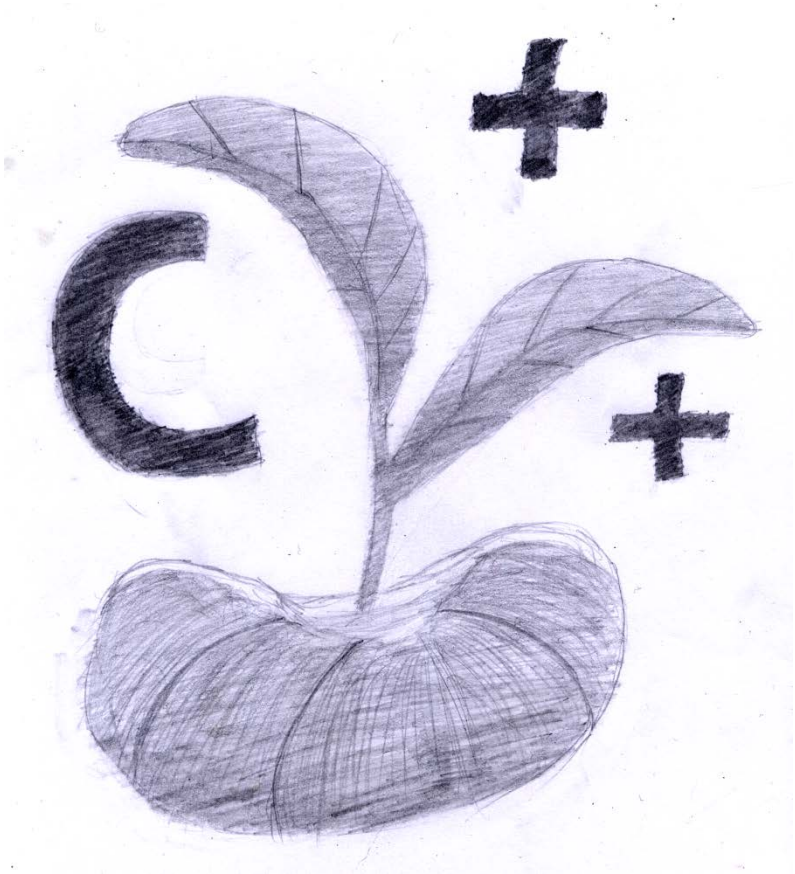
Никехин А.А.

Основы C++ для моделирования и расчетов



Санкт-Петербург
2014

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ

Никехин А.А.

Основы C++ для моделирования и расчетов

Учебное пособие

Санкт-Петербург
2014

Никехин А.А. Основы C++ для моделирования и расчетов. Учебное пособие. СПб: НИУ ИТМО, 2014. – 106 с.

Пособие адресовано для студентов, обучающихся по направлениям 223200, «Техническая физика», 241000 «Энерго- и ресурсосберегающие процессы в химической технологии, нефтехимии и биотехнологии» и содержит сведения об основах языка программирования C++ для моделирования и расчетов.

Рекомендовано к печати Ученым советом инженерно-физического факультета, протокол № 10 от 14.10.2014.

В 2009 году Университет стал победителем многоэтапного конкурса, в результате которого определены 12 ведущих университетов России, которым присвоена категория «Национальный исследовательский университет». Министерством образования и науки Российской Федерации была утверждена программа его развития на 2009–2018 годы. В 2011 году Университет получил наименование «Санкт-Петербургский национальный исследовательский университет информационных технологий, механики и оптики»

История, предназначение и перспективы языка

Это слегка скучный, но обязательный раздел для пояснения того, чего следует ожидать на последующих страницах документа.

Документ посвящен методам моделирования систем и состоит из двух частей: "Основы C++ для моделирования и расчетов" и "Моделирование на C++". В первой части будет дано краткое введение в язык программирования C++. Вторая часть будет составлена из конкретных примеров построения моделей. Это будет совершенно конкретный и практичный документ. В ходе изложения будут рассмотрены способы статистического моделирования, моделирования систем на основе дифференциальных уравнений и другие модели. Реализация моделей будет написана на языке программирования C++.

Язык C++, который будет использован в качестве рабочего инструмента в этом документе, представляет собой объектно-ориентированный язык общего назначения, особенно удобный для системного программирования, но предназначенный для промышленной разработки прикладного программного обеспечения в различных областях: научные вычисления, операционные системы, игры и мультимедийные приложения, встраиваемые системы, другие языки программирования, бизнес-приложения, мобильные приложения и т.д.. Существуют реализации языка (компиляторы) для различных целевых процессорных платформ, таких как x86 (Intel и совместимые), PowerPC, Blackfin, MIPS, XScale, ARM или Cortex.

Разработка языка началась в начале 1980-х годов Бьёрном Страуструпом. Первое издание его книги "Язык программирования C++" вышло в 1985 году. Последнюю на данный момент 4-ю редакцию книги, выпущенную в 2013 году, настоятельно рекомендуем в качестве полного справочника по языку в его современном состоянии. Официальный стандарт, известный как C++98, был принят в 1998 году, спустя 13 лет после выхода в свет книги. В целом история языка оказалась наполненной событиями, информацию о которых вы легко можете найти, и не закончилась в наше время. Сегодня действующим стандартом является C++11, добавивший много существенных возможностей (мы также настоятельно рекомендуем иметь его под рукой), и язык продолжает активно развиваться к следующим запланированным версиям стандарта C++14 и C++17. Значимость и актуальность языка подчеркивается вручением Бьерну Страуструпу в 2013 году диплома и мантии почетного доктора НИУ ИТМО.

Основные достоинства языка с точки зрения обучения:

- практичность, выражающаяся в распространенности языка, доступности средств разработки, применимости для решения разнообразных практических задач, существования сообщества специалистов, доступности библиотек и приложений, востребованности знающих его специалистов на рынке труда.
- поддержка различных подходов к программированию: процедурного, объектно-ориентированного, обобщенного и функционального.
- относительная простота.
- родственность синтаксиса с множеством языков (C, Java, C#, PHP, JS…)

Основные достоинства языка для моделирования систем:

- высокое быстродействие
- использование на мультипроцессорных и гетерогенных (разнопроцессорных) системах
- возможность прямого доступа к аппаратным ресурсам
- наличие развитых библиотек упрощающих создание моделируемых систем
- широкая распространенность и переносимость

Универсальность языка позволяет использовать его в задачах моделирования. Более того, при разработке языка его ключевые концепции классов и объектов были заимствованы из языка Simula, специально предназначенного, как явствует из названия, для компьютерного "симулирования". Помимо разработки самостоятельных программных продуктов C++ может использоваться для расширения функциональности языков программирования Python, R, mathlab, scilab и других, которые могут быть использованы в компьютерном моделировании.

Язык C++ является компилируемым языком (в отличие от интерпретируемых языков и языков с компиляцией времени исполнения). Исходный текст C++ преобразуется компилятором языка в машинный код, исполняемый непосредственно процессорным устройством. Это свойство позволяет писать программы для платформ, не имеющих исполнительных систем времени выполнения или исполняемых библиотек, например, для микропроцессоров, сигнальных или графических процессоров в тех случаях, когда эти платформы позволяют наиболее эффективно решить задачу.

Для C++ существует множество сред разработки, включающих в себя компиляторы языка, которые работают под управлением различных операционных систем: Windows, Linux, OS X (Mac) и др., и создающих исполняемый код как для процессоров, на которых работают сами, так и для внешних целевых процессоров.

Язык является статически типизированным (static typing) т.е. типы переменных определяются на этапе компиляции. Допустимы неявные преобразования типов (слабая типизация или weak typing). Типы переменных задаются явно (explicit typing) исключения C++11.

Объектно-ориентированный подход к созданию программ будет основным при описании языка в данном документе. Преимуществами подхода являются:

- эффективность при работе с множеством объектов немногочисленных типов
- безопасность данных
- модульность программы
- абстрагирование от деталей реализации типа

Принципы объектно-ориентированного подхода:

- Все переменные в программе являются объектами, где объект - экземпляр класса.

• Класс (объектов) является типом переменных, определяющим как сами данные, входящие в состав переменной, так и операции над ними (процедуры, функции, методы класса).
• Программа строится как взаимодействие объектов путем вызова их методов.
• Классы организованы в иерархию наследования, при которой производные классы наследуют свойства базового класса и расширяют или уточняют его функциональность.

Термины и определения

Здесь, в начале, приведены основные понятия, которые могут понадобиться для восприятия дальнейшего материала.

• Целевая платформа (target) - платформа, для которой ведется разработка и на которой исполняется код программы. Пример: микроконтроллер.
• Хост-платформа (host) - платформа на которой ведется разработка. Часто хост-платформа является также целевой. Пример: x86 с ОС Windows (при этом программа может предназначаться для исполнения на микроонтрллере).
• Тип данных - способ хранения, соответствующее ему множество значений/состояний и набор операций, относящихся к данным. Типизация - принадлежность переменных определенным типам данных. Пример: целый тип, булевский тип.
• Статическая типизация (static typing) - язык программирования имеет статическую типизацию если проверка типов осуществляется на этапе компиляции программы. Языки со статической типизацией: ActionScript 3, Ada, C, D, Eiffel, F#, Fortran, Go, Haskell, haXe, JADE, Java, ML, Objective-C, OCaml, Pascal, Seed7, Scala.
• Динамическая типизация (dynamic typing) - язык программирования имеет динамическую типизацию если проверка типов осуществляется на этапе исполнения программы. Языки с динамичекой типизацией: APL, Erlang, Groovy, JavaScript, Lisp, Lua, MATLAB, GNU Octave, Perl, PHP, Pick BASIC, Prolog, Python, R, Ruby, Smalltalk, Tcl
• Класс (class) - составной тип данных, объединяющий данные и операции, относящиеся к этим данным. Переменные класса (типа) называются объектами или экземплярами класса. Пример: класс точек на плоскости.
• Объявление (declaration) - указывает на существование объекта или функции но не приводит к их реализации компилятором. Пример: class Point; (реализация класса не указана)
• Определение (definition) - реализация объекта или функции, приводящая к размещению компилятором памяти для хранения данных и генерированию кода для функций. Определение одновременно является объявлением. Пример: class Point{...реализация...};
• Инициализация (initialization) - присвоение переменной или объекту (экземпляру класса) начального значения.

• Компилятор (compiler) - программа, преобразующая исходный текст на языке программирования в исполняемый код.
• Компоновщик (linker) - программа, собирающая исполняемый код различных модулей в исполняемую программу.
• Байт (byte) - минимальная единица хранения информации, имеющая собственный адрес.

Основы C++
Структура программы

Программа на C++ представляет собой набор файлов реализации и заголовочных файлов, содержащих исходные тексты на языке C++ в текстовой, читаемой и редактируемой человеком форме. Заголовочные файлы включаются препроцессором (буквально вставляются как есть в тексовом виде) в файлы реализации или в другие заголовочные файлы директивой #include. Расширение файла может быть произвольным. Часто для файлов реализации используют расширение *.cpp и для заголовочных файлов - *.h. Но могут быть приняты и другие соглашения о наименовании файлов.

В общем случае заголовочных файлах содержатся объявления классов, типов, переменных и функций, в файле реализации эти объявления определяются или используются.

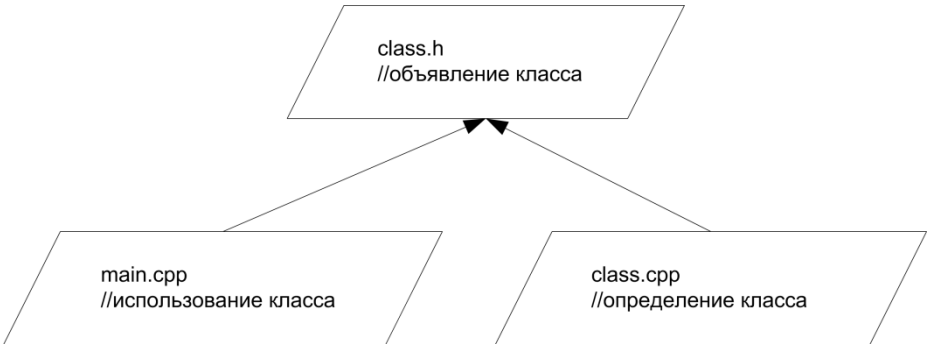


Рисунок 1. Структура программы.

В простой структуре, изображенной на Рисунке 1, предполагается, что содержимое заголовочного файла class.h директивой препроцессора #include целиком включено в файлы main.cpp и class.cpp.

Если исключить из рассмотрения статические и динамические библиотеки, и рассматривать только запускаемые автономные программы, то точкой входа, т.е. началом исполнения программы, будет функция main(). Пример простой программы:

```
#include <iostream> //подключаемый заголовочный файл для
        //вывода строки «Hellow World» в conio


// - здесь и ниже обозначает комментарий и при компиляции все,
// что стоит за ним – игнорируется.
// int – тип возвращаемого значния (целое)
```

```
// main – имя функции
// () – пустой список параметров
int main ()
{   // начало блока-тела функции
   std::cout << "Hello World!"; // вывод строки в консоль
   return 0; // возврат из функции со значением «0»
   // обратите внимание на символ ';' после каждого оператора
}   // конец блока
```

Функцию main() в отличие от других функций нельзя вызвать из других частей программы. Если у функции main() отсутствует оператор "return", то функция возвращает "0" после завершения исполнения. Для любой другой функции, возвращающей значение, отсутствие оператора возврата вызовет ошибку компиляции.

Аргументы командной строки могут передаваться в функцию с помощью параметров:

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

Здесь параметр argc - количество аргументов, argv - указатель на массив строк, содержащих аргументы. Про строки и массивы информация появится чуть позже.

Функция main() является функцией с заранее определенными именем, которой передается управление при старте программы. Программист может создавать свои собственные функции и передавать управление им т.е. вызывать их из других частей собственного кода.

Пример определения (definition) функции:

```
// double – тип возвращаемого значния (вещественное двойной точности)
// sinus – имя функции
// double x - тип и имя аргумента функции
double sinus(double x)
{
   ... // операторы вычисления синуса
   return y; // возврат результата
}
```

Операторы завершаются символом ';'. Просто символ ';' означает пустой оператор.

Обратите внимание на отступы в тексте. Они не обязательны, можно все написать в строчку, разделяя операторы точкой с запятой. Строгих требований к форматированию текста нет. Допустимы множественные пробелы, знаки табуляции, переходы на новую строку везде между отдельными словами (token) исходного текста. Отступы содержательного значения, как в Python, не имеют и используются для форматирования текста так, чтобы он выглядел понятно. Правила форматирования задают так называемые кодовые стандарты, написанные для их соблюдения программистами, участвующими в разработке кода.

Функция должна быть объявлена до первого использования. Пример:

```
#include <iostream>

// Объявление функции sinus() до своего первого использования.
// В этом месте объявление может быть заменено определением
// (см. конец примера).
double sinus(double x);

int main ()
{
   std::cout << "sin="<<sinus(3.14); // вывод в начале строки, затем
                                     // результата работы функции sinus()
                                     // в консоль
   return 0; // возврат из функции со значением «0»
}

// определение (тела) функции sinus()
double sinus(double x)
{
   // вычисление возвращаемого значения по первым членам ряда Тейлора
   return x - x*x*x/6;
}
```

Для того, чтобы функцию можно было использовать в разных исходных файлах, объявление переносят в заголовочный файл *.h, который затем включается директивой #include в каждый из файлов реализации* .cpp, в которых функция будет вызываться.

Операторы языка группируются в блоки, начало и конец которых отмечается фигурными скобками. Тело функции, в том числе функции main(), является таким блоком. Тело функции - не единственный пример использования блоков кода, блоки используются также и там, где необходимы несколько операторов:

```
int func()
{
   if(x>0)
      a;// оператор a. Требуется выполнение только одного оператора

   if(y>0)
   {
      a;// оператор a. Требуется выполнение нескольких операторов
      b;// оператор b.
   }
}
```

При сборке программы компилятор игнорирует комментарии, которые могут быть однострочными "//..." или многострочными "/* ... */".

// Однострочный комментарий до конца строки

/ Многострочный комментарий*
до закрывающей этот комментарий звездочки
**/*

Для создания первой программы и проверки процедуры ее сборки можно воспользоваться любой бесплатной интегрированной средой разработки, доступной для вашей операционной системы.Примеры таких интегрированных сред разработки: Visual Studio Express для Windows (Desktop); Qt Creator; Code::Blocks; Eclips CDT; NetBeans.

В качестве самой простой, нетребовательной к ресурсам персонального компьютера, бесплатной и свободной от юридических ограничений использования, доступной на Mac, Win, и Lin машинах предлагаем Code::Blocks (www.codeblocks.org).

Как показывает опыт преподавания MOOC (massive online open courses) большая часть студентов (~30%) бросает учебу на этапе установки и настройки среды разработки. Не бросайте, установить и запустить Code::Blocks очень просто.

Примеры для этого документа созданы в среде Code::Blocks. После установки и запуска этой среды разработки нужно выбрать меню File->New->Project... дальше выбрать иконку "Console application", язык C++, дать латиницей название для нового проекта и и путь к файлам, в котором будет автоматически создан исходный файл main.cpp, содержащий функцию main(). В других средах проекты создаются также очень легко и все подробности описаны в прилагаемой к ним документации.

Упражнение: Соберите и запустите программу. Бегло изучите структуру меню интегрированной среды.

Процесс сборки программы

Программа - совокупность единиц трансляции - отдельных исходных файлов со всеми своими включениями. Процесс сборки программы представляет собой многоэтапное преобразование исходного кода программы, написанной на языке C++, в машинно-зависимый исполняемый код. В соответствии с принципом раздельной компиляции каждый файл исходного текста (translation unit) компилируется в объектный код отдельно от других исходных файлов. Этапами сборки программы являются: 1. препроцессинг (обработка исходных текстов) 2. компиляция (преобразование единиц трансляции, предобработанных исходных файлов, в машинный код) 3. компоновка (или линковка - окончательная сборка исполняемого модуля из единиц трансляции)

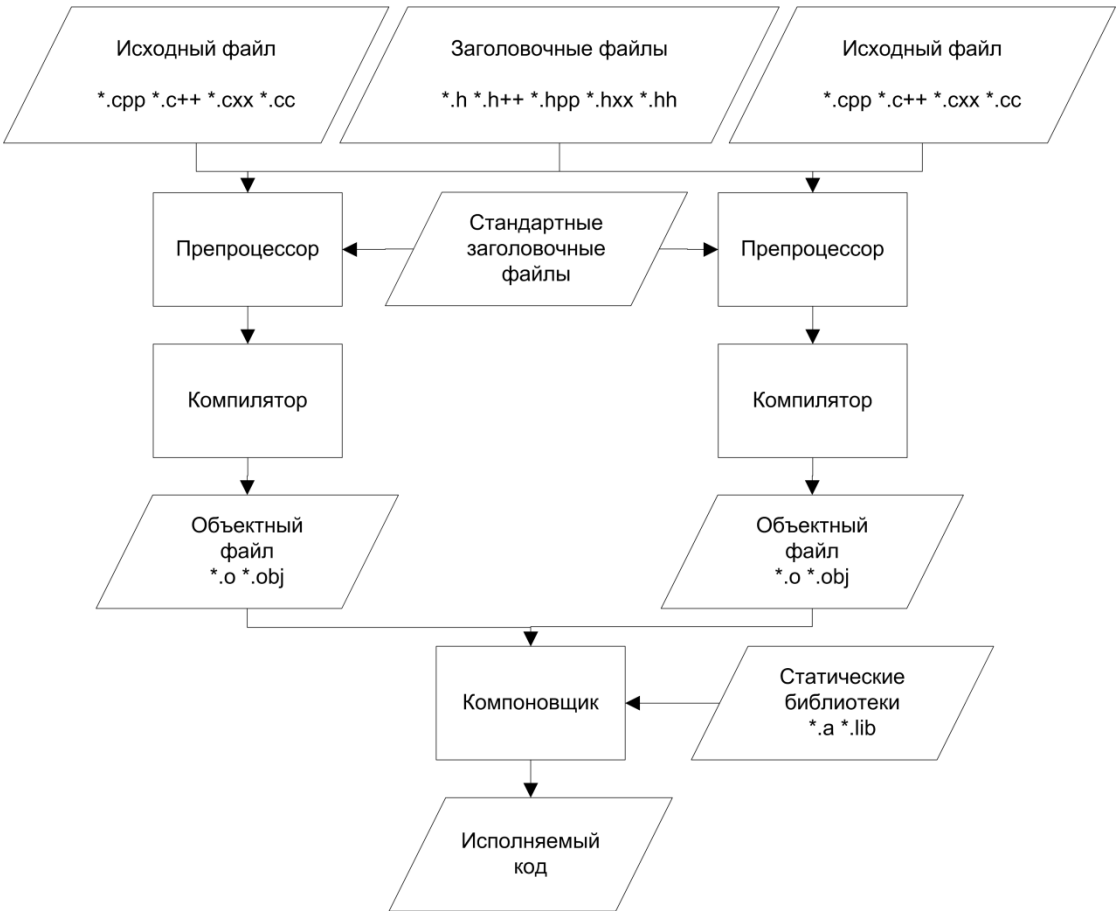


Рисунок 2. Этапы сборки программы

В результате выполнения этих этапов компиляции будет создан файл с исполняемым кодом, например, EXE-файл приложения. Исполняемый код программы исполняется непосредственно процессором целевой платформы. В результате сборки программы на C++ можно создать исполняемые файлы различных типов. Например, статические библиотеки (static library) – код, который подключается во время компиляции компоновщиком и используется в другой программе; динамические библиотеки (dynamic library) то же самое, только библиотека подключается к исполняемой программе во время исполнения, т.е. на лету, динамически; и, наконец, собственно исполняемый код.

Исполнение и отладка

Разработка программы и ее выполнение – это два разных процесса, несмотря на то, что в современных средах разработки они тесно интегрированы. Эти процессы могут выполняться на разных машинах как, например, это происходит при разработке приложений для мобильных телефонов. При запуске программы возможно подключение к ней отладчика, позволяющего останавливать программу в точках остановки (breakpoint), считывать значения ячеек памяти и переменных.

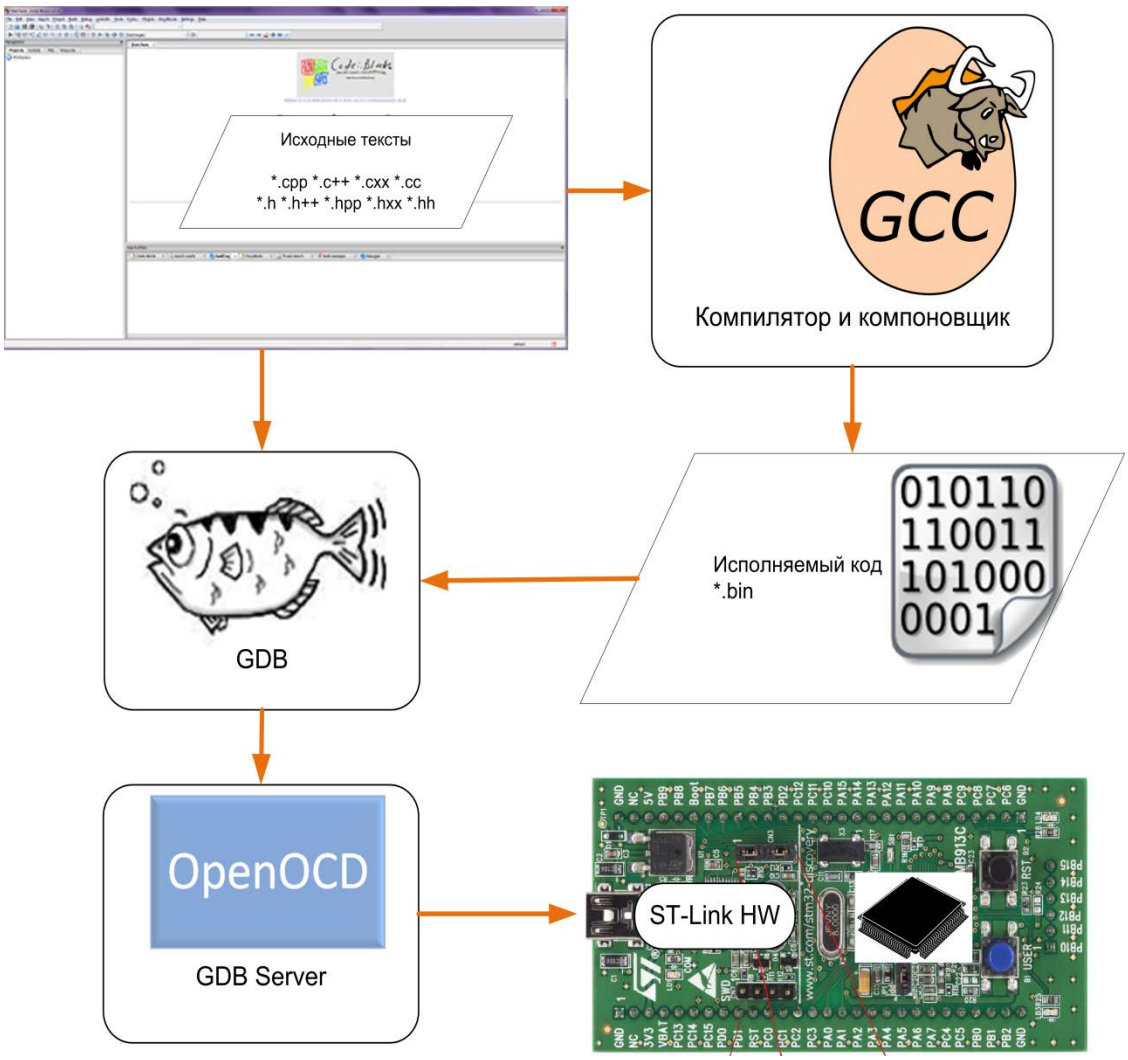


Рисунок 3. Среда разработки

Препроцессор

Перед началом компиляции исходные тексты программы обрабатываются препроцессором. Для управления его работой используются директивы, которые предваряются символом #. Примеры использования препроцессора:

```
#include <iostream>      // Включить заголовок из из стандартных директориев
#include "header1.h"     // Включить заголовок из текущего директория
#include "inc/header2.h" // Включить заголовок из поддиректория inc
                         // текущего директория

#define PI 3.14          // Заменить PI на текст "3.14" везде ниже по тексту
#undef PI                // Удалить определение и ниже по тексту больше не
                         // заменять PI на "3.14".

#define sum(a,b) a+b     // Заменить sum(1,2) на 1+2
#define E \
  любой текст            // Замена E на "любой текст" и продолжение
                         // директивы на следующую строку
```

```
#if defined(X)          // Условная компиляция (то же самое что #ifdef X)
#else                   // альтернативная ветка (#ifndef X or #if !defined(X))
#endif                  // Конец блока условной компиляции #if, #ifdef

#if X>200               // Сравнение с константой
int table[X];           // Использование в коде при определении переменной
#endif                  // Конец блока условной компиляции #if, #ifdef
```

Мы не будем использовать препроцессор активно, но он необходим как минимум для решения двух задач: 1) для включения заголовочного файла:

```
#include "header1.h"    // Включить заголовок из текущего директория
```

и для проверки того, что заголовочный файл не будет включен многократно. Здесь небольшое пояснение: одни заголовочные файлы могут включать в себя другие и в случае если несколько из них включают один и тот же файл, то при сборке проекта возникнет ошибка повторного объявления. Например, если header1.h и header2.h включают common.h, а программа включает оба заголовочных файла как в этом примере:

```
#include "header1.h"
#include "header2.h"
int main ()
{
```

то common.h окажется включенным в модуль с функцией main() дважды. Чтобы этого избежать используют следующую конструкцию:

```
#ifndef COMMON_H // если переменная COMMON_H не определена то
                 // используем все что ниже до соответствующего #endif
#define COMMON_H // определяем переменную

// здесь все объявления файла common.h

#endif // COMMON_H
```

Блоки #if...#endif, #ifdef...#endif, #ifndef...#endif могут быть вложенными.

Объектно-ориентированный подход и начало практической работы

Начнем с определений. Терминология этого документа умышленно немного сужена с целью сформировать понимание концепции объектно-ориентированного подхода в программировании, являющегося основным подходом языка C++, и избежать терминологического наследства процедурных языков, в частности C, расширением которого изначально был C++ (он даже был назван вначале C с классами).

Все что в процедурных языках называется "переменная" в C++ с позиций ООП правильнее называть экземплярами класса или *объектами*. Переменная - это именованная область памяти. Объект (object) - это экземпляр (instance) определенного класса (object class) объектов, типа.

Все объекты в C++ имеют свой *класс* или тип объектов. Класс определяет не только данные объектов класса, но и операции, которые можно производить с объектом. Например, класс комплексных чисел должен будет содержать для всех своих объектов данные о действительной и мнимой составляющих, а также операции умножения, деления, сложения, вычитания и модуля. Такие операции называются методами класса, а данные - полями класса (также данные-члены - data member, member variable).

Еще раз, есть типы - классы, и есть их экземрпляры - объекты.

Классы, необходимые для работы программы, создает программист, но язык сразу предоставляет *фундаментальные типы данных*, такие как *int, double, bool, char*.

Объектно-ориентированный подход в программировании строится на следующих принципах: * *Инкапсуляция* - разделение интерфейсов использования класса и его внтренней реализации. * *Наследование* - возможность расширения классов путем создания классов наследников, включающих в себя поля и методы классов предшественников, и привносящих в них новые поля и/или методы. * *Полиморфизм* - изменение функциональности методов с одним и тем-же именем в зависимости от конкретного объекта, для которого этот метод вызывается. * *Композиция* - конструирование нового класса, состоящего из других классов.

Возвращаясь к практике, для того чтобы приступить к работе необходимо создать новый класс, объект этого класса и начать этот объект использовать в разрабатываемой программе.

Файл "main.cpp":

```
#include "example_01.h" // В этом файле дано определение класса
int main ()
{
   A a;         // Создание объекта с именем "a" класса "A".
   a.print();   // Использование объекта - вызов извне класса
                // метода print() для объекта "a".
}
```

Файл "example_01.h":

```
// Препроцессорная защита от повторного включения содержимого этого
файла
#ifndef EXAMPLE_01_H
#define EXAMPLE_01_H

// Заголовочный файл стандартной библиотеки для консольного вывода
#include <iostream>

// Определение класса "A"
class A
{
   double x, y; // поля класса. По умолчанию недоступны извне класса
public: // позволяет использовать извне класса методы и поля объявленные
         // после этого модификатора доступа
         // Определение метода класса print()
   void print(){
     // Вывод в консоль средствами стандартной библиотеки
     std::cout << "Hello world!" << std::endl;
   }
};

#endif // EXAMPLE_01_H
```

Объявления и определения

Еще немного очень важной терминологии: * *Объявление* (declaration) объявляет о существовании имени (идентификатора) и его типа или атрибутов. * *Определение* (definition) - объявление, которое дополнительно обеспечивает конкретную реализацию для данного идентификатора, заключающуюся в выделении памяти для объекта, соответствующей идентификатору, и возможно, инициализации этой памяти.

Объявлений может быть много. Главное, чтобы они не противоречили друг другу. В пределах одной единицы трансляции допустимо только одно определение (One Definition Rule или ODR).

Примеры объявлений (из стандарта):

```
extern int a;         // объявление 'a'. Ключевое слово extern говорит о
                      // том, что определение дано в другом месте
extern const int c;// объявление 'c'. Значение константы не указано.
                    // Константа не инициализирована благодаря 'extern'.
int f(int);         // объявление 'f'. Объявление функции. Тело функции,
                    // ее реализация, отсутствует.
class S;            // объявление 'S'. Объявление класса "S". Содержание
                    // класса отсутствует.
typedef int Int;    // объявление 'Int'. Объявление синонима типа.
using N::d;         // объявление некого 'd' из пространства имен 'N'.
```

Объявления только создают новое имя, но не сопостовляют его с конкретной реализацией. В отличие от объявления, определение создает соответствие имени и его воплощения. Примеры взяты также из стандарта:

```
int a;                          // определение 'a' типа int
extern const int c = 1;         // определение константы 'c' с конкретным
                                // занчением '1' и типом int
int f(int x) { return x+a; }  // определение функции 'f' и параметра 'x'
class S { int a; int b; };      // определение класса 'S' и его
                        // членов 'S::a' и 'S::b'
class X {               // определение 'X'
   int x;               // определение члена 'x'
   static int y;        // обявление статического члена 'y'. Для того
                        // чтобы оно стало определением статический
                        // член должен быть проинициализирован.
                        // Инициализация статических членов делается
                        // вне класса. О статических членах будет
                        // отдельный рассказ.
   X(): x(0) { }        // определение конструктора 'X'. Что такое
                        // конструктор узнаете позже.
};
int X::y = 1;           // определение статического члена кдасса 'X::y'
enum { up, down };   // определение энумераторов перечисления 'up'
                     // и 'down' (о перечислениях также
                     // будет рассказ ниже)
namespace N { int d; }      // определение пространства имен 'N' и объекта
                            // в нем 'N::d'
namespace N1 = N;           // определение пространства имен 'N1' как
                        // синонима 'N'. Такая конструкция согласно
                        // стандарту считается определением потому что
                        // новое имя N1 благодаря ей имеет вполне
                        // законченный смысл.
X anX;                  // определение объекта 'anX' класса 'X'
```

Язык C++ изначально обладает набором *фундаментальных типов данных*, иначе именуемых встроенными типами, и не требующих определения пользователем (программистом, который их использует). Определение объектов фундаментальныхъ типов, часто называемых переменными, происходит указанием имени объекта после указания типа. Пример:

```
int x;
```

Целые (integer) фундаментальные типы данных с примером определения объекта (переменной):

```
int main ()
{
   // char может быть знаковым (signed) или беззнаковым (unsigned)
   // в зависимости от реализации компилятора
   char x;
   // следующие типы могут принимать отрицательные значения т.е.
   // являются знаковыми (signed)
   short int y;          // также можно писать просто "short y;"
   int z;                // просто целое
   long int v;           // эквивалентно "long v;"
   long long int w;  // эквивалентно "long long w;"
}
```

Перечисленные выше типы отличаются числом бит, отводимых для хранения числа. Стандартом гарантируется неравенство

sizeof(char) ≤ sizeof(short) ≤ sizeof(int) ≤ sizeof(long) ≤ sizeof(long long) где sizeof() - функция вычисления размера в байтах, необходимого для хранения объекта указанног в ее параметрах типа. Также функцию можно использовать для вычисления реального числа байт, отведенных для хранения конкретного объекта, например sizeof(z) выведет число байт, занимаемых "z".

int имеет "естественный размер определяемый архитектурой процессора" и в 32-битных системах как правило составит 4 байта.

Для любого из перечисленных целых типов может быть явно указан признак знаковости или беззнаковости - signed или unsigned. Пример:

```
int main ()
{
   signed char x;        // Теперь char точно  со знаком и может
                         // принимать отрицательные значения
   unsigned short y;   // Беззнаковый short
}
```

Упражнение: найти размер целых фундаментальных типов в вашей системе. Для этого взять пример выше и вместо строки "Hello World" вывести размер:

```
std::cout << sizeof(char) << std::endl;
```

Логические (булевские) фундаментальные типы данных с примером определения объекта:

```
bool f;
```

Целые (integer) и булевские (boolean) типы типы совместно называются целочисленными (integral types).

Вещественные фундаментальные типы данных с примером определения объекта:

float p;
double q;
long double r;

Вещественные и целочисленные типы совместно называются арифметическими.

Символьные типы:

```
char      // то же самое, что и целый char выше. Используется для хранения
          // любого из символов основной кодировки
wchar_t   // Для хранения любого из символов расширенной кодировки (в основе
          // использует char16_t или char32_t)
```

Исследовать особенности реализации фундаментальных арифметических типов на вашей платформе можно с помощью sizeof() или стандартной библиотеки std:numeric_limits, вызывая ее методы:

```
// std - пространство имен стандартной библиотеки
// numeric_limits - шаблон класса реализующего нужные методы. О шаблонах
// будет информация ниже.
// <char> - параметр шаблона класса, имя арифметического типа для анализа
std::numeric_limits<char>::func()
```

Пример использования библиотеки:

```
#include <iostream> //подключаемый заголовочный файл для вывода в conio
#include <limits>     //функции стандартной библиотеки std::numeric_limits

using namespace std; //использование пространства имен std

int main () {
   // размер в байтах
   cout << "sizeof="<< sizeof(char) << endl;
   // минимальное значение
   cout << "min="   << std::numeric_limits<char>::min() << endl;
   // максимальное значение
   cout << "max="   << std::numeric_limits<char>::max() << endl;
   // число значащих бит (исключая знаковый и экспоненту для
   // вещественных чисел)
   cout << "bits="  << std::numeric_limits<char>::digits << endl;
   return 0;
}
```

Операции с фундаментальными типами

Класс объекта характеризуется не только данными, но и операциями, которые можно производить с ними. Для арифметических типов в языке определены операции умножения, деления, сложения, вычитания и другие операции. Соответствие типа и доступных операций сведено в таблице

Операция	**Описание**	**Целые (int)**	**Вещественные (double)**	**Булевские (bool)**
x++	прибавление 1 к **x** (постфиксный)	+	-	-
x--	вычетание 1 из **x**	+	-	-
++x	прибавление 1 к **x** (префиксный)	+	-	-
--x	вычетание 1 из **x**	+	-	-
-x	унарный минус	+	+	-
+x	унарный плюс	+	+	-
x * y	умножение	+	+	-
x / y	деление (с округлением целых вниз)	+	+	-
x % y	деление по модулю (сохраняется знак **x**)	+	-	-
x + y	сложение	+	+	-
x - y	вычитание	+	+	-
x << y	побитовый сдвиг **x** на **y** бит влево **(x * pow(2, y))**	+	-	-
x >> y	побитовый сдвиг **x** на **y** бит вправо**(x / pow(2, y))**	+	-	-
x < y	логическое меньше	+	+	-
x <= y	меньше или равно	+	+	-
x > y	больше	+	+	-
x >= y	больше или равно	+	+	-
x == y	равно	+	+	+
x != y	не равно	+	+	+
x & y	побитовое "и"	+	-	-
x ^ y	исключающее "или"	+	-	-
x | y	побитовое "или"	+	-	-
~x	побитовая инверсия x	+	-	+

!x	логическое отрицание	+	-	+
x && y	логическое "и"	+	-	+
x || y	логическое "или"	+	-	+
x = y	присвоение	+	+	+
x += y	операция с присвоением **x = x + y,** также -= *= /= <<= >>= &= |= ^=	см. соотв. операцию		

Идентификатор (имя) объекта никаким образом не определяет его тип (класс). Идентификатором может быть последовательность букв латинского алфавита, цифр и символа "_". Идентификаторы не могут начинаться с цифры. Идентификатор не может совпадать с зарезервированными ключевыми словами. Ключевые слова C++ (сразу море полезной но не обязательно понятной информации в одном месте; всему будет объяснение позже):

```
alignas    // директива выравнивания данных по границам в байтах
alignof    // возвращает значение границы выравнивания
and        // альтернатива логическому И - "&&"
and_eq     // альтернатива побитовому И с присваиванием "&="
asm        // декларация встраиваемого ассемблерного блока
auto       // начиная с C++11 обозначает неявное определение типа
           // по контексту переменной
bitand     // альтернатива побитовому И - "&"
bitor      // альтернатива логическому ИЛИ - "|"
bool       // булевский фундаментальный тип
break      // выход из внутреннего цикла или оператора switch
case       // ветвь оператора switch
catch      // конец блока обработки исключений
char       // символьный тип
char16_t // символьный тип в кодировке UTF-16
char32_t // символьный тип в кодировке UTF-32
class      // декларация класса
compl      // альтернатива побитовому дополнению (унарному НЕТ) - "~"
const      // неизменямый (кнстантный) тип
constexpr // значение функции или переменной может быть вычислено во
          // время компиляции
const_cast  // преобразование типов с различными const-volatile
            // спецификаторами
continue    // пропуск оставшегося для исполнения тела внутреннего цикла
decltype    // определение типа выражения
default     // ветвь по умолчанию в операторе switch
            // или в C++11 указание явной инструкции созлдания
```

```
            //конструкторов
delete      // удаление объектов
do          // цикл do-while
double      // тип чисел с плавающей точкой двойной точности
dynamic_cast // преобразование типов в пределах иерархии наследования
else        // альтернативная ветвь оператора if
enum        // декларация перечисляемого типа
explicit    // запрет неявных преобразований в конструкторах инициализации
            // и при копировании
export      // deprecated, запрет разворачивания шаблонов inline
extern      // внешняя декларация
false       // булевский литерал "ложь"
float       // тип чисел с плавающей точкой одинарной точности
for         // цикл for
friend      // спецификатор, разрешающий доступ к закрытым и защищенным
            // полям и методам извне класса
goto        // безусловный переход управления
if          // оператор условия
inline      // использование кода функции без инструкции вызова
int         // целочисленный тип
long        // большой целочисленный тип
mutable     // изменяемый член константного класса
namespace   // пространство имен
new         // динамическое создание и инициализация объектов
noexcept // проверка или декларация возможности исключений
not         // альтернатива логическому НЕТ - "!"
not_eq      // альтернатива логическому НЕТ с присваиванием - "!="
nullptr     // литерал, означающий нулевой указатель
operator // перегруженный оператор
or          // альтернатива логическому ИЛИ - "||"
or_eq       // альтернатива логическому ИЛИ с присваиванием - "|="
private     // спецификатор закрытого доступа к полям и методам или
            // видимости при наследовании
protected   // спецификатор защищенного доступа к полям и методам или
            // видимости при наследовании
public      // спецификатор публичного доступа к полям и методам или
            // видимости при наследовании
register    // рекомендация компилятору для размещения переменной в регистре
reinterpret_cast // преобразование типов методом изменения
                 // интерпретации бинарных данных
return      // возврат из функции
short       // короткий целочисленный тип
signed      // целочисленный тип со знаком
sizeof      // размер переменной или типа в байтах
```

```
static       // поле класса, общее для всех объектов. Декларация
             // статической переменной.
static_assert  // проверка выражения времени компиляции
static_cast    // преобразование типов времени компиляции
struct       // декларация структуры
switch       // оператор выбора
template     // декларация шаблона класса или функции
this         // указатель на объект класса, из которого был произведен
             // вызов метода
thread_local // декларация потоковой переменной
throw        // сигнализация исключения
true         // булевский литерал "истина"
try          // начало блока обработки исключений
typedef      // создание синонима для имени типа
typeid       // возвращает идентификацию типа переменной
typename     // альтернатива ключевому слову class в шаблонах
union        // декларация союза
unsigned     // целочисленный тип без знака
using        // выбор пространства имен; создание синонима для имени типа
virtual      // методы класса, которые могут быть переопределены
             // при наследовании
void         // функция не возвращает значения
volatile     // изменяемый тип объекта, могут быть отключены
             // некоторые оптимизации
wchar_t      // символьный тип для символов, занимающих больше одного байта
while        // цикл while и do-while
xor          // альтернатива побитовому исключающему ИЛИ - "^"
xor_eq       // альтернатива побитовому исключающему ИЛИ - "^="
```

В дополнение к ключевым словам есть два идентификатора, которые имеют значение в определенном контексте, но вне его могут использоваться как идентификаторы объектов.

override - в производном классе указывает на то, что виртуальная функция переопределяется.

final - в базовом классе указывает на то, что виртуальная функция не может быт переопределена.

C++ чувствителен к регистру. Это означает, что идентификаторы, написанные строчными и прописными буквами - разные идентификаторы.

Идентификаторы, содержащие двойное подчеркивание __ или начинающиеся с подчеркивания и заглавной буквы зарезервированы и не должны использоваться как идентификаторы. Все имена, начинающиеся с подчеркивания также зарезервированы для использования в глобальном пространстве имен, но могут использоваться в пользовательском пространстве имен, например, в качестве имен полей класса и т.д.

Приведение типа

Приведение типа - преобразование одного типа в другой *Неявное приведение типов* - преобразование типа без явного указания в тексте программы, к какому типу оно осуществляется. Пример:

```
double pi=3.14;
int x=pi; // вещественное число неявно преобразуется в целое
```

Явное приведение типов - преобразование типа с явным указанием результирующего типа. Пример:

```
double pi=3.14;
int x=static_cast<int>(pi); // вариант №1 (основной). Также см dynamic_cast,
                            // reinterpret_cast, const_cast
int x=int(pi);              // вариант №2 (функциональна нотация)
int x=(int)pi;              // вариант №3 (в стиле С)
```

Динамическое преобразование

Динамическое преобразование позволяет во время исполнения программы преобразовать тип указателей или ссылок.В случае преобразования к несовместимому типу будет получен нулевой указатель для преобразования указателя и сгенерировано исключение 'bad_cast' для ссылки. Преобразование корректно используется для приведения объектов производных классов к базовым классам.

```
class Base
{
};
class Derived: public Base
{
};

Base b, *pb;
Derived d, *pd;

pb = dynamic_cast<Base*>(&d);       // ok: производный к базовому
pd = dynamic_cast<Derived*>(&b);  // ошибка при компиляции: неполиморфный
                        // базовый приводится к производному
                        // если базовый сделать полиморфным (дать
                        // ему виртуальную функцию) то
                        // преобразование вернет нулевой указатель
```

Статическое преобразование

Явное статическое преобразование используется для приведение одного типа к другому на этапе компиляции. Преобразоввываться могут объекты, ссылки и указатели. Допустимо преобразование указателя базового класса к указателю производного или указателя типа 'void*' к типизированному указателю. Корректость преобразования лежит в ответственности программиста, его использующего.

```
pd = static_cast<Derived*>(pb); // преобразование указателя к указателю на
                                // производный класс может привести к
                                // неопределенному поведению

#include <cstdlib>
Base *p = static_cast<Base*>(malloc( sizeof(Base) )); // преобразование
                                                      // указателя 'void*'
                                                      // к 'Base*'
```

Также статическое преобразованием используется для явного преобразования встроенных типов и преобразования между типами, для которых оно определено явным образом ('operator Type()')

```
double pi=3.14;
int i = static_cast<int>(pi);

class A
{

};
class B
{
public:
   operator A() // оператор преобразования в класс 'A';
   {
      return A();// может быть что то более осмысленное, возвращающее 'A'.
   }
};

A a;
B b;

a=static_cast<A>(b);

a = b; // неявное преобразование к A и затем присваивание
```

Произвольное преобразование

Позволяет на свой страх и риск производить преобразования между указателями, указателем и целым, ссылками.

```
A *pa = new A;
B *pb = reinterpret_cast<B*>(pa);
```

Преобразование константных типов

Для преобразования указателя или ссылки на объект константного типа в указатель или ссылку на неконстантный объект используется оператор const_cast(выражение), возвращающий значение указанного неконстантного типа:

const int i = 0, &ri = i;
const_cast<int&>(ri)=1; // изменение значения константы
Оператор 'const_cast' может аналогично const изменять спецификатор volatile.

Инициализация объекта

Существует множество способов определить объект класса или переменную фундаментального типа, проинициализировав ее *литералом* (фиксированным значением определенного типа, заданным в исходном тексте программы), или другим объектом данного класса или класса, который может быть *неявно приведен* к данному классу.

Можно даже определить объект, оставив его неинициализированным:
int x01; // неинициализированный объект (переменная)

Во время исполнения объект **x01** будет хранить случайные данные, случайно находящиеся в оперативной памяти в момент его создания. Это приемлемо, если **x01** в дальнейшем будет присвоено новое значение. Присвоить нужное значение сразу при определении объекта можно следующими многочисленными способами:

```
int x02{};      // инициализация по умолчанию (значением 0)
int x03=int();
int x04=int{};
int x05=10;     // инициализация литералом (значением 10)
int x06(10);
int x07{10};    // также: int x07={10};
int x08=int(10);
int x09=int{10};
int x10=x02;    // инициализация копированием (объектом x02)
int x11(x02);
int x12{x02};
int x13={x02};
```

Предпочтительным вариантом инициализации является вариант с использованием фигурных скобок ={}.

Литералы

Литерал - фиксированное значение определенного типа, заданное в исходном тексте программы (в отличие от констант, являющихся неизменяемым объектами, часто инициализированными все теми же литералами).

Числовые литералы

Числовые литералы представляют собой собственно вещественные или целые числа, записанные в исходном тексте программы. Целые могут быть представлены в разных системах счисления:

```
222                 // десятичное число
0xDE,0XDE           // шестнадцатеричное число
0336                // восьмеричное число
0b11011110, 0B11011110  // бинарное число
```

Целые числовые литералы могут дополняться суффиксами, определяющими их длину, знаковость или то и другое вместе: 'U' или 'u' - суффикс знака означает, что литерал беззнаковый 'L' или 'l' - суффикс длины означает, что литерал имеет тип long 'LL' или 'll' - суффикс длины означает, что литерал имеет тип long long Пример:

222UL // десятичное беззнаковое число unsigned long int

Вещественные литералы отличает десятичный разделитель '.' в числе, наличие десятичной экспоненты 'e' или 'E', или и разделитель и экспонента. Примеры вещественных литералов, перечисленные через запятую:

3.14 // целая и дробная части, разделенные '.'
2e+6 // 2 миллиона с символом экспоненты 'e' и знаком '+'
2E+6 // 2 миллиона с символом экспоненты 'E' и знаком '+'
2E6 // 2 миллиона с символом экспоненты 'E'
2e-3 // 0.002 с символом экспоненты 'e' и знаком '-'
2.99792458e+8 // скорость света, м/с
6.626e-34 // постоянная планка, Дж*с
.25 // четвертинка, целая часть не приведена
33. // 33, дробная часть не приведена

Вещественный литерал без суффикса имеет тип double. Суффикс f или F означает тип float, l или L - long double.

Символьные литералы

Символьные литералы используются для определения отдельных символов текста. Для обозначения символа используются одинарные кавычки и в случае использования расширенной кодировки один из префиксов 'u', 'U' или 'L'. Примеры:

'w' // символ, тип char
L'w' // символ wchar_t (UTF-1 ISO 10646)
u'w' // символ char16_t (UCS-2 ISO 10646)
U'w' // символ char32_t (UCS-4 ISO 10646)

Ряд символов, в частности непечатываемых, задается *управляющей последовательностью*, начинающейся с обратной черты '\' - перевод строки (LF), '\' - возврат каретки (CR), '\' - табуляция, '\'' - кавычка.

Строковые литералы

Строковые литералы определяют строки текста и обозначены двойными кавычками:

"hello world" // строка символов типа char
L"hello world" // строка символов типа wchar_t (UTF-1 ISO 10646)
u"hello world" // строка символов типа char16_t (UCS-2 ISO 10646)
U"hello world" // строка символов типа char32_t (UCS-4 ISO 10646)
u8"hello world" // строка символов типа char (UTF-8)

В сочетании с префиксами кодировки или без них может использоваться префикс 'R', означающий "raw string" - буквальную строку без управляющих последовательностей.

u8R'(symbols\t ")' // эквивалентно строке: symbols\t "

Пользовательские литералы

Создатель своей программы может создать свои собственные литералы с помощью несколько неожиданной, но вполне работоспособной конструкции.

```
constexpr double operator"" _km(long double x)  { return 1E3*x; }
constexpr double operator"" _cm(long double x)  { return 1E-2*x; }
constexpr double operator"" _mm(long double x)  { return 1E-3*x; }

double x = 25.0_cm; // 0.25 - длина в метрах
```

Необязательный спецификатор "constexpr" говорит о том, что выражение должно быть вычислено на этапе компиляции. Т.е компилятор производит все необходимые вычисления и в объектный код вставляет результат этих вычислений. При исполнении программы этот уже готовый результат только используется.

Параметром такого литерала может выступать простой литерал следующих типов (в примере тело оператора заменено бессодержательным return):

```
constexpr double operator"" _cm(long double x)          { return 0.; }      // №1
constexpr double operator"" _cm(unsigned long long x)   { return 0.; }  // №2
constexpr double operator"" _cm(const char* x, size_t)  { return 0.; }  // №3
constexpr double operator"" _cm(const char16_t* x, size_t){ return 0.; } // №4
constexpr double operator"" _cm(const char32_t* x, size_t){ return 0.; } // №5
constexpr double operator"" _cm(const wchar_t* x, size_t) { return 0.; } // №6
constexpr double operator"" _cm(const char* x)          { return 0.; }     // №7
constexpr double operator"" _cm(char x)                 { return 0.; }       // №8

double x1 = 25.0_cm;    // вызывается оператор №1 - long double x
double x2 = 25_cm;      // вызывается оператор №2 - unsigned long long x
double x3 = "25"_cm;    // вызывается оператор №3 - const char* x, size_t
double x4 = u"25"_cm;   // вызывается оператор №4 - const char16_t* x, size_t
double x5 = U"25"_cm;   // вызывается оператор №5 - const char32_t* x, size_t
double x6 = L"25"_cm;   // вызывается оператор №6 - const wchar_t* x, size_t
double x7 = u8R"(25)"_cm; // вызывается оператор №7 - const char* x
double x8 = '2'_cm;     // вызывается оператор №8 - char x
```

Тип возвращаемого значения может быть любым. Суффикс литерала не обязан начинаться с подчеркивания, однако литералы без подчеркивания зарезервированы для будущих возможностей стандарта и их использование нежелательно.

Булевские литералы

Литералы типа bool:

true, false

Кстати, не пишите конструкции вида

```
if(x==true)
  ...
```

Если 'x' имеет значение true, то условие 'x==true' не добавляет смысла. Достаточно написать так:

```
if(x)
  ...
```

Литералы-объекты

Литерал может быть составным и состоять из нескольких простых литералов:

```
class Point{
   int _x,_y;
public:
   constexpr Point(int x, int y): _x{x},_y{y}{} // constexpr конструктор
};

Point p={0,0};
```

Для того, чтобы класс был литералом, конструктор должен быть объявлен со спецификатором constexpr.

Литерал-указатель

Литерал, означающий указатель с нулевым значением:

```
nullptr // нулевой указатель
```

Если тип объекта может быть определен компилятором из типа выражения, которым он инициализирован, может использоваться тип 'auto'. Примеры:

```
int x{5};
float y{1.0F};
auto z{x * y}; // auto -> float: будет проведено неявное преобразование
               // выражения к более широкому типу
int (*g)();     // g - указатель на функцию вида int g();
auto f1 = g;   // auto -> int g();
auto f2 = [](){return 0;}; // auto -> лямбда-функция
```

Упражнения

- Написать функцию вычисления целой части квадратного корня вещественного аргумента. Прототип функции:

```
int squareroot(double x);
```

Значение функции вычислять на основании свойства квадратов целых чисел: квадрат целого числа 'n' равен сумме нечетных чисел, не превышающих '2n'.

- Написать функцию вычисления целочисленной степени вещественного аргумента:

```
int power(double x, int y);
```

Производные типы

Выше были рассмотрены фундаментальные (встроенные) типы данных а также составные типы. Язык предоставляет пользователю производные типы данных, такие как указатели, ссылки, и константы.

Указатели (pointers)

Объект является типизированной областью памяти. Функции при этом объектами не являются несмотря на то, что они также как и объекты могут занимать память. В C++ фундаментальной единицей хранения является байт, состоящий из последовательности бит, числокоторых определяется реализацией. Да, их в байте может быть не восемь, хотя это большая редкость. Каждый байт имеет уникальный адрес. Указатель хранит адрес байта с которого начинается типизированная память. Указатель однозначно связан с типом, который хранится по этому адресу. Указатели объявляются путем добавления символа '*' к идентификаатору.

```
double *x;   // неинициализированный указатель на double
double* y;   // то же самое. и не важно, где стоит '*' но минимум один
             // пробел нужен между идентификатором и типом;
A *pa;       // укаазатель на объект
A* pa, *pb; // два указателя на объекты одного типа
```

Физически указатель представляет собой адрес памяти. Для 32-битных архитектур процессоров размер адреса как правило составляет 32 бита. Проверьте в своей среде разработки сами:

```
cout << sizeof(A*)<<endl; // размер в байтах!
```

Инициализация указателей

```
A a;                 // определение объекта с выделением памяти
A*pa = {&a};    // '&' - оператор взятия адреса объекта
A *pb=new A;   // создание нового объекта в динамической памяти
A *pc{nullptr}; // инициализация указателем-литераом
```

Чтение значения по указателю

```
double x{3.14};   // определение и инициализация 'x'
double *px{&x}; // '*' - указывает на то, что 'px' это указатель;
                // '&' - оператор взятия адреса
cout << *px << endl;  // вне определения оператор '*', разыменование
                      // указателя, означает объект, на который укаазатель
                      // ссылается. Результат:  3.14
```

Запись в память по указателю

```
double x, *px{&x};  // объявление 'x' и там же объявление указателя на него
*px = 3.14;           // "разыменование" указателя и запись по адресу на
                      // который он указывает
cout << x << endl; // 3.14
```

Арифметика указателей для работы с указателями возможно использование операторов сложения, вычитания и сравнения.

```
if (pa != pb) // копирование объектов только если адреса разные
  *pa=pb;

while(*pa++ = *pb++); // копирование массива 'pb' в массив 'pa' до первого
                      // нулевого элемента
```

Массивы

Массивы в C++ унаследованы от C и предсталяют собой последовательность элементов одного типа, расположенную в памяти непрерывно т.е. один за другим. Массив объявляется с помощью квадратных скобок:

```
A aa[5]; // неинициализированный массив из 5 элементов
         // от aa[0] до aa[4] типа 'A'
```

Размерность массива может быть вычислена по инициализирующему его выражению.

```
int a[]={0,1,2};      // Инициализированный массив (a[3]={0,1,2}; )
char s[]="hello";     // Строка. Строки в C завершаются нулевым символом
                      // '\0'. В данном случае длина строки 6 символов,
                      // включая '\0'.
char* s="hello";      // указатель на строку "hello", включающую '\0'
A a;
A aa = {a,a,a};       // массив из 3 копий объекта 'a'
A bb = new A[3];      // массив из 3 новых объектов типа 'A', созданных
                      // в динамической памяти
```

Массив может быть многомерным:

```
double xx[3][5];   // двумерный массив 3x5
double yy[2][3]={{1,2,3},{4,5,6}}; // Двумерный массив с элементами
                                   // типа 'double'
double zz[][]={{1,2,3},{4,5,6}};   // То же самое, только не указаны
                                   // размерности т.к. они понятны из
                                   // инициализации
double vv[3][5][2];                // трехмерный массив 3x5x2.
extern int n, m;
double ww[n][m];                   // размерность может быть задана
                                   // целочисленным литералом или
                                   // выражением целого типа.
```

Массивы независимо от размерности занимают непрерывный блок памяти. При этом младшие размерности (указанные правее) изменяются быстрее. Образец размещения в памяти трехмерного массива 'double yy[2][3][2]' (всего 12 элементов) приведен на рисунке:

[0][0][0]	[0][0][1]	[0][1][0]	[0][1][1]	[0][2][0]	[0][2][1]	[1][0][0]	[1][0][1]	[1][1][0]	[1][1][1]	[1][2][0]	[1][2][1]

Рисунок 4. Трехмерный массив в памяти.

Доступ к элементам массива осуществляется по индексам. Нумерация индексов начинается с 0. Да, a[1] это второй элемент массива.

```
for( int i=0; i<n; i++)
for( int j=0; j<m; j++)
   zz[i][j]=j+i*m;
```

```
for (auto &x: xx) // для записи переменная цикла должна иметь вид ссылки:
                  // auto &x. Т.к. тип элементов 'xx' известен, можно
                  // использовать переменную цикла типа double: double &x.
  xx = 0;
```

Чтение элементов массива:

```
for( int i=0; i<n; i++)
for( int j=0; j<m; j++)
   cout << zz[i][j] << endl;

for (autox: xx)
  cout << xx << endl;
```

Массив в C++ реализован как указатель на первый элемент массива, соответственно возможны преобразования массива в указатель и операции с ним

```
#include <typeinfo>
A aa[5], *pa = aa;
if(typeid(pa) != typeid(aa))
{
 cout << typeid(pa).name() << endl;
 cout << typeid(aa).name() << endl;
}
*(pa+1); // второй элемент массива - то же самое что и pa[1],
      // aa[1] или *(aa+1)
```

Двумерный массив - это массив массивов или указатель на казатель.

```
A aaa[3][4],**ppa=aaa, *pa=aaa[0]; // здесь 'pa' - указатель на первый из
                        // блоков по 4(второе измерение)???
```

Есть также такое понятие, как указатель на функцию:

```
int (*f)(float)=nullptr; // Неинициализированный указатель на функцию,
                         // возвращающую int с одним параметром float
x[0] = f(3.14f);         // Вызов функции через указатель
```

Упражнение: написать процедуру умножения матриц фиксированной размерности

Ссылки (reference)

Ссылки являются синонимами для имени объекта. Ссылки объявляются путем добавления символа '&' к идентификатору. Ссылки обязательно должны быть инициализированы. Обращение к ссылкам производится так же как и к объектам.

```
double &rx = x; // в этом контексте символ '&' означает ссылку.
rx=3.14;
cout<<x<<endl; // 3.14
```

В отличие от указателей ссылки не могут быть неинициализированными или иметь нулевое значение, невозможно изменить значение ссылочного

объекта - работа с ним происходит как с синонимом объекта, на который он ссылается. Между ссылкой и указателем действуют операторы преобразования

```
A a;
A *pa{&a};
A &ra{*pa}; // получение ссылки по указателю на объект.
             // Результат - ссылка на объект, на который указывает
            // указатель
pa=&ra;     // взятие адреса объекта по указателю на этот объект.
           // Результат - указатель на объект 'a'.
```

Ссылки могут ссылаться на константные объекты

```
const A a;          // константа
const A& cr=a;  // cr является ссылкой на константу
```

В C++11 были введены ссылки на временные объекты, не имеющие собственного имени (rvalue-references), которые можно использовать для переменщения объектов. При обычном копировании объект-источник продолжает существовать и производися накладная операция копирования старого объекта в новый объект. При перемещении же старый объект переносится в новый без дублирования данных.

Например, есть контейнер (массив) из 10 больших объектов. При добавлении 11-го объекта программист должен получить память для хранения 11 объектов, скопировать в новую память 10 старых объектов, вызвав для них конструктор копирования, и добавить новый объект, затем удалить 10 старых объектов и память для их хранения. Операция достаточно ресурсоемкая. Перемещение позволяет объекты массива перенести без необходимости копирования объектов.

Ссылки на временные объекты объявляются путем добавления двух символов '&&' к идентификатору.

Примеры возникновения временных объектов:

```
struct A{
   int x,y;
};
A f()
{
   return {1,2};
}

A a,b;
A &&r1 = f();     // ссылка на возвращаемое функцией значение
                  // (это rvalue т.к. не может стоять слева от присваивания)
A &&r2 = {0,0};   // ссылка на литерал (это rvalue т.к. литерал также не
                   // может стоять слева от присваивания)
A &&r3 = move(a); // функция std::move() превращает объект в rvalue

// Обмен значениями между 'a' и 'b'
```

```
A tmp(move(a)); // копия не создается, происходит перемещение в 'tmp'
a = move(b);
b = move(tmp);
```

NB: терминология rvalue/lvalue появилась раньше возникновения C++ и означала выражения, которые могут быть слева от оператора присваивания (lvalue) и выражения, которые могут стояь справа от знака присваивания (lvalue).

Константы (const values)

Можно запретить изменение объекта программой сделав его тип константным. При этом объект получит новый тип, отличающийся от неконстантного. Запрет изменения (константность) задается ключевым словом *const*.

```
const int c=3;      // константа всегда инициализируется
const A a;          // константный (неизменяемый) объект
A const b;          // то же самое
```

Константность может сочетаться с указателями и ссылками.

```
const int* p=a;          // Значения, на которые указывает p константны
int* const p=a;          // Указатель p является константой
const int* const p=a;    // Все константы, и значение и указатель

const A *pa = {&a};   // указатель на константный объект
const A &ra = {a};    // ссылка на константный объект
*pa = b;              // ошибка: объект 'a' изменять нельзя, он константный
pa=nullptr;           // а сам указатель не константный
a = b;                // ошибка: объект 'a' изменять нельзя
```

Будет легче понять смысл типа, если читать справа налево. Например, " const int*" - указатель на целую константу. " int* const " - константный указатель на целое.

Константный объект должен быть инициализирован в момент определения. Инициализатором может выступать как литерал, известный в момент компиляции, так и выражение или просто объект, получаемые во время исполнения программы.

Константное выраженние (constexpr)

Во время исполнения компилятор может производить вычисления их результат использовать в окончательном коде программы. Исходными данными для вычисления могут быть перечисления, арифметические и пользовательские литералы. При этом можно использовать функции, определеные как constexpr и операторы, не меняющие операнды (не '++', '--', '=' и их комбинации). Пример:

```
constexpr double pi=3.14;
// constexpr площадь круга
constexpr double area(double d)
{
  return pi*d*d/4;
}
```

```
int main()
{
  double s = area(4); // вычисляется во время компиляции
}
```

В вычислениях могут участвовать простые пользовательские классы. Для этого они должны иметь любой конструктор со спецификатором constexpr кроме конструктора копирования или перемещения или иметь явно определенный тривиальный конструктор по умолчанию или быть составным (aggregate) типом (не быть производным, не иметь явно определенных пользовательских конструкторов, закрытых или защищенных полей, не иметь виртуальных функций). Особое изящество эта конструкция приобретает при использовании пользовательских литералов:

```
class Speed
{
   const long double value; // скорость в метрах в секунду
public:
   constexpr Speed(long double speed):value {speed}
   {
   }
   constexpr long double getvalue()
   {
      return value;
   }
};
constexpr Speed operator"" _ms(long double x){return x;}
constexpr Speed operator"" _kmh(long double x){return x/3600.0*1000.0;} //тысячу можно и сократить, но constexpr же
constexpr Speed operator"" _mph(long double x){return x/3600.0*1609.0;}

Speed s={50.0_kmh};
cout << s.getvalue() << endl;
```

Составные и прочие типы данных

Язык предоставляет программисту средства для создания новых типов данных, расширяющих фундаментальные типы. Ключевым способом создания новых типов являются классы. Они будут предметом подробного описания в разделе объектно-ориентированное программирование. Здесь будут кратко перечислены прочие типы, доступные в языке.

Перечисления (enum)

Перечисления - это тип данных, объекты которого могут принимать ограниченное число значений.

```
enum state {on,off};    // тип перечисления со значениями ON=0 и OFF=1
enum state st;          // переменная типа state может принимать только
                        // значения перечисления 'state'
```

Перечисление создает в области видимости новые имена значений перечисления. В данном случае это 'on' и 'off'.

```
enum state {on=0,off=1}; // Явное определение значений
state st;                // Определение объекта 'st'. Ключевое
                         // слово 'enum' не обязательно.
enum {on,off} st; // Анонимный тип перечисления (объект назван, а тип - нет)
```

Перечисления были реализованы как целочисленные значения, поэтому если перечисление не типизировано, возможно свободное преобразование значения перечисления в целое.

```
state st;
st = off;
int x = st;
int y = off;
```

Перечисления могут быть типизированными с указанием базового типа, лежащего в их основе (underlying type). Типизация добавляется ключевым словом *class*.

```
enum class state : bool {on=true,off=false}; // типизированное перечисление.
                                             // базовый тип - bool
state st = state::off;   // Пример использования
state st = off;          // Ошибка - значение перечисление находится в облати
                         // видимости типа 'state'
state st = false;        // Ошибка - литерал false не принадлежит типу state
bool b = st;             // Ошибка - тип 'st' не совместим с 'bool'
```

Структуры (struct)

Объекты уже известных нам типов могут быть объединены в группы, называемые структурами

```
struct T{              // структура: составной тип из набора
                       // полей и методов
  double x;            // поля структуры, являющиеся данными
  double y;
  double mod(){    // метод - функция, вложенная в структуру
    return sqrt(x*x+y*y);
  }
};
```

Объявление объектов-структур:

```
T s;                // без инициализации
T q{1.0, 2.0};    // x=1.0, y=2.0
T *p = &q;
```

Обращение к методам и полям класса происходит с помощью операторов селекторов. Для объектов и ссылок:

```
s.x = 1.0; // запись значений в поля
s.y = 1.0;

cout << s.x; // чтение значений полей для вывода
```

```
p->y = 1;
cout << s.mod();     // 1.41421
cout << p->mod(); // 2.23607
```

Для указателей

```
p->x= -1; // используя селектор указателя
cout << p->x;
pa->mod();

cout << (*p).x; // либо разыменовывая указатель
(*p).mod();
```

Структуры могут содержать другие структуры.

```
// Структура, задающа точку на плоскости
struct Point
{
   double x;
   double y;
};
// Структура, задающая линию между двумя точками на плоскости
struct Line
{
   Point start;
   Point end;
} line1;
Line *pline = &line; // указатель на объект типа Line
```

Можно даже определять одни структуры внутри других - вложенные структуры (nested).

```
// Структура, задающая линию между двумя точками на плоскости
struct Line
{
   // Структура, задающа точку на плоскости
   struct Point
   {
      double x;
      double y;
   };
   Point start;
   Point end;
} line;
Line *pline = &line; // указатель на один из объектов
```

Структуры, как и перечисления могут быть анонимными. Т.е. тип структуры не получит имени и его нельзя будет использовать в другом месте программы. Обратите внимание на отсутствие названия типов из предыдущего примера 'Line' и 'Point'.

```
// Структура, задающая линию между двумя точками на плоскости
struct
{
   // Структура, задающа точку на плоскости
   struct
   {
      double x;
      double y;
   } start, end;
} line;
Line *pl1 = &line; // указатель на один из объектов
```

Обращения к полям осуществляется с помощью селекторов. Только в слеучае вложенных структур они также становятся вложенными. Например:

```
line.start.x = 0;
pline->end.y = 0;
```

Объединения (union)

Объединения - группы данных, в которых для их хранения используется одно и то же пространство памяти.

```
union Bytes {
  int i;
  float f;
  char c[4];
} bytes;
```

В этом примере bytes.i, bytes.f, bytes.c[4] расположены в одном месте в памяти. Размер членов объединения не обязательно должен совпадать. Объединения удобно использовать для переинтерпретации содержимого памяти, например если требуется побайтная передача данных с последующей интерпретацией полученного потока, или для экономии памяти.

```
#include <iomanip>
using namespace std;
union Bytes {
   int i;
   float f;
};

int main ()
{
   Bytes u;
   u.f = 3.14f;

   cout << hex;
   cout << setiosflags (ios::showbase);
   cout << u.i << endl;
```

```
    return 0;
}
```

Битовые поля

Битовые поля позволяют оперировать структурами, состоящими из отдельных битов.

```
struct Date {
  unsigned int year   : 23;  // 23 бита достаточно для хранения
                             // значений 0-8388607
  unsigned int month  : 4; // 5 бита достаточно для хранения значений 0-15
  unsigned int day    : 5;   // 5 битов достаточно для хранения значений 0-31
};
```

В этой структуре год занимает 23 бита, месяц 4 бита и день 5 битов. В памяти эти биты хранятся последовательно, и вся структура из примера займет 32 бита. Для выравнивания битовое поле может не иметь имени. Неименованное поле 0-й длины приводит к выравниванию до следующей границы, кратной размерности базового типа члена битового поля (в данном случае 'unsigned int'). В качестве базового типа можно указать любой из арифметических фундаментальных типов. Обращение к членам битового поля подчиняется тем же правилам, что и доступ к членам обычных структур.

```
cout << dec << setfill ('0'); // в предыдущем примере вывод переключался на
                              // 16-ричный формат, возвращаем к десятичному

Date date={2014,9,1}; // инициализация при объявлении
cout << setw(2) << date.day << "." << setw(2) << date.month << "." << date.year
<< endl;
// изменение полей объекта
date.year = 2013;
date.month = 8;
date.day = 31;
// изменение полей объекта по указателю на объект
Date *pdate=&date;
pdate->year = 2015;
pdate->month = 1;
pdate->day = 1;
cout << setw(2) << date.day << "." << setw(2) << date.month << "." << date.year
<< endl;
```

Для типов, уже сществующих в программе, можно вводить новые названия, синонимы типов. Для этого есть два механизма: typedef и using. В примерах ниже создаются названия типов Str и DWORD.

```
typedef char* Str;       // тип 'Str' является синонимом 'char*'
using Str=char*;         // то же самое, только более понятным образом

typedef unsigned long int DWORD;
using DWORD=unsigned long int;
```

Далее можно объявлять объекты нового типа:

```
Str s="hello";
```

При декларации типов полезной может оказаться функция типа выражения 'decltype'.

```
int x;
double y;
using Num=decltype(x*y);
Num value1;
decltype(x*y) value2;
```

'decltype' возвращает тип выражения, передаваемого как параметр и может использоваться в декларациях.

Типы данных стандартной библиотеки

Стандартная библиотека C++ (C++ Standard Library) предоставляет набор готовых библиотек, реализующих разнообразные базовые алгоритмы, типы и функции. Основные категории библиотек: * языковая поддержка * диагностика * общие утилиты * строки * локализация * контейнеры * итераторы * алгоритмы * числовая библиотека * ввод/вывод * регулярные выражения * атомарные операции * поддержка поткового исполнения Все имена стандартной библиотеки принадлежат пространству имен std. Для доступа к ним необходимо использовать разрешение области видимости 'std::...' или директиву 'using namespace std;'. Документацию к библиотеке несложно найти самостоятельно. Ниже очень кратко будут упомянуты возможности основных библиотек, которые на наш взгляд с наибольшей вероятностью могут понадобиться при разработке программ моделирования.

Строки (string)

В библиотеке реализован тип для рабты со строками. Библиотека подключается директивой '#include ' в единице трансляции, в которой необходимо использование типов и объектов, определенных в ней.

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

int main ()
{
  string name={"Jekky"}; // тип 'string' определен в <string>
  string family={"Chan"};
  cout << name + " " + family << endl; // конкатенация (сложение) строк

  string str{"my constant string literal"};
  str.replace(str.begin()+3,str.end(),"replaced string"); // метод 'replace'
                                                          // реализован в
                                                         // стандартной
                                                        // библиотеке
  cout << str<< endl;
```

```
  return 0;
}
```

Массив числовых данных (valarray)

Это класс для работы с массивом числовых данных, предоставляющий кроме функций массива как контейнера, методы для работы с его числами: поиск минимального из чисел массива, максимального числа, суммы и т.д. Размерность задается при определении массива.

```
#include <iostream>
#include <valarray>
#include <random>
using namespace std;
int main ()
{
  // Ввод размерности массива
  int n=0;
  cout << "n=";
  cin >> n;
  //Массив из n чисел типа 'int'. Можно использовать другие арифметические
  // типы, для котрых определены операции сложения, вычитания, сравнения.
  valarray<int> intvals (n);

  // Генератор случайных чисел
  default_random_engine engine;
  // Равномерное распределение от 0 до 4 включительно
  uniform_int_distribution<int> distribution(0, 4);

  for (auto &i:intvals)
     i = distribution(engine);

  for (auto i:intvals)
     cout << i << endl;
  cout << "sum=" << intvals.sum() << endl;
  cout << "min=" << intvals.min() << endl;
  cout << "max=" << intvals.max() << endl;
  return 0;
}
```

Массив элементов одного типа (array)

Массив фиксированного размера для работы с любым типом элементов, не только числовыми.

```
#include <iostream>
#include <array>
using namespace std;
int main ()
```

```
{
  array<int,3> intvals;
  intvals[0]=10;
  intvals[1]=20;
  intvals[2]=30;
  for (auto i:intvals)
    cout << i << endl;
  return 0;
}
```

Вектор (vector)

Массив с динамической размерностью, в который можно добавлять и убавлять элементы. Элементы добавляются в конец вектора методом 'push_back()' и удаляются 'pop_back()'. При добалении элемента размер вектора увеличивается на один элемент.

```
#include <iostream>
#include <vector>
using namespace std;
int main ()
{
  vector<int> intvals(3); // вектор из 3-х элементов (равны '0')
  intvals.push_back(10);  // добавление 4-го элемента
  intvals.push_back(20);  // добавление 5-го элемента
  intvals.push_back(30);  // добавление 6-го элемента
  for (auto i:intvals)
    cout << i << endl;
  return 0;
}
```

Двумерный массив заданной размерности

Готовых многомерных массивов в стандартной библиотеке нет, но возможно использовать существующие одномерные массивы для конструирования многомерного.

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

int main ()
{
   // Матрица из MxN чисел (M строк, N столбцов)
   const int M{4},N{3}; // размерности должны быть const или constexpr
   array< array<double,N>, M> matrix; // массив, элементами которого
                                      // являются тоже массивы.
   // Можно создать синоним типа следующим образом
   // using Matrix=array< array<double,N>, M>;
   // Объявление матричного объекта будет выглядеть понятнее:
```

```
    // Matrix matrix

    // Инициализация
    int count=0;
    for (auto &row:matrix)
        for (auto &cell:row)
            cell = count++;

    // Вывод (можно сделать по аналогии с инициализацией с помощью
    // цикла range-for)
    for (int i = 0; i<M;i++)
    {
        for(int j = 0; j<N;j++)
        {
            cout << matrix[i][j] << "\t";
        }
        cout << endl;
    }

    return 0;
}
```

Двумерный массив переменной размерности

Для создания массива переменной размерности следует использовать динамически расширяемый 'vector'.

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

int main ()
{
    // Ввод размерности массива
    int n{0},m{0};
    cout << "Enter dimensions" << endl;
    cout << "n="; cin >> n;
    cout << "m="; cin >> m;

    // Синонимы типов: ряда, и 2-мерного массива
    typedef vector<double> row_in_array2d;
    typedef vector <row_in_array2d> array2d;

    // Матрица из MxN чисел (M строк, N столбцов)
    row_in_array2d row_value(n);
    array2d matrix(m,row_value);
```

```
    // Инициализация
    int count=0;
    for (auto &row:matrix)
        for (auto &cell:row)
            cell = count++;

    // Вывод
    for (int i = 0; i<m;i++)
    {
        for(int j = 0; j<n;j++)
        {
            cout << matrix[i][j] << "\t";
        }
        cout << endl;
    }

    return 0;
}
```

Кортеж (tuple)

Последовательность из нескольких объектов, которые могут иметь различные типы. Стандартный заголовок '<tuple>'.

Двусторонняя очередь (dequeue)

Контейнер с динамически изменяемой размерностью по мере добавления/удаления элементов, позволяющий быстро добавлять и удалять элементы в начало или конец. Поддерживает возможность прямого доступа по индексу. Стандартный заголовок '<deque>'

Двусвязный список (list)

Контейнер с динамически изменяемой размерностью, обеспечивающий быструю вставку/удаление в любом месте. Прямого доступа к элементам не имеет. Реализован как набор элементов, имеющих ссылку на следующий и предыдущий элемент в списке. Стандартный заголовок '<list>'

Односвязный список (forward_list)

Аналогичен двусвязному списку, но поддерживает перебор элементов в одном направлении. Реализован как набор элементов, имеющих ссылку на следующий элемент в списке. Стандартный заголовок '<forward_list>'

Ассоциативный контейнер map (словарь)

'map' состоит из упорядоченных по уникальному ключу пар ключ-значение. 'multimap' - из упорядоченных неуникальных (имеющих возможность повторяться). Стандартный заголовок '<map>'. Неупорядоченный вариант - unordered_map и unordered_multimap из заголовка <unordered_map>.

Ассоциативный контейнер set (множество)

'set' состоит из упорядоченных уникальных значений. 'multiset' - коллекция упорядоченных неуникальных значений. Стандартный заголовок

'<set>'. Неупорядоченный вариант - unordered_set и unordered_multiset из заголовка <unordered_set>.

Очередь (queue)

Представляет собой адаптер контейнера, реализующий логику FIFO - первый элемент, вставляемый в очередь первым из нее извлекается. Стандартный заголовок '<queue>'.

Приоритетная очередь (priority queue)

Адаптер контейнера, предствляющий упорядоченную по приоритетам очередь. Первым извлекается элемент с наибольшим приоритетом. Стандартный заголовок '<queue>'.

Стек (stack)

Представляет собой адаптер контейнера, реализующий логику LIFO (last in first out) - первым извлекается последний вставленный элемент. Стандартный заголовок '<stack>'.

Ввод-вывод, чтение/запись файлов

Язык C++ средствами стандартной библиотеки предоставляет возможности для чтения и записи файлов. Интерфейс библиотеки не зависит от выбора целевой платформы и операционных систем, на которых будет работать программа.

Весь ввод и вывод в языке происходит через входные/выходные потоки - файлы, буферы памяти, консоль в которые операторм '<<' записываются и оператором '>>' считываются последовательности данных. Запись в файл с точки зрения ее описания в программе не отличается от записи в консоль.

```
#include <fstream> // классы для работы с файлами
ofstream file( "output-file.txt" );
file << "output string";

#include <iostream> // классы и определения 'cin', 'cout' для
                    // консольного ввода/вывода
cout << "output string";
```

Для работы с файлами используется заголовочный файл . В нем определены типы 'ofstream' - файл для записи, 'ifstream' - файл для чтения, 'fstream' - файл как для чтения, так и для записи. Для начала работы необходимо создать объект одного из этих типов и открыть его для чтения. #### Чтение текстового файла Открытие текстового файла можно сделать в момент конструирования объекта, указав путь к файлу

```
#include <fstream>
ifstream file( "input-file.txt" ); // открытие файла  при создании объекта
// или
ifstream file; // объект создан, но покак не ассоциирован с каким либо файлом
file.open( "input-file.txt" ); // явное открытие файла вызовом метода
...
file.close(); // при явном открытии нелишним будет так же явно файл
закрыть.
```

// Впрочем, если это не сделано деструктор объекта закроет
// открытый файл.

При открытии файла можно указать режим, в котором он будет открыт. Режим задается следующими значениями перечисления:

- fstream::in (input) файл для чтения. Для ifstream он включен по умолчанию
- fstream::out (output) файл для записи. Для ofstream он включен по умолчанию
- fstream::binary (binary) бинарный файл (данные в файле не интерпретируются как символы, в частности 0x0D 0x0A не означат перевод строки). По умолчанию включен текстовый режим.
- fstream::ate (at end) текущая позиция записи установлена в конец файла. запись осуществляется в нее, но позицию можно поменять
- fstream::app (append) запись осуществляется только в конец файла независимо от текущей позиции записи
- fstream::trunc (truncate) очистка содержимого файла при открытии.

```
#include <fstream>
// открытие бинарного файла для чтения
fstream file( "input-file.txt",  fstream::binary | fstream::in);
```

После открытия файла можно проводить операцию чтения его содержимого аналонично тому, как это происходит при чтении клавиатурного ввода. Код для чтения тектсового файла со значениями смешанного типа - целые, вещественные, строки:

```
1.0 2.0 3.0 11 "str1"
1.1 2.1 3.1 22 str2
1.2 2.2 3.2 33 "str3"
```

...может выглядеть следующим образом:

```
#include <fstream>
#include <string>      // в коде используется тип 'string'
#include <iomanip> // требуется для форматирования отладочного вывода


ifstream file("dat/example-005-in.txt"); // файл лежит в подкаталоге 'dat/'

// структура данных, соответствующая записи в файле
struct Record
{
   double p,q,r;
   int i;
   string s;
} rec;

// проверка успешного открытия файла
if(file)
```

```
{
   // чтение файла до конца записей
   while( file >> rec.p >> rec.q >> rec.r >> rec.i >> rec.s )
   {
      // отладочный вывод считанных значений на консоль
      cout << setprecision(2) << fixed;
      cout << "p=" << rec.p;
      cout << " q=" << rec.q;
      cout << " r=" << rec.r;
      cout << " i=" << rec.i;
      cout << " s=" << rec.s << endl;
   }
}
```

Внимание: строки в файле должны соответствовать предполагаемому формату записей. В данном случае: три вещественных числа, одно целое и строка.

Чтение текстового файла может осуществляться функцией 'getline()' построчно с последующим разбором строки на отдельные элементы. Образец файла:

```
1.0 01/09/2014 11 "str1"
1.1 02/09/2014 22 "str2"
1.2 03/09/2014 33 "str3"
struct Record
{
   float p;
   int day,month,year;
   int i;
   char c[256]; // sscanf, используемый в этом примере не
                // поддерживает тип 'string'
} rec;
string str;
while( getline(file,str) )
{
   cout << "line content " << str << endl;

   // Существуют разные способы. Например с вызовом 'sscanf()'
   sscanf(str.c_str(), "%g %2d/%2d/%4d %d %s",
         &rec.p,&rec.day,&rec.month,&rec.year,&rec.i, rec.c);
   cout << "parsed=" << rec.p<< " "<<rec.day<< " "<<rec.month<<
         " "<<rec.year<< " "<<rec.i << " "<<rec.c << endl;
}
```

В контесте разбора строки на элементы стоит обратить внимание на регулярные выражения, также входящие в состав стандартной библиотеки, правда на момент написания этого документа не везде и не полностью еще

реализованные. Регулярные выражения выходят за рамки этого документа. Информацию без труда вы сможете найти в Интернет.

Для различных преобразований существуют группы функций, определенных в: std::to_string - конвертирование числовых типов в строки (string str = to_string(3.14);) std::stoi, std::stol, std::stoll - конвертирование строки в целое соотвествующей имени функции длины (int i = stoi("3");) std::stof, std::stod, std::stold - конвертирование строки в вещественное число соотвествующей имени функции длины (double d = stod("3.14");)

Запись в текстовый файл

Для записиси в файл его также необходимо открыть на запись.

```
#include <fstream>
// открытие бинарного файла для записи (для примера,
// одновременно установлен флаг и для чтения также)
fstream file( "input-output-file.txt",  fstream::binary | fstream::out | fstream::in);
//или
ofstream file( "output-file.txt"); // открытие текстового файла для записи
```

Запись в файл осуществляется оператором '<<'. Для форматирования вывода можно воспользоваться манипуляторами формата из заголовка

```
#include <fstream>
#include <string>
#include <iomanip>

ofstream file("dat/example-005-out.txt"); // файл лежит в подкаталоге 'dat/'.
// Файл будет перезаписываться при каждом запуск программы.
// Используйте режим fstream::app для добавления в конец
if(file)
{
   struct Record
   {
      float p;
      int i;
      string s;
   } rec;
   rec.p = 3.14;
   rec.s = "record No";

   for(int i=0; i<10; i++)
   {
      rec.i = i;
      rec.p *= i + 1;
      file << rec.s << setw(2) << setfill('_') << rec.i <<
         ": floating number=" << scientific << rec.p << endl;
   }
}
```

В примере используются манипуляторы 'setw', 'setfill' и флаг 'scientific'. Манипуляторы действуют до своей отмены кроме манипулятора 'setw', который влияет только на следующий вывод в поток. Список доступных манипуляторов:

• setbase - установить основание системы счисления

cout << setbase(16); // то же самое: 'setbase(hex)' - шестнадцетиричное число
cout << setbase(10); // то же самое: 'setbase(dec)' - десятичное число
cout << setbase(8); // то же самое: 'setbase(oct)' - восьмеричное число

• setprecision - установить точность

cout << setprecision(4); // число значащих цифр в вещественном числе

• setw - установить ширину поля

cout << setw(10); // установить ширину поля следующего вывода в поток

• setfill - установить символ для заполнения полей

cout << setfill('0'); // заполнение нулями до ширины поля

• get_money - считать денежное значение
• put_money - записать денежное значение

long double price;
cin >> get_money(price);

• get_time - прочесть значение времени
• put_time - записать значение времени

tm now; // структура 'tm' определена в <ctime>
cin >> get_time(&now);

• setiosflags/resetiosflags - установить и сбросить флаги форматов. Пример:

cout << setiosflags (scientific | uppercase);

Список и назначение флагов:

* boolalpha - прочитать/записать булевское значение как текст 'true' или 'false'
* showbase - при записи указать основание системы счисления числа (например, 0xAFCD)
* showpoint - при записи в вещественных числах всегда ставить точку
* showpos - при записи у неотрицательных чисел всегда ставить '+'
* skipws - пропускать пробелы при вводе
* unitbuf - производить вывод данных буфера после каждой записи в поток
* uppercase - при записи выводить строки прописными символами
* dec - чтение/запись в десятичном формате
* hex - чтение/запись в шестнадцатеричном формате
* oct - чтение/запись в восьмеричном формате
* fixed - запись вещественных чисел в формате с фиксированной точкой (0.0001)
* scientific - запись вещественных чисел в формате с фиксированной точкой (1E-4)
* internal - заполнение символом, заданным 'setfill', внутри поля (-***1)
* left - выравнивание по левой кромке поля (-1***)
* right - выравнивание по правой кромке поля (***-1)

Генерация случайных чисел

Генерацию (псевдо-)случайных чисел производит некоторый алгоритм. Простой алгоритм может выглядеть так:

```
uint32_t rnd()
{
   static uint32_t x = 1; // инициализация начального значения. Объект
                          // объявлен static - он виден только внутри
                          // функции, но существует все время работы
                          // программы, не только при входе в функцию.
   return x = 64525*x + 1013904223; // следующее значение вычисляется на
                          // основе предыдущего. Значением
                          // оператора присваивания является
                          // присваиваемое значение. Его то и
                          // возвращает функция.
}
```

Пример использования алгоритма:

```
for(int i=0; i< 10;i++)
   cout << rnd() << endl;
```

Так вот, стандартная библиотека предоставляет генераторы псевдо-случайных чисел, использующие разные алгоритмы генерации (Вихрь Мерсенна, линейный конгруэнтный генератор, разновидность метода Фибоначчи с запаздываниями - subtract with carry).

Различные генераторы на основе перечисленных алгоритмов и их модификаций:

- default_random_engine - линейный конгруэнтный генератор
- minstd_rand - линейный конгруэнтный генератор вида 'x = x * 48271 % 2147483647'
- minstd_rand0 - линейный конгруэнтный генератор 'x = x * 16807 % 2147483647'
- mt19937 - генератор Вихрь Мерсенна 32-битных чисел с 19937-битным состоянием
- mt19937_64 - генератор Вихрь Мерсенна 64-битных чисел с 19937-битным состоянием
- ranlux24_base - метод Фибоначчи по основанию 24
- ranlux48_base - метод Фибоначчи по основанию 48
- ranlux24 - модификация прдыдущего варианта по основанию 24 с отбрасыванием блока
- ranlux48 - модификация прдыдущего варианта по основанию 48 с отбрасыванием блока
- knuth_b - модификация линейного конгруэнтного генератора с перемешиванием

И, наконец, недетерминированный генератор

- random_device - использует устройства компьютера для генерации непсевдослучайной последовательности. Доступен не на всех платформах и

генерирует исключение во время объявления, если такого устройства на платформе нет.

Использоваться генераторы могут так же, как и в самодельном примере:

```
#include <random>

minstd_rand0 rnd(1); // 1 - инициализатор псевдослучайной последовательности

for(int i=0; i< 10;i++)
   cout << rnd() << endl;
```

Упражнение: посчитать количество уникальных значений в псевдослучайной последовательности. Определить период.

Используя генератор можно создавать последовательности с различным распределением (например, распределение суммы равномерно распределенных случайных величин приближается к нормальному распределению с ростом числа слагаемых). Распределения, который поддерживает стандартная библиотека:

Равномерные распределения:

- uniform_int_distribution - равномерное целочисленное распределение
- uniform_real_distribution - равномерное распределение для вещественных чисел

Семейство распределений Бернулли:

- bernoulli_distribution - распределение Бернулли
- binomial_distribution - биномиальное распределение
- geometric_distribution - геометрическое распределение
- negative_binomial_distribution - отрицательное биномиальное распределение

Частотные распределения:

- poisson_distribution - распределение Пуассона
- exponential_distribution - экспоненциальное распределение
- gamma_distribution - гамма-распределение
- weibull_distribution - распределение Вейбулла
- extreme_value_distribution - распределение экстремальной величины

Нормальные распределения:

- normal_distribution - нормальное распределение
- lognormal_distribution - логонормальное распределение
- chi_squared_distribution - распределение хи-квадрат
- cauchy_distribution - распределение Коши
- fisher_f_distribution - распределение Фишера
- student__distribution - распределение Стьюдента

Дискретные распределения:

- discrete_distribution - дискретное распределение
- piecewise_constant_distribution - дискретное равномерное распределение
- piecewise_linear_distribution дискретное линейное распределение

Распределение использует тот или иной генератор случайных чисел. Использование на примере равномерного распределения вещественных чисел типа 'double' в диапазоне [-1.0;1.0] показано ниже:

```
default_random_engine rnd(1);
//равномерное распределение от -1.0 до 1.0
uniform_real_distribution<double> distribution (0.0,1.0);

for (int i=0; i<10; i++)
   cout << distribution(rnd) << std::endl;
```

Область видимости (scope)

Объекты и функции должны быть объявлены до их первого использования в программе. После объявления объект может быть использован в пределах своей *видимости*. Объект может иметь глобальную или локальную область видимости. Глобальные объекты, которые могут быть использованы в любом месте программы, объявляются вне блоков кода и функций в то время как локальные объекты объявляются внутри. Область видимости локальных объектов ограничена пространством внутри фигурных скобок ({}), в которых они объявлены. Таким пространством могут служить блок кода, функция, класс, пространство имен.

```
// глобальные объекты (переменные)
A aobj; // объект доступен в любой точке кода после своего объявления
double x;
...
int main()
{
  //локальные объекты
  double y;
  if(x!=0 || y!=0) // 'x' инициализирован 0, а 'y' содержит
                   // случайное значение
  {
    int i; // локальная переменная может использоваться только
           // внутри блока if
    aobj.print(); // использование глобальной переменной

  }
  ...
}
```

Глобальные переменные фундаментальных типов инициализируются значением 0, для объектов классов вызывается указанный при определении конструктор или конструктор по умолчанию, инициализирующие объект заданным образом.

Классы памяти (storage duration)

Объекты размещаются в памяти в соответствии со своими классами памяти: *automatic* - объект существует внутри блока, и удаляется при выходе из

него. Размещение происходит в стеке программмы. *static* - объект размещается при запуске программы и удаляется из памяти при её завершении. Размещение происходит в BSS сегменте памяти программы. *thread* - объект размещается при запуске потока и удаляется из памяти при его завершении. *dynamic* - объект размещается и удаляется при вызове функций динамической памяти. Размещение происходит в свободной динамической памяти программы (heap)

```
A a1; //  static - глобальный объект, создаваемый при запуске
      // прооограммы в момент "до его первого использования"
static A a2; // static - то же самое, только область видимости
             // ограничена данным модулем трансляции
int main()
{
   {
      A a2;          // automatic - объект создается в момент
                     // объявления и существует до конца исполнения
                     // блока или функции
   } //здесь объект 'a' будет разрушен идалее он недоступен
   {
      static A a3;  // static - объект создается при запуске
                     // программы, инициализируется конструктором по
                     // умолчанию (переменные фундаментальных типов
                     // инициализируются 0), существует постоянно до
                     // завершения программы, доступ данном случае
                     // только локальный т.к. объект объявлен в блоке
      extern A a4;// объявление, не приводящее к размещению
                     // в памяти.
      register A a5; // automatic - объект существует в области
                     // видимости, по возможности в регистровой
                     // памяти процессора
      thread_local A a6;  // thread - объект существует во время исполнения
                     // потока. здесь поток один - основной поток,
                     // в котором выполняется main()
   }
}
```

Объекты могут быть созданы в любой момент в динамической памяти глобальными функциями размещения

```
void* operator new(std::size_t);
void* operator new[](std::size_t);
```

и удалены соответствующими им функциями удаления:

```
void operator delete(void*);
void operator delete[](void*);
```

Пример

```
int main()
{
```

```
   A *a = new A; // создание объекта в памяти.
   // Можно написать A *a = new A(); - в обоих слуаях будет вызван
   // конструктор по умолчанию 'A()'.

   // ... использование объекта
   delete a; // удаление объекта из памяти. Дальнейшее использование
         // невозможно несмотря на то, что укаазатель 'a' продолжает
         // хранить адрес. Удаление строго обязательно!
}
```

В случае automatic, static и thread компилятор самостоятельно создаст и удалит такие объекты по перечисленным выше правилам. В случае динамического размещения, забота о своевременном создании и удалении объектов лежит на разработчике программы, которому придется решить две возникающие проблемы: неудаление созданных и больше не используемых объектов, приводящее к непроизводительному задействованию ресурсов, например, утечкее памяти, и несвоевременное удаление объектов, которые еще используются в программе. Для избежания таких ситуаций существуют интеллектуальные, или умные, указатели (smart pointers). Такие указатели реализованы как классы, точнее шаблоны классов, и обеспечивают своевременное автоматческое удаление объекта без явного вызова 'delete'. Интеллектуальные указатели упрощают работу программиста и страхуют от ошибок. Код программы выглядит как в языках со сбором мусора (garbage collection) - процессом автоматического уничтожения неиспользуемых объектов.

Предыдущий пример с обязательным удалением указателя с применением интеллектуального указателя 'unique_ptr' становится существенно более лаконичным:

```
#include <memory> // заголовок стандартной библиотеки необходим для
                  // использования интеллектуальных указателей
int main()
{
   unique_ptr<A> a(new A);// создание объекта в памяти и передача
                          // полученного указателя в интеллектуальный
                          // указатель
   // ... использование объекта точно такое же, как если бы 'a' был простым
   // указателем. Тут может быть столько кода, что автор может забыть, что
   // ему надо что-то удалять
   // А удалять объект и не требуется. Интеллектуальный указатель
   // удалит его сам
}
```

Уникальные указатели не могут быть скопированы (на то они и уникальные), но могут быть переданы с помощью перемещения.

Разделяемый указатель 'shared_ptr' содержат помимо собственно указателя счетчик своих копий (да, shared_ptr можно и нужно копировать, он же "shared"). Этот указатель хранит счетчик своих копий. Каждая новая кпия увеличивает счетчик на единицу. Объект, на который ссылается указатель удаляется из памяти, когда счетчик копий становится равным нулю.

Слабо связанные указатели 'weak_ptr' можно рассматривать как разновидность 'shared_ptr', которая не увеличивает счетчик. Его особенность в том, что 'weak_ptr' с помощью метода 'expired()' позволяет проверить, не был ли удален объект, на который указывает этот указатель. Может также использоваться для разрыва циклических или взаимных ссылок между указателями 'shared_ptr'.

Управление выполнением

В языках, родственных процедурным, исполнение программы происходит последовательно. Это не универсальное правило, но как правило это так и направление исполнения совпадает с последовательностью чтения строк программмы сверху вниз. Отчасти это соответствует архитектуре процессора, в которой (у каждого ядра) есть счетчик инструкций (progrramm counter, PC), автоматически увеличиваемый процессором при исполнении каждой процессорной инструкции. Исполнение кода происходит условно следующим образом:

```
uint32_t pc= main; // начало исполнения с функции main, счетчик
                   // указывает на начало
while(true) // бесконечный цикл
  execute(*pc++); // исполнение текущей инструкции и увеличение счетчика
          // инструкций.
          // ('*pc' - разыменование указателя, т.е.чтение инструкции
          // по адресу, который он содержит; 'pc++' увеличение адреса
          // на единницу - указатель на следующую инструкцию)
```

Примеры инструкций процессора с системой команд x86:

```
; Сложение
ADD r0, r1, r2  ; r0 = r1 + r2
; Вычитание
SUB r0, r1, r2  ; r0 = r1 — r2
; Умножение
MUL r0, r1, r2  ; r0 = r1 * r2
; Копирование
MOV r0, r1      ; r0 = r1
; Логическое ИЛИ
ORR r0, r1, r2  ; r0 = r1 | r2
; Загрузка
LDR r4, [r5]    ; r4 = *r5
; Сохранение
STR r4, [r5]    ; *r5 = r4
```

; Переход исполнения на соответствующий адрес
B label

Компилятор из исхдного текста выстраивает эти и другие инструкции процессора в последовательность в памяти. Процессор читает и выполняет эту последовательность в потоке исполнения (thread), исполняемый одним ядром.

Последовательностью исполнения кода можно управлять с помощью управляющих операторов (statements), обеспечивающих ветвление и управление циклами. Также исполнение можно разбивать на блоки кода, вызываемые по имени - функции. Определение функции выглядит следующим образом:

<тип возвращаемого значения. если значение не возвращается, то тип 'void'>
<имя функции в соответствии с правилами именования идентификаторов>
(разделенный запятыми список формальных параметров с их типами.
Параметров у функции может не быть)
{
... // операторы тела функции
}

Вызов функции:

<имя функции>(список фактических параметров);

Примеры:

```
int f(int x, int);          // f функция принимающая 2 int, возвращающая int
void f();                   // f функция без параметров
void f(int a=0);            // Значение параметра по умолчанию f()
                            // эквивалентно f(0)
inline void f();            // Развернуть код функции в месте вызова
T f() { statements; }       // определение функции, возвращающей T
T operator+(T x, T y);      // бинарный оператор сложения x+y
T operator-(T x);           // унарный оператор. '-x' вызывает 'operator-(x)'
extern "C" {void f();}      // объявление f(), которая была скомпилирована на C
```

Функция должна быть объявлена или определена перед использованием. Функции с одинаковым именем, но различными параметрами являются разными функциями и выбираются в зависимости от параметров (перегружаются). Операторы за исключением "::" "." ".*" "?:" могут быть перегружены. Приоритетность при этом не изменяется. Новые операторы созданы быть не могут.

Объявление функции должно предшествовать ее использованию (вызову). Напоминаем, что определение также является объявлением.

Пример:

```
// Определение функции sinus
double sinus(double x)
{
  ...
  return retval; // возвращаемое значение должно было быть вычислено
                 // в теле функции
```

```
}
// Определение функции pow(), возвращающей степень первого параметра/
// Второй параметр - значение степени.
double pow(double x, int n)
{
...
   return retval;
}
// определение функции извлечения квадратного корня
double squareroot(double x)
{
...
return retval;
}
// определение функции print
void print()
{
  cout << "hello world";
}
// Вызов функции
double x=sinus(0.1);
double y=pow(x,2);
print();
```

При передаче параметров можно использовать как их значение так и ссылки или указатели. При передаче по значению фактический параметр, существующий в вызывающем коде, копируется в формальный параметр. Поэтому все иизменения параметров внутри функции не отражаются в вызывающей программе. При передаче по значению в прототипе функции указаны типы параметров.

```
void f(A a) // параметр можно назвать любым именем на свой вкус. Как его
            // назовете, так и придется к нему обращаться в теле функции.
            // Здесь 'A a' является определением локального для функции
            // объекта
{
  a.x = -1.0;
}

A a{0,0};
f(a);
cout << a.x; // 0.0
```

При передаче по ссылке значения не копируются, в функцию передаются ссылки на параметры. Следствием этого является возможность внутри функции параметры изменить и увидеть эти изменения в том месте, откуда функция была вызвана.

```
void f(A &a) // передача 'a' по ссылке
{
  a.x = -1.0;
}
A a{0,0};
f(a);
cout << a.x; // -1.0
```

При передаче указателя в функцию происходит копирование значения этого указателя, но не объекта, на который он указыает. Поэтому объект из функции можно изменить, но при этом значение самого указателя - нет.

```
void f(A *pa) // передача указателя 'pa'
{
  pa->x = -1.0;
  pa=nullptr; // не имеет эффекта за пределами функции 'f'
}
A a{0,0};
f(&a); // передача адреса 'a' в функцию. оператор '&' - взятие адреса объекта
cout << a.x; // -1.0
```

С функциями связан еще один тип данных - указатели на функции. Такие указатели в отличие от указателей на данные указывают на адрес функции в памяти, содержащей код программмы. Указатели на функции, как правило, используются для передачи в качестве параметра в другие функции и могут быть полезны, если требуется динамическое изменение выполнения программы: в указателях можно передавать различные функции в зависимости от обстоятельств. Например, в программу рисования графиков, при едином алгоритме рисования можно в форме указателей на функцию передавать способ расчета значений по шкале ординат.

Объявляются указатели на функции как и указатели на данные с помощью '*', хотя и несколько необычно:

```
double (*trigonometric)(double); // объявление функции 'double
trigonometric(double);'
trigonometric = sin; // Теперь указатель указывает на функцию 'sin'.
                     // 'sin' объявляется в заголовке <cmath>. Также можно
                     // получить адрес с помощью операции взятия адреса:
                     // 'trigonometric = &sin;'. Оба способа равнозначны.
```

В объявлении указателя на функцию указываются возвращаемое значение и список параметров (в примере оба - 'double'), а также имя указателя - 'trigonometric'. Вызов осуществляется обычным образом:

```
double x={1.0}, y;
y = trigonometric(x); // Вызов функции. Также функцию по указателю можно
                      // вызвать вначале его разыменовав: (*trigonometric)(x)
cout << "sin(1.0)=" << y << endl;
```

Функции внутри функций объявлять и определять нельзя. Зато функции могут быть объявлены и принадлежать классу (методы класса), пространству имен в т.ч. глобальному ::f(), единице трансляции (static).

```
#include <iostream>

class A
{
   double x,y;
   public:
   double mod(); // объявление метода класса A::mod()
};
// определение метода класса A::mod()
double A::mod()
{
   return squareroot(x*x+y*y);
}

// определение глобальной функции
double mod(A &a)
{
   return squareroot(a.x*a.x+a.y*a.y)
}

// определение статической (локальной для данной единицы трансляции)
// функции. Вызвать эту функцию из другого файла не получится т.к. она
// объявлена  'static'.
static double smod(A &a)
{
   return squareroot(a.x*a.x+a.y*a.y)
}

namespace example_space
{
   // определение функции внутри пространства имен
   double mod(A &a)
   {
      return squareroot(a.x*a.x+a.y*a.y)
   }
}

int main()
{
   A a{0,0};
   cout << a.mod() << endl;   // использование (вызов) метода класса
```

```
   cout << mod(a) << endl;   // использование глобальной функции
   cout << smod(a) << endl; // использование статической функции
   cout << example_space::mod(a) << endl; // использование функции из
                                          // пространства имен
}
```

При этом язык предоставляет возможность создания объектов-функций -- функторов, лямбда-функций и замыканий.

Функтор

В классе может быть определен оператор '()'.

```
class A
{
  double x,y;
public:
 double mod();
  double operator()(int pow)
  {
   double retval = mod();
    for(int i=0;i<pow;i++)
      retwal *= retwal; // умножение величины на само себя
  }
  return retval;
};

// пример использования функтора
int main()
{
 A a{1,1};
  cout << a(2) << endl; // вызов функтора
}
```

Лямбда-функция

Лямбда-функция (lambda, lambda function, lambda expression) - это еще один механизм создания функторов. Рассмотрим пример:

```
auto f = [](double x, int i){ return power(x,i);};
```

где

- 'auto f' -- объявление объекта-лямбда-функции. Тип, характеризуемый списком параметров, типом возвращаемого значения и телом лямбда-функции выводится автоматически.
- [] -- список захвата локальных переменных. (переменных из области видимости) если список пуст '[]', то переменнные не захватываются и в лямбде недоступны если список выглядит так: '[&]', то захватываются все используемые в лябде локальные переменные по ссылке если список выглядит так: '[=]', то захватываются все используемые в лябде локальные переменные по значению Можно указать способ захвата для отдельных локальных объектов: [&x,=y] -- 'x' захвачен по ссылке, 'y' по значению. Остальные объекты не

захвачены. [&x,y] -- то же самое. Если символ '&' перед переменной не указан, то захват происходит по значению. [&,=x,=y] -- 'x' и 'y' захвачены по значению. Остальные объекты захвачены по ссылке. [=,&x,&y] -- 'x' и 'y' захвачены по ссылке. Остальные объекты захвачены по значению.

Оператор условного исполнения

Оператор предназначен для исполнения того или иного блока кода в зависимости от условия

```
if (x) a;              // Если выражение 'x' равно true (или отлично
                       // от 0), выполнить оператор 'a'.
else if (y) b;         // иначе если выражение 'y' равно true,
                       // выполнить оператор 'b'
else c;                // если ни одно из условий 'x' и 'y'
                       // не выполнено
... // продолжение исполнения кода
```

Оператор 'a' и 'b' могут быть составными, т.е. состоять из одного или нескольких операторов, заключенных в фигурные скобки

```
{                  // открывающая скобка блока, составного оператора
 A a;              // можно определять объекты,'a' видима внутри блока
 a.sin();          // определения должны предшествоватьих использованию
}
```

Размещение операторов в строках исходного текста не определено строго и зависит от принятого стиля оформления программы. Примеры разных стилей:

```
if( x>0 ){
  y = 0;
}
else {
  y = x;
}

if( x>0 )
{
  y = 0;
}
else
{
  y = x;
}

if( x>0 )
  y = 0;
else
  y = x;
```

Программист вправе руководствоваться собственным вкусом при выборе форматирования. Вариант стиля кода можно посмотреть здесь: [http://google-

styleguide.googlecode.com/svn/trunk/cppguide.xml] Согласно этому стилю рекомендуется писать так:

```
if (condition) {  // no spaces inside parentheses
  ...             // 2 space indent.
} else if (...) {  // The else goes on the same line as the closing brace.
  ...
} else {
  ...
}
```

Циклы

Цикл while

Это цикл, который выполняется пока условие (выражение), указанное в круглых скобках оператора while, истинно (если быть точным и откровенным, то ради поддержки конструкций языка 'C', не имевшего типа bool, пока условие не равно нулю).

```
while (x) a;          // Повторить 0 и более раз пока x равен true
```

Цикл используется в случаях, когда число повторений заранее неизвестно и может быть нулевым. Пример чтения файла.

```
#include <iostream>
#include <fstream>

using namespace std;

ifstream file("dat/example-001-in.txt"); // файл для чтения
while( file.good() ) // проверка достижения конца файла
{
   wchar_t wc = file.get();
   wcout << wc; // вывод содержимого на консоль ('wcout' - аналог 'cout'
                // для символов типа 'wchar_t')
}
```

Цикл do-while

Цикл также выполняется до тех пор, пока условие истинно. Его отличие от 'while' в том, что этот цикл будет выполнен минимум один раз.

```
do a; while (x);      // Повторить 1 и более раз пока x равен true
                      // Эквивалентно:: a; while(x) a;
```

Пример подскажет жизнь.

Цикл for

Самый распространенный способ сделать цикл - использовать оператор 'for'. Оператор внутри себя использует три выражения.

```
for (x; y; z) a;      // Эквивалентно: x; while(y) {a; z;}
```

Очень часто, хотя и не всегда, оператор используется для циклов со счетчиком или циклов с итераторами (объектами доступа к элементам контейнера)

```
// пример со счетчиком
for(int i = 5; i > 0; i--)
```

```
   cout << i << " ";// здесь может быть составной оператор,
                    // обрамленный фигурными скобками
cout << endl;       // перевод строки выполняется уже после завершения цикла

// пример с итератором
vector<int> v{1,2,3,4,5}; // в данном случае контейнером является 'vector'
for(auto it=v.begin();it != v.end(); ++it)
   cout << *it << " ";
cout << endl;
```

Цикл по диапазону (range based for)

Удобной заменой цикла с итератором является цикл по диапазону, появившийся в C++11.

```
for (x: y) a; // x - переменная цикла, означающая элемент контейнера;
              // y - контейнер
//  range-for со стандартным контейнером vector
for(auto v: vv)
   cout << v;
cout << endl;

// range-for с обычным массивом
int cc[] = {1,2,3,4,5};
for(auto &c: cc) c = 0; // если использовать ссыки то можно мерять
                        // значение переменной цикла
for(int &c: cc) c = 1;  // то же самое, только явно указан тип элемента

// range-for для заданного списка значений
for (auto c: {1,2,3,4,5}) cout << c;
cout << endl;
```

Оператор 'continue' передает управление в конец цикла. Оператор 'break' завершает исполнение цикла.

```
while (true)
{
char c = file.get();
if( !file.good() ) // ошибка чтения из файла
  break;
if (c >= '0' && c <= '9') // пропустить обработку цифр
 continue;

cout << c;
}
while(true) continue; // просто бесконечный цикл.То же что и while(1);
                      // или for(;;); только более наглядно.
```

Оператор switch

Используется для выбора среди большого числа вариантов, например при обработк параметров командной строки, в которой может использоваться множество разнообразных ключей.

```
switch (x) {              // x должен быть целочисленным
                          // (т.е. char, enum, bool тоже подойдут)
 case X1: a; break;  // если x == X1 (константа), выполнить a
 case X2: b; break;  // иначе если x == X2, выполнить b
 default: c;              // если ни одно условие не выполнено (опционально)
}
```

Оператор break здесь необязателен. Он используется для перехода в конец оператора switch:

```
switch(x)
{
case '0':
case '1':
case '2':
case '3':
case '4':
case '5':
case '6':
case '7':
case '8':
case '9':
 ...// обработка цифровых символов
break;
default:
...// обработка всех прочих символов
}
```

Оператор goto

Пользоваться этим оператором категорически не рекомендуется. Тем не менее он есть.

```
some_label:
...
goto some label;
```

Возврат из функции

Оператор возвращает управление из функции в вызывающий контекст, то есть туда, откуда функция была вызвана и, если функция возвращает значение, передает его. Возможно использование нескольких равноценных форм оператора - со скобками и без.

```
return x;   // x - любое выражение с типом совпадающим с типом
            // возвращаемого значения функции
return (x); // то же самое
```

Обработка исключений

Во время работы программы могут возникать различные ошибки, которые необходимо корректно обработать. Есть несколько способов в вызываемом методе класса или функции сообщить о возникновении ошибки: изменить глобальный объект или поле класса, видимый из других частей программы; вернуть в возвращаемом значении код успешности операции (так функция main() возвращает '0' в случае успешного завершения и значение, отличное от нуля, в случае возникновения той или иной ошибки). Оба способа не позволяют прервать исполнение кода после возникновения ошибки. Возможность прерывания исполнения предоставляют исключения (exeptions).

```
try
{
   ...;  // в блоке может быть создано исключение, вызвав 'throw'
}
catch (T t)
{
   ...; // операторы обработки исключения если создано исключение типа T
}
catch (S t)
{
   ...; // операторы обработки исключения если создано исключение типа S
}
catch (...)
{
   ...; // во всех остальных случаях
}
```

Исключение создает функция 'throw'. В момент исключения исполнение кода прерывается и продолжается в соответствующем обработчике (блоке 'catch').

В случае возникновения ошибки во время выполнения конструктора исключение корректным и удобным способом ее обработать. Использовать исключения допустимо не только для обработки ошибок, но и для нормального прерывания выполнения функции, например, функции ввода при вводе пользователем определенной команды.

Пример использования стандартного исключения для обработки ошибки, предоставляемого стандартной библиотекой в заголовке

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

int main()
{
   try {
      cout << "Starting try block" << endl;
```

```
        throw runtime_error("any text here: test exception");
        cout << "Ending try block" << endl;
    } catch (const exception& e) {
        cout << "Exception: " << e.what() << endl;
    }
}
```

В качестве исключений можно использовать объект любого класса, например фундаментального типа 'int'. Можно использовать класс, наследующий классу 'exception' стандартной библиотеки или можно использовать любой другой класс. Объект-исключение можно создавать оператором 'new' в динамической памяти. Важно не забыть удалить его в обработчике.

```
#include <exception>
class E: public exception
{
    virtual const char* what() const throw()
    {
        return "another exception";
    }
} ;

try
{
    // выберите любой из трех вариантов создать исключение,
    // перечисленных ниже:
    //throw 1;
    //throw E();
    //throw new E;
}
catch (int& e)
{
    cout << "integer as an exception" << e << endl;
}
catch (exception& e)
{
    cout << "standard exception" << e.what() << endl;
}
catch (exception* e)
{
    cout << "exception object pointer " << e->what() << endl;
    delete e;
}
```

При объявлении функций и методов после ее сигнатуры (типа возвращаемого значения, имени и списка параметров) можно указать тип исключения, которое может создвать функция

```
void f(); // может инициировать исключения любого типа
void f() throw(int, runtime_error); // может создать 'int'
                    // или 'runtime_error'
void f() throw(); // пустой список - не инициирует исключений
void f() throw; // если за сигнатурой функции не указан ни один тип
                // исключения, функция не генерирует новые исключения, но
                // может передавать дальше исключения, полученные от
                // вызываемых ею функций.
```

Объектно-ориентированное программирование

Объявление и определение классов

Класс (class) - составной тип данных, содержащий поля и методы. Поля и методы могут использоваться извне класса, например, при вызове методов в том месте, где класс был создан. Поля и методы класса также могут использоваться внутри самого класса в его методах. Для ограничения доступа к внутреннему устройству класса извне существуют модификаторы доступа:

- 'public' - поля и методы доступны извне класса
- 'private' - поля и методы закрыты извне и доступны только внутри класса
- 'protected' - аналогично предыдущему, поля и методы закрыты для доступа извне (есть отличия при наследовании классов, которые будут описаны в соответствующем разделе)

Внутри класса все его содержимое, и поля и методы, всегда доступны. Т.е. метод может обращаться к любым данным и другим методам своего класса независимо от их модификатора доступа. Класс не отличается от структуры (struct) ничем, кроме того что по умолчанию в структуре все члены открыты (public), а в классе все члены закрыты (private).

Для того чтобы в порядке исключения дать доступ к закрытым или защищенным членам класса определенным другим классам или внешним функциям можно указать их как дружественные. Это делается с помощью ключевого слова 'friend'.

```
class C
{
   ...
   friend double square(double x); //функция теперь может использовать
                                   // закрытые и защищенные члены объектов
                                   // класса C
   friend class D; // класс D получил доступ к закрытым и защищенным
                   // членам объектов класса C
   friend int E::mod(); // дружественный метод класса
   ...
};
```

Раздел 'public', 'private', или 'protected' класса, в котором сделаны объявления дружественности роли не играет.

Определение полей (данных) класса происходит так же как и определение любых объектов , но делается это внутри класса:

```
//c.h
class C
{//по умолчанию все поля закрыты
public:
  double x,y; //открытые поля класса
protected:
  string description; //защищенные поля класса
private:
  double x2,y2; //закрытые поля класса
};
```

Доступ к полям осуществляется с помощью селекторов доступа '.' для объекта класса или '->' для указателя на объект класса.

```
//main.cpp
#include "c.h"
int main()
{
  C obj, pobj={&obj};
  obj.x = 0.0; // Изменение открытых полей
  obj.y = 0.0;

  pobj->x = 1.0; // Опа, снова поменяли
  pobj->y = 1.0;

  obj.x2 = 0; // Ошибка. Доступ закрыт.
}
```

Объявление и определение методов класса происходит подобно объявлению и определению функций, но только внутри класса

```
//c.h
class C
{
public:
  double x,y;      // открытые поля из предыдущего примера
  double mod(); // открытый метод класса. здесь только объявление,
                // определение метода (тело функции) сделано отдельно, хотя
                // можно было бы определить здесь
protected:
  string description; // метод 'mod()' имеет доступ к этому полю
private:
```

```
  double x2,y2; // и к этим полям тоже. а извне к ним доступа нет
};
```

Метод, объявленный в классе, но не определенный там же, должен быть определен отдельно. Определение вне класса дается аналогично определению функции, но добавляется оператор разрешения доступа '::', указывающий на то, что метод определен внутри класса. Имя класса указывается в определении.

Обычно объявления классов выделяются в заголовочные файлы. Это позволяет использовать имя класса во всех файлах реализации, в которых включен данный заголовочный файл. А определение методов, которое не может дублироваться, производится единожды в файле реализации.

```
//c.cpp
#include "c.h" // включение заголовка для использования объявлений класса 'C'

double C::mod() // определение метода <имя класса>::<имя метода>
{
  x2 = x*x; // внутри методов класса доступ к другим полям и методам
            // осуществляется без селектора доступа. Здесь это
            // иллюстрирует запись значений в 'x2' и 'y2'.
  y2 = y*y;
  return( squareroot(x2+y2) );
}

//main.cpp
#include "c.h" // снова включение того-же заголовка для того, чтобы ниже
               // создать объект класса 'C'
int main()
{
  C obj1, obj2;
  obj1.x = 0.0; // Изменение открытых полей
  obj1.y = 0.0;
  obj2.x = 1.0; // Изменение открытых полей
  obj2.y = 1.0;

  double abs1 = obj1.mod(); //вызов метода класса первого объекта
  double abs2 = obj2.mod(); //вызов того же метода для второго объекта

}
```

Вот в этом месте очень важно понять различие между классом и объектом. Класс определяет тип и свойства типа - поля и методы. Объект является экземпляром класса и имеет все свойства класса при этом значение этих свойств у каждого экземпляра может быть разным. У одного класса может быть множество объектов.

Конструкторы и деструктор

Все объекты имеют определенный срок своего существования и соответственно момент создания и уничтожения. Для создания объекта могут потребоваться определенные компьютерные ресурсы, такие как оперативная память, для хранения полей объекта, объекты и ресурсы операционной системы, файлы, периферийные устройства или каналы коммуникаций. Выделение и инициализация всех необходимых ресурсов осуществляется в момент конструирования объекта. Освобождение ресурсов происходит при уничтожении объекта.

Для конструирования и уничтожения объектов класса есть специальные методы класса - конструкторы и деструктор. Эти методы не возвращают значений и даже не имеют типа 'void'. Имя конструктора и деструктора совпадает с именем класса, перед деструктором ставится символ '~'.

Деструктор у классса может быть только один, а конструкторов несколько. Все конструкторы подразделяются на четыре типа: * конструктор по умолчанию - конструктор без параметров. Он может быть только в единственном числе. Его создает компилятор если пользователь не создал ни одного конструктора самостоятельно * конструктор инициализации - конструктор с параметрами. Таких конструкторов, различающихся набором параметров, у одного класса может быть несколько. * конструктор копирования с параметром вида 'const T&', где 'T' - класс для которого создается конструктор * конструктор перемещения с параметром вида 'T&&', где 'T' - класс для которого создается конструктор

Пример определения различных конструкторов для класса 'Example':

```
// Ниже перечислено содержимое заголовочного файла Example.h
// Объявление класса
class Example
{
public:
  Example(); // Конструктор по умолчанию - без параметров
  Example(double x, double y); // Конструктор инициализации - с параметрами
  Example(const Example&);    // Конструктор копирования - с
                              // параметром 'const Example&'
  Example& operator=(const Example&); // Оператор присваивания
                                      // (по сути, тоже копирования)
  Example(Example&&); // Конструктор перемещения -
                      // с параметром 'Example&&'
  Example& operator=(Example&&); // Оператор перемещения
   ~Example(); // Деструктор всего один и параметров у него нет
};
```

Конструкторы и деструктор также как и другие методы могут быть открытыми, закрытыми или защищенными, при этом нужно понимать, что объект с закрытым конструктором может быть создан только в том контексте,

который имеет доступ к закрытым членам класса, например, из дружественной функции.

Конструкторы и деструктор определяются также как и другие методы класса. Их выделяет только отсутствие возвращаемого значения.

```
// Ниже пример содержимого файла реализации Example.cpp
// Определение конструктора
Example::Example()
{
  x = 0;
  y = 0;
}
```

Если пользователь не определил конструктор самостоятельно, компилятор создаст при необходимости конструктор по умолчанию и конструктор копирования за него. Если пользователем определен хотя бы один конструктор, компилятор не вмешивается в работу программиста и автоматически не создает ничего.

То же самое относится и к деструктору: если пользователь его не определил, то деструктор будет создан автоматически компилятором. Но при этом надеяться, что он выполнит сложные действия по очистк памяти и освобождению ресурсов не стоит - будет только освобождена память, необходимая для хранения объекта. Динамическая память для полей класса и другие ресурсы не освобождаются.

В примере показаны ситуации, в которых вызывается тот или иной конструктор:

```
// Вызов конструктора по умолчанию
Example x, y;
Example x{};
Example *x = new Example;

// Вызов конструктора инициализации
Example x{1.0, 2.0};

// Вызов конструктора копирования
Example y{x};
Example y = x;

// Вызов конструктора перемещения
Example z = std::move(y);
Example z = func();
```

Оператор присваивания и перемещения по смыслу выполняет аналогичные конструкторам копирования и перемещения действия с отличием в том, что операторы вызываются для уже сконструированных объектов, например, вне деклараций.

```
// Вызов оператора копирования
y = x;

// Вызов оператора перемещения
z = std::move(y);
z = func();
```

Деструктор вызывается при выходе объекта из области видимости или явном удалении оператором 'delete'.

```
Example x; // глобальный объект, деструктор вызывается при
           // завершении программы
void f(Example e);
int main()
{
   Example y; // деструктор вызывается при выходе из main()
   Example z = new Example();

   f(y); // 'y' передается по значению т.е. делается копия и для этого
         // вызывается конструктор копирования
   delete z; // вызов деструктора оператором 'delete' при уничтожении
             // динамического объекта
  // деструктор также можно вызвать как обычный метод:
  // 'z->~Example();' или
  // так 'z->Example::~Example();'
}

// определение функции f
void f(Example obj)
{
   // деструктор формального параметра 'obj'- копии фактического
   // параметра 'y' - будет вызван при выходе из функции.

Example x; // деструктор локального объекта будет вызван при
           // выходе из функции
...
}
```

Конструктор может быть явным образом запрещен в классе. Например, можно явно запретить создание объектов без параметров инициализации и запретить конструктор по умолчанию. Аналогично можно запретить операции копирования и перемещения. Для этого используется ключевое слово 'delete' в строго определенном контексте, отличающемся от его использования для удаления объекта. 'delete' может быть использован для любого метода.

Иногда возникает необходимость явно указать на использование конструктора или оператора, создаваемых компилятором. Для этого

используется ключевое слово 'default'. В примере ниже пользователь, создав конструктор инициализации 'Example(double x, double y)', уже не может рассчитывать на то, что компилятор сгенерирует конструктор по умолчанию и конструктор копирования. Чтобы все же ими воспользоваться, нужно явно указать их использование признаком 'default'.

У конструкторов перемещения и инициализации способов их задать по умолчанию нет. Связано это с тем, что их невозможно однозначно определить на основе декларации класса. При инициализации возможно использование разнообразных параметров, при перемещении - процедур очистки перемещаемого объекта. С копированием и конструктором по умолчанию все понятно: копирование объектов осуществляется побитово, инициализация по умолчанию - нулями.

Конструктор может быть указан как вызываемый явным образом.

```
class Example
{
public:
  explicit Example(double x); // Конструктор инициализации - с параметрами
};
```

При таком объявлении неявное конструирование, т.е. создание экземпляров этого класса без явного вызова 'Example(...)' станет невозможно.

```
Example x(0.0);            // ОК, явный вызов конструктора
Example x = 0.0;           // ошибка, неявный вызов конструктора
void func(Example x);  // объявление функции с параметром типа 'Example'
func(0.0); // ошибка - вызов функции с неявным конструированием параметра
func( Example(0.0) );  // ОК - вызов функции с явным конструированием
                       // параметра
```

Конструктор класса осуществляет инициализацию всех необходимых ресурсов, включая поля класса. При вызове конструктора класса последовательно в порядке объявления вызываются конструкторы всех полей класса. Последовательность вызова конструкторов не зависит от выбора того или иного способа инициализации, описанных немного дальше.

```
// Класс, задающий точку на плоскости из примера про структуры
class Point
{
   // закрытые по умолчанию поля
   double x;
   double y;
public:
   Point(string str)
   {
     cout << "Point::Point(" << str << ")" << endl;
   }
};
// Класс, задающий линию между двумя точками на плоскости
```

```
class Line
{
   Point start{"start"};
   Point end{"end"};
public:
   Line()
   {
     cout << "Line::Line" << endl;
   }
}line;
```

При создании объекта 'line' класса 'Line' код из примера выше вначале вызовет конструктор для поля 'start', затем 'end' и уже затем конструктор Line::Line(). На консоли будет напечатано:

```
Point::Point(start)
Point::Point(end)
Line::Line
```

Допустимы несколько способов инициализации полей класса. Наиболее простой способ - инициализация в классе. Она осуществляется как обычная инициализация объектов за исключением инициализации в функциональном стиле с круглыми скобками: 'int x(1);'

```
class Point{
   int _x{1};  // один способ написать инициализацию в классе
   int _y = 2; // другой способ написать инициализацию в классе
public:
   // конструктор инициализации
   Point(int x, int y)
   {
     ...// дальнейшая инициализация по необходимости. _x и _y уже были
        // проинициализированы в порядке объявления
   }
};
```

Следующий способ - инициализация в списке инициализации конструктора, который начинается после заголовка конструктора, отделяется двоеточием, а элементы списка разделяются запятой. Порядок следования элементов в списке инициализации не имеет значения, инициализация все равно осуществляется в порядке объявления. Можно совмещать с предыдущим способом инициализации в классе, но способ инициализации в списке конструктора будет иметь преимущество, и вызван будет только он. При инициализации в списке оператор присваивания (знак '=') не применяется.

```
class Point{
   int _x,_y;
public:
   Point(int x, int y): _x{x},_y{y}
   {
```

```
    ...// дальнейшая инициализация
  }
};
class Point{
  int _x,_y;
public:
  Point(int x, int y)
  {
   // здесь это не очень удачный способ. Вначале будут вызваны
   // конструкторы (в данном примере - по умолчанию), и потом значения
   // полей будут переопределены операторами ниже
   _x = x;
   _y = y;
  }
};
```

При вызове деструктора класса деструкторы его полей вызываются в обратном порядке по сравнению с порядком конструирования.

Для подведения итога небольшое резюме всего сказанного о классах до этого:

```
class Example {     // Новый тип - класс Example
private:            // закрытая секция доступна только для членов класса
protected:          // защищенная секция доступна для членов класса и
                    // членов производных классов
public:             // члены открытой секции доступны всем
 int x;             // поле класса
 void f();          // метод класса
 void g() {return;}// метод, определенный в классе, разворачиватся как
                    // inline функция
 void h() const;    // метод, не изменяющий поля класса
 int operator+(int y); // t+y вызывает t.operator+(y)
 int operator-();      // -t вызывает унарный оператор t.operator-()
 Example(): x(1) {}    // Конструктор со списком инициализации
 Example(const Example& e): x(e.x) {}  // Конструктор копирования
 Example& operator=(const Example& t) {x=t.x; return *this; } // Оператор
                                                              // присваивания
 ~Example();              // Деструктор
 explicit Example(int a);// Разрешить инициализацию t=Example(3),
                         // запретив t=3
 operator int() const {return x;}  // Оператор преобразования.
                        // Разрешить преобразование int(t)
 friend void i();       // Глобальная функция i() имеет доступ к "private"
 friend class U;        // Члены класса U имеют доступ к "private"
 static int y;          // Общее поле для всех объектов типа Example
 static void l();       // Общий код для всех объектов типа T.
```

```
                    // имеет доступ только к статическим данным
 class Z {};        // Вложенный class Example::Z
 typedef int V;     // Вложенное определение типа.
                    // Example::V означает int
};


void Example::f() {     // Определение функции члена f класса Example
 this->x = x;}          // this адрес собственного экземпляра (means x=x;)
int Example::y = 2;     // Инициализация статического члена обязательна
Example::l();           // Вызов статического метода

struct Example {        // В struct всё открыто по умолчанию:
                        // class Example { public:
```

Перегрузка операторов (operator overloading)

Для фундаментальных типов данных определены и формат самих данных и операции, которые можно над ними производить. Это логические (например '>', '==', '&&', '!') арифметические ('+', '*', '%'), побитовые операторы ('&', '^', '|', '>>'). Смысл этих операторов для пользовательских типов можно определить с помощью метода специального вида, имя которого начинается со слова 'operator', за которым следует символ оператора, требующий переопределения. Большинство операторов можно определить как метод соответствующего класса или как независимую от класса функцию. Некоторые операторы определяются только как метод класса. Некоторые операторы переопределить нельзя.

Пример переопределения оператора сложения '+' для некоторого пользовательского класса 'C'.

```
//c.h
class C
{
public:
  ...
  C operator+(const C&); // Оператор сложения
  ...
};

//c.cpp
C C::operator+(const C& c2)
{
  C res;
  res.x = x+c2.x;
  res.y = y+c2.y;
```

```
  return res;
}

//main.cpp
int main ()
{
   C a1,a2,a3;
   a3 = a1+a2; // аналогично явному вызову оператора как метода класса
                  // 'a3 = a1.operator+(a2);'
   return 0;
}
```

Перегрузка операторов в форме внешней функции происходит схожим образом:

```
class C; // предварительная декларация для использования имени 'C'
          // в объявлении внешней функции

// объявление внешней функции для последующего объявления ее как 'friend'
C operator+(const C&, const C&);

// класс, для которого переопределяется оператор
class C
{
friend C operator+(const C&, const C&); // внешний оператор сложения часто
                                         // делают дружественным если для
                                         // операции требуется доступ к
                                        // защищенным членам класса
public:
  ...
};

// определение внешниего оператора сложения
C operator+(const C& c1, const C& c2)
{
  C res;
  res.x = c1.x+c2.x;
  res.y = c1.y+c2.y;
  return res;
}

//main.cpp
int main ()
{
   C a1,a2,a3;
   a3 = a1+a2; // аналогично явному вызову оператора как внешней функции
```

```
                // 'a3 = operator+(a1,a2);'
    return 0;
}
```

Перегрузка операторов - арифметические операторы

Пример перегрузки арифметических операторов для класса 'Ops'

```
class Ops{
    int v;
public:
    Ops():v(0){}
    Ops(int v){this->v=v;}
    //Присваивание x = y
    Ops& operator =(const Ops &y){
        if(this != &y)
            v=y.v;
        return *this;
    }
    //Унарный плюс +x
    Ops& operator +(){return *this;}
    //Унарный минус -x
    Ops operator -(){return Ops(-v);}
    //Сложение x + y
    Ops operator +(const Ops &y){return Ops(v+y.v);}
    //Вычитание x - y
    Ops operator -(const Ops &y){return Ops(v-y.v);}
    //Умножение x * y
    Ops operator *(const Ops &y){return Ops(v*y.v);}
    //Деление x / y
    Ops operator /(const Ops &y){return Ops(v/y.v);}
    //Остаток деления x % y
    Ops operator %(const Ops &y){return Ops(v%y.v);}
    //Префиксный инкремент/декремент ++x (x=x+1)
    Ops& operator ++(){v++; return *this;}
    Ops& operator --(){v--; return *this;};
    //Постфиксный инкремент/декремент x++ (xx=x,x=x+1,xx)
    Ops operator ++(int){ Ops ret(v); v++; return ret;}
    Ops operator --(int){ Ops ret(v); v--; return ret;};
    //Оператор приведения к целому
    operator int(){return v;}
};
```

С объектами класса 'Ops' можно производить лбые арифметические действия так же, как это делается для арифметических фундаментальных типов. При этом их поведение полностью будет задано в переопределенных выше операторах.

Ops x{1},y{2},z{3};

z=x+y; //z.v == 1+2
--z; //z.v == 2

Перегрузка операторов - логические и пр. операторы

Названия для перегружаемых операторов приводятся в списке ниже. Тип 'T', это тип аргумента, который выбирает программист в соответствии со смыслом перегружаемого оператора и класса, для которого это делается.

- Присваивание с суммированием x += y (x = x + y): operator +=(T y);
- Присваивание с вычитанием x -= y (x = x - y): operator -=(T y);
- Присваивание с умножением x = *y (x = x* y): operator *=(T y);
- Присваивание с делением x /= y (x = x / y): operator /=(T y);
- Присваивание по модулю x %= y (x = x % y): operator %=(T y);
- Присваивание с побитовым И x &= y (x = x & y): operator &=(T y);
- Присваивание с побитовым ИЛИ x |= y (x = x | y): operator |=(T y);
- Присваивание с побитовым исключающим ИЛИ x ^= y (x = x ^ y): operator ^=(T y);
- Присваивание с побитовым сдвигом влево x <<= y (x = x << y): operator <<=(T y);
- Присваивание с побитовым сдвигом вправо x >>= y (x = x >> y): operator >>=(T y);
- Равенство x == y: operator ==(T y);
- Неравенство x != y: operator !=(T y);
- Больше x > y: operator >(T y);
- Меньше x < y: operator <(T y);
- Больше или равно x >= y: operator >=(T y);
- Меньше или равно x <= y: operator <=(T y);
- Логическое отрицание НЕ !x: operator !();
- Логическое И x && y: operator &&(T y);
- Логическое ИЛИ x || y: operator ||(T y);
- Побитовая инверсия ~x: operator ~();
- Побитовое И x & y: operator &(T y);
- Побитовое ИЛИ x | y: operator |(T y);
- Побитовое исключающее ИЛИ x ^ y: operator ^(T y);
- Побитовый левый сдвиг x << y: operator <<(T y);
- Побитовый правый сдвиг x >> y: operator >>(T y);

Перегрузка операторов - обращение и память

- Обращение к элементу массива x[y]: operator C:\Users\Admin\AppData\Local\Temp\T y;
- Обращение с разыменованием («объект на который указывает x») *x: operator* ();
- Взятие адреса &x: operator &();
- Обращение к члену объекта x->y: operator ->();
- Обращение к объекту, на который указывает y: x->*y: operator* ->(T x);

- Вызов функции x(a1, a2): operator ()(T a1, U a2, ...);
- Оператор "запятая" x, y: operator ,(T y);
- Преобразование типа (type) x: operator T();
- Выделение памяти new type: void* operator new(size_t x);
- Выделение памяти (массив) new type[n]: void* operator newC:\Users\Admin\AppData\Local\Temp\size_t x;
- Освобождение памяти delete x: operator delete(void* x);
- Освобождение памяти (массив) delete[] x: operator deleteC:\Users\Admin\AppData\Local\Temp\void* x;

Не перегружаемые операторы

- Обращение к члену структуры x.y
- Обращение к объекту, на который указывает y x.*y
- Условный оператор x ? y : c
- Оператор области видимости x::y
- Размер sizeof(x) sizeof(type)
- Выравнивание alignof(type)
- Идентификация типа typeid(x)

Наследование (inheritance)

Классы в C++ могут быть расширены с помощью механизма наследования, позволяющего создавать новые классы на основе имеющихся. Класс, которому наследуют, называется базовым (base class). Класс, который наследует - производным (derived class). При наследовании производный класс наследует поля и методы своего базового класса, как правило, добавляя при этом новые поля и/или методы. Производный класс, добавляя поля и методы, является более частным типом, а базовый класс - более общим. Пример: базовый класс - класс геометрических фигур на плоскости, происходные классы - круг, квадрат, требугольник. В результате построения классификации в предметной области программы возникает диаграмма классов (class diagramm).

С технической точки зрения объекты производного класса содержат внутри себя все поля и методы базового. При этом закрытые поля 'private' базового класса в производном классе недоступны. Наследование может быть открытым, закрытым и защищенным (public, private, protected). Если тип наследования не указан, то наследование считается закрытым для класса и открытым для структуры. При открытом наследовании в производном классе поля и методы базового класса сохраняют свои спецификаторы доступа. При закрытом наследовании в производном классе всё становится закрытым. При защищенном - защищенным.

Тип наследования	Спецификатор в базовом классе	Доступ в производном классе
public	private	недоступен
	protected	protected
	public	public
potected	private	недоступен

	protected	protected
	public	protected
private	private	недоступен
	protected	private
	public	private

Наследование может производиться несколько раз, при этом при каждом наследовании будут выполняться описанные выше правила доступа

```
// базовый класс
class Example
{
public:
   double re, im;
   double mod();
};

// производный класс, открытое наследование
class Derived: public Example
{
  // новых полей в класс Derived не добавили.
  // новый класс будет содержать только поля базового.
};
// еще один производный класс, открытое наследование
class DerivedTwice: public Derived
{
  // новых полей в класс Derived не добавили.
  // новый класс будет содержать только поля базового.
};

...
// Определение объектов этих классов
Example x;
Derived y;
DerivedTwice z;
...
// доступ к полям и методам
x.re = y.re = z.re = 1.0;
x.im = y.im = z.im = 1.0;
cout << x.mod();
cout << y.mod();
cout << z.mod();
```

Наконец, наследование может быть множественным, в котором случае производный класс наследует поля и методы двух или более классов.

```
// базовый класс
class Sample
```

```
{
public:
   bool validity;
};

// производный класс, открытое наследование
class Derived: public Example, public Sample
{
  // Доступ к полям базовых классов по имени или в случае конфликата имен
  // используя селектор Example::mod(), Sample::validity
};

...
// Определение объектов этих классов
Derived x;
...
// доступ к полям и методам
x.validity = true;
cout << x.mod();
```

Виртуальное наследование (virtual inheritance) и то, зачем оно необходимоо здесь подробно рассматривать не будем. За разъяснениями можно обратиться сюда: http://ru.wikipedia.org/wiki/Виртуальное_наследование

Его смысл в том, что при множественном наследовании от классов, имеющих общий базовый класс, общий предок передается потомку в единственном экземпляре, а не множественном, от каждого из родителей.

```
class B { };
class X : public virtual B { };
class Y : public virtual B { };
class XY : public X, public Y { };
XY xy; // объект 'xy' содержит только один экземпляр полей класса 'B'.
```

Конструкторы, операторы копирования, перемещения и деструкторы не наследуются. Но их реализацией в базовом классе можно воспользоваться с помощью ключевого слова 'using'.

```
class Derived: public Example
{
public:
   using Example::Example(); // использование конструктора базового
                             // класса в производном
   Derived(int i);           // новый конструктор
}
```

Объявление в производном классе полей и методов с тем же исменем, что и в базовом, скроет имена базового класса от пользователя.

При создании объекта производного класса соблюдается строгий порядок инициализации. Сначала вызываются конструкторы базовых классов в порядке

их перечисления в списке наследования. Затем вызываются конструкторы полей класса в порядке их объявления в объявлении класса. После того как будут сконструированы все базовые классы и поля, выполняется тело конструктора.

Конструкторы копий базовых классов вызываются в том порядке, в котором они объявлены в списке наследования. Конструкторы копий полей класса вызываются в том порядке, в котором они объявлены в объявлении класса.

Деструкторы всегда вызываются в порядке, обратном порядку вызова конструкторов.

Полиморфизм

Полиморфизм позволяет единообразным образом использовать объекты различных классов. Методы с одним и тем же именем для разных объектов могут иметь разное значение. Нужный метод связывается с объектом и выбирается во время исполнения. Например, при наследовании классов возникает необходимость переопределения некоторых функций, принадлежащих базовому классу. Во время исполнения при этом должен корректно определяться и вызываться соответствующий классу объекта метод.

```
// Виртуальные функции
#include <iostream>
#include <vector>
using namespace std;

class Object // базовый класс
{
protected:
   double size;
public:
   Object(double s):size {s}
   {
   }
   virtual double area () = 0; // чисто виртуальная (pure virtual)  функция.
                               // Из-за нее объект этого класса создать
                               // нельзя, а класс называется
                               // абстрастным (abstract class)
};

class Square: public Object // производный класс, с отношением "Square
                            // является Object'ом"
{
   using Object::Object; // использование всех конструкторов базового
                         // класса. Модификаторы доступа соответствуют
                         // определениям в базовом класса и здесь не
                         // переопределяются
```

```
public:
   double area () // в базовом классе метод был объявлен виртуальным.
                  // Здесь он тоже виртуальный, хотя явно это не указано
   {
      return (size * size);
   }
};

class Triangle: public Object
{
   using Object::Object;
public:
   double area () // имя функции прежнее, но суть другая
   {
      return (size * size / 2); // считаем, что треугольник прямоугольный
   }
};

class Circle: public Object
{
   using Object::Object;
public:
   double area () // еще одна реализация функции
   {
      return (size * size * 3.14 / 4.0);
   }
};

double size= {5.0}; // песть все объекты будут одинакового размера, например
5.0
Square fig1(size);
Triangle fig2(size);
Circle fig3(size);

vector<Object*> vec; // создаем вектор и наполняем его объектами
vec.push_back(&fig1);
vec.push_back(&fig2);
vec.push_back(&fig3);

for(auto fig: vec)
   cout << fig->area() << endl; // чудо случается здесь: полиморфизм вовремя
                                // исполнения позволяет выбрать из трех
                                // методов 'area()' метод соответствующий
```

```
                                    // своему объекту, и в 'cout' будут выведены
                                    // разные значения.
```

Некоторые особенности наследования на примере

Здесь на небольшом примере будут комментариями к исходному тексту программы отмечены некоторые особенности наследования и полиморфизма.

```
class A
{
public:
   A(){}; // простейший конструктор по умолчанию. Конструктор
          // не может быть виртуальным.
   operator A*() // оператор преобразования к типу 'A*'.
                 // Тип возвращаемого значения не  требуется.
   {
      return this;
   }
   void afunc(); // в базовом классе не объявлена как виртуальная
                 // и больше никогда виртуальной не станет
   virtual void bfunc(); // здесь в базовом классе объявлена как открытая,
                         // при наследовании модификатор доступа можено
                         // будет поменять
   virtual void cfunc(); // Не наследуется классом Class
   virtual ~A(){};       // Простой виртуальный деструктор.
                         // У чисто виртуального
                         // деструктора все равно должно быть тело
};
void A::afunc()
{
   cout<<"A::afunc()"<<endl;
}
void A::bfunc()
{
   cout<<"A::bfunc()"<<endl;
}
void A::cfunc()
{
   cout<<"A::cfunc()"<<endl;
   cout<<"Calling the bfunc() from A::cfunc()...";
   bfunc(); // bfunc() - виртуальный. Может быть вызван метод производного
            // класса, о котором здесь, в базовом классе ничего не известно.
}

class B:public A
{
public:
```

```
   virtual void afunc(); // не была в базовом классе объявлена как
                         // виртуальная. В производном классе виртуальной ее
                         // не сделать. Компилятор игнорирует ключевое
                         // слово 'virtual'
   //virtual void bfunc(int); // это объявление скроет методы базового
                              // класса с тем же именем несмотря на то, что
                              // сигнатура отличается параметром
};
void B::afunc()
{
   cout<<"B::afunc()"<<endl; // не виртуальная
}
/*
void B::bfunc(int x)
{
   cout<<"B::bfunc()"<<endl; // виртуальная
}
*/
class C:public B
{
public:
   C() {}; // наличие пользовательского конструктора не препятствует
           // автоматическому созданию конструктора копирования
protected:
   virtual void afunc();
   void bfunc();// этот метод в производном классе стала защищенной.
                // Модификаторы доступа можно менять. И, кстати,
                // осталась виртуальной несмотря на отсутствие ключевого
                // слова 'virtual'.
};
void C::afunc()
{
   cout<<"C::afunc()"<<endl;//
   cout<<"Calling bfunc() from C: ";
   bfunc(); // вызов собственного метода класса
   cout<<"Calling B::bfunc from C: ";
   B::bfunc(); // вызов метода базового класса
   cout<<"Calling A::bfunc from C: ";
   A::bfunc(); // вызов метода базового-базового класса
}
void C::bfunc()
{
   cout<<"C::bfunc()"<<endl;//
}
```

```
int main()
{

   A a,*pa=&a; // объявление объекта и указателя на него
   B b,*pb=&b; // ...
   C c;
   C d(c);// вызывается конструктор копирования. Не важно, есть ли в 'C'
          // другие пользовательские конструкторы; в отличие от
          // конструкторов без аргументов, конструктор копий
          // доступен всегда

   A *pc=&c; // указатель типа 'A*' указывает на объект класса 'C'

   cout<<"calling A::afunc() from main:"<<endl;
   a.afunc(); // метод не виртуальный, вызывается метод класса 'A' т.к.
              // объект объявлен как объект класса 'A'
   cout<<"calling PB->afunc() from main:"<<endl;
   pb->afunc(); // указатель и объект, на коорый он указывает имеют тип 'B'
   cout<<"calling PA->afunc() from main:"<<endl;//
   pc->afunc(); // в C-классе этот виртуальных метод и вовсе защищенный,
                // но вызывается не он а не виртуальный метод класса 'A' т.к.
                // указатель 'pc' имеет тип 'A*'.

   cout<<"calling PB->bfunc() from main:"<<endl;//
   pb->bfunc(); // в классе B этот метод закомментирован. Используется
                // метод базового класса. Но если бы в классе был определена
                // 'bfunc(int)', то метод базового класса становится скрытым
                // и эта строка вызвала бы ошибку компилляции
   cout<<"calling PA->bfunc() from main:"<<endl;//
   pc->bfunc(); // полиморфизм в действии. 'pc' имеет тип 'A*', но
                 // вызывается метод класса 'C', соответсвующего объекту,
                 // на который указывает указатель.
   cout<<"calling PA->cfunc() from main:"<<endl;//
   pc->cfunc(); // cfunc() определена только в 'A', зато она уже вызывает
                // метод производного класса
   ...
   return 0;
}
```

Чтобы избежать недоразумений, связанных с наследованием, существуют спецификаторы 'override' и 'final'.
'final' отмечает класс или виртуальный метод как не подлежащие наследованию

```
class A final // класс не может иметь наследников,
              // хотя сам может иметь базовый класс
{
};
class B
{
    virtual void func() final; // метод должен быть виртуальным.
                               // В наследниках его переопределить
                               // будет нельзя
};
```

'override' указывает компилятору на то, что виртуальный метод переопределяется.

```
class A
{
public:
    virtual void func();
};
class B: public A
{
    virtual void func() override; // переопределяемый метод должен быть
                                  // виртуальным. Если в базовых классов он
                                  // не обнаружен, или не является
                                  // виртуальным, компилятор выдаст ошибку
};
```

Шаблоны (template)

Шаблоны позволяют писать уже знакомые языковые конструкции - функции и классы с используемыми в них типами, задаваемыми параметрами. Например, класс комплексных чисел может быть определен как для сотавляющих типа 'double', так и для 'float' и даже 'int'. Без шаблонов пришлось бы писать три одинаковых класса. Шаблоны позволяют отвлечься от особенностей типов и написать класс, использующий некий неопределенный тип 'T', который в дальнейшем примет значения 'double', 'float' и 'int'. Шаблонизация производится с помощью ключевого слова 'template'. Языком поддерживаются шаблоны классов и шаблоны функций.

Шаблоны классов (class templates)

Шаблоны класов позволяют создавать классы C++ с типами и значениями в качестве параметров объявления класса.

```
// объявление шаблона класса
template <typename T> // можно также написать 'template <class T>' -
                      // варианты идентичны
class Complex
{
   T re, im;
```

```
public:
   Complex();
   Complex(T re, T im);
};

// определение конструктора по умолчанию
template <typename T>
Complex::Complex(): re{0},im{0}
{
}
// определение конструктора инициализации
template <typename T>
Complex::Complex(T r, T i): re{r},im{i}
{
}

// создание объекта на основе шаблона класса
Complex<double> x;
Complex<float> y;
Complex<int> z(1,1);
```

Шаблон раскрывается компилятором C++ в обычный класс с подстановкой вместо формального параметра-типа факического типа, указанного при инстанциировании. В коде из примера выше будут созданы три класса по одному для каждого из параметров 'double', 'float', 'int'. Например, для 'double' будет создан такой класс:

```
class Complex
{
   double re, im;
public:
   Complex();
   Complex(double re, double im);
};
```

Определение методов аналогично тому, как они определяются в обычных классах можно дать при объявлении класса либо в файле реализации указав что метод принадлежит шаблону класса.

```
// Определение внутри класса
template <typename T>
class Complex
{
   ...
   Complex(double re, double im): this->re{re}, this->im{im}
   {
     ...
   }
```

```
    ...
};

// Определение вне класса. Необходимо отдельно указать
// что это метод класса-шаблона
template <typename T>
Complex::Complex(double re, double im): this->re{re}, this->im{im}
{
    ...
}
```

В шаблоне можно задать число параметров больше одного или их переменное число (0 или больше). Можно комбинировать определенные параметры и переменные.

```
template <typename T, typename U> // два параметра - два типа, которые при
                                  // инстанциировании получат
                                  // собственные значения
class A;
A<int,int> x; // инстанциирование шаблона с двумя параметрами

template <typename T, typename ...U> // один обязательный параметр и
                                     // переменное число параметров.
class B;
B<int> y; // инстанцирование шаблона с переменным числом параметров.
          // Здесь после int могут быть перечислены один или более
          // параметров
```

Кроме типов параметрами шаблона могут выступать значения. Аналогично функциям параметры шаблонов могут иметь значения по умолчанию.

```
template <typename T, typename U=T, int n=0>  // Шаблон с параметрами
                                              // по умолчанию
class D;

// Возможные варианты инстанциирования такого шаблона
D<float> x;
D<float, double> y;
D<float, double, 5> z; // в качестве параметров указаны и типы и
                       // значение параметра
```

Шаблоны функций (function template)

В дополнение к шаблонам классов в языке C++ есть возможность создавать шаблоны функций. Принципиальным отличием шаблонов функций от шаблонов классов в том, что параметры шаблона не обязательно указывать явно - компилятор выводит их тип из фактических параметров шаблона.

```
template <typename T>
T f(T t);       // объявлении функции 'f()' для любого типа 'T'.
```

```
auto x = f(3.14); // разворачивается в 'double x = f<double>(3.14);'

auto x = f<double>(3.14); // параметр можно указать явно
```

Пространства имен

Типы и объекты можно создавать в именованных областях, называемых пространствами имен. Пространства имен позволяют разделить объявления и определения на отдельные независимые области, в которых можно .

```
namespace N
{
  class T {}; // класс T объявлен внутри пространства имен N
  T t;          // объект t  объявлен внутри пространства имен N
}
namespace O
{
  class T {}; // класс T объявлен внутри пространства имен O и не
              // конфликтует с тем же именем в N
  T t; // объект t  объявлен внутри пространства имен O и не конфликтует с
       // тем же именем в N
}

N::T t1;                // Использование T в пространстве имен N
N::t = t1;
using namespace N; // Использовать пространство имен N по умолчанию
T t2;
t = O::t;                // Используется пространство имен N по умолчанию.
                         // Для объектов пространства имен 'O' необходимо
                         // указывать спецификатор доступа 'O::'
```

Многопоточное программирование

В этом разделе очень сжато скажем несколько слов о том, как ускорить вычисления. До этого момента все вычисления производились последовательно, используя один поток исполнения. Если система позволяет использовать одновременно несколько вычислительных ядер центрального процессора, то вычисления, которые можно разделить и сделать независимо, можно существенно ускорить. Для распараллеливания вычислений есть несколько способов. Наиболее простым, не требующим детальных знаний о многопоточном программировании, является применение декларативных расширений языка, а именно открытого стандарта распараллеливания 'OpenMP' для компиляторов C, C++, FORTRAN или расширения языка от Intel - Cilk Plus. И 'OpenMP' и 'Cilk Plus' не являются подключаемыми библиотеками, а представляют собой расширение языка, реализованное в компиляторе.

'OpenMP' поддерживается компилятором MinGW в редакции TDM, устанавливаемым вместе с Code::Blocks. Для того, чтобы воспользоваться преимуществами многопоточности нужно сделать два действия: включить в исходный текст заголовочный файл и в настройках проекта, во вкладке настроек компоновщика (Project->Build options... вкладка 'Linker settings') добавить имя библиотеки: 'gomp'.

Ниже приведен пример из Wikipedia. К используемым директивам даны комментарии. Более полный список параметров для директив OpenMP в одном месте собран там же, на английской странице OpenMP в Wikipedia.

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

#include <omp.h>

int main(int argc, char *argv[])
{
  // выполняется в каждом ядре процессора. 4 ядра -
  // будет выведено 4 строки.
  #pragma omp parallel
  cout << "This runs in each core in parallel" << endl;
  // выполняется в каждом ядре процессора своя часть цикла
  #pragma omp parallel for
  for (int i = 0; i < 4; i++)
    cout << "\'for\' body is paralleled. This is No" << i << endl;
  int th_id, nthreads;
  // распараллеливает исполнение
  #pragma omp parallel private(th_id) shared(nthreads)
  {
    th_id = omp_get_thread_num();
    #pragma omp critical // взаимное исключение обеспечивает что этот блок
                         // единовременно исполняет только один поток и дает
                        // допечатать строку до конца
    {
      cout << "Hello World from thread " << th_id << endl;
    }
    #pragma omp barrier // дождаться завершения всех потоков в этой точке

    // это не распараллеливается (master thread only)
    #pragma omp master
    {
      nthreads = omp_get_num_threads();
      cout << "There are " << nthreads << " threads" << endl;
    }
  }
```

```
  return 0;
}
```

Расширением директив 'OpenMP', использующим также вычислительные возможности графических ускорителей, является 'OpenACC'.

Потоки стандартной библиотеки и синхронизация потоков

В стандартную библиотеку C++ включены стандартные, доступные на всех реализациях C++, полностью поддерживающих стандарт, классы для работы с потоками. Они определены в заголовке . Поток - это последовательность инструкций, которая может выполняться одновременно с другими потоками одновременно. Одновременность выполнения реализована либо с помощью разделения выполнения потоков на отдельных физических ядрах процессора (параллельное выполнение, parallelism) или с помощью механизмов многозадачности как правило предоставляемых операционной системой (многозадачность, concurrency).

Использование потоков на C++ происходит следующим образом:

```
#include <thread>
#include <string>

using namespace std;

// обычная функция будет использоваться в качестве потока
int worker1(const string &str, int num)
{
   cout << "function " << str << num << endl;
   return num;
}

// функтор, или функция с состоянием, в качестве потока
class Worker
{
public:
   int operator()(const string &str, int num) const
   {
      cout << "functor " << str << num << endl;
      return num;
   }

};

int main()
{
  Worker worker2;
  // лямбда в качестве потока
```

```
auto worker3 = [](const string &str, int num)
{
   cout << "lambda " << str << num << endl;
   return num;
};
// количество аппаратных исполниетелей потока
// (как правило, количество ядер процессора)
cout << "hardware concurrency threads: "
      << thread::hardware_concurrency() << endl;
// создание и запуск потоков на исполнение
thread trd1(worker1, "worker thread No ", 1); // вызов функции в отдельном
                                              // потоке (на отдельном ядре
                                              // процессора)
thread trd2(worker1, "worker thread No ", 2); // ничто не мешает запустить
                                              // одно и то же дважды
thread trd3(worker2, "worker thread No ", 3); // вызов функтора в отдельном
                                              // потоке
thread trd4(worker3, "worker thread No ", 4); // вызов лямбда-выражение в
                                              // отдельном потоке
cout << "main thread" << endl; // основной поток тоже выполняется
// ожидание завершения соответствующих потоков
trd1.join(); // ожидание завершения птока №1
trd2.join(); // ...
trd3.join();
trd4.join();
...
```

Можно заметить, что консольны вывод одновременно выполняющихся потоков перемешивается. Чтобы этого избежать, а также упорядочить совместный доступ к данным, необходимо использование методов синхронизации потоков. В распоряжении программиста есть стандартное средство - mutex (mutual exclusion, взаимное исключение). Взаимное исключение защищает критическую секцию кода (critical section): захват взаимного исключения заставляет другие потоки, пытающиеся захватить то же взаимное исключение, ожидать его освобождения. Взаимные исключения определены в заголовочном файле. В предыдущем примере использование мьютексов позволит синхронихировать вывод потока в общую консоль. Для этого перед выводом в консоль необходимо делать захват взаимного исключения, и после завершения освобождать его, давая другим потокам захватить мьютекс и начать свой вывод.

```
include <mutex>
using namespace std;

mutex mtx;
```

```
int worker1(const string &str, int num)
{
   mtx.lock();
   cout << "function " << str << num << endl;
   mtx.unlock();
   return num;
}
```

В стандартной библиотеке определены и другие механизмы синхронизации, которые следует подробно изучить, если потребуется использовать многопоточное программирование. Также отлаженная и устойчивая реализация многопоточности есть в нестандартных библиотеках: Intel Threading Building Blocks (TBB), Boost.Threads, POCO.

Использование графических ускорителей.

Графические ускорители или видеокарты предоставляют для вычислений дополнительные мощьности, характеризуемые высокой степенью параллелизма: обычные пользовательские видеокарты содержат десятки и сотни вычислительных ядер. При том, что производительность отдельного ядра GPU меньше производительности CPU, в параллелизируемых вычислениях GPU позволяют получить существенный прирост производительности.

Для программирования графических ускорителей производители плат созданы соответствующие технологии OpenCL/CUDA. Эти технологии представляют собой компилятор и интерфейс взаимодействия с графическим ускорителем. OpenCL поддерживается ведущими производителями видеокарт - NVIDIA(линейка продуктов GeForce и др.), Intel (встроенные ускорители Intel HD Graphics), AMD (линейка продуктов Radeon и др.); CUDA - технология, созданная NVIDIA для своих карт и работающая только на них; ATI Stream - аналог CUDA от AMD.

Для высокоуровневой работы с OpenCL/CUDA на C++ созданы библиотеки, существенно упрощающие программирование.

Библиотека	Сайт проекта	поддержка OpenCL	поддержка CUDA
Boost.Compute	http://kylelutz.github.io/compute/	да	нет
VexCL	https://github.com/ddemidov/vexcl	да	да
Thrust	http://thrust.github.io/	нет	да
Bolt	https://github.com/HSA-Libraries/Bolt	да (на AMD)	нет

Отдельно стоит упомянуть технологию Microsoft C++ AMP, работающую в Visual C++ (http://msdn.microsoft.com/ru-ru/library/hh265137.aspx) и в основе использующую DirectX 11 (DirectCompute).

Библиотека Boost.Compute работает на большинстве графических карт ведущих производителей, поэтому кратко расскажем о том как ее подключить и начать использовать. Настройка Boost.Compute предельно проста:

В каталог, в котором установлен Code::Blocks необходимо загрузить и распаковать библиотеку Boost и самостоятельно добавить в нее Boost.Compute т.к. последняя пока официально не включена в состав Boost. Например, можно создать такую структуру: ...CodeBlocks/boost/compute .../accumulators .../algorithm .../archive .../asio ...

В исходных файлах включить нужные заголовки:

```
#include <boost/compute/device.hpp>
#include <boost/compute/system.hpp>
#include <boost/compute/platform.hpp>
namespace compute = boost::compute; // для сокращения названия
                                    // пространства имен
using namespace compute; // чтобы пространство имен Boost.Compute
                         // не указывать
```

Добавить '$(APP_PATH)' в пути поиска компилятора (это макроопределение среды Code::Blocks, указывающее на каталог в котором он установлен). Пути поиска компилятора задаются в меню Project->Build options... вкладка 'Search directories->Compiler' для выбранной конфиграции (Debug и/или Release).

В настройках компоновщика (Project->Build options... вкладка 'Linker settings') добавить имя библиотеки: 'OpenCL'.

Далее в зависимости от типа видеокарты сделать следующее:

Intel GPU

Открыть страницу с SDK (Software Development Kit):
http://software.intel.com/en-us/vcsource/tools/opencl-sdk

Загрузить и установить библиотеку времени исполнения, соответствующую вашей платформе (Win64, Win32, Linux, Mac), только для CPU (Central Processing Unit). Например:
intel_sdk_for_ocl_applications_2013_r3_runtime_x64_setup.msi
Загрузить и установить OpenCL SDK для вашей платформы intel_sdk_for_ocl_applications_2013_r3_x64_setup.exe
Добавить в пути поиска компилятора (в дополнение к ранее добавленному макросу '$(APP_PATH)'):

- C:\Program Files (x86)\Intel\OpenCL SDK\3.0\include

Добавить в пути поиска компоновщика (меню Project->Build options... вкладка 'Search directories->Linker'):

- C:\Program Files (x86)\Intel\OpenCL SDK\3.0\lib\x86

Точные пути зависят от вашей платформы и от каталога, указанного вами при установке SDK.

NVidia GPU

Загрузить и установить CUDA SDK https://developer.nvidia.com/cuda-downloads
Добавить в пути поиска компилятора:

- C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v6.0\include

Добавить в пути поиска компоновщика:

- C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v6.0\lib\Win32

AMD GPU

Загрузить и установить OpenCL SDK http://developer.amd.com/tools-and-sdks/opencl-zone/opencl-tools-sdks/amd-accelerated-parallel-processing-app-sdk/
Добавить в пути поиска компилятора:

- C:\Program Files\AMD APP SDK\2.9\include

Добавить в пути поиска компоновщика:

- C:\Program Files\AMD APP SDK\2.9\lib\x86

Попробовать собрать простой проект с файлом 'main.cpp', содержащим следующий код (один из многочисленных примеров на сайте проекта: https://github.com/kylelutz/compute/tree/master/example):

```
#include <iostream>

#include <boost/compute/device.hpp>
#include <boost/compute/system.hpp>
#include <boost/compute/platform.hpp>

namespace compute = boost::compute;

using namespace std;

int main()
{
   std::vector<compute::platform> platforms = compute::system::platforms();

   for(size_t i = 0; i < platforms.size(); i++){
      const compute::platform &platform = platforms[i];

      std::cout << "Platform '" << platform.name() << "'" << std::endl;

      std::vector<compute::device> devices = platform.devices();
      for(size_t j = 0; j < devices.size(); j++){
         const compute::device &device = devices[j];

         std::string type;
         if(device.type() & compute::device::gpu)
            type = "GPU Device";
         else if(device.type() & compute::device::cpu)
            type = "CPU Device";
         else if(device.type() & compute::device::accelerator)
            type = "Accelerator Device";
         else
            type = "Unknown Device";
```

```
        std::cout << "  " << type << ": " << device.name() << std::endl;
      }
   }

   return 0;
}
```

Сторонние мультиплатформенные библиотеки

Boost

Сообществом разработчиков C++ созданы мультиплатформенные библиотеки, позволяющие писать единый портируемый код, работоспособный на различных операционных системах и аппаратных архитектурах. В первую очередь стоит познакомиться с возможностями библиотеки Boost (http://www.boost.org/). Весь пакет Boost состооит из отдельных библиотек, предназначенных для решения той или иной задачи - синхронизация потоков, регулярные выражения, математические вычисления и т.д. Список библиотек есть на сайте (http://www.boost.org/doc/libs/). За немногочисленными исключениями библиотеки реализованы в виде заголовочных файлов. Для их использования достаточно включить заголовок в исходный код:

```
#include <boost/circular_buffer.hpp> // корневой каталог, в котором лежит
                                     // 'boost' должен быть в путях
                                     // компилятора.

boost::circular_buffer<int> cb(3); // циклический буфер для 3-х объектов
                                   // типа 'int'
cb.push_back(0); // при последовательном вызове, записывает значение
                 // параметра (здесь это 0) в буфер, при переполнении буфера
                 // начинает писать в первую ячейку буфера т.е. циклически.
...
```

Включая заголовок не забудте добавить в опциях среды разработки путь до папки, в которой лежит Boost.

POCO C++ Libraries

Пакет библиотек POCO (http://pocoproject.org/) содержит разнообразные мультиплатформенные средства для работы с TCP/IP сетями, файловой системой, текстовыми и XML файлами, алгоритмами компрессии, мультипоточностью, работой с базами данных, криптографией, журналированием и т.д. Библиотека требует предварительной сборки для целевой платформы.

Reason

Reason (http://reasoning.biz/index.htm) поддерживает работу с потоками, сетевое взаимодействие, базы данных и другие группы функций.

Eigen - библиотека линейной алгебры

Библиотека Eigen (http://eigen.tuxfamily.org/) поддерживает операции линейной алгебры: матричные векторные вычисления, решения линейных уравнений.

Folly

Библиотека Folly (https://github.com/facebook/folly) от Facebook - разнообразные вспомогательные классы и алгоритмы.

Список литературы

1. Stroustrup, Bjarne.The C++ programming language.—Fourth edition. Addison-Wesley ISBN 978-0321563842. May 2013.
2. Бьерн Страуструп. Язык программирования C++. Специальное издание. Пер. с англ. А. А. Кузьмичевой под ред. Н. Н. Мартынова .— М.: Издательство Бином, 2011 .— 1136 с.: ил.
3. Лаптев В. В. C++. Объектно-ориентированное программирование: Учебное пособие. – СПб.: Питер 2008. – 464 с.: ил. – (Серия «Учебное пособие»).
4. Лаптев В.В., Морозов А.В., Бокова А.В. C++. Объектно-ориентированное программирование: Задачи и упражнения. – СПб.: Питер 2007. – 288 с.: ил.
5. Скотт Мейерс. Эффективное использование C++: Пер. с англ. – М.: Книга по Требованию, 2006. – 300 с.: ил. (Серия «Профессиональная серия от Addison-Wesley»).
6. Скотт Мейерс. Наиболее эффективное использование C++. 35 новых рекомендаций по улучшению ваших программ и проектов: Пер. с англ. – М.: ДМК Пресс, 2014. – 298 с.: ил.
7. Стандарт C++11 ISO/IEC 14882:2011
   http://isocpp.org/std/the-standard

В 2009 году Университет стал победите конкурса, в результате которого определены 12 ведущи сии, которым присвоена категория «Национальный исследовательский университет». Министерством образования и науки Российской Федерации была утверждена программа его развития на 2009–2018 годы. В 2011 году Университет получил наименование «Санкт-Петербургский национальный исследовательский университет информационных технологий, механики и оптики»

КАФЕДРА ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ ТОПЛИВНО-ЭНЕРГЕТИЧЕСКОГО КОМПЛЕКСА

Кафедра химии входила в состав первых 14 кафедр ЛИТМО, сформированных в 1930 году. В 1930–1960 годах кафедра работала в рамках факультета Точной механики; в период деятельности Инженерно-физического факультета (ИФФ) с 1946 года по 1954 год кафедра входила в состав ИФФ. С 1933 года – кафедру возглавлял известный специалист в области оптического стекла профессор В.Г. Воано, позже – известный русский ученый-химик профессор С.А. Щукарев. С 1954 по 1972 год кафедру возглавлял доцент Г.С. Кошурников.

С момента второго рождения инженерно-физического факультета в 1976 г. кафедра химии вошла в его состав. В это время на кафедре стали развиваться, в основном, три научно-технологических направления: создание новых композиционных оптических материалов; разработка химических сенсоров; технология оптического волокна.

В последующие годы сотрудники кафедры, прежде всего, профессора Новиков А.Ф. и Успенская М.В., существенно переработали методику преподавания курса химии, адаптировав ее к активно внедрявшейся тогда в Университете системе дистанционного обучения. В результате, преподавание курса химии в Университете ИТМО вышло на новый более высокий уровень.

В дальнейшем на кафедре под руководством профессора М.В. Успенской активно развивалось научно-техническое направление в области химии и физики сорбирующих полимерных материалов и нанокомпозитов. В частности, на основе акриловых супервлагоабсорбентов разработан ряд новых материалов многофункционального назначения: сенсоры, жидкие линзы, раневые повязки, искусственные почвы для сельского хозяйства, огнестойкие конструкционные элементы и др.

В связи с этим в 2011 году данная кафедра (исторически – кафедра химии) позиционировала себя как отдельное структурное подразделение Национального исследовательского университета ИТМО в качестве кафедры “Информационных технологий топливно-энергетического комплекса”.

С переходом отечественных предприятий на международные стандарты продукции, повышением требований к охране окружающей среды и внедрением сложных аналитических автоматизированных систем контроля качества и мониторинга, с 2008 года в рамках направления «Техническая физика» кафедра проводит подготовку магистров и бакалавров по профилю «Физико-технические аспекты аналитического приборостроения».

Подготовка включает в себя следующие разделы:

- Компьютерные комплексы для автоматизированного контроля физических, химических, механических, термических, реологических и некоторых других свойств нефтяного сырья и продуктов нефтепереработки;
- Встроенные микропроцессорные комплексы для управления технологическими процессами и измерением широкого круга параметров энергетических установок и систем энергоснабжения;
- Физико-математическое моделирование технологических процессов нефтепереработки и топливно-энергетического комплекса;
- Информационно-аналитические системы и комплексы различного профиля, адаптированные под специфические условия работы на предприятиях ТЭК.

Уникальная программа обучения сочетает фундаментальную подготовку в области информационных систем, физической оптики, молекулярной спектроскопии, аналитической и физической химии, компьютерной метрологии, общехимической технологии и автоматики.

В рамках специальных дисциплин изучаются приборы и методы контроля качества продукции и принципы построения автоматизированных анализаторных систем для предприятий ТЭК, нефтяной и химической промышленности.

Такие системы как основа информационных технологий контроля качества и мониторинга безопасности могут успешно применяться практически на всех предприятиях и лабораториях химического и нефтехимического профиля, а также в металлургической, пищевой и фармацевтической промышленности.

Выпускники кафедры имеют широкие перспективы трудоустройства в современных крупных компаниях ТЭК, таких как Роснефть, ПТК, Газпром, Киришинефтеоргсинтез, Лукойл, ТНК-ВР, а также на предприятиях и лабораториях пищевой, фармацевтической и других отраслях промышленности.

Практика эксплуатации предприятий ТЭК подтверждает необходимость создания и применения эффективных систем контроля за безопасностью и систем экологического мониторинга.

В связи с этим с 2011 года были разработаны и открыты бакалаврская и магистерская программы по направлению подготовки 241000 " Энерго- и ресурсосберегающие процессы в химической технологии, нефтехимии и биотехнологии ". Основной целью образовательной магистерской программы

"Информационные ресурсосберегающие технологии и экологические аспекты на предприятиях ТЭК" является подготовка высококвалифицированных специалистов, соответствующих современным требованиям к выпускникам вуза, с учетом потребностей рынка труда Санкт-Петербурга и регионов России. Будущие магистры будут способны использовать информационные технологии и математическое моделирование для описания различных физических и физико-химических процессов, для контроля качества продукции нефтепереработки, работать на современном оборудовании в научных, научно-производственных и производственных лабораториях по исследованию выпускаемой продукции и т.д.

Основными направлениями научной деятельности в рамках магистерской программы являются:

- Создание приборов и датчиков физических величин и физико-химических параметров углеводородного сырья и продуктов (в том числе на основе нанотехнологий);
- Разработка приборов для измерения параметров качества нефтепродуктов и пищевых продуктов на основе компьютерных технологий;
- Создание эффективных информационных систем контроля качества продукции и коммерческого учета на предприятиях ТЭК на основе приборов и устройств различного назначения;
- Создание эффективных информационных систем мониторинга безопасности эксплуатации объектов ТЭК.

Подготовка магистров ведется с участием ряда промышленных предприятий, научно-производственных объединений, научно-исследовательских институтов и вузов Санкт-Петербурга, что дает возможность получить отличные знания и неоценимый опыт в различных сферах деятельности: производственной, научно-исследовательской, административной и т.д.

Биотехнология и биоинженерия являются приоритетными направлениями современной науки и промышленного производства. Продукты биотехнологии и биоинженерии востребованы в медицине, фармации, биологии, и других высокотехнологичных отраслях народного хозяйства. Разработка новых источников энергии, создание биосовместимых материалов и синтез биологически активных веществ – главные составляющие этих двух наук и отраслей производства. В частности, интенсивно развиваются производство и применение ферментов в переработке различных видов сырья и в получении биопрепаратов. Ферментные технологии имеют преимущества с экономической, технологической и экологической точек зрения, поэтому годовой оборот ферментных препаратов составляет десятки миллионов долларов США и он непрерывно растёт. По объёму производства ферментные препараты занимают третье место после аминокислот и антибиотиков. Ферментативные процессы, применяемые в технологиях, аналогичны

природным, но они более безопасны и для здоровья человека и для окружающей среды.

Развитие этих отраслей сдерживается недостатком специалистов высшего уровня, подготовленных в области информационного обеспечения и средств измерения живых систем и биологических структур.

Для решения проблемы подготовки магистров на стыке информационных технологий, биологии и инженерии объединены усилия двух кафедр: Кафедра химии и молекулярной биологии ИХиБТ и кафедра ИТТЭК, имеющих опыт подготовки специалистов бакалавров и магистров в информационных технологиях и биотехнологии.

В учебный план предлагаемой программы включены, наряду с общеобразовательными, дисциплины по информационной, биологической, химической, технологической подготовке и ряду других отраслей знаний, необходимых в подготовке специалистов заявленного уровня.

В настоящее время на каф. ИТТЭК под руководством проф. Успенской М.В., ведутся работы по направлениям, связанных с созданием материалов для фармакологии и регенеративной медицины, предметов санитарно-гигиенического назначения, а также биосовместимых и биодеградируемых материалов.

Также на кафедре под руководством проф. Неелова И.М. активно развивается моделирование полимеров и биополимеров, начиная от структуры веществ и физико-химических процессов, протекающих в живых организмах до физико-механических и эксплуатационных характеристик материалов и биосистем.

Профессорско-преподавательский состав на кафедре насчитывает 18 человек, из них 6 профессоров и докторов наук.

В настоящее время на базе кафедр НИУ ИТМО создан Международный научно-исследовательский институт биоинженерии, возглавляемый проф. М.В. Успенской, что значительно расширяет экспериментальную базу и научный потенциал кафедр и способствует повышению уровня подготовки кадров высшей категории.

В настоящее время на кафедре трудятся 18 преподавателей, шестеро из них являются докторами наук, профессорами, признанными на международном уровне, членами ученых советов в России и за рубежом.

Содержание

Никехин А.А.

Основы C++ для моделирования и расчетов
Учебное пособие

авторской редакции
Редакционно-издательский отдел НИУ ИТМО
Зав. РИО Н.Ф. Гусарова
Лицензия ИД № 00408 от 05.11.99
Подписано к печати
Заказ № 3185
Тираж 100
Отпечатано на ризографе

Редакционно-издательский отдел
Санкт-Петербургского национального исследовательского университета
информационных технологий, механики и оптики

197101, Санкт-Петербург, Кронверкский пр., 49