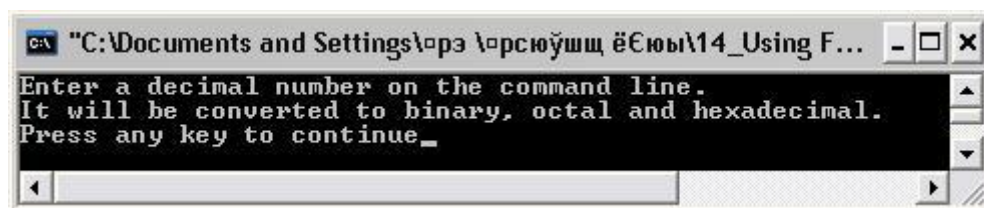
// Шестнадцатеричное значение равно
cout << "The hexadecimal value is: " << hex << ivalue << endl;

printf ("\n\nPress any key to finish\n");
_getch();
return(0);
}

void vbinary(int idigits)
{
int t=0;
int iyourarray[50];
while(idigits != 0)
{
iyourarray[t]=(idigits %2);
idigits/=2;
t++;
}
t--;

// Двоичное значение равно
cout << "The binary value is: ";
for(;t>=0;t--)
cout << dec << iyourarray[t];
cout << endl;
}

```



Числа с плавающей точкой.

Следующий пример на С позволяет ввести из командной строки значения нескольких углов. Вычисляются косинусы углов и печатаются на экране. Поскольку у значений углов тип float, они могут выглядеть по-разному, например: 12.0, 45.78, 0.12345 или 15.

```
/*08FARGV.C
```

Программа на С иллюстрирует ввод в программу чисел

с плавающей точкой из командной строки

```
C:\> 08FARGV.EXE 90 45.78 0.12345 45*/
```

```
#include "stdafx.h"
```

```
#include <iostream>
```

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

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

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

```
#include <ctype.h>
```

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

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

```
using namespace std;
```

```
const double dPi=3.14159265359;
```

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

```
{
```

```
int t;
```

```
double ddegree;
```

```
if(argc < 2)
```

```
{
```

```
/* Введите несколько углов в командной строке. */
```

```
/* Программа вычислит и напечатает */
```

```
/* косинусы введенных углов. */
```

```
printf("Type several angels on the command line.\n");
```

```
printf("Programm will colculate and print\n");
```

```
printf("the cosine af the angles entered.\n");
```

```
exit(1);
```

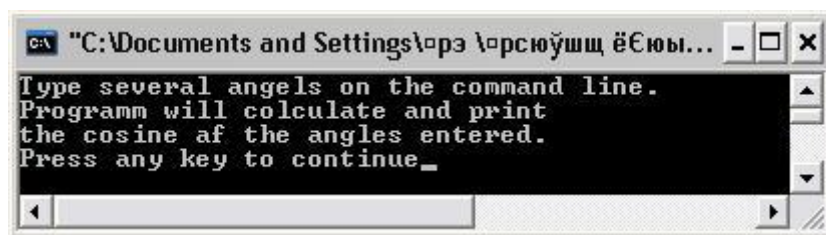
```
}
```

```

for(t=1;t<argc;t++)
{
ddegree=(double) atof(argv[t]);
printf("The cosine of %f is %15.14lf\n",
ddegree, cos((dPi/180)*ddegree));
}

printf ("\n\nPress any key to finish\n");
_getch();
return(0);
}

```



Функция `atof()` используется для преобразования строковых аргументов командной строки в тип `float`. В программе для вычисления косинуса в функции `printf()` используется функция `cos()`.

85. Использование функций. Важные возможности C++. Встраивание (`inline`). Перегрузка (`overloading`). Многоточие (`..`).

Функция — это самостоятельная единица программы, которая спроектирована для реализации конкретной подзадачи.

Функция является подпрограммой, которая может содержаться в основной программе, а может быть создана отдельно (в библиотеке). Каждая функция выполняет в программе определенные действия.

Сигнатура функции определяет правила использования функции. Обычно сигнатура представляет собой описание функции, включающее имя функции, перечень формальных параметров с их типами и тип возвращаемого значения.

Семантика функции определяет способ реализации функции. Обычно представляет собой тело функции.

Определение функции

Каждая функция в языке Си должна быть определена, то есть должны быть указаны:

- тип возвращаемого значения;
- имя функции;
- информация о формальных аргументах;
- тело функции.

Определение функции имеет следующий синтаксис:

```
ТипВозвращаемогоЗначения ИмяФункции(СписокФормальныхАргументов)
{
    ТелоФункции;
    ...
    return(ВозвращаемоеЗначение);
}
```

Пример: Функция сложения двух вещественных чисел

```
float function(float x, float z)
{
    float y;
    y=x+z;
    return(y);
}
```

Объявление функций: прототипы функций

```
1    returnDataType functionName( dataType argName1, dataType argName2, ..., dataType argNameN);
```

где,

returnDataType — возвращаемый тип данных

functionName — имя функции

dataType — тип данных

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

Смотрим пример объявления функции:

```
1    // Объявление прототипа функции с двумя целыми параметрами
2    // функция принимает два аргумента и возвращает их сумму
3    int sum(int num1, int num2);
```

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

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

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

Определение функций

```
1  returnType functionName( dataType argName1, dataType argName2, ..., dataType argNameN)
2  {
3      // тело функции
4  }
```

Рассмотрим определение функции на примере функции sum.

```
1  // определение функции, которая суммирует два целых числа и возвращает их сумму
2  int sum(int num1, int num2)
3  {
4      return (num1 + num2);
5  }
```

Важные возможности C++.

Для создания функций на C++ можно использовать некоторые специфические возможности языка. Одним из его достоинств является возможность написания встроенных функций. Код некоторой встроенной функции воспроизводится в том месте программы, в котором она вызывается. Поскольку компилятор располагает этот код в точке вызова функции, то при использовании коротких, часто вызываемых функций, сокращается время выполнения программы.

Встраивание (inline).

Кроме обычных функций в C++, о которых вы уже знаете, есть еще встроенные функции. Встроенные функции не так значимы, но желательно в них разбираться. Основная идея в том, чтобы ускорить программу ценой занимаемого места. Встроенные функции во многом похожи на заполнитель. После того как вы определите встроенную функцию с помощью ключевого слова `inline`, всякий раз когда вы будете вызывать эту функцию, компилятор будет заменять вызов функции фактическим кодом из функции.

Как это делает программу быстрее? Легко, вызовы функций занимают больше времени, чем написание всего кода без функции. Просмотр вашей программы и замена функции, которую вы использовали 100 раз с кодом из функции, займет очень много времени. Конечно, используя встроенные функции для замены обычных вызовов функций, вы также значительно увеличите размер вашей программы.

Использовать ключевое слово `inline` легко, просто поставьте его перед именем функции. Затем, используйте её как обычную функцию.

Пример встроенной функции

```
1
2     #include <iostream>
3
4     using namespace std;
5
6     inline void hello()
7     {
8         cout<<"hello";
9     }
10    int main()
11    {
12        hello(); //Call it like a normal function...
13        cin.get();
14    }
```

Однако, как только программа будет скомпилирована, вызов `hello();` будет заменен на код функции.

Встроенные функции очень хороши для ускорения программы, но если вы используете их слишком часто или с большими функциями, у вас будет чрезвычайно большая программа. Иногда большие программы менее эффективны, и поэтому они будут работать медленнее, чем раньше. Встроенные функции лучше всего подходят для небольших функций, которые часто вызываются. Наконец, обратите внимание, что компилятор может, по своему желанию, игнорировать ваши попытки сделать функцию встроенной. Так что если вы ошибетесь и сделаете встроенной чудовищную функцию в пятьдесят строк, которая вызывается тысячи раз, компилятор может игнорировать вас.

Ключевое слово `inline` можно рассматривать как директиву или, лучше сказать, как указание компилятору C++ на встраивание этой функции. По разным причинам компилятор может проигнорировать это указание. Например, функция может быть слишком длинной. Встраиваемые функции используются в первую очередь для сокращения времени выполнения программы при частом вызове коротких функций.

```
// Программа на C++ иллюстрирует использование встроенных функций.
// Встроенные функции работают лучше всего в тех случаях, когда часто
// повторяются короткие функции. В этом примере некоторое сообщение
// выводится на экран несколько раз
#include "stdafx.h"
#include <iostream>
#include <conio.h>
#include <stdio.h>
#include <process.h>
#include <ctype.h>
```

```

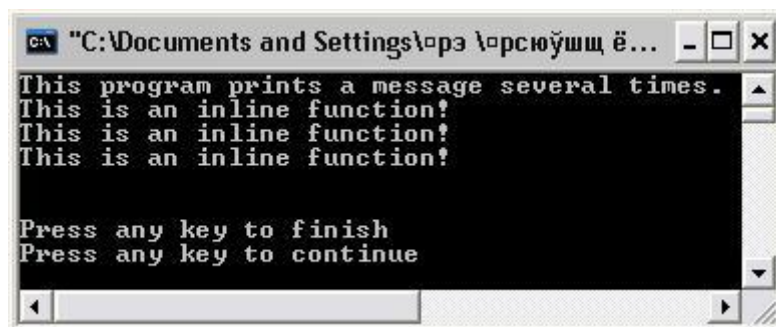
#include <stdlib.h>
#include <math.h>
using namespace std;

// Это - встроенная функция
inline void voutput(void)
{
    cout << "This is an inline function!" << endl;
}

main()
{
    int t;
    // Программа печатает сообщение несколько раз
    cout << "This program prints a message several times." << endl;
    for(t=0;t<3;t++)
        voutput();

    printf ("\n\nPress any key to finish\n");
    _getch();
    return(0);
}

```



Перегрузка (overloading).

Следующий пример иллюстрирует перегрузку функции. Обратите внимание на то, как описаны прототипы двух функций с одинаковым именем и областью действия. Конкретная функция будет выбираться в зависимости от передаваемых аргументов. Функцию `adder()` можно вызывать с параметрами целого типа или типа `float`.

```

// Программа на C++ иллюстрирует перегрузку функций.

```

```

// Перегружаемая функция получает массив целых чисел или чисел с
// плавающей точкой и возвращает их сумму в виде числа int или float
#include "stdafx.h"
#include <iostream>
#include <conio.h>
#include <stdio.h>
#include <process.h>
#include <ctype.h>
#include <stdlib.h>
#include <math.h>
using namespace std;

int adder(int iarray[]);
float adder(float farray[]);

main()
{
int iarray[7]={5,1,6,20,15,0,12};
float farray[7]={3.3,5.2,0.05,1.49,3.12345,31.0,2.007};
int isum;
float fsum;
isum=adder(iarray);
fsum=adder(farray);
// Сумма целых чисел равна
cout << "The sum of the integer numbers is: "
<< isum << endl;
// Сумма чисел с плавающей точкой равна
cout << "The sum of the float numbers is: "
<< fsum << endl;

printf ("\n\nPress any key to finish\n");
_getch();
return(0);
}

int adder(int iarray[])

```

```

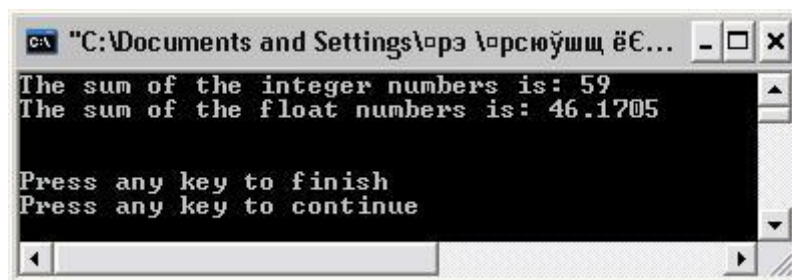
{
int i;
int ipartial;
ipartial=iarray[0];
for (i=1;i<7;i++)
ipartial+=iarray[i];
return (ipartial);
}

```

```

float adder(float farray[])
{
int i;
float fpartial;
fpartial=farray[0];
for (i=1;i<7;i++)
fpartial+=farray[i];
return (fpartial);
}

```



```

C:\Documents and Settings\user\Рабочий стол\...
The sum of the integer numbers is: 59
The sum of the float numbers is: 46.1705

Press any key to finish
Press any key to continue

```

Многоточие (..).

Многоточие используется тогда, когда количество аргументов неизвестно. Неопределенное количество аргументов может быть указано в операторе функции с формальными параметрами. Например:

```
void yourfunction(int t,float u, . ..) ;
```

Такая запись указывает компилятору C, что за обязательными переменными t и u могут следовать, а могут и не следовать другие аргументы. Конечно же, многоточие приостанавливает проверку типов.

86. Использование функций. Области видимости. Локальные и глобальные переменные Сложности в правилах области действия (scope rules). Неопределенные символы в программе на С. Использование переменной с файловой областью действия.

Функция — это самостоятельная единица программы, которая спроектирована для реализации конкретной подзадачи.

Функция является подпрограммой, которая может содержаться в основной программе, а может быть создана отдельно (в библиотеке). Каждая функция выполняет в программе определенные действия.

Сигнатура функции определяет правила использования функции. Обычно сигнатура представляет собой описание функции, включающее имя функции, перечень формальных параметров с их типами и тип возвращаемого значения.

Семантика функции определяет способ реализации функции. Обычно представляет собой тело функции.

Определение функции

Каждая функция в языке Си должна быть определена, то есть должны быть указаны:

- тип возвращаемого значения;
- имя функции;
- информация о формальных аргументах;
- тело функции.

Определение функции имеет следующий синтаксис:

```
ТипВозвращаемогоЗначения ИмяФункции(СписокФормальныхАргументов)
{
    ТелоФункции;
    ...
    return(ВозвращаемоеЗначение);
}
```

Пример: Функция сложения двух вещественных чисел

```
float function(float x, float z)
{
    float y;
    y=x+z;
    return(y);
}
```

Объявление функций: прототипы функций

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

где,

returnDataType — возвращаемый тип данных

functionName — имя функции

dataType — тип данных

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

Смотрим пример объявления функции:

```
1 // Объявление прототипа функции с двумя целыми параметрами
2 // функция принимает два аргумента и возвращает их сумму
3 int sum(int num1, int num2);
```

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

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

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

Определение функций

```
1 returnDataType functionName( dataType argName1, dataType argName2, ..., dataType argNameN)
2 {
3     // тело функции
4 }
```

Рассмотрим определение функции на примере функции sum.

```
1 // определение функции, которая суммирует два целых числа и возвращает их сумму
2 int sum(int num1, int num2)
3 {
4     return (num1 + num2);
5 }
```

Область видимости переменных в C++

Область видимости переменных — это те части программы, в которой пользователь может изменять или использовать переменные в своих нуждах.

В C++ существуют отдельные блоки, которые начинаются с открывающей скобки ({) и заканчиваются соответственно закрывающей скобкой (}). Такими блоками являются циклы (for, while, do while) и функции.

```

int func () {
    // блок (функция - func)
}

int main() {
    // блок (функция - main)
    for (int i = 0; i < 10; i++) {
        // блок (цикл - for), также является дочерним блоком функции main
        for (int j = 0; j < 5; j++) {
            // блок (цикл - for), но он еще является и дочерним блоком для первого цикла
        }
    }

    system("pause");
    return 0;
}

```

Если переменная была создана в таком блоке, то ее **областью видимости** будет являться этот блок от его начала (от открывающей скобки — {) и до его конца (до закрывающей скобки — }) включая все дочерние блоки созданные в этом блоке.

В примере ниже, программист ошибся с областью видимости:

- Он создал переменную j во втором цикле.
- Используя ее в первом цикле for он вынудил компилятор сообщить об ошибке (переменной j больше нет, поскольку второй цикл закончил свою работу).

```

1 int main() {
2
3     for (int i = 0; i < 10; i++) {
4         int b = i;
5         for (int j = 0; j < 5; j++) {
6             cout << b + j;
7         }
8         cout << j; // ошибка: так как переменная j была создана в другом блоке
9     }
10
11     system("pause");
12     return 0;
13 }

```

А вот ошибки в строке 6 нет, поскольку второй цикл находится в первом цикле (является дочерним блоком первого цикла) и поэтому переменная b может спокойно там использоваться.

Глобальные переменные в C++

Глобальными переменными называются те переменные, которые были созданы вне тела какого-то блока. Их можно всегда использовать во всей вашей программе, вплоть до ее окончания работы. В примере ниже мы создали две глобальные переменные global и global_too и использовали их в функции summa:

```

1 int global = 5;    // глобальные
2 int global_too = 10; // переменные
3
4 int summa() {
5     cout << global + global_too; // суммируем числа
6 }
7
8 int main() {
9     summa(); // вызываем функцию summa
10
11     system("pause");
12     return 0;
13 }

```

Вот, что выведет данная программа:

area_sea.cpp

15

Process returned 0 (0x0) execution time : 0.010 s

Press any key to continue.

Как видите глобальные переменные видны везде. В нашем примере внутри функции `summa` мы не создавали ни какие переменные, мы лишь использовали две глобальные переменные, которые были созданы раньше.

Локальные переменные

Локальные переменные — это переменные созданные в блоках. Областью видимости таких переменных является блоки (и все их дочерние), а также их область видимости не распространяется на другие блоки. Как ни как, но эти переменные созданы в отдельных блоках. Из этого можно сделать вывод: у нас есть возможность создавать переменные с одинаковыми именами, но в разных блоках (или другими словами, чтобы их область видимости не совпадала друг с другом).

```

1 int main() {
2     for (int i = 0; i < 2; i++) {
3         int b = i; // локальная переменная (она находится в блоке for)
4         cout << b;
5     }
6     system("pause");
7     return 0;
8 }

```

В примере выше блоком где была создана локальная переменная `b` является цикл `for` (2 — 5). А вот если бы мы захотели вывести переменную `b` вне блока `for`, компилятор сообщил бы нам об ошибке, подробнее об этом говорится ниже.

Распространенной ошибкой среди начинающих программистов является использование локальных переменных в других блоках. Например ниже мы решили использовать переменную `cost` в функции `summa`, хотя мы ее создали в совершенно другой функции — `main`.

```
1 int summa () {
2     cout << cost; // ошибка
3 }
4
5 int main() {
6
7     int cost = 10; // переменная созданная в блоке main
8
9     summa();
10
11     system("pause");
12     return 0;
13 }
```

Нужно запомнить! Если вы создали локальную переменную, то вы должны понимать, что использование ее в других блоках будет невозможным.

Глобальная переменная уступает локальной

Если мы создадим глобальную переменную и с таким же именем локальную, то получится, что там где была создана локальная переменная будет использоваться именно **локальная переменная**, а не глобальная. Так как локальная переменная считается по приоритету выше глобальной. Давайте разберем, как это работает на примере ниже:

```
1 string str = "You lucky!";
2
3 void message() {
4     string str = "You very lucky man!";
5     cout << str;
6 }
7 void sait_message() {
8     cout << str;
9 }
```

Мы создали глобальную переменную `str` со значением «You lucky!» и локальную переменную с таким же названием в функции `message`, но с другим значением — «You very lucky man!». Если мы вызовем функцию `message`, то результатом будет:

message.cpp

You very lucky man!

Process returned 0 (0x0) execution time : 0.010 s

Press any key to continue.

А вот, если мы вызовем функцию `sait_message` то результатом будет:

```
sait_message.cpp
```

```
You lucky!
```

```
Process returned 0 (0x0) execution time : 0.010 s
```

```
Press any key to continue.
```

Вот так глобальная переменная уступает локальной!

Мы советуем вам не создавать переменные с одинаковыми именами, поскольку в будущем вам будет тяжело разобраться в коде, если там будут присутствовать одинаковые переменные.

Сложности в правилах области действия (scope rules).

Если используются переменные с различной областью действия, то можно столкнуться с совершенно неожиданными результатами программирования, называемыми побочными эффектами. Например, как вы уже знаете, может существовать переменная (вернее две переменные с одинаковым именем) как с файловой, так и с локальной областью действия. Правила области действия констатируют, что переменная с локальной областью действия (называемая локальной переменной) имеет приоритет по сравнению с переменной с файловой областью действия (называемой глобальной переменной). Хотя это выглядит достаточно просто, давайте рассмотрим несколько проблем, которые не так очевидны и с которыми вы можете столкнуться при программировании.

Неопределенные символы в программе на С.

В следующем примере четыре переменные имеют локальную область действия внутри функции `main()`. Копии переменных `il` и `im` передаются в функцию `iproduct()`. Это не нарушает правила области действия. Однако, когда функция `iproduct()` пытается обратиться к переменной `in`, она ее не находит. Почему? Потому что область действия этой переменной локальна для функции `main()`.

```
/*08SCOPE.C
```

Программа на С иллюстрирует проблемы, связанные с правилами области действия. Предполагается, что функция вычисляет произведение трех чисел. Компилятор выдает ошибку, поскольку

переменная `in` не известна функции, вычисляющей произведение*/

```
#include "stdafx.h"
```

```
#include <iostream>
```

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

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

```

#include <process.h>
#include <ctype.h>
#include <stdlib.h>
#include <math.h>
#include <stdarg.h>
#include <string.h>
using namespace std;

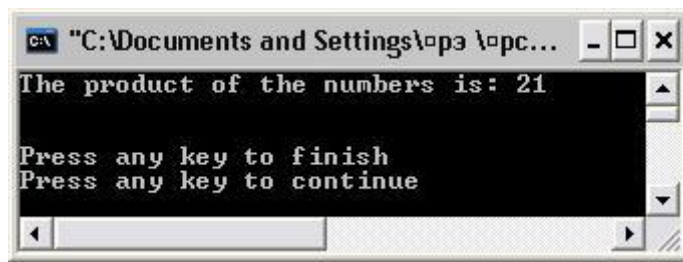
int iproduct(int iw,int ix);
/* int in=10; */

main ()
{
int il=3;
int im=7;
int in=10;
int io;
io=iproduct(il,im);
/* Произведение чисел равно */
printf("The product of the numbers is: %d\n", io);

printf ("\n\nPress any key to finish\n");
_getch();
return(0);
}

int iproduct(int iw,int ix)
{
int iy;
iy=iw*ix;
/* iy=iw*ix*in; */
return (iy);
}

```



Использование переменной с файловой областью действия.

В следующем примере переменной `in` определена файловая область действия. Глобальная видимость переменной `in` во всем файле позволяет ее использовать как функции `main()`, так и функции `iproduct()`. Также надо заметить, что обе функции, `main()` и `iproduct()`, могут изменять значение переменной. Существует хорошее программистское правило: если функции должны быть действительно переносимыми, не следует позволять им менять глобальные переменные.

/*08FScope.C

Программа на C иллюстрирует проблемы, связанные с правилами области действия. Предполагается, что функция вычисляет произведение трех чисел. Предыдущая проблема решена: третьей переменной задана файловая область действия*/

```
#include "stdafx.h"
#include <iostream>
#include <conio.h>
#include <stdio.h>
#include <process.h>
#include <ctype.h>
#include <stdlib.h>
#include <math.h>
#include <stdarg.h>
#include <string.h>
using namespace std;
```

```
int iproduct(int iw,int ix);
int in=10;
```

```
main()
{
    int il=3;
```



```

int im=7;
int io;
io=iproduct(il,im);
printf("The product is: %d\n", io);

printf ("\n\nPress any key to finish\n");
_getch();
return(0);
}

int iproduct(int iw,int ix)
{
int iy;
iy=iw*ix*in;
return(iy);
}

```

Эта программа будет компилироваться без ошибок и напечатает на экране значение произведения 210.



87. Использование функций. Приоритет переменных с файловой и локальной областями действия. Проблемы области действия в C++. Операция уточнения области действия в C++.

Функция — это самостоятельная единица программы, которая спроектирована для реализации конкретной подзадачи.

Функция является подпрограммой, которая может содержаться в основной программе, а может быть создана отдельно (в библиотеке). Каждая функция выполняет в программе определенные действия.

Сигнатура функции определяет правила использования функции. Обычно сигнатура представляет собой описание функции, включающее имя функции, перечень формальных параметров с их типами и тип возвращаемого значения.

Семантика функции определяет способ реализации функции. Обычно представляет собой тело функции.

Определение функции

Каждая функция в языке Си должна быть определена, то есть должны быть указаны:

- тип возвращаемого значения;
- имя функции;
- информация о формальных аргументах;
- тело функции.

Определение функции имеет следующий синтаксис:

```
ТипВозвращаемогоЗначения ИмяФункции(СписокФормальныхАргументов)
{
    ТелоФункции;
    ...
    return(ВозвращаемоеЗначение);
}
```

Пример: Функция сложения двух вещественных чисел

```
float function(float x, float z)
{
    float y;
    y=x+z;
    return(y);
}
```

Объявление функций: прототипы функций

```
1    returnDataType functionName( dataType argName1, dataType argName2, ..., dataType argNameN);
```

где,

returnDataType — возвращаемый тип данных

functionName — имя функции

dataType — тип данных

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

Смотрим пример объявления функции:

```
1    // Объявление прототипа функции с двумя целыми параметрами
2    // функция принимает два аргумента и возвращает их сумму
3    int sum(int num1, int num2);
```

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

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

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

Определение функций

```
1     returnType functionName( dataType argName1, dataType argName2, ..., dataType argNameN)
2     {
3         // тело функции
4     }
```

Рассмотрим определение функции на примере функции sum.

```
1     // определение функции, которая суммирует два целых числа и возвращает их сумму
2     int sum(int num1, int num2)
3     {
4         return (num1 + num2);
5     }
```

Приоритет переменных с файловой и локальной областями действия.

Правила области действия констатируют, что у переменной, имеющей как локальную, так и файловую область действия, используется ее локальное, а не глобальное значение. Это положение иллюстрирует следующая небольшая программа:

```
// 08SCOPE.CPP
/*Программа на С иллюстрирует проблемы, связанные с правилами
области действия. Предполагается, что функция вычисляет
произведение трех чисел, но каких чисел? Две переменные задаются
как параметры функции. Третья переменная имеет и файловую и
локальную область действия*/
#include "stdafx.h"
#include <iostream>
#include <conio.h>
#include <stdio.h>
#include <process.h>
#include <ctype.h>
#include <stdlib.h>
```

```

#include <math.h>
#include <stdarg.h>
#include <string.h>
using namespace std;

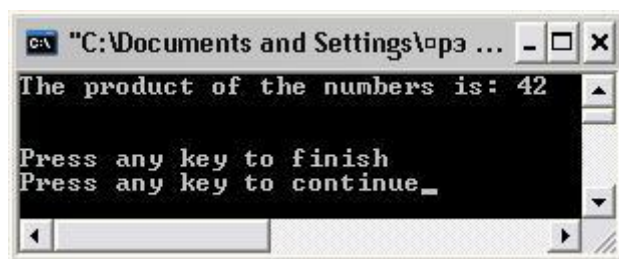
int iproduct(int iw,int ix);
int in=10;

main()
{
int il=3;
int im=7;
int io;
io=iproduct(il,im);
printf("The product of the numbers is: %d\n", io);

printf ("\n\nPress any key to finish\n");
_getch();
return(0);
}

int iproduct(int iw,int ix)
{
int iy;
int in=2;
iy=iw*ix*in;
return(iy);
}

```



Проблемы области действия в C++.

Следующий пример на C++ работает нормально до момента вывода информации на экран. Оператор cout правильно распечатывает значения переменных il и im. Но при обращении к переменной in он выбирает глобальную переменную с файловой областью действия. Результат, выдаваемый программой, $3*7*10 = 42$, является явной ошибкой. Как вы знаете, в подобной ситуации функция iproduct() использует локальное значение переменной in.

```
// 08SCOPE.CPP
// Программа на C++ иллюстрирует проблемы, связанные с правилами
// области действия. Предполагается, что функция вычисляет
// произведение трех чисел. Переменная in имеет локальную область
// действия и используется в функции для вычисления произведения.
// Однако, в главной функции определено, что значение этой переменной
// равно 10. Что здесь неправильно?

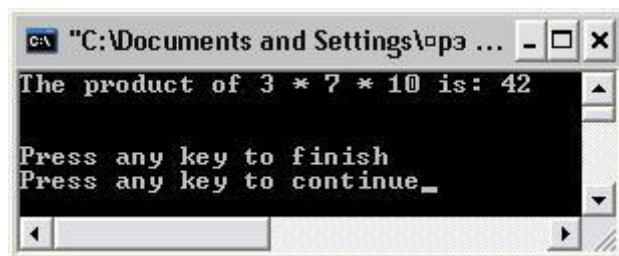
#include "stdafx.h"
#include <iostream>
#include <conio.h>
#include <stdio.h>
#include <process.h>
#include <ctype.h>
#include <stdlib.h>
#include <math.h>
#include <stdarg.h>
#include <string.h>
using namespace std;

int iproduct(int iw,int ix);
int in=10;

main()
{
    int il=3;
    int im=7;
    int io;
    io=iproduct(il,im);
    cout << "The product of " << il <<" * " << im << " * " << in << " is: " << io << endl;
```

```
printf ("\n\nPress any key to finish\n");
_getch();
return(0);
}
```

```
int iproduct(int iw,int ix)
{
int iy;
int in=2;
iy=iw*ix*in;
return (iy);
}
```



Операция уточнения области действия в C++.

В следующем примере для того, чтобы избежать конфликтов между переменными как с файловой, так и с локальной областями действия, используется операция уточнения области действия (scope resolution operator ::). Предыдущая программа выдавала неправильное значение произведения, так как в вычислениях использовалось локальное значение. Обратите внимание на то, как в приведенном листинге в функции iproduct() используется операция уточнения области действия.

```
// 08GSCOPE.CPP
// Программа на C++ иллюстрирует проблемы, связанные с правилами
// области действия, и использование операции уточнения области
// действия. При вычислении произведения с помощью этой операции,
// в функции используется переменная с файловой областью действия

#include "stdafx.h"
#include <iostream>
#include <conio.h>
#include <stdio.h>
#include <process.h>
```

```

#include <ctype.h>
#include <stdlib.h>
#include <math.h>
#include <stdarg.h>
#include <string.h>
using namespace std;

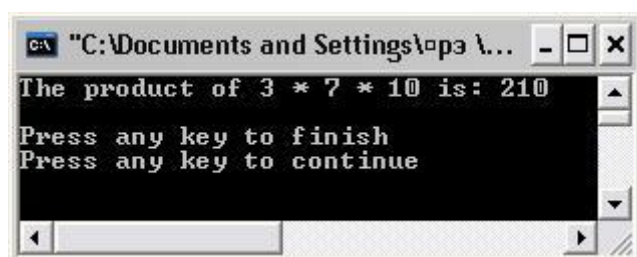
int iproduct(int iw,int ix);
int in=10;

main()
{
int il=3;
int im=7;
int io;
io=iproduct(il,im);
cout << "The product of " << il <<" * " << im
<< " * " << in << " is: " << io;

printf ("\n\nPress any key to finish\n");
_getch();
return(0);
}

int iproduct(int iw,int ix)
{
int iy;
int in=2;
iy=iw*ix*(::in);
return(iy);
}

```



88. Использование функций. Математические функции. Функции округления. Компиляция программ, состоящих из двух или более функций.

Функция — это самостоятельная единица программы, которая спроектирована для реализации конкретной подзадачи.

Функция является подпрограммой, которая может содержаться в основной программе, а может быть создана отдельно (в библиотеке). Каждая функция выполняет в программе определенные действия.

Сигнатура функции определяет правила использования функции. Обычно сигнатура представляет собой описание функции, включающее имя функции, перечень формальных параметров с их типами и тип возвращаемого значения.

Семантика функции определяет способ реализации функции. Обычно представляет собой тело функции.

Определение функции

Каждая функция в языке Си должна быть определена, то есть должны быть указаны:

- тип возвращаемого значения;
- имя функции;
- информация о формальных аргументах;
- тело функции.

Определение функции имеет следующий синтаксис:

```
ТипВозвращаемогоЗначения ИмяФункции(СписокФормальныхАргументов)
{
    ТелоФункции;
    ...
    return(ВозвращаемоеЗначение);
}
```

Пример: Функция сложения двух вещественных чисел

```
float function(float x, float z)
{
    float y;
    y=x+z;
    return(y);
}
```

Объявление функций: прототипы функций

```
1    returnDataType functionName( dataType argName1, dataType argName2, ..., dataType argNameN);
```

где,

returnDataType — возвращаемый тип данных

functionName — имя функции

dataType — тип данных

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

Смотрим пример объявления функции:

```
1 // Объявление прототипа функции с двумя целыми параметрами
2 // функция принимает два аргумента и возвращает их сумму
3 int sum(int num1, int num2);
```

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

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

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

Определение функций

```
1 returnDataType functionName( dataType argName1, dataType argName2, ..., dataType argNameN)
2 {
3     // тело функции
4 }
```

Рассмотрим определение функции на примере функции sum.

```
1 // определение функции, которая суммирует два целых числа и возвращает их сумму
2 int sum(int num1, int num2)
3 {
4     return (num1 + num2);
5 }
```

Математические функции

Математические функции хранятся в стандартной библиотеке math.h. Аргументы большинства математических функций имеют тип double. Возвращаемое значение также имеет тип double. Углы в тригонометрических функциях задаются в радианах.

Основные математические функции стандартной библиотеки.

Функция	Описание
int abs(int x)	Модуль целого числа x
double acos(double x)	Арккосинус x
double asin(double x)	Арсинус x
double atan(double x)	Арктангенс x
double cos(double x)	Косинус x
double cosh(double x)	Косинус гиперболический x
double exp(double x)	Экспонента x
double fabs(double x)	Модуль вещественного числа
double fmod(double x, double y)	Остаток от деления x/y
double log(double x)	Натуральный логарифм x
double log10(double x)	Десятичный логарифм x
double pow(double x, double y)	x в степени y
double sin(double x)	Синус x
double sinh(double x)	Синус гиперболический x
double sqrt(double x)	Квадратный корень x
double tan(double x)	Тангенс x
double tanh(double x)	Тангенс гиперболический x

Функции округления

round, roundf, roundl – округление до ближайшего целого

Синтаксис:

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

```
double round (double x);
```

```
float roundf (float x);
```

```
long double roundl (long double x);
```

Аргументы:

x – число, которое необходимо округлить.

Возвращаемое значение:

Округленный аргумент.

Описание:

Функции округляют аргумент x до ближайшего целого числа. Если округляемый аргумент отстоит от наибольшего и наименьшего целого на одну и ту же величину, то округления произведется до ближайшего большего целого числа. Аргумент и возвращаемое значение функций являются значениями с плавающей точкой.

Причем в функции `roundf` аргумент для расчета и возвращаемое значение задаются числами с плавающей точкой (тип `float`, точность не менее шести значащих десятичных цифр, разрядность - 32).

В функции `round` аргумент и возвращаемое значение задаются числами с плавающей точкой двойной точности (тип `double`, точность не менее десяти значащих десятичных цифр, разрядность - 64).

В функции `roundl` аргумент и возвращаемое значение задаются числами с плавающей точкой повышенной точности (тип `long double`, точность не менее десяти значащих десятичных цифр, разрядность - 80).

Пример:

В примере число 2.83 округляется с помощью функций `round`, `roundf`, `roundl` и результат выводится на консоль.

```
#include <stdio.h> //Для printf
#include <math.h> //Для round, roundf, roundl

int main (void)
{
    //Вывод значения аргумента
    printf ("Аргумент: 2.83\n");
    //Расчет и вывод результата работы функции roundf
    printf ("roundf : %.1f\n", roundf (2.83) );
    //Расчет и вывод результата работы функции round
    printf ("round : %.1f\n", round (2.83) );
    //Расчет и вывод результата работы функции roundl
    printf ("roundl : %.1Lf\n", roundl (2.83) );

    return 0;
}
```

Результат:

```
Аргумент: 2.83  
roundf : 3.0  
round  : 3.0  
roundl  : 3.0
```

Компиляция программ, состоящих из двух или более функций.

Простейший способ использования нескольких функций в одной программе заключается в том, чтобы поместить их в один файл, после чего осуществить компиляцию программы, содержащейся в этом файле так, как будто она состояла из одной функции

Второй способ заключается в применении директивы `#include` Если одна функция содержится в файле с именем `file1.c`, а вторая в файле `file2.c`, поместите эту директиву в файл `file1.c`