

ФЕДЕРАЛЬНОЕ АГЕНТСТВО ПО ОБРАЗОВАНИЮ
МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ СТРОИТЕЛЬНЫЙ УНИВЕРСИТЕТ

Факультет информационных систем, технологий и автоматизации в строительстве

Кафедра информационных систем и технологий управления в строительстве



КОНСПЕКТ ЛЕКЦИЙ
по курсу
«Технология программирования»

Б. С. Садовский

Москва 2009

Конспект лекций предназначен для студентов специальности 230102, изучающих дисциплину «Технология программирования». Содержит минимальный теоретический материал необходимый для освоения Си++, как объектно-ориентированного языка программирования, и выполнения лабораторных работ по курсу. Лекции составлены на основе литературы представленной в конце конспекта и оригинальных материалов разработанных самим автором лекций.

Оглавление

1 Состав и выбор инструментальных средств разработки в Microsoft® Windows®.....	5
2 Описание использования и настройки ИСР (IDE) Code::Blocks.....	9
2.1 Общие описание ИСР Code::Blocks.....	9
2.2 Установка (для использование с компилятором GCC).....	9
2.3 Настройка.....	9
2.4 Работа.....	10
2.4.1 Создание проекта.....	10
2.4.2 Работа с проектом.....	10
2.5 Удаление.....	10
3 Проблема ввода/вывода в консоли Windows XP®	11
4 Язык Си.....	17
4.1 Операторы и операции.....	20
4.2 Директивы препроцессора.....	22
4.2.1 Подробнее про директиву препроцессора #define.....	22
4.3 Типы данных.....	23
4.4 Массивы.....	24
4.4.1 Одномерный массив.....	24
4.4.2 Двумерный массив.....	24
4.5 Структуры.....	25
4.6 Объединения.....	26
4.7 Перечисления.....	27
4.8 Указатели.....	28
4.8.1 Использование указателей при работе со структурой.....	31
4.9 Функции.....	32
4.9.1 Прототипы функции.....	34
4.9.2 Функция не принимающая аргументов и не возвращающая значений (тип данных void).....	34
4.9.3 Передача аргументов (параметров) в функцию по значению.....	35
4.9.4 Передача аргументов (параметров) в функцию через указатель.....	35
4.9.5 Передача массива в функцию как параметра.....	35
4.9.6 Аргументы функции main() и возвращаемое значение.....	36
5 Основы Си++, система ввода/вывода, типы данных, функции и другие возможности.....	38
5.1 Основы Си++.....	38
5.2 Типы данных, ввод/вывод и функции в Си++.....	40
5.2.1 Поток ввода/вывода.....	40
5.2.2 Строки	41
5.2.3 Файловый ввод/вывод.....	42
5.2.4 Булевый (логический) тип данных.....	45
5.2.5 Новый синтаксис инициализации переменной.....	45
5.2.6 Встраиваемые (встроенные, подставляемые, inline) функции (методы).....	45
5.2.7 Аргументы функции, определяемые по умолчанию.....	47
5.2.8 Ссылки.....	47
5.2.9 Передача аргумента в функцию через ссылку.....	48
5.2.10 Константы.....	49
5.2.11 Константы, ссылки и указатели.....	50
5.2.12 Динамическое выделение памяти.....	51
5.3 Пространство имён Си++.....	54
Пространства имён структур и объединений.....	55

Определяемые пользователем пространства имён в Си++.....	56
5.3.1 Объявления using и директивы using.....	58
6 Классы, объекты и инкапсуляция.....	60
6.1 Описание классов и объектов.....	60
6.2 Конструктор и деструктор.....	63
6.3 Указатель на объект и this.....	65
6.4 Передача аргумента по ссылке на объект.....	66
6.4.1 Использование констант в классах	67
Константные методы.....	67
Способы создания констант в классе.....	67
6.5 Друзья.....	69
6.5.1 Дружественные функции.....	69
6.5.2 Дружественные классы	71
6.5.3 Дружественные функции-члены.....	72
7 Наследование и полиморфизм.....	74
7.1 Передача параметров в базовый класс через конструктор.....	76
7.2 Полиморфизм.....	77
7.2.1 Перегрузка функций (полиморфизм функций).....	77
7.2.2 Перегрузка операций.....	78
7.2.3 Полиморфное наследование (динамический полиморфизм).....	81
Переопределение методов.....	81
Виртуальные функции.....	83
7.3 Множественное наследование.....	86
7.3.1 Виртуальный класс.....	90
8 Не объектно-ориентированные средства языка Си++.....	92
8.1 Шаблоны.....	92
8.1.1 Шаблоны функций.....	92
8.1.2 Шаблоны классов.....	93
8.2 Стандартная библиотека шаблонов (STL).....	94
8.2.1 Контейнеры.....	94
8.2.2 Итераторы	96
8.2.3 Функциональные объекты (Функторы).....	96
8.2.4 Алгоритмы.....	96
8.3 Работа с исключениями.....	97
Литература.....	98

1 Состав и выбор инструментальных средств разработки в Microsoft® Windows®

На сегодня существует много различных систем по созданию программного обеспечения (ПО), все они различаются: набором инструментальных средств, документацией, лицензиями и т. д.

Но все они состоят из следующих модулей:

А) Транслятор (компилятор, интерпретатор, ассемблер)

Транслятор — программа, которая принимает на вход программу на одном языке (он в этом случае называется исходный язык) и преобразует её в программу, написанную на другом языке.

I. Компилятор — это транслятор, который осуществляет перевод всей исходной программы в эквивалентную ей результирующую программу на языке машинных команд или на языке ассемблера.

Процесс компиляции состоит из следующих этапов:

1. **Лексический анализ.** На этом этапе последовательность символов исходного файла преобразуется в последовательность лексем. **Лексе́ма** (от греч. *lexis* — слово, выражение, оборот речи).

2. **Синтаксический (Грамматический) анализ.** Последовательность лексем преобразуется в дерево разбора.

3. **Семантический анализ.**

Дерево разбора обрабатывается с целью установления его семантики (смысла) - напр. привязка идентификаторов к их декларациям, типам, проверка совместимости, определение типов выражений и т. д. Результат обычно называется

«промежуточным

представлением/кодом» и может

быть дополненным деревом разбора, новым деревом, абстрактным набором команд или чем-то еще, удобным для дальнейшей обработки.

4. **Оптимизация.** Выполняется удаление излишних конструкций и упрощение кода с сохранением его смысла. Оптимизация может быть на разных уровнях и этапах, напр. над промежуточным кодом или над конечным машинным кодом.

5. **Генерация кода.** Из промежуточного представления выдается код на целевом языке.

В конкретных реализациях компиляторов, эти этапы могут быть разделены или совмещены в том или ином виде.

II. Интерпретатор — транслятор для непосредственного исполнения программ (производства вычислений, предписываемых этими программами) из исходного кода на определённом языке. Простые интерпретаторы анализируют и выполняют (интерпретируют) программу последовательно (покомандно или построчно). Синтаксические ошибки обнаруживаются такими интерпретаторами только когда интерпретатор приступает к выполнению команды (строки) содержащей ошибку, это

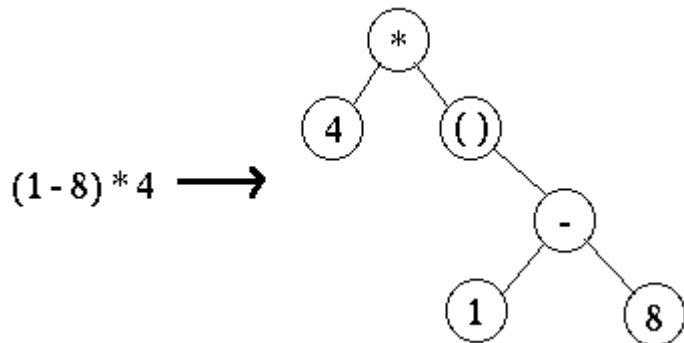


Рис. 1.1. Простой пример «Дерева разбора».

может быть удобно начинающим.

III. Ассемблер (от англ. *assemble* — собирать) — компилятор с языка ассемблера в команды машинного языка. Русифицированное название — *мнемокод*. Предназначен для представления в удобном (мнемоническом) виде машинные коды команд. Пример команд на мнемокоде: СЛЖ — сложить, ВЧТ — вычесть, ПРХ — переход и другие. Обеспечивает наиболее эффективное использование ресурсов системы (процессор, память, периферия). Используется в «узких» местах — требуется большое быстродействие, ограничение по размеру оперативной памяти и другие. Ассемблером также называют иногда саму систему команд центрального процессора. Под каждую архитектуру процессора и под каждую ОС или семейство ОС существует свой Ассемблер. Существуют также, так называемые «кросс-ассемблеры», позволяющие на машинах с одной архитектурой (или в среде одной ОС) ассемблировать программы для другой целевой архитектуры или другой ОС, и получать исполняемый код в формате, пригодном к исполнению на целевой архитектуре или в среде целевой ОС.

При сборке проекта появляются объектные файлы, например с расширениями:

- .obj (обычно в Windows)
- .o (обычно в *nix)
- .tds (C++ Builder)
- moc_ИМЯ_МОДУЛЯ.cpp (при сборке Qt программ)

При редактировании файлов Си и Си++ также могут появляться файлы, которые являются backup копией файла. Обычно содержат в имени знак (~) (тильда), например:

- ИМЯ_МОДУЛЯ.~cpp
- ИМЯ_МОДУЛЯ.~h
- ИМЯ_МОДУЛЯ.cpp~
- и т. д.

Объектный модуль (также — объектный файл, англ. *object file*) — файл с промежуточным представлением отдельного модуля программы, полученный в результате обработки исходного кода компилятором. Объектный файл содержит в себе, особым образом подготовленный, код (часто называемый бинарным), который может быть объединён с другими объектными файлами при помощи редактора связей (линковщика), для получения готового исполняемого модуля, либо библиотеки.

В) Компоновщик

Компоновщик (также **редактор связей** или **линковщик**, англ. *linker, link editor*) — программа, которая принимает на входе один или несколько объектных модулей и собирает по ним исполняемый модуль.

Для связывания модулей линковщик использует таблицы имён, созданные компилятором в каждом из объектных модулей. Такие имена могут быть двух типов:

Определённые или экспортируемые имена — функции и переменные, определённые в данном модуле и предоставляемые для использования другим модулям

Неопределённые или импортируемые имена — функции и переменные, на которые ссылается модуль, но не определяет их внутри себя

Работа компоновщика заключается в том, чтобы в каждом модуле разрешить (найти) ссылки на неопределённые имена. Для каждого импортируемого имени находится его определение в других модулях, упоминание имени заменяется на его адрес.

С) Текстовый редактор

Текстовый редактор — компьютерная программа, предназначенная для создания и изменения текстовых файлов, а также их просмотра на экране, вывода на печать, поиска фрагментов текста и т. п.

Для написания программ подходит любой текстовый редактор, даже простой блокнот Microsoft® Windows®, но удобней использовать специализированные редакторы, так как они предоставляют хорошую функциональность. Элементами этой функциональности являются: подсветка синтаксиса, конвертация текста разных кодировок, хорошую систему поиска и замены текста, сортировку строк, показ кодов символов, авто-форматирование текста, фолдинг.

Д) Отладчик

Отладчик — модуль среды разработки или отдельное приложение, предназначенное для поиска ошибок в программе. Отладчик позволяет выполнять пошаговую трассировку, отслеживать значения переменных в процессе выполнения программы, устанавливать точки или условия останова и т. д.

Е) другие средства облегчающие написание программ

Другие средства — это разнообразные средства, упрощающие проектирование, написание, документирование ПО. Обычно к ним относят следующие: дизайнер, система управления версиями, браузер классов, инспектор объектов, авто-дополнение кода и т. д.

Дизайнер, позволяющий визуально создавать графический интерфейс пользователя (ГИП). Дизайнер представляет собой визуальную среду, в которой можно с помощью мыши располагать на форме компоненты программы, такие как: кнопки, панели, радиокнопки, поля ввода, метки и т. д.

Система управления версиями — программное обеспечение для облегчения работы с изменяющейся информацией. Система управления версиями позволяет хранить несколько версий одного и того же документа, при необходимости, возвращаться к более ранним версиям, определять, кто и когда сделал то или иное изменение и многое другое.

Браузер классов или **исследователь классов** (*Class Explorer*) позволяет просматривать и отображать в удобной форме список всех типов, классов, свойств, методов, глобальных переменных и функций, содержащихся в модуле или программе.

Инспектор объектов обеспечивает удобный интерфейс для изменения свойств объектов, и управление событиями, на которые реагирует объект.

Авто-конструирование и **авто-дополнение** кода позволяют автоматизировать и упростить написание ПО, за счёт автоматического написания необходимых, но общих строк кода самой средой разработки, а так же авто-дополнение обеспечивает удобный поиск необходимых методов и данных, действительных внутри конкретного класса, шаблона или структуры.

Интегрированная среда разработки (ИСР) (англ. *IDE – Integrated Development Environment*) — система программных средств, используемая программистами для

разработки программного обеспечения. А также поддерживает концепцию **RAD**. **RAD** (англ. ***Rapid Application Development***) быстрая разработка приложений – концепция создания средств разработки программных продуктов, уделяющая особое внимание скорости и удобству программирования, созданию технологического процесса, позволяющего программисту максимально быстро создавать компьютерные программы. Концепцию RAD также часто связывают с концепцией визуального программирования. **Среда визуальной разработки** — среда разработки программного обеспечения, в которой наиболее распространённые блоки программного кода представлены в виде графических объектов. Применяются для создания прикладных программ и любительского программирования. Среда визуальной разработки – это частный случай ИСР.

Обычно она включает в себя описанные выше средства, такие как, компилятор и интерпретатор, либо что то одно, компоновщик, текстовый редактор, средства автоматизации сборки приложения и отладчик, а также различные дополнительные средства.

На сегодняшний день наибольшее распространение получили следующие IDE:

- Microsoft® Visual Studio®
- Borland® C++ Builder®
- Eclipse
- Borland® Kylix®
- Code::Blocks и другие.

Более полный список можно посмотреть в [14].

Для выполнения практических заданий лучшим выбором является IDE Code::Blocks, поскольку обладает следующими преимуществами:

- является бесплатной и свободно распространяемой;
- довольно проста в использовании;
- компилятор и отладчик не входят в её состав, что **позволяет использовать любой распространённый компилятор**, но предпочтительно использовать компилятор GCC (входит в состав MinGW) и отладчик Gdb (*GNU debugger*);
- **GCC (*GNU Compiler Collection*)** современный компилятор наиболее полно соответствует стандарту ISO/IEC 14882:2003;
- имеет удобный текстовый редактор кода;
- встроенные средства для автоматизации разработки;
- Имеет очень небольшой размер в запакованном виде около 10 Мб.
- Имеет типовой интерфейс, что позволяет без лишних проблем перейти на другие IDE;
- за счёт открытости и модульности может свободно масштабироваться;
- среда является кросс-платформенной (не зависит от ОС).

Выявленные недостатки:

- Большое количество программных ошибок при работе ИСР (программа очень динамично развивается, по этому имеются основания считать, что ошибки носят временный характер и в последствии будут устранены);

2 Описание использования и настройки ИСР (IDE) Code::Blocks

2.1 Общие описание ИСР Code::Blocks

Code::Blocks имеет типичный для IDE внешний интерфейс на английском языке.

2.2 Установка (для использование с компилятором GCC)

Для работы в этой IDE необходимо установить следующие компоненты;

- компилятор GCC;
- отладчик Gdb;
- саму среду Code::Blocks.

Все эти программы можно получить у преподавателя, либо скачать с сайта кафедры, или с сайта разработчиков этих проектов. В конце методички есть ссылки на эти сайты. Детально, процесс установки описан в отдельном документе, предоставляемым студенту преподавателем курса.

Компилятор GCC входит в состав пакета утилит разработки ПО MinGW. Установка осуществляется запуском файла *MinGW.exe*. Далее в каждом диалоге следует нажимать **Next**, если не требуется каких либо специальных настроек. В последнем диалоге для закрытия окна следует выбрать **Finish**.

Установка Gdb осуществляется аналогичным MinGW образом описанным выше.

Установка самой среды осуществляется распаковкой её из архива *CodeBlocks.rar*, в предварительно созданную директорию, например CodeBlocks в корне диска C:\ или другое место (выбор места установки по желанию студента). Для удобства запуска рекомендуется создать ярлык программы на рабочем столе.

При установке Code::Blocks версии взятой с сайта разработчиков необходимо иметь установленный на компьютере архиватор 7-Zip. Архиватор 7-Zip можно скачать здесь [10].

2.3 Настройка

При первом запуске, программа спросит: какой компилятор использовать и будет ли она программой по умолчанию для всех файлов *.c/*.cpp и *.h/*.hpp?

Необходимо проверить, в какой кодировке вы работаете, это можно сделать, посмотрев в строку состояния внизу экрана. Справа будет текущая кодировка ИСР. Пример того, что может быть написано: **UTF-8, WINDOWS-1251** и т. д. Если у вас установлена в качестве кодировки по умолчанию, кодировка отличная от UTF-8, то её надо будет сменить на UTF-8. Это можно сделать, зайдя в **Settings->Editor...**, после чего откроется окно **Configure editor** настроек текстового редактора кода. Здесь, в выпадающем списке **Default encoding when opening files**, надо выбрать кодировку **UTF-8**. После чего нажать **OK** и переоткрыть все открытые файлы в ИСР.

Если программа не компилируется и в панели **Messages** во вкладке **Build log** появляется сообщение: *«cclplus.exe: error: unrecognized command line option „-Wfatal-errors“»*, надо зайти в пункт меню **Settings -> Compiler and debugger...**, и в открывшемся окне, при условии, что выбран компилятор **GNU GCC Compiler**, в соответствующем, выпадающем

списке **Selected compiler**, во вкладке **Compiler Flags**, надо снять галочку напротив пункта «*Stop compiling after first error [-Wfatal-errors]*». Нажать на кнопку **OK**.

Если панели **Messages** внизу экрана невидны, то нажмите клавишу **F2**, чтобы отобразить её.

2.4 Работа

2.4.1 Создание проекта

В общем случае, создание программы начинается с создания проекта программы. В Code::Blocks для этого надо выбрать в меню **File->New->Project...** В появившемся окне выбрать **Console application**. Далее, в появившемся диалоговом окне нажать кнопку **Next**. В следующем диалоге следует указать название проекта, где он будет расположен, имя файла проекта и после ввода нажать кнопку **Next**. В следующем диалоге надо выбрать компилятор по умолчанию, стоит GCC, также надо отметить: какие режимы будет использовать ваш проект **Debug** - режим сборки приложения для отладки **Release** - режим сборки готового приложения. После выбора всего необходимого, следует нажать кнопку **Next**. В последнем диалоговом окне надо выбрать язык разработки приложения Си или Си++. И нажать кнопку **Finish**.

Подробнее создание проекта описано здесь [9].

2.4.2 Работа с проектом

При открытии нового проекта, надо в правой панели Management выбрать проект, и сделать двойной щелчок указателем мыши по объекту с именем *Source*. Список файлов проекта. Надо сделать ещё один щелчок по файлу, которой надо редактировать.

Подробнее создание проекта описано здесь [9].

2.5 Удаление

Для удаления среды необходимо просто удалить каталог с её файлами, куда была осуществлена установка. Для удаления MinGW и Gdb необходимо зайти в **Панель управления**, запустить **Установка и удаление программ**. Найти в списке установленных программ MinGW и Gdb, и выбрать удалить. Больше процесс удаления IDE никаких действий не требует.

3 Проблема ввода/вывода в консоли Windows XP®

К сожалению при работе программы в консоли Windows возникают проблемы правильного отображения символов кириллицы. Есть несколько способов решить данную проблему. Эти способы будут рассмотрены далее. По идее какой-нибудь из них должен заработать.

Для начала необходимо убедиться, что исходный текст программы, который вы набираете, является текстом в кодировке UTF-8. Как это можно сделать для программы Code::Blocks описано в 2.3. Для других редакторов смотрите руководства по соответствующим IDE.

Подробнее можно почитать здесь[15].

Способ №1

Способ заключается в использовании функции `WriteConsoleW()`. Эта функция использует непосредственно вывод в консоль необходимого текста.

Чтобы работать с этой функцией, надо подключить заголовочный файл `windows.h`.

```
#include <windows.h>
```

А использовать её можно, например так:

```
#include <iostream>
#include <windows.h>

using namespace std;

int main()
{
    wchar_t * wstr = L"Привет мир!\n"; // наша строка для вывода
    DWORD written;
    WriteConsoleW( GetStdHandle(STD_OUTPUT_HANDLE), wstr,
                  wcslen(wstr) + 1, &written, 0);

    return 0;
}
```

где `wchar_t` — тип данных для работы с широкими символами, `L` — указывает, что передаваемая строка состоит из широких символов.

Способ №2

Можно воспользоваться функцией `CharToOemW()` для конвертации строк во время компиляции.

Чтобы работать с этой функцией надо подключить заголовочный файл `windows.h`.

```
#include <windows.h>
```

Использовать её можно так:

```
#include <iostream>
#include <windows.h>

using namespace std;
```

```

int main()
{
    char buf[255]; // буфер — куда будет помещена
                  // конвертированная из UTF-8 в 866 строка.
    CharToOemW(L"Привет мир!\n", buf); // конвертация строки
    cout << buf << endl; // вывод строки в кодировке 866

    return 0;
}

```

Или, более удобный способ с использованием функции:

```

#include <iostream>
#include <windows.h>

using namespace std;

char * UtfTo866(const wchar_t * str)
{
    char * buf = new char[wcslen(str) + 1];
    CharToOemW(str, buf);
    return buf;
}

int main()
{
    char * p = UtfTo866(L"Привет мир!\n");
    cout << p << endl;
    delete [] p;

    return 0;
}

```

Ещё один вариант с использованием типа string вместо указателя:

```

#include <iostream>
#include <string>
#include <windows.h>

using namespace std;

string UtfTo866(const wchar_t * str)
{
    char * buf = new char[wcslen(str) + 1];
    string tstr;
    CharToOemW(str, buf);
    tstr = buf;
    delete buf;

    return tstr;
}

int main()

```

```

{
    cout << UtfTo866(L"Привет мир!\n") << endl;

    return 0;
}

```

Или, более правильный способ для языка Си++ с использованием класса:

файл utf8to866.h

```

#ifndef UTFTO866_H
#define UTFTO866_H

#include <iostream>
#include <windows.h>

class UtfTo866
{
    std::string str;
public:
    UtfTo866();
    UtfTo866(const UtfTo866& other);
    virtual ~UtfTo866();
    UtfTo866& operator=(const UtfTo866& other);

    std::string utf8to866(const wchar_t * m_str);
    void setstring(std::string m_str);
    void setutf8(const wchar_t * m_str);
    std::string get866() const;
};

#endif // UTFTO866_H

```

файл utf8to866.cpp

```

#include "utf8to866.h"

UtfTo866::UtfTo866()
{
    //ctor
}

UtfTo866::~~UtfTo866()
{
    //dtor
}

UtfTo866::UtfTo866(const UtfTo866& other): str(other.str)
{
    //ctor copy
}

UtfTo866& UtfTo866::operator=(const UtfTo866& rhs)

```

```

{
    if(this == &rhs) return *this;
    str = rhs.str;

    return *this;
}

std::string UtfTo866::utf8to866(const wchar_t * m_str)
{
    char * std_char = new char[std::wcslen(m_str) + 1];

    CharToOemW(m_str, std_char);

    str = std_char;

    delete std_char;

    return str;
}

void UtfTo866::setstring(std::string m_str)
{
    char * std_char = new char[m_str.length() + 1];
    wchar_t * s = new wchar_t[m_str.length() + 1];
    strcpy(std_char, m_str.c_str());

    s = (wchar_t*)std_char;
    CharToOemW(s, std_char);
    str = std_char;

    delete [] std_char;
    delete [] s;
}

void UtfTo866::setutf8(const wchar_t * m_str)
{
    char * std_char = new char[std::wcslen(m_str) + 1];

    CharToOemW(m_str, std_char);

    str = std_char;

    delete [] std_char;
}

std::string UtfTo866::get866() const
{
    return str;
}

```

файл main.cpp

```
#include "utfto866.h"

using namespace std;

int main()
{
    UtfTo866 m;
    // поэтапное использование класса
    // сначала помещение строки, потом вывод
    m.utf8to866(L"1 - Привет мир!\n"); // помещение строки в класс
    cout << m.get866() << endl; // вывод строки из класса
    // одновременное помещение строки и вывод
    cout << m.utf8to866(L"2 - Привет мир!\n") << endl;

    return 0;
}
```

Способ №3

Следует задать в консоли кодировку, в которой будет выполняться ваша программа.

Задаётся она следующей командой:

chcp 65001

Но прежде, в консоли (командной строке) необходимо поменять шрифт с «Точечные шрифты» на «Lucinda Console». Делается это через свойства командной строки, которое можно открыть, щёлкнув правой кнопкой мыши по заголовку окна «Командная строка».

Для непосредственного выполнения команды `chcp` в вашей программе можно воспользоваться функцией `system()`, например так:

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

int main()
{
    system("chcp 65001");
    printf("Привет мир!\n");
    return 0;
}
```

Пример был приведён для языка Си но для Си++ он также работоспособен.

Способ №4

Последний способ заключается в использовании функций в Си или методов Си++ специально предназначенных для работы с широкими символами. Таких как `wcout` или `wprintf()`.

Прежде чем использовать данные функции, необходимо установить локаль (англ. `locale`) приложения, это можно сделать функцией `setlocale()`.

Чтобы работать с этой функцией, надо подключить заголовочный файл `locale.h`.

```
#include <locale.h>
```

Пример использования объекта `wcout`, `wcin`, и типа `wstring`:

```
#include<iostream>
#include<locale>

using namespace std;

int main()
{
    setlocale(LC_ALL, "ru_RU.UTF-8"); // устанавливаем локаль и
                                     // кодировку

    wchar_t prompt[] = L"Введите имя файла..."; // пример строки
    wstring filename;

    wcout << prompt << endl;
    wcin >> filename;
    wcout << filename << endl;

    return 0;
}
```

Несмотря на большое количество описанных выше способов, работает каждый из них не всегда. В каждой конкретной системе по-разному. В некоторых могут работать и все описанные методы. Поэтому рекомендуется проверить их все, чтобы понять, какой способ работает и лучше всего подходит именно вам.

4 Язык Си

Язык Си появился в начале 70-х годов, как основной язык программирования, разрабатываемой операционной системы UNIX. Главными создателями языка являются сотрудники компании «Bell Laboratories» Кен Томпсон (англ. *Kenneth Thompson*) и Денис Ритчи (англ. *Dennis MacAlistair Ritchie*). Язык является стандартизированным и за свою историю подвергался пересмотру несколько раз. Он продолжает развиваться и сейчас. Появление новых стандартов повлияло на образование нескольких диалектов языка. Существующие на настоящее время диалекты: «K&R» C — 1978 г., ANSI/ISO C (позже C89) — 1989 г., ANSI/ISO C99 — 1999 г.[3]. Далее рассматривается вариант языка стандарта ISO/C99.

Для примера рассмотрим листинг следующей программы.

Содержание главного файла:

```
/* Пример программы на Си стандарта ISO/C99 */
// main.c
// Главный файл программы
#include <stdio.h>
#include <conio.h>
#include "result.h"

int main( void )
{
    int x, y, result;          /* элементы вычисляемого выражения */
    char char_sign = 0;        /* знак операции */
    char char_quit = 0;        /* флаг выхода из программы */

    while ( 'q' != char_quit ){ // если флажок равен "q" то выходим

        /*ввод элементов выражения пользователем */
        printf( "\nВведите переменные и операцию!" );
        printf( "\nx = " );
        scanf( "%d", &x);

        printf( "\ny = " );
        scanf( "%d", &y);

        switch ( char_sign = getch() ) { // ввод знака операции
                                           // пользователем
            case '+' : result = addition( x, y );
            break;

            case '-' : result = subtraction( x, y );
            break;

            case '*' : result = multiplication( x, y );
            break;

            case '/' : result = division( x, y );
            break;

            default :
                printf( "Ошибка! Введённый " /* значение */
                        "операция не определён!"); /* выводимое если */
                                                         /* пользователь ввёл */
                                                         /* неизвестный знак */
        }
        /* вывод решения */
        printf( "\n%d %c %d = %d\n", x, char_sign, y, result);

        /* вопрос пользователю о выходе из программы */
        printf( "\nЕсли хотите выйти то нажмите \"q\"\n" );

        char_quit = getch();
    }

    return 0;
}
```

Содержание второго файла:

```
// result.c
// файл содержит реализацию функций основных математических операций

#include "result.h"

int addition(int x, int y)
{
    return ( x + y );
}

int subtraction(int x, int y)
{
    return ( x - y );
}

int multiplication(int x, int y)
{
    return ( x * y );
}

int division(int x, int y)
{
    return ( x / y );
}
```

Содержание третьего «заголовочного» файла:

```
// result.h
// заголовочный файл содержащий описание функций

#ifndef RESULT_H
#define RESULT_H

/* описание функций (их прототипы)*/
int addition      (int, int);      // сложение
int subtraction   (int, int);      // вычитание
int multiplication(int, int);      // умножение
int division      (int, int);      // деление

#endif
```

Стоит обратить внимание, что исходный текст программы состоит из трёх файлов: главного файла `main.c`, файла реализации функций математических операций (описания функций) `result.c` и заголовочного файла (объявления функций) `result.h`.

На рис. 4.1 изображена структура организации файлов в типичном проекте на языке Си и показаны этапы создания исполняемого модуля, с участием этих файлов.

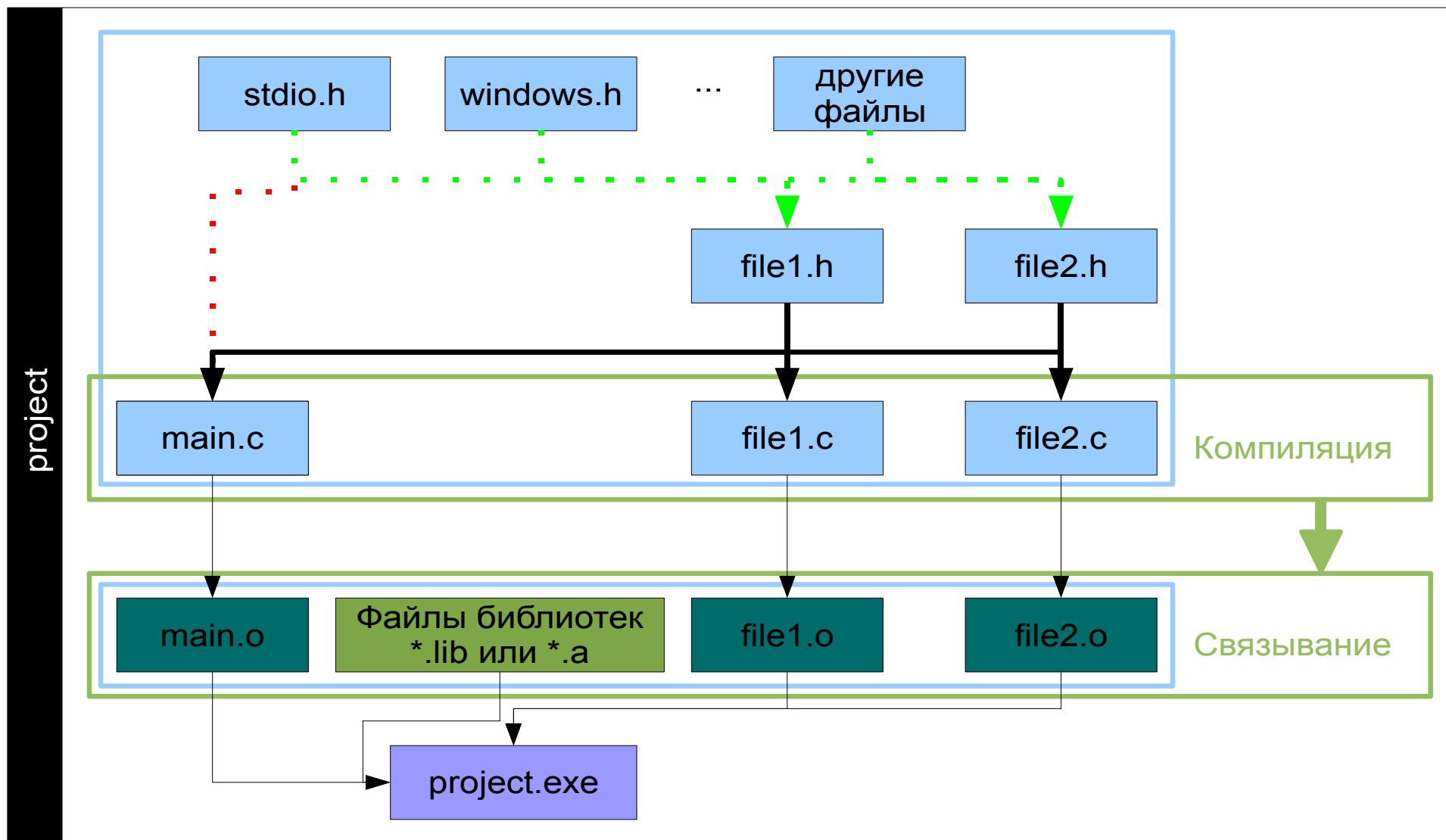


Рис. 4.1: Структура файлов в проекте Си.

4.1 Операторы и операции

В языках программирования часто используются понятия оператор и операция. К сожалению из-за путаницы с терминологией эти понятия часто путают. Путаница усугубляется ещё и тем, что в языке Си/Си++ инкремент/декремент и присваивание являются и операторами и операциями.

- Оператор или инструкция, команда (англ. *statement*) — наименьшая атомная часть языка программирования. При этом обычно программа состоит из последовательности операторов оканчивающихся точкой с запятой.

Примеры операторов:

- операторы описания переменных и других объектов;
- оператор-выражения;
- оператор присвоения;
- оператор условия **if**;
- операторы циклов **while**, **do-while**, **for**;
- оператор варианта **switch**;
- операторы перехода **goto**, **break**, **continue**, **return**;
- блок **{ }**;
- пустой оператор.
- Операция (англ. *operator*) — синтаксическая конструкция, определяющая совершаемое действие внутри некоторых операторов. Операции часто обозначаются различными символами пунктуации и в общем похожи на математические. Английское слово *operator* часто неправильно переводят как «оператор»[1].

Примеры некоторых операций:

- математические операции $x + y$, $x - y$, $x * y$, x / y
- логические операции $x \&\&y$, $x || y$, $x !$, $x == y$, $x < y$, $x > y$
- операции работы с памятью **sizeof(x)**, **new**, **delete**
- операция принадлежности(членства) $a.x$, $a \rightarrow x$
- операция разрешения контекста $a :: x$
- операция индексации $a[i]$
- операция присваивания $x = y$
- операция инкремента/декремента **++**/**--**
- операция вызова функции $f()$
- и. т.

В языке Си++, поскольку запись операций синтаксически очень похожа на запись таковых операций в математике, применяются следующие формы нотации:

1. $+xy$ — префиксная;
2. $x+y$ — инфиксная;
3. $xy+$ — постфиксная.

Подробнее тут [1, 7].

4.2 Директивы препроцессора

Препроцессор – это программа или часть программы, которая выполняет обработку исходного кода до начала компиляции[1].

Таблица 1. Часто используемые директивы и их действие[6].

Директива препроцессора	Выполняемое действие	Примечание
#include <файл> или #include «файл»	Указанный файл вставляется в код вместо директивы	Используется для включения заголовочных файлов, чтобы данный код мог задействовать функциональность, определённую где-то в другом месте
#define ключ значение	Каждое вхождение заданного элемента ключ заменяется заданным элементом значение	В Си используется для определения константного значения или макроопределения. В Си++ используется редко, так как есть const .
#ifdef ключ или #ifndef ключ #endif	Код внутри ifdef-endif или ifndef-endif блоков включается или опускается в зависимости от того, определён ли заданный элемент ключ с помощью директивы #define	В основном используется для защиты от многократных включений кода.

Файлы подключаемые посредством директивы `#include` в языке Си++:

`iostream` — содержит определения потоков ввода/вывода;

`fstream` — содержит определения потоков файлового ввода/вывода;

`string` — содержит определение типа `string` для работы со строками в стиле Си++.

4.2.1 Подробнее про директиву препроцессора **#define**

Директива `#define` определяет идентификатор (ключ) и последовательность символов, которой будет замещаться данный идентификатор при его обнаружении в тексте программы. Идентификатор называется именем макроса, а процесс замещения подстановкой макроса.

Общая форма объявления директивы `#define`:

```
#define имя_макроса (ключ) последовательность_символов
```

Примеры использования директивы:

```
#define TRUE      1
#define FALSE     0

#define PI        3.141592653
```

```
#define ONE      1
#define TWO      ONE + ONE
#define THREE    ONE + TWO
```

Рекомендуется использовать внутри заголовочного файла `*.h` или `*.hpp`, в следующем виде:

```
#ifndef ИМЯ_ФАЙЛА
#define ИМЯ_ФАЙЛА

// ваш код

#endif
```

4.3 Типы данных

Тип данных определяет *какого рода* будет храниться информация в переменной после её объявления. Типы данных бывают простыми (базовые) и сложными (составные, пользовательские).

Простые типы — типы данных, объекты которых *характеризуются* отсутствием доступной программисту внутренней структурой.

Сложные типы — тип данных, объекты которых характеризуются наличием внутренней структуры доступной программисту.

Простые типы данных в языках Си и Си++:

- целые **int**;
 - **unsigned int** диапазон значений от 0 до 65 535;
 - **signed int** диапазон значений от -32 767 до 32 767;
- вещественные **float**;
- вещественные **double**;
 - **long double**;
- символьные **char**;
 - **unsigned char** диапазон значений от 0 до 255;
 - **signed char** диапазон значений от -127 до 127;
- пустой тип **void**.

Составные типы данных в языках Си и Си++:

- массивы (англ. array);
- структуры **struct**;
- объединения **union**;
- перечисления **enum**;
- указатели (англ. pointer).

4.4 Массивы

4.4.1 Одномерный массив

Массив — это неупорядоченный набор данных, который содержит множество значений, относящихся к одному и тому же типу и хранящийся в смежных ячейках памяти. Массив является составным типом данных[1].

Общая форма объявления массива:

```
тип имя_массива[размер_массива];
```

Для доступа к элементам массива используется индекс, заключённый в квадратные скобки. Отсчёт начала массива ведётся с нуля. Например, в массиве `int array[10];` обратится к первому элементу можно через `array[0]`, а к последнему через `array[9]`.

При создании массива обязательно указывать его размер. Исключением является случай, когда массив инициализируется конкретными значениями.

Пример инициализации без указания размера массива:

```
int array[] = {2, 6, 9, 5};
```

4.4.2 Двумерный массив

Двумерный массив — это массив, элементами которого являются другие массивы.

Общая форма объявления двумерного массива:

```
тип имя_массива[размер_внешнего_массива][размеры_внутренних_массивов];
```

Пример инициализации:

```
int array[2][5] = {{1, 2, 3, 4, 5}, {6, 7, 8, 9, 10}};
```

Двумерный массив можно визуализировать как матрицу, где первый индекс это - строка, а второй – столбец. `array[строка][столбец]`

	i \ j	0	1	2	3	4
0	<code>array[0][0]</code>		<code>array[0][1]</code>	<code>array[0][2]</code>	<code>array[0][3]</code>	<code>array[0][4]</code>
1	<code>array[1][0]</code>		<code>array[1][1]</code>	<code>array[1][2]</code>	<code>array[1][3]</code>	<code>array[1][4]</code>

Более правильному представлению соответствует схема (рис. 4.2).

Массив `a[2][5] = {{1, 2, 3, 4, 5}, {6, 7, 8, 9, 10}};`

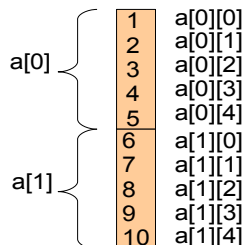


Рис. 4.2. Расположение данных в двумерном массиве.

4.5 Структуры

Структура — составной тип данных, который может хранить элементы более чем одного типа (в отличие от массива), располагая их вместе под одним именем. Структура, тоже тип данных, как и **int**, **char**, **float**, **double**, только в отличие от них, она инкапсулирует данные других типов внутри себя. Такой подход позволяет использовать структуру для хранения разнотипных данных вместе. Например, база данных работников. Для работы со структурой требуется сначала создать её описание, включающее поля данных, необходимых типов. Затем можно создавать переменные (объекты) этого типа данных и использовать их. В языке Си++, при создании переменной, от структурного типа не требуется писать ключевое слово **struct**.

Общая форма объявления структуры:

```
struct имя_типа
{
    список_полей_структуры;
};
```

Пример использования:

```
int main()
{
    struct employee // объявление структуры служащий
    {
        int id;           // поле идентификатора
        char f_name[40]; // поле имени
        char l_name[40]; // поле фамилии
        double salary;   // поле зарплата
    };

    employee empl1 = \ // пример инициализации
        {1, "Пётр", "Петров", 26459.4};
    employee empl;

    // пример записи данных в структуру
    empl.id = 2;
    strcpy(empl.f_name, "Иван");
    strcpy(empl.l_name, "Иванов");
    empl.salary = 25456.45;

    cout << empl1.id << endl;
    cout << empl1.f_name << endl;
    cout << empl1.l_name << endl;
    cout << empl1.salary << endl;
    cout << empl.id << endl;
    cout << empl.f_name << endl;
    cout << empl.l_name << endl;
    cout << empl.salary << endl;

    return 0;
}
```

В данном примере используется инициализация переменной `empl1` при её создании.

Для доступа к элементам структуры используется операция членства точка (`.`).

4.6 Объединения

Объединение — это формат данных, который может хранить разные типы данных в пределах одной области памяти, но в каждую единицу времени только один из них. То есть, в то время, как структура может содержать, скажем **int** и **long** и **double**, то объединение может содержать **int** или **long** или **double**[1].

Общая форма объявления объединения:

```
union имя_типа
{
    список_объявлений_переменных;
};
```

Пример использования*:

```
int main()
{
    union uni // создаём объединение uni
    {
        int    int_val; // создание целочисленного имени переменной
        long   long_val; // создание вещественного имени переменной
        double double_val; // создание двойного вещественного имени переменной
    };

    uni uni_val; // создание переменной uni_val
    uni_val.int_val = 1005; // присвоение переменной uni_val целого числа 1005
    cout << uni_val.int_val << endl;
    uni_val.long_val = 9223372036854775807; // присвоение переменной uni_val
                                         // целого 64 разрядного числа 9223372036854775807
    cout << uni_val.long_val << endl;
    uni_val.double_val = 454.6529; // присвоение переменной uni_val
                                   // двойного вещественного числа 454.6529
    cout << uni_val.double_val << endl;

    return 0;
}
```

Здесь видно, что переменная `uni_val` становится то типом **int**, то **long**, то **double**. Контекст типа переменной, с которой необходимо работать в данный момент, определяется именем поля, указываемым после точки (операции принадлежности).

* - на 32 разрядной целевой машине (Microsoft® Windows XP®/Vista™ или Linux) размер типа **long** равен 32 бита (4 байта), а на 64 разрядной 64 бита (8 байт). Размеры целочисленных типов данных, вообще очень сильно зависят от системы (процессор, ОС, системные библиотеки), в которой они используются.

4.7 Перечисления

Перечисление — это тип данных, позволяющий создавать упорядоченный набор символьных константы. Тип данных является очень ограниченным в своих возможностях. В основном используется для повышения читабельности кода программы, путём устранения константных чисел (магических чисел), идентифицирующих определённые значения. Для этого типа определена только операция присваивания.

Общая форма объявления перечисления:

```
enum имя_перечисления {список_символических_констант(перечислители)};
```

Общая форма объявления перечисления как типа данных:

```
имя_перечисления имя_переменной;
```

Пример использования:

```
int main()
{
    enum colors {red, blue, orange, green, black, white, yellow, brown};
    cout << red << blue << orange << green
         << black << white << yellow << brown << endl;
    colors clr; // создание переменной clr
    clr = white; // присваивание ей константы white

    return 0;
}
```

Присвоить переменной `clr` можно только значение констант, записанных в перечислении `colors`.

4.8 Указатели

Указатель — специальный тип данных, который может хранить адрес, как значение[1]. Имя указателя — это адрес в памяти компьютера.

Для работы с указателями применяют символы (&) и (*):

- **&** - операция взятия адреса. Её используют, чтобы узнать, по какому адресу в оперативной памяти компьютера расположена переменная.
- ***** - операция разыменования. Её используют **только с переменными указателями** для получения значения, хранящегося в оперативной памяти по данному адресу.

Пример использования операции (&):

```
int main()
{
    long x = 5;

    cout << x << endl;      // печать содержимого переменной x
    cout << &x << endl;    // печать адреса переменной x

    return 0;
}
```

Пример использования операция (*):

```
int main()
{
    int y = 5;           // 1
    int* x = &y;        // 2
    // или так
    // int* x = 0; // указатель обязательно должен куда то ссылаться! Если
    // заранее не известно куда он должен указывать то его инициализируют 0
    // x = &y;
    //

    cout << *x << endl;   // печать содержимого переменной x
    cout << x << endl;    // печать адреса переменной x

    return 0;
}
```

Пояснение к строке 1 и 2.

Поскольку **x** - это указатель(переменная содержащая адрес), то ей необходимо назначить адрес переменной или присвоить нуль. Если в данный момент нет той переменной, у которой требуется взять адрес, то присваивается нуль. В нашем примере, создадим сначала переменную **y** со значением 5. Потом создадим указатель **x**, и с помощью операции (&) возьмём у переменной **y** её адрес, который присвоим указателю **x**.

При работе с указателем, если требуется присвоить или считать адрес, который он хранит, то **достаточно указать его имя**.

Если требуется присвоить или считать данные, находящиеся по этому адресу, то **надо использовать операцию разыменования (*)**.

Схематическое представление работы с оперативной памятью

Переменную можно представить на схеме, как набор полей, описывающих её сущность. Верхнее поле показывает данные переменной, нижнее адрес переменной в памяти ЭВМ и под адресом ставится имя переменной ассоциированное с этим адресом.

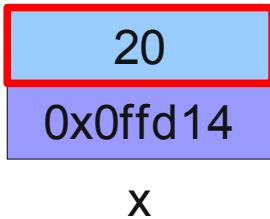
При этом красным выделяется доступное в данный момент поле.



Создание обычной (автоматической) переменной **x**

```
int x = 20;
```

Примерное схематическое представление размещения переменной **x** в памяти.



Получение адреса переменной **x**

Для получения адреса переменной **x** используется **&**.

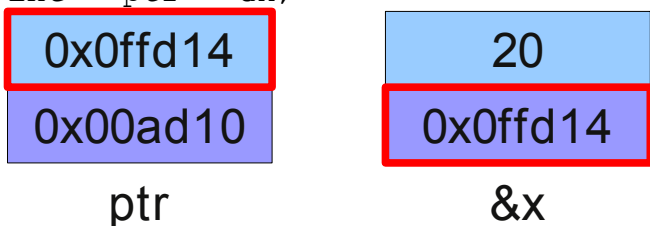


Создание указателя **ptr** на переменную **x**

```
int x = 20;
```

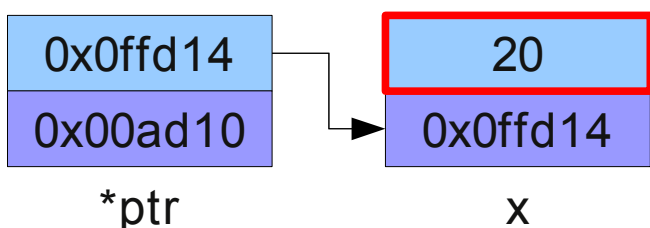
Создаём указатель **ptr** и инициализируем его передав адрес переменной **x**.

```
int * ptr = &x;
```



Вызов, обращение к переменной **x** через указатель **ptr**

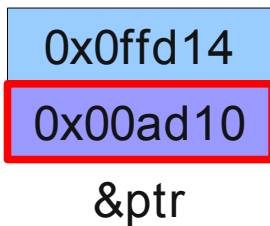
Теперь, чтобы обратиться к переменной **x** через указатель **ptr**, надо добавить **(*)** к **ptr**.



Звёздочка **(*)** - позволяет перейти к содержимому, находящемуся по адресу, хранимому в **ptr** (переход показан стрелкой).

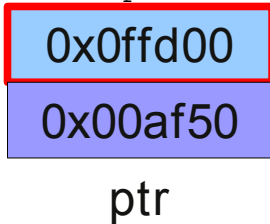
Получение адреса самого указателя `ptr`

Также может понадобиться получить адрес самого указателя для этого используется снова `(&)` к `ptr`.



Прямое присвоение адреса указателю `ptr`

`int * ptr = (int*)0x0ffd00.`



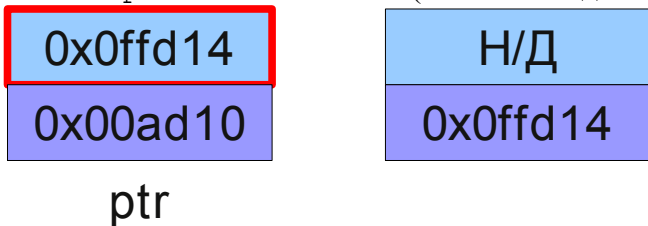
Теперь вариант динамического размещения переменной в языке Си++

Создание динамической переменной и передача её адреса указателю `ptr` без инициализации значением.

Создаём указатель `ptr` и инициализируем его передав адрес динамической переменной созданной `malloc()` или `new` (подробнее про `new` и `delete` см. в разделе 5.2.12) для Си++.

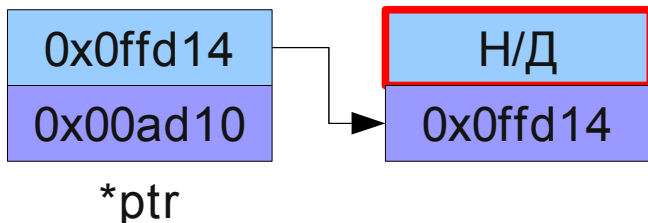
`int * ptr = (int *) malloc(sizeof(int));` (для языка Си)

`int * ptr = new int;` (тоже самое для языка Си++)



Вызов, обращение к динамической переменной через указатель `ptr`

Чтобы обратиться через указатель `ptr`, надо добавить `(*)` к `ptr`.

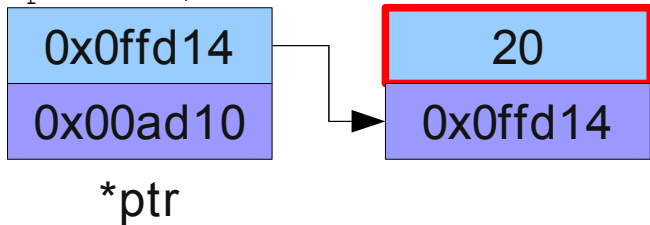


Звёздочка `(*)` - позволяет перейти к содержимому, находящемуся по адресу, хранимому в `ptr` (переход показан стрелкой).

Вызов, обращение к динамической переменной через указатель `ptr` и присвоение ей значения 20

Теперь, чтобы присвоить значение надо также добавить **(*)** к `ptr`.

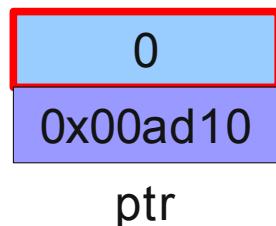
```
*ptr = 20;
```



Удаление динамической переменной через указатель `ptr`

Чтобы удалить данные через указатель `ptr` надо воспользоваться функцией `free()` для Си или операцией **`delete`** для Си++.

```
free(ptr);  
delete ptr;
```



При этом указатель `ptr` можно продолжать использовать дальше и передать ему адрес новой динамической переменной.

4.8.1 Использование указателей при работе со структурой

При использовании указателя на структуру невозможно применить операцию точка, так как указатель на структуру содержит адрес структуры, но сам не является структурой. Для работы с элементами структуры необходимо использовать другой синтаксис.

Пример доступа к элементам структуры через указатель:

```
struct A // создаём структуру
{
    int x; // список переменных членов структуры
    int y;
};

int main()
{
    A a;           // создание переменной "a" по образцу структуры A
    A *ptr_a = &a; // создание указателя на переменную "a"
```

```

a.x = 10;      // присвоение значений членам x и y переменной "a"
a.y = 20;

(*ptr_a).x = 15; // присвоение новых значений x и y
(*ptr_a).y = 25;

printf("%d", a.x); // вывод содержимого членов x и y переменной "a"
printf("%d", a.y); // изменённого при доступе через указатель "ptr_a"

return 0;
}

```

Поскольку `ptr_a` — указатель, то обратиться просто через точку мы не можем. По правилам работы с указателями, чтобы получить доступ к значению, а не адресу, надо применить к переменной `ptr_a` операцию разыменования. Отсюда получаем `*ptr_a.x = 15;`. Но операция разыменования имеет меньший приоритет, чем операция членства, следовательно она будет выполнена последней. Следовательно, для доступа к членам переменной `ptr_a` необходимо повысить приоритет операции разыменования. Получаем синтаксис следующего вида `(*ptr_a).x = 15;`.

Есть способ доступа к элементам структуры синтаксически проще, используя операцию указателя на членство (`->`) — стрелка.

Пример использования операции стрелка:

```

struct A // создаём структуру
{
    int x; // список переменных членов структуры
    int y;
};

int main()
{
    A a;          // создание переменной "a" по образцу структуры
    A *ptr_a = &a; // создание указателя на переменную "a"

    a.x = 10;      // присвоение значений членам x и y переменной "a"
    a.y = 20;

    ptr_a->x = 15; // присвоение новых значений x и y
    ptr_a->y = 25;

    printf("%d", a.x); // вывод содержимого членов x и y переменной "a"
    printf("%d", a.y); // изменённого при доступе через указатель "ptr_a"

    return 0;
}

```

В приведённом примере показан, менее нагруженный синтаксис доступа к членам структуры.

4.9 Функции

Функция (в программировании) — подпрограмма (блок, подсистема), на вход которой поступают данные в виде значений аргументов (оказывается управляющие воздействие), а в выход выдаётся (возвращается) результат её работы (выполнения подпрограммы).

Общий вид 3-х функций:

```

область_глобальных_объявлений

```



```

тип_возвращаемого_значения main( список_аргументов(параметров) )
{
    последовательность_операторов
}

тип_возвращаемого_значения function_1( список_аргументов(параметров) )
{
    последовательность_операторов
}

тип_возвращаемого_значения function_2( список_аргументов(параметров) )
{
    последовательность_операторов
}

```

Общий вид описания функции

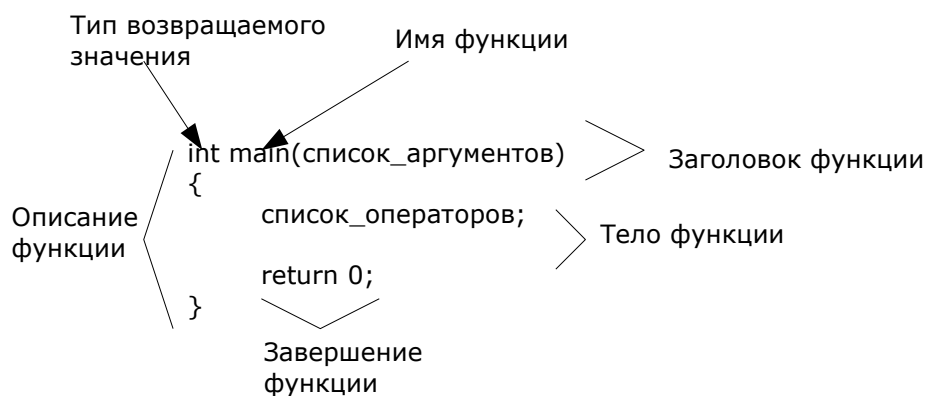


Рис. 4.3. Схема функции на примере функции *main()*.

Описание (определение) функции — фактическая реализация функции. Описание состоит из двух частей: заголовка и тела функции.

1. **Заголовок функции** — описание интерфейса функции с остальной частью программы. Заголовок состоит из трёх частей: типа, возвращаемого значения, имени функции и списка аргументов (параметров).
 - а) **Тип возвращаемого значения** — тип данных, выдаваемый в место (точку) вызова функции оператором **return**.
 - б) **Имя функции** — собственное имя функции, используемое для её однозначной идентификации в программе.
 - с) **Список аргументов (параметров)** — перечисление переменных и их инициализация значениями, получаемыми в месте вызова функции. Переменная для приёма значения называется формальной, а для передачи действительной. Принято аргументом называть действительный аргумент (параметр), а параметром формальный параметр (аргумент).
2. **Тело функции** — описание инструкций компилятору, которыми описывается действия данной функции. Тело состоит из двух частей: списка операторов и завершения функции.
 - а) **Список операторов** — последовательность операторов. **оператор** — любая полная инструкция (команда) компилятору. В языке Си/Си++ каждый оператор должен завершаться символом точки с запятой.
 - б) **Завершение функции** — оператор возврата, завершающий функцию, в случае,

если определено, что функция должна вернуть какое то значение (в случае с функцией `main()` должно быть возвращено целое число), в противном случае в месте, где указывается тип возвращаемого значения, должен стоять тип **void**.

4.9.1 Прототипы функции

Прототип описывает интерфейс функции для компилятора. То есть, он сообщает компилятору, каков тип возвращаемого значения, если он есть у функции, а также количество и типы аргументов данной функции.

Программисту прототипы необходимы для снижения вероятности ошибки. Они помогают обеспечить следующее:

1. Компилятор корректно обрабатывает возвращаемое значение.
2. Компилятор проверяет, указано ли правильное количество аргументов.
3. Компилятор проверяет правильность типов аргументов. Если тип не подходит, компилятор преобразует его в правильный, когда это возможно.

Общая форма объявления прототипа:

```
тип_возвращаемого_значения имя_функции( список_аргументов(параметров) );
```

Пример прототипа функции:

```
void function();
```

или

```
int func( float x );
```

Прототипы рекомендуется включать в заголовочные файлы.

Три шага для создания функции в Си++:

1. написать определение функции;
2. написать прототип функции;
3. вызвать функцию.

4.9.2 Функция не принимающая аргументов и не возвращающая значений (тип данных void)

Функция, не принимающая аргументов, записывается с пустыми скобками в её заголовке. Функция, не возвращающая значений, в заголовке определяется типом **void**.

Тип **void** – пустой тип данных, сообщающий компилятору на отсутствие возвращаемого функцией значения. **Не работает с оператором return!**

Пример:

```
void function()
{
    // какой-то код...
}
```

Также с помощью типа **void** можно создавать указатель неопределённого типа.

Пример:

```
void * ptr = 0;
```

4.9.3 Передача аргументов (параметров) в функцию по значению

При передаче по значению все изменения, которым подвергается переменная внутри функции, не влияют на переменную в месте вызова этой функции.

Пример функции, принимающей один параметр `x`, целого типа по значению:

```
void funct( int x ) // в функцию передаётся (копируется) значение переменной z
{
    x = 10;        // значение z не меняется так как x и z разные переменные
}
```

Вызывать её можно следующими способами:

```
int main()
{
    int z = 1;

    funct( z );        // вызываем функцию foo и передаём параметр z
    funct( z + 1 );    // значение x равно 10 а z равно 1
    funct( 1000 );     // значение x попрежнему равно 10 а z равно 1

    return 0;
}
```

4.9.4 Передача аргументов (параметров) в функцию через указатель

Изменить значение в функции можно передав не саму переменную, а указатель на неё. Такой способ передачи называется передачей через указатель.

Пример функции, принимающей один параметр `x`, целого типа через указатель:

```
void funct( int *x ) // инициализируется указатель на участок памяти z
{
    *x = 10;         // теперь можно изменить значение z поскольку,
                    // x это указатель на z
}
```

Вызывать её можно следующим способом:

```
int main()
{
    int z = 1;
    funct( &z );     // вызывается функция, в которую передаётся адрес z
    /* остальные варианты больше не имеют смысла */
    /* z теперь равно 10 */

    return 0;
}
```

4.9.5 Передача массива в функцию как параметра

Примеры передачи массива, как параметра в функцию:

```
void foo_array_1( int * array ) // через указатель
{
```

```

        for (register int i = 0; i < 3; i++)
            cout << array[i] << "\n";
    }

void foo_array_2( int array[] ) // как массив неопределённой длины
{
    for (register int i = 0; i < 3; i++)
        cout << array[i] << "\n";
}
// передача двумерного массива (матрицы)
void foo_array_3( int array[][5] )
{
    for ( register int i = 0; i < 2; i++ ) {
        for (register int j = 0; j < 5; j++)
            cout << array[i][j] << " ";
        cout << "\n";
    }
}

```

Вызывать эти функции можно следующими способами:

```

int main()
{
    int array[] = { 1, 2, 3 };
    int array_3[2][5] = { {11, 12, 13, 14, 15},
                          {21, 22, 23, 24, 25} };

    foo_array_1( array );
    foo_array_2( array );
    foo_array_3( array_3 );

    return 0;
}

```

Стоит отметить, что для передачи массива в функцию, как аргумента, следующие объявления:

```

тип_данных имя_массива[]; // одномерный массив
тип_данных (*имя_массива)[]; // двумерный массив

```

можно записать как:

```

тип_данных *имя_массива; // одномерный массив
тип_данных имя_массива[][размер_вложенного_массива]; // двумерный массив

```

то есть, они эквивалентны.

4.9.6 Аргументы функции `main()` и возвращаемое значение

Аргумент функции — значение (число, указатель и т. д.), передаваемое функции, а также символьное имя (название переменной) в тексте программы, выступающее в качестве идентификатора этого значения. Почти синонимичным этому понятию является термин — **параметр**. Слово **аргумент** обычно используется во внешнем, по отношению к функции контексту[4].

Пример функции `main()` с двумя аргументами:

```

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

```

```
{
    //..... какой то код
    return 0;
}
```

int argc — этот параметр содержит количество аргументов в командной строке.

int *argv[] — это параметр — указатель на массив указателей, содержащих аргументы командной строки.

Пример использования:

```
int main(int argc, char *argv[])
{
    printf("%d", argc);
    printf("%s", argv[1]);

    return 0;
}
```

Запустить программу можно следующим образом:

C:\app.exe help copy

Результат работы программы:

2
help

Функция **main()** также возвращает значение, используя оператор **return**. Оператор **return** пишется в конце функции, указывая операционной системе, каким образом завершилась программа. В общем случае по стандарту ISO **main()** должна возвращать в случае корректного завершения **0** и в случае не корректного (ошибки) **1**.

Как вариант можно использовать следующие определения:

- Успех — **EXIT_SUCCESS**, что по сути является нулём.
- Ошибка — **EXIT_FAILURE**, соответственно является единицей.

Пример использования:

```
int main()
{
    int * ptr;
    ptr = malloc(1000); // выделение 1000 байт
    if (!ptr) {
        printf("Нехватает памяти!");
        return 1; // или return EXIT_FAILURE;
    }

    return 0; // или return EXIT_SUCCESS;
}
```

5 Основы Си++, система ввода/вывода, типы данных, функции и другие возможности

Язык Си++ компилируемый строго типизированный язык программирования общего назначения. Поддерживает разные парадигмы программирования: процедурную, обобщённую, функциональную; наибольшее внимание уделено поддержке объектно-ориентированного программирования[4].

В 1990-х годах язык стал одним из наиболее широко применяемых языков программирования общего назначения[4].

При создании Си++ стремились сохранить совместимость с языком Си. Большинство программ на Си будут исправно работать и с компилятором Си++. Си++ имеет синтаксис, основанный на синтаксисе Си[4].

Язык возник в начале 1980-х годов, когда сотрудник фирмы «Bell Laboratories» Бьёрн Страуструп (англ. Bjarne Stroustrup) придумал ряд усовершенствований к языку Си под собственные нужды. До начала официальной стандартизации язык развивался в основном силами Страуструпа в ответ на запросы программистского сообщества. В 1998 году был ратифицирован международный стандарт языка Си++: ISO/IEC 14882:1998 «Standard for the C++ Programming Language»; после принятия технических исправлений к стандарту в 2003 году нынешняя версия этого стандарта — ISO/IEC 14882:2003[4].

Название «Си++» происходит от Си, в котором унарный оператор ++ обозначает приращение[4].

5.1 Основы Си++

В Си++ существует собственная система ввода/вывода, пример её использования представлен в листинге в следующей программе. Программа состоит из трёх файлов: главного, файла реализации функций математических операций и заголовочного файла.

Содержание первого файла:

```
// main.cpp
// Главный файл программы

#include <iostream> // реализация стандартного потока ввода/вывода
#include <conio.h> // реализация вызова функции getch()
#include "result.h"

using namespace std; // так же необходимо для использования потоков ввода/вывода

int main(int argc, char* argv[])
{
    char char_quit = 0; // флажок выхода из программы

    while ('q' != char_quit) { // если флажок равен "q" то выходим
        int x, y, result; // элементы вычисляемого выражения

        /*ввод элементов выражения пользователем */
        cout << "\nВведите переменные и операцию!" << endl;
        cout << "\nx = ";
        cin >> x;
        cout << "\ny = ";
        cin >> y;

        char char_sign = 0; // знак операции

        switch (char_sign = getch()) { // ввод знака операции
            // пользователем

            case '+' : result = addition( x, y );
            break;
```

```

        case '-' : result = subtraction( x, y );
        break;

        case '*' : result = multiplication( x, y );
        break;

        case '/' : result = division( x, y );
        break;

        default :
            cerr << "Ошибка! Введённый"           // Сообщение
                 << "операция не определён!" << endl; // выводимое если
                                                         // пользователь ввёл
                                                         // неизвестный знак операции
    }

    cout << "\n" << x << " " << char_sign // вывод решения
         << " " << y << " = " << " "
         << result << endl;

    /* вопрос пользователю о выходе из программы */
    cout << "\nЕсли хотите выйти то нажмите \"q\" " << endl;

    char_quit = getche();
}

return 0;
}

```

Содержание второго файла:

```

// result.cpp
// файл содержит реализацию функций основных математических операций

#include "result.h"

inline int addition(int& x, int& y)
{
    return ( x + y );
}

inline int subtraction(int& x, int& y)
{
    return ( x - y );
}

inline int multiplication(int& x, int& y)
{
    return ( x * y );
}

inline int division(int& x, int& y)
{
    return ( x / y );
}

```

Содержание третьего «заголовочного» файла:

```

// result.hpp
// заголовочный файл содержащий описание функций

#ifndef RESULT_HPP
#define RESULT_HPP

/* описание функций (их прототипы) */
inline int addition      (int&, int&); // сложение
inline int subtraction   (int&, int&); // вычитание
inline int multiplication(int&, int&); // умножение
inline int division      (int&, int&); // деление

#endif

```

5.2 Типы данных, ввод/вывод и функции в Си++

В языке появились новые типы данных, а также подверглись изменениям некоторые взятые из Си.

Простые типы данных:

wchar_t — появился

логические **bool** — появился

Составные типы данных:

строки (англ. *string*) — появились

структуры **struct** — изменились

классы **class** — появились

ссылки — появились

5.2.1 Потоки ввода/вывода

В представленном примере стоит обратить внимание на систему ввода/вывода, используемую в Си++, а именно на используемые потоки `cout` и `cin` вместо функций `printf()` и `scanf()` соответственно. Сам *поток* — это абстракция, помогающая унифицировать процесс чтения и записи данных из различных объектов данных, в том числе разной физической природы в приложение.

Чтобы использовать потоки ввода/вывода, в программе должен быть включён заголовочный файл `iostream`.

```
#include <iostream>
```

И объявлено пространство имён в котором они описаны (про пространство имён см. в разделе 4.3).

```
using namespace std;
```

Ниже перечислены основные потоки ввода/вывода языка Си++.

`cout` (читается как «си аут») – вывести (распечатать) данные на пользовательскую консоль.

`cin` (читается как «си ин») – получить данные с устройства ввода (клавиатуры).

`cerr` (читается как «си эрр») – вывести данные на консоль ошибок (не буферизируется).

`clog` (читается как «си лог») – также вывести на консоль ошибок (буферизируется).

Операции (`<<`) и (`>>`) означают соответственно поместить в поток и взять из потока.

`endl` – манипулятор вывода, служит для вывода на печать всего, что есть в буфере потока вывода (выталкивание содержимого буфера), и перевода на новую строку. Как альтернатива можно рассматривать символ `\n`.

Простое выталкивание содержимого буфера осуществляет манипулятор `flush`.

```
cout << flush;
```

Ещё существуют объекты для ввода/вывода расширенных (англ. *wide*) символов. Эти объекты используются точно также как и представленные выше, только начинаются с буквы `w`.

Например: `wcout`, `wcin`, `wcerr` и т. д.

Строки которые в них отправляются должны начинаться с символа `L` и иметь тип `wchar_t`.

Например: `wchar_t str = L"Привет мир!";`.

Общая форма использования ввода/вывода:

```
cout << "text" << i << " " << "\n";
cin >> j;
```

Пример использования ввода/вывода:

```
int main()
{
    using namespace std;
    cout << "Введите числа: ";
    int sum = 0;
    int input;
    while (cin >> input)
    {
        sum += input;
    }
    cout << "Последнее введенное значение = " << input << endl;
    cout << "Сумма = " << sum << endl;

    return 0;
}
```

5.2.2 Строки

Стандарт ISO/ANSI Си++ расширил библиотеку Си++, добавив класс `string`. Поэтому, отныне, вместо применения символьных массивов для хранения строк, можно применять переменные типа `string`. Класс `string` проще в использовании, чем массив, и к тому же предлагает более естественное представление строки, как типа[1].

Для работы с расширенными строками используется тип `wstring`.

Чтобы использовать класс `string`, в программе должен быть включён заголовочный файл **string**.

```
#include <string>
```

И объявлено пространство имён в котором они описаны(про пространство имён см. в разделе 4.3).

```
using namespace std;
```

Общий вид объявления строки следующий:

```
string имя_переменной;
```

Примеры поддерживаемых типом, `string`, способов работы со строками:

```
string a = "12345", b = "6789", c, d = "654321";
// копирование
c = a;
// конкатенация (сцепление)
c = a + b;
```

```
// добавление строки к существующей строке
с += d;
// сравнение строк
if (a == b) cout << "a равно b!" << endl;
else cout << "a не равно b!" << endl;
или
if (a > b) cout << "a больше b!" << endl; // сравнение по коду символа в ANSI
else cout << "a меньше b!" << endl;
```

Пример простого использования:

```
int main()
{
    string s1 = "penguin";
    string s2, s3;
    cout << "Вы можете присвоить один объект другому: s2 = s1\n";
    s2 = s1;
    cout << "s1: " << s1 << ", s2: " << s2 << endl;
    cout << "Вы можете присвоить объекту string строку в стиле C. \n";
    cout << "s2 = \"buzzard\"\n";
    s2 = "buzzard";
    cout << "s2: " << s2 << endl;
    cout << "Вы можете сцеплять строки: s3 = s1 + s2\n";
    s3 = s1 + s2;
    cout << "s3: " << s3 << endl;
    cout << "Вы можете добавлять строки. \n";
    s1 += s2;
    cout << "s1 += s2 yields s1 = " << s1 << endl;
    s2 += " for a day";
    cout << "s2 += \" for a day\" yields s2 = " << s2 << endl;

    return 0;
}
```

5.2.3 Файловый ввод/вывод

В языке Си++, так же как и в языке Си, имеется система файлового ввода/вывода. Но эта система, отлична от системы языка Си, и использует объектно-ориентированные методы ввода/вывода.

Чтобы работать с файловым вводом/выводом, надо подключить заголовочный файл **fstream**.

```
#include <fstream>
```

И объявлено пространство имён в котором они описаны (про пространство имён см. в разделе 5.3).

```
using namespace std;
```

Для работы с файловым вводом/выводом на запись, необходимо сделать следующие действия:

1. создать объект типа `ofstream` для управления выходным потоком;
2. ассоциировать этот объект с конкретным файлом;
3. использовать объект тем же способом, как используется `cout`. Единственное отличие в том, что вывод направляется в файл вместо экрана;
4. после окончания работы с ним закрыть файл методом `close()`.

Для работы с файловым вводом/выводом на чтение, необходимо сделать следующие действия:

1. создать объект типа `ifstream` для управления выходным потоком;
2. ассоциировать этот объект с конкретным файлом;
3. использовать объект тем же способом, как используется `cin`.
4. после окончания работы с ним закрыть файл методом `close()`.

Шаг 1. Запись в файл. Создание объекта типа `ofstream`:

```
ofstream outFile;
```

Шаг 2. Ассоциирование его с файлом:

```
outFile.open("text.txt");
```

Два предыдущих шага вместе:

```
ofstream outFile("text.txt");
```

Шаг 3. Пример записи в файловый поток:

```
outFile << "Строка";
```

Для закрытия потока:

```
outFile.close();
```

Чтения из файла происходит аналогично кроме шага 3.

Пример использования файлового ввода/вывода в языке:

```
int main()
{
    string filename;
    cout << "Введите имя нового файла: ";
    cin >> filename;
    // создать объект выходного потока для нового файла и назвать его fout
    ofstream fout(filename.c_str());
    fout << "Только для ваших глаз!\n";           // писать в файл
```

```
cout << "Введите секретное число: ";           // писать на экран
float secret;
cin >> secret;
fout << "Ваше секретное число " << secret << endl;
fout.close(); // закрыть файл
// создать объект входного потока для нового файла и назвать его fin
ifstream fin(filename.c_str());
cout << "Вот содержимое " << filename << ":\n";
char ch;
while (fin.get(ch)) // читать символы из файла
    cout << ch; // и писать их на экран
cout << "Готово.\n";
fin.close();
return 0;
}
```

ещё пример чтения информации из потока

```
while ( !inFile.eof() ) { // цикл работает пока не будет достигнут конец файла
    getline(inFile, str); // чтение строки
    length = str.find_first_of(";"); // поиск точки с запятой в строке
    MyString = str.substr(lbegin, length); // копирование подстроки от lbegin
                                         // до length
    str.erase(lbegin, length + 1); // удаление той подстроки из str,
                                   // которую мы скопировали в MyString
    inFile.setstate(ios_base::eofbit); // установка указания
                                     // на окончание чтения
}
```

Запись в файл

```
outFile << "Привет мир!" << endl; // для записи достаточно операции поместить в
                                     // поток
```

5.2.4 Булевый (логический) тип данных

Булевый (логический) тип данных — в информатике является примитивным типом данных, имеющим два возможных значения, иногда называемых **true** (на русском «истина») и **false** (на русском «ложь»). В языке C++ этот тип данных определяется как **bool**, и может принимать соответствующие значения **true** или **false**.

Общая форма объявления:

```
bool имя_переменной;
```

Пример использования:

```
bool a = false;
```

5.2.5 Новый синтаксис инициализации переменной

Из языка Си в Си++ пришёл синтаксис инициализации переменных вида:

```
int n = 10;
```

Но в языке Си++ используется ещё один синтаксис инициализации переменных вида:

```
int n(10);
```

Этот новый синтаксис не совместим с языком Си.

5.2.6 Встраиваемые (встроенные, подставляемые, inline) функции (методы)

Так же в программе используются встраиваемые функции. Встраиваемые функции являются усовершенствованием языка Си++ (есть и в ANSI/ISO C99), предназначенным для ускорения работы программы. Их использование позволяет повысить скорость работы программы, но увеличивает размер исполняемого модуля. Это происходит потому что компилятор подставляет код такой функции в точку её вызова, вместо генерации кода вызова (при вызове функции программа сохраняет адрес команды, следующей сразу после вызова функции, копирует аргументы функции в стек, переходит к ячейке памяти, обозначающей начало функции, выполняет код функции, а затем переходит к команде, адрес которого сохранён).

Общий вид встраиваемой функции (inline) следующий:

```
inline тип_возвращаемого_значения имя_функции( список_аргументов )
{
```

```
// ваш код;
}
```

Пример того, что сделает компилятор с программой.

Что напишите вы:

```
inline int multiplication(int m, int n)
{
    return m * n;
}

int main(int argc, char* argv[])
{
    int x = 2, y = 4;
    cout << multiplication( x, y );

    return 0;
}
```

К какому виду преобразует эту программу компилятор:

```
int main(int argc, char* argv[])
{
    int x = 2, y = 4;
    cout << x * y;

    return 0;
}
```

Другим вариантом встраиваемой функции, который используется в структурах, объединениях и классах (тема 6.1) без ключевого слова **inline**, является следующий вариант:

```
struct имя_структуры
{
    тип_возвращаемого_значения имя_функции ( список_аргументов )
    { /* ваш код функции */ }
};
```

Обратить внимание в этом надо на то, что тело функции находится вместе с её объявлением.

Пример использования:

```
int main()
{
    union A
    {
        int a;
        int b;
        int f() // встраивая функция(inline)
        {
            cout << "test" << endl;
            b = 8;
            cout << a << endl;
        }
    };

    A uni_ob; // создание переменной uni_ob типа объединение A

    uni_ob.a = 5; // работа с переменной
```

```

uni_ob.f(); // вызов встраиваемой функции

return 0;
}

```

Пример демонстрирует не только возможность использования встраиваемой функции, но использование возможности создавать функции внутри объединения. Создать функцию возможно также и в структуре. Создание функций внутри объединений и структур допускается только в Си++.

5.2.7 Аргументы функции, определяемые по умолчанию

Аргументы, определяемые по умолчанию — это значения, которые используются автоматически, если соответствующие фактические параметры в обращении к функции опущены[1]. Аргументы по умолчанию определяются только в прототипе функции. Поскольку компилятор использует прототип, чтобы узнать, сколько аргументов имеет функция, его также можно использовать для сообщения компилятору о возможном использовании аргументов, заданных по умолчанию.

Общая форма объявления:

```

тип_возвращаемого_значения имя_функции(тип_данных имя_переменной =
значение_присваиваемое_по_умолчанию);

```

Пример использования:

```

int left(int a, int b = 1, int c = 5010); // прототип

int main()
{
    int result = 0;

    result = left(5, 0, 0); // обычный вызов функции переопределяет аргументы по
                           // умолчанию
    cout << result << endl;

    result = left(5);      // вызов функции с использованием аргументов по
                           // умолчанию
    cout << result << endl;

    return 0;
}

int left(int a, int b, int c)
{
    return a + b + c;
}

```

Для функции со списком аргументов, принимаемые по умолчанию значения, присваиваются в направлении справа налево. Иначе говоря, нельзя присвоить значение по умолчанию некоторому аргументу, пока не будут присвоены значения для всех остальных аргументов, размещённых справа от него. Фактические значения присваиваются соответствующим формальным аргументам, также в направлении слева направо. Пропуск аргументов не допускается.

5.2.8 Ссылки

Ссылка — это неявный указатель служащий в качестве псевдонима переменной, для которой он был проинициализирован.

Ссылки в основном используют для передачи параметров функциям. Ссылку в общем случае можно понимать, как альтернативное имя для переменной.

Пример ссылки:

```
int x = 10; // здесь мы создали переменную x и инициализировали её значением 10
int &y = x; // а здесь создали ссылку y на переменную x
```

Теперь мы можем работать с переменной **y** как с переменной **x**.

По сути ссылка — это указатель, который:

- обязательно при создании инициализировать каким-то значением;
- после этого нельзя изменять.

Общая форма объявления:

```
тип_ссылки & имя_ссылки = имя_переменной; // имя_переменной — это имя
переменной, которой инициализируется ссылка, и на которую она будет указывать.
```

При выборе между ссылкой и указателем, ссылку рекомендуется использовать везде, где это возможно, во всех остальных случаях указатель.

Пример ошибки в программе, допущенной программистом, при использовании указателя:

```
int x;
int *px = &x;
px = 0;
*px++; // Ошибка!
```

При использовании ссылки такой ошибки не произойдёт!

5.2.9 Передача аргумента в функцию через ссылку

Передача аргумента в функцию через ссылку имеет преимущество перед указателем, поскольку повышает читаемость программы, позволяет не заботиться о защите от непроизвольного изменения ссылки.

Пример использования ссылки для модификации переменной x:

```
void foo( int & z ) // инициализация ссылки адресом памяти переменной z
{
    z = 10; // значение z также можно менять
}

int main(int argc, char * argv[]) {
    int x = 1;

    foo( x );
    /* x теперь равно 10 */

    return 0;
}
```

Стоит обратить внимание на то, что здесь **int& x** это не операция взятие адреса, а инициализация ссылки!

Работа с ссылкой на структуру осуществляется через операцию точка.

Пример:

```
struct A
```



```

{
    int x;
    int y;
};

void f(A & ref)
{
    cout << ref.x << endl;
    cout << ref.y << endl;
}

int main()
{
    A s = {1, 2};

    f(s);

    return 0;
}

```

5.2.10 Константы

Квалификатор (также классификатор, спецификатор, модификатор, описатель) **константа** — это переменная, которую обязательно инициализировать, и которая после этого не меняет своего значения.

Константа служит для защиты переменных от попытки изменить их значение. Константой переменную делает, использование перед типом данных, квалификатор **const** при инициализации этой переменной.

Общая форма объявления:

```
const тип имя_переменной = значение_константы;
```

Пример:

```
const int MONTHS = 12;
```

Например, переменная MONTHS, это количество месяцев в году. Оно, по определению, всегда будет 12. если в программе просто поставить число 12, то спустя какое то время, велика вероятность, что программист уже не вспомнит, что означает данное число в конкретном месте программы. По этому, константы позволяют также повысить удобочитаемость и информативность текста программы. Константы пришли в дополнение к макросу `#define`. По негласному соглашению имена макросов и констант пишутся в верхнем регистре.

Константы обязательно инициализировать, например:

```
const int foo = 10; /* можно */
const int bar;    /* нельзя */
```

Это логично, поскольку, если мы не инициализируем ее сразу, то вообще никогда не сможем ничего ей присвоить, поэтому, в таком случае, она окажется бесполезной.

Преимущество констант перед `#define` заключается в следующем:

- позволяют явным образом определить тип данных;
- к ним можно применять правила обзора данных, чтобы ограничить описание для определённых функций или файлов;

- константами могут быть более сложные типы данных (составные), например массивы и структуры;
- константы могут использоваться для указания размера массива;
- имена констант можно использовать при отладке программы;
- использование `#define` ведёт к трудно уловимым ошибкам, связанными с не ожидаемым поведением макроподстановки.

Простой пример использования **const**:

```
int main()
{
    const int Size = 10;
    int array[Size] = {0};
    for (int i = 0; i < Size; i++)
        array[i] = i;
    for (int i = 0; i < Size; i++)
        cout << array[i] << " ";

    return 0;
}
```

Также существует ещё ряд спецификаторов, подробности использования, которых хорошо описаны здесь[1]. Здесь они только коротко описаны.

Например спецификаторы классов хранения (памяти):

1. **auto** — используется всегда не явно компилятором (если не определено другое).
2. **register** — указывает, что переменная будет храниться в регистрах процессора.
3. **static** — определяет связывание переменной в файле.
4. **extern** — определяет, что переменная определена в другом файле.
5. **mutable** — применяется в контексте **const**. В некоторых случаях, если структура или класс объявлен как **const**, но требуется снять это ограничение с какой-то конкретной переменной.

cv-спецификаторы:

1. **const** — создаёт константу;
2. **volatile** — указывает на возможность изменения переменной из внешней среды.

Спецификаторы классов хранения и **cv**-спецификаторы являются носителями дополнительной информации о хранении переменной.

5.2.11 Константы, ссылки и указатели

Константы очень интересно сочетаются с указателями и ссылками.

Пример:

```
const int * foo; // или int const * foo;
```

и

```
int * const foo;
```

не одно и то же.

В случае указателя на **const int**. Значение указателя изменить можно (так, чтобы он указывал на что-нибудь другое), а вот значение переменной, на которую он указывает, менять нельзя.

```
int * const foo = &x;
```

Константный (неизменный) указатель на **int**. Значение указателя менять нельзя (как будто это ссылка, а не указатель). Значение того, на что он указывает, менять можно. Заметьте, что константный указатель обязательно инициализировать, как и любую другую константу.

```
const int * const foo = &x;
```

Смесь двух предыдущих пунктов означает: ничего нельзя изменить, ни значение указателя, ни значение того, на что он указывает. Опять же, инициализация обязательна.

У ссылок разнообразия значительно меньше, ибо «указательная» часть ссылки и так всегда константа. Значит, бывает только:

```
const int &foo = x;
```

Ссылка на **int**, который мы (с помощью этой ссылки) не сможем изменить.

И, наконец, пусть наши заявления про неизменность ссылок обретут некий формальный вид. Следующие строки очень похожи по смыслу:

```
int &foo = x;      // инициализация ссылки
int * const bar = &x; // инициализация указателя
```

Фактически, после этого `foo` и `bar` отличаются только синтаксически (везде нужно писать `*bar`, а не просто `foo`, и если мы везде заменим `foo` на `*bar` или наоборот, ничего не изменится[3]).

5.2.12 Динамическое выделение памяти

В языке Си++ для динамического распределения памяти существуют две операции. Это **new** для выделения памяти и **delete** для освобождения памяти.

Общий вид записи:

```
переменная_указатель = new тип_переменной;
delete переменная_указатель;
```

Если операция **new** не может выделить необходимое количество памяти, то операция **new** генерирует исключение. Тип исключения зависит от среды программирования. По стандарту ANSI Си++ должно быть `bad_alloc`.

Если программа не перехватит это исключение, то она снимается с выполнения операционной системой.

Преимущества использования **new** и **delete** перед функциями `malloc()` и `free()` соответственно заключаются в следующем:

1. операция **new** автоматически вычисляет размер необходимой памяти. Нет необходимости в использовании операция **sizeof()**, что предотвращает случайное

выделение неправильного количества памяти;

2. операция **new** автоматически возвращает указатель требуемого типа, в следствии чего отпадает необходимость использовать операцию преобразования типа;
3. Появляется возможность инициализации объекта при использовании операции **new**.
4. Появляется возможность перегрузить операции **new** и **delete** глобально или по отношению к тому классу, который создаётся.

Пример использования:

```
int main(int argc, char *argv[])
{
    int *ptr = 0;
    ptr = new int;
    if (!ptr) {
        cerr << "Размещение объекта закончилось неудачей!";
        return 1;
    }

    *ptr = 20; // присвоение участку памяти значения 20
    cout << *ptr; // вывод на экран значения указателя ptr

    delete p; // освобождение памяти

    return 0;
}
```

Размещение в памяти одномерного массива

Общая форма объявления:

```
переменная_указатель = new тип_переменной[размер];
delete [] переменная_указатель;
```

Пример размещения одномерного массива:

```
int main()
{
    float *ptr;
    ptr = new float [10]; // выделение памяти для указателя типа float
    if (!ptr) {
        cerr << "Размещение объекта закончилось неудачей!";
        return 1;
    }

    // присвоение значений от 100 до 109
    for(int i = 0; i < 10; i++)
        p[i] = 100.00 + i;

    // вывод на экран содержимого массива
    for(int i = 0; i < 10; i++)
        cout << p[i] << " ";

    delete [] p; // освобождения памяти (удаление всего массива)

    return 0;
}
```

Размещение в памяти двумерного массива

Общая форма объявления:

```
переменная_указатель_на_указатели = new тип_переменной*[размер];
цикл
    переменная_указатель_на_указатели[индекс] = new тип_переменной[размер];

// для удаления двумерного массива
цикл
    delete [] переменная_указатель_на_указатели[индекс];
delete [] переменная_указатель_на_указатели;
```

Пример размещения двумерного массива:

```
int main()
{
    const int row = 2; // количество строк
    const int col = 5; // количество столбцов
    int x = 20;
    int **pArray = new int *[row]; // создание массива указателей
    for (int i = 0; i < row; i++)
        pArray[i] = new int [col];

    if (!pArray) {
        cerr << "Размещение объекта закончилось неудачей!" << endl;
        return 1;
    }

    // присвоение значений от 100 до 109
    for (int i = 0; i < row; i++)
        for (int j = 0; j < col; j++) {
            pArray[i][j] = x;
            x++;
        }

    // вывод на экран содержимого массива
    for (int i = 0; i < row; i++) {
        for (int j = 0; j < col; j++)
            cout << pArray[i][j] << " ";
        cout << endl;
    }

    for (int i = 0; i < row; i++) // освобождения памяти (удаление всего
        delete [] pArray[i];    // массива)
    delete [] pArray;

    return 0;
}
```

5.3 Пространство имён Си++

Пространство имён (namespace) — существует для устранения проблемы конфликтов имён возникающих между различными частями кода программы. Эти конфликты возникают при написании программ, как если программист использует сторонние библиотеки разных производителей, так и при использовании своих написанных ранее библиотек, где уже могли быть определены какие-то имена необходимые сейчас. Рассмотрим пример, у вас есть программа содержащая функцию с именем `function()`. И вам неожиданно понадобилось использовать стороннюю библиотеку, где тоже есть функция с таким же именем `function()`, делающая совсем другую работу. Понятно, что изменить имя библиотечной функции вы не можете, поэтому придётся изменять имя вашей функции, что может вызвать большие трудности, если программа велика.

Термин «пространство имён» (**namespace**) — один из основных в языках программирования.

Проведём аналогию с адресом. Если вы скажете: «Я живу в доме 56», ваше заявление не будет иметь никакого смысла, пока вы не уточните, на какой улице расположен этот дом. На каждой улице могут быть свои номера домов; на разных улицах номера могут повторяться, и будут обозначать совершенно разные дома.

Теперь рассмотрим следующий фрагмент программы на Си (заметьте, сейчас всё, что мы говорим, относится как к Си, так и к Си++):

```
int x; /* первая x */

void foo(char x) /* вторая x */
{
    x = 12; /* здесь x -- это вторая x */
}

void bar()
{
    x = 12; /* здесь x -- это первая x */
}

int main()
{
    double x; /* третья x */
    x = 12; /* здесь x -- это третья x */

    return 0;
}
```

В этой программе есть три разных переменные **x**. Это возможно потому, что каждая функция имеет своё пространство имён. Имена (в частности, имена переменных), определённые в нём, видны только из этого пространства (то есть — изнутри этой функции).

Также есть глобальное пространство имён. Первое **x** — именно оттуда. Глобальное пространство имён видно везде. Однако, если то же имя объявить в локальном пространстве имён (как мы поступили, определив второе и третье **x**), оно для этого пространства оказывается «ближе к сердцу».

Но это ещё не всё. Каждый блок имеет своё пространство имён. Например:

```
int main()
{
    int x, y = 321;
```

```

x = 1;
{
    int x;
    x = 2;
    {
        int x;
        x = 3;
        y = 123;
    }
    x = 4;
}
x = 5;

return 0;
}

```

Здесь в одной функции объявлено три разных **x**. Когда внутри блока определяется очередной **x**, он скрывает все предыдущие переменные с таким же именем. Но как только блок заканчивается, «его собственный» **x** исчезает, и вновь становится доступным **x** из родительского блока.

В примере была ещё переменная **y**. В отличие от **x**, её никто не переопределял, поэтому и в самом вложенном, и во внешнем блоке **y** обозначает одну и ту же переменную.

Пространства имён структур и объединений

В Си++ структуры (**struct**) и объединения (**union**) стали создавать полноценные типы данных. Кроме того, каждая структура и объединение (как и, разумеется, каждый класс) получили своё полноценное пространство имён.

Например, вы можете написать:

```

struct Foo
{
    typedef unsigned char byte;
    byte data[16];
};

```

В структуре определено имя **byte**. Но оно «просто так» не видно за пределами данной структуры. Если вы захотите воспользоваться типом **byte** из структуры **Foo**, то вот как это говорят в Си++: **Foo::byte**.

Например:

```

int main()
{
    Foo::byte x;
    x = 17;

    return 0;
}

```

Символ двойное двоеточие (**::**) называется операцией разрешения области видимости (контекста). Он используется, чтобы указать «путь», по которому можно добраться до нужного имени.

Заметим, что в таком пространстве имён можно определять всё, что вам заблагорассудится.

Например, другие структуры:

```
struct Foo
{
    struct Bar
    {
        struct Boz
        {
            int x, y;
        };
    };
};

int main()
{
    Foo::Bar::Boz wow;
    wow.x = 7;
    wow.y = 11;

    return 0;
}
```

Определяемые пользователем пространства имён в Си++

Во всех предыдущих примерах пространства имён были «приложением» к чему-то другому — будь то блок, функция, структура, класс или объединение. Однако, в Си++ есть конструкция, которая позволяет создавать пространство имён в чистом виде.

Пример:

```
namespace MyFavouriteTypes
{
    typedef unsigned char byte;
}

int main()
{
    MyFavouriteTypes::byte data[16];

    return 0;
}
```

Конструкция **namespace** обладает одним уникальным качеством. Пространство имён, ею созданное, можно дополнять в разных местах программы. Пример:

```
namespace MyFavouriteTypes
{
    typedef unsigned char byte;
}

namespace MyFavouriteTypes
{
    typedef char *string;
    typedef const char *const_string;
}

int main()
{
    MyFavouriteTypes::byte data [16];
    MyFavouriteTypes::const_string prompt = "Please enter your name: ";
}
```



```

    return 0;
}

```

Вы должны понимать, что в наших примерах всё записано в одном файле. Однако, в реальной жизни разные куски кода могут приходить из разных *.h - файлов.

Пример:

файл mytypes.h

```

namespace MyFavouriteTypes
{
    typedef unsigned char byte;
}

```

файл histypes.h

```

namespace MyFavouriteTypes
{
    typedef char *string;
    typedef const char *const_string;
}

```

файл main.cpp

```

#include "mytypes.h"
#include "histypes.h"

int main()
{
    MyFavouriteTypes::byte data[16];
    MyFavouriteTypes::const_string prompt = "Please enter your name: ";

    return 0;
}

```

Как и всегда, стоит ли говорить, что пространства имён могут быть вложенными?

Например:

```

namespace Foo
{
    namespace Bar
    {
        namespace Boz
        {
            int z;
        }
    }
}

int main()
{
    Foo::Bar::Boz::z = 17;
}

```

5.3.1 Объявления `using` и директивы `using`

Объявление **`using`** и директива **`using`** — это способ упрощения использования имён переменных и функций некоторого пространства имён.

Объявление **`using`** — обеспечивает доступ к некоторым отдельным идентификаторам.

Директива **`using`** — после её объявления делает доступным пространство имён в целом.

Возможно, вам показалось неудобным всегда использовать названия вроде `MyFavouriteTypes::byte`. И правда, есть способ их подсократить.

Выглядит он так:

файл `mytypes.h`

```
namespace MyFavouriteTypes
{
    typedef unsigned char byte;
}
```

файл `main.c`

```
#include "mytypes.h"

using namespace MyFavouriteTypes;

int main()
{
    byte data[16];

    return 0;
}
```

С помощью директивы **`using namespace`** вы говорите, что «дальше я буду ссылаться на такое-то пространство имён без указания его имени». Эту директиву можно написать и внутри функции:

```
#include "mytypes.h"

int main()
{
    using namespace MyFavouriteTypes;
    byte data[16];

    return 0;
}
```

Тогда её действие распространяется только на эту функцию. Но вспомним, помимо директивы **`using`** есть и объявление **`using`**. Он позволяет использовать одно какое-нибудь имя, не трогая другие. Например:

```
#include <iostream>

using std::cout;
using std::endl;

int main()
{
    cout << "Hello, world!" << endl;
```

```
    return 0;  
}
```

Здесь не создаются свои собственные типы данных, а используются стандартные из библиотеки Си++. Все имена из стандартной библиотеки находятся в пространстве имён `std`. [3]

6 Классы, объекты и инкапсуляция

6.1 Описание классов и объектов

Класс представляет собой тип данных, определяемый пользователем. Класс имеет такое же отношение к объекту, какое имеется между типом и переменной. То есть, в определении класса описывается *форма данных* и способы их использования, а **объект** — это сущность, созданная в соответствии со спецификацией *формы данных*. Класс описывает все свойства и методы типа данных, а объект является сущностью, созданной в соответствии с этим описанием. Также, класс можно представить как тип данных, а объект, как составную переменную.

Пример класса:

```
int n; // создаём переменную целого типа

class A
{
    int i;
    public:
    A() {}
    ~A() {}
};

A obj; // создаём составную переменную(объект) типа A
      // (тип данных A описан через класс)
```

Одним из важнейших механизмов в объектно-ориентированном программировании является **инкапсуляция** (англ. *encapsulation*). Инкапсуляция — свойство, позволяющие объединить данные и код (*методы*) в одном месте (*объекте*) и скрыть реализацию этого объекта от пользователя предоставив ему *интерфейс* (спецификацию). Работа пользователя с объектом осуществляется только через предоставленный интерфейс, а всё внутреннее устройство полностью скрыто. В языке Си++ способ управления доступом осуществляется с помощью использования ключевых слов **private** и **public** подробнее про них написано ниже. При этом **public** создаёт общедоступный интерфейс а **private** скрывает данные в классе[2, 7, 11].

Интерфейс — это совместно используемая часть, предназначенная для взаимодействия двух систем — например, между компьютером и принтером, или пользователем и компьютером. В отношении классов мы говорим об общедоступном интерфейсе. В этом случае потребителем его является программа, использующая класс, система взаимодействия состоит из объектов класса, а интерфейс состоит из методов, предоставленных тем, кто написал этот класс. Интерфейс позволяет вам, как программисту, написать код, взаимодействующий с объектами класса, и таким образом, позволяет программе взаимодействовать с объектами класса. Например, чтобы определить количество символов в объекте класса `string`, не надо просматривать, как этот объект работает внутри. А просто можно воспользоваться методом `size()` этого класса, предоставленного его разработчиком.

В классе также существует понятия **данных** и **методов**. Данные могут создаваться и храниться внутри объекта, а методы позволяют манипулировать этими данными. Данные обычно представлены в виде списка аргументов в классе, а методы в виде списка функций (называемых функциями-членами) этого же класса.

Описание класса для глобальных объектов:

```
class имя_класса
{
    описание_данных_и_методов_класса;
} имя_объект_или_список_имён_объектов;
// или для локальных:
```

```
class имя_класса
{
    описание_данных_и_методов_класса;
};
```

и далее в функции где необходимо использовать наш класс:

```
имя_класса имя_объект_или_список_имён_объектов;
```

в классе можно описывать методы (функции-члены) работы с классом:

```
class имя_класса
{
    имя_функции();
};
```

в таком случае, тело функции описывается следующим образом:

```
имя_класса::имя_функции( список_аргументов )
{
    // код
}
```

или, если тело функции будет небольшим, его можно записать прямо в классе следующим образом:

```
class имя_класса
{
    имя_функции() { // код }
};
```

такой способ записи — это разновидность **INLINE**-функции (встраиваемой функции)

Работа с объектом осуществляется через операцию — стрелка (**->**) или операцию — точка (**.**), в зависимости от того, как объявлен объект (аналогично структурам и объединениям в Си). Стрелка используется, если работа с объектом осуществляется **через указатель**. Точка используется **во всех остальных случаях (в том числе и через ссылку)**.

общий вид:

```
имя_объекта.имя_свойства (метода)
или
```

```
имя_объекта->имя_свойства (метода)
```

Каждому члену класса можно установить его область (уровень) доступа (англ. access control level). Область доступа члена класса определяет участки кода, из которых к этому элементу будет возможно обращаться. В языке Си++ поддерживаются следующие области доступа:

- **private** (закрытый, частный, внутренний член конкретного класса) — обращения к элементу допускаются только из кода методов класса, в котором этот элемент определён. Любые наследники класса уже не смогут получить доступ к этому члену;
- **protected** (защищённый, внутренний член иерархии классов) — обращения к

элементу допускаются из кода методов класса, в котором этот элемент определён, или из любых его классов-наследников;

- **public** (публичный, общедоступный, открытый член класса) — обращения к элементу допускаются из любого места.

Общие описания вида класса с использованием спецификатора доступа (последовательность может быть любая):

```
class имя_класса
{
    описание_данных_и_методов_класса; // по умолчанию будет private:
public:
    описание_данных_и_методов_класса;
protected:
    описание_данных_и_методов_класса;
private:
    описание_данных_и_методов_класса;
};
```

Пример:

```
class queue
{
    int q[100]; // класс queue инкапсулирует
    int sloc, rloc; // свойства q[100], sloc и rloc
public:
    queue(); // конструктор
    ~queue(){} // встроенный (инлайн) деструктор
    void qput(int i);
    int qget();
};

queue::queue()
{
    rloc = sloc = 0; // пример применения конструктора для инициализации
                    // свойств rloc и sloc нулём.
}

void queue::qput(int i)
{
    if (sloc == 99) {
        cout << "Очередь заполнена.\n";
        return;
    }
    sloc++;
    q[sloc] = i;
}

int queue::qget()
{
    if (rloc == sloc) {
        cout << ".\n";
        return 0;
    }
    rloc++;
    return q[rloc];
}

int main()
{
```

```

queue a, b;

a.qput(10);
b.qput(19);

a.qput(20);
b.qput(1);

cout << a.qget() << " ";
cout << a.qget() << " ";
cout << b.qget() << " ";
cout << b.qget() << endl;

return 0;
}

```

В языке Си++ расширены возможности структуры, если сравнивать их с языком Си. В Си++ классы и структуры фактически **взаимозаменяемы**, поскольку структура в Си++ может включать как данные, так и код, который может манипулировать этими данными таким же образом, как и класс. Структуры также могут содержать конструкторы и деструкторы. Единственное отличие между ними связано с тем, что **по умолчанию члены класса имеют в качестве спецификатора доступа `private`**, тогда как спецификатором доступа членов структуры служит **`public`**.

6.2 Конструктор и деструктор

Конструктор – это функция (**функция-член**), которая выполняется один раз при инициализации объекта. Это означает, что он вызывается тогда, когда происходит объявление объекта.

Иногда перед уничтожением объекта требуется выполнить некоторые действия. Для выполнения этих действий существует функция **деструктор**. Например, если в конструкторе будет выделена память с помощью **`new`**, то в деструкторе следует поместить операцию **`delete`**, освобождающую память. Конструктор и деструктор должны иметь такое же имя, что и класс, в котором они объявлены. Конструктор может быть перегружен. Конструктор класса может быть вызван при инициализации объекта явно или неявно.

Общая форма объявления конструктора:

```
имя_класса(список_аргументов);
```

Общая форма объявления деструктора:

```
~имя_класса(список_аргументов);
```

Пример использования конструктора и деструктора в классе A:

```

class A
{
    public:
        A(int n, char d[10]) {} // конструктор
        ~A() {}                // деструктор
};

```

Явный вызов конструктора:

```
A a = A(1, "test");
```

Неявный вызов конструктора:

```
A a(1, "test");
```

Конструктор по умолчанию - это конструктор, применяемый для создания объекта, когда

не предоставляются явные инициализирующие значения.

То есть, это конструктор, который используется в объявлении следующим образом:

```
A a; // здесь конструктор по умолчанию вызванный неявно.
```

Общий вид конструктора и деструктора в описании класса:

```
class имя_класса
{
    имя_класса(); // конструктор 1 (конструктор по умолчанию)
    имя_класса( список_аргументов ); // конструктор 2
    ~имя_класса(); // деструктор
};
```

Пример:

```
class A
{
    int g; // закрытая переменная
public:
    A(); // перегруженный конструктор без параметров
    A(int n); // перегруженный конструктор с 1-им параметром
    ~A() {} // деструктор
};

A::A() // тело конструктора
{
    cout << "start";
}

A::A(int n) // ещё одна версия тела конструктора
{
    cout << n;
}

int main()
{
    A d; // создание объекта d без параметров неявно
    A f(1); // создание объекта f с инициализацией и передачей параметров
            // в конструктор A также неявно

    return 0;
}
```

Конструктор по умолчанию — это один из **неявных функций-членов**, создаваемых компилятором автоматически и их поведение может не соответствовать индивидуальным особенностям дизайна конкретного класса. Ниже перечислены три неявных функции-члена и две операции:

1. конструктор по умолчанию (создаётся, если не было определено ни одного конструктора в классе);
2. конструктор копирования (создаётся, если он не был определён);
3. операция присваивания (создаётся, если она не была определена);
4. деструктор по умолчанию (создаётся, если он не был определён);
5. операция взятия адреса (создаётся, если она не была определена).

Конструктор копирования служит для копирования объекта, в заново созданный объект. **Используется только во время инициализации**, а не обычного присваивания (для присваивания есть перегруженная операция присваивания). То есть, конструктор копирования активизируется всякий раз, когда создаётся новый объект, и в

качестве его первоначального значения, выбирается существующий объект того же типа. Конструктор копирования по умолчанию производит по членное копирование не статических членов, также иногда называемое поверхностным копированием.

Общий вид прототипа конструктора копирования:

```
имя_класса(const имя_класса & имя_объекта);
```

Операция присваивания — это автоматически перегруженная операция присваивания для класса. Используется она, когда необходимо присвоить один объект другому, существующему объекту.

Деструктор по умолчанию ничего не делает.

Неявная операция взятия адреса, возвращает адрес, вызывающего объекта (то есть, значение указателя **this**).

6.3 Указатель на объект и **this**

Для доступа к данным или методам объекта, через сам объект, используется операция членства; точка (.). Если используется указатель на объект, тогда необходимо использовать операцию указателя на членство; стрелка (->). Использование этих операций, аналогично их использованию в структурах. Указатель на объект создаётся с таким же синтаксисом, как и для обычных переменных.

Пример работы указателя на объект и **this**:

```
class A
{
    int num;
public:
    void set_num(int val) {num = val;}
    void show_num();
    int get_num();
    void show_this();
};

void A::show_num()
{
    cout << num << endl; // переменная доступна за счёт не явного указателя this
                        // явный вызов бедет выглядеть this->num
}

int A::get_num()
{
    return this->num; // пример доступа к переменной через
                    // явный указатель this (или можно так (*this).num)
}

void A::show_this()
{
    cout << this << endl;
}

int main()
{
    A obj, *ptr = 0; // создание объекта obj и указателя ptr

    // прямой доступ к объекту через операцию точка
    obj.set_num(1); // передача значения num объекту obj
    obj.show_num(); // вывод значения num на экран

    // доступ к объекту через указатель через операцию стрелка
```

```

ptr = &obj; // присвоение адреса obj указателю p
ptr->show_num(); // вывод значения num на экран через указатель

cout << obj.get_num() << endl; // получение доступа к num через this

cout << &obj << endl; // печать адреса объекта
cout << obj.show_this() << endl; // тоже печать адреса объекта

return 0;
}

```

6.4 Передача аргумента по ссылке на объект

Передавать объект в функцию можно для тех же целей, что и обычную переменную. Когда объект передаётся в функцию, как аргумент, то создаётся копия этого объекта. При таком копировании **не происходит вызова конструктора а происходит вызов «конструктора копирования»**. Если, по каким-либо причинам, нежелательно, чтобы создавалась копия объекта, то надо передавать объект в функцию по ссылке. Также передача через ссылку позволяет избежать некоторых ошибок при работе с классами, таких как вызов деструктора после завершения функции.

Пример передачи объекта по ссылке:

```

class Base
{
    int id;
public:
    int i;
    Base(int i);
    ~Base();
    void neg(Base & b); // инициализируем указатель
};

Base::Base(int num)
{
    cout << "Ctor " << num << endl;
    id = num;
}

Base::~~Base()
{
    cout << "Dtor " << id << endl;
}

void Base::neg(Base & b) // инициализация ссылки на сам объект b_obj, но это
{                          // всё то же b_obj
    b.i = -b.i;
}

int main()
{
    Base b_obj(1); // создаём объект

    b_obj.i = 10;
    b_obj.neg(b_obj); // вызываем функцию neg объекта b_obj с аргументом
                      // b_obj
}

```

```
cout << b_obj.i << endl;

return 0;
}
```

Этот пример очень интересен, поскольку, если попробовать передать объект не по ссылке, а по значению, последовательность действий будет не такой, как ожидается на первый взгляд. Поэтому рекомендуется набрать данный пример на ЭВМ и попробовать передать объект не по ссылке.

6.4.1 Использование констант в классах

Константные методы

Если требуется в программе создать объект константу, например:

```
const A obj(34);
```

а потом вызвать его метод:

```
obj.show();
```

то компилятор ответит ошибкой! Это происходит потому, что код `show()` не гарантирует того, что он не будет модифицировать объект, который не должен быть изменён (поскольку он объявлен **const**).

Для решения этой проблемы необходимо указать **const** после закрывающей скобки списка аргументов функции и до открытия фигурных скобок.

Например так:

```
void show() const;           // прототип внутри класса
void A::show() const        // определение функции
{
    // код
}
```

Общий вид объявления:

```
тип_возвращаемого_значения f( список_аргументов(параметров) ) const;
тип_возвращаемого_значения имя_класса::f( список_аргументов(параметров) ) const
{
    последовательность_операторов
}
```

Следует взять за правило объявлять константными все методы, которые не модифицируют объект, чтобы вы могли использовать в своих программах ссылки на `const-объект[1]`.

Способы создания констант в классе

Поскольку объявление класса описывает как выглядит объект, но не создаёт его, то до тех пор пока объект не создан значение переменных хранить негде. А значит внутри класса невозможно производить инициализацию константы. Но если это всё таки нужно сделать, то возможность обойти это ограничение всё таки существует. Следующие три способа показывают как это сделать.

1. Первый способ использовать перечисление **enum**.

Перечисления — этот тип данных имеет область видимости всего класса и может свободно использоваться как символическое имя целочисленной константы

Пример создания такой константы:

```
class A
{
    enum {Count = 20}; // константа как перечислимый тип
    char Array[Count]; // массив демонстрирующий работу с константой
    // остальной код
};
```

В случае использования **enum** переменная член класса не создаётся. `Count` является просто символическим именем, которое компилятор заменяет числом 20 когда встречает его[1].

2. Второй способ объявить константу как статическую переменную.

Статическая переменная создаётся в момент запуска программы, но область видимости у неё ограничена блоком, в котором она объявлена. При создании статической переменной внутри класса она будет существовать в одном экземпляре для всех объектов данного класса.

Пример создания статической константы:

```
class A
{
    static const int Count = 20; // статическая константа
    char Array[Count]; // массив демонстрирующий работу с константой
    // остальной код
};
```

В приведённом примере существует только одна переменная `Count` разделяемая между всеми объектами класса `A`[1].

3. Третий способ объявить константу как обычно, но инициализировать в заголовке конструктора с помощью списка инициализаторов членов.

Собственно создать константу внутри класса было бы возможно, если бы её в момент создания не требовалось обязательно инициализировать некоторым значением. Эту проблему можно решить используя конструктор класса, так как его вызов создаёт объект до того, как начнёт выполняться код самого конструктора. Способом провести такую инициализацию является использование синтаксиса списка инициализаторов. Располагается он между закрывающей скобкой списка аргументов и скобкой открывающей тело функции.

Пример инициализации константной переменной:

```
class A
{
    const int Count = 20; // статическая константа
    int k; // обычная переменная
    char Array[Count]; // массив демонстрирующий работу с константой
public:
    A() {}
```

```

        A(int n);
        ~A() {}

        // остальной код
};

A::A(int n, int f) : Count(n)
{
    k = f; // инициализация обычной переменной
    // остальной код конструктора
}

```

Используя список инициализаторов так же можно инициализировать и ссылки.

Пример ниже показывает вариант инициализации не только константы, но и обычной переменной помещённой в список инициализаторов членов класса A:

```

A::A(int n, int f) : Count(n), k(f) // инициализация константы и обычной
{
    // переменной в списке инициализаторов
    // остальной код конструктора
}

```

В нём инициализируется переменная k. В случае для обычных переменных нет разницы использовать «список инициализаторов» или «присваивание» внутри тела функции.

Синтаксис списка инициализаторов членов доступен только для конструктора класса!

Также важно помнить, что порядок инициализации определяет порядок следования в определении класса, а не в порядке, заданном самим списком инициализаторов (некоторые компиляторы могут выдавать предупреждение)[6].

6.5 Друзья

Для управлением доступом к различным частям класса используются в основном ключевые слова **public** и **private**. Но в языке предусмотрена ещё одна форма доступа друзья. Друзья позволяют использовать внутренние закрытые **private** данные класса.

Существуют следующие формы дружественного доступа:

1. Дружественные функции.
2. Дружественные классы.
3. Дружественные функции-члены.

6.5.1 Дружественные функции

В случае с объявлением функции другом она получает те же права на доступ к данным класса, как и функция-член. То есть функция не является частью класса но может работать с его данными.

Объявление функции другом в классе выглядит следующим образом:

```

friend тип_возвращаемого_значения имя_функции(список_аргументов);

```

Обязательным является использование ключевого слова **friend**, которое пишется перед именем функции в описании класса.

Пример в программе:

```
class A
{
    int n;          // закрытый элемент класса
public:
    A() {}
    ~A() {}
    friend int f1(A & a); // функция друг класса A
    void f2() {}         // функция-члена класса A
};

int f1(A & a)
{
    return a.n;
}
```

Ещё одним отличием функции-члена от функции-не-члена является отсутствие передаваемого указателя **this**, что не даёт возможности непосредственного использования данных класса. То есть функцию из предыдущего примера нельзя написать так:

```
int f1()
{
    return n; // переменная n недоступна!
}
```

Примером, для чего могут понадобиться функции друзья, является невозможность использования перегрузки операций (про перегрузку операций см. 7.2.2) в операторах вида:

```
A m = 4, k;
k = 5 + m; // ошибка! Так как получается операция 5.operator+(m), где 5
           // целочисленной константы 5 в принципе нет методов.
```

Чтобы обойти данное ограничение можно создать функцию друга классу A, которая будет иметь вид:

```
A operator+(const int c, const A & a)
{
    return c + a.x;
}
```

И использовать её, как показано в примере:

```
class A
{
    int x;
public:
    A() : x(0) {} // тоже самое x = 0;
    A(int n) : x(n) {} // тоже самое x = n;
    ~A() {}
    void Set_A(int const n) {x = n;}
    int Get_A() const {return x;}
    void Show() const {cout << x << endl;}
    friend A operator+(const int c, const A & a); // а здесь, объявляем её
                                                    // другом
};

A operator+(const int c, const A & a)
{
    return c + a.x;
}
```

```
int main()
{
    A m = 4, k; // создание и инициализация переменных класса A

    k = 5 + m; // сложение 5 и m теперь работает

    cout << k; // печать результата

    return 0;
}
```

Дружественные функции также необходимо использовать для перегруженной операции << — поместить в поток (см. 7.2.2). Для перегрузки этой операции существует два способа.

Первый способ заключается в определении функции, как **void**:

```
void operator<<(ostream& os, const A& a)
{
    os << "x = " << a.x;
}
```

Недостаток данного способа в том, что он помещает всю строку целиком в поток вывода, и делает невозможным использование комбинированного вывода. Например такого:

```
cout << k << "см." << endl;
```

Второй способ заключается в определении функции, как возвращающей объект того же типа, что и класс:

```
ostream & operator<<(ostream& os, const A& a)
{
    os << "x = " << a.x;

    return os;
}
```

В данном варианте становится возможно использовать комбинированный вывод, приведённый выше.

6.5.2 Дружественные классы

Дружественные классы используются главным образом для моделирования самим классом объектов реального мира. В случае дружественного класса любой его метод может иметь доступ к приватным и защищённым методам класса, к которому он является другом.

Общая форма:

```
friend class имя_класса;
```

Пример объявления в классе:

```
class A
{
    int x;
public:
    friend class B; // Класс B объявляется другом класса A
    A() : x(0) {}
    A(int n) : x(n) {}
    ~A() {}
};
```

6.5.3 Дружественные функции-члены

В случае, если дружественный класс использует незначительное количество методов базового класса, то имеет смысл сделать дружественными не весь класс, а только те методы базового класса, которые действительно используются.

Общая форма:

```
class имя_класса
{
    friend тип имя_класса_друга::имя_функции_класса_друга(список_аргументов);
};
```

Пример объявления в классе:

```
class B
{
    public:
        friend void A::f(); // функция f() класса A является другом класса B
        A() {}
        ~A() {}
};
```

Для обработки данного объявления компилятор должен видеть класс A до начала описания класса B. Что делать, если в качестве параметров в функции f() используется переменные типа B. Например так:

```
class A
{
    public:
        A() {}
        ~A() {}
        f(B & b) {} // для использования класса B класс A должен располагаться
                    // после класса B
};
class B
{
    public:
        friend void A::f(B & b); // для использования требуется знать класс A
        A() {}
        ~A() {}
};
```

В этом случае в классе A используется класс B. Но класс B объявлен позже, а значит не доступен классу A.

Выходом является использование **опережающего объявления**.

Общая форма записи опережающего объявления:

```
class имя_класса;
```

Пример использования:

```
class B; // опережающее объявление
class A
{
    public:
        A() {}
        ~A() {}
        f(B & b) {} // для использования класса B класс A должен располагаться
                    // после класса B
};
class B
{
    public:
```



```
public:
    friend void A::f(B & b); // для использования требуется знать класс A
    A() {}
    ~A() {}
};
```

Опережающее объявление даёт знать компилятору, что существует такой класс, но не говорит о его структуре. Этого достаточно, чтобы использовать его, как тип данных, но недостаточно для полноценной работы. То есть, из нашего примера нельзя вызвать метод класса B, поскольку до непосредственного описания класса B ничего не известно, о том какие у него существуют методы и данные.

7 Наследование и полиморфизм

Наследование (англ. *inheritance*) представляет собой процесс, благодаря которому один объект может наследовать, приобретать свойства от другого объекта. Это свойство поддерживает концепцию классификации, чем и обуславливается его важность. Эта концепция лежит в основе классификации знаний. Например, красное яблоко представляет собой класс яблоко, который, в свою очередь, представляет собой часть класса фрукт, который в свою очередь входит в большой класс продуктов питания. Без использования классификации каждый объект должен был бы определять все свои характеристики сам. На основе классификации объект нуждается только в определении таких качеств, которые отличают его от других объектов этого класса.

Общий вид наследования классов

```
class имя_базового_класса
{
    данные_базового_класса;
};

class имя_класса_потомка: спецификатор_доступа имя_базового_класса
{
    данные_класса_потомка;
};
```

где, спецификатор_доступа — это один из спецификаторов **public**, **private**, **protected**.

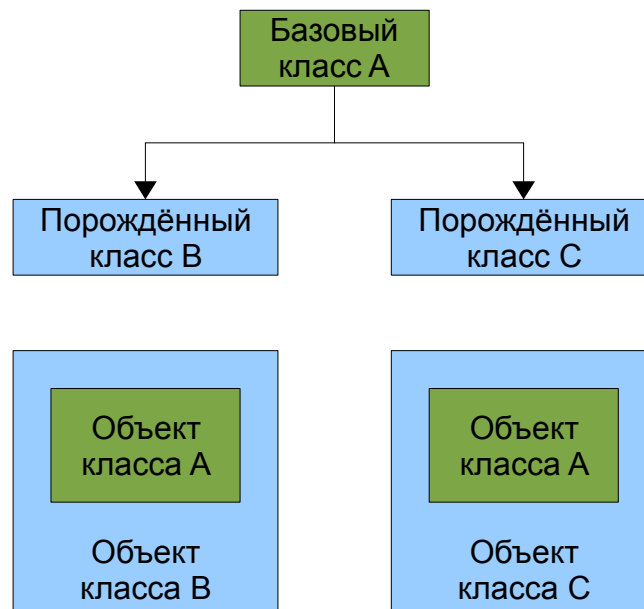


Рис. 7.1. Схема наследования.

Пример наследования с использованием различных спецификаторов доступа:

<pre> class A { private: int i; public: int c; protected: int f; }; class B : public A // наследуется { public: void f_1(); }; class C : public B { public: void f_2(); }; void B::f_1() { c = 10; f = 20; } void C::f_2() // Здесь в функции f_2() { // переменная f доступна f = 10; } int main() { A a_obj; B b_obj; C c_obj; a_obj.d; b_obj.f_1(); c_obj.f_2(); return 0; } </pre>	<pre> class A { private: int i; public: int c; protected: int f; }; class B : private A { public: void f_1(); }; class C : public B { public: void f_2(); }; void B::f_1() { c = 10; f = 20; } void C::f_2() // Здесь в функции f_2() { // переменная f недоступна f = 10; } int main() { A a_obj; B b_obj; C c_obj; a_obj.d; b_obj.f_1(); c_obj.f_2(); return 0; } </pre>
--	--

При этом при наследовании для каждого объекта, созданного по образцу класса потомка, создается свой независимый объект класса предка (рис. 7.1).

7.1 Передача параметров в базовый класс через конструктор

Когда базовый класс имеет конструктор с аргументами, производные классы должны явным образом обрабатывать эту ситуацию путём передачи базовому классу необходимых аргументов.

Пример передачи аргументов из класса предка классу потомку:

```
class A1
{ // 1-ый класс предок (базовый класс)
    protected:
        int a;
    public:
        A1(int i) { a = i; }
};

class A2
{ // 2-ой класс предок (базовый класс)
    protected:
        int b;
    public:
        A2(int i) { b = i; }
};

// В наследует свойства а и b как от A1, так и от A2
class B: public A1, public A2
{ // производный класс
    public:
        B(int x, int y) : A1(x), A2(y){} // конструкторы
                                     // используются для
                                     // передачи значений 10 и
                                     // 20 в классы A1 и A2 соответственно.

        int make_ab() { return a * b; }
};

int main()
{
    B i(10, 20); // создание объекта

    cout << i.make_ab() << endl;

    return 0;
}
```

7.2 Полиморфизм

Кратко смысл полиморфизма можно выразить фразой: «Один интерфейс, множество методов». В Си++ полиморфизм реализован в виде перегрузки функций, перегрузки операций и полиморфного наследования.

7.2.1 Перегрузка функций (полиморфизм функций)

Перегрузка функций (полиморфизм функций, совмещение имён) — позволяет программисту определять несколько функций с одним и тем же именем (интерфейсом). Перегрузка в языке Си++ является практической реализацией принципа полиморфизма времени компиляции.

Перегрузка функций подразумевает использование одного и того же имени функции при разных аргументах. То есть, главным способом различия функций с одинаковыми именами является задание аргументов разного типа и количества. Имена переменных (аргументов функции) не имеют значения.

Общий вид перегрузки функций:

```
тип_возвращаемого_значения имя_перегружаемой_функции( аргумент_1, аргумент_2,
аргумент_3 )
{
    // код;
}

тип_возвращаемого_значения имя_перегружаемой_функции( аргумент_2, аргумент_4 )
{
    // код;
}

тип_возвращаемого_значения имя_перегружаемой_функции( аргумент_1 )
{
    // код;
}
```

Пример перегрузки функции `sqr ()` 4-и раза:

```
int    sqr(    int i        ); // записываем прототипы перегружаемых функций
int    sqr(    int i, int d);
long   sqr(    long l        );
double sqr(double d        );

int main()
{
    cout << sqr(10)        << endl;
    cout << sqr(11.0)      << endl;
    cout << sqr(9L)        << endl;
    cout << sqr(10, 20)    << endl;

    return 0;
}

int sqr( int i ) // перегруженная функция с одним параметром целого типа
{
    return i * i;
}

int sqr(int i, int d) // перегруженная функция с двумя параметрами целого типа
{
    return i * d;
}
```

```

    return i * d;
}

double sqr( double d ) // перегруженная функция с одним параметром дробного типа
{
    return f * f;
}

long sqr( long l ) // перегруженная функция с одним параметром целого длин. типа
{
    return l * l;
}

```

7.2.2 Перегрузка операций

Для повышения возможностей моделирования объектов реального мира в классе добавлена возможность перегрузки операций. Перегрузка операций позволяет определить (переопределить) операции, что даёт возможность их использовать с объектом класса.

На самом деле в Си++ некоторые операции уже перегружены. Например операция (*) при использовании с двумя операндами перемножает их, а при применении её к адресу даёт доступ к значению, расположенному по этому адресу.

Чтобы понять, зачем нужна перегрузка операций сравните два следующих оператора присваивания:

```
strcpy(string_1, string_2);
```

и

```
string_1 = string_2;
```

Скажите, какая форма присваивания строки проще и понятней?

Сама перегрузка операций синтаксически выглядит, как функция.

Общий вид прототипа перегрузки операции:

```
operatorop(список_аргументов);
```

где, op — это символ, перегружаемой операции. Например: **operator+()**.

Допускается перегружать только определённые в Си++ операции. Допустим нельзя перегрузить \$ или @, так как в языке их нет(то есть нельзя создавать новые операции). Также Си++ налагает дополнительные ограничения на перегрузку, перечисленные далее.

- Перегружаемые операции должны иметь, как минимум один операнд пользовательского типа.
- Нельзя перегружать операцию с нарушением её исходного синтаксиса. Например, если операция определена для работы с двумя операндами, то и перегруженную операцию нужно использовать с двумя операндами, а не со одним.
- Нельзя перегружать следующие операции:

sizeof () - операция получения размера;

. - операция принадлежности(членства);

.*- операция принадлежности к указателю;

:: - операция разрешения контекста;

: ? - условная операция.

- Для следующих операций допустимо использовать только функции-члены:

= - операция присваивания;

() - операция вызова функции;

[] - операция индексации;

-> - операция доступа к членам класса через указатель.

При перегрузке операций, допустим, если требуется сложить два объекта, то это можно записать так:

```
n = k + t;
```

Компилятор, в свою очередь, определив операцию класса, объектами которого являются операнды, заменит эту запись на следующую:

```
n = k.operator+(t);
```

Пример класса с использованием функции `sum()` для сложения объектов за место перегрузки операций:

```
class A
{
    int x;
public:
    A() : x(0) {} // тоже самое x = 0;
    A(int n) : x(n) {} // тоже самое x = n;
    ~A() {}
    void Set_A(int const n) {x = n;}
    int Get_A() const {return x;}
    void Show() const {cout << x << endl;}
    A sum(const A & a) {return x + a.x;}
};

int main()
{
    A n = 3, m = 4, k; // создание и инициализация переменных класса A

    cout << n << " " << m << endl; // вывод их содержимого

    k = n.sum(m); // сложение переменных n и m

    cout << k; // печать результата

    return 0;
}
```

Пример класса с реализованной перегрузкой операции сложения:

```
class A
{
    int x;
public:
    A() : x(0) {} // тоже самое x = 0;
    A(int n) : x(n) {} // тоже самое x = n;
    ~A() {}
    void Set_A(int const n) {x = n;}
    int Get_A() const {return x;}
    void Show() const {cout << x << endl;}
    A operator+(const A &a) {return x + a.x;}
}
```

```
};

int main()
{
    A n = 3, m = 4, k; // создание и инициализация переменных класса A

    cout << n << " " << m << endl; // вывод их содержимого

    k = n + m; // сложение переменных n и m

    cout << k; // печать результата

    return 0;
}
```

Существует ещё одна особенность перегрузки. Предыдущий пример показал, как перегрузить операцию, если левый операнд является объектом, а что делать, если он будет цифровой константой? Например:

```
k = 4 + m;
```

С логической точки зрения это абсолютно нормальная операция, но если вспомнить, что компилятор преобразует её к следующему виду:

```
k = 4.operator+(m);
```

то появляется проблема, налагающая некоторые особенности по работе с перегрузкой операций. Решить её помогут функции друзья. Подробнее про друзей читайте в теме 6.5.1.

Вывод данных из класса на экран можно организовать за счёт перегрузки операции << — поместить в поток. Если добавить эту операцию и операцию присвоения, то наш пример станет выглядеть так:

```
class A
{
    int x;
public:
    A() : x(0) {} // тоже самое x = 0;
    A(int n) : x(n) {} // тоже самое x = n;
    ~A() {}
    void Set_A(int const n) {x = n;}
    int Get_A() const {return x;}
    void Show() const {cout << x << endl;}
    A operator+(const A &a) {return x + a.x;}
    A& operator=(const A& rhs); // перегрузка операции присвоения
    A operator<<(ostream & os) {os << x << endl;} // перегрузка операции
    поместить в поток
};
```

Однако, возникает проблема связанная с тем, что воспользоваться перегруженной операцией поместить в поток, необходимо использовать следующий код:

```
k << cout;
```

который является довольно запутанным. По этому, в данном случае, также придётся воспользоваться функциями друзьями.

Перегрузка операции присваивания почти ничем не отличается от перегрузки других операций, за исключением того, что требуется вставить проверку: не присваивает ли объект сам себя. Пример такого присваивания:

```
k = k;
```

Код этой проверки в месте с кодом перегруженной операции присваивания выглядит так:

```
A& A::operator=(const A& rhs)
{
```



```

    if (this == &rhs) return *this; // проверка — текущий объект и объект rhs
                                   // не являются ли, одним и тем же объектом

    x = rhs.x; // присваиваем данные из объекта rhs в текущий

    return *this;
}

```

7.2.3 Полиморфное наследование (динамический полиморфизм)

При использовании механизма наследования мы можем использовать методы из базового класса в классах потомках. То есть, если в базовом классе определена функция, то мы можем её использовать в классе наследнике. Но иногда возникают ситуации, в которых требуется использовать унаследованный метод, который вёл бы себя иначе, чем метод определённый в базовом классе. Такой способ поведения называется полиморфным. При этом ценность полиморфного наследования заключается в том, что можно использовать один метод для разных объектов. При этом модель поведения метода будет выбираться в зависимости от контекста.

Бывают следующие виды полиморфного наследования:

- переопределение методов базового класса в производном классе;
- использование виртуальных функций.

Переопределение методов

Переопределение методов (функций-членов) происходит при использовании заголовка функции в классе наследнике, повторяющего заголовок функции базового класса.

Пример переопределения метода:

```

class A
{
    public:
        void show() {cout << "A" << endl;} // переопределённая функция
};

class B : public A
{
    public:
        void show() {cout << "B" << endl;} // переопределённая функция
};

int main()
{
    A a;
    B b;

    a.show(); // вызвана функция описанная в классе A
    b.show(); // вызвана функция описанная в классе B

    return 0;
}

```

При работе с указателем или ссылкой на объект с переопределённым методом появляются интересные особенности в работе механизма наследования. А именно, можно определять указатель базового класса на класс наследник.

```

B b;
A *a;
a = &b;

```

При вызове метода объекта класса B будет вызван метод класса A. То есть при вызове метода класса всегда будет вызываться метод базового класса.

```

class A      // базовый класс
{
    public:
        void show() {cout << "A" << endl;} // переопределённая функция
};

class B : public A // класс наследник
{
    public:
        void show() {cout << "B" << endl;} // переопределённая функция
};

int main()
{
    A *a;
    B b;
    a = &b;

    a->show(); // вызвана функция описанная в классе A
    b->show(); // и здесь вызвана функция описанная в классе A

    return 0;
}

```

Недопустимо использовать указатель класса наследника на базовый класс без приведения типа. То есть например так:

```

B *b;
A a;
b = &a; // Ошибка!
B = (B *)&a; // Нормально

```

Виртуальные функции

Виртуальная функция – это функция, объявленная с ключевым словом **virtual** в базовом классе и переопределённая в одном или в нескольких производных классах. Виртуальные функции являются особыми функциями, потому что при вызове метода, производного класса через указатель или ссылку на него Си++ определяет какой метод вызвать, основываясь на типе объекта. Для разных объектов вызываются разные версии виртуальной функции. Класс содержащий одну или более виртуальных функций называется **полиморфным классом**.

Виртуальная функция объявляется только в базовом классе с ключевым словом **virtual**, в производных классах повторять ключевое слово нет необходимости.

Пример работы виртуальной функции:

```
class A
{
    public:
        virtual void show()
        { cout << "class A" << endl; }
        // функция show() в базовом классе объявлена, как виртуальная
};

class B: public A
{
    public:
        void show() { cout << "class B" << endl; }
};

class C: public A
{
    public:
        void show() { cout << "class C" << endl; }
};

int main()
{
    A a;
    A *ptr; // создание указателя типа A
    B b;
    C c;

    ptr = &a; // передача адреса базового класса A указателю типа A
    ptr->show(); // вызов виртуальной функции класса A

    ptr = &b; // передача адреса порождённого класса B указателю типа A
    ptr->show(); // вызов виртуальной функции класса B

    ptr = &c; // передача адреса порождённого класса C указателю типа A
    ptr->show(); // вызов виртуальной функции класса C

    return 0;
}
```

Если планируется переопределить какой-либо метод базового класса в производном классе, то обычно этот метод объявляется как виртуальный в базовом классе. Это вынуждает программу выбирать версию метода, основываясь на типе объекта вместо типа ссылки или указателя. Также принято объявлять виртуальный деструктор в базовом классе. Особенность работы виртуальной функции (по сравнению с перегруженной и др.) заключается в том, что решение, о том какую функцию использовать, принимается не во *время компиляции*, а во

время *выполнения программы*.

Если при моделировании необходимо, чтобы один класс наследовался от другого, но при этом есть методы или данные одного класса, которые противоречат по смыслу методам и данным другого класса, то можно создать специальный базовый класс, в который следует поместить все общие методы. В свою очередь, сами эти классы станут наследниками. При этом базовый класс обязан иметь все методы порождённых классов. Но реализовать это не возможно, по причине отсутствия определения данных, появляющихся только в классах наследниках. Выходом является использование чисто виртуальных функций.

Чисто виртуальная функция — функция, которая может не иметь определения.

Общий вид:

```
virtual тип_возвращаемого_значения имя_функции(список_аргументов) = 0;
```

Чисто виртуальная функция состоит из прототипа определённого в классе и обязательно содержит инструкцию `= 0`. А сам класс, будет называться абстрактным базовым классом (АБК). Не возможно создать объект по АБК. Не смотря на то, что чисто виртуальная функция в основном не имеет тела, его создание всё таки допускается.

Абстрактный базовый класс — класс, имеющий хотя бы одну чисто виртуальную функцию.

Ниже представлен пример где показано применение виртуальных функций.

```
class A
{
    int a;
    public:
        A() {}
        A(int n): a(n) {}
        virtual ~A() {}
        virtual void show() { cout << "class A" << endl; }
        // функция show() в базовом классе объявлена, как виртуальная
};

class B: public A
{
    int b;
    public:
        B() {}
        B(int n): b(n) {}
        virtual ~B() {}
        void show() { cout << "class B" << endl; }
};

class C: public A
{
    int c;
    public:
        C() {}
        C(int n): c(n) {}
        virtual ~C() {}
        void show() { cout << "class C" << endl; }
};

int main()
{
    A *ptr[2]; // создание массив указателей типа A
    A a(1);
    B b(2);
```

```
C c(3);

ptr[0] = &a;
ptr[1] = &b;
ptr[2] = &c;

for (int i = 0; i < 3; i++)
    ptr[i]->show(); // какую переменную вызвать определяется
                    // во время компиляции
return 0;
}
```

7.3 Множественное наследование

Обычно наследование осуществляется по схеме:

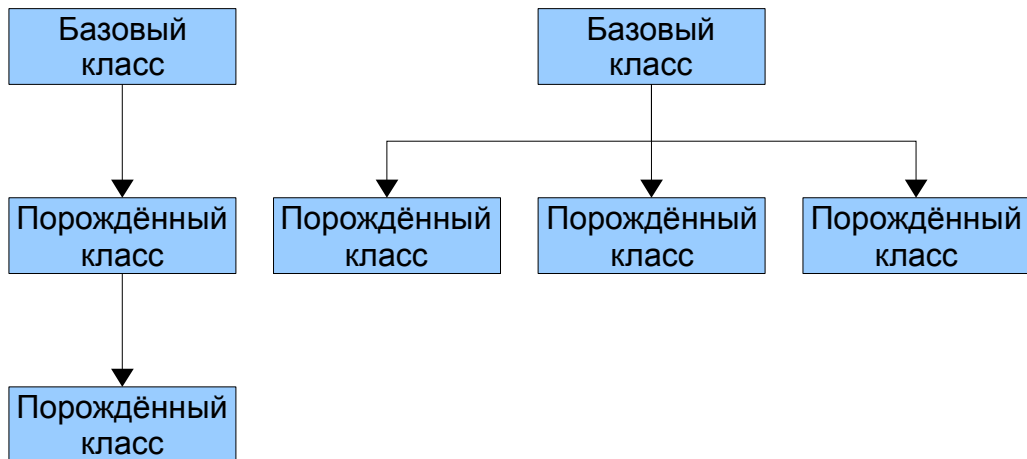


Рис. 7.2. Основные формы наследования, «один ко многим». Но возможны и другие варианты например:

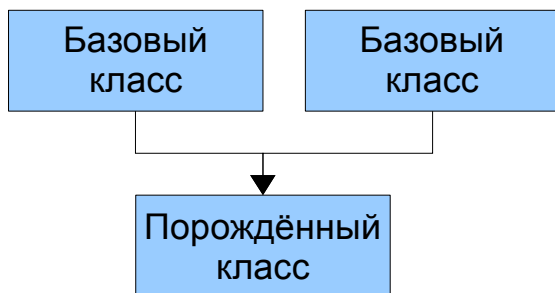


Рис. 7.3. Наследование «многие к одному».

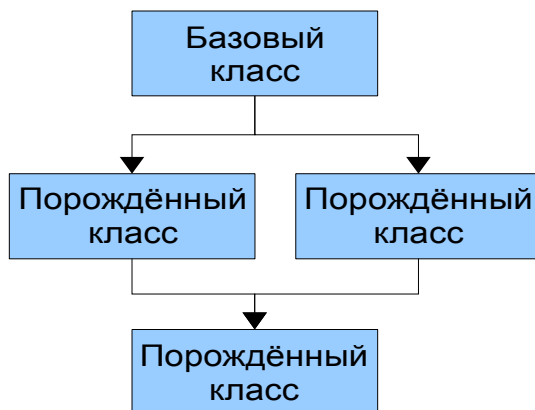


Рис. 7.4. Наследование «один ко многим и многие к одному».

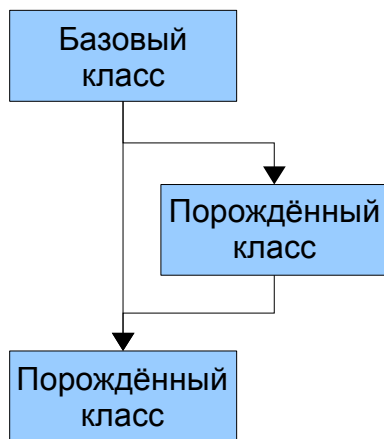


Рис. 7.5. Наследование «прямое и косвенное»

Виды иерархий представленная на рисунках 7.3, 7.4, 7.5 называются множественным наследованием.

Один класс может наследовать атрибуты двух и более классов одновременно. Для этого используются список базовых классов, в котором каждый из базовых классов отделён от других запятой.

Общая форма:

```

class имя_класса_потомка: список_базовых_классов
{
    // ваш код
}
  
```

Пример множественного наследования реализующего иерархию по (рис. 7.6):

```

class A
{ // класс предок
    protected:
        int a;
    public:
        A(): a(0){}
        virtual ~A(){}
        void make(int i) { a = i; }
};

class B : public A
{ // 1-ый класс потомок
    protected:
        int b;
    public:
        B(): A(), b(0){}
        virtual ~B(){}
        void make(int i) { b = i; }
};

class C: public A
{ // 2-ой класс потомок
    protected:
        int c;
    public:
        C(): c(0){}
  
```

```

        virtual ~C(){}
        void make(int i) { c = i; }
};

// в D переопределяется метод make из B(A) и C(A)
class D : public B, public C
{
    int d;
public:
    D(): B(), d(0) {}
    virtual ~D(){}
    void make(int i) { d = i; }
};

int main()
{
    A *ptr;
    D d;
    ptr = (B *)&d; // для передачи адреса объекта d
                  // необходимо сделать приведение типов
    ptr->make(3);
    ptr = (C *)&d; // для передачи адреса объекта d
                  // необходимо сделать приведение типов
    ptr->make(4);

    return 0;
}

```

Здесь каждый объект B и C будет иметь свой экземпляр объекта A, который выше их по иерархии. Для работы с такой иерархией классов требуется использовать приведение типов.

А простая передача адреса становится не однозначной.

```

A *ptr;
D d;
ptr = &d; // непонятно адрес какого базового класса передавать
          // B(A) или C(A)

```

Использование множественного наследования не рекомендуется так как это приводит к неоднозначности базовых классов и коллизии имён переопределяемых методов.

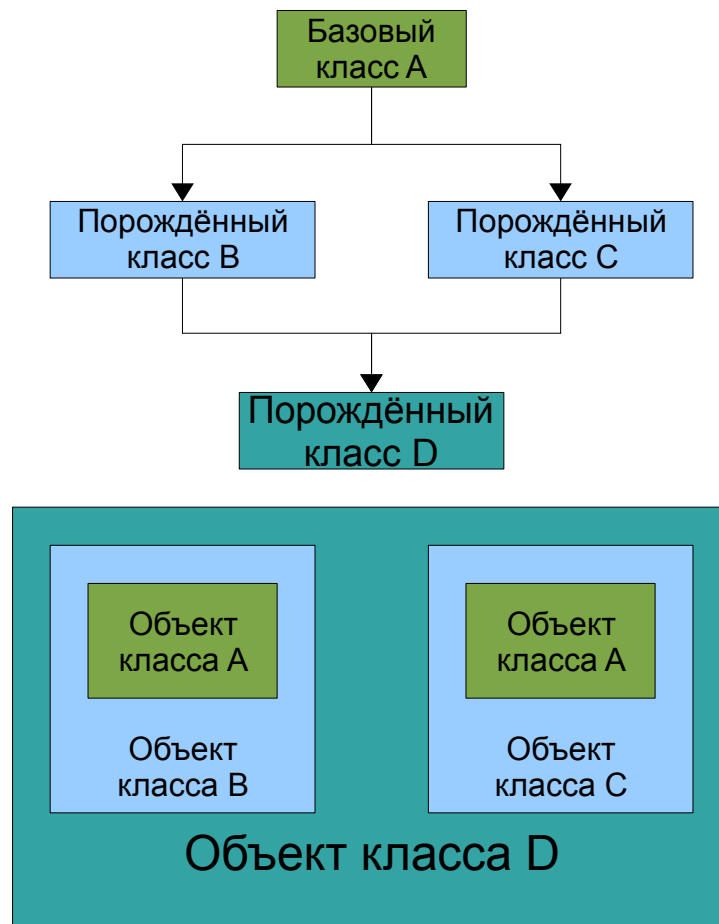


Рис. 7.6. Неоднозначность выбора объекта *A* в объекте *D*.

7.3.1 Виртуальный класс

Если требуется, чтобы объект базового класса был общим для других объектов классов наследников (рис. 7.7), то надо объявить его виртуальным.

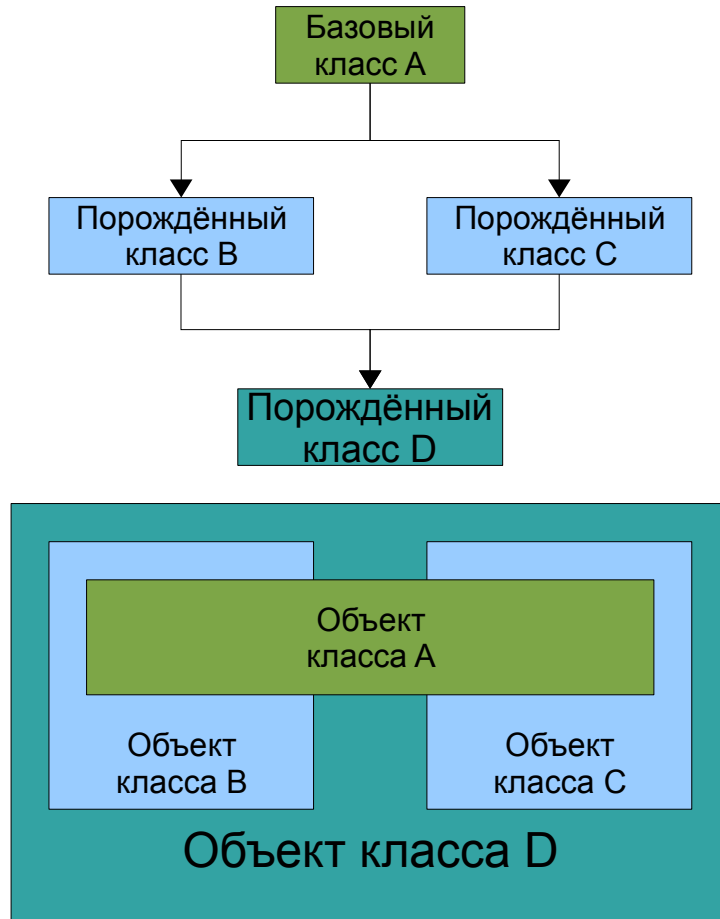


Рис. 7.7. Виртуальный объект A является общим для объектов C из объекта D.

Пример виртуальных базовых классов по (рис. 7.7):

```
class A
{ // класс предок
    protected:
        int a;
    public:
        A(): a(0){}
        A(int n): a(n) {}
        virtual ~A(){}
        virtual void make(int i) { a = i; }
        virtual void show() {cout << a << endl;}
};

class B : public virtual A
{ // 1-ый класс потомок
    protected:
        int b;
```

```

    public:
        B(): A(), b(0){}
        B(int n): A(n), b(n) {}
        virtual ~B(){}
        void make(int i) { b = i; }
        void show() {cout << b << endl;}
};

class C: public virtual A
{ // 2-ой класс потомок
    protected:
        int c;
    public:
        C(): c(0){}
        C(int n): A(n), c(n) {}
        virtual ~C(){}
        void make(int i) { c = i; }
        void show() {cout << c << endl;}
};

// в D переопределяется метод make
class D : public B, public C
{
    int d;
    public:
        D(): B(), d(0) {}
        D(int n): A(n), B(n), C(n), d(n) {}// для передачи данных в
        // виртуальных классах необходимо вызывать базовый конструктор

        virtual ~D(){}
        void make(int i) { d = i; }
        void show() {cout << d << endl; A::show();B::show();C::show();}
};

int main()
{
    A *ptr;
    A a(1);
    B b(2);
    C c(3);
    D d(4); // объект d включает объекты b, c и общий для них всех a
            // надо понимать что для d создаются свои объекты a, b, c

    ptr = &a;
    ptr->show();
    ptr = &b;
    ptr->show();
    ptr = &c;
    ptr->show();
    ptr = &d;
    ptr->show();

    return 0;
}

```

Виртуальные классы дают возможность наследовать один совместно используемый объект как на (рис. 7.7).

8 Не объектно-ориентированные средства языка Си++

8.1 Шаблоны

8.1.1 Шаблоны функций

Шаблоны функций — это обобщённое описание функций. Иначе говоря, шаблон определяет функцию на основе обобщённого типа, вместо которого может быть подставлен определённый тип данных, такой как **int** или **double**. Передавая шаблону тип в качестве параметра, можно заставить компилятор сгенерировать функцию для этого типа данных. Поскольку шаблон позволяет программировать на основе обобщённого типа вместо определённого типа данных, этот процесс иногда называют *обобщённым программированием*. Поскольку типы представлены параметрами, шаблоны функций иногда называют *параметризованными типами*.

Общая форма использования:

```
template <typename имя_произвольного_типа>
тип_возвращаемого_значения имя_функции( имя_произвольного_типа имя_переменной )
{
    последовательность_операторов;
}
```

Пример шаблона по обмену значениями:

```
template <typename A_B>
void A_swap(A_B &a, A_B &b)
{
    A_B temp;

    temp = a;
    a     = b;
    b     = temp;
}
```

здесь **A_B** — это произвольный тип данных. Если необходимо в программе вызвать функцию типа **int**, то компилятор не подставит эту функцию, а сгенерирует на её основе функцию с соответствующим целым типом.

Пример использования шаблона:

```
template <typename A_B>
void A_swap(A_B &a, A_B &b); // прототип шаблона

int main()
{
    int i = 10, j = 20;           // целые
    float x = 10.1, y = 23.3;    // дробные

    cout << "i, j: " << i << ' ' << j << endl;
    A_swap(i, j);                // вызов шаблона функции
    cout << "i, j: " << i << ' ' << j << endl;

    cout << "x, y: " << x << ' ' << y << endl;
    A_swap(x, y);                // вызов шаблона функции
    cout << "x, y: " << x << ' ' << y << endl;

    return 0;
}
```

```
// определение шаблона функции
template <typename A_B>
void A_swap(A_B &a, A_B &b)
{
    A_B temp;

    temp = a;
    a     = b;
    b     = temp;
}
```

8.1.2 Шаблоны классов

Как и в случае с шаблонами функций, есть способ создания класса, не зависящего от типа данных. Шаблон класса — это механизм позволяющий создать класс с типом данных определяемым в момент создания объекта этого класса. Типом данных класса может быть любой тип, как простой так и составной. Определения функций-членов класса необходимо располагать в одном файле с описанием самого класса.

Общий вид шаблона класса:

```
template <typename тип_данных>
class имя_класса
{
    // данные_и_методы_класса
};
```

Общий вид описания функции-члена класса:

```
template <typename тип_данных>
имя_класса<тип_данных>::имя_функции()
{
    список_операторов;
}
```

Общий вид использования шаблона класса:

```
имя_шаблона<тип_данных> имя_переменной;
```

Пример использования шаблона класса:

```
template <typename Type> // создание типа Type
class A
{
    Type n; // создание переменной типа Type
public:
    A();
    ~A();
    Type & f1(Type & a);
};

template <typename Type> // определение принадлежности к классу типа Type
A<Type>::A()
{
    // список операторов
}

template <typename Type> // определение принадлежности к классу типа Type
A<Type>::~~A()
{
    // список операторов
}
```

```

}

template <typename Type> // определение принадлежности к классу типа Type
Type & A<Type>::f1(Type & a)
{
    // список операторов
}

int main()
{
    A<int> obj1; // создание объекта целого типа
    A<double> obj2; // создание объекта с плавающей точкой
    int p = 10;

    obj1.f1(p);

    return 0;
}

```

На примере показано, как можно использовать шаблон класса. При этом по описанию шаблона класса A будет создано два класса для A<int> и для A<double>. Во время создания класса будет произведена замена **параметра типа** Type на реальные типы **int** и **double** соответственно.

8.2 Стандартная библиотека шаблонов (STL)

STL (Standart Template Library) базируется не на объектно-ориентированной парадигме программирования, а на обобщённой. Использование STL позволяет быстрее создавать программу не концентрируя внимание на внутренней организации данных. Сама библиотека состоит из шаблонов классов, представляющих контейнеры, итераторы, функциональные объекты и алгоритмы.

Далее представлены краткие сведения по элементам библиотеки STL.

8.2.1 Контейнеры

Контейнер — это объект, который хранит в себе другие объекты одного и того же типа. Хранимые объекты могут быть объектами в смысле объектно-ориентированного программирования, либо они могут быть значениями встроенных типов[1].

Контейнеры делятся на последовательности, ассоциативные и другие.

Контейнерами последовательностями являются следующие шаблоны: vector, deque, list, queue, priority_queue, stack, bitset, valarray. Для работы с шаблонным классом требуется подключать соответствующий заголовочный файл, например для vector:

```
#include <vector>
```

1) Шаблонный класс vector

vector — это динамический массив с возможностью произвольного доступа к его элементам, автоматическим изменением размера при добавлении/удалении. Время, необходимое на добавление/удаление элемента зависит от его расположения. В конце вектора операции добавления/удаления происходят быстрее, чем в середине или начале, и зависят от общего числа элементов вектора.

2) Шаблонный класс deque

deque — это двусторонняя очередь очень похожая на вектор, но позволяющая вставлять

элемент в начало очереди за постоянное время, то есть не зависит от количества элементов очереди. При этом доступ к произвольному элементу и вставка в середину вектора происходит несколько быстрее.

3) Шаблонный класс `list`

`list` — это двусвязанный список с непоследовательным расположением его элементов в памяти. Поэтому операции добавления/удаления происходят быстро, а операции поиска и доступа медленно.

4) Шаблонный класс `queue`

`queue` — эта очередь отличается от `deque`, большей ограниченностью в методах работы. По сути это классическая очередь с возможностями вставки в конец, извлечения из начала очереди элементов, получения размера очереди и проверкой - пуста ли она. Основой служит `vector`, также как и в `deque`.

5) Шаблонный класс `priority_queue`

`priority_queue` — это очередь, отличающаяся от `queue` тем, что наибольшее значение перемещается в начало очереди.

6) Шаблонный класс `stack`

`stack` — это стэк, в основе которого, также как и в очереди, лежит вектор, но предоставляется интерфейс классического стека, с соответствующими операциями, то есть более ограничен. Классический стек позволяет помещать элемент в конец, извлекать из конца, узнавать размер и проверять не пуст ли стек.

Ассоциативными контейнерами являются следующие: `set`, `multiset`, `map`, `multimap`.

1) Шаблонный класс `set`

`set` — это набор отсортированных данных с уникальным ключом. Может использоваться, как простая база данных. Тип ключа не отличается от типа значения, и может иметь только одно значение, которым является он сам.

2) Шаблонный класс `multiset`

`multiset` — это набор данных, как и `set`, но в отличие от `set`, в `multiset` одним ключом допускается ассоциация с более чем одним значением.

3) Шаблонный класс `map`

`map` — это набор данных, где тип значения отличается от типа ключа. Ключи при этом также уникальны.

4) Шаблонный класс `multimap`

`multimap` — это набор данных, как и `set`, но ключ и данные могут быть разного типа, и с одним ключом допускается ассоциация более одного значения.

Другие типы контейнеров

1) Шаблонный класс `bitset`

`bitset` похож на `vector`, только предназначен для работы на уровне битов, поэтому использует минимум памяти.

2) Шаблонный класс `valarray`

`valarray` похож на `vector`, только специально оптимизирован для скоростного выполнения математических операций и численных расчётов. Отсутствует часть методов STL.

8.2.2 Итераторы

Итераторы — это объекты, позволяющие перемещаться внутри контейнера, подобно тому, как с помощью указателей можно перемещаться по массиву; они являются обобщениями указателей[1].

8.2.3 Функциональные объекты (Функторы)

Функциональный объект — это такой объект, который может быть использован на манер функции. Функторы это часто классы с перегруженной операцией `operator()` (то есть круглых скобок).

8.2.4 Алгоритмы

Алгоритмы — это способы решения определённых задач манипулированием данными внутри контейнеров. Например: поиск, сортировка, копирование, перестановка.

8.3 Работа с исключениями

Исключения (англ. exception) - это непредвиденные (исключительные) ситуации, возникающие во время выполнения программы.

Обработка исключений позволяет упорядочить обработку ошибок времени выполнения. Используя обработку исключений Си++, программа может автоматически вызывать функцию-обработчик ошибок, тогда, когда ошибка возникает. Принципиальным достоинством обработки исключений является то, что она позволяет автоматизировать большую часть кода для обработки ошибок, для чего раньше требовалось ручное кодирование.

Используются следующие ключевые слова **try**, **catch**, **throw**.

Общая форма объявления блоков исключения:

```
try {
    // здесь содержится код программы, в котором
    // происходит генерация исключений
}
catch( тип1 аргумент ) {
    // здесь содержится код, который будет выполнен в случае
    // перехвата исключения тип1 данных в блоке try
}
catch( тип2 аргумент ) {
    // здесь содержится код, который будет выполнен в случае
    // перехвата исключения тип2 данных в блоке try
}
.
.
.
catch( ... ) {
    // здесь содержится код, который будет выполнен в случае
    // перехвата любого исключения независимо от типа данных в блоке try
}
```

Если никакого исключения (ошибки) не возникло, то блок **catch()** выполнен не будет. Каждый блок **catch()** перехватывает исключение только своего типа. Для принудительной генерации исключения служит инструкция **throw**. Она должна находиться в блоке **try** соответственно.

Общая форма объявления:

```
throw исключение;
```

Пример обработки исключения:

```
int main()
{
    try {
        cout << "Начало блока try";
        throw 100;
        cout << "Эта часть не будет выполнена!";
    } catch ( int i ) { // точка куда переходит управление программой после
        // перехвата исключения целого типа
        cout << "Было перехвачено целочисленное исключение!";

        return 1;
    }

    return 0;
}
```

Литература

1. Прата, Стивен. Язык программирования C++. Лекции и упражнения, 5-е изд., Пер. с англ. - М.: издат. Дом «Вильямс», 2007.
2. Шилдт, Герберт. Программирование на Borland C++ для профессионалов/ Пер. с англ. А. И. Панасюк, А. Н. Филимонов; Худ. обл. М. В. Драко. - Мн.: ООО «Попурри», 1999. - 800 с.
3. Си (язык программирования) [Электронный ресурс]: [http://ru.wikipedia.org/wiki/Си_\(язык_программирования\)](http://ru.wikipedia.org/wiki/Си_(язык_программирования))
4. Си++ [Электронный ресурс]: <http://ru.wikibooks.org/wiki/Си++>
5. Параметр (программирование) [Электронный ресурс]: [http://ru.wikipedia.org/wiki/Параметр_\(программирование\)](http://ru.wikipedia.org/wiki/Параметр_(программирование))
6. Солтер, Николас А., Клепер, Скотт Дж. C++ для профессионалов.: Пер. С англ. — М.: ООО «И. Д. Вильямс», 2006. — 912 с.: ил. — Парал. тит. Англ.
7. Хабибуллин И. Ш., Программирование на языке высокого уровня. C/C++. — Спб.: БВХ-Петербург, 2006. — 512 с.: ил.
8. Садовский Б. С., Руководство по установке интегрированной среды разработки (ИСР) Code::Blocks и компилятора MinGW(GCC). - М., 2008.
9. Садовский Б. С., Руководство по использованию интегрированной среды разработки (ИСР) Code::Blocks. - М., 2008.
10. Сайт проекта 7-zip [Электронный ресурс]: <http://www.7-zip.org>
11. Инкапсуляция (программирование) [Электронный ресурс]: [http://ru.wikipedia.org/wiki/Инкапсуляция_\(программирование\)](http://ru.wikipedia.org/wiki/Инкапсуляция_(программирование))
12. Полиморфизм (программирование) [Электронный ресурс]: [http://ru.wikipedia.org/wiki/Полиморфизм_\(программирование\)](http://ru.wikipedia.org/wiki/Полиморфизм_(программирование))
13. Наследование (программирование) [Электронный ресурс]: [http://ru.wikipedia.org/wiki/Наследование_\(программирование\)](http://ru.wikipedia.org/wiki/Наследование_(программирование))
14. Comparison of integrated development environments [Electronic resource]: http://en.wikipedia.org/wiki/Comparison_of_integrated_development_environments
15. [FAQ] Каракули вместо русских букв, Обсуждаем статью для FAQ [Электронный ресурс]: <http://forum.vingrad.ru/forum/topic-189083.html>