МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ

НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Физический факультет
Кафедра автоматизации физико-технических исследований

К. Ф. ЛЫСАКОВ

ИНФОРМАТИКА И ПРОГРАММИРОВАНИЕ (C++)

Учебное пособие

Новосибирск
2013

Лысаков К. Ф. Информатика и программирование (C++): Учеб. пособие / Новосиб. гос. ун-т. Новосибирск, 2013. 181 с.

Дисциплина «Программирование и информатика (C++)» рассчитан на иностранных студентов, для которых программирование не является профилирующим предметом и читается для студентов 2 курса ФФ Китайско-Российского института (КРИ). Целью курса является получение базовых знаний о принципах программирования, обучению программному конструированию на языке C++.

В рамках курса «Программирование и информатика (C++)» рассмотрены основы структурного и объектно-ориентированного программирования. При этом основной упор делается на алгоритмическое решение задачи, а программная реализация рассматривается лишь как завершающий этап. Это позволяет рассматривать программирование как инструмент, а основной задачей ставить понимание алгоритмического решения задачи.

В пособии рассмотрены основы структурного и объектно-ориентированного подхода в программировании на примере языка C++. При этом основной упор делается на правильную постановку задачи, ее анализ и алгоритмическое решение, а программная реализация рассматривается лишь как завершающий этап.

Пособие ориентированно на иностранных студентов, слабо владеющих русским и английским языками.

Учебное пособие подготовлено в рамках реализации программы развития НИУ-НГУ на 2009-2018 гг.

ОГЛАВЛЕНИЕ

1. ВВЕДЕНИЕ

1.1. Понятие программирования

Программирование — это процесс создания (разработки) программы, который может быть представлен последовательностью следующих шагов:

1. Спецификация (определение, формулирование требований к программе).
2. Разработка алгоритма.
3. Кодирование (запись алгоритма на языке программирования).
4. Отладка.
5. Тестирование.

В большинстве своем, различные самоучители и другие печатные издания, во главу угла ставят изучение какого-либо языка программирования, на примере которого решают задачи. При этом, для людей впервые встречающихся с программированием, наибольшую сложность представляют именно два первых пункта.

В данном учебном пособии, основной упор делается именно на постановку задачи и на ее алгоритмическое решение. Если алгоритмическое решение составлено верно, то кодирование может происходить на любом языке программирования.

В качестве языка программирования, в пособии рассмотрен симбиоз языков C и C++, которые позволяет описывать программные реализации значительно проще чем классический язык C, но при этом не использованы основы объектно-ориентированного программирования (классы, полиморфизм, перегрузка и наследование).

1.2. Описание алгоритмов решения

Для решения любой задачи, необходимо выполнить следующие этапы:

1. Четко определить условия задачи, входные данные и какой результат должен быть получено после решения задачи.
2. Какие дополнительные данные необходимы для решения задачи.
3. Составить блок-схему решения задачи и записать ее в виду удобного описания.
4. Анализ всех возможных проблем и усовершенствование алгоритма.

Рассмотрим на примере, каким образом выполняются перечисленные этапы при решении задачи вычисления корней квадратного уравнения.

1.2.1. Этап первый: постановка задачи

Для определенности будем решать уравнение следующего вида:

$$ax^2 + bx + c = 0$$

Таким образом, входными данными для нашей задачи являются три коэффициента: a, b и c.

Решение задачи предполагает вычисление возможных корней уравнения. При этом, так как количество корней возможно от 0 до 2, то необходимо в качестве решения указать количество корней и собственно перечислить их.

1.2.2. Этап второй: дополнительные данные

При решении квадратного уравнения необходимо вычислить значение дискриминанта. В нашей задаче, дискриминант является промежуточным результатом, необходимым для решения задачи.

1.2.3. Этап третий: составление блок-схемы решения

Описание блок-схемы может производиться с помощью различных средств и обозначений. Основной принцип заключается в наглядности шагов исполнения и однозначности переходов при ветвлении.

На рисунке (рис. 1) приведена основная блок-схема, к которой чаще всего приходят студенты при решении квадратного уравнения.

Рис. 1. Блок-схема решения квадратного уравнения

1.2.4. Этап четвертый: корректность и усовершенствования

Данный этап предполагает анализ созданной блок-схемы решения задачи. При этом необходимо ответить на два вопроса:

1. Будет ли схема корректно работать во всем диапазоне входных параметров?
2. Возможны ли оптимизации, которые позволят ускорить процесс выполнения задачи?

При анализе совершаемых действий, очевидно, что при значении коэффициента **a = 0**, происходит деление на **0!** В качестве решения данной ситуации можно предложить два основных метода:

- Добавить проверку корректности введенных пользователем значений, и при вводе **a=0**, выдавать сообщение об ошибке: «Уравнение не является квадратным!».
- Допустить возможность решения линейных уравнений, по сути, расширив диапазон применения вашей реализации. Для этого необходимо добавить такую проверку до вычисления дискриминанта, и идти описанным путем лишь при коэффициенте **a** отличном от 0, иначе добавить еще одну ветвь исполнения.

Что касается оптимизации, то основной ее смысл в удалении лишний действий, а также в недопустимости совершения дублирующий действий. На приведенной выше схеме, дублированным действием является извлечение квадратного корня из дискриминанта.

1.3. Отладка программной реализации

После создания программной реализации, нередки случаи когда ее функционирование отличается от запланированного. Для выяснения причин некорректного поведения существуют два основных метода отладки:

- метод отладочной печати,
- метод пошаговой отладки.

1.3.1. Метод отладочной печати

Одним из основных средств отладки является отладочная печать, позволяющая получить данные о ходе и состоянии процесса вычислений. Обычно разрабатываются специальные отладочные методы, вызываемые в критических точках программы - на входе и выходе программных модулей, на входе и выходе циклов и так далее. Искусство отладки в том и состоит, чтобы получить нужную информацию о прячущихся ошибках, проявляющихся, возможно, только в редких ситуациях.

Иногда пошаговая отладка невозможна. Например, программа может быть связана с внешним процессом, который не будет ждать, пока разработчик проверяет значения переменных. Другой весьма вероятный вариант — программа скомпилирована без отладочной информации с оптимизацией. В таких случаях единственным способом узнать, что происходит во время исполнения, остаётся вывод сообщений, по которым можно судить о состоянии программы.

Суть метода заключается в том, что программист вставляет в код программы вывод определенных сообщений, по которым впоследствии можно проанализировать какая именно ветвь программы исполняется, какие получаются промежуточные результаты и сравнить это с предполагаемым ходом выполнения и вычислениями.

1.3.2. Метод пошаговой отладки

Метод пошаговой отладки, заключается в том, что программист заставляет компьютер выполнять программу по шагам, инструкцию за инструкцией, а сам следит за состоянием программы на каждом шаге. При поиске ошибки очень удобно видеть, какая строка программы сейчас

выполняется, и выполнять программу строка за строкой в соответствии со строками исходного кода.

Команды управления отладкой собраны в меню Debug. Для удобства разработчика на самые нужные команды назначены клавиши.

Программа запускается в отладочном режиме командой Debug (при стандартных настройках клавиша F5). При этом за работой программы следит отладчик, переводящий исполнение в пошаговый режим в двух случаях: при возникновении ошибки, которая в обычном режиме привела бы к аварийному завершению программы, или по достижении точки останова.

Точка останова (англ. ***Breakpoint***), может быть установлена в любой строке программы, содержащей выполняемую инструкцию. Это может быть начало цикла, вызов функции, выражение или даже фигурная скобка, закрывающая блок. При стандартных настройках точка останова устанавливается или снимается нажатием клавиши F9.

После того как отладчик перевёл исполнение в пошаговый режим, можно заняться непосредственно отладкой.

Для того чтобы увидеть значение переменной, существующей в текущем контексте, достаточно навести указатель мыши на её идентификатор в исходном тексте. Этой же цели служит окно ***Watch***: впишите в левую колонку интересующие вас идентификаторы и их значения будут показаны всё время, пока идентификаторы имеют смысл.

Две команды пошагового исполнения позволят вам проконтролировать работу интересующего вас фрагмента кода:

- ***Step Over*** выполняет очередную строку и останавливает программу на следующей строке. Если текущая строка содержит вызов функции, то данная команда выполнит этот вызов в обычном режиме, не показывая по шагам подробности внутренней работы вызываемой функции.

- ***Step Into*** заставляет отладчик войти в функцию, вызов которой находится в очередной строке. Когда интересующий вас фрагмент пройден, можно выйти из пошагового режима командой ***Debug*** (***F5***). Исполнение программы продолжится до следующей точки останова или критической ошибки.

1.4. Работа с проектами в Microsoft Visual C++

В данном пособии в качестве среды разработке рассмотрена MSVS (Microsoft Visual Studio), как наиболее дружелюбная системы для разработки программ на языках C и C++.

1.4.1. Общие сведения

Для того, чтобы писать программу, вам *необходимы* проект (*project*) и рабочее пространство (*solution*).

Проект — это специальный файл, имя которого имеет расширение *vcproj*. В проекте содержится информация о том, из каких исходных файлов строится программа. Всё, что перечислено в проекте, будет скомпоновано в один целевой модуль (обычно исполняемый файл — **.exe*, но есть и другие варианты); если вам нужно получить два исполняемых файла, придётся сделать несколько проектов.

Среда разработки следит за изменениями файлов, перечисленных в файле проекта, и при сборке проекта компилирует заново только файлы, изменённые после предыдущей компиляции.

Компилятор обрабатывает только те файлы, которые перечислены в файле проекта. При этом совершенно всё равно, как называется папка в проекте — *Source files* или *Include files*: инструмент для обработки файла выбирается, исходя из расширения имени файла, а папки вы можете создавать для собственного удобства в любом количестве.

Рабочее пространство — это специальный файл, имеющий расширение *sln*, который описывает зависимости между отдельными

проектами, входящими в рабочее пространство: разрабатывая большую программу, состоящую из ряда модулей, удобно собрать несколько проектов в одно пространство и получать последнюю версию всех модулей нажатием одной клавиши!

Visual Studio устроена так, что проект обязательно должен содержаться внутри рабочего пространства. Если у вас уже есть рабочее пространство, то открывать в *Visual Studio* нужно его, а не проект или файл с исходным текстом!

Файл рабочего пространства создаётся автоматически, когда вы создаёт новый проект.

1.4.2. Создание проекта

Visual Studio предоставляет большое количество шаблонов проектов для разных целей. Полный список шаблонов, сгруппированных по категориям, можно увидеть, выбрав в меню:

File->New->Project... (рис. 2).

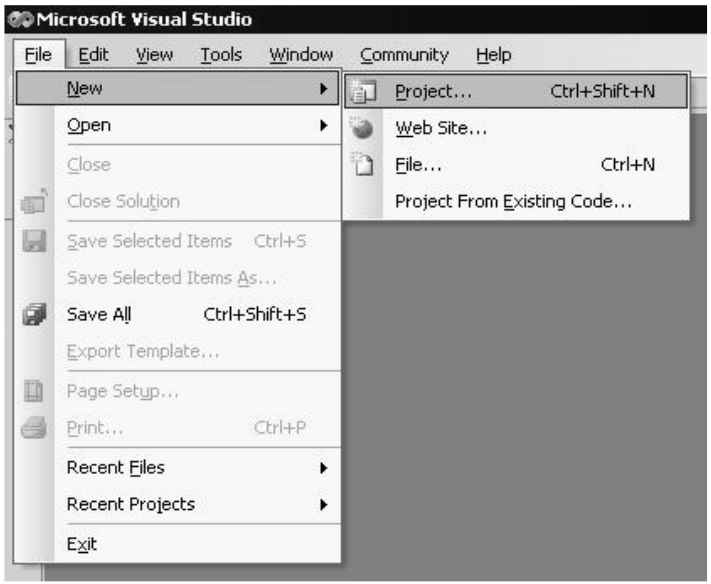


Рис. 2. Создание нового проекта

В зависимости от выбранного шаблона, среда добавляет во вновь создаваемый проект те или иные файлы, выполняя за разработчика часть рутинной работы по первоначальной настройке рабочего окружения.

Для большинства учебных заданий идеальным стартом является пустой проект консольного приложения. Для того чтобы создать такой проект, нужно выбрать подкатегорию *General* в категории проектов *Visual C++*. В окне выбора шаблона выбрать пустой проект *Empty project* (рис. 3).

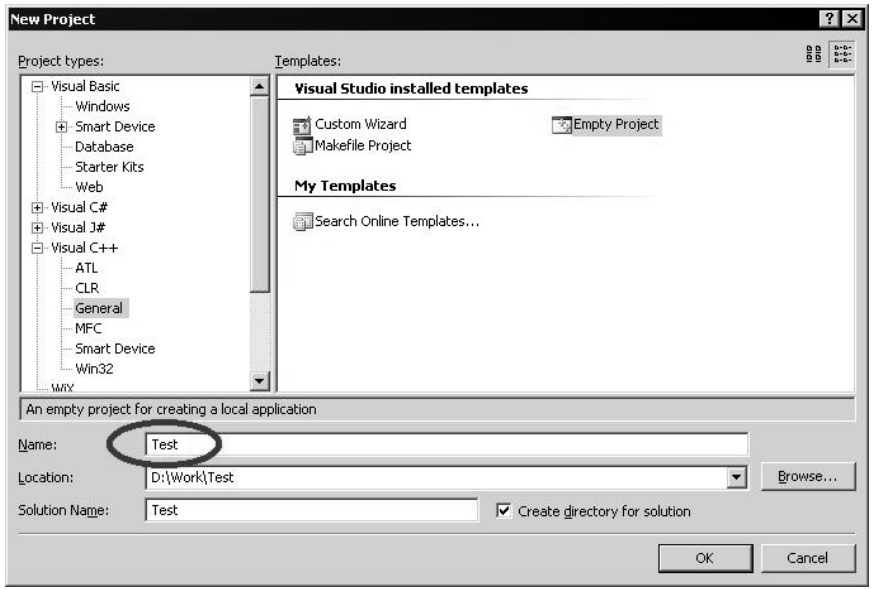


Рис. 3. Выбор пустого проекта

Новому проекту нужно дать имя. Правила для имён проектов такие же, как для имён файлов, но с русскими буквами или пробелами могут возникнуть проблемы. Имя проекта вводится тут же в поле Name (обведено на рисунке). После этого можно нажать OK, и Visual Studio создаст файлы рабочего пространства и проекта. Рабочее пространство получит то же имя, что и проект.

Ниже (рис. 4) показано, как выглядит новое рабочее пространство после добавления пустого проекта. Окно, отображающее структуру рабочего пространства, называется Solution Explorer, его можно открыть через меню View->Solution Explorer.

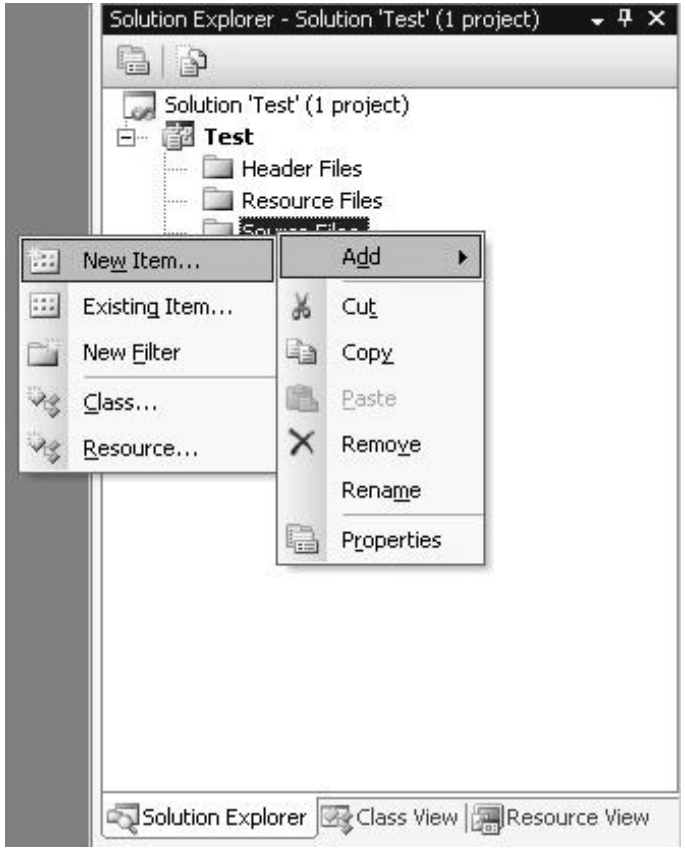


Рис. 4. Добавление файлов в проект

Как видно, проект *Test* не содержит никаких исходных текстов; для того чтобы начать программировать, нужно добавить в проект файл исходного кода. Откройте панель структуры проекта (*Solution Explorer*),

щёлкните правой кнопкой мыши на папке *Source Files* и выберите в меню *Add->New Item...*, затем выберите категорию *Code*, в ней выберите *C++ File* и впишите имя нового файла (рис. 5).

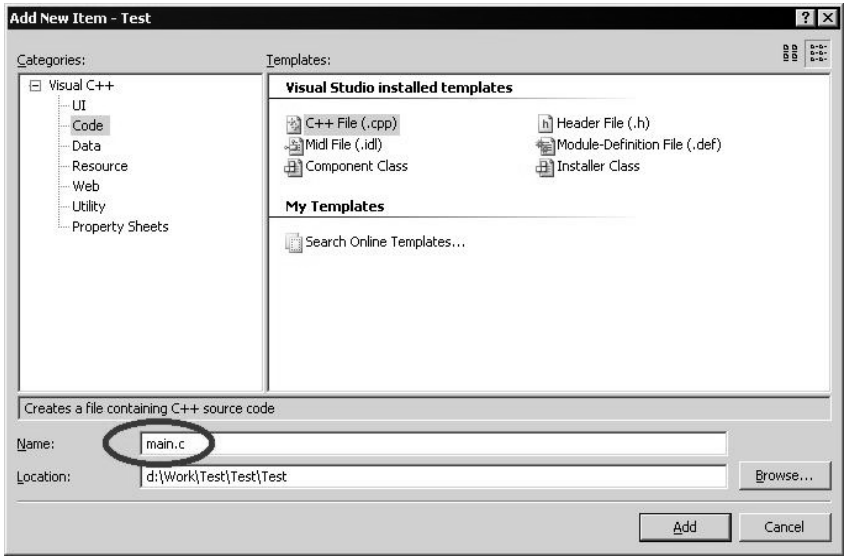


Рис. 5. Добавление файла программы на языке C

Важно явно указать расширение имени: «*.c*» для программы на *C* и «*.cpp*» для программы на *C++* (обведено на рис. 4). Если расширение не указано, *Visual Studio* автоматически добавит расширение «*.cpp*»; такой файл будет обрабатываться компилятором *C++*, что может помешать, если вы пишете программу на *C*.

Введите имя нового файла, нажмите *Add*, и вы увидите, что в списке файлов проекта появился новый элемент. Файл автоматически откроется в редакторе — можно начинать программировать (рис. 6).

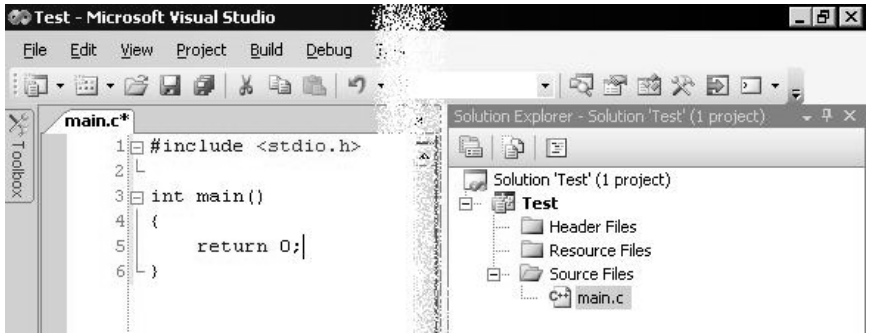


Рис. 6. Вид рабочего проекта

1.4.3. Сборка проекта

После того как программист написал некоторое количество исходного кода, у него обычно возникает желание посмотреть, как этот код работает. Для этого нужно скомпилировать и скомпоновать программу.

Компилятор и компоновщик запускаются из меню *Build->Build Solution* (или нажатием соответствующей «горячей клавиши»). Кроме того, можно скомпилировать только тот файл, который вы в данный момент редактируете (*Build->Compile*), однако в этом случае запустить исполнение кода не получится: для этого необходимо скомпилировать и скомпоновать весь проект.

Если всё получилось (в коде нет ошибок), то результатом работы компилятора и компоновщика будет исполняемый файл (с расширением *exe*), который можно будет запустить. Запускать можно прямо из среды разработки: выбор *Debug->Start Without Debugging* запускает программу в виде самостоятельного процесса, *Debug->Start Debugging* запускает программу в режиме отладки.

1.4.4. Конфигурации проектов

В реальной жизни разработчику часто бывает необходимо иметь несколько вариантов одной и той же программы: одну — с отладочной информацией и без оптимизации, для рабочей отладки, другую — без

отладочной информации, но удобную для тестирования (например, предварительно настроенную на тестовые данные), третью — оптимизированную, готовую для демонстрации заказчику и т. д.

Очевидно, все эти версии программы должны строиться из одних и тех же исходных файлов, иначе работа программиста превратится в кошмар. Возможность настроить различные варианты сборки программы дают *конфигурации*.

Конфигурация — это набор настроек, которому дано имя. Конфигурация определяется в рабочем пространстве, но распространяет влияние также и на все проекты, входящие в пространство. На уровне рабочего пространства конфигурация определяет, во-первых, какие проекты необходимо собирать, а во-вторых, куда помещать основной вывод (*Primary Output*) этих проектов. На уровне проекта можно для каждой возможной конфигурации задать полный набор настроек.

По умолчанию новое рабочее пространство имеет две конфигурации: *Debug* и *Release*. Настройки этих конфигураций соответствуют названиям: *Debug* предназначена для сборки отладочной версии, *Release* — для рабочей. Соответственно отладочная версия содержит полную отладочную информацию, код не оптимизируется компилятором, объявлена директива препроцессора *_DEBUG*. В рабочей версии всё наоборот: код оптимизируется, отладочная информация не генерируется, объявлена директива препроцессора *_RELEASE*.

1.4.5. Файловая структура рабочего пространства

Когда вы создаёте новый проект с именем, например, *sample*, *Visual Studio* создаёт следующие директории и файлы:

- [sample] — корневая директория рабочего пространства. Всё, что относится к рабочему пространству и вложенным в него проектам, создаётся и содержится в этой директории, если ничего не менять в настройках. Все директории и файлы, перечисленные ниже, создаются в корневой директории;

- [sample \ debug] — директория, в которой создаются результаты сборки проектов, входящих в рабочее пространство *sample*, при выборе конфигурации *debug*. Как правило, это основной вывод (*Primary Output*) проектов — исполняемые файлы (**.exe)*, базы данных для отладки (**.pdb*) и некоторые другие файлы. После того как вы закрыли среду разработки, можно удалить эту директорию со всем содержимым, потому что она будет создана заново при следующей сборке проектов, входящих в рабочее пространство;
- [sample \ release] — аналогично [sample \ debug], но для конфигурации *release*;
- Sample \ sample.sln — рабочее пространство *sample*;
- Sample \ sample.ncb, sample \ sample.suo — служебные файлы Visual Studio, относящиеся к рабочему пространству *sample*. Эти два файла генерируются средой разработки, и их тоже можно удалить, если свободного места на диске остро не хватает, закрыв предварительно соответствующее рабочее пространство;
- [sample \ sample] — корневая директория проекта *sample*, входящего в рабочее пространство *sample*. Все файлы, относящиеся к проекту *sample*, по умолчанию создаются в этой директории;
  - [sample \ sample\debug] — директория, в которой создаются все промежуточные файлы при сборке проекта *sample* в конфигурации *debug*. После того как вы закрыли среду разработки, можно удалить эту директорию со всем содержимым, потому что она будет создана заново при следующей сборке проекта;
  - [*sample* \ *sample* \ *release*] — аналогично предыдущему, только для конфигурации *release*;
  - [*Sample* \ *sample* \ *sample.vcproj*] — файл проекта *sample*, входящего в рабочее пространство *sample*.

Все файлы с исходными текстами (*.*c*, *.*cpp*) и заголовочные файлы (*.*h*) среда разработки тоже предлагает создавать в корневой директории соответствующего проекта.

1.5. Использование справочной системы

Справочная система — без преувеличения главный помощник программиста. В ней описаны стандартные функции, классы, интерфейсы и библиотеки. Кроме того, справочная система содержит весьма подробное описание языка программирования, включающее сведения о синтаксисе, типах данных, приоритетах операций и т. п.

Справку можно вызвать двумя способами: через меню *Help->Index* или непосредственно из текстового редактора: по любому слову в программе, будь то идентификатор стандартной функции, ключевое слово языка или директива препроцессора, установив курсор в середине слова и нажав *F1*.

Для ускорения загрузки рекомендуется выбирать локальные файлы справки.

Обычно справочная система предлагает несколько статей. Многие стандартные функции имеют различные реализации для разных платформ, поэтому, если в настройках не указаны предпочтения, вам предложат выбор доступных вариантов. Для функций стандартной библиотеки *C* обычно следует выбирать вариант *C Runtime Library*.

В большинстве статей, описывающих стандартные функции, приводится пример использования (обычно в конце статьи).

2. БАЗОВЫЕ КОНСТРУКЦИИ ЯЗЫКОВ С И С++

Данная глава посвящена описанию основных принципов построения программ, включая не только функциональное описание программной реализации, но и стиль написания.

2.1. Потоки ввода / вывода

Поток — это абстракция источника (поток ввода) или назначения (поток вывода) при последовательной передаче данных. Посредством различных реализаций потоков программа может взаимодействовать с файлами на жёстком диске, клавиатурой, экраном, внешними устройствами и даже памятью, используя унифицированный набор операций.

Для того чтобы пользоваться стандартными потоками, необходимо указать библиотеку, в которой находится реализация этих потоков. Такая библиотека имеет название **iostream** (сокрашщение от InputOutputStream – потоки ввода-вывода) и включается в текст программы следующим образом:

```
#include <iostream>
```

Здесь ***#include*** – служебная команда, означающая, что необходимо к тексту описываемой ниже программы включить указанную в угловых скобках библиотеку.

После включения таким образом библиотеки, пользователь может использовать стандартный ввод-вывод следующим образом:

```
std::cout << "Hello"
```

- **cout** – поток вывода. Обычно поток cout связан с экраном, и все что передается в поток печатается в текстовом виде на экране консоли**.**

- **std::** – префикс, означающий, что используется реализация из стандартной библиотеки.
- << – в данном контексте, это операция берущая данные, указанные слева, преобразующая их в текстовый вид, и отсылающая, в поток, указанный слева**.**

Для упрощения операций ввода-вывода существует возможность включить в программу библиотеку **iostream** и сразу указать, что все команды будут по умолчанию использоваться из стандартной библиотеки. Для этого необходимо написать следующим образом:

```
#include <iostream>
using namespace std;
```

В этом случае, префикс std:: можно опустить, и команда вывода строк на экран будет выглядеть следующим образом:

```
cout << "Hello"
```

2.2. Переменные

Переменная — это именованная область памяти, в которую могут быть записаны различные значения. Также из этой области памяти может быть извлечено значение переменной, используя ее имя. В каждый момент времени переменная может иметь только одно значение.

Значения, которые может хранить переменная, определяется ее **типом**.

В простейшем виде переменную можно определять следующим образом:

Тип список_имен_переменных

Тип переменной определяет значения, которые может принимать переменная.

Типы могут быть следующими:

char — целое значение, 8 бит (диапазон от –128 до 127);

int — целое значение, обычно 4 байта, зависит от платформы;

float **—** вещественные числа;

double— вещественные числа удвоенной точности.

Каждый из целочисленных типов может быть определен как знаковый **signed** либо как беззнаковый **unsigned** (по умолчанию **signed**).

Пример определения переменных:

```
char my_symbol
int val, val2
double Result
```

После того как переменная создана, ей можно присваивать значения. Это можно сделать как при определении переменной, так и после этого:

Тип имя_переменной = начальное_значение;
имя_переменной = начальное_значение;

Например:

```
int val = 5
val = 5;
```

Необходимо помнить, что задавать значения переменной можно только после того, как эта переменная создана. А если значение явно не определено при создании, то по умолчанию оно ничему не присваивается. То есть попытка вывести значения неинициализированной переменной на экран может привести либо к выводу «мусора», либо даже к ошибке выполнения программы (в зависимости от среды выполнения).

2.3. Структура программы

Классической первой программой, которую обычно пишут, является вывод на экран «Hello World!». Ниже приведен код программы, которая это делает.

```
1 #include <iostream>
2 using namespace std;
3
4 void main()
5 {
6    cout << "Hello World!";
7 }
```

Разберем подробно структуру программы.

В строках 1 и 2 происходит подключение стандартной библиотеки для использования потоками ввода-вывода

Строчка 3 остается пустой для большей наглядности программы

В строке 4 описывается функция void main(). Подробное изучение функций происходи ниже, поэтому пока такое описание необходимо принять за аксиому.

В 5 строке открывается фигурная скобка, которая обозначает начало функции main, а в 7 строке фигурная скобка закрывается, обозначая что функция закончена. Внутри этих скобок собственно и происходит описание функции.

Функция main является главной функцией программы, так как только она исполняется в программе. То есть при запуске программы, происходит выполнение тех команд, которые написаны в этой функции. В описанном примере это одна команда – вывод на экран текстового сообщения.

Внутри функции main (а также всех других функций), каждая строка должна заканчиваться символом **;**.

Написанные код, как видно, имеет удобное форматирование, удобное для чтения. Форматирование производится командой табуляции (кнопка **Tab** на клавиатуре). Но при этом, если написание программы происходит построчно, то при переходе на следующую строчку (кнопка **Enter** на клавиатуре), текст форматируется автоматически.

2.4. Ввод и вывод переменных

При работе программы, зачастую необходимо чтобы пользователь сообщал программе различные параметры, а программа, по ходу выполнения, выдавала результат вычислению пользователю.

В языке программирования С, для вывода значений переменных на консоль, необходимо дополнительно указывать тип данных. Поэтому вывод переменных выглядит следующим образом:

```
int a = 4;
char ch = 'Q';
double pi = 3.141592;

printf("%d", a);
printf("%c", ch);
printf("%f", pi);
```

Помимо этого, при вводе значений переменных с консоли, программист должен априорно быть знакомым с понятиями ссылок и указателей на объкты, а также помнить, что для дробных чисел ввод и вывод осуществляются по-разному.

```
double pi;
scanf("%lg", &pi);
```

Язык программирования С++, позволяет значительно упростить операции ввода-вывода значений переменных, за счет использования потоков.

```
int a = 4;
char ch = 'Q';
double pi = 3.141592;

cout << a;
cout << ch;
cout << pi;
```

Ввод значений переменных также осуществляется без дополнительной информации о переменной.

```
double pi;
cin >> pi;
```

- **cin** – поток ввода. Обычно поток cin связан с клавиатурой, и весь набираемый в консоли текст, хранится в потоке.
- >> – в данном контексте, это операция извлекающая данные из потока справа, преобразующая их из текстового вида, и складывающая их в переменную, указанную слева.

При операциях вывода значений переменных возможны комбинации переменных и текстовых выражений. При операциях ввода, такие средства не предусмотрены.

Ниже приведен фрагмент кода, запрашивающий у пользователя значение переменной, и выводящей ее квадрат.

```
int a;
int Res;

cout << "Input value a = ";
cin >> a;

Res = a*a;

cout << "a*a = " << Res << endl;
```


Обратите внимание, что символы >> и << показывают направление потока – в переменную, или из нее на консоль.

Специальная команда **endl** означает перевод строки.

2.5. Арифметические операции и их использование

Для вычислений выражений в языках C и C++ могут использоваться различные арифметические операции. При этом каждая операция имеет свой ранг, определяющий приоритет ее выполнения. Чем ниже ранг операции, тем больший приоритет имеет операция. Операции одного ранга исполняются согласно правилам ассоциативности либо слева направо (–>), либо справа налево (<–). В таблице разобраны типовые операции по рангам и тип их ассоциативности:

Ранг	***Операции***	***Ассоциативность***
1	*() [] –> .*	➔
2	*! ~ + – ++ -- & * (тип) sizeof()*	➔
3	** / %* *(мультипликативные бинарные)*	➔
4	*+ – (аддитивные бинарные)*	➔
5	*<< >> (поразрядного сдвига)*	➔
6	*< <= >= > (отношения)*	➔
7	*== != (отношения)*	➔
8	*& (поразрядная конъюнкция «И»)*	➔
9	*^ (поразрядное исключающее «ИЛИ»)*	➔
10	*| (поразрядное дизъюнкция «ИЛИ»)*	➔
11	*&& (конъюнкция «И»)*	➔
12	*|| (дизъюнкция «ИЛИ»)*	➔

13	*?: (условная операция)*	←
14	= *= /= %= += –= &= ^= |= <<= >>=	←
15	*, (операция «запятая» – перечисление)*	→

2.5.1. Выражения и приведение арифметических типов

Каждое выражение состоит из одного или нескольких операндов, символов операций и ограничителей, в качестве которых чаще всего выступают круглые скобки ( ). Назначение любого выражения — формирование некоторого значения. Тип результата определяется типом выражений. Если значениями выражения являются целые и вещественные типы, то говорят об арифметических выражениях, в которых допустимы следующие операции:

+ — сложение;

– — вычитание;

* — умножение;

/ — деление;

% — получение остатка от целочисленного деления.

Примеры выражений с двумя операндами:

```
A + b
12.3 – x
3.14169 * Z
e / 8
8 % i
```

Также существуют специфичные унарные операции **++** и **--** для изменения операнда на 1. При этом эти операнды могут применяться как

после операнда (a++), так и до него (--x). При этом результат будет различный:

- Если изначально a было равно 6. То после выполнения z = a++, результатом будет: z = 6; a = 7;
- Если изначально a было равно 6. То после выполнения z = --a, результатом будет: z = 5; a = 5.

2.5.2. Отношения и логические выражения

Отношение определяется как пара выражений, между которыми стоит знак операции отношения. Допустимы следующие операции:

== — равно (два знака «равно»);

!= — не равно;

> — больше;

< — меньше;

>= — больше или равно;

<= — меньше или равно.

Примеры отношений:

```
a – b > 6.8
(x – 8) * 3 == 15
5 >= 1
```

Логический тип (истина или ложь) в языке С отсутствует. Поэтому принято, что отношение имеет ненулевое значение (обычно 1), если оно истинно, и равно 0, если оно ложно.

Логических операций в языке С три:

!	— отрицание	— логическое НЕ
&&	— конъюнкция	— логическое И
||	— дизъюнкция	— логическое ИЛИ

Так как значением отношения является целое, то ничего не противоречит применению логических операций к целочисленным значениям. При этом любое положительное ненулевое значение будет восприниматься как истинное. Иначе говоря значением !0 будет 1, а !54 будет 0.

2.5.3. Приведение типов

Особенность языков C и C++ заключается в том, что при вычислении значений выражений, результат всегда приводится к типу первого операнда. То есть при делении двух целых значений результат будет также иметь тип целого числа, поскольку каждый из операндов представлен целым типом.

```
int a = 5;
int b = 2;
double res;

res = a / b;     // Результатом будет 2
```

Для получения вещественного типа необходимо, чтобы хотя бы один операнд имел тип вещественного числа. Для этого можно использовать приведение типов. В коде программ достаточно часто приводится явное приведение типов, что не влечет изменение типа самой переменной. Рекомендуется в случае подобных вычислений приводить к желаемому типу первый операнд, что упрощает чтение и последующую отладку программ.

```
int a = 5;
int b = 2;
double res;

res = (double)a / b;
```


2.5.4. Выражения с поразрядными операциями

Такие операции позволяют конструировать выражения, в которых обработка операндов выполняется побитно (в двоичном представлении числа). Возможны следующие операции:

~ — инвертирование битов; // ~170 равно 85
>> — сдвиг последовательности битов вправо; // 100 >> 2 равно 25
<< — сдвиг последовательности битов влево // 5 << 2 равно 20
^ — поразрядное исключающее ИЛИ;
| — поразрядное ИЛИ;
& — поразрядное И;

Операции побитового сдвига часто применяются в программировании, поскольку позволяют значительно ускорять операции умножения или целочисленного деления на 2^n.

```
int a;

cout << "Input value a = ";
cin >> a;

cout << "a*16 = " << (a << 4) << endl;
cout << "a/8 = " << (a >> 3) << endl;
```

2.6. Операторы ветвления

В языках C и C++ существует два оператора ветвления:

- условный оператор if.
- переключатель switch.

2.6.1. Оператор if

Оператор **if** может использоваться либо для условного выполнения определенного действия, либо для разветвления программы на небольшое количество ветвей.

В случае условного исполнения, оператор имеет следующий вид:

```
if ( условие )
    оператор;
```

При этом, если необходимо исполнить несколько операторов, то их необходимо выделить в ТЕЛО, которое обозначается фигурными скобками.

```
if ( условие )
{
    оператор_1;
    оператор_2;
    оператор_3;
}
```

В случае ветвления исполнения, оператор имеет следующий вид:

```
if ( условие )
    оператор_1;
else
    оператор_2;
```

При этом возможно ветвление и на большее количество ветвей:

```
if ( условие_1 )
    оператор_1;
else if(условие_2 )
    оператор_2;
```



```
else
    оператор_3;
```

В качестве условия могут использоваться арифметические выражения, отношения и логические выражения (x > 10 && x < 87).

Приведем фрагмент программы для определения корней уравнения вида

$$ax^2 + bx + c = 0$$

```
D = b*b-4*a*c;

if ( D < 0)
   cout << "No roots" << endl;

else if(D == 0)
{
   x = (double)-b / (2*a);
   cout << "1 root: x = " << x << endl;
}

else
{
   x1 = ( - b + sqrt(D)) / (2*a);
   x2 = ( - b - sqrt(D)) / (2*a);
   cout << "2 root: x1 = " << x1 << " x2 = " << x2;
}
```

2.6.2. Переключатель switch

Данный оператор используется для организации мультиветвления следующим и выглядит следующим образом:

```
switch (выражение)
{
    case Константа_1: операторы_1;
    case Константа_2: операторы_2;
```

```
    ....
    default: операторы;
}
```

При первом совпадении значения выражения с константой происходит выполнение операторов, помеченных данной меткой. Если после их выполнения не предусмотрено никаких операторов перехода, то выполняются также все последующие операторы. То есть по сути, CASE является меткой, обозначающей место выполнения программы после SWITCH.

Оператором перехода может быть **break**, который осуществляет выход из тела (фрагмент кода, обозначенный фигурными скобками).

Переключатели чаще всего используются, когда количество ветвей больше 3, либо необходимо перебрать заданные константы.

Для иллюстрации приведен фрагмент программы, которая выводит на экран название введенной цифры от 0 до 9:

```
int a;
cout << "Input a = ";
cin >> a;

switch (a)
{
  case 1: cout << a << " - One" << endl;
    break;
  case 2: cout << a << " - Two" << endl;
    break;
  case 3: cout << a << " - Three" << endl;
    break;
  case 4: cout << a << " - Four" << endl;
    break;
  case 5: cout << a << " - Five" << endl;
    break;
  case 6: cout << a << " - Six" << endl;
    break;
  case 7: cout << a << " - Seven" << endl;
    break;
```



```
  case 8: cout << a << " - Eight" << endl;
    break;
  case 9: cout << a << " - Nine" << endl;
    break;
  case 0: cout << a << " - Zero" << endl;
    break;
  default: cout << "The value is not from 0 to 9";
}
```

Еще раз отметим, что если бы в программе отсутствовали операторы break, то при вводе например цифры 6, на экран бы распечаталось следующее:

```
6 - Six
7 - Seven
8 - Eight
9 - Nine
0 - Zero
The value is not from 0 to 9
```

2.7. Операторы циклов

В языках C и C++ существуют три разных способа оргинизации циклов. Ниже мы их разберем с примерами использования.

2.7.1. Цикл for

Оператор **for** является, пожалуй базовым оператором для организации циклов. Цикл **for** — это параметрический цикл, то есть имеется возможность задавать параметры для выполнения цикла: начальные значение и условия. Форма записи выглядит следующим образом:

for (выражение_1; условие_цикла; выражение_2)
Тело цикла

выражение_1 – определяет действия, выполняемые до начала цикла.

условие_цикла – обычно логическое или арифметическое условие. Пока это условие истинно, выполняется цикл. Проверка происходит перед началом очередной итерации тела цикла.

выражение_2 — это действие, выполняемые в конце каждой итерации цикла.

.

В качестве примера рассмотрим программу, распечатывающую все числа, которые делятся на 2 до введенного пользователем значения.

```
int i, Max;

cin >> Max;

for(i = 0; i <= Max; i++)
{
   if(i%2 == 0)
      cout << i << endl;
}
```

Выражения, указывающиеся в скобках оператора **for** не являются обязательными. Ниже приведен полностью идентичный код, выполняющий точно такой же цикл.

Помните, что даже если программист не указывает внутри оператора for какие-либо выражения, все 3 поля должны быть оставлены пустыми!

```
int i, Max;
cin >> Max;
i = 0;
for(;;)
{
   if(i <= Max)
   {
      if(i%2 == 0)
      cout << i << endl;

      i++;
```



```
   }
   else
      break;
}
```

2.7.2. Цикл while

Цикл **while** — это цикл с предусловием, то есть перед каждой итерацией проверяется условие, и только если оно истинно, то происходит выполнение тела цикла. Он имеет следующий вид:

while (выражение условия цикла)

Тело цикла

Приведем фрагмент кода, который также распечатывает все числа, которые делятся на 2 до введенного пользователем значения

```
i = 0;
while(i <= Max)
{
   if(i%2 == 0)
      cout << i << endl;

   i++;
}
```

Как видно из примера, условие цикла записывается в более наглядном виде, чем при использовании **for**. Но при таком описании довольно часто возникают ошибки связанные с тем, что необходимо явно описывать инкрементацию счетчика цикла.

2.7.3. Цикл do-while

Цикл **do** — это цикл с постусловием, то есть проверка условия цикла происходит после каждой итерации. Это обеспечивает, по крайней мере,

одну итерацию выполнения вне зависимости от условий. Он имеет следующий вид:

do

Тело цикла

while (выражение условия цикла);

2.8. Массивы данных

В случае, если программы обрабатывает множество однотипных данных, именовать каждую переменную не только неудобно, но порой и просто невозможно из-за их большого количества. Для обработки однотипных данных применяются ***массивы***, которые позволяют компоновать переменные.

Наиболее наглядными примерами применения массивов являются задачи статистической обработки данных. Также можно упомянуть обработку векторов различной размерности.

Для определения массива необходимо указать, какой тип будут иметь его элементы, и количество этих элементов:

```
int mas[10];
```

Таким образом, определен одномерный массив, состоящий из целых чисел, имеющий 10 элементов. Элементы в массиве нумеруются начиная с 0, поэтому в приведенном примере массив содержит элементы с номерами от 0 до 9. Обращаться к его элементам следует следующим образом:

mas[i]

Ниже приведен фрагмент кода, в котором пользователь задает 10 значений, а программа выводит их сумму.

```
int mas[10];
```



```
int i;

for(i=0; i < 10; i++)
{
  cout << "a[" << i << "] = ";
  cin >> mas[i];
}

int Sum = 0;
for(i=0; i < 10; i++)
Sum += mas[i];

cout << "Summa = " << Sum << endl;
```

Из примера видно, что работа с элементами массивов мало чем отличается от работы с любыми другими переменными.

2.9. Функции

Теперь некоторые пояснения к самой программе. Любая программа на языке С, состоит из одной или более "***функций***", указывающих фактические операции компьютера, которые должны быть выполнены. По своей сути, в функции перечисляется порядок действий, а также заводятся переменные для хранения данных.

Функция с именем **main** – особенная. Именно выполнение этой функции и происходит при запуске любой программы. Это означает, что каждая программа должна в каком-то месте содержать функцию с именем **main**. Для выполнения определенных действий функция **main** обычно обращается к другим функциям, часть из которых находится в той же самой программе, а часть - в библиотеках, содержащих ранее написанные функции.

Действия, выполняемые при обращении к функции, задает ее тело, которое выделяется фигурными скобками **{ }**. Структура определения функции имеет вид:

Тип_результата Имя функции (список_параметров);

Реализация функции происходит следующим образом:

```
Тип_результата Имя_Функции (список_параметров)
{
   ...
}
```

Необходимо помнить, что в отличие от объявления переменных, в списке параметров функции недопустимо перечислять несколько имен переменных одного типа – для каждой переменной должен быть явно указан тип.

Реализация функции всегда должна происходить после ее объявления. В ряде случаев (например когда вся программа состоит только из одного файла), допускается не делать отдельных объявлений функций.

Помните, что объявление (или реализация, если объявления не существут) должно производится до того места, где функция вызывается! То есть если создается пользовательская функция, которая будет вызываться из функции **main**, то она должна быть описана до функции **main**.

Если не указан тип возвращаемого значения, по умолчанию считается, что функция возвращает тип ***int***. Для возврата значения используется команда **return** (она может быть выполнена в любом месте кода тела, но выполнение функции после этого заканчивается). Если функция не возвращает никаких значений, то используется специальный тип **void**.

В качестве примера рассмотрим программу, в которой описана пользовательская функция для вычисления дискриминанта квадратного уравнения, вида

$$ax^2 + bx + c = 0:$$

```
#include <iostream>
using namespace std;
```



```
double Discr_Calc(int a, int b, int c)
{
  double D;
  D = b*b - 4*a*c;
  return D;
}

void main()
{
  int a, b, c;
  double D;
  a = 1;
  b = 5;
  c = 3;
  D = Discr_Calc(a, b, c);
  cout << "D = " << D << endl;
}
```

Функция может возвращать только одно значение.

Обычно функции используются для написания более простых для чтения программ. Так программа, которая содержит только функцию **main()**, состоящую из 13 окон теста, очень трудна для понимания, тем более сторонним человеком. Поэтому часто используется негласное соглашение, что текст любой функции не должен превышать по размеру одного экрана — размера, которые человек может охватить взглядом. При выборе имен функций программист также должен руководствоваться тем принципом, чтобы постороннему человеку стало интуитивно понятно, что делает функция.

2.10. Локальные и глобальные переменные

Все создаваемые в программе переменные, имеют определенную область видимости. Для переменных выделяют две основных области видимости: **локальные** и **глобальные**.

2.10.1. Глобальные переменные

Из названия этого класса переменных становится понятно, что они доступны всем. Под всеми подразумеваются все функции проекта.

Для того чтобы переменная была глобальная, она должна быть объявлена вне функций. Ниже приведен текст программы, в которой переменная с именем D является глобальной. При этом к ней обращаются две различные функции, для которых эта переменная является общей.

```
#include <iostream>
using namespace std;

double D = 0;

void Discr_Calc(int a, int b, int c)
{
  D = b*b - 4*a*c;
}

void main()
{
  int a, b, c;
  a = 1; b = 5; c = 3;

  Discr_Calc(a, b, c);
  cout << "D = " << D << endl;
}
```

2.10.2. Локальные переменные

Локальные переменные обладают ограничением области видимости – они видны и доступны только внутри того блока, в котором они созданы. Важно помнить, что локальные переменные перестают существовать при выходе из блока, в котором они были объявлены.

Блоками, ограничивающими видимость, могут являться функции, либо просто набор действий, выделенных в отдельное тело.

Таким образом, если бы в тесте предыдущей программы описать функцию вычисления дискриминантом следующим образом:


```
void Discr_Calc(int a, int b, int c)
{
  double D;
  D = b*b - 4*a*c;
}
```

То в ней существует собственная локальная переменная D, которая перестает существовать, как только исполнение программы выходит из этой функции. Другими словами, локальные переменные перекрывают глобальные, в результате чего, выполнение описанный программы с такими изменениями приведет к тому, что на экран всегда будет выводиться НОЛЬ!

Ниже приведен пример еще одной неправильно написанной программы, которая не сможет быть даже скомпилирована.

```
#include <iostream>
using namespace std;

void Discr_Calc(int a, int b, int c)
{
  D = b*b - 4*a*c;
}

void main()
{
    int a, b, c;
    double D;
    a = 1;
    b = 5;
    c = 3;

    Discr_Calc(a, b, c);
    cout << "D = " << D << endl;
}
```

В приведенной программе ошибка заключается в том, что переменная с именем **D** определена в функции ***main()***, и не существует за ее пределами. Таким образом, в функции ***Discr_Calc()*** происходит обращение к несуществующей переменной **D**.

Переменные, перечисленные в списке параметров функции, также являются локальными для этой функции. Их отличие от описанных в теле функции только в том, что они будут гарантированно инициализированы к моменту вызова функции.

3. Работа с динамической памятью

Работа с динамической памятью позволяет программисту решать задачи с неизвестным количеством исходных данных, позволяя добавлять новые возможности в функционал программ.

Основными понятиями для работы с динамической памятью являются указатели и ссылки. Именно с помощью переменных этих типов осуществляется управление памятью, ее выделение и освобождение.

3.1. Указатели и работа с ними

Каждая переменная представляет собой именованный участок памяти ПК, в котором хранится ее значение. При этом, чтобы получить доступ к значению переменной, необходимо обратиться к ней по ее имени.

```
int a;
a = 1;
```

Для того чтобы получить значение адреса в памяти, по которому находится значение конкретной переменной, необходимо произвести **унарную операцию разыменования &.** Выражение **&a** позволяет получить адрес участка памяти, выделенного для переменной a.

```
int a;
a = 5;
cout << &a;
```

Имея возможность определять адрес переменной или другого объекта программы, нужно уметь его сохранять, преобразовывать и передавать. Для этих целей введены переменные типа «**указатель**».

Указатель — это переменная, значение которой является адресом объекта конкретного типа. Для обозначения значения указателя, который никуда не указывает, используется специальная константа **NULL**.

При определении указателя необходимо обозначать, на какой тип данных указывает эта переменная. Здесь и далее при объявлении указателей будем использовать символ «**p**» в названиях указателей. Это позволит легко отличать в тексте программы указатели, от собственно переменных. Примеры определения указателя и присваивания ему значения:

```
int Ch = 14;
int* pCh = NULL;
pCh = &Ch;
```

Чтобы получить значение, которое находится по указанному адресу, применяется операция разыменования – «*»:

```
int Ch = 14;
int* pCh = &Ch;
cout << *pCh;
```

3.2. Арифметика указателей и массивы

В языке С допустимы только две арифметические операции над указателями: суммирование и вычитание. При этом данные операции обеспечивают передвижение по памяти не по байтам и битам, а на размер типа данных, который указан при создании переменной указателя.

Указатели непосредственно связаны с массивами данных, поскольку при создании массива все его элементы создаются в памяти последовательно, т. е. массив представляет собой единую и неразрывную область памяти. При этом если обратиться к массиву через его имя, то результатом будет адрес первого элемента массива (имеющего индекс 0).

```
int mas[3] = {1, 2, 3};
int* pVal;
pVal = mas; // тождественно pVal = &mas[0]
cout << *pVal;
```


Для указателей определены арифметические операции. Таким образом, к указателям можно добавлять или отнимать целые значения. Так операция «++», примененная к указателю, изменяет адрес, хранящийся в указатели на число байт, соответствующее размеру одной переменной типа, соответствующую типа указателя. Иначе говоря, если у нас имеется указатель на тип ***int***, занимающий 4 байта, то операции ++ сместит указатель в памяти на 4 байта.

Поэтому для адресации внутри массива бывает удобно использовать указатели. В качестве примера приведем фрагмент кода для инициализации массива:

```
int mas[10], i;
int* pMas;

for(i = 0, pMas = mas; i < 10; i++)
  cin >> *(pMas+i);
```

2.3.1. Динамическая память. Массивы

До сих пор все массивы данных у нас были строго определенного размера. В некоторых задачах это бывает неудобно. Например, требуется, чтобы пользователь задал некий массив данных, при этом неизвестен его размер. Можно, конечно, заранее выделить под массив 100 МБайт памяти и считать, что этого всегда хватит. Но возможны два варианта:

- массив все-таки окажется недостаточным, поскольку данные занимают 101 Майт.
- Для конкретной задачи достаточно 10 элементов по 4 КБайт.

В обоих этих случаях необходимый объем памяти становится известен только на момент использования программы. Поэтому было бы намного удобнее, если бы программист имел возможность выделять необходимый объем памяти по требованию пользователя.

Такая возможность имеется и реализуется с помощью указателей и средств для динамического выделения памяти, из которых основными являются функции **new** и **delete**. Эти функции используются для выделения и освобождения памяти.

Приведем фрагмент кода, в котором происходит выделение необходимого количества памяти для хранения данных, количество которых задает пользователь. А также происходит инициализация значений, поиск максимального элемента и среднего арифметического значения.

```
int* pMas;
int Count;

cout << "Input elements count: ";
cin >> Count;

pMas = new int [Count];

int i;
for(i = 0; i < Count; i++)
  cin >> *(pMas+i);

int Sum = 0;
for(i = 0; i < Count; i++)
  Sum += *(pMas+i);

double Avg;
Avg = (double)Sum/Count;

cout << "Summa = " << Sum << endl;
cout << "Avg = " << Avg << endl;
```

В языке программирования C++ оператор **delete** возвращает память, выделенную оператором **new**. Вызов **delete** должен происходить для каждого вызова **new**, дабы избежать утечки памяти. После вызова **delete**

объект, указывающий на этот участок памяти, становится некорректным и не должен больше использоваться. При этом важно помнить, что если память была выделена при помощи **new**, то она освобождается при помощи **delete**, а если **new[]**, то должен быть вызван **delete[]**.

```
int* pA;
int* pMas;

pA = new int;
pMas = new int[10];

delete pA;
delete[] pMas;
```

3.3. Передача переменных по ссылке

Еще раз напомним про различие локальных и глобальных переменных.

```
void MyFunc(int a, int b, int c)
{
  a = 11;
  b = 12;
  c = 13;
}

void main()
{
  int a, b, c;
  a = 1; b = 2; c = 3;

  MyFunc(a, b, c);

  cout << a << b << c;
}
```

В результате исполнения приведенного кода, на экран будет выведено **1 2 3.** Другими словами, переменные в функции ***main()*** не изменят своего значения. В функции ***MyFunc()*** определены собственные локальные

переменные с такими же именами, но которые не имеют отношения к переменным в функции ***main()***.

При написании программ часто возникает желание произвести внутри функции манипуляции с переменными таким образом, чтобы в вызываемой функции они также изменили значения. Например, в программах часто применяется метод инициализации всех переменный внутри отдельного блока (функции). **Одним из способов** решения такой задачи является заведение глобальных переменных следующим образом.

```
int a, b, c;

void MyFunc()
{
  a = 11;
  b = 12;
  c = 13;
}

void main()
{

  a = 1; b = 2; c = 3;

  MyFunc();

  cout << a << " " << b << " "  << c;
}
```

В этом случае никаких переменных в функцию передавать не надо, так как она имеет непосредственный доступ ко всем глобальным переменным.

Но помните, если вы все-таки напишите следующим образом:


```
int a, b, c;

void MyFunc(int a, int b, int c)
{
  a = 11;
  b = 12;
  c = 13;
}

void main()
{

  a = 1; b = 2; c = 3;
  MyFunc(a, b, c);
  cout << a << " " << b << " "  << c;
}
```

то в функции будут использоваться уже не глобальные, а локальные переменные. Это вновь приведет к тому, что значение глобальных переменных не изменится после выполнения функции.

Второй способ заключается в передаче переменных по ссылке.

Оператор **&** — это унарный оператор, возвращающий адрес своего операнда. (Напомним, что унарный оператор имеет один операнд). Например, если написать **&count**, то результатом будет адрес переменной **count**. Оператор **&** можно представить себе как оператор, возвращающий адрес объекта.

Для хранения адресов используются указатели, а для присвоения значения переменной через указатель на нее, унарный оператор разыменования *****. Таким образом, корректной является запись

```
int* p;
int count;
p = &count;

*p = 5;
cout << count;
```

Таким образом, приведем фрагмент кода, в котором переменные передаются по ссылке в функции, которая изменяет их значения.

```
void MyFunc(int* a, int* b, int* c)
{
  *a = 11;
  *b = 12;
  *c = 13;
}

void main()
{
  int a, b, c;

  MyFunc(&a, &b, &c);

  cout << a << " " << b << " "  << c;
}
```


4. СТРУКТУРНОЕ ПРОГРАММИРОВАНИЕ

Методология структурного программирования появилась как следствие возрастания сложности решаемых на компьютерах задач, и соответственного усложнения программного обеспечения. В 70-е годы XX века объёмы и сложность программ достигли такого уровня, что «интуитивная» (неструктурированная, или «рефлекторная») разработка программ, которая была нормой в более раннее время, перестала удовлетворять потребностям практики. Программы становились слишком сложными, чтобы их можно было нормально сопровождать, поэтому потребовалась какая-то систематизация процесса разработки и структуры программ.

Наиболее сильной критике со стороны разработчиков структурного подхода к программированию подвергся оператор **GOTO** (оператор безусловного перехода), имевшийся тогда почти во всех языках программирования. Неправильное и необдуманное использование произвольных переходов в тексте программы приводит к получению запутанных, плохо структурированных программ (т.н. спагетти-кода), по тексту которых практически невозможно понять порядок исполнения и взаимозависимость фрагментов.

Следование принципам структурного программирования сделало тексты программ, даже довольно крупных, нормально читаемыми. Серьёзно облегчилось понимание программ, появилась возможность разработки программ в нормальном промышленном режиме, когда программу может без особых затруднений понять не только её автор, но и другие программисты. Это позволило разрабатывать достаточно крупные для того времени программные комплексы силами коллективов разработчиков, и сопровождать эти комплексы в течение многих лет, даже в условиях неизбежных изменений в составе персонала.

Перечислим некоторые достоинства структурного программирования:

1. Структурное программирование позволяет значительно сократить число вариантов построения программы по одной и той же спецификации, что значительно снижает сложность программы и, что ещё важнее, облегчает понимание её другими разработчиками.
2. В структурированных программах логически связанные операторы находятся визуально ближе, а слабо связанные — дальше, что позволяет обходиться без блок-схем и других графических форм изображения алгоритмов (по сути, сама программа является собственной блок-схемой).
3. Сильно упрощается процесс тестирования и отладки структурированных программ.

4.1. Методология

Структурное программирование — методология разработки программного обеспечения, в основе которой лежит представление программы в виде иерархической структуры блоков, предложенная в 70-х годах XX века Э. Дейкстрой, а разработана и дополнена Н. Виртом.

В соответствии с данной методологией

1. Любая программа представляет собой структуру, построенную из трёх типов базовых конструкций:
    a. ***последовательное исполнение*** — однократное выполнение операций в том порядке, в котором они записаны в тексте программы;
    b. ***ветвление*** — однократное выполнение одной из двух или более операций, в зависимости от выполнения некоторого заданного условия;
    c. ***цикл*** — многократное исполнение одной и той же операции до тех пор, пока выполняется некоторое заданное условие (условие продолжения цикла).

2. Повторяющиеся фрагменты программы, либо представляющие собой логически целостные вычислительные блоки, оформляются в виде функций (подпрограмм).
3. Разработка программы ведётся пошагово, методом «сверху вниз».

4.1.1. Создание программ

При решении любой задачи сначала пишется текст основной программы, в котором, вместо каждого связного логического фрагмента текста, вставляется вызов подпрограммы, которая будет выполнять этот фрагмент. Вместо настоящих, работающих подпрограмм, в программу вставляются «заглушки», которые ничего не делают. Полученная программа проверяется и отлаживается. После того, как программист убедится, что подпрограммы вызываются в правильной последовательности (то есть общая структура программы верна), подпрограммы-заглушки последовательно заменяются на реально работающие, причём разработка каждой подпрограммы ведётся тем же методом, что и основной программы. Разработка заканчивается тогда, когда не останется ни одной «затычки», которая не была бы удалена.

Такая последовательность гарантирует, что на каждом этапе разработки программист одновременно имеет дело с обозримым и понятным ему множеством фрагментов, и может быть уверен, что общая структура всех более высоких уровней программы верна. При сопровождении и внесении изменений в программу выясняется, в какие именно процедуры нужно внести изменения, и они вносятся, не затрагивая части программы, непосредственно не связанные с ними. Это позволяет гарантировать, что при внесении изменений и исправлении ошибок не выйдет из строя какая-то часть программы, находящаяся в данный момент вне зоны внимания программиста.

4.2. Программа «бродилка»

Суть программы заключается в том, чтобы пользователю было представлено игровое поле, на котором были случайным образом разбросаны препятствия. При этом пользователь мог бы управлять игроком внутри поля и перемещать его внутри лабиринта.

4.2.1. Подключение внешней библиотеки

Для работы с текстовой графикой предлагается использовать библиотеку текстовой графики Conlib.

Библиотека предоставляется в виде двух файлов:

- **Conlib.h** – описание реализованных в библиотеке функций и описание их вызовов.
- **Conlib.lib** – реализация функций.

Для работы необходимо скопировать оба эти файла в директорию проекта. После этого прописать добавление деклараций функций библиотеки:

```
#include "conlib.h"
```

Если работа происходит в **MSVS2008**, то отдельно прописывать **Conlib.lib** нигде не нужно.

Если же работа происходит в **MSVS2005**, то необходимо явно указать в настройках проекта файл **Conlib.lib**. Это делается следующим образом (рис. 7):

Настройки проекта –> Configuration options -> Linker ->
-> Input -> Additional Dependencies -> … прописать название

После этого библиотека будет включаться в проект и будут доступны все ее функции.

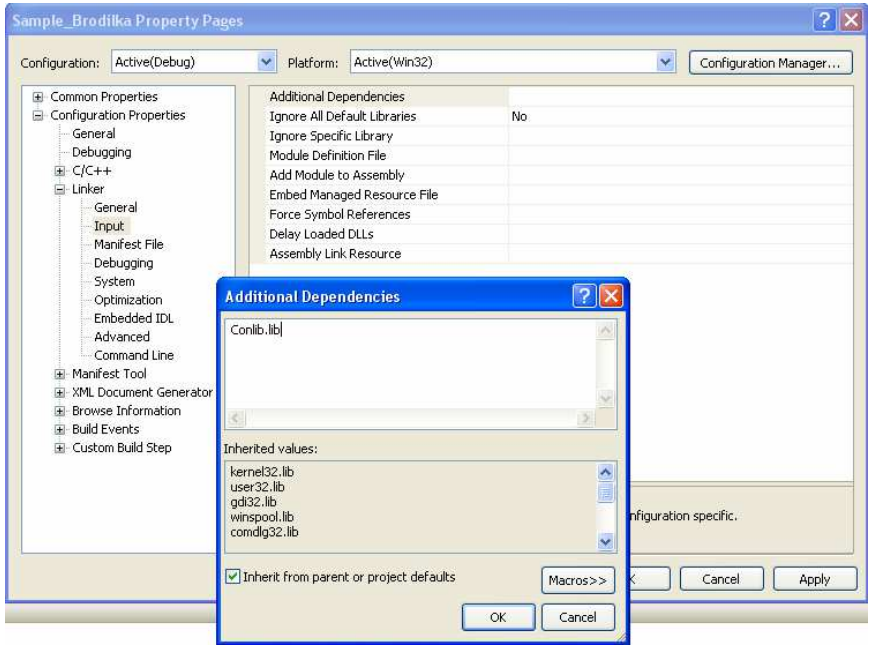


Рис. 7. Добавление внешней библиотеки в проект

4.2.2. Основные функции библиотеки Conlib

Основными функциями данной библиотеки для программы бродилка являются функции позиционирования на экране и обработка нажатий клавиш на клавиатуре для реализации интерактивности и перемещения игрока по экрану.

- ***int GotoXY(int x, int y);***
  функция перемещает курсор в заданную координату на экране.

- ***int MaxXY(int *px, int *py);***
  функция определяет максимальные размеры консоли

- ***int ClearConsole();***
  производит полную очистку консоли от всего содержимого

- ***int KeyPressed();***
  возвращает не 0, если была нажата какая-либо клавиша на клавиатуре

- ***int GetKey();***
  возвращает код нажатой клавиши

- ***int SetColor(short color);***
  позволяет задавать цвет шрифта и фона

4.2.3. Создание каркаса программы

Согласно сформулированным в первой главе описанием алгоритмов решения задачи, необходимо во-первых определиться с входными данными и результатом программы.

Итак, ***входными данными являются***:

- размер игрового поля;
- количество препятствий внутри лабиринта (либо процент заполнения лабиринта препятствиями).

Результатом является представление игрового поля в следующем виде (рис. 8):

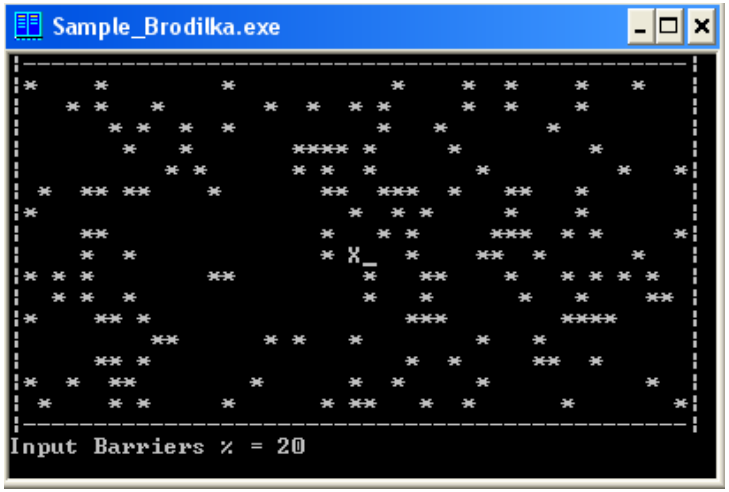


Рис. 8. Проект «бродилка»

Далее необходимо составить алгоритм решения задачи. Если у человека не возникает больших трудностей с формулировками, то решение может быть сразу представлено в виде кода программы, в которой будут использованы функции в качестве логических блоков.

```
int main()
{

  int MaxX, MaxY;
  int BarrierCount;

  Init(&MaxX, &MaxY, &BarrierCount);
  int* pPole;
  pPole = new int[MaxX * MaxY];

  FillPole(pPole, MaxX, MaxY, BarrierCount);
  PrintPole(pPole, MaxX, MaxY);

 int PlayerX, PlayerY;

  PlayerX = MaxX / 2;
  PlayerY = MaxY / 2;
```

```
    Play(pPole, PlayerX, PlayerY, MaxX, MaxY);

    ClearConsole();
    GotoXY(10, 5);
    cout << "Press any key to continue...";
}
```

Разберем более подробно данный код:

- функция **Init()** определяет входные параметры для программы, в которые входят размеры игрового поля и количество препятсвий;
- функция **FillPole()** производит заполнение игрового поля содержимым: границами и препятствиями
- функция **PrintPole()** распечатывает игровое поле на экран.
- После этого определяются начальные координаты игрока и вызывается функция **Play()**, которая отвечает за обработку клавиатуры и собственно перемещение игрока по полю.

4.2.4. Инициализация переменных программы

Функция **Init()** получает в качестве аргументов адреса переменных, куда производит запись данных. Таким образом, вызывая одну функцию, мы получаем все необходимые значения переменных.

```
void Init(int* pMaxX, int* pMaxY, int* pBarrierCount)
{
    MaxXY(pMaxX, pMaxY);

  int Pr;
  cout << "Input Barriers % = ";
  cin >> Pr;

  int PoleSize = (*pMaxX) * (*pMaxY);
  *pBarrierCount = PoleSize * Pr / 100;
}
```


4.2.5. Заполнение игрового поля

Разумным решением является разделить этот этап на несколько частей:

- Прописать все значения в поле нулем. Это необходимо, поскольку при выделении памяти этого автоматически не происходит.
- Расставить границы поля.
- Расставить препятствия внутри поля.

Помимо этого нам потребуются некоторые вспомогательные функции, которые мы должны реализовать сами.

Получение случайного числа

Стандартная функция rand() позволяет получить псевдослучайное число в диапазоне от 0 до RAND_MAX.

Описанная ниже функция позволяет получить псевдослучайное значение в заданном диапазоне.

```
int GetRandomValue(int Min, int Max)
{
  int Val = rand();

  int Range = Max - Min;

  double q = ((double)Val / RAND_MAX) * Range;

  return (q + Min);
}
```

Правило отображения игрового поля

Сложность состоит в том, что игровое поле является двумерным, а выделенная память – линейной. Разумным является расположить все строки последовательно в памяти. В этом случае функция преобразования двумерных координат в индекс массива будет выглядеть следующим образом:

```
int GetPoleIndex(int x, int y, int MaxX)
{
  return (y*MaxX + x);
}
```

Определение границ игрового поля

Функция последовательно расставляет верхнюю, нижнбб, левую и правую границы. Особенностями является то, что горизонтальные границы закодированы единицей, а вертикальные – двойкой.

Это необходимо, поскольку в постановке задачи границы имеют различное отображение.

```
void CreatePoleBorders(int* pPole,
                       int MaxX, int MaxY)
{
  int x, y;

  y = 0;
  for(x= 0; x < MaxX; x++)
    *(pPole + GetPoleIndex(x, y, MaxX)) = 1;

  y = MaxY-1;
  for(x= 0; x < MaxX; x++)
    *(pPole + GetPoleIndex(x, y, MaxX)) = 1;

  x = 0;
  for(y= 0; y < MaxY; y++)
    *(pPole + GetPoleIndex(x, y, MaxX)) = 2;

  x = MaxX-1;
  for(y= 0; y < MaxY; y++)
    *(pPole + GetPoleIndex(x, y, MaxX)) = 2;
}
```


Результирующая функция заполнения игрового поля

С использование всех перечисленных вспомогательных функций, требуемый блок по заполнению поля препятствиями будет иметь следующий вид:

```
void FillPole(int* pPole, int MaxX,
              int MaxY, int BarrierCount)

{
  int x, y;

  for(y = 0; y < MaxY; y++)
    for(x= 0; x < MaxX; x++)
      *(pPole + GetPoleIndex(x, y, MaxX)) =
0;

  CreatePoleBorders(pPole, MaxX, MaxY);

  int i;

  for(i = 0; i < BarrierCount; i++)
  {
    x = GetRandomValue(1, MaxX-1);
    y = GetRandomValue(1, MaxY-1);

    *(pPole + GetPoleIndex(x, y, MaxX)) = 3;
  }

}
```

4.2.6. Отображение игрового поля

Отображение поля использует созданные выше вспомогательные функции отображения.

При отображении необходимо учесть возможность сопоставлять различным кодам – различные символы на экране. Удобнее всего использовать для этого переключатель **switch**.

```
void PrintPole(int* pPole, int MaxX, int MaxY)
{

  int x, y;
  GotoXY(0, 0);
  for(y = 0; y < MaxY; y++)
  {
    for(x= 0; x < MaxX; x++)
      switch(*(pPole + GetPoleIndex(x, y, MaxX)))
      {
      case 1: cout << '-'; break;
      case 2: cout << '|'; break;
      case 3: cout << '*'; break;
      default: cout << ' ';break;
      }
    cout << endl;
  }
}
```

4.2.7. Обработка клавиатуры и перемещение игрока

```
    void Play(int *pPole, int CurX, int CurY,
              int MaxX, int MaxY)
    {
      GotoXY(CurX, CurY);
      cout << 'X';

      int key;
      int bExit = 0;
      while(!bExit)
        {
        while(!KeyPressed());
        key = GetKey();

        GotoXY(CurX, CurY);
        cout << ' ';

        if (key == KEY_ESC)
          bExit = 1;
        else if (key == KEY_UP)
          CurY--;
```



```
        else if (key == KEY_DOWN)
          CurY++;
        else if (key == KEY_LEFT)
          CurX--;
        else if (key == KEY_RIGHT)
          CurX++;

        GotoXY(CurX, CurY);
        cout << 'X';
        }
    }
```

5. Алгоритмы сортировки данных

Сортировка — это упорядочивание элементов по выбранному признаку. В случае чисел критерием может быть признак «больше» или «меньше», в результате чего мы получим последовательность, отсортированную по возрастанию (первый элемент будет самым маленьким) либо по убыванию (первый элемент будет иметь самое большое значение).

Единственного эффективнейшего алгоритма сортировки не существует ввиду множества параметров оценки эффективности:

1. **Время** — основной параметр, характеризующий быстродействие алгоритма. Называется также вычислительной сложностью. Для упорядочения важны худшее, среднее и лучшее поведение алгоритма в терминах размера списка (n). Для типичного алгоритма хорошее поведение — это O(n log n) и плохое поведение — это O(n2). Идеальное поведение для упорядочения — O(n). Алгоритмы сортировки, использующие только абстрактную операцию сравнения ключей, всегда нуждаются по меньшей мере в O(n log n) сравнениях в среднем.
2. **Память** — ряд алгоритмов требует выделения дополнительной памяти под временное хранение данных. При оценке используемой памяти не будет учитываться место, которое занимают исходный массив и не зависящие от входной последовательности затраты, например, на хранение кода программы.
3. **Устойчивость** (stability) — устойчивая сортировка не меняет взаимного расположения равных элементов.
4. **Естественность поведения** — эффективность метода при обработке уже упорядоченных или частично упорядоченных данных. Алгоритм ведёт себя естественно, если учитывает эту характеристику входной последовательности и работает лучше.

5.1. Сортировка пузырьком

Расположим массив сверху вниз, от нулевого элемента к последнему.

Идея метода: шаг сортировки состоит в проходе снизу вверх по массиву. По пути просматриваются пары соседних элементов. Если элементы некоторой пары находятся в неправильном порядке, то меняем их местами (рис. 9).

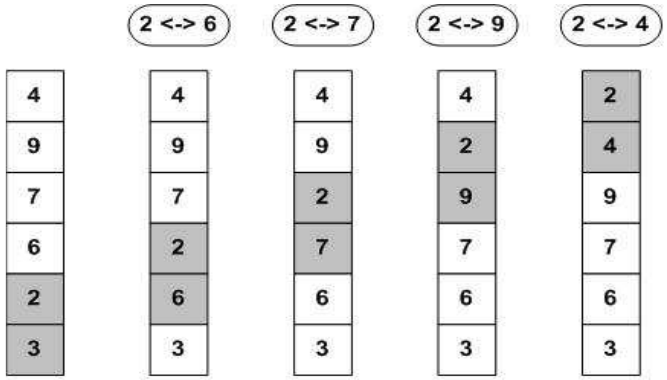


Рис. 9. Нулевой проход

После нулевого прохода по массиву «вверху» оказывается самый «легкий» элемент — отсюда аналогия с пузырьком. Следующий проход делается до второго сверху элемента, таким образом, второй по величине элемент поднимается на правильную позицию.

Делаем проходы по все уменьшающейся нижней части массива до тех пор, пока в ней не останется только один элемент (рис. 10). На этом сортировка заканчивается, так как последовательность упорядочена по возрастанию.

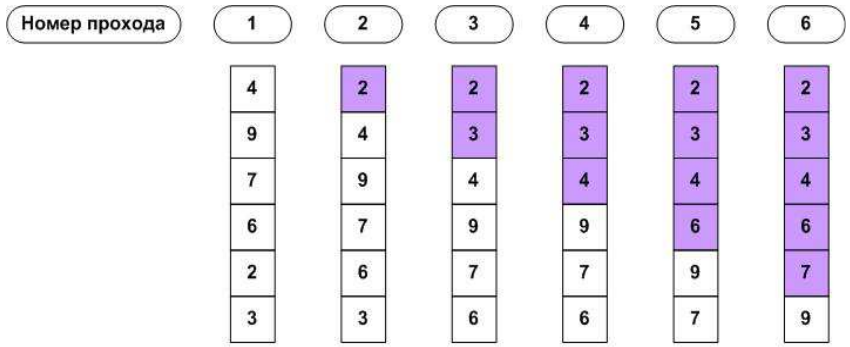


Рис. 10. Сортировка массива за шесть проходов

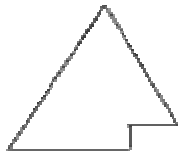
Среднее число сравнений и обменов имеют квадратичный порядок роста: $O(n^2)$, отсюда можно заключить, что алгоритм пузырька очень медлителен и малоэффективен.

Тем не менее, у него есть громадное преимущество в том, что он прост в реализации, и его можно улучшать различными способами.

5.2. Пирамидальная сортировка

Пирамидальная сортировка является методом, быстродействие которого оценивается как O(n log n).

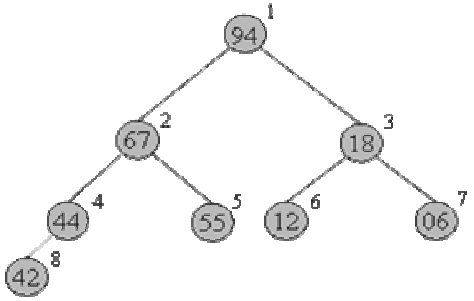
- Назовем пирамидой (Heap) бинарное дерево высоты k, в котором все узлы имеют глубину k или (k – 1) — дерево сбалансированное;
- при этом уровень (k – 1) полностью заполнен, а уровень k заполнен слева направо, т. е форма пирамиды имеет приблизительно такой вид:



;

- выполняется «свойство пирамиды»: каждый элемент меньше либо равен родителю.

Как хранить пирамиду? Наименее хлопотно поместить ее в массив.



Соответствие между геометрической структурой пирамиды как дерева и массивом устанавливается по следующей схеме:

- в a[0] хранится корень дерева;
- левый и правый сыновья элемента a[i] хранятся соответственно в a[2i + 1] и a[2i + 2].
-
- Поэтому для массива, хранящего в себе пирамиду, выполняется следующее свойство: a[i] >= a[2i + 1] и a[i] >= a[2i + 2].

Плюсы такого хранения пирамиды очевидны:

- никаких дополнительных переменных, нужно лишь понимать схему;
- узлы хранятся от вершины и далее вниз, уровень за уровнем;
- узлы одного уровня хранятся в массиве слева направо.

Запишем в виде массива пирамиду, изображенную выше. Слева направо, сверху вниз:

94 67 18 44 55 12 06 42.

На рисунке место элемента пирамиды в массиве обозначено цифрой справа и вверху от него.

Восстановить пирамиду из массива как геометрический объект легко: достаточно вспомнить схему хранения и нарисовать, начиная от корня.

Фаза 1 сортировки: построение пирамиды

Начать построение пирамиды можно с a[k]...a[n], k = [size / 2]. Эта часть массива удовлетворяет свойству пирамиды, так как не существует индексов i,j: i = 2i + 1 (или j = 2i + 2)... Просто потому, что такие i и j находятся за границей массива.

Следует заметить, что неправильно говорить о том, что a[k]..a[n] является пирамидой как самостоятельный массив. Это, вообще говоря, не верно: его элементы могут быть любыми. Свойство пирамиды сохраняется лишь в рамках исходного, основного массива a[0]...a[n].

Далее будем расширять часть массива, обладающую столь полезным свойством, добавляя по одному элементу за шаг. Следующий элемент на каждом шаге добавления — тот, который стоит перед уже готовой частью.

Чтобы при добавлении элемента сохранялась пирамидальность, будем использовать следующую процедуру расширения пирамиды a[i + 1]..a[n] на элемент a[i] влево:

1. Смотрим на сыновей слева и справа — в массиве это a[2i + 1] и a[2i + 2] и выбираем наибольшего из них.
2. Если этот элемент больше a[i] — меняем его с a[i] местами и идем к шагу 2, имея в виду новое положение a[i] в массиве. Иначе конец процедуры.

Новый элемент «просеивается» сквозь пирамиду.

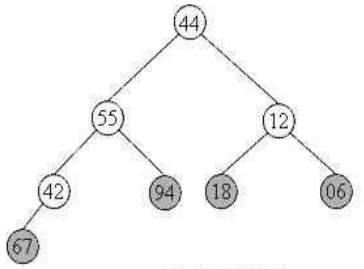
Ниже дана иллюстрация процесса для пирамиды из 8-и элементов:


```
 44   55   12   42 // 94   18   06   67
 44   55   12 // 67   94   18   06   42
 44   55 // 18   67   94   12   06   42
 44 // 94   18   67   55   12   06   42
// 94   67   18   44   55   12   06   42
```

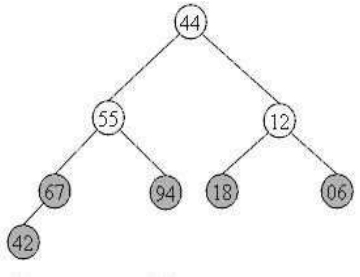
Справа — часть массива, удовлетворяющая свойству пирамиды. Остальные элементы добавляются один за другим, справа налево.

В геометрической интерпретации ключи из начального отрезка a[size / 2] ... a[n] является листьями в бинарном дереве, как изображено ниже. Один за другим остальные элементы продвигаются на свои места, и так будет продолжаться, пока не будет построена вся пирамида.

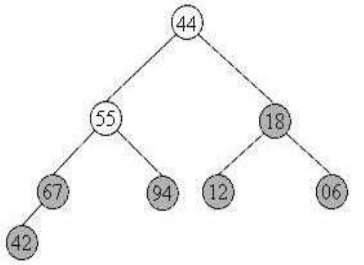
На рисунках ниже изображен процесс построения. Неготовая часть пирамиды (начало массива) окрашена в белый цвет, удовлетворяющий свойству пирамиды конец массива — в темный.



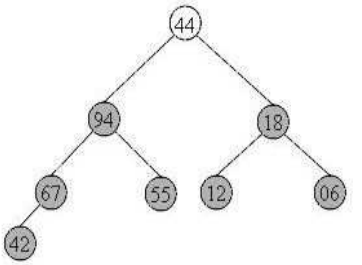
Исходная картина: 94, 18, 06, 67 – листья



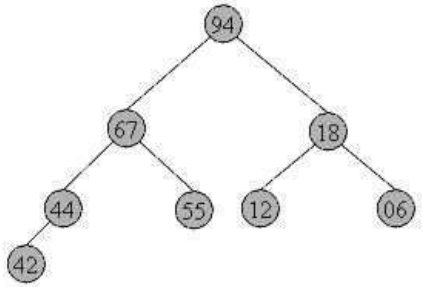
42 сравнили с 67 и поменяли их местами



12 сравнили с max (18, 6) = 18



55 сравнили с max (67, 94) = 94

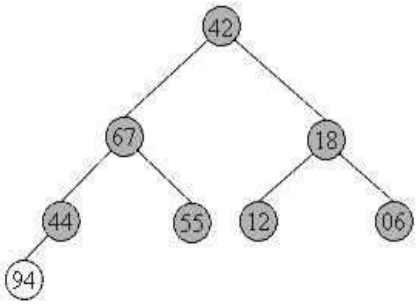


44 просеяли сквозь 94 и 67, остановились на 42

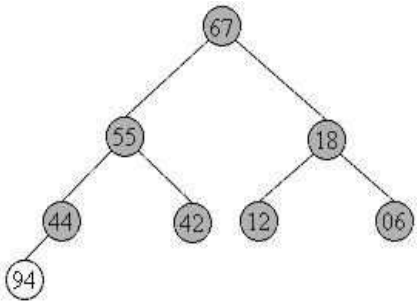
Фаза 2: собственно сортировка

Итак, задача построения пирамиды из массива успешно решена. Как видно из свойств пирамиды, в корне всегда находится максимальный элемент. Отсюда вытекает алгоритм фазы 2:

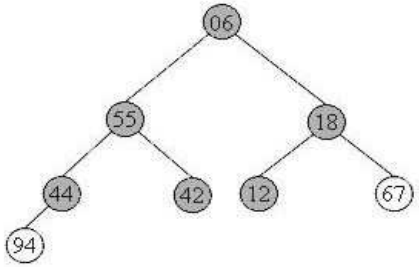
1. Берем верхний элемент пирамиды a[0]...a[n] (первый в массиве) и меняем его с последним элементом местами. Теперь «забываем» об этом элементе и далее рассматриваем массив a[0]...a[n–1]. Для превращения его в пирамиду достаточно просеять лишь новый первый элемент.
2. Повторяем шаг 1, пока обрабатываемая часть массива не уменьшится до одного элемента.



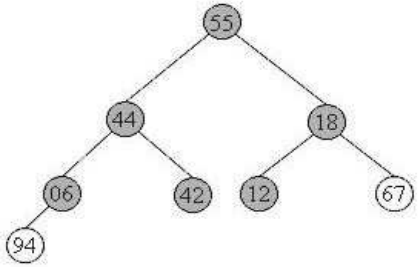
Обменяли 94 и 42, «забыли» о 94



Просеяли 42 сквозь 67 и 55



Обменяли 06 и 67, «забыли» о 67



Просеяли 06 сквозь 55 и 44

Очевидно, в конец массива каждый раз попадает максимальный элемент из текущей пирамиды, поэтому в правой части постепенно возникает упорядоченная последовательность.

Ниже приведена иллюстрация второй фазы сортировки во внутреннем представлении пирамиды.

```
94  67  18  44  55  12  06  42 //
67  55  44  06  42  18  12 // 94
55  42  44  06  12  18 // 67  94
44  42  18  06  12 // 55  67  94
42  12  18  06 // 44  55  67  94
18  12  06 // 42  44  55  67  94
12  06 // 18  42  44  55  67  94
06 // 12  18  42  44  55  67  94
```

- Пирамидальная сортировка не использует дополнительной памяти.
- Метод не является устойчивым: по ходу работы массив так «перетряхивается», что исходный порядок элементов может измениться случайным образом.
- Поведение неестественно: частичная упорядоченность массива никак не учитывается.

6. Производные типы данных. Списки данных

Из базовых типов языка С можно формировать производные типы данных, к которым относятся массивы, структуры и объединения. Также можно отнести сюда и строки, которых до сих пор не хватало, чтобы достаточно простым образом оперировать с различными именами и т. п.

6.1. Строки

Строка определяется как частный случай одномерного массива (поскольку состоит их набора символов). При этом последний элемент обязан быть равен **'/0'**, что обозначает конец строки.

Приведем пример определения строки и считывания имени пользователя, и затем его распечатки:

```
void main()
{
  char name[80];
  char zapros[80] = "Input your name: ";

  cout << zapros;

  cin >> name;
  cout << "Hello " << name << "!" << endl;
}
```

Если необходимо определить длину введенной строки, то можно использовать следующий способ:

```
int n;
for(n=0;;n++)
{
  if(name[n] == '\0')
    break;
}
cout << "Your name contain " << n << " letters";
```


6.2. Структурный тип

Этот тип задает внутреннее строение определяемых с помощью него структурных переменных. Часто применяется просто название **«структура»**.

Структура — это объединенное в единое целое множество поименованных элементов данных. В отличие от массивов данных, всегда состоящих из данных одного типа, структуры могут содержать различные типа данных. Например, может быть введена структура, описывающая студента университета со следующими компонентами (**полями**):

- имя;
- фамилия;
- номер группы;
- год рождения.

Для данного примера структура создается следующим образом:

```
struct Student
{
char Name[80];
char Surname[80];
int Group;
int Birth_year;
};
```

Таким образом, для того чтобы ввести в программу студентов определенного факультета, можно, например, создать массив, состоящий из таких структур данных. Это позволит упростить написание подобных программ.

Обратите внимание, что поскольку структурный тип данных является пользовательским, то он должен быть определен вне функций для обеспечения максимальной области видимости.

Переменная структурного типа создается таким же образом, как и любые другие переменные. Также можно создавать и указатели на

структуры. При этом объем необходимой памяти определяется с помощью функции **sizeof()**, которой в качестве параметра можно передать имя или тип структуры.

Доступ к полям структур осуществляется следующим образом:

Имя_Объекта . поле

либо ***Указатель_на_Объект -> поле***

Приведем в качестве примера фрагмент кода, в котором создается переменная структурного типа, осуществляется инициализация всех полей и распечатка через указатель на структуру.

```
#include <iostream>
using namespace std;

struct Student
{
  char Name[80];
  char Surname[80];
  int Group;
  int Birth_year;
};
void main()
{
  Student Vasya;

  cin >> Vasya.Name;
  cin >> Vasya.Surname;
  cin >> Vasya.Group;
  cin >> Vasya.Birth_year;

  Student* pVasya;
  pVasya = &Vasya;

  cout << pVasya->Name;
  cout << pVasya->Surname;
  cout << pVasya->Group;
  cout << pVasya->Birth_year;
}
```


6.3. Списки данных

Списки данных являются в некотором роде альтернативой динамическим массивам. В некоторых задачах, например база данных о студентах университета или телефонная книга, пользователь сам зачастую не знает, какое количество записей у него будет. Таким образом, нужен иной инструмент, позволяющий по ходу программы изменять количество элементов в некоей базе данных. Таким инструментом и являются списки данных.

Список — это упорядоченная последовательность связанных данных, связанных между собой. При этом списки бывают двух основных видов, в зависимости от характера связей элементов: односвязные и двусвязные.

Для хранения списка, и организации работы с ним достаточно лишь одного указателя, который будет хранить адрес первого элемента списка. Зная первый элемент, вы всегда сможете последовательно пройти по списку и извлечь любой элемент.

Помните, что если вы измените значение указателя начала списка (например перейдете к другому элементу в односвязном списке), то вы можете навсегда потерять первый элемент. Поэтому на практике, никогда не работают с указателем на начало списка, а создают временные указатели.

6.3.1. Односвязный список данных

Односвязный список характеризуется наличием одной связи у соседних элементов. Другими словами, каждый элемент знает об одном соседе (рис. 11). По такому списку можно передвигаться только последовательно в одном направлении: от начала к концу.

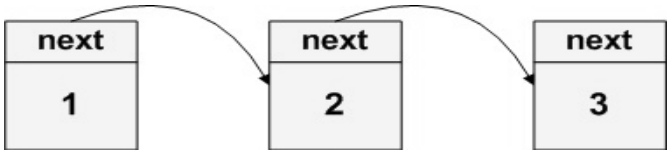


Рис. 11. Односвязный список

Разберем, каким образом будет выглядеть элемент односвязного списка, содержащий данные о студенте университета (описанные выше) и указатель на следующий элемент списка.

```
struct Student
{
   char Name[80];
   char Surname[80];
   int Group;
   int Birth_year;

   Student* pNext;
};
```

Заметим, что если программа компилируется с использованием C-компилятора, в типе указателя необходимо указывать не просто имея структуры, а именно с указанием того что это структура. Это связано с тем что определение структурного типа еще не закончено.

```
struct Student* pNext;
```

6.3.2. Двусвязный список данных

Двусвязный список отличается наличием двух связей у каждого элемента. При этом организуется двунаправленность списка (рис. 12). Другими словами, зная любой элемент списка, можно получить информацию как о следующем элементе списка, так и о предыдущем.

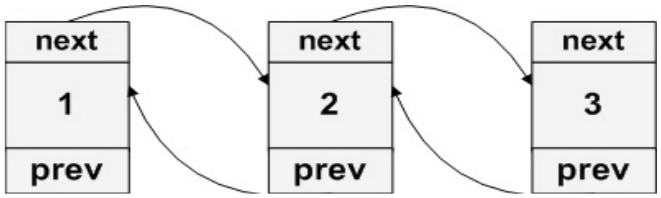


Рис.12. Двусвязный список

Для управления списком необходимо знать только первый его элемент — начало списка. Последний элемент определяется по признаку того, что следующего за ним не существует, т. е. указатель равен NULL (рис. 13).

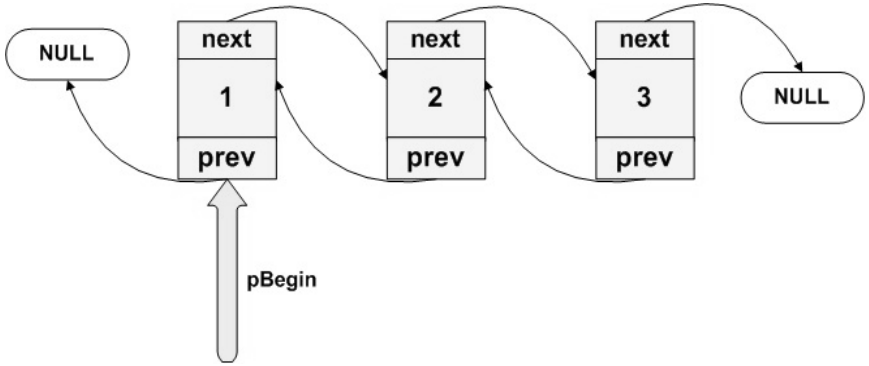


Рис.13. Указатель на начало двусвязного списка

Если следующего элемента не существует, то указатель pnext необходимо задать равным NULL.

6.3.3. Распечатка элементов списка

Функция для распечатки существующего списка может выглядеть следующим образом (в качестве параметра функция получает указатель на первый элемент списка):

```
void PrintAll(Student* pBegin)
{
   if (pBegin == NULL)
   {
      printf("No elements\n");
      return;
   }

   Student* pCur = pBegin;

   do
   {
      cout << pCur->Name;
      cout << pCur->Surname;
      cout << pCur->Group;
      cout << pCur->Birth_year;

      pCur = pCur->pNext;
   }
   while(pCur != NULL);
}
```

6.3.4. Добавление элемента в существующий список

Разберем на примере односвязного списка студентов процесс добавления элемента и реализуем для этого функцию.

В качестве входных параметров функция должна получить указатель на существующий список, собственно указатель на добавляемый элемент и номер, которым он должен стать в списке.

При добавлении элемента в список возможны три различных ситуации:

- список пуст, соответственно независимо от номера, элемент становится первым;
- элемент необходимо вставить в конец списка;
- элемент вставляется в середину списка.

Помните, что если элемент оказывается последним, то необходимо в поле указателя на следующий элемент прописать **NULL**.


```
void AddElement(Student* pBegin,
Student* pAdd, int number)
{
   // Если список был пуст - на первое место
   if(pBegin == NULL)
   {
      pBegin = pAdd;
      pAdd->pNext = NULL;
      return;
   }

   // Ищем в списке элемент
   // за которым должны вставить
   int n = 0;
   Student* pCur = pBegin;

   for(n = 0; n < number; n++)
      pCur = pCur->pNext;

   // Если нужно вставить последним
   if(pCur->pNext == NULL)
   {
      pCur->pNext = pAdd;
      pAdd->pNext = NULL;
      return;
   }

   // Определяем следующий элемент
   Student* pNext = pCur->pNext;

   // Вставляем элемент в список
   pCur->pNext = pAdd;
   pAdd->pNext = pNext;
}
```

Обратите внимание, что в приведенном примере не производится обработка ситуаций с некорректным номером элемента.

7. Работа с файлами

Работа с файлами осуществляется различным способом в языках C и C++. Далее разобрано оба подхода, поскольку они оба актуальны и могут использоваться как в существубщих, так и во вновь разрабатываемых программах.

7.1. Файлы в языке C

Первоначально язык C был реализован в операционной системе UNIX. Как таковые, ранние версии C (да и многие нынешние) поддерживают набор функций ввода/вывода, совместимый с UNIX. Этот набор иногда называют UNIX-подобной системой ввода/вывода или небуферизованной системой ввода/вывода. Однако когда C был стандартизован, то UNIX-подобные функции в него не вошли — в основном из-за того, что оказались лишними. Кроме того, UNIX-подобная система может оказаться неподходящей для некоторых сред, которые могут поддерживать язык C, но не эту систему ввода/вывода.

В этой главе описана работа с файловой системой в языке C. В языке C система ввода/вывода реализуется с помощью библиотечных функций, а не ключевых слов. Благодаря этому система ввода/вывода является очень мощной и гибкой. Например, во время работы с файлами данные могут передаваться или в своем внутреннем двоичном представлении или в текстовом формате, то есть в более удобочитаемом виде. Это облегчает задачу создания файлов в нужном формате.

Библиотека C поддерживает три уровня ввода-вывода: потоковый ввод-вывод, ввод-вывод нижнего уровня и ввод-вывод для консоли и портов.

7.1.1. Потоки и файлы

Перед тем как начать изучение файловой системы языка C, необходимо уяснить, в чем разница между потоками и файлами. В системе ввода/вывода C для программ поддерживается единый интерфейс, не

зависящий от того, к какому конкретному устройству осуществляется доступ. То есть в этой системе между программой и устройством находится нечто более общее, чем само устройство. Такое обобщенное устройство ввода или вывода (устройство более высокого уровня абстракции) называется потоком, в то время как конкретное устройство называется файлом. (Впрочем, файл — тоже понятие абстрактное.) Очень важно понимать, каким образом происходит взаимодействие потоков и файлов.

Файловая система языка C предназначена для работы с самыми разными устройствами, в том числе терминалами, дисководами и накопителями на магнитной ленте. Даже если какое-то устройство сильно отличается от других, буферизованная файловая система все равно представит его в виде логического устройства, которое называется потоком. Все потоки ведут себя похожим образом. И так как они в основном не зависят от физических устройств, то та же функция, которая выполняет запись в дисковый файл, может ту же операцию выполнять и на другом устройстве, например, на консоли. Потоки бывают двух видов: текстовые и двоичные.

Текстовый поток — это последовательность символов. В стандарте C считается, что текстовый поток организован в виде строк, каждая из которых заканчивается символом новой строки. Однако в конце последней строки этот символ не является обязательным. В текстовом потоке по требованию базовой среды могут происходить определенные преобразования символов. Например, символ новой строки может быть заменен парой символов — возврата каретки и перевода строки. Поэтому может и не быть однозначного соответствия между символами, которые пишутся (читаются), и теми, которые хранятся во внешнем устройстве. Кроме того, количество тех символов, которые пишутся (читаются), и тех, которые хранятся во внешнем устройстве, может также не совпадать из-за возможных преобразований.

Двоичный поток — это последовательность байтов, которая взаимно однозначно соответствует байтам на внешнем устройстве, причем никакого преобразования символов не происходит. Кроме того, количество тех байтов, которые пишутся (читаются), и тех, которые хранятся на внешнем устройстве, одинаково. Однако в конце двоичного потока может добавляться определяемое приложением количество нулевых байтов. Такие нулевые байты, например, могут использоваться для заполнения свободного места в блоке памяти незначащей информацией, чтобы она в точности заполнила сектор на диске.

В языке С файлом может быть все что угодно, начиная с дискового файла и заканчивая терминалом или принтером. Поток связывают с определенным файлом, выполняя операцию открытия. Как только файл открыт, можно проводить обмен информацией между ним и программой.

Но не у всех файлов одинаковые возможности. Например, к дисковому файлу прямой доступ возможен, в то время как к некоторым принтерам — нет. Таким образом, мы пришли к одному важному принципу, относящемуся к системе ввода/вывода языка С: все потоки одинаковы, а файлы — нет.

Если файл может поддерживать запросы на местоположение (указатель текущей позиции), то при открытии такого файла указатель текущей позиции в файле устанавливается в начало. При чтении из файла (или записи в него) каждого символа указатель текущей позиции увеличивается, обеспечивая тем самым продвижение по файлу.

Файл отсоединяется от определенного потока (т.е. разрывается связь между файлом и потоком) с помощью операции закрытия. При закрытии файла, открытого с целью вывода, содержимое (если оно есть) связанного с ним потока записывается на внешнее устройство. Этот процесс, который обычно называют дозаписью потока, гарантирует, что никакая информация случайно не останется в буфере диска. Если программа завершает работу нормально, т.е. либо main() возвращает управление

операционной системе, либо вызывается exit(), то все файлы закрываются автоматически. В случае аварийного завершения работы программы, например, в случае краха или завершения путем вызова abort(), файлы не закрываются.

У каждого потока, связанного с файлом, имеется управляющая структура, содержащая информацию о файле; она имеет тип FILE. В этом блоке управления файлом никогда ничего не меняйте.

Если вы новичок в программировании, то разграничение потоков и файлов может показаться излишним или даже "заумным". Однако надо помнить, что основная цель такого разграничения — это обеспечить единый интерфейс. Для выполнения всех операций ввода/вывода следует использовать только понятия потоков и применять всего лишь одну файловую систему. Ввод или вывод от каждого устройства автоматически преобразуется системой ввода/вывода в легко управляемый поток.

Функции библиотеки ввода-вывода языка С, поддерживающие обмен данными с файлами на уровне потока, позволяют обрабатывать данные различных размеров и форматов, обеспечивая при этом буферизованный ввод и вывод. Таким образом, поток — это файл вместе с предоставляемыми средствами буферизации.

7.1.2. Основы файловой системы

Файловая система языка С состоит из нескольких взаимосвязанных функций. Самые распространенные из них показаны в ниже таблице. Для их работы требуется заголовок <stdio.h>.

Часто используемые функции файловой системы С	
fopen()	Открывает файл
fclose()	Закрывает файл
putc()	Записывает символ в файл

fputc()	То же, что и putc()
getc()	Читает символ из файла
fgetc()	То же, что и getc()
fgets()	Читает строку из файла
fputs()	Записывает строку в файл
fseek()	Устанавливает указатель позиции на определенный байт
ftell()	Возвращает текущее значение указателя в файле
fprintf()	Для файла то же, что printf() для консоли
fscanf()	Для файла то же, что scanf() для консоли
feof()	Возвращает значение true, если достигнут конец файла
ferror()	Возвращает значение true, если произошла ошибка
rewind()	Устанавливает указатель позиции в начало файла
remove()	Стирает файл
fflush()	Дозапись потока в файл

Заголовок <stdio.h> предоставляет прототипы функций ввода/вывода и определяет следующие три типа: size_t, fpos_t и FILE. size_t и fpos_t представляют собой определенные разновидности такого типа, как целое без знака. А о третьем типе, FILE, рассказывается в следующем разделе.

Кроме того, в <stdio.h> определяется несколько макросов. Из них к материалу этой главы относятся NULL, EOF, FOPEN_MAX, SEEK_SET, SEEK_CUR и SEEK_END. Макрос NULL определяет пустой (null) указатель. Макрос EOF, часто определяемый как -1, является значением, возвращаемым тогда, когда функция ввода пытается выполнить чтение после конца файла. FOPEN_MAX определяет целое значение, равное максимальному числу одновременно открытых файлов. Другие макросы

используются вместе с fseek() — функцией, выполняющей операции прямого доступа к файлу.

7.1.3. Указатель файла

Указатель файла — это то, что соединяет в единое целое всю систему ввода/вывода языка С. Указатель файла — это указатель на структуру типа FILE. Он указывает на структуру, содержащую различные сведения о файле, например, его имя, статус и указатель текущей позиции в начало файла. В сущности, указатель файла определяет конкретный файл и используется соответствующим потоком при выполнении функций ввода/вывода. Чтобы выполнять в файлах операции чтения и записи, программы должны использовать указатели соответствующих файлов. Чтобы объявить переменную-указатель файла, используйте такого рода оператор:

```
FILE *fp;
```

7.1.4. Открытие файла

Функция fopen() открывает поток и связывает с этим потоком определенный файл. Затем она возвращает указатель этого файла. Чаще всего (а также в оставшейся части этой главы) под файлом подразумевается дисковый файл. Прототип функции fopen() следующий:

FILE *fopen(const char *имя_файла, const char *режим);

где имя_файла — это указатель на строку символов, представляющую собой допустимое имя файла, в которое также может входить спецификация пути к этому файлу. Строка, на которую указывает режим, определяет, каким образом файл будет открыт. Ниже в таблице показано, какие значения строки режим являются допустимыми. Строки, подобные "r+b" могут быть представлены и в виде "rb+".

Допустимые значения режим	
r	Открыть текстовый файл для чтения
w	Создать текстовый файл для записи
a	Добавить в конец текстового файла
rb	Открыть двоичный файл для чтения
wb	Создать двоичный файл для записи
ab	Добавить в конец двоичного файла
r+	Открыть текстовый файл для чтения/записи
w+	Создать текстовый файл для чтения/записи
a+	Добавить в конец текстового файла или создать текстовый файл для чтения/записи
r+b	Открыть двоичный файл для чтения/записи
w+b	Создать двоичный файл для чтения/записи
a+b	Добавить в конец двоичного файла или создать двоичный файл для чтения/записи

Как уже упоминалось, функция fopen() возвращает указатель файла. Никогда не следует изменять значение этого указателя в программе. Если при открытии файла происходит ошибка, то fopen() возвращает пустой (null) указатель.

В следующем коде функция fopen() используется для открытия файла по имени TEST для записи.

```
FILE *fp;
fp = fopen("test", "w");
```

Хотя предыдущий код технически правильный, но его обычно пишут немного по-другому:


```
FILE *fp;

if ((fp = fopen("test","w"))==NULL)
  printf("Ошибка при открытии файла.\n");
```

Этот метод помогает при открытии файла обнаружить любую ошибку, например, защиту от записи или полный диск, причем обнаружить еще до того, как программа попытается в этот файл что-либо записать. Вообще говоря, всегда нужно вначале получить подтверждение, что функция - fopen() выполнилась успешно, и лишь затем выполнять с файлом другие операции.

Хотя название большинства файловых режимов объясняет их смысл, однако не помешает сделать некоторые дополнения. Если попытаться открыть файл только для чтения, а он не существует, то работа fopen() завершится отказом. А если попытаться открыть файл в режиме дозаписи, а сам этот файл не существует, то он просто будет создан. Более того, если файл открыт в режиме дозаписи, то все новые данные, которые записываются в него, будут добавляться в конец файла. Содержимое, которое хранилось в нем до открытия (если только оно было), изменено не будет. Далее, если файл открывают для записи, но выясняется, что он не существует, то он будет создан. А если он существует, то содержимое, которое хранилось в нем до открытия, будет утеряно, причем будет создан новый файл. Разница между режимами r+ и w+ состоит в том, что если файл не существует, то в режиме открытия r+ он создан не будет, а в режиме w+ все произойдет наоборот: файл будет создан! Более того, если файл уже существует, то открытие его в режиме w+ приведет к утрате его содержимого, а в режиме r+ оно останется нетронутым.

Из таблицы видно, что файл можно открыть либо в одном из текстовых, либо в одном из двоичных режимов. В большинстве реализаций в текстовых режимах каждая комбинация кодов возврата каретки (ASCII 13) и конца строки (ASCII 10) преобразуется при вводе в символ новой строки. При выводе же происходит обратный процесс:

символы новой строки преобразуются в комбинацию кодов возврата каретки (ASCII 13) и конца строки (ASCII 10). В двоичных режимах такие преобразования не выполняются.

Максимальное число одновременно открытых файлов определяется FOPEN_MAX. Это значение не меньше 8, но чему оно точно равняется – это должно быть написано в документации по компилятору.

7.1.5. Закрытие файла

Функция fclose() закрывает поток, который был открыт с помощью вызова fopen().Функция fclose() записывает в файл все данные, которые еще оставались в дисковом буфере, и проводит, так сказать, официальное закрытие файла на уровне операционной системы. Отказ при закрытии потока влечет всевозможные неприятности, включая потерю данных, испорченные файлы и возможные периодические ошибки в программе. Функция fclose() также освобождает блок управления файлом, связанный с этим потоком, давая возможность использовать этот блок снова. Так как количество одновременно открытых файлов ограничено, то, возможно, придется закрывать один файл, прежде чем открывать другой. Прототип функции fclose() такой:

```
int fclose(FILE * файл);
```

Возвращение нуля означает успешную операцию закрытия. В случае же ошибки возвращается EOF. Чтобы точно узнать, в чем причина этой ошибки, можно использовать стандартную функцию ferror() (о которой вскоре пойдет речь). Обычно отказ при выполнении fclose() происходит только тогда, когда диск был преждевременно удален (стерт) с дисковода или на диске не осталось свободного места.

7.1.6. Запись символа

В системе ввода/вывода языка С определяются две эквивалентные функции, предназначенные для вывода символов: putc() и fputc(). (На самом деле putc() обычно реализуется в виде макроса.) Две идентичные функции имеются просто потому, чтобы сохранять совместимость со старыми версиями С. В этой книге используется putc(), но применение fputc() также вполне возможно.

Функция putc() записывает символы в файл, который с помощью fopen() уже открыт в режиме записи. Прототип этой функции следующий:

```
int putc(int ch, FILE * файл);
```

Указатель файла сообщает putc(), в какой именно файл следует записывать символ. Хотя ch и определяется как int, однако записывается только младший байт.

Если функция putc() выполнилась успешно, то возвращается записанный символ. В противном же случае возвращается EOF.

7.1.7. Чтение символа

Для ввода символа также имеются две эквивалентные функции: getc() и fgetc(). Обе определяются для сохранения совместимости со старыми версиями С. В этой книге используется getc() (которая обычно реализуется в виде макроса), но если хотите, применяйте fgetc().

Функция getc() записывает символы в файл, который с помощью fopen() уже открыт в режиме для чтения. Прототип этой функции следующий:

```
int getc(FILE * файл);
```

Функция getc() возвращает целое значение, но символ находится в младшем байте. Если не произошла ошибка, то старший байт (байты) будет обнулен.

Если достигнут конец файла, то функция getc() возвращает EOF. Поэтому, чтобы прочитать символы до конца текстового файла, можно использовать следующий код:

```
do
{
  ch = getc(fp);
}
while(ch!=EOF);
```

Однако getc() возвращает EOF и в случае ошибки. Для определения того, что же на самом деле произошло, можно использовать ferror().

7.1.8. Использование fopen(), getc(), putc(), и fclose()

Функции fopen(), getc(), putc() и fclose() — это минимальный набор функций для операций с файлами. Следующая программа, представляет собой простой пример, в котором используются только функции putc(), fopen() и fclose(). В этой программе символы считываются с клавиатуры и записываются в дисковый файл до тех пор, пока пользователь не введет знак доллара. Имя файла определено в программа «test.txt».

```
void main()
{
  FILE *fp;  char ch;

  if((fp=fopen("test.txt", "w"))==NULL)
  {
    printf("ERROR\n");
    return;
  }

  do {
```



```
    ch = getchar();
    putc(ch, fp);
  }
  while (ch != '$');

  fclose(fp);
}
```

Следующая программа, читает любой текстовый файл и выводит его содержимое на экран.

```
void main()
{
  FILE *fp;  char ch;

  if((fp=fopen("test.txt", "r")) == NULL)
  {
    printf("ERROR.\n");
    return;
  }

  ch = getc(fp);

  while (ch!=EOF)
  {
    putchar(ch);
    ch = getc(fp);
  }

  fclose(fp);
}
```

7.1.9. Использование feof()

Как уже говорилось, если достигнут конец файла, то getc() возвращает EOF. Однако проверка значения, возвращенного getc(), возможно, не является наилучшим способом узнать, достигнут ли конец файла. Во-первых, файловая система языка С может работать как с текстовыми, так и с двоичными файлами. Когда файл открывается для двоичного ввода, то может быть прочитано целое значение, которое, как выяснится при

проверке, равняется EOF. В таком случае программа ввода сообщит о том, что достигнут конец файла, чего на самом деле может и не быть. Во-вторых, функция getc() возвращает EOF и в случае отказа, а не только тогда, когда достигнут конец файла. Если использовать только возвращаемое значение getc(), то невозможно определить, что же на самом деле произошло. Для решения этой проблемы в C имеется функция feof(), которая определяет, достигнут ли конец файла. Прототип функции feof() такой:

```
int feof(FILE * файл);
```

Если достигнут конец файла, то feof() возвращает true (истина); в противном же случае эта функция возвращает нуль. Поэтому следующий код будет читать двоичный файл до тех пор, пока не будет достигнут конец файла:

```
while(!feof(fp)) ch = getc(fp);
```

Ясно, что этот метод можно применять как к двоичным, так и к текстовым файлам.

В следующей программе, которая копирует текстовые или двоичные файлы, имеется пример применения feof(). Файлы открываются в двоичном режиме, а затем feof() проверяет, не достигнут ли конец файла.

```
void main()
{
  FILE *in, *out;  char ch;

  if((in=fopen("in.txt", "rb"))==NULL)
  {
    printf("ERROR\n");
    return;
  }
```



```
  if((out=fopen("out.txt", "wb")) == NULL)
  {
    printf("ERROR\n");
    return;
  }

  while(!feof(in))
  {
    ch = getc(in);
    if(!feof(in)) putc(ch, out);
  }

  fclose(in);
  fclose(out);
}
```

7.1.10. Ввод / вывод строк: fputs() и fgets()

Кроме getc() и putc(), в языке C также поддерживаются родственные им функции fgets() и fputs(). Первая из них читает строки символов из файла на диске, а вторая записывает строки такого же типа в файл, тоже находящийся на диске. Эти функции работают почти как putc() и getc(), но читают и записывают не один символ, а целую строку. Прототипы функций fgets() и fputs() следующие:

```
int fputs(const char *cmp, FILE * файл);
char *fgets(char *cmp, int длина, FILE * файл);
```

Функция fputs() пишет в определенный поток строку, на которую указывает cmp. В случае ошибки эта функция возвращает EOF.

Функция fgets() читает из определенного потока строку, и делает это до тех пор, пока не будет прочитан символ новой строки или количество прочитанных символов не станет равным длина-1. Если был прочитан разделитель строк, он записывается в строку, чем функция fgets() отличается от функции gets(). Полученная в результате строка будет оканчиваться символом конца строки (***'0'***). При успешном завершении работы функция возвращает cmp, а в случае ошибки — пустой указатель (null).

В следующей программе показано использование функции fputs(). Она читает строки с клавиатуры и записывает их в файл, который называется TEST. Чтобы завершить выполнение программы, введите пустую строку. Так как функция gets() не записывает разделитель строк, то его приходится специально вставлять перед каждой строкой, записываемой в файл; это делается для того, чтобы файл было легче читать:

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

void main(void)
{
  char str[80]; FILE *fp;

  if((fp = fopen("test.txt", "w"))==NULL)
  { printf("Error\n");
    return;
  }

  do{
    printf("Input string\n");
    gets(str);
    strcat(str, "\n");
    fputs(str, fp);
  } while(*str!='\n');

  fclose(fp);
}
```

7.1.11. Функция rewind()

Функция rewind() устанавливает указатель текущей позиции в файле на начало файла, указанного в качестве аргумента этой функции. Иными словами, функция rewind() выполняет "перемотку" (rewind) файла. Вот ее прототип:

```
void rewind(FILE * файл);
```


Чтобы познакомиться с rewind(), изменим программу из предыдущего раздела таким образом, чтобы она отображала содержимое файла сразу после его создания. Чтобы выполнить отображение, программа после завершения ввода "перематывает" файл, а затем с помощью fback() читает его с самого начала. Обратите внимание, что сейчас файл необходимо открыть в режиме чтения/записи, используя в качестве аргумента, задающего режим, строку "w+".

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

void main(void)
{
  char str[80]; FILE *fp;

  if((fp = fopen("test.txt", "w+"))==NULL)
  {
    printf("Ошибка при открытии файла.\n"); return;
  }
  do
  {
    printf("Введите строку (пустую - для
выхода):\n");
    gets(str);
    strcat(str, "\n");  /* ввод разделителя строк */
    fputs(str, fp);
  }
  while(*str!='\n');

  rewind(fp);

  while(!feof(fp))
  {
    fgets(str, 79, fp);
    printf(str);
  }
  fclose(fp);
}
```

7.1.12. Функция ferror()

Функция ferror() определяет, произошла ли ошибка во время выполнения операции с файлом. Прототип этой функции следующий:

```
int ferror(FILE *файл);
```

Функция возвращает значение true (истина), если при последней операции с файлом произошла ошибка; в противном же случае она возвращает false (ложь). Так как при любой операции с файлом устанавливается свое условие ошибки, то после каждой такой операции следует сразу вызывать ferror(), а иначе данные об ошибке могут быть потеряны.

7.1.13. Стирание файлов

Функция remove() стирает указанный файл. Вот ее прототип:

```
int remove(const char *имя_файла);
```

В случае успешного выполнения эта функция возвращает нуль, а в противном случае — ненулевое значение.

7.1.14. Функции fread() и fwrite()

Для чтения и записи данных, тип которых может занимать более 1 байта, в файловой системе языка С имеется две функции: fread() и fwrite(). Эти функции позволяют читать и записывать блоки данных любого типа. Их прототипы следующие:

```
size_t fread(void *буфер, size_t колич_байт,
             size_t счетчик, FILE * файл);

size_t fwrite(const void *буфер, size_t колич_байт,
              size_t счетчик, FILE * файл);
```


Для fread() буфер — это указатель на область памяти, в которую будут прочитаны данные из файла. А для fwrite() буфер — это указатель на данные, которые будут записаны в файл. Значение счетчик определяет, сколько считывается или записывается элементов данных, причем длина каждого элемента в байтах равна колич_байт. (Вспомните, что тип size_t определяется как одна из разновидностей целого типа без знака.) И, наконец, уф — это указатель файла, то есть на уже открытый поток.

Функция fread() возвращает количество прочитанных элементов. Если достигнут конец файла или произошла ошибка, то возвращаемое значение может быть меньше, чем счетчик. А функция fwrite() возвращает количество записанных элементов. Если ошибка не произошла, то возвращаемый результат будет равен значению счетчик.

Как только файл открыт для работы с двоичными данными, fread() и fwrite() соответственно могут читать и записывать информацию любого типа. Например, следующая программа записывает в дисковый файл данные типов double, int и long, а затем читает эти данные из того же файла. Обратите внимание, как в этой программе при определении длины каждого типа данных используется функция sizeof().

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

void main(void)
{
  FILE *fp;
  double d = 12.23;
  int i = 101;
  long l = 123023L;

  if((fp=fopen("test", "wb+"))==NULL) {
    printf("Error");
```

```
    return;
  }

  fwrite(&d, sizeof(double), 1, fp);
  fwrite(&i, sizeof(int), 1, fp);
  fwrite(&l, sizeof(long), 1, fp);

  rewind(fp);

  fread(&d, sizeof(double), 1, fp);
  fread(&i, sizeof(int), 1, fp);
  fread(&l, sizeof(long), 1, fp);

  printf("%f %d %ld", d, i, l);

  fclose(fp);
}
```

Как видно из этой программы, в качестве буфера можно использовать (и часто именно так и делают) просто память, в которой размещена переменная. В этой простой программе значения, возвращаемые функциями fread() и fwrite(), игнорируются. Однако на практике эти значения необходимо проверять, чтобы обнаружить ошибки.

Одним из самых полезных применений функций fread() и fwrite() является чтение и запись данных пользовательских типов, особенно структур. Например, если определена структура:

```
struct struct_type
{
  float balance;
  char name[80];
} cust;
```

то следующий оператор записывает содержимое cust в файл, на который указывает fp:

```
fwrite(&cust, sizeof(struct struct_type), 1, fp);
```


7.1.15. Функции fprinf() и fscanf()

Кроме основных функций ввода/вывода, о которых шла речь, в системе ввода/вывода языка С также имеются функции fprintf() и fscanf(). Эти две функции, за исключением того, что предназначены для работы с файлами, ведут себя точно так же, как и printf() и scanf(). Прототипы функций fprintf() и fscanf() следующие:

```
int fprintf(FILE *файл, const char *строка, ...);
int fscanf(FILE *файл, const char *строка, ...);
```

Операции ввода/вывода функции fprintf() и fscanf() выполняют с тем файлом, указанным в параметрах функции.

В качестве примера предлагается рассмотреть следующую программу, которая читает с клавиатуры строку и целое значение, а затем записывает их в файл на диске; имя этого файла — **test**. После этого программа читает этот файл и выводит информацию на экран. После запуска программы проверьте, каким получится файл **test**. Как вы и увидите, в нем будет вполне удобочитаемый текст.

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

void main(void)
{
  FILE *fp;
  char s[80];
  int t;

  if((fp=fopen("test", "w")) == NULL)
  {
    printf("Error\n");
    return;
  }
```

```
    printf("Введите строку и число: ");

    /* читать с клавиатуры */
    fscanf(stdin, "%s%d", s, &t);

     /* писать в файл */
    fprintf(fp, "%s %d", s, t);
    fclose(fp);

    if((fp=fopen("test","r")) == NULL)
    {
      printf("Error\n");
      return;
    }

    fscanf(fp, "%s%d", s, &t);
    fprintf(stdout, "%s %d", s, t);
}
```

Предупреждение: хотя читать разносортные данные из файлов на дисках и писать их в файлы, расположенные также на дисках, часто легче всего именно с помощью функций fprintf() и fscanf(), но это не всегда самый эффективный способ выполнения операций чтения и записи. Так как данные в формате ASCII записываются так, как они должны появиться на экране (а не в двоичном виде), то каждый вызов этих функций сопряжен с определенными накладными расходами. Поэтому, если надо заботиться о размере файла или скорости, то, скорее всего, придется использовать fread() и fwrite().

7.2. Файлы в языке C++

В данном разделе будут рассмотрены только основы файловых потоков в C++, поскольку более детальное изучение требует знаний основ объектно-ориентированного подхода в программировании.

7.2.1. Открытие и закрытие файловых потоков

Для ввода-вывода информации на консоль мы используем стандартные потока ввода-вывода ***cin/cout***. Это существующие стандартные потоки, связанные с консолью. В случае работы с файлами, нам необходимо:

1. подключить библиотеку файловых потоков fstream
2. создать отдельный поток, ассоциировав его с конкретным файлом.

При этом необходимо напомнить, что файловые потоки, также как и потоки ввода-вывода на консоль имеют направление.

Для того чтобы открыть файловый поток для чтения данных из файла необходимо написать:

```
#include <fstream>
using namespace std;

void main()
{
  ifstream input_file("FileName");
}
```

Для того чтобы открыть файловый поток для записи данных в файл необходимо написать:

```
#include <fstream>
using namespace std;

void main()
{
  ofstream output_file("FileName");
}
```

После того как файловый поток больше не нужен - его необходимо закрыть:

```
input_file.close();
```

7.2.2. Посимвольное чтение и запись текстовых файлов

Работа с файловыми потоками (после того как они открыты) ничем не отличается от работы со стандартными потоками ввода-вывода.

Приведем пример программы для посимвольного чтения данных из текстового файла и вывода из на консоль. При этом концом файла будем считать точку (".").

```
#include <iostream>
#include <fstream>
using namespace std;

void main()
{
   ifstream ifs("1.txt");

   if (!ifs.is_open())
   {
    cout << "Can not open input file" << endl;
    return;
   }
   char ch;
   do
   {
    ch = ifs.get();
    if(ch == '.')
      break;
    cout << ch;
   }while(1);

   cout << endl;

   ifs.close();
}
```

А теперь наоборот, будем считывать символы из консоли до ввода точки и записывать их в файл.


```
#include <iostream>
#include <fstream>
using namespace std;

void main()
{
  ofstream ofs("1.txt");

  if (!ofs.is_open())
  {
    cout << "Can not open output file" << endl;
    return;
  }

  char ch;
  do
  {
    cin >> ch;
    if(ch == '.')
      break;
    ofs << ch;
  }while(1);

  ofs << endl;

  ofs.close();
}
```

7.2.3. Чтение и запись текстовых файлов по словам

По умолчанию слова из файла читаются также как и с консоли, то есть разделителями слов служат пробелы, символы табуляции и переводы строк. Поэтому при чтении по словам и выводе данных на экран необходимо дополнительно вставлять символы разделения строк.

В качестве примера приведем программу чтения из файла слов и распечатки их на консоль в столбец.

```
#include <iostream>
#include <fstream>
using namespace std;

void main()
{
   ifstream ifs("1.txt");

   if (!ifs.is_open())
   {
    cout << "Can not open intput file" << endl;
    return;
   }


   char word[80];
   while (1)
   {
    ifs >> word;
    if(word[0] == '.')
      break;
    cout << word << endl;
   }
   cout << endl;

   ifs.close();
}
```

А теперь наоборот, будем считывать слова из консоли до ввода отдельной точки и записывать их в файл, разделяя при этом пробелом.

```
#include <iostream>
#include <fstream>
using namespace std;

void main()
{
  ofstream ofs("1.txt");
```



```
if (!ofs.is_open())
{
  cout << "Can not open output file" << endl;
  return;
}


char word[80];
while (1)
{
  cin >> word;
  if(word[0] == '.')
    break;
  ofs << word << endl;
}
ofs << endl;

ofs.close();
}
```

7.2.4. Определение конца файлов

В предыдущих примерах мы опирались на базовой предположение что файл заканчивается специальным символом (точкой). Причем в случае чтения по словам "точка" должна быть отделена пробелом от предыдущего слова! Такой способ разумеется не особо удобен при работе, поэтому чаще всего пользуются функцией определения конца файлового потока eof().

Приведем пример программы чтение текстового файла по славам с использованием функции определения конца файлового потока.

```
#include <iostream>
#include <fstream>
using namespace std;

void main()
{
   ifstream ifs("1.txt");
```

```
    if (!ifs.is_open())
    {
      cout << "Can not open output file" << endl;
      return;
    }


    char word[80];
    while (!ifs.eof())
    {
      ifs >> word;
      cout << word << endl;
    }
    cout << endl;

    ifs.close();
}
```


8. ВВЕДЕНИЕ В ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

Данное учебное пособие основывается на знание структурного программирования и языка С, поскольку его синтаксис во многом схож с языком С++.

8.1. С и С++.

Основной идеей языка С, является использование структурного программирования. При такой точке зрения, задача рассматривается как набор данных (структур) и действия, которые с ними происходят. Другими словами можно сказать что все данные хранятся в определенных структурах, а за поведение программы отвечают функции, оперирующие с ними.

Однако достаточно часто, особенно при работе над большими программами и проектами, становится нереальным отделить сами данные от их поведения. Например, при решении какой либо трудоемкой задачи, может кончиться оперативная память ПК, или кончится место на диске компьютера, отвечающего за процесс автоматизации многомесячного эксперимента и так далее. Все это приводит к усложнению модели поведения и очень сильной связи работы алгоритмов к состоянию данных.

Основная концепция объектно-ориентированных языков программирования – введение понятия класса (class) как объединения данных (структур) и поведения (функций). То есть в одном целом объединено и хранение данных, и поведение, и собственно контроль за их правильным состоянием и обработкой ошибок. Все это приводит к тому, что при сборе программной системы, состоящей из нескольких классов существенно упрощается контроль над корректностью работы каждого компонента в отдельности.

8.2. История языка C++

Язык C++ был разработан Бьёрном Страуструпом (датск. Bjarne Stroustrup) в начале 1980-х гг. Страуструп взял за основу язык C и дополнил его возможностями, имеющимися в языке моделирования Симула (являющимся первым в истории объектно-ориентированным языком).

Первый стандарт C++ появился в 1998 г. В 2003 г. была опубликована исправленная версия стандарта. синтаксис C++ является развитием (в целом, непротиворечивым) синтаксиса C. Программу, написанную на C, обычно получается компилировать компилятором C++ после некоторых несущественных исправлений. Все языковые конструкции C поддерживаются в C++, все стандартные типы данных C доступны в C++, и стандартная библиотека C отлично работает в C++. Помимо классов и объектно-ориентированной стандартной библиотеки, в C++ добавлены такие возможности, как ссылки, константы, улучшенная проверка типов, перегрузка функций и операторов, шаблоны, исключения, пространства имён, булевский тип данных и новый стиль комментариев.

8.3. Принципы ООП

Объектный подход нередко иллюстрируют устройством окружающего нас мира, состоящего из предметов, имеющих свойства. Характерными признаками объектного подхода, отражающими свойства реального мира, являются наследование, полиморфизм и инкапсуляция.

8.3.1. Наследование

Наследованием называется возможность строить новый класс (класс-наследник) на основе существующего (класса-предка) с сохранением

(наследованием) всех атрибутов и методов предка и добавлением новых, если это необходимо.

Наследование (***inheritance***) — это процесс, посредством которого один объект может приобретать свойства другого. Точнее, объект может наследовать основные свойства другого объекта и добавлять к ним черты, характерные только для него.

Наследование является важным, поскольку оно позволяет поддерживать концепцию иерархии классов (hierarchical classification). Применение иерархии классов делает управляемыми большие потоки информации. Например, подумайте об описании жилого дома. Дом — это часть общего класса, называемого строением. С другой стороны, строение — это часть более общего класса — конструкции, который является частью еще более общего класса объектов, который можно назвать созданием рук Человека.

В каждом случае порожденный класс наследует все, связанные с родителем, качества и добавляет к ним свои собственные определяющие характеристики. Без использования иерархии классов, для каждого объекта пришлось бы задать все характеристики, которые бы исчерпывающе его определяли. Однако при использовании наследования можно описать объект путем определения того общего класса (или классов), к которому он относится, с теми специальными чертами, которые делают объект уникальным.

8.3.2. Полиморфизм

Полиморфизм — это в некотором смысле обратная сторона наследования. Классы-потомки могут изменять реализацию методов класса-предка, сохраняя их сигнатуру. Полиморфизм позволяет обрабатывать объекты классов-потомков как однотипные объекты базового класса.

Полиморфизм (***polymorphism***) (от греческого polymorphos) — это свойство, которое позволяет одно и то же имя использовать для решения

двух или более схожих, но технически разных задач. Целью полиморфизма, применительно к объектно-ориентированному программированию, является использование одного имени для задания общих для класса действий. Выполнение каждого конкретного действия будет определяться типом данных. Например, для языка С, в котором полиморфизм поддерживается недостаточно, нахождение абсолютной величины числа требует трех различных функций: abs(), labs() и fabs(). Эти функции подсчитывают и возвращают абсолютную величину целых, длинных целых и чисел с плавающей точкой соответственно.

В C++ каждая из этих функций может быть названа abs(). Тип данных, который используется при вызове функции, определяет, какая конкретная версия функции действительно выполняется. Как вы увидите, в C++ можно использовать одно имя функции для множества различных действий. Это называется перегрузкой функций (function overloading).

В более общем смысле, концепцией полиморфизма является идея "один интерфейс, множество методов". Это означает, что можно создать общий интерфейс для группы близких по смыслу действий. Преимуществом полиморфизма является то, что он помогает снижать сложность программ, разрешая использование того же интерфейса для задания единого класса действий. Выбор же конкретного действия, в зависимости от ситуации, возлагается на компилятор. Вам, как программисту, не нужно делать этот выбор самому. Нужно только помнить и использовать общий интерфейс.

Пример из предыдущего абзаца показывает, как, имея три имени для функции определения абсолютной величины числа вместо одного, обычная задача становится более сложной, чем это действительно необходимо.

Полиморфизм может применяться также и к операторам. Фактически во всех языках программирования ограниченно применяется полиморфизм, например, в арифметических операторах. Так, в С, символ + используется для складывания целых, длинных целых, символьных

переменных и чисел с плавающей точкой. В этом случае компилятор автоматически определяет, какой тип арифметики требуется. В C++ вы можете применить эту концепцию и к другим, заданным вами, типам данных. Такой тип полиморфизма называется перегрузкой операторов (operator overloading),

Ключевым в понимании полиморфизма является то, что он позволяет вам манипулировать объектами различной степени сложности путем создания общего для них стандартного интерфейса для реализации похожих действий.

8.3.3. Инкапсуляция

Инкапсуляция — это принцип «чёрного ящика»: пользователь класса использует только предоставляемые атрибуты и методы класса, но не вникает в его внутреннюю реализацию. Например, несмотря на сложность внутреннего устройства современного автомобиля, управление этим автомобилем сводится к вращению руля и нажатиям на рычаги и педали. Для того чтобы научиться правильно вращать руль и нажимать на педали, знать, как устроен автомобиль, не нужно: инженеры постарались сделать так, чтобы тормоза, подача топлива, электроника управления двигателем и множество других бортовых систем работали, как одно целое, и не напоминали о себе, пока всё в порядке.

Инкапсуляция (***encapsulation***) — это механизм, который объединяет данные и код, манипулирующий с этими данными, а также защищает и то, и другое от внешнего вмешательства или неправильного использования. В объектно-ориентированном программировании код и данные могут быть объединены вместе; в этом случае говорят, что создается так называемый "черный ящик". Когда коды и данные объединяются таким способом, создается объект (object). Другими словами, объект — это то, что поддерживает инкапсуляцию.

Внутри объекта коды и данные могут быть закрытыми (private) для этого объекта или открытыми (public). Закрытые коды или данные

доступны только для других частей этого объекта. Таким образом, закрытые коды и данные недоступны для тех частей программы, которые существуют вне объекта.

Если коды и данные являются открытыми, то, несмотря на то, что они заданы внутри объекта, они доступны и для других частей программы. Характерной является ситуация, когда открытая часть объекта используется для того, чтобы обеспечить контролируемый интерфейс закрытых элементов объекта.

При правильном применении инкапсуляции можно минимизировать количество связей между классами и существенно упростить разработку. На том же примере автомобиля легко видеть, что принципы управления в течение последних пятидесяти лет практически не менялись, несмотря на то, что во внутреннем устройстве уже произошла не одна революция.

9. КЛАССЫ И ОБЪЕКТЫ

Данная глава посвящена рассмотрению классов и объектов. Класс – это механизм для создания объектов. В этом смысле класс лежит в основе многих свойств объектно-ориентированных языков программирования.

9.1. Первое знакомство

Класс объявляется с помощью ключевого слова ***class***. Синтаксис объявления класса похож на синтаксис объявления структуры. Ниже приведено объявление класса с названием ***MyClass***:

```
class MyClass
{
  // закрытые функции и переменные
public:
  // открытые функции и переменные
};
```

Имя класса становится именем нового типа данных, которое используется для объявления объектов класса.

Функции и переменные, объявленные внутри объявления класса, становятся, как говорят, членами (***members***) этого класса. По умолчанию все функции и переменные, объявленные в классе, становятся закрытыми для класса. Это означает, что они доступны только для других членов того же класса. Для объявления открытых членов класса используется ключевое слово ***public***, за которым следует двоеточие. Все функции и переменные, объявленные после слова ***public***, доступны как для других членов класса, так и для любой другой части программы, в которой находится этот класс.

Для примера создадим простой класс, который будет хранить целочисленную переменную. При этом переменную класса имеет смысл сделать закрытой, а пользователю дать доступ к функциям задания переменной и чтения ее.

```
class MyClass
{
  int a;
public:
  void Set(int x);
  int Get(void);
};
```

Обратите внимание, что прототипы функций объявляютсявнутри класса. Функции, которые объявляются внутри класса, называются ***функциями-членами*** (***member functions***).

Поскольку a является закрытой переменной класса, она недоступна для любой функции вне ***MyClass***. Однако поскольку ***Set()*** и ***Get()*** являются членами ***MyClass***, они имеют доступ к ***a***. Более того, ***Set()*** и ***Get()***, являясь открытыми членами ***MyClass***, могут вызываться из любой части программы, использующей ***MyClass***.

Хотя функции ***Set()*** и ***Get()*** и объявлены в ***MyClass***, они еще не определены. Для определения функции-члена вы должны связать имя класса, частью которого является функция-член, с именем функции. Это достигается путем написания имени функции вслед за именем класса с двумя двоеточиями. Два двоеточия называются оператором расширения области видимости (scope resolution operator). Например, далее показан способ определения функций-членов ***Set()*** и ***Get()***:

```
void MyClass::Set(int x)
{
  a = x;
}

int MyClass::Get(void)
{
  return a;
}
```


Объявление класса ***MyClass*** не задает ни одного объекта типа ***MyClass***, оно определяет только тип объекта, который будет создан при его фактическом объявлении. Чтобы создать объект, используйте имя класса, как спецификатор типа данных. Например, в этой строке объявляются два объекта типа ***MyClass***:

```
MyClass MyA, b;//это объекты типа MyClass
```

После того как объект класса создан, можно обращаться к открытым членам класса, используя оператор точка (.), аналогично тому, как осуществляется доступ к членам структуры.

```
MyA.Set(1); // задается значение 1
b.Set(7); // задается значение 7
```

Каждый объект содержит собственную копию всех данных, объявленных в классе, то есть значение ***a*** в ***MyA*** отлично от ***a*** в объекте ***b***.

9.2. Введение в перегрузку функций

Правила языка ***C*** требуют, чтобы все функции в пределах программ мы имели уникальные имена. Данное требование нельзя назвать излишним, но в больших библиотеках функций оно приводит к появлению групп функций, выполняющих схожие действия и имеющих имена, отличающиеся одной или двумя буквами. Хорошим примером может служить стандартная библиотека ***C***: помимо хорошо знакомой читателю функции ***printf***, имеются также функции *fprintf*, ***sprintf*** и ***vprintf***, которые делают в целом всё то же, что и ***printf***, хотя и с некоторыми незначительными отличиями. Стандартная библиотека содержит массу подобных примеров.

Отличие между вариантами функции состоит, как правило, в наборе параметров. Это легко видеть на примере упомянутой функции ***printf***:

функции ***fprintf*** и ***sprintf*** имеют один дополнительный параметр — файловый указатель или строковый буфер, в который они производят вывод; в остальном их поведение не отличается от ***printf***.

Отличительной чертой языка C++ является перегрузка функций (***function overloading***), не только обеспечивающая механизм, посредством которого в C++ достигается один из типов полиморфизма, но и формирует то ядро, вокруг которого развивается вся среда программирования на C++ .

В C++ две или более функции могут иметь одно и то же имя, отличаясь либо типом, либо числом своих аргументов, либо и тем и другим. Если две или более функции имеют одинаковое имя. говорят, что они перегружены.

Перегруженные функции позволяют упростить программы, допуская обращение к одному имени для выполнения близких по смыслу действий.

Перегрузка функции реализуется очень легко: просто объявляются и определяются все требуемые варианты. Компилятор автоматически выберет правильный вариант вызова на основании числа и/или типа используемых в функции аргументов.

Для примера рассмотрим перегрузку функции задания переменных для класса, рассмотренного в предыдущей главе.

```
class MyClass
{
  int a;
public:
  void Set(int x);
  void Set(float x);
  void Set(double x);
  int Get(void);
};
```

В рассмотренном коде, перегруженной функцией является функция Set. При этом смысл задания дробных чисел можно определить как хранение округленного значения:


```
void MyClass::Set(float x)
{
  a = (int)(x+0.5);
}

void MyClass::Set(double x)
{
  a = (int)(x+0.5);
}
```

9.3. Доступ к атрибутам и методам, указатель this

Как показано в предыдущем примере, для доступа извне к атрибутам и методам объекта используется операция «точка». Здесь необходимо сделать дополнение: если имеется указатель на объект, то следует использовать операцию «стрелка»:

```
MyClass MyA;

MyClass* pMy;
pMy = & MyA;

pMy->Set(6);
```

Внутри метода класса имеется возможность узнать, для какого объекта он был вызван. Для этого в любом методе имеется скрытый параметр this. Этот параметр имеет тип указателя на класс, которому принадлежит метод, и содержит адрес объекта, для которого метод вызван. Указатель this можно использовать для доступа к методам и атрибутам объекта в случаях, когда возникает конфликт имён.

В качестве иллюстрации можно рассмотреть метод ***Set()*** из примеры выше, который задает значение переменной ***a*** внутри класса. Было бы гораздо логичнее, если бы этот метод приминал в качестве параметра переменную ***a***, а не ***x***. Однако в этом случае неизбежно возникает конфликт имен внутри реализации метода: есть локальная переменная ***a***,

заданная в качестве параметра метода и есть переменная класса. Ниже продемонстрировано решение этой проблемы с использованием указателя ***this***:

```
void MyClass::Set(int a)
{
  this->a = a;
}
```

Имена локальных объектов имеют приоритет выше, чем имена атрибутов класса. Поэтому имя параметра ***a*** в методе Set() перекроет имя ***a***, принадлежащее атрибуту класса. Однако, используя указатель ***this***, мы можем указать явно, что имеется в виду атрибут.

9.4. Конструкторы и деструкторы

Как и обычные локальные переменные, атрибуты объектов требуют инициализации. Однако в C++ существует возможность автоматизировать процесс инициализацию – использование ***функции-конструктора*** (***constructor function***), включаемая в описание класса.

Конструктор класса вызывается всякий раз при создании объекта этого класса. Таким образом, любая необходимая объекту инициализация при наличии конструктора выполняется автоматически.

Конструктор имеет то же имя, что и класс, частью которого он является, и не имеет возвращаемого значения.

```
class MyClass
{
  int a;
public:

  MyClass();

  void Set(int x);
  void Set(float x);
  void Set(double x);
  int Get(void);
};
```



```
MyClass::MyClass()
{
  a = 0;
}
```

Таким образом, при создании экземпляра класса MyClass, его содержимое всегда будет определено начальным значением 0.

Необходимо отметить, что конструктор всегда описывается в публичной части класса.

Конструктор, как и любая другая функция в языке C++ может быть перегружена. Конструктору можно передавать аргументы, то есть функция-конструктор может быть перегружена. Для этого добавляются необходимые параметры в объявление и определение конструктора. Затем при объявлении объекта задайте параметры в качестве аргументов.

```
class MyClass
{
int a;
public:

MyClass();
MyClass(int a);

void Set(int x);
void Set(float x);
void Set(double x);
int Get(void);
};


MyClass::MyClass()
{
a = 0;
}
```

```
MyClass::MyClass(int a)
{
this->a = a;
}
```

При этом в тексте программы объект можно двумя способами:

```
MyClass MyA;
MyClass MyB(6);
```

Важно помнить, что в ситуации когда программист определил конструктор с параметрами, а конструктор без параметров нет, то создание объекта класса без параметров становится невозможным.

Функцией, обратной конструктору, является ***деструктор*** (***destructor***). Эта функция вызывается при удалении объекта. Обычно при работе с объектом в момент его удаления должны выполняться некоторые действия. Например, при создании объекта для него выделяется память, которую необходимо освободить при его удалении. Имя деструктора совпадает с именем класса, но с символом ~ (тильда) в начале. Пример класса с деструктором:

```
class MyClass
{
int a;
public:

MyClass();
MyClass(int a);

void Set(int x);
void Set(float x);
```



```
void Set(double x);
int Get(void);

~MyClass();
};

MyClass::~MyClass()
{
  cout << "Object deleted" << endl;
}
```

Деструктор класса вызывается при удалении объекта. Локальные объекты удаляются тогда, когда они выходят из области видимости. Глобальные объекты удаляются при завершении программы.

Деструктор не возвращает никаких значений и не может их принимать.

9.5. Конструктор копий

Конструктор копии – это специальный вид конструктора. Он вызывается в ситуациях, когда создается копия объекта. Например функция, принимающая параметр создает копию передаваемой переменной и записывает ее в качестве локальной переменной:

```
void MyFunc(int x)
{
  cout << (x*x);
}

void main()
{
  int a;
  cin >> a;
  MyFunc(a);
}
```

В данном примере, в функции ***MyFunc*** создается локальная переменная ***x***, и в нее копируется содержимое передаваемой переменной ***a***.

Аналогичным образом в качестве параметра может передаваться и объект класса ***MyClass***, рассмотренный выше:

```
void MyPrint(MyClass My)
{
  cout << My.Get();
}
void main()
{
  MyClass MyA;
  MyA.Set(7);
  MyPrint(MyA);
}
```

Если конструктор копии не описан явно, компилятор сгенерирует его; автоматически сгенерированный конструктор копии копирует значения всех атрибутов исходного объекта. И это работает совершенно корректным образом в случае, если все параметра класса являются переменными.

Однако, если в классе присутствуют указатели на область памяти, в которых хранятся данные, принадлежащие конкретному объекту, то использование конструктора копий по умолчанию приведет к тому что несколько объектов будут указывать на одну и ту же область памяти. Таком образом, при уничтожении одного из них, память будет очищена и второй объект будет содержать некорректный указатель, что приведет к некорректной работе.

Ниже приведен код, который будет работать некорректно из-за того что будет использоваться конструктор копий по умолчанию:

```
class MyClass
{
  int* pA;
public:
  MyClass();
  MyClass(int a);
```



```
  int Get(void);

  ~MyClass();
};

MyClass::MyClass()
{
  pA = new int;
  *pA = 0;
}


MyClass::MyClass(int a)
{
  pA = new int;
  *pA = a;
}
MyClass::~MyClass()
{
  if(pA)
    delete pA;
}
int MyClass::Get(void)
{
  return *pA;
}
void MyPrint(MyClass My)
{
  cout << My.Get() << endl;
}


void main()
{
  MyClass MyA(5);
  MyPrint(MyA);
  cout << MyA.Get() << endl;
}
```

Этот конструктор копии принимает единственный параметр, являющийся константной ссылкой на объект такого же класса. В описанном примере необходимый конструктор копии должен выглядеть следующим образом:

```
MyClass(const MyClass& src);
```

А его реализация быть такой:

```
MyClass::MyClass(const MyClass& src)
{
   pA = new int;
   *pA = *(src.pA);
}
```

9.6. Уровни доступа. Друзья

В языке *C++* поддерживаются три уровня доступа:

- *private*;
- *protected*;
- *public*.

Перечисление сделано по убыванию строгости: ***private*** разрешает доступ только методам объектов данного класса, ***protected*** разрешает доступ методам объектов данного класса и классов наследников, ***public*** разрешает свободный доступ кому угодно.

Самое важное применение уровней доступа — контроль корректности состояния объекта. Рассмотрим последний пример из предыдущего раздела. Конструктор и деструктор гарантируют, что атрибут ***pA*** будет корректно инициализирован при рождении объекта и затем освобождён при уничтожении. Но если бы к нему был публичный доступ, то это могло бы легко нарушить целостность объекта:

```
void main()
{  MyClass MyA;
   MyA.pA = (int*)0x123123;
}
```


В момент завершения функции ***main*** будет вызван деструктор объекта ***MyA***, который попытается освободить память по указателю, значение которого мы изменили, пользуясь вседозволенностью публичного доступа. В итоге, во-первых, произойдёт утечка участка памяти, адрес которого содержался в указателе до нашего изменения, а во-вторых, мы с вероятностью, близкой к единице, получим нарушение защиты доступа к памяти при попытке освободить память, которая не была нами выделена.

Если же переменная ***pA*** будет храниться в защищенной области класса:

```
class MyClass
{
   int* pA;
public:
   MyClass();
   MyClass(int a);
   MyClass(const MyClass& src);
   int Get(void);
   ~MyClass();
};
void main()
{
   MyClass MyA;
   MyA.pA = (int*)0x123123;
}
```

При попытке скомпилировать этот код мы получим сообщение об ошибке компиляции:

cannot access private member declared in class ' MyClass '

Компилятор указывает нам, что доступ к этому члену класса возможен только для других членов этого же класса.

Вообще говоря, область безопасного применения публичного доступа к атрибутам достаточно узка. В реальной жизни подавляющее большинство объектов имеют ряд строгих ограничений, начиная от общих

(масса не может быть меньше нуля, скорость — превышать световую) и заканчивая логикой конкретного класса объектов (бензобак вмещает 50 л топлива, в университете пять этажей и так далее).

В некоторых случаях бывает необходимо обеспечить глобальной функции или некоторому классу доступ к защищённым членам другого класса. Для этого в C++ предусмотрен механизм объявления друзей.

Друзья класса — это функции или классы, которым данный класс «доверяет» и которые имеют доступ к его защищённым членам. Друзья объявляются при помощи ключевого слова ***friend*** внутри тела класса:

```
class MyClass
{
  int* pA;
public:
  MyClass();
  MyClass(int a);
  MyClass(const MyClass& src);
  int Get(void);
  ~MyClass();

  friend void MyPrint(MyClass My);
};

void MyPrint(MyClass My)
{
  cout << *(My.pA) << endl;
}
```

Несмотря на то, что всё содержимое класса ***MyClass*** является закрытым, друзья этого класса – функция ***MyPrint()*** имеют к нему доступ. Попытка же записать значение в атрибут объекта в теле функции ***main()*** вызовет ошибку компиляции.

9.7. Перегрузка операторов

Операторы, несмотря на новое название, должны быть уже хорошо знакомы. Это сложение, вычитание, операции декремента и инкремента, сравнение, скобки и т. д. — словом, все те операции, которые явным или неявным образом выполняются, когда мы пишем арифметический знак, квадратные скобки или любой другой специальный символ.

Несколько примеров применения операторов:

```
int a = 5;     // присваивание
int b = a + 8;// присваивание, сложение
int ar[10];    // нет операций
ar[0] = 3;     // выбор элемента массива по индексу,
               присваивание
int * pa = &a;// присваивание, взятие адреса
*pa = -b;      // разыменование, присваивание, унарный
               минус
```

Как видно, любой, даже очень простой код буквально наводнён операторами.

Операторы, перечисленные в следующей таблице, могут быть перегружены.

Оператор	***Значение***	***Тип***
=	*Пприсваивание*	*Бинарный*
+ - * / %	*Сложение, вычитание, умножение, деление, деление по модулю*	*Бинарный*
+= -= *= /= %=	*Присваивание результата сложения, вычитания, умножения, деления и деления по модулю*	*Бинарный*
+ -	*Унарные плюс и минус*	*Унарный*
++ --	*Инкремент и декремент*	*Унарный*
< <= > >= == !=	*Меньше, меньше либо равно, больше, больше либо равно, равно, не равно*	*Бинарный*
<< >>	*Побитовый сдвиг влево и*	*Бинарный*

	вправо	
<<= >>=	*Присваивание результата побитового сдвига влево и вправо*	*Бинарный*
& | ^	*Побитовые И, ИЛИ, исключающее ИЛИ*	*Бинарный*
&= |= ^=	*Присваивание результата побитовых И, ИЛИ, исключающего ИЛИ*	*Бинарный*
&& ||	*Логическое И, ИЛИ*	*Бинарный*
!	*Логическое НЕ*	*Унарный*
~	*Побитовое отрицание*	*Унарный*
&	*Взятие адреса*	*Унарный*
->	*Выбор члена (метода или атрибута)*	*Бинарный*
->*	*Разыменование указателя на член класса*	*Бинарный*
*	*Разыменование указателя*	*Унарный*
[]	*Выбор элемента массива по индексу*	—
()	*Вызов функции*	—
,	*Запятая*	*Бинарный*
new	*Выделение памяти*	—
delete	*Освобождение памяти*	—
преобразование типа		*Унарный*

За несколькими исключениями, операторы делятся на унарные и бинарные, т. е. требующие один или два параметра, и возвращают результат определённого типа. Очевидно, они очень похожи на обычные функции. Поэтому определение собственного оператора тоже мало отличается от определения функции:

```
class MyNumber
{public:
  int val;
};
```



```
MyNumber operator + (MyNumber left, int
right)
{
  MyNumber Res;
  Res.val = left.val + right;
  return Res;
}

void main()
{
  MyNumber My1, My2;
  My1.val = 3;
  My2 = My1 + 5;
  cout << My2.val;
}
```

Как видно, оператор, объявленный глобально, отличается от обычной функции только именем: для операторов имя состоит из ключевого слова ***operator*** и следующего за ним обозначения операции. Первый параметр такой функции соответствует левому параметру операции, второй — правому, (значение имеет только порядок, но не имена). Для унарных операторов единственный параметр соответствует объекту, на который действует оператор.

Приведённый пример перестанет работать, если сделать атрибут ***val*** защищённым: глобальные функции не имеют доступа к защищённым членам объектов. В этом случае можно поступить двумя способами:

1. указать, что глобальная функция-оператор является другом класса.
2. перенести определение оператора сложения внутрь класса.

Первый вариант ничем не отличается от объявления друзей, рассмотренного выше. Второй вариант имеет свои особенности. Дело в том, что для операторов, являющихся методами класса, предполагается, что они вызываются как методы того объекта, который стоит слева

в бинарной операции, либо просто как методы без параметров в случае унарной операции над объектом.

Приведённый выше пример может быть переписан следующим образом:

```
class MyNumber
{
  int val;
public:
  void Set(int val);
  int Get(void);
  MyNumber operator + (int right);

};

MyNumber MyNumber::operator + (int right)
{
  MyNumber Res;
  Res.val = val + right;
  return Res;
}
void MyNumber::Set(int val)
{
  this->val = val;
}
int MyNumber::Get(void)
{
  return val;
}

void main()
{
  MyNumber My1, My2;
  My1.Set(3);
  My2 = My1 + 5;
  cout << My2.Get();
}
```


При этом помните, что если параметрами класса являются указатели, то вы обязаны определить конструктор копий!

9.8. Перегрузка потоков ввода/вывода

Зачастую, для удобства пользования объектами классов, необходимо обеспечить возможность их ввода/вывода без использования специальных функций. Например, вместо

```
MyNumber My1;
int q;
cin >> q;
My1.Set(q);
```

было бы здорово, если можно было бы написать так:

```
MyNumber My1;
cin >> My1;
```

Инструментарий языка C++ позволяет перегрузить операторы потока ввода вывода, что делает ввод/вывод объектов совершенно очевидным.

```
class MyNumber
{
  int val;
public:
  MyNumber operator + (int right);

  friend istream& operator>>(istream&, MyNumber&);
  friend ostream& operator<<(ostream&, MyNumber&);
};

MyNumber MyNumber::operator + (int right)
{
  MyNumber Res;
  Res.val = val + right;
  return Res;
}

istream& operator>>(istream& st, MyNumber& obj)
```

```
{
   st >> obj.val ;
   return st;
}
ostream& operator<<(ostream& st, MyNumber& obj)
{
   st << obj.val << endl;
   return st;
}

void main()
{
   MyNumber My1;
   cin >> My1;

   MyNumber My2;
   My2 = My1 + 5;
   cout << My2;
}
```

Потоки ввода вывода не могут являться операторами класса, поэтому их необходимо объявить друзьями класса, если необходимо обеспечить им доступ к закрытым членам класса.

10. НАСЛЕДОВАНИЕ

Применительно к C++ наследование – это механизм, посредством которого один класс может наследовать свойства другого. Наследование позволяет строить иерархию классов, переходя от более общих к более специализированными.

Для начала необходимо определить два термина, обычно используемые при обсуждении наследования. Когда один класс наследуется другим, класс, который наследуется, называют ***базовым классом*** (***base class***). Наследующий класс называют ***производным классом*** (***derived class***).

Обычно процесс наследования начинается с задания базового класса. Базовый класс определяет все те качества, которые будут общими для всех производных от него классов. В сущности, базовый класс представляет собой наиболее общее описание ряда характерных черт. Производный класс наследует эти общие черты и добавляет свойства, характерные только для него.

Рассмотрим пример наследования для иллюстрации его смысла. Для начала – объявление базового класса:

```
class MyClass
{
   int a;
public:
   void Set(int a);
};
```

Теперь объявим производный класс, наследующий этот базовый:

```
class NewClass : public MyClass
{
   int b;
public:
   void Set(int b);
   int Sum(void);
};
```

Обратите внимание, что после имени класса ***NewClass*** имеется двоеточие, за которым следует ключевое слово ***public*** и имя класса ***MyClass***. Для компилятора это указание на то, что класс ***NewClass*** будет наследовать все компоненты класса ***MyClass***. Само ключевое слово ***public*** информирует компилятор о том, что, поскольку класс ***MyClass*** будет наследоваться, значит, все открытые элементы базового класса будут также открытыми элементами производного класса.

Важно помнить, что все закрытые элементы базового класса останутся закрытыми и к ним не будет прямого доступа из производного класса. В этом месте полезно вспомнить о разнице между полями доступа:

- ***private*** – разрешает доступ только методам объектов данного класса;
- ***protected*** – разрешает доступ методам объектов данного класса и классов наследников.

Поэтому в приведенном примере, поскольку по умолчанию параметр ***a*** находится в области ***private***, доступ к нему будет закрыт для методов класса ***NewClass***.

По смыслу, в наследуемый класс добавляет хранение еще одной переменной и функцию суммирования хранимых параметров. Ниже приведен код реализации функций и собственно программа, демонстрирующая возможность использования этих классов.

```
class MyClass
{
protected:
  int a;
public:
  void Set_a(int a);
};

class NewClass : public MyClass
{
protected:
  int b;
```



```
public:
  void Set_b(int b);
  int Sum(void);
};

void MyClass::Set_a(int a)
{
  this->a = a;
}

void NewClass::Set_b(int b)
{
  this->b = b;
}

int NewClass::Sum(void)
{
  return (a+b);
}

void main()
{
  NewClass My;
  My.Set_a(5);
  My.Set_b(6);
  cout << My.Sum();
}
```

А теперь предположим, что нам необходимо создать класс, который будет хранить 2 параметра, а также поизводить функцию суммирования и умножения. Если бы не наследование, нам бы прошлось просто «с нуля» написать такой класс, а наследование позвоняет пронаследовать новый класс от реализованного ***NewClass***, реализовав в нем лишь функцию умножения:

```
class NewMult : public NewClass
{
public:
  int Mult(void);
};
```

```
int NewMult::Mult(void)
{
  return (a*b);
}

void main()
{
  NewMult My;
  My.Set_a(5);
  My.Set_b(6);
  cout << My.Mult();
}
```

10.1. Конструкторы и деструкторы при наследовании

Если у базового и у производного классов имеются конструкторы и деструкторы, то конструкторы выполняются в порядке наследования, а деструкторы – в обратном порядке.

Таким образом, конструктор базового класса выполняется раньше конструктора производного класса. Для деструкторов правилен обратный порядок: деструктор производного класса выполняется раньше деструктора базового класса.

Последовательность выполнения конструкторов и деструкторов достаточно очевидна. Поскольку базовый класс "не знает" о существовании производного, любая инициализация выполняется в нем независимо от производного класса и возможно становится основой для любой инициализации, выполняемой в производном классе. Поэтому инициализация в базовом классе должна выполняться первой.

С другой стороны, деструктор производного класса должен выполняться раньше деструктора базового класса потому, что базовый класс лежит в основе производного. Если бы деструктор базового класса выполнялся первым, это бы разрушило производный класс. Таким образом, деструктор производного класса должен вызываться до того, как объект прекратит свое существование.

В этой очень короткой программе показано, в каком порядке выполняются конструкторы и деструкторы базового и производного классов:

```
class CBase
{
public:
  CBase()
  {cout << "CBase" << endl;}
  ~CBase ()
  {cout << "~CBase" << endl;}
};

class CDerived : public CBase
{
public:
  CDerived()
  {cout << "CDerived" << endl;}
  ~CDerived ()
  {cout << "~CDerived" << endl;}
};

void main ()
{
  CDerived Obj;
}
```

После выполнения программы на экран выводится следующее:

```
CBase
CDerived
~CDerived
~ CBase
```

Как видите, конструкторы выполняются в порядке наследования, а деструкторы – в обратном порядке.

10.1.1. Конструкторы с параметрами при наследовании

В случае, когда класс-наследник имеет конструктор с параметром, а у родительского класса используется конструктор без параметров, все происходит интуитивно просто:

```
class CBase
{
public:
   CBase()
   {cout << "CBase" << endl;}
   ~CBase ()
   {cout << "~CBase" << endl;}
};

class CDerived : public CBase
{
   int a;
public:
   CDerived(int a)
   {cout << "CDerived" << endl;
   this->a = a;}
   ~CDerived ()
   {cout << "~CDerived" << endl;}
};

void main ()
{
   CDerived Obj(5);
}
```

Иная ситуация возникает, если в родительском классе определен также конструктор с параметром. Приведенная ниже программа демонстрирует способ вызова конструктора с параметром родительского класса из класса-наследника.

```
class MyClass
{
protected:
   int a;
```



```
public:
   MyClass(int a);
};

MyClass::MyClass(int a)
{
   this->a = a;
}

class NewClass : public MyClass
{
protected:
   int b;
public:
   NewClass(int a, int b);
   int Sum(void);
};

NewClass::NewClass(int a, int b) : MyClass(a)
{
   this->b = b;
}

int NewClass::Sum(void)
{
   return (a+b);
}

void main()
{
   NewClass My(1,2);
   cout << My.Sum();
}
```

Именно конструкция ***:MyClass(a)***, при описании конструктора класса-наследника обеспечивает вызов конструктор родительского класса с параметром.

10.2. Преобразования типов в иерархии классов

Если рассматривать объект класса, не имеющего предков, с точки зрения расположения атрибутов в памяти, то окажется, что объект занимает непрерывный блок памяти, в котором друг за другом расположены атрибуты в порядке их объявления в теле класса.

Перейдём к ситуации, когда класс рассматриваемого объекта имеет одного предка. В этом случае окажется, что блок памяти, занимаемый объектом, состоит из двух частей, первая из которых содержит экземпляр базового класса.

Рассмотрим следующий пример:

```
class A
{ int a;
public:
   A(int a);
   void AMethod();
};
A::A(int a)
{ this->a = a;
}

void A::AMethod()
{ cout << a << endl;
}

class B : public A
{ int b;
public:
   B(int a, int b);
   void BMethod();
};

B::B(int a, int b) : A(a)
{ this->b = b;
}

void B::BMethod()
{ cout << b << endl;
}
```



```
void Print_aVal(A * obj)
{
   obj->AMethod();
}

void main()
{
   A aObj(5);
   B bObj(10, 20);
   Print_aVal(&aObj);
   Print_aVal(&bObj);
}
```

Самое главное содержится в функции ***main()***. Несмотря на то, что функция ***Print_aVal*** требует параметр типа ***A***, в неё безо всяких проблем передаётся объект класса ***B***, являющегося потомком ***A***.

Такое возможно благодаря разрешённому преобразованию типов «снизу вверх»: от потомка к предку. Иными словами, объект может быть автоматически преобразован к объекту любого типа, являющегося базовым для фактического типа, и это преобразование будет безопасно и не потребует никаких специальных действий со стороны программиста.

Возможность привести объект к базовому типу является одной из двух составляющих, обеспечивающих полиморфизм объектов.

Такое преобразование зачастую используется в ситуациях, когда в один список необходимо включить объекты разных классов, но обладающие одинаковыми свойствами. Другими словами есть один родительский класс и несколько классов наследников, объекты которых необходимо включить в единый список: работники университета (преподаватели, студенты, деканат – все это люди), учет склада вещей (ботинки, туфли, рубашки, носки – все это вещи) и так далее.

10.3. Виртуальные функции

Виртуальная функция (***virtual function***) является членом класса. Она объявляется внутри базового класса и переопределяется в производном классе. Для того, чтобы функция стала виртуальной, перед объявлением функции ставится ключевое слово ***virtual***.

```
class A
{
  int a;
public:
  A(int a);
  virtual void Method();
};
```

Если класс, содержащий виртуальную функцию, наследуется, то в производном классе виртуальная функция переопределяется.

По существу, виртуальная функция реализует идею "один интерфейс, множество методов", которая лежит в основе полиморфизма. Виртуальная функция внутри базового класса определяет вид интерфейса этой функции. Каждое переопределение виртуальной функции в производном классе определяет ее реализацию, связанную со спецификой производного класса. Таким образом, переопределение создает конкретный метод.

При переопределении виртуальной функции в производном классе, ключевое слово ***virtual*** не требуется.

```
class B : public A
{
  int b;
public:
  B(int a, int b);
  void Method();
};
```


Виртуальная функция может вызываться так же, как и любая другая функция-член. Однако наиболее интересен вызов виртуальной функции через указатель, благодаря чему поддерживается динамический полиморфизм.

Из предыдущего раздела вы знаете, что указатель базового класса можно использовать в качестве указателя на объект производного класса. Если указатель базового класса ссылается на объект производного класса, который содержит виртуальную функцию и для которого виртуальная функция вызывается через этот указатель, то компилятор определяет, какую версию виртуальной функции вызвать, основываясь при этом на типе объекта, на который ссылается указатель. При этом определение конкретной версии виртуальной функции имеет место не в процессе компиляции, а в процессе выполнения программы. Другими словами, тип объекта, на который ссылается указатель, и определяет ту версию виртуальной функции, которая будет выполняться. Поэтому, если два или более различных класса являются производными от базового, содержащего виртуальную функцию, то, если указатель базового класса ссылается на разные объекты этих производных классов, выполняются различные версии виртуальной функции. Этот процесс является реализацией принципа динамического полиморфизма.

Фактически, о классе, содержащем виртуальную функцию, говорят как о ***полиморфном классе*** (***polymorphic class***).

Рассмотрим еще раз пример из предыдущего раздела при немного измененном коде:

```
class A
{
protected:
  int a;
public:
  A(int a);
  virtual void Method();
};
```

```
A::A(int a)
{
  this->a = a;
}

void A::Method()
{
  cout << a << endl;
}

class B : public A
{
  int b;
public:
  B(int a, int b);
  void Method();
};

B::B(int a, int b) : A(a)
{
  this->b = b;
}

void B::Method()
{
  cout << a << "\t" << b << endl;
}

void Print_aVal(A * obj)
{
  obj->Method();
}

void main()
{
  A aObj(5);
  B bObj(10, 20);
  Print_aVal(&aObj);
  Print_aVal(&bObj);
}
```


В данной ситуации функция ***Method*** класса ***A*** является виртуальной. Таким образом, функция ***Method*** класса ***B*** ее переопределяет. Поэтому при вызове функции ***Method*** у объекта класса ***B***, несмотря на то, что вызов реализован через указатель на родительский тип, будет вызвана именно реализации класса ***B***. В результате на экран будет выведено следующее:

```
5
10   20
```

Переопределение виртуальной функции внутри производного класса может показаться похожим на перегрузку функций. Однако эти два процесса совершенно различны. Во-первых, перегружаемая функция должна отличаться типом и/или числом параметров, а переопределяемая виртуальная функция должна иметь точно такой же тип параметров, то же их число, и такой же тип возвращаемого значения.

Если при переопределении виртуальной функции вы изменяете число или тип параметров, она просто становится перегружаемой функцией и ее виртуальная природа теряется.

Далее, виртуальная функция должна быть членом класса. Это не относится к перегружаемым функциям. Кроме этого, если деструкторы могут быть виртуальными, то конструкторы нет. Чтобы подчеркнуть разницу между перегружаемыми функциями и переопределяемыми виртуальными функциями, для описания переопределения виртуальной функции используется термин ***подмена*** (***overriding***).

10.3.1. Чисто виртуальные функции и абстрактные классы

На практике встречаются ситуации, когда виртуальная функция объявляется в базовом классе, не выполняет никаких значимых действий. Это вполне обычная ситуация, поскольку часто базовый класс используется для возможности объединения наследник по единому типу и заданию общего для всех поведения. То есть в таких базовых классах

просто содержится базовый набор функций-членов и переменных, для которых в производном классе определяется все недостающее. Когда в виртуальной функции базового класса отсутствует значимое действие, в любом классе, производном от этого базового, такая функция обязательно должна быть переопределена.

Для реализации этого положения в C++ поддерживаются так называемые ***чисто виртуальные функции*** (***pure virtual function***).

Чисто виртуальные функции не определяются в базовом классе. Туда включаются только прототипы этих функций. Для чисто виртуальной функции используется такая основная форма:

```
virtual void Method() = 0;
```

Ключевой частью этого объявления является приравнивание функции нулю. Это сообщает компилятору, что в базовом классе не существует тела функции. Если функция задается как чисто виртуальная, это предполагает, что она обязательно должна подменяться в каждом производном классе. Если этого нет, то при компиляции возникнет ошибка.

Таким образом, создание чисто виртуальных функций — это путь, гарантирующий, что производные классы обеспечат их переопределение.

Если класс содержит хотя бы одну чисто виртуальную функцию, то о нем говорят как об ***абстрактном классе*** (***abstract class***). Поскольку в абстрактном классе содержится, по крайней мере, одна функция, у которой отсутствует тело функции, технически такой класс неполон, и ни одного объекта этого класса создать нельзя.

Абстрактные классы могут быть только наследуемыми. Они никогда не бывают изолированными. Важно понимать, однако, что по-прежнему можно создавать указатели абстрактного класса, благодаря которым

достигается динамический полиморфизм. (Также допускаются и ссылки на абстрактный класс.)

Абстрактные классы в C++ играют роль ***интерфейсов***: они описывают набор предоставляемых объектом методов, но не предоставляют реализацию этих методов. Иными словами, они описывают абстракцию какого-либо поведения на уровне набора сигнатур методов.

На использовании интерфейсов основано компонентное программирование.

11. ФАЙЛОВЫЕ ПОТОКИ В ЯЗЫКЕ C++

В этой главе мы более подробно познакомимся с особенностями работы с файловыми потоками, уже опираясь на базовые знания объектно-ориентированного подхода.

11.1. Вывод в файловых поток

cout представляет собой объект типа **ostream** (выходной поток). Используя класс **ostream**, ваши программы могут выполнять вывод в **cout** с использованием оператора вставки или различных методов класса, например **cout.put()**. Заголовочный файл ***iostream*** определяет выходной поток **cout**. Аналогично, заголовочный файл ***fstream*** определяет класс выходного файлового потока с именем **ofstream**. Используя объекты класса **ofstream**, ваши программы могут выполнять вывод в файл. Для начала вы должны объявить объект типа **ofstream**, указав имя требуемого выходного файла как символьную строку – параметр конструктора класса:

```
ofstream file_object("FILENAME.EXT");
```

Если вы указываете имя файла при объявлении объекта типа **ofstream**, C++ создаст новый файл на вашем диске, используя указанное имя, или перезапишет файл с таким же именем, если он уже существует на вашем диске Следующая программа создает объект типа **ofstream** и затем использует оператор вставки для вывода нескольких строк текста в файл:

```
#include <fstream>
using namespace std;

void main(void)
{
   ofstream book_file("BOOKINFO.DAT");
   book_file << "Hello!" << " It is test " << endl;
```



```
   book_file << "file sterams" << endl;
}
```

Как видите, в C++ достаточно просто выполнить операцию вывода в файл.

11.2. Чтение из входного файлового потока

Только что вы узнали, что, используя класс **ofstream**, ваши программы могут быстро выполнить операции вывода в файл. Подобным образом ваши программы могут выполнить операции ввода из файла, используя объекты типа **ifstream**. Опять же, вы просто создаете объект, передавая ему в качестве параметра требуемое имя файла:

```
ifstream input_file("filename.EXT");
```

Следующая программа открывает файл, который мы создали с помощью предыдущей программы, и читает, а затем отображает первые три элемента файла:

```
#include <iostream>
#include <fstream>
using namespace std;

void main(void)
{
   ifstream input_file("BOOKINFO.DAT") ;
   char one[64], two[64];

   input_file >> one;
   input_file >> two;

   cout << one << endl;
   cout << two << endl;

}
```

Если вы откомпилируете и запустите эту программу, то, вероятно, предположите, что она отобразит первые три строки файла. Однако, подобно **cin**, входные файловые потоки используют пустые символы, чтобы определить, где заканчивается одно значение и начинается другое.

11.2.1. Чтение целой строки файлового ввода

Иногда бывает удобно вычитывать данные из файла по строкам. В этом случае в массиве символов будут сохранены все пробелы, поскольку разделителем будет считаться только перевод строки.

```
#include <iostream>
#include <fstream>
using namespace std;

void main(void)
{
   ifstream input_file("BOOKINFO.DAT");
   char one[64], two[64];
   input_file.getline(one, sizeof(one)) ;
   input_file.getline(two, sizeof(two));
   cout << one << endl;
   cout << two << endl;
}
```

В данном случае программа успешно читает содержимое файла, потому что она знает, что файл содержит три строки. Однако во многих случаях ваша программа не будет знать, сколько строк содержится в файле. В таких случаях ваши программы будут просто продолжать чтение содержимого файла пока не встретят конец файла.

11.2.2. Определение конца файла

Обычной файловой операцией в ваших программах является чтение содержимого файла, пока не встретится конец файла. Чтобы определить конец файла, ваши программы могут использовать функцию **eof()** потокового объекта. Эта функция возвращает значение 0, если конец файла еще не встретился, и 1, если встретился конец файла. Используя

цикл **while**, ваши программы могут непрерывно читать содержимое файла, пока не найдут конец файла, как показано ниже:

```
while (! input_file.eof())
{
   // Операторы
}
```

В данном случае программа будет продолжать выполнять цикл, пока функция **eof()** возвращает ложь (0).

Следующая программа использует функцию **eof()** для чтения содержимого файла, пока не достигнет конца файла.

```
#include <iostream>
#include <fstream>
using namespace std;

void main(void)
{
   ifstream input_file("BOOKINFO.DAT");
   char line[64];
   while (! input_file.eof())
   {  input_file.getline(line, sizeof(line));
      cout << line << endl;
   }
}
```

Аналогично, следующая программа читает содержимое файла по одному слову за один раз, пока не встретится конец файла.

```
#include <iostream>
#include <fstream>
using namespace std;

void main(void)
{
   ifstream input_file("BOOKINFO.DAT");
   char word[64] ;
```

```
   while (! input_file.eof())
   {
      input_file >> word;
      cout << word << endl;
   }

}
```

И наконец, следующая программа читает содержимое файла по одному символу за один раз, используя функцию **get()**, пока не встретит конец файла:

```
#include <iostream>
#include <fstream>
using namespace std;

void main(void)
{
   ifstream input_file("BOOKINFO.DAT");
   char letter;
   while (! input_file.eof())
   {  letter = input_file.get();
      cout << letter;
   }
}
```

11.3. Проверка ошибок при выполнении файловых операций

Программы, представленные до настоящего момента, предполагали, что во время файловых операций ввода-вывода не происходят ошибки. К сожалению, это сбывается не всегда. Например, если вы открываете файл для ввода, ваши программы должны проверить, что файл существует. Аналогично, если ваша программа пишет данные в файл, вам необходимо убедиться, что операция прошла успешно (к примеру, отсутствие места на диске, скорее всего, помешает записи данных). Чтобы помочь вашим программам следить за ошибками, вы можете использовать функцию **fail()**

файлового объекта. Если в процессе файловой операции ошибок не было, функция возвратит ложь (0). Однако, если встретилась ошибка, функция **fail()** возвратит истину. Например, если программа открывает файл, ей следует использовать функцию **fail()**, чтобы определить, произошла ли ошибка, как это показано ниже:

```
ifstream input_file("FILENAME.DAT");
if (input_file.fail())
{  cout << "Error opening FILENAME.EXT" << endl;
   exit(1);
}
```

Таким образом, программы должны убедиться, что операции чтения и записи прошли успешно. Следующая программа использует функцию **fail()** для проверки различных ошибочных ситуаций:

```
#include <iostream>
#include <fstream>
using namespace std;

void main(void)
{
  char line[256] ;
  ifstream input_file("BOOKINFO.DAT") ;

  if (input_file.fail())
    cout << "Error opening BOOKINFO.DAT" << endl;
  else
  {
    while ((! input_file.eof()) &&
           (! input_file.fail()))
    {
      input_file.getline(line, sizeof(line)) ;
      if (! input_file.fail())
        cout << line << endl;
    }
  }
```

11.4. } Закрытие файлов

При завершении вашей программы операционная система закроет открытые ею файлы. Однако, как правило, если вашей программе файл больше не нужен, она должна его закрыть. Для закрытия файла ваша программа должна использовать функцию **close()**, как показано ниже:

```
input_file.close ();
```

Когда вы закрываете файл, все данные, которые ваша программа писала в этот файл, сбрасываются на диск, и обновляется запись каталога для этого файла.

ВНИМАНИЕ! Если во время работы программы пока файл еще не закрыт, открыть его внешним редактором, то операционная система не гарантирует синхронность отображаемых данных с работой программы, поскольку общение с файлом происходит через промежуточный буфер.

11.5. Управление открытием файлов

В примерах программ, представленных в данном уроке, файловые операции ввода и вывода выполнялись с начала файла. Однако, когда вы записываете данные в выходной файл, вероятно, вы захотите, чтобы программа добавляла информацию в конец существующего файла. Для открытия файла в режиме добавления вы должны при его открытии указать второй параметр, как показано ниже:

```
ifstream output_file("FILENAME.EXT", ios::app);
```

В данном случае параметр ***ios::app*** указывает режим открытия файла. По мере усложнения ваших программ они будут использовать сочетание значений для режима открытия файла, которые перечислены в табице:

Режим открытия	Назначение
ios::app	Открывает файл в режиме добавления, располагая файловый указатель в конце файла.
ios::ate	Располагает файловый указатель в конце файла.
ios::in	Указывает открыть файл для ввода .
ios::nocreate	Если указанный файл не существует, не создавать файл и возвратить ошибку.
ios::noreplace	Если файл существует, операция открытия должна быть прервана и должна возвратить ошибку.
ios::out	Указывает открыть файл для вывода.
ios::trunc	Сбрасывает (перезаписывает) содержим, з существующего файла.

Следующая операция открытия файла открывает файл для вывода, используя режим ***ios::noreplace***, чтобы предотвратить перезапись существующего файла:

```
ifstream output_file("FIlename.EXT", ios::out | ios::noreplace);
```

11.6. Выполнение операций чтения и записи

Все программы, представленные в данной главе, выполняли файловые операции над символьными строками. По мере усложнения ваших программ, возможно, вам понадобится читать и писать массивы и структуры. Для этого ваши программы могут использовать функции **read()** и **write()**.

При использовании функций **read()** и **write()** вы должны указать буфер данных, в который данные будут читаться или из которого они будут записываться, а также длину буфера в байтах, как показано ниже:

```
input_file.read(buffer, sizeof(buffer)) ;
output_file.write(buffer, sizeof(buffer));
```

Например, следующая программа использует функцию **write()** для вывода содержимого структуры в файл.

```
#include <iostream>
#include <fstream>
using namespace std;

struct employee
{
   char name[64];
   int age;
   float salary;
};

void main(void)
{
   employee worker = { "Джон Дой", 33, 25000.0 };

   ofstream emp_file("EMPLOYEE.DAT") ;
   emp_file.write((char *) &worker,
                  sizeof(employee));

   emp_file.close();
}
```

Функция **write()** обычно получает указатель на символьную строку. Символы (char *) представляют собой оператор приведения типов, который информирует компилятор, что вы передаете указатель на другой тип.

Подобным образом следующая программа использует метод **read()** для чтения из файла информации о служащем:

```
#include <iostream>
#include <fstream>
using namespace std;

struct employee
{
   char name[64];
   int age;
   float salary;
};

void main(void)
{
   employee worker;

   ifstream emp_file("EMPLOYEE.DAT");
   emp_file.read((char *) &worker,
   sizeof(employee));

   cout << worker.name << endl;
   cout << worker.age << endl;
   cout << worker.salary << endl;

   emp_file.close();
}
```

11.7. ЧТО НЕОБХОДИМО ПОМНИТЬ

- **Заголовочный файл fstream определяет классы ifstream и ofstream**, с помощью которых ваша программа может выполнять операции файлового ввода и вывода.
- Для открытия файла на ввод или вывод вы должны объявить объект типа **ifstream** или **ofstream**, передавая конструктору этого объекта имя требуемого файла.

- После того как ваша программа открыла файл для ввода или вывода, она может читать или писать данные, используя операторы извлечения (>>) и вставки (<<).
- Ваши программы могут выполнять ввод или вывод символов в файл или из файла, используя функции **get()** и **put()**.
- Ваши программы могут читать из файла целую строку, используя функцию **getline()**.
- Большинство программ читают содержимое файла, пока не встретится конец файла. Ваши программы могут определить конец файла с помощью функции **eof()**.
- Когда ваши программы выполняют файловые операции, они должны проверять состояние всех операций, чтобы убедиться, что операции выполнены успешно. Для проверки ошибок ваши программы могут использовать функцию **fail()**.
- Если вашим программам необходимо вводить или выводить такие данные, как структуры или массивы, они могут использовать методы **read()** и **write()**.
- Если ваша программа завершила работу с файлом, его следует закрыть с помощью функции **close()**.

12. ЗНАКОМСТВО С БИБЛИОТЕКОЙ MFC

Библиотека MFC – достаточно мощный и гибкий инструмент для создания Windows-приложений на базе языка C++. В данной главе происходит знакомство с этой библиотекой и рассматриваются принцип создания простейшего пользовательского приложения с различными органами управления.

Первая версия MFC была выпущена вместе с седьмой версией 16-разрядного компилятора языка C/C++ компании Microsoft в 1992 году. Для тех, кто занимался разработкой приложений с использованием API-функций, пакет MFC обещал весьма значительное повышение производительности процесса программирования.

Одной из примечательных особенностей MFC является префикс «***Afx***», используемый в именах многих функций, макросов и названии стандартного заголовочного файла «stdafx.h». На ранней стадии разработки, то, что впоследствии стало называться MFC, имело название «**A**pplication **F**ramework e**X**tensions» и аббревиатуру «***Afx***». Решение изменить название на Microsoft Foundation Classes (MFC) было принято слишком поздно, чтобы менять упоминания Afx в коде.

В настоящее время упор компании Microsoft на MFC был ослаблен в пользу Microsoft .NET Framework. Однако, несмотря на это, MFC по-прежнему остается популярной среди разработчиков.

12.1. Введение

Библиотека ***MFC*** (***Microsoft Foundation Classes***) – библиотека на языке C++, разработанная Microsoft и призванная облегчить разработку ***GUI*** (графический пользовательский интерфейс – **G**raphical **U**ser **I**nterface) приложений для Microsoft Windows.

Библиотека MFC, облегчает работу с GUI путем ***создания каркаса приложения*** — «скелетной» программы, автоматически создаваемой по заданному макету интерфейса и полностью берущей на себя рутинные

действия по его обслуживанию (отработка оконных событий, пересылка данных между внутренними буферами элементов и переменными программы и т. п.). Программисту после генерации каркаса приложения необходимо только вписать код в места, где требуются специальные действия. Каркас должен иметь вполне определенную структуру, поэтому для его генерации и изменения в Visual C++ предусмотрены мастера.

Кроме того, MFC предоставляет объектно-ориентированный слой оберток (англ. ***wrappers***) над множеством функций ***Windows API*** (**A**pplication **P**rogramming **I**nterfaces), делающий несколько более удобной работу с ними. Этот слой представляет множество встроенных в систему объектов (окна, виджеты, файлы и т. п.) в виде классов и опять же берет на себя рутинные действия вроде закрытия дескрипторов и выделения/освобождения памяти.

Добавление кода приложения к каркасу реализовано двумя способами. Первый использует механизм наследования: основные программные структуры каркаса представлены в виде классов, наследуемых от библиотечных. В этих классах предусмотрено множество виртуальных функций, вызываемых в определенные моменты работы программы. Путем доопределения (в большинстве случаев необходимо вызвать функцию базового класса) этих функций программист может добавлять выполнение в эти моменты своего кода.

Второй способ используется для добавления обработчиков оконных событий. Мастер создает внутри каркасов классов, связанных с окнами, специальные массивы – ***карты (оконных) сообщений*** (англ. ***message map***), содержащие пары «ИД сообщения – указатель на обработчик». При добавлении/удалении обработчика мастер вносит изменения в соответствующую карту сообщений.

12.2. Работа с MFC

К работе с MFC следует подходить именно как к инструменту, который хоть и принимает на себя значительную часть работы, требует от программиста знания принципов и особенностей объектно-ориентированного программирования.

12.2.1. Получение информации о приложении

Когда пишется приложение с использованием классов MFC, то программист создает единственный объект, являющийся наследником от класса ***CWinApp***. Для получения из любого места программы информации об этом объекте и связанных с ним параметрах, программиста может использовать следующие функции:

- CWinApp* AfxGetApp() – возвращает указатель на объект CWinApp
- LPCNSRT AfxGetAppName() – возвращает имя приложения

12.2.2. Иерархия классов MFC

Все классы в ***MFC*** можно условно разделить на две группы: классы производные от ***CObject***, и класса, не зависящие от него (рис.7).

На вершине иерархии находится ***CObject***. Как и положено, каждый новый производный класс обладает как свойствами, унаследованными от родительских классов, так и приобретает новые, характерные именно для него.

Единственной переменной класса ***CObject*** является статическая переменная ***classObject*** типа ***CRuntimeClass***, которая хранит информацию об ассоциированном с классом ***CObject*** объекте во время выполнения программы.

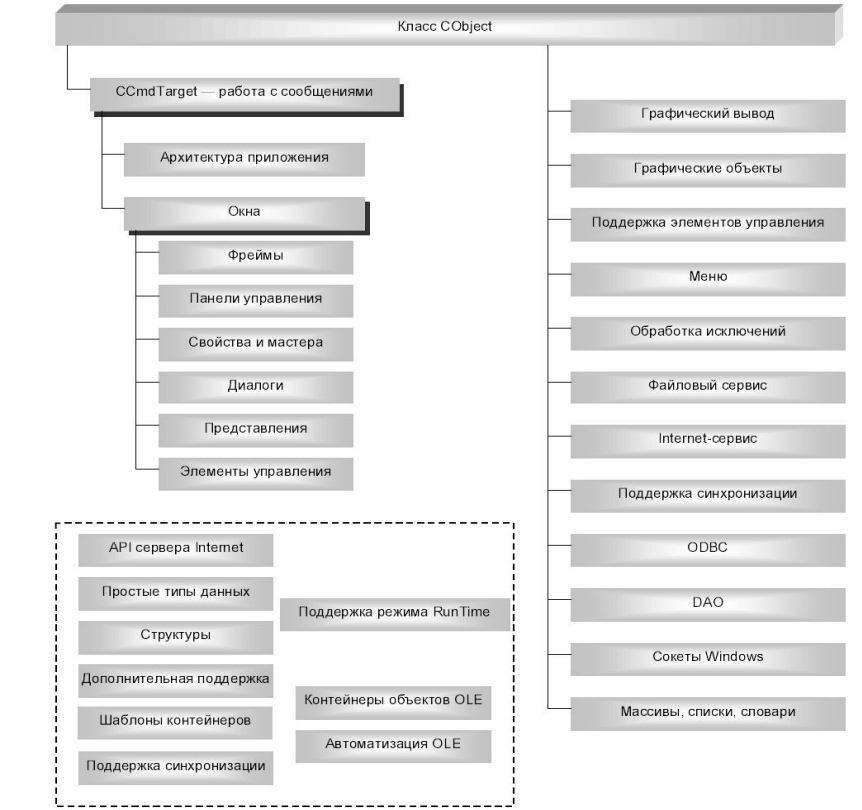


Рис. 7. Иерархия классов MFC

12.3. Создание приложений на базе MFC

В рамках данного пособия мы знакомимся с использованием библиотеки MFC для создания простейших оконных приложений для взаимодействия с пользователем.

Для создания проекта с использование MFC, необходимо создать новый проект в MSVS типа MFC, и дополнительно указать что мы ходим создать именно пользовательское приложение с использованием MFC (MFC Application). Также необходимо задать имя проекта, рабочего пространства и их местоположение на диске (рис. 8).

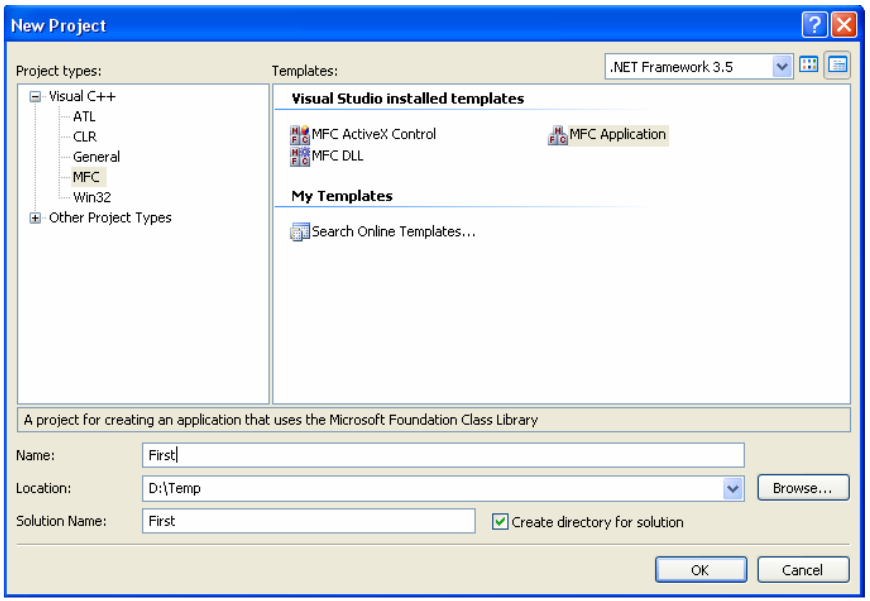


Рис. 8. Создание пользовательского приложения MFC

12.3.1. Типы приложений MFC

Существует три основных типа пользовательских приложений с использованием MFC:

- ***Single Document*** – приложение, позволяющее работать только с одним документом. Примером такого приложения могут быть программы текстовый редактор Блокнот или графический редактор Paint.
- ***Multiple Documents*** – приложение для одновременной работы с несколькими документами или с несколькими копиями одного и того же документа. Примером такого приложения могут быть некоторые версии текстового редактора Word, Adobe PhotoShop.
- ***Dialog Based*** – приложение, основанное на диалоге, имеет отличительную черту – диалоговую панель, используемую для ввода данных в программу пользователем или для вывода результатов вычислений на экран.

Мы будем создавать приложение диалогового типа (рис. 9).

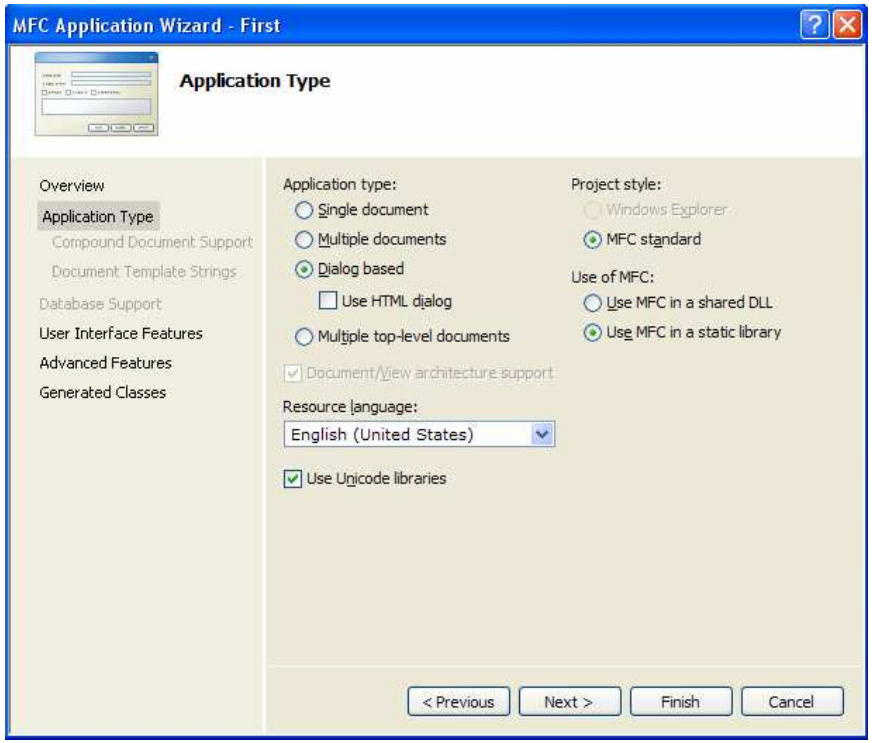


Рис. 9. Выбор типа приложения MFC

Обратите внимание на вариант использования библиотеки MFC:

- ***Use MFC in a shared DLL*** – такой способ использования подходит если приложение будет запускаться на ПК с установленной MSVS, либо отдельно установленной библиотекой MFC.
- ***Use MFC in a static library*** – в данном варианте результатом будет приложение в которое уже включены все необходимые функции из библиотеки MFC, что обеспечит возможность его запуска практически на любом ПК с установленной ОС Windows.

12.3.2. Дополнительные параметры создания приложения

Вы можете задать возможную трансформацию вашего диалога: возможность сворачиваться, работать в полноэкранном режиме, иметь описание (***About Box***) и так далее. Помимо этого, на данном этапе задается имя диалога – то самое имя, которое будет отображаться в его названии при запуске (рис. 10).

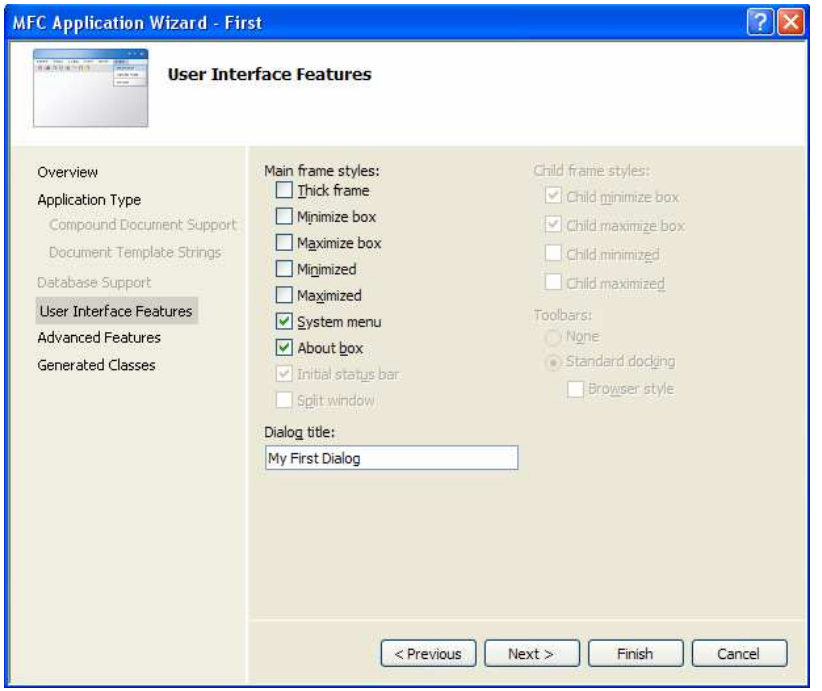


Рис. 10. Дополнительные параметра приложения MFC

На завершающем шаге выводится информация о том какие классы будут сгенерированы для приложения, и в каких файлах будет находится их описание (рис. 11).

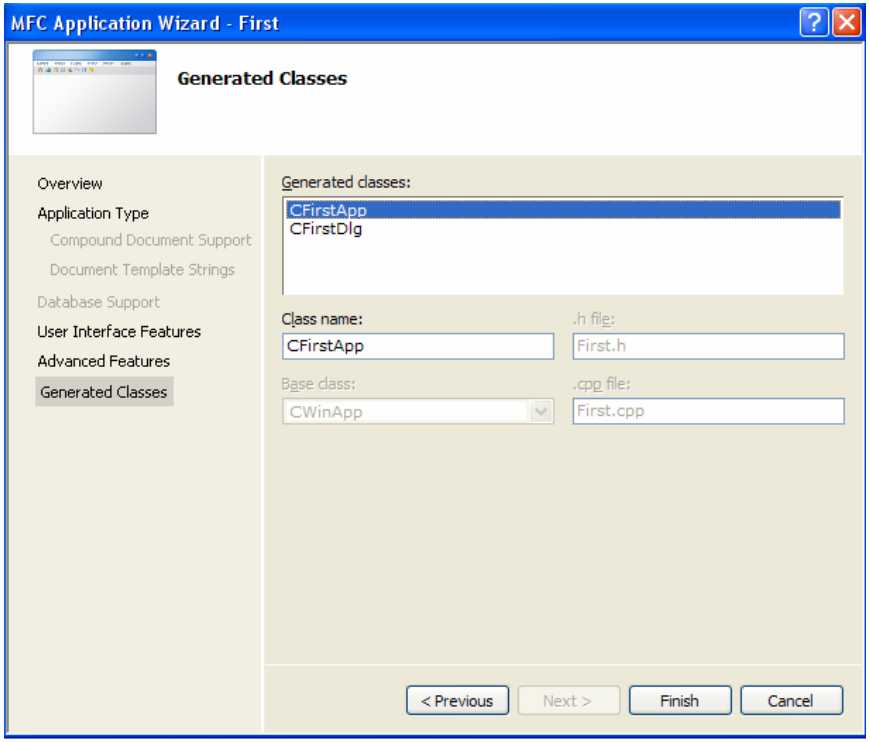


Рис. 11. Завершающий шаг создания приложения MFC

12.4. Разработка приложений на базе MFC

После того как все шаги мастера по созданию приложений пройдены, появляется рабочее пространство подобное представленному на рис. 12.

По умолчанию, на диалоговом окне всегда присутствуют две кнопки: ***OK*** и ***Cancel***, и тестовое поля в котором можно делать статические надписи. При желании, все это можно легко удалить с вашего диалога.

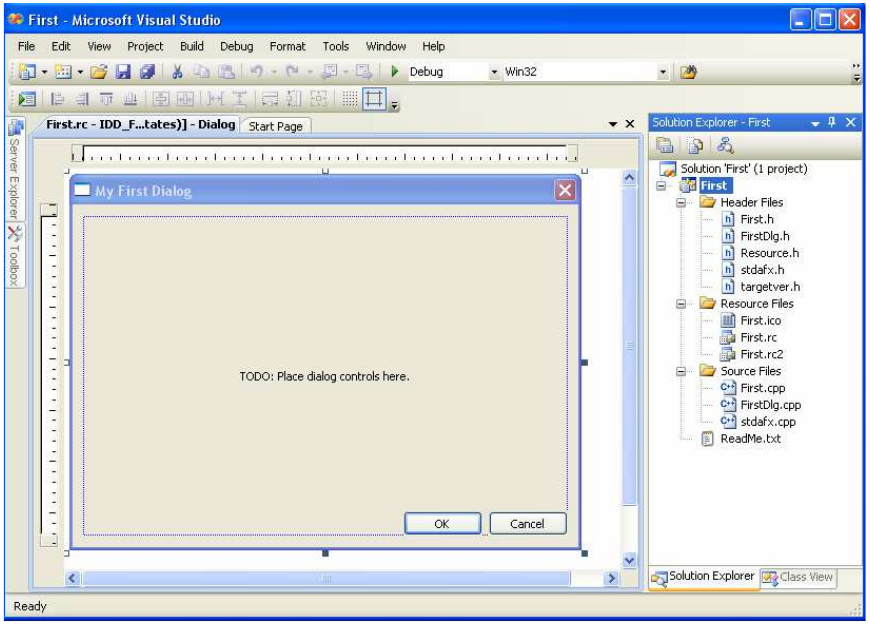


Рис. 12. Рабочее пространство разработки приложения MFC

Справа (или слева, в зависимости от настроек MSVS), имеется знакомое окно ***Solution Explorer***, с вкладками списка файлов проекта и ***Class View*** (рис. 13).

При необходимости сделать какие-либо действия при инициализации диалогового окна или приложения, это делается путем добавления кода в функции ***OnInitDialog()*** и ***OnInitInstance()*** соотвественно.

Места, в которых разрешается пользователю писать код выделены следующим образом:

```
// TODO: Add extra initialization here
```

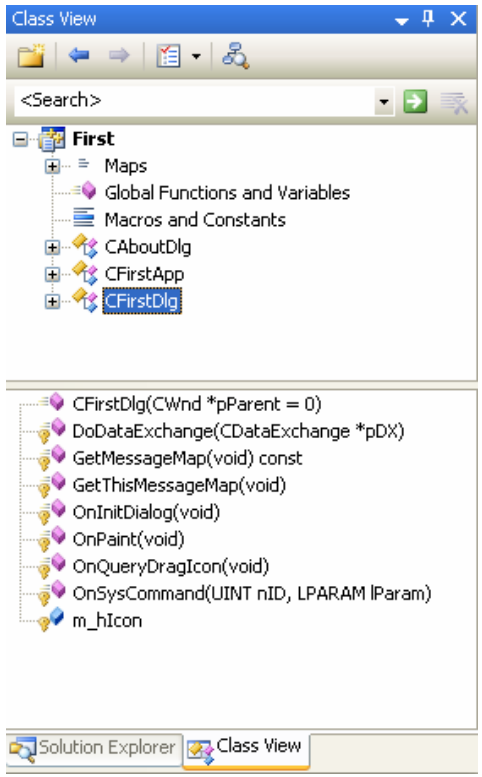


Рис. 13. Окно Class View

Для облегчения добавления различных элементов добавления в оконное приложение, существует вкладка ***Toolbox***, включающая в себя все основные элементы управления и взаимодействия с пользователем. Для того чтобы добавить элемент управления на окно, он выделяется на Toolbox, после чего мышкой выделяется место в окне, куда он будет добавлен. Обратите внимание, что мышкой задается регион (левый верхний угол и затем, держа левую кнопку мыши, правый левый), таким образом задается и положение и размер элемента управления (рис. 14).

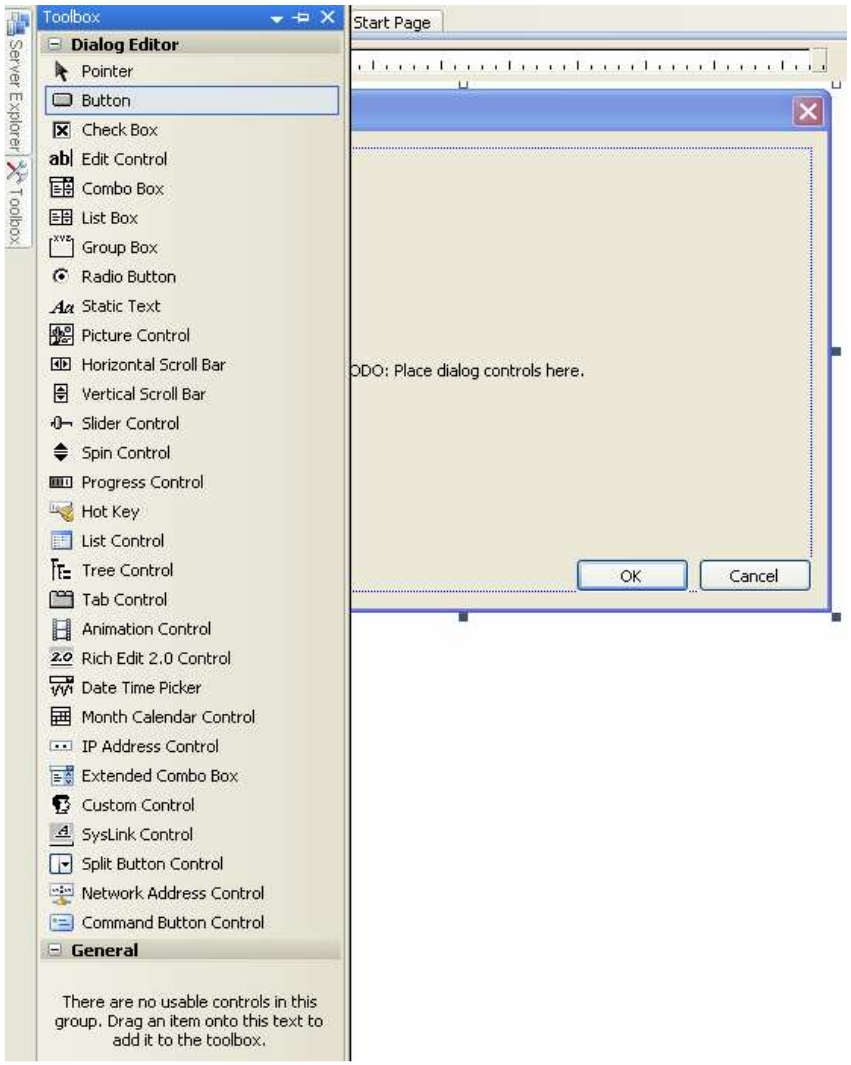


Рис. 14. Вкладка Toolbox с элементами управления

После того, как элемент управления добавлен, в правой части экрана активируется окно свойств этого элемента управления, в котором задается в том числе и его имя в коде программе (***ID***), а также его подпись в пользовательском окне (***Name***) (рис. 15).

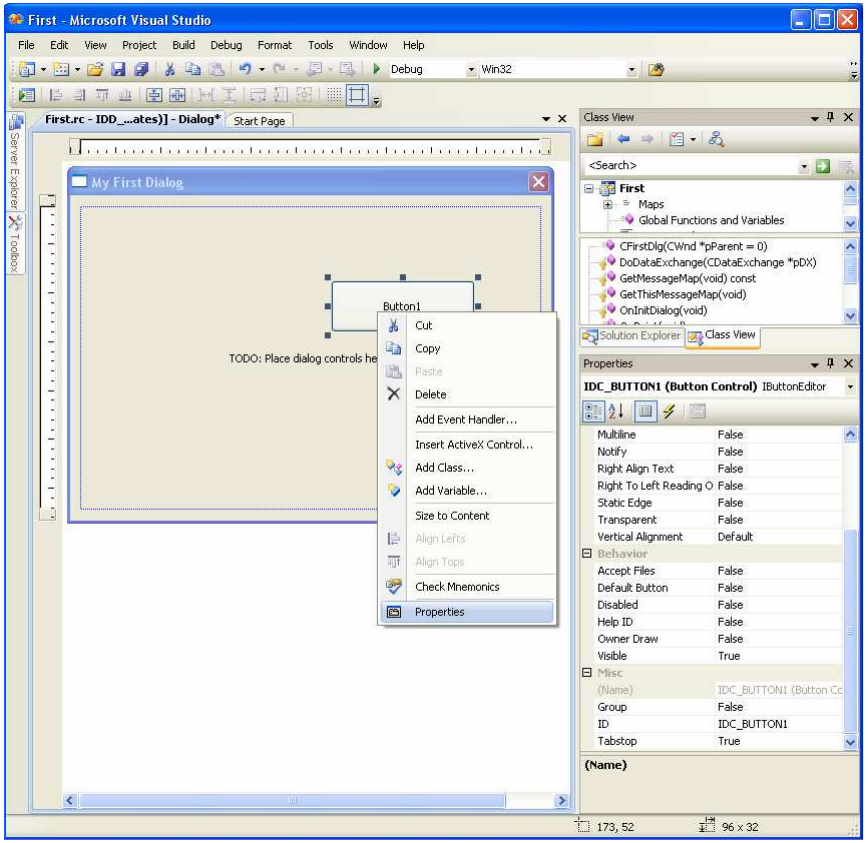


Рис. 15. Свойства элемента управления

При двойном щелчке мышкой по элементу управления в коде генерируется каркас функции для обработки элемента управления следующего вида:

```
void CFirstDlg::OnBnClickedButtonMy()
{
   // TODO: Add your control notification handler code here
}
```


Другими словами, в нашем примере эта функция будет вызываться при нажатии пользователем на эту кнопку.

Добавим код, который будет при нажатии на кнопку выводить сообщение с текстом Hello. Поскольку у нас оконное приложение, то мы можем для сообщение создать отдельное окно и вписать в него текст следующим образом:

```
void CFirstDlg::OnBnClickedButtonMy()
{
   CString My("Hello");
   AfxMessageBox(My);
}
```

В таком случае вид нашего приложения при нажатии кнопки будет следующим (рис. 16):

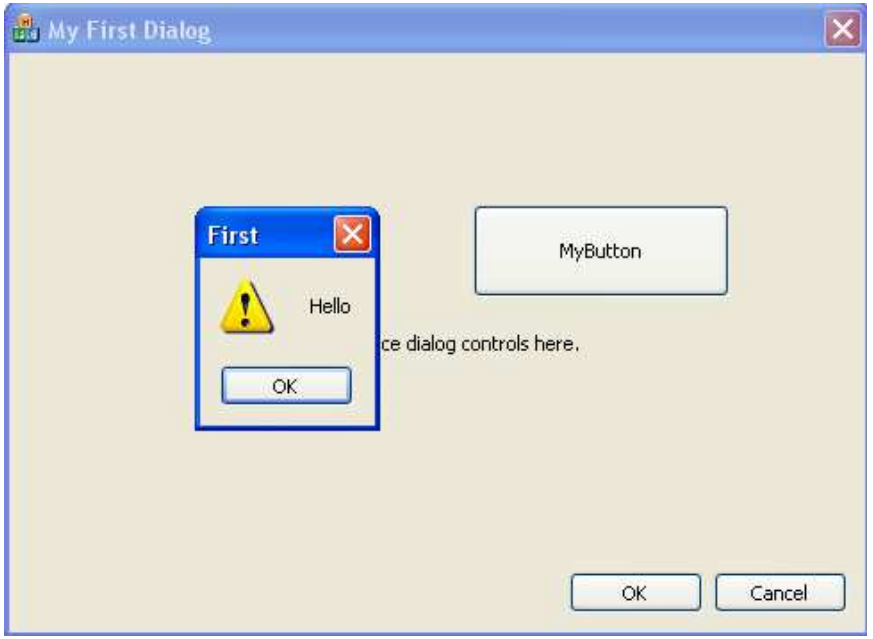


Рис. 16. Приложение, создающее окно типа ***AfxMessageBox***

12.5. Создание приложение «калькулятор»

Поставим задачу – разработать программу типа простейшего калькулятора со следующим пользовательским интерфейсом (рис. 17):

Рис. 17. Приложение «калькулятор»

Для этого нам понадобиться два типа элементов управления: ***Button*** и ***Edit Control***.

Разберем ключевые места для создания такого приложения.

12.5.1. Использование CButton

В случае создания калькулятора нам понадобится обрабатывать нажатие каждой кнопке.

В идеологии MFC, нажатие кнопки – это событие. Соответственно для каждого события в системе возможно создать обработчик - функцию которая будет вызываться и производить определенные действия.

Для того чтобы добавить функцию-обработчика событий в среде предусмотрен специальный мастер. Он вызывается при нажатии правой клавиши мыши по элементу управления и выбору меню «***Add Event Handler***» (рис. 18).

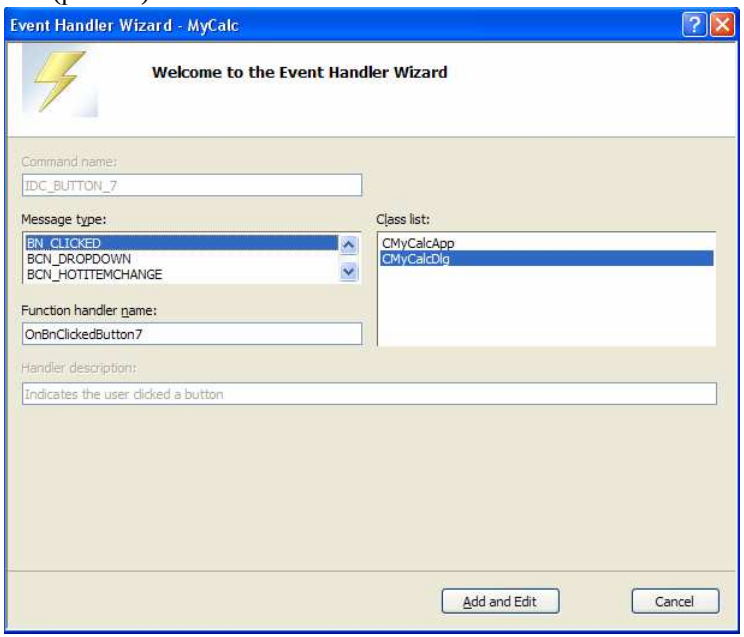


Рис. 18. Мастер добавления обработчика событий

В случае создания калькулятора нам понадобятся сообщения ***BN_Clicked*** у каждой кнопки.

12.5.2. Использование CEditWindow

Поскольку нам необходимо каким-то образом управлять данным объектом (как минимум обновлять содержимое в нем), необходимо завести переменную, соответствующую объекту этого класса. Для этого можно воспользоваться мастером добавления переменных – правой клавиши мыши по элементу управления и выбору меню «***Add Variable***» (рис. 19).

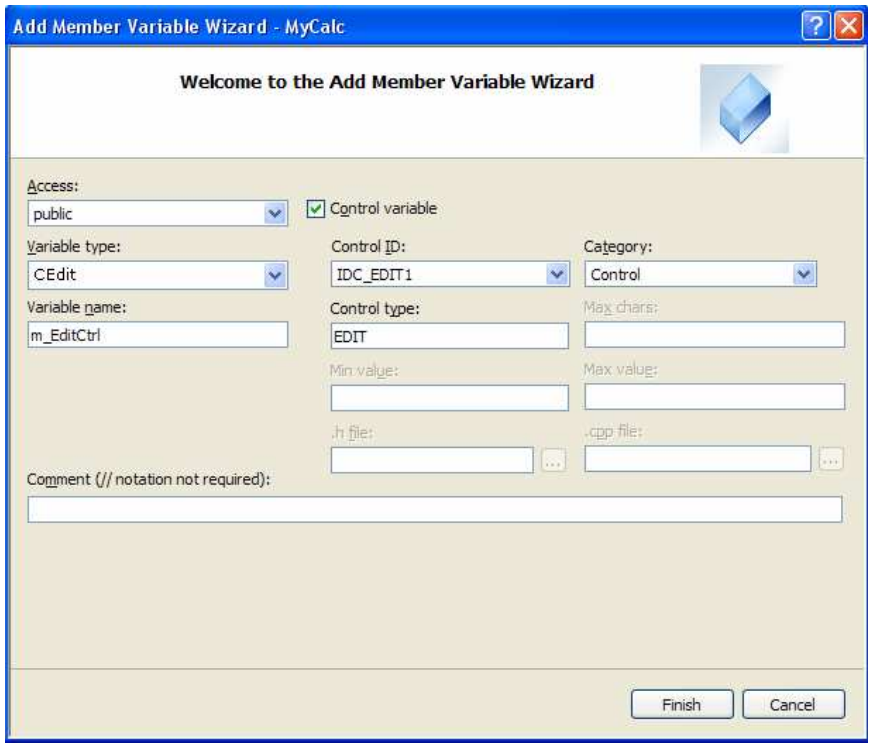


Рис. 19. Мастер добавления обработчика событий

В нашем случае необходимо создать переменную для управления, поэтому устанавливается категория ***Control*** и указывается ее имя (например ***m_EditCtrl***).

Для обеспечения невозможности вода пользователем текста в наше окно, можно прописать вызов следующей функции в OnInitDialog нашего диалога:

```
BOOL CMyCalcDlg::OnInitDialog()
{
   ...

   // TODO: Add extra initialization here
   m_EditCtrl.SetReadOnly(true);
}
```


Содержимое ***CEdit*** всегда сопоставляется со строкой. В C++ (в частности в MFC) зачастую используется класс ***CString*** для работы со строками. Поэтому вывод в окно какой-либо строки и его обновление осуществляется следующим образом:

```
CString My("Sample text");
m_EditCtrl.SetWindowText(My);
```

12.5.3. Особенности работы с CString

Поскольку калькулятор должен отображать численные переменные, то их необходимо преобразовывать к строковому типу. Одним из способов является использование следующих функций:

- `char* _itoa(int val, char* string, int radix);`
- `char* _gcvt(double val, int digits, char* buff);`

Фрагмент кода для перевода переменной типа double в 10-значную строку типа CString приведен ниже:

```
double q = 12.34556;
char* p = new char[10];
_gcvt(q, 10, p);

CString My(p);
```

12.5.4. Хранение временных переменных

Программисту часто необходимо переменные, область видимости которых превышает какую-либо функцию.

В идеологии MFC вы можете использовать для этого либо глобальные переменные, либо создавать переменные внутри диалогового класса (***CMyFirstDialog*** в предыдущем разделе).

Необходимо помнить, что проекты с использованием MFC содержать весьма большое количество различных классов, поэтому не рекомендуется

создавать для всей иерархии глобальные переменные, которые нужны только внутри одного экземпляра одного диалогового класса.

12.6. Графические объекты Windows в MFC

Графические объекты представляют собой инструменты для рисования в контексте дисплея: перья, кисти, шрифты, растровые изображения, области, палитры. В MFC включено несколько классов, предназначенных для создания графических объектовWindows. Так как графические объекты в Windows доступны через дескрипторы, каждому дескриптору ставится в соответствие класс MFC.

Инструменты	Тип дескриптора	КлассMFC
Перья	HPEN	CPen
Кисти	HBRUSH	CBrush
Шрифты	HFONT	CFont
Растровые изображения	HBITMAP	CBitmap
Области	HRGN	CRgn
Палитры	HPALETTE	CPalette

Все классы графических объектов имеют один базовый класс – ***CGdiObject***, который в свою очередь имеет базовый класс ***CObject***.

Рассмотрим порядок работы с графическими объектами. Объявим объект в пределах блока программного кода и выполним инициализацию методом ***Create()*** для конкретного объекта, например, для палитры следует использовать метод ***CreatePalette()***. Выбираем новый объект в текущем контексте: ***CDC::SelectObject()***. Метод возвращает указатель на замещаемый объект (палитру, кисть и т. д.).

После выполнения графической операции возвращаем замещенный графический объект обратно в контекст.

12.6.1. Графические операции в MFC

Графические операции в MFC представляют собой методы класса ***CDC*** и его производных классов. Рассмотрим примеры построения графических примитивов.

```
CPaintDC dc(this);
dc.TextOut(200,200,_T("TextOut Samples"));
```

В данном случае если такой код добавить в метод ***OnPaint***() вашего диалогового окна, то в координате (200,200) относительно верхнего левого угла окна появится соответствующая надпись.

Вывод точки в заданной координате и цветом в пространстве RGB:

```
CPaintDC dc(this);
dc.SetPixel(200,200,RGB(255,0,0));
```

Рисование дуги окружности:

```
dc.Arc(x1,y1,x2,y2,x3,y3,x4,y4)
```

где (x1,y1,x2,y2) – координа тыпрямоугольника, в который вписывается эллипс, (x3,y3) – начальная точка, (x4,y4) – конечная точка прямой линии, отсекающей от эллипса дугу. Дуга чертится против часовой стрелки от точки (x3,y3) до точки (x4,y4).

```
CPaintDC dc(this);
dc.Arc(200,200,100,100,400,400,10,10);
```

Рисование эллипса:

```
CPaintDC dc(this);
dc.Ellipse(350,300,50,150);
```

Рисование прямой линии:

```
CPaintDC dc(this);
dc.MoveTo(200,200); //Установка курсора
dc.LineTo(100,100); //Рисование линии
```

12.6.2. Квадрат в центре диалогового окна

Здесь представлен код, который будет рисовать белый квадрат в центре рабочей области пользовательского диалогового окна.

Этот код, разумеется, должен быть расположен в методе ***OnPaint()*** вашего диалогового окна,

```
CPaintDC dc(this); //Объявление контекста экрана
CRect rect; //Объект для хранения координат квадрата
CRect clientrect; // хранения координат клиентской
                  // области окна
GetClientRect(clientrect); //Получение координат
                    // клиентской области окна

//Координаты центра клиентской области окна
int mx=clientrect.left+(clientrect.right-
clientrect.left)/2;
int my=clientrect.top+(clientrect.bottom-
clientrect.top)/2;

//Длина стороны квадрата = 50
int width=50;

//Координаты прямоугольника
rect.bottom=my+width;
rect.left=mx-width;
rect.top=my-width;
rect.right=mx+width;

dc.Rectangle(rect); //Рисование прямоугольника
```


Учебное издание

Лысаков Константин Федорович

ИНФОРМАТИКА И ПРОГРАММИРОВАНИЕ (C++)

УЧЕБНОЕ ПОСОБИЕ