**Язык программирования C++**

Язык программирования С++ представляет высокоуровневый компилируемый язык программирования общего назначения со статической типизацией, который подходит для создания самых различных приложений. На сегодняшний день С++ является одним из самых популярных и распространенных языков.

Своими корнями он уходит в язык Си, который был разработан в 1969—1973 годах в компании Bell Labs программистом Деннисом Ритчи (Dennis Ritchie). В начале 1980-х годов датский программист Бьерн Страуструп (Bjarne Stroustrup), который в то время работал в компании Bell Labs, разработал С++ как расширение к языку Си. Фактически вначале C++ просто дополнял язык Си некоторыми возможностями объектно-ориентированного программирования. И поэтому сам Страуструп вначале называл его как "C with classes" ("Си с классами").

Впоследствии новый язык стал набирать популярность. В него были добавлены новые возможности, которые делали его не просто дополнением к Си, а совершенно новым языком программирования. В итоге "Си с классами" был переименован в С++. И с тех по оба языка стали развиваться независимо друг от друга.

С++ является мощным языком, унаследовав от Си богатые возможности по работе с памятью. Поэтому нередко С++ находит свое применение в системном программировании, в частности, при создании операционных систем, драйверов, различных утилит, антивирусов и т.д. К слову сказать, ОС Windows большей частью написана на С++. Но только системным программированием применение данного языка не ограничивается. С++ можно использовать в программах любого уровня, где важны скорость работы и производительность. Нередко он применяется для создания графических приложений, различных прикладных программ. Также особенно часто его используют для создания игр с богатой насыщенной визуализацией. Кроме того, в последнее время набирает ход мобильное направление, где С++ тоже нашел свое применение. И даже в веб-разработке также можно использовать С++ для создания веб-приложений или каких-то вспомогательных сервисов, которые обслуживают веб-приложения. В общем С++ - язык широкого пользования, на котором можно создавать практически любые виды программ.

С++ является компилируемым языком, а это значит, что компилятор транслирует исходный код на С++ в исполняемый файл, который содержит набор машинных инструкций. Но разные платформы имеют свои особенности, поэтому скомпилированные программы нельзя просто перенести с одной платформы на другую и там уже запустить. Однако на уровне исходного кода программы на С++ по большей степени обладают переносимостью, если не используются какие-то специфичные для текущей ос функции. А наличие компиляторов, библиотек и инструментов разработки почти под все распространенные платформы позволяет компилировать один и тот же исходный код на С++ в приложения под эти платформы.

В отличие от Си язык C++ позволяет писать приложения в объектно-ориентированном стиле, представляя программу как совокупность взаимодействующих между собой классов и объектов. Что упрощает создание крупных приложений.

Основные этапы развития

В 1979-80 годах Бьерн Страуструп разработал расширение к языку Си - "Си с классами". В 1983 язык был переименован в С++.

В 1985 году была выпущена первая коммерческая версия языка С++, а также первое издание книги "Языка программирования C++", которая представляла первое описание этого языка при отсутствии официального стандарта.

В 1989 была выпущена новая версия языка C++ 2.0, которая включала ряд новых возможностей. После этого язык развивался относительно медленно вплоть до 2011 года. Но при этом в 1998 году была предпринята первая попытка по стандартизации языка организацией ISO (International Organiztion for Standartization). Первый стандарт получил название ISO/IEC 14882:1998 или сокращенно С++98. В дальнейшем в 2003 была издана новая версия стандарта C++03.

В 2011 году был издан новый стандарт C++11, который содержал множество добавлений и обогащал язык С++ большим числом новых функциональных возможностей. После этого в 2014 году было выпущено небольшое добавление к стандарту, известное также как C++14. И еще один ключевой релиз языка намечен на 2017.

Компиляторы и среды разработки

Для разработки программ на С++ необходим компилятор - он транслирует исходный код на языке С++ в исполняемый файл, который затем можно запускать. Но в настоящий момент есть очень много различных компиляторов. Они могут отличаться по различным аспектам, в частности, по реализации стандартов. Базовый список компиляторов для С++ можно посмотреть в [википедии](https://en.wikipedia.org/wiki/List_of_compilers#C.2B.2B_compilers). Рекомендуется для разработки выбирать те компиляторы, которые развиваются и реализуют все последние стандарты. Так, на протяжении всего руководства преимущественно будет использоваться свободно распространяемый компилятор g++, разработанный в рамках проекта GNU.

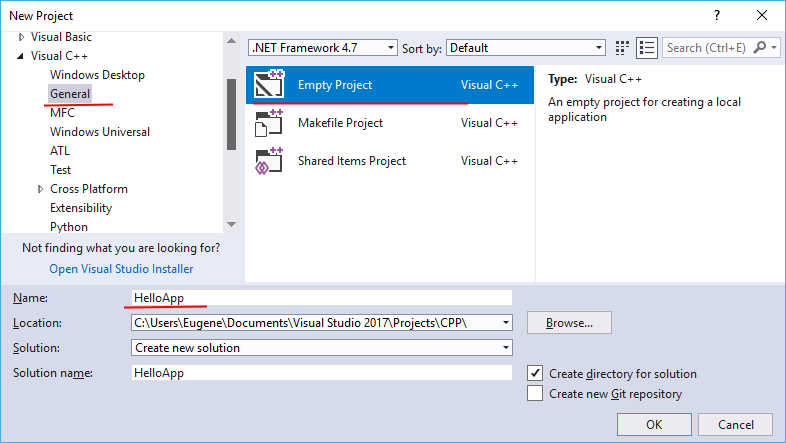
Также для создания программ можно использовать интегрированные среды разработки IDE, такие как Visual Studio, Netbeans, Eclipse, Qt и т.д.

**Первая программа**

Для создания программы на C++ нам нужны, как минимум, две вещи: текстовый редактор для набора кода и компилятор для превращения этого кода в приложение. При этом для компиляции необходимо запускать консоль или терминал. Однако есть и более удобный способ - использование различных сред разработки или IDE. Они, как правило, содержит встроенный текстовый редактор, компилятор и позволяют скомпилировать и запустить программу по одному клику мыши, а также имеют еще множество разных вспомогательных возможностей.

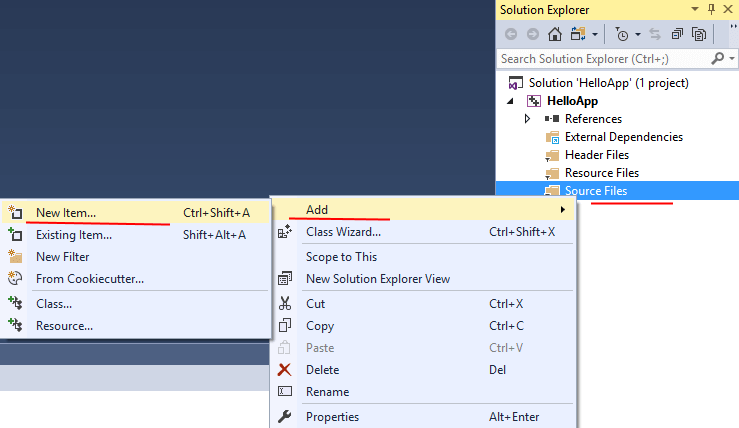
Для программирования под Windows наиболее популярной средой разработки, если говорить о C++, является Visual Studio. Данную среду можно найти по ссылке<https://www.visualstudio.com/ru/vs/>. Она доступна в ряде редакций. В частности, можно использовать бесплатную и полнофункциональную версию Visual Studio 2017 Community.

После установки Visual Studio создадим первый проект. Для этого перейдем в меню File (Файл) -> New (Создать) -> Project... (Проект), и нам откроется окно создания нового проекта. В нем перейдем в левой части окна к языку С++ и выберем его подсекцию General:

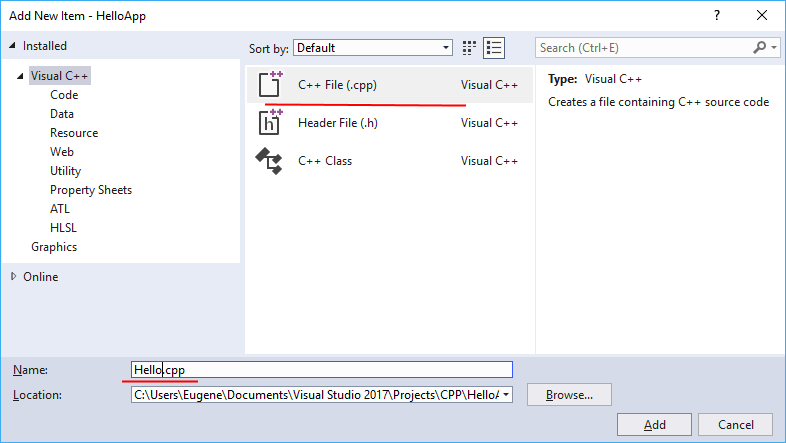


В центральной части окна в качестве типа проекта выберем Empty Project, а внизу окна в поле для имени проекта дадим проекту имя HelloApp и нажмем на ОК для создания проекта.

После этого Visual Studio создаст пустой проект. Добавим в него текстовый файл для набора исходного кода. Для этого в окне Solution Explorer (Обозреватель решений) нажмем правой кнопкой мыши на узел Source Files и в контекстом меню выберем Add -> New Item...:

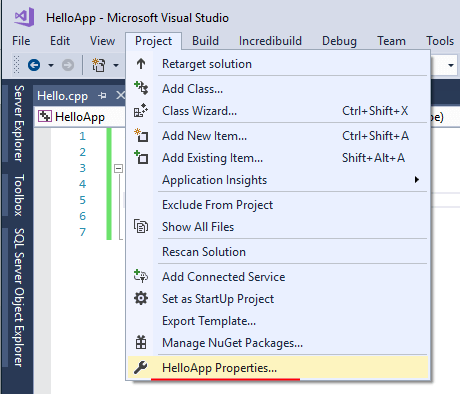


Затем нам откроется окно для добавления нового элемента:

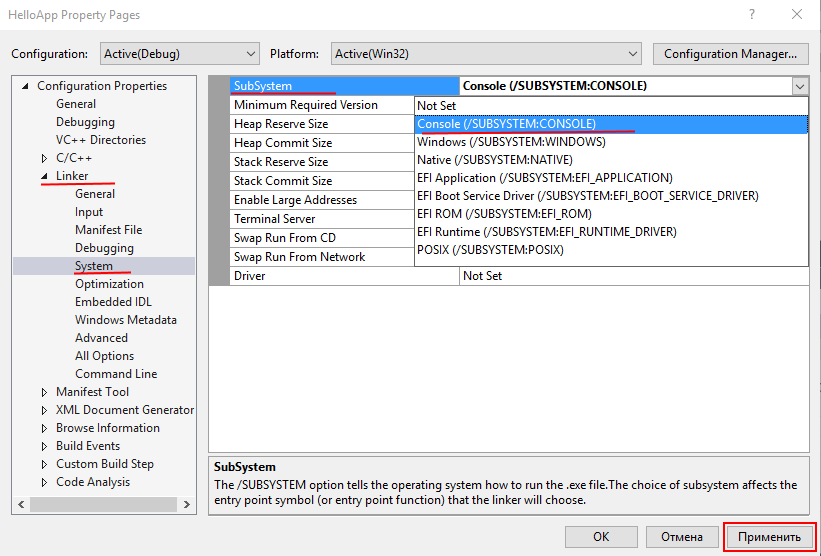


Здесь нам надо выбрать пункт C++ File(.cpp), а внизу окна укажем для файла имя Hello.cpp. Как правило, исходные файлы на C++ имеют расширение .сpp.

После добавления файла изменим опции проекта. Для этого перейдем к пункту меню Project -> Properties:

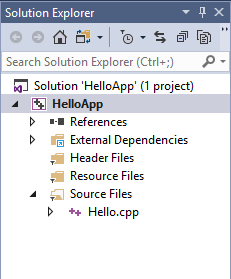


И в открывшемся окне свойств проекта в левой части перейдем к секции Linker ->System и далее для поля SubSystem установим значение Console(/SUBSYSTEM:CONSOLE), выбрав нужный элемент в списке:



Тем самым мы указываем, что мы хотим создать консольное приложение. После установки этого значения нажмем на кнопку "Применить", чтобы новые настройки конфигурации вступили в силу.

После добавления файла проект будет иметь следующую структуру:



Вкратце пробежимся по этой структуре. Окно Solution Explorer содержит в решение. В данном случае оно называется HelloApp. Решение может содержать несколько проектов. По умолчанию у нас один проект, который имеет то же имя - HelloApp. В проекте есть ряд узлов:

* External Dependencies: отображает файлы, которые используются в файлах исходного кода, но не являются частью проекта
* Header Files: предназначена для хранения заголовочных файлов с расширением .h
* Resource Files: предназначена для хранения файлов ресурсов, например, изображений
* Source Files: хранит файлы с исходным кодом

Теперь собственно создадим первую программу и определим в файле Hello.cpp простейший код, который будет выводить строку на консоль:

#include <iostream>               // подключаем заголовочный файл iostream

int main()                          // определяем функцию main

{                                   // начало функции

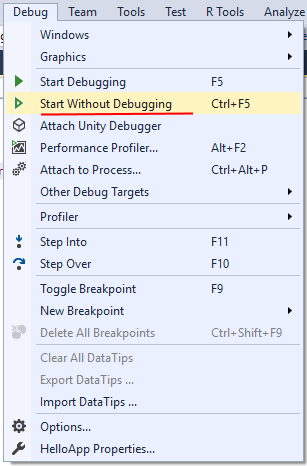
    std::cout << "Hello World!";  // выводим строку на консоль

    return 0;                       // выходим из функции

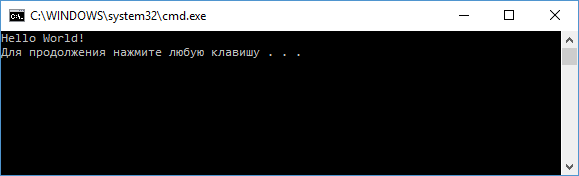
}

Здесь использован весь тот код, который был рассмотрен в предыдущих темах про компиляцию с помощью g++.

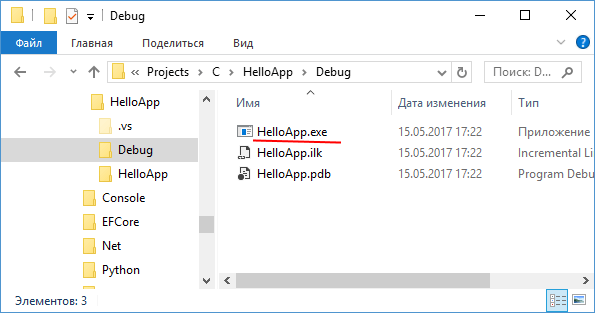
Теперь запустим программу. Для этого в Visual Studio нажмем на сочетание клавиш Ctrl+F5 или выберем пункт меню Debug -> Start Without Debugging:



И в итоге Visual Studio передаст исходный код компилятору, который скомпилирует из кода исполняемый файл exe, который потом будет запущен на выполнение. И мы увидим на запущенной консоли наше сообщение:



После этого на жестком диске в проекте в папке Debug можно увидеть скомпилированный файл exe, который мы можем запускать независимо от Visual Studio:



В данном случае файл HelloApp.exe как раз и представляет скомпилированный исполняемый файл. И, кроме того, в той же папке автоматически генерируются два вспомогательных файла:

* HelloApp.ilk: файл "incremental linker", который используется компоновщиком для ускорения компоновки
* HelloApp.pdb: файл, который содержит отладочную информацию

**Локализация и кириллица в консоли**

Если программа при выводе на консоль использует кириллицу, то мы можем столкнуться с ситуацией, когда вместо кириллических символов будут отображаются непонятные знаки. Особенно это актуально для ОС Windows. И в этом случае необходимо явным образом задать текущую локаль (культуру) для вывода символов. В языке C++ это можно сделать с помощью встроенной функции setlocale().

Итак, изменим код, который использовался в прошлых темах следующим образом:

#include <iostream>

int main()

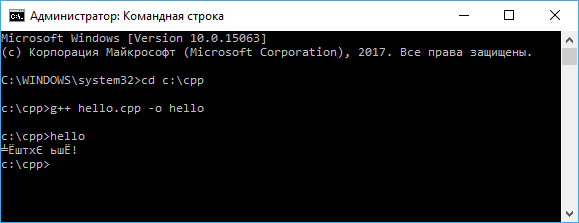
{

    std::cout << "Привет мир!";

    return 0;

}

Компиляция и запуск в ОС Windows может выглядеть следующим образом:



Вместо ожидаемого текста я получаю какие-то непонятные символы. Теперь изменим код, применив функцию setlocale:

#include <iostream>

int main()

{

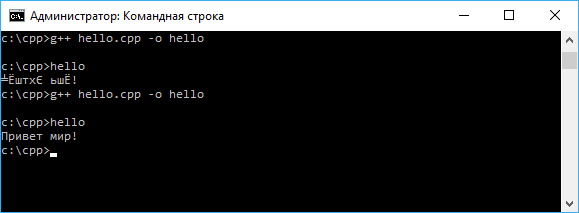
    setlocale(LC\_ALL, "");

    std::cout << "Привет мир!";

    return 0;

}

Повторно компилируем и запустим приложение:



Стоит отметить, что в качестве кодировки текстового файла в этом случае должна использоваться кодировка ANSI или Windows-1251, но не UTF-8.

**Структура программы**

Инструкции

Программа на С++ состоит из набора инструкций. Каждая инструкция (statement) выполняет определенное действие. В конце инструкции в языке C++ ставится точка с запятой (;). Данный знак указывает компилятору на завершение инструкции. Например:

std::cout << "Hello World!";

Данная строка выводит на консоль строку "Hello world!", является инструкцией и поэтому завершается точкой с запятой.

Набор инструкций может представлять блок кода. Блок кода заключается в фигурные скобки, а инструкции помещаются между открывающей и закрывающей фигурными скобками:

{

    std::cout << "Hello World!";

    std::cout << "Bye World!";

}

В этом блоке кода две инструкции, которые выводят на консоль определенную строку.

Функция main

Каждая программа на языке С++ должна иметь как минимум одну функцию - функцию main(). Именно с этой функции начинается выполнение приложения. Ее имя mainфиксировано и для всех программ на Си всегда одинаково.

Функция также является блоком кода, поэтому ее тело обрамляется фигурными скобками, между которыми определяется набор инструкций.

В частности, при создании первой программы использовалась следующая функция main:

#include <iostream>               // подключаем заголовочный файл iostream

int main()                          // определяем функцию main

{                                   // начало функции

    std::cout << "Hello World!";   // выводим строку на консоль

    return 0;                       // выходим из функции

}                                   // конец функции

Определение функии main начинается с возвращаемого типа. Функция main в любом случае должна возвращать число. Поэтому ее определение начинается с ключевого слова int.

Далее идет название функции, то есть main. После названия в скобках идет список параметров. В данном случае функция main не принимает никаких параметров, поэтому после названия указаны пустые скобки. Однако есть другие варианты определения функции main, которые подразумевыют использование параметров. В частности, нередко может встречаться следующее определение функции main, использующей параметры:

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

{

}

И после списка параметров идет блок кода, который и содержит в виде инструкций собственно те действия, выполняемые функцией main.

Директивы препроцессора

В примере выше на консоль выводится строка, но чтобы использовать вывод на консоль, необходимо в начале файла с исходным кодом подключать библиотеку iostream с помощью директивы include.

Директива include является директивой препроцессора. Каждая директива препроцессора размещается на одной строке. И в отличие от обычных инструкциий языка C++, которые завершаются точкой с запятой ; , признаком завершения препроцессорной директивы является перевод на новую строку. Кроме того, директива должна начинаться со знака решетки #. Непосредственно директива "include" определяет, какие файлы и библиотеки надо подвключить в данном месте в код программы.

Комментарии

Исходный код может содержать комментарии. Комментарии позволяют понять смыл программы, что делают те или иные ее части. При компиляции комментарии игнорируются и не оказывают никакого влияние на работу приложения и на его размер.

В языке C++ есть два типа комментариев: однострочный и многострочный. Однострочный комментарий размещается на одной строке после двойного слеша //:

#include <iostream>               // подключаем библиотеку iostream

int main()                          // определяем функцию main

{                                   // начало функции

    std::cout << "Hello World!";   // выводим строку на консоль

    return 0;                       // выходим из функции

}                                   // конец функции

Многострочный комментарий заключается между символами /\* текст комментария \*/. Он может размещаться на нескольких строках. Например:

#include <iostream>

/\*

    Определение функции Main

    Выводит на консоль строку Hello World!

\*/

int main()

{

    std::cout << "Hello World!"; // вывод строки на консоль

    return 0;

}

**Переменные**

Как и во многих языках программирования, в C++ для хранения данных используются переменные. Переменная имеет тип, имя и значение. Тип определяет, какую информацию может хранить переменная.

Перед использованием любую переменную надо определить. Синтаксис определения переменной выглядит следующим образом:

тип\_переменной имя\_переменной;

Простейшее определение переменной:

int age;

Здесь определена переменная age, которая имеет тип int. Поскольку определение переменной представляет собой инструкцию, то после него ставится точка с запятой.

Имя переменной может представлять последовательность символов латинского алфавита, чисел и знака подчеркивания. При этом имя должно начинаться либо с алфавитного символа, либо со знака подчеркивания.

int \_age33;

Других символов в названии переменной не должно быть. Например, следующее определение будет неправильным:

int $age;

Так как символ $ не является ни буквой, ни цифрой, ни символом подчеркивания.

Также стоит учитывать, что C++ - регистрозависимый язык, а это значит, что регистр символов имеет большое значение. То есть в следующем коде будут определяться две разные переменные:

int age;

int Age;

Поэтому переменная Age не будет представлять то же самое, что и переменная age.

Кроме того, в качестве имени переменной нельзя использовать ключевые слова языке C++, например, for или if. Но таких слов не так много: alignas, alignof, asm, auto, bool, break, case, catch, char, char16\_t, char32\_t, class, const, constexpr, const\_cast, continue, decltype, default, delete, do, double, dynamic\_cast, else, enum, explicit, export, extern, false, float, for, friend, goto, if, inline, int, long, mutubale, namespace, new, noexcept, nullptr, operator, private, protected, public, register, reinterpret\_cast, return, short, signed, sizeof, static, static\_assert, static\_cast, struct, switch, template, this, thread\_local, throw, true, try, typedef, typeid, typename, union, unsigned, using, virtual, void, volatile, wchar\_t, while.

Также нельзя объявить больше одной переменной с одним и тем же именем, например:

int age;

int age;

Подобное определение вызовет ошибку на этапе компиляции.

И в довершеие следует сказать, что переменным стоит давать осмысленные имена, которые будут говорить об их предназначении.

Инициализация

После определения переменной можно присвоить некоторое значение:

int age;

age = 20;

Например, определим в прогамме переменную и выведем ее значение на консоль:

#include <iostream>

int main()

{

    int age;

    age = 28;

    std::cout<<"Age = " << age;

    return 0;

}

С помощью последовательности операторов << можно вывести несколько значений на консоль.

После компиляции и запуска скомпилированной программы на консоль будет выведено число 28.

Однако также можно сразу при определении переменной дать ей некоторое начальное значение. Данный прием называется инициализацией, то есть присвоение переменной начального значения:

#include <iostream>

int main()

{

    int age = 28;

    std::cout<<"Age = " << age;

    return 0;

}

Инициализация по умолчанию

Если переменную не инициализировать, то происходит ее инициализация по умолчанию. И переменная получает некоторое значение по умолчанию, которое зависит от места, где эта переменная определена.

Если переменная, которая представляет встроенный тип (например, тип int), определена внутри функции, то она получает неопределенное значение. Если переменная встроенного типа определена вне функции, то она получает то значение по умолчанию, которое соответствует ее типу. Для числовых типов это число 0. Например:

#include <iostream>

int x;

int main()

{

    int y;

    std::cout <<"X = " << x << "\n";

    std::cout <<"Y = " << y;

    return 0;

}

Переменная x определена вне функции, и поэтому она получит значение по умолчанию - число 0. А переменная y определена внутри функции main, поэтому ее значение будет неопределенным. В частности, вывод программы на консоль может выглядеть следующим образом:

X = 0

Y = 4200475

Но в любом случае перед использованием переменной лучше явным образом назначать ей определенное значение, а не полагаться на значение по умолчанию.

Изменение значения

Ключевой особенностью переменных является то, что мы можем изменять их значения:

#include <iostream>

int main()

{

    int x = 6;

    x = 8;

    x = 10;

    std::cout <<"X = " << x; // X = 10

    return 0;

}

**Типы данных**

Каждая переменная имеет определенный тип. И этот тип определяет, какие значения может иметь переменная, какие операции с ней можно производить и сколько байт в памяти она будет занимать. В языке C++ определены следующие базовые типы данных:

* bool: логический тип. Может принимать одну из двух значений true (истина) иfalse (ложь). Размер занимаемой памяти для этого типа точно не определен.
* char: представляет один символ в кодировке ASCII. Занимает в памяти 1 байт (8 бит). Может хранить любое значение из диапазона от -128 до 127, либо от 0 до 255
* signed char: представляет один символ. Занимает в памяти 1 байт (8 бит). Может хранить любой значение из диапазона от -128 до 127
* unsigned char: представляет один символ. Занимает в памяти 1 байт (8 бит). Может хранить любой значение из диапазона от 0 до 255
* wchar\_t: представляет расширенный символ. На Windows занимает в памяти 2 байта (16 бит), на Linux - 4 байта (32 бита). Может хранить любой значение из диапазона от 0 до 65 535 (при 2 байтах), либо от 0 до 4 294 967 295 (для 4 байт)
* char16\_t: представляет один символ в кодировке Unicode. Занимает в памяти 2 байта (16 бит). Может хранить любой значение из диапазона от 0 до 65 535
* char32\_t: представляет один символ в кодировке Unicode. Занимает в памяти 4 байта (32 бита). Может хранить любой значение из диапазона от 0 до 4 294 967 295
* short: представляет целое число в диапазоне от –32768 до 32767. Занимает в памяти 2 байта (16 бит).

Данный тип также имеет синонимы short int, signed short int, signed short.

unsigned short: представляет целое число в диапазоне от 0 до 65535. Занимает в памяти 2 байта (16 бит).

Данный тип также имеет синоним unsigned short int.

int: представляет целое число. В зависимости от архитектуры процессора может занимать 2 байта (16 бит) или 4 байта (32 бита). Диапазон предельных значений соответственно также может варьироваться от –32768 до 32767 (при 2 байтах) или от −2 147 483 648 до 2 147 483 647 (при 4 байтах). Но в любом случае размер должен быть больше или равен размеру типа short и меньше или равен размеру типа long

Данный тип имеет синонимы signed int и signed.

unsigned int: представляет положительное целое число. В зависимости от архитектуры процессора может занимать 2 байта (16 бит) или 4 байта (32 бита), и из-за этого диапазон предельных значений может меняться: от 0 до 65535 (для 2 байт), либо от 0 до 4 294 967 295 (для 4 байт).

В качестве синонима этого типа может использоваться unsigned

long: представляет целое число в диапазоне от −2 147 483 648 до 2 147 483 647. Занимает в памяти 4 байта (32 бита).

У данного типа также есть синонимы long int, signed long int и signed long

unsigned long: представляет целое число в диапазоне от 0 до 4 294 967 295. Занимает в памяти 4 байта (32 бита).

Имеет синоним unsigned long int.

long long: представляет целое число в диапазоне от −9 223 372 036 854 775 808 до +9 223 372 036 854 775 807. Занимает в памяти, как правило, 8 байт (64 бита).

Имеет синонимы long long int, signed long long int и signed long long.

unsigned long long: представляет целое число в диапазоне от 0 до 18 446 744 073 709 551 615. Занимает в памяти, как правило, 8 байт (64 бита).

Имеет синоним unsigned long long int.

float: представляет вещественное число ординарной точности с плавающей точкой в диапазоне +/- 3.4E-38 до 3.4E+38. В памяти занимает 4 байта (32 бита)

double: представляет вещественное число двойной точности с плавающей точкой в диапазоне +/- 1.7E-308 до 1.7E+308. В памяти занимает 8 байт (64 бита)

long double: представляет вещественное число двойной точности с плавающей точкой не менее 8 байт (64 бит). В зависимости от размера занимаемой памяти может отличаться диапазон допустимых значений.

void: тип без значения

Таким образом, все типы данных за исключением void могут быть разделены на три группы: символьные (char, wchar\_t, char16\_t, char32\_t), целочисленные (short, int, long, long long) и типы чисел с плавающей точкой (float, double, long double).

Символьные типы

Для представления символов в приложении используются типы char, wchar\_t, char16\_tи char32\_t.

Определим несколько переменных:

char c ='d';

wchar\_t d ='c';

Переменная типа char в качестве значения принимает один символ в одинарных кавычках: char c ='d'. Также можно присвоить число из указанного выше в списке диапазона: char c = 120. В этом случае значением переменной c будет тот символ, который имеет код 120 в таблице символов ASCII.

Стоит учитывать, что для вывода на консоль символов wchar\_t следует использовать не std::cout, а поток std::wcout:

#include <iostream>

int main()

{

    char a = 'H';

    wchar\_t b = 'e';

    std::wcout << a << b << '\n';

    return 0;

}

При этом поток std::wcout может работать как с char, так и с wchar\_t. А поток std::cout для переменной wchar\_t вместо символа будет выводить его числовой код.

В стандарте С++11 были добавлены типы char16\_t и char32\_t, которые ориентированы на использование Unicode. Однако на уровне ОС пока не реализованы потоки для работы с этими типами. Поэтому если потребуется вывести на консоль значения переменных этих типов, то необходимо преобразовать переменные к типам char или wchar\_t:

#include <iostream>

int main()

{

    char a = 'H';

    wchar\_t b = 'e';

    char16\_t c = 'l';

    char32\_t d = 'o';

    std::cout << a << (char)b << (char)c << (char)d << "\n";

    return 0;

}

В данном случае при выводе перед переменными указывается операция приведения к типу char - (char), благодаря чему значения переменных b, c и d преобразуются в тип char и могут быть выведены на консоль с помощью потока std::cout.

Целочисленные типы

Целочисленные типы представлены следующими типами: short, unsigned short, int,unsigned int, long, unsigned long, long long и unsigned long long:

* short a = -10;
* unsigned short b= 10;
* int c = -30;
* unsigned int d = 60;
* long e = -170;
* unsigned long f = 45;
* long long g = 89;

Типы чисел с плавающей точкой

Типы чисел с плавающей точкой иили дробные числа представлены такими типами как float, double и long double:

* float a = -10.45;
* double b = 0.00105;
* long double c = 30.890045;

Размеры типов данных

В выше приведенном списке для каждого типа указан размер, который он занимает в памяти. Однако стоит отметить, что предельные размеры для типов разработчики компиляторов могут выбирать самостоятельно, исходя из аппаратных возможностей компьютера. Стандарт устанавливает лишь минимальные значения, которые должны быть. Например, для типов int и short минимальное значение - 16 бит, для типа long - 32 бита, для типа long double. При этом размер типа long должен быть не меньше размера типа int, а размер типа int - не меньше размера типа short, а размер типа long double должен быть больше double. К примеру, компилятор g++ под Windows для long double использует 12 байт, а компилятор, встроенный в Visual Studio и также работающий под Windows, для long double использует 8 байт. То есть даже в рамках одной платформы разные компиляторы могут по разному подходить к размерам некоторых типов данных. Но в целом используются те размеры, которые указаны выше при описании типов данных.

Однако бывают ситуации, когда необходимо точно знать размер определенного типа. И для этого в С++ есть оператор sizeof(), который возвращает размер памяти в байтах, которую занимает переменная:

#include <iostream>

int main()

{

    long double number = 2;

    std::cout << "sizeof(number) =" << sizeof(number);

    return 0;

}

При этом при определении переменных важно понимать, что значение переменной не должно выходить за те пределы, которые очерчены для ее типа. Например:

unsigned short number = -65535;

Компилятор G++ при компиляции программы с этой строкой выдаст ошибку о том, что значение -65535 не входит в диапазон допустимых значений для типа unsigned short и будет усечено.

В Visual Studio компиляция может пройти без ошибок, однако при этом переменная number получит значение 2 - результат преобразования числа -65535 к типу unsigned short. То есть опять же результат будет не совсем тот, который ожидается. Значение переменной - это всего лишь набор битов в памяти, которые интерпретируются в соответствии с определенным типом. И для разных типов один и тот же набор битов может интерпретироваться по разному. Поэтому важно учитывать диапазоны значений для того или иного типа при присвоении переменной значения.

Спецификатор auto

Иногда бывает трудно определить тип выражения. И согласно последним стандартам можно предоставить компилятору самому выводить тип объекта. И для этого применяется спецификатор auto. При этом если мы определяем переменную со спецификатором auto, эта переменная должна быть обязательно инициализирована каким-либо значением:

auto number = 5;

На основании присвоенного значения компилятор выведет тип переменной. Неинициализированные переменные со спецификатором auto не допускаются:

auto number;

**Конструкция if**

Конструкция **if** проверяет истинность условия, и если оно истинно, выполняет блок инструкций. Этот оператор имеет следующую сокращенную форму:

if (условие){

  инструкции;

}

В качестве *условия* использоваться условное выражение, которое возвращает **true** или **false**. Если условие возвращает true, то выполняются последующие инструкции, которые входят в блок if. Если условие возвращает false, то последующие инструкции не выполняются. Блок инструкций заключается в фигурные скобки.

Например:

#include <iostream>

int main()

{

    int x = 60;

    if(x > 50)

    {

        std::cout << "x is greater than 50 \n";

    }

    if(x < 30)

    {

        std::cout << "x is less than 30 \n";

    }

    std::cout << "End of Program" << "\n";

    return 0;

}

Здесь определены две условных конструкции if. Они проверят больше или меньше значение переменной x, чем определенное значение. В качестве инструкции в обоих случаях выполняется вывод некоторой строки на консоль.

В первом случае x > 50 условие истинно, так как значение переменной x действительно больше 50, поэтому это условие возвратит true, и, следовательно, будут выполнятся инструкции, которые входят в блок if.

Во втором случае операция отношения x < 30 возвратит false, так как условие ложно, поэтому последующий блок инструкций выполняться не будет. В итоге при запуске программы вывод консоли будет выглядеть следующим образом:

x greater than 50

End of Program

Также мы можем использовать полную форму конструкции if, которая включает оператор else:

if(выражение\_условия)

    инструкция\_1

else

    инструкция\_2

После оператора else мы можем определить набор инструкций, которые выполняются, если условие в операторе if возвращает false. То есть если *условие* истинно, выполняются инструкции после оператора if, а если это выражение ложно, то выполняются инструкции после оператора else.

int x = 50;

if(x > 60)

    std::cout << "x is greater than 60 \n";

else

    std::cout << "x is less or equal 60 \n";

В данном случае условие x > 60 ложно, то есть возвращает false, поэтому будет выполняться блок else. И в итоге на консоль будет выведена строка "x is less or equal 60 \n".

Однако нередко надо обработать не два возможных альтернативных варианта, а гораздо больше. Например, в случае выше можно насчитать три условия: переменная x может быть больше 60, меньше 60 и равна 60. Для проверки альтернативных условий мы можем вводить выражения **else if**:

int x = 60;

if(x > 60)

{

    std::cout << "x is greater than 60 \n";

}

else if (x < 60)

{

    std::cout << "x is less than 60 \n";

}

else

{

    std::cout << "x is equal 60 \n";

}

То есть в данном случае мы получаем три ветки развития событий в программе.

Если в блоке if или else или else-if необходимо выполнить только одну инструкцию, то фигурные скобки можно опустить:

int x = 60;

if(x > 60)

    std::cout << "x is greater than 60 \n";

else if (x < 60)

    std::cout << "x is less than 60 \n";

else

    std::cout << "x is equal 60 \n";

**Конструкция switch**

Другую форму организации ветвления программ представляет конструкция **switch...case**. Она имеет следующую форму:

switch(выражение)

{

    case константа\_1: инструкции\_1;

    case константа\_2: инструкции\_2;

    default: инструкции;

}

После ключевого слова **switch** в скобках идет сравниваемое выражение. Значение этого выражения последовательно сравнивается со значениями после оператора **сase**. И если совпадение будет найдено, то будет выполняться определенный блок **сase**.

В конце конструкции switch может стоять блок **default**. Он необязателен и выполняется в том случае, если значение после switch не соответствует ни одному из операторов case. Например:

#include <iostream>

int main()

{

    int x = 2;

    switch(x)

    {

        case 1:

            std::cout << "x = 1" << "\n";

            break;

        case 2:

            std::cout << "x = 2" << "\n";

            break;

        case 3:

            std::cout << "x = 3" << "\n";

            break;

        default:

            std::cout << "x is undefined" << "\n";

            break;

    }

    return 0;

}

Чтобы избежать выполнения последующих блоков case/default, в конце каждого блока ставится оператор **break**. То есть в данном случае будет выполняться оператор

case 2:

    std::cout << "x = 2" << "\n";

    break;

После выполнения оператора break произойдет выход из конструкции switch..case, и остальные операторы case будут проигнорированы. Поэтому на консоль будет выведена следующая строка

x = 2

Стоит отметить важность использования оператора break. Если мы его не укажем в блоке case, то после этого блока выполнение перейдет к следующему блоку case. Например, уберем из предыдущего примера все операторы break:

#include <iostream>

int main()

{

    int x = 2;

    switch(x)

    {

        case 1:

            std::cout << "x = 1" << "\n";

        case 2:

            std::cout << "x = 2" << "\n";

        case 3:

            std::cout << "x = 3" << "\n";

        default:

            std::cout << "x is undefined" << "\n";

    }

    return 0;

}

В этом случае опять же будет выполняться оператор case 2:, так как переменная x=2. Однако так как этот блок case не завершается оператором break, то после его завершения будет выполняться набор инструкций после case 3: даже несмотря на то, что переменная x по прежнему равна 2. В итоге мы получим следующий консольный вывод:

x = 2

x = 3

x is undefined

**Циклы**

Для выполнения некоторых действий множество раз в зависимости от определенного условия используются циклы. В языке C++ имеются следующие виды циклов:

**for**

**while**

**do...while**

Цикл while

Цикл while выполняет некоторый код, пока его условие истинно, то есть возвращает true. Он имеет следующее формальное определение:

while(условие)

{

    // выполняемые действия

}

После ключевого слова **while** в скобках идет условное выражение, которое возвращает true или false. Затем в фигурных скобках идет набор инструкций, которые составляют тело цикла. И пока условие возвращает true, будут выполняться инструкции в теле цикла.

Например, выведем квадраты чисел от 1 до 9:

#include <iostream>

int main()

{

    int i = 1;

    while(i < 10)

    {

        std::cout << i << " \* " << i << " = " << i \* i << std::endl;

        i++;

    }

    return 0;

}

Здесь пока условие i < 10 истинно, будет выполняться цикл while, в котором выводится на консоль квадрат числа и инкрементируется переменная i. В какой-то момент переменная i увеличится до 10, условие i < 10 возвратит false, и цикл завершится.

Консольный вывод программы:

1 \* 1 = 1

2 \* 2 = 4

3 \* 3 = 9

4 \* 4 = 16

5 \* 5 = 25

6 \* 6 = 36

7 \* 7 = 49

8 \* 8 = 64

9 \* 9 = 81

Каждый отдельный проход цикла называется итерацией. То есть в примере выше было 9 итераций.

Если цикл содержит одну инструкцию, то фигурные скобки можно опустить:

int i = 0;

while(++i < 10)

    std::cout << i << " \* " << i << " = " << i \* i << std::endl;

Цикл for

Цикл for имеет следующее формальное определение:

for (выражение\_1; выражение\_2; выражение\_3)

{

    // тело цикла

}

*выражение\_1* выполняется один раз при начале выполнения цикла и представляет установку начальных условий, как правило, это инициализация счетчиков - специальных переменных, которые используются для контроля за циклом.

*выражение\_2* представляет условие, при соблюдении которого выполняется цикл. Как правило, в качестве условия используется операция сравнения, и если она возвращает ненулевое значение (то есть условие истинно), то выполняется тело цикла, а затем вычисляется *выражение\_3*.

*выражение\_3* задает изменение параметров цикла, нередко здесь происходит увеличение счетчиков цикла на единицу.

Например, перепишем программу по выводу квадратов чисел с помощью цикла for:

#include <iostream>

int main()

{

    for(int i =1; i < 10; i++)

    {

        std::cout << i << " \* " << i << " = " << i \* i << std::endl;

    }

    return 0;

}

Первая часть объявления цикла - int i = 1 - создает и инициализирует счетчик i. Фактически это то же самое, что и объявление и инициализация переменной. Счетчик необязательно должен представлять тип int. Это может быть и другой числовой тип, например, float. И перед выполнением цикла его значение будет равно 1.

Вторая часть - условие, при котором будет выполняться цикл. В данном случае цикл будет выполняться, пока переменная i не станет равна 10.

И третья часть - приращение счетчика на единицу. Опять же нам необязательно увеличивать на единицу. Можно уменьшать: i--. Можно изменять на другое значение: i+=2.

В итоге блок цикла сработает 9 раз, пока переменная i не станет равна 10. И каждый раз это значение будет увеличиваться на 1. И по сути мы получим тот же самый результат, что и в случае с циклом while:

1 \* 1 = 1

2 \* 2 = 4

3 \* 3 = 9

4 \* 4 = 16

5 \* 5 = 25

6 \* 6 = 36

7 \* 7 = 49

8 \* 8 = 64

9 \* 9 = 81

Необязательно указывать все три выражения в определении цикла, мы можем одно или даже все из них опустить:

int i = 1;

for(; i < 10;)

{

    std::cout << i << " \* " << i << " = " << i \* i << std::endl;

    i++;

}

Формально определение цикла осталось тем же, только теперь первое и третье выражения в определении цикла отсутствуют: for (; i < 10;). Переменная-счетчик определена и инициализирована вне цикла, а ее приращение происходит в самом цикле.

Можно определять вложенные циклы. Например, выведем таблицу умножения:

#include <iostream>

int main()

{

    for (int i=1; i < 10; i++)

    {

        for(int j = 1; j < 10; j++)

        {

            std::cout << i \* j << "\t";

        }

        std::cout << std::endl;

    }

    return 0;

}

Цикл do

В цикле do сначала выполняется код цикла, а потом происходит проверка условия в инструкции while. И пока это условие истинно, то есть не равно 0, то цикл повторяется. Формальное определение цикла:

do

{

    инструкции

}

while(условие);

Например:

#include <iostream>

int main()

{

    int i = 6;

    do

    {

        std::cout << i << std::endl;

        i--;

    }

    while(i>0);

    return 0;

}

Здесь код цикла сработает 6 раз, пока i не станет равным нулю.

Но важно отметить, что цикл do гарантирует хотя бы однократное выполнение действий, даже если условие в инструкции while не будет истинно. То есть мы можем написать:

int i = -1;

do

{

    std::cout << i << std::endl;

    i--;

}

while(i>0);

}

Хотя у нас переменная i меньше 0, цикл все равно один раз выполнится.

Операторы continue и break

Иногда возникает необходимость выйти из цикла до его завершения. В этом случае можно воспользоваться оператором **break**. Например:

#include <iostream>

int main()

{

    int i = 1;

    for ( ; ; )

    {

        std::cout << i << " \* " << i << " = " << i \* i << std::endl;

        i++;

        if (i > 9) break;

    }

    return 0;

}

Здесь когда значение переменной i достигнет 10, осуществляется выход из цикла с помощью оператора break.

В отличие от оператора break, оператор **continue** производит переход к следующей итерации. Например, нам надо посчитать сумму только нечетных чисел из некоторого диапазона:

#include <iostream>

int main()

{

    int result = 0;

    for (int i=0; i<10; i++)

    {

        if (i % 2 == 0) continue;

        result +=i;

    }

    std::cout << "result = " << result << std::endl; // 25

    return 0;

}

Чтобы узнать, четное ли число, мы получаем остаток от целочисленного деления на 2, и если он равен 0, то с помощью оператора continue переходим к следующей итерации цикла. А если число нечетное, то складываем его с остальными нечетными числами.

**Массив**

Массив представляет набор однотипных данных. Формальное определение массива выглядит следующим образом:

тип\_переменной название\_массива [длина\_массива]

После типа переменной идет название массива, а затем в квадратных скобках его размер. Например, определим массив из 4 чисел:

int numbers[4];

Данный массив имеет четыре числа, но все эти числа имеют неопределенное значение. Однако мы можем выполнить инициализацию и присвоить этим числам некоторые начальные значения через фигурные скобки:

int numbers[4] = {1,2,3,4};

Значения в фигурных скобках еще называют инициализаторами. Если инициализаторов меньше, чем элементов в массиве, то инициализаторы используются для первых элементов. Если в инициализаторов больше, чем элементов в массиве, то при компиляции возникнет ошибка:

int numbers[4] = {1, 2, 3, 4, 5, 6};

Здесь массив имеет размер 4, однако ему передается 6 значений.

Если размер массива не указан явно, то он выводится из количества инициализаторов:

int numbers[] = {1, 2, 3, 4, 5, 6};

В данном случае в массиве есть 6 элементов.

Свои особенности имеет инициализация символьных массивов. Мы можем передать символьному массиву как набор инициализаторов, так и строку:

char s1[] = {'h', 'e', 'l', 'l', 'o'};

char s2[] = "world";

Причем во втором случае массив s2 будет иметь не 5 элементов, а 6, поскольку при инициализации строкой в символьный массив автоматически добавляется нулевой символ '\0'.

При этом не допускается присвоение одному массиву другого массива:

int nums1[] = {1,2,3,4,5};

int nums2[] = nums1;    // ошибка

nums2 = nums1;          // ошибка

После определения массива мы можем обратиться к его отдельным элементам по индексу. Индексы начинаются с нуля, поэтому для обращения к первому элементу необходимо использовать индекс 0. Обратившись к элементу по индексу, мы можем получить его значение, либо изменить его:

#include <iostream>

int main()

{

    int numbers[4] = {1,2,3,4};

    int first\_number = numbers[0];

    std::cout << first\_number << std::endl; // 1

    numbers[0] = 34;                        // изменяем элемент

    std::cout << numbers[0] << std::endl; // 34

    return 0;

}

Число элементов массива также можно определять через константу:

const int n = 4;

int numbers[n] = {1,2,3,4};

Перебор массивов

Используя циклы, можно пробежаться по всему массиву и через индексы обратиться к его элементам:

#include <iostream>

int main()

{

    int numbers[4] = {1,2,3,4};

    int size = sizeof(numbers)/sizeof(numbers[0]);

    for(int i=0; i < size; i++)

        std::cout << numbers[i] << std::endl;

    return 0;

}

Чтобы пройтись по массиву в цикле, вначале надо найти длину массива. Для нахождения длины применяется оператор **sizeof**. По сути длина массива равна совокупной длине его элементов. Все элементы представляют один и тот же тип и занимают один и тот же размер в памяти. Поэтому с помощью выражения sizeof(numbers) находим длину всего массива в байтах, а с помощью выражения sizeof(numbers[0]) - длину одного элемента в байтах. Разделив два значения, можно получить количество элементов в массиве. А далее с помощью цикла for перебираем все элементы, пока счетчик i не станет равным длине массива. В итоге на консоль будут выведены все элементы массива:

Но также есть и еще одна форма цикла **for**, которая предназначена специально для работа с коллекциями, в том числе с массивами. Эта форма имеет следующее формальное определение:

for(тип переменная : коллекция)

{

    инструкции;

}

Используем эту форму для перебора массива:

#include <iostream>

int main()

{

    int numbers[4] = {1,2,3,4};

    for(int number : numbers)

        std::cout << number << std::endl;

    return 0;

}

При переборе массива каждый перебираемый элемент будет помещаться в переменную number, значение которой в цикле выводится на консоль.

Если нам неизвестен тип объектов в массиве, то мы можем использовать спецификатор auto для определения типа:

for(auto number : numbers)

    std::cout << number << std::endl;

**Функции**

Определение функции

Функция определяет действия, которые выполняет программа. Функции позволяют выделить набор инструкций и придать ему имя. А затем многократно по присвоенному имени вызывать в различных частях программы. По сути функция - это именованный блок кода.

Формальное определение функции выглядит следующим образом:

тип имя\_функции(параметры)

{

    инструкции

}

Первая строка представляет заголовок функции. Вначале указывается возвращаемый тип функции. Если функция не возвращает никакого значения, то используется тип **void**.

Затем идет имя функции, которое представляет произвольный идентификатор. К именованию функции применяются те же правила, что и к именованию переменных.

После имени функции в скобках идет перечисление параметров. Функция может не иметь параметров, в этом случае указываются пустые скобки.

После заголовка функции в фигурных скобках идет тело функции, которое содержит выполняемые инструкции.

Для возвращения результата функция применяет оператор **return**. Если функция имеет в качестве возвращаемого типа любой тип, кроме void, то она должна обязательно с помощью оператора return возвращать какое-либо значение.

Например, определение функции main, которая должна быть в любой программе на языке C++ и с которой начинается ее выполнение:

int main()

{

    return 0;

}

Возвращаемым типом функции является тип **int**, поэтому функция должна использовать оператор **return** и возвращать какое-либо значение, которое соответствует типу int. Возвращаемое значение ставится после оператора return.

Но если функция имеет тип **void**, то ей не надо ничего возвращать. Например, мы могли бы определить следующую функцию:

void hello()

{

    std::cout << "hello\n";

}

Выполнение функции

Для выполнения функции ее необходимо вызвать. Вызов функции осуществляется в форме:

имя\_функции(аргументы);

После имени функции указываются скобки, в которых перечисляются аргументы - значения для параметров функции.

Например, определим и выполним простейшую функцию:

#include <iostream>

void hello()

{

    std::cout << "hello\n";

}

int main()

{

    hello();

    hello();

    return 0;

}

Здесь определена функция hello, которая вызывается в функции main два раза. В этом и заключается преимущество функций: мы можем вынести некоторые общие действия в отдельную функцию и затем вызывать многократно в различных местах программы. В итоге программа два раза выведет строку "hello".

hello

hello

Объявление функции

При использовании функций стоит учитывать, что компилятор должен знать о функции до ее вызова. Поэтому вызов функции должен происходить после ее определения, как в случае выше. В некоторых языках это не имеет значение, но в языке C++ это играет большую роль. И если, к примеру, мы сначала вызовем, а потом определим функцию, то мы получим ошибку на этапе компиляции, как в следующем случае:

#include <iostream>

int main()

{

    hello();

    hello();

    return 0;

}

void hello()

{

    std::cout << "hello\n";

}

В этом случае перед вызовом функции надо ее дополнительно объявить. Объявление функции еще называют прототипом. Формальное объявление выглядит следующим образом:

тип имя\_функции(параметры);

Фактически это заголовок функции. То есть для функции hello объявление будет выглядеть следующим образом:

void hello();

Используем объявление функции:

#include <iostream>

void hello();

int main()

{

    hello();

    hello();

    return 0;

}

void hello()

{

    std::cout << "hello\n";

}

В данном случае несмотря на то, что определение функции идет после ее вызова, но так как функция уже объявлена до ее вызова, то компилятор уже будет знать о функции hello, и никаких проблем в работе программы не возникнет.

**Двумерные массивы**

Кроме одномерных массивов в C++ есть многомерные. Элементы таких массивов сами в свою очередь являются массивами, в которых также элементы могут быть массивами. Например, определим двухмерный массив чисел:

int numbers[3][2];

Такой массив состоит из трех элементов, при этом каждый элемент представляет массив из двух элементов. Инициализируем подобный массив:

int numbers[3][2] = { {1, 2}, {4, 5}, {7, 8} };

Вложенные фигурные скобки очерчивают элементы для каждого подмассива. Такой массив еще можно представить в виде таблицы:

Также при инициализации можно опускать фигурные скобки:

int numbers[3][2] = { 1, 2, 4, 5, 7, 8 };

Возможна также инициализация не всех элементов, а только некоторых:

int numbers[3][2] = { {1, 2}, {}, {7} };

И чтобы обратиться к элементам вложенного массива, потребуется два индекса:

int numbers[3][2] = { {1, 2}, {3, 4}, {5, 6} };

std::cout << numbers[1][0] << std::endl;    // 3

numbers[1][0] = 12;             // изменение элемента

std::cout << numbers[1][0] << std::endl;    // 12

Переберем двухмерный массив:

#include <iostream>

int main()

{

    const int rows = 3, columns = 2;

    int numbers[rows][columns] = {{1,2},{3,4},{5,6} };

    for(int i=0; i < rows; i++)

    {

        for(int j=0; j < columns; j++)

        {

            std::cout << numbers[i] [j] << "\t";

        }

        std::cout << std::endl;

    }

    return 0;

}

Также для перебора элементов многомерного массива можно использовать другую форму цикла for:

#include <iostream>

int main()

{

    const int rows = 3, columns = 2;

    int numbers[rows][columns] = { {1, 2}, {3, 4},

{5, 6} };

    for(auto &subnumbers : numbers)

    {

        for(int number : subnumbers)

        {

            std::cout << number << "\t";

        }

        std::cout << std::endl;

    }

    return 0;

}

Для перебора массивов, которые входят в массив, применяются ссылки. То есть во внешнем цикле for(auto &subnumbers : numbers) &subnumbers представляет ссылку на подмассив в массиве. Во внутреннем цикле for(int number : subnumbers) из каждого подмассива в subnumbers получаем отдельные его элементы в переменную number и выводим ее значение на консоль.

**Работа с текстовыми файлами**

Потоки для работы с текстовыми файлами представляют объекты, для которых не задан режим открытия **ios::binary**.

Запись в файл

Для записи в файл к объекту ofstream или fstream применяется оператор **<<** (как и при выводе на консоль):

#include <iostream>

#include <fstream>

int main()

{

    std::ofstream out;          // поток для записи

    out.open("D:\\hello.txt"); // окрываем файл для

записи

    if (out.is\_open())

    {

        out << "Hello World!" << std::endl;

    }

    std::cout << "End of program" << std::endl;

    return 0;

}

Данный способ перезаписывает файл заново. Если надо дозаписать текст в конец файла, то для открытия файла нужно использовать режим **ios::app**:

std::ofstream out("D:\\hello.txt", std::ios::app);

if (out.is\_open())

{

    out << "Welcome to CPP" << std::endl;

}

out.close();

Чтение из файла

Если надо считать всю строку целиком или даже все строки из файла, то лучше использовать встроенную функцию **getline()**, которая принимает поток для чтения и переменную, в которую надо считать текст:

#include <iostream>

#include <fstream>

#include <string>

int main()

{

    std::string line;

    std::ifstream in("D:\\hello.txt"); // окрываем файл

для чтения

    if (in.is\_open())

    {

        while (getline(in, line))

        {

            std::cout << line << std::endl;

        }

    }

    in.close();     // закрываем файл

    std::cout << "End of program" << std::endl;

    return 0;

}

|  |
| --- |
|  |

Также для чтения данных из файла для объектов ifstream и fstream может применяться оператор **>>** (также как и при чтении с консоли):

#include <iostream>

#include <fstream>

#include <vector>

struct Operation

{

    int sum;        // купленная сумма

    double rate;    // по какому курсу

    Operation(double s, double r) : sum(s), rate(r)

    {}

};

int main()

{

    std::vector<Operation> operations = {

        Operation(120, 57.7),

        Operation(1030, 57.4),

        Operation(980, 58.5),

        Operation(560, 57.2)

    };

    std::ofstream out("D:\\operations.txt");

    if (out.is\_open())

    {

        for (int i = 0; i < operations.size(); i++)

        {

            out << operations[i].sum << " " <<

operations[i].rate << std::endl;

        }

    }

    out.close();

    std::vector<Operation> new\_operations;

    double rate;

    int sum;

    std::ifstream in("D:\\operations.txt"); // открываем

файл для чтения

    if (in.is\_open())

    {

        while (in >> sum >> rate)

        {

          new\_operations.push\_back(Operation(sum, rate));

        }

    }

    in.close();

    for (int i = 0; i < new\_operations.size(); i++)

    {

        std::cout << new\_operations[i].sum << " - " <<

new\_operations[i].rate << std::endl;

    }

    return 0;

}

Здесь вектор структур Operation записывается в файл.

for (int i = 0; i < operations.size(); i++)

{

    out << operations[i].sum << " " << operations[i].

rate << std::endl;

}

При записи в данном случае будет создаваться файл в формате

120 57.7

1030 57.4

980 58.5

560 57.2

Используя оператор >>, можно считать последовательно данные в переменные sum и rate и ими инициализировать структуру.

while (in >> sum >> rate)

{

    new\_operations.push\_back(Operation(sum, rate));

}

**Строки**

Для хранения строк в C++ применяется тип **string**. Для использования этого типа его необходимо подключить в код с помощью директивы **include**:

#include <string>

#include <iostream>

int main()

{

    std::string hello = "Hello World!";

    std::cout << hello << "\n";

    return 0;

}

Тип string определен в стандартной библиотеке и при его использовании надо указывать пространство имен **std**.

Либо можно использовать выражение using, чтобы не указывать префикс std:

using std::string;

В данном случае значение переменной hello, которая представляет тип string, выводится на консоль.

При компиляции через g++ может потребоваться указать флаг *-static*. То есть если код определен в файл hello.cpp, то команда на компиляцию для g++ может выглядеть следующим образом:

g++ hello.cpp -o hello -static

Для инициализации строк можно использовать различные способы:

#include <string>

#include <iostream>

int main()

{

    std::string s1;                 // пустая строка

    std::string s2 = "hello";       // hello

    std::string s3("welcome");      // welcome

    std::string s4(5, 'h');     // hhhhh

    std::string s5 = s2;            // hello

    std::cout << s1 << "\n";

    std::cout << s2 << "\n";

    std::cout << s3 << "\n";

    std::cout << s4 << "\n";

    std::cout << s5 << "\n";

    return 0;

}

Консольный вывод данной программы:

hello

welcome

hhhhh

hello

Если при определении переменной типа string мы не присваиваем ей никакого значения, то по умолчанию данная переменная содержит пустую строку:

std::string s1;

Также можно инициализировать переменную строчным литералом, который заключается в двойные кавычки:

std::string s2 = "hello";

В качестве альтернативы можно передавать строку в скобках после определения переменной:

std::string s3("welcome");

Если необходимо, чтобы строка содержала определенное количесто определеных символов, то можно указать в скобках количество символов и сам символ:

std::string s4(5, 'h');

И также можно передать переменной копию другой строки:

std::string s5 = s2;

Конкатенация строк

Над строками можно выполнять ряд операций. В частности, можно объединять строки с помощью стандартной операции сложения:

#include <iostream>

using std::cout;

using std::endl;

using std::string;

int main()

{

    string s1 = "hello";

    string s2 = "world";

    string s3 = s1 + " " + s2; // hello world

    cout << s3 << endl;

    return 0;

}

Сравнение строк

К строкам можно применять операции сравнения. Оператор **==** возвращает true, если все символы обеих строк равны.

std::string s1 = "hello";

std::string s2 = "world";

bool result = s1 == s2;     // false

result = s1 == "Hello";     // false

result = s1 == "hello";     // true

При этом символы должны совпадать в том числе по регистру.

Операция **!=** возвращает true, если две строки не совпадают.

std::string s1 = "hello";

std::string s2 = "world";

bool result = s1 != s2;     // true

result = s1 != "Hello";     // true

result = s1 != "hello";     // false

Остальные базовые операции сравнения **<**, **<=**, **>**, **>=** сравнивают строки в зависимости от регистра и алфавитного порядка символов. Например, строка "b" условно больше строки "a", так как символ b по алфавиту идет после символа a. А строка "a" больше строки "A". Если первые символы строки равны, то сравниваются последующие символы:

std::string s1 = "Aport";

std::string s2 = "Apricot";

bool result = s1 > s2;       // false

В данном случае условие s1 > s2 ложно, то есть s2 больше чем s1, так как при равенстве первых двух символов ("Ap") третий символ второй строки ("o") стоит в алфавите до третьего символа второй строки ("p"), то есть "o" меньше чем "p".

Размер строки

С помощью метода **size()** можно узнать размер строки, то есть из скольких символов она состоит:

std::string s1 = "hello";

std::cout << s1.size() << std::endl;    // 5

Если строка пустая, то она содержит 0 символов. В этом случае мы можем применить метод **empty()** - он возвращает true, если строка пустая:

std::string s1 = "";

if(s1.empty())

    std::cout << "String is empty" << std::endl;

Чтение строки с консоли

Для считывания введенной строки с консоли можно использовать объект std::cin:

#include <iostream>

#include <string>

int main()

{

    std::string name;

    std::cout << "Input your name: ";

    std::cin >> name;

    std::cout << "Your name: " << name << std::endl;

    return 0;

}

|  |
| --- |
|  |

Консольный вывод:

Input your name: Tom

Your name: Tom

Однако если при данном способе ввода строка будет содержать подстроки, разделенные пробелом, то std::cin будет использовать только первую подстроку:

Input your name: Tom Smith

Your name: Tom

Чтобы считать всю строку, применяется метод **getline()**:

#include <iostream>

#include <string>

int main()

{

    std::string name;

    std::cout << "Input your name: ";

    getline(std::cin, name);

    std::cout << "Your name: " << name << std::endl;

    return 0;

}

Метод getline принимает два объекта - std::cin и переменную, в которую надо считать строку.

Консольный вывод:

Input your name: Tom Smith

Your name: Tom Smith

Получение и изменение символов строки

Подобно массиву мы можем обращаться с помощью индексов к отдельным символам строки, получать и изменять их:

std::string hello = "Hello";

char c = hello[1];      // e

hello[0]='M';

std::cout << hello << std::endl;    // Mello

Символьные массивы

Массив символов, последний элемент которого представляет нулевой символ '\0', может использоваться как строка:

#include <iostream>

int main()

{

    char letters[] = {'h', 'e', 'l', 'l', 'o', '\0'};

    std::cout << letters << std::endl;

    return 0;

}

Данный код выведет на консоль строку "hello". Подобное определение массива строк будет также эквивалентно следующему:

char letters[] = "hello";

Однако подобное использование массива строк унаследовано от языка Си, а при написании программ на С++ при работе со строками следует отдавать предпочтение встроенному типу string, а не массиву символов.

**Сортировки**

Было подсчитано, что до четверти времени централизованных компьютеров уделяется сортировке данных. Это потому, что намного легче найти значение в массиве, который был заранее отсортирован. В противном случае поиск немного походит на поиск иголки в стоге сена.

Есть программисты, которые всё рабочее время проводят в изучении и внедрении алгоритмов сортировки. Это потому, что подавляющее большинство программ в бизнесе включает в себя управление базами данных. Люди ищут информацию в базах данных всё время. Это означает, что поисковые алгоритмы очень востребованы.

Но есть одно "но". Поисковые алгоритмы работают намного быстрее с базами данных, которые уже отсортированы. В этом случае требуется только линейный поиск.

В то время как компьютеры находятся без пользователей в некоторые моменты времени, алгоритмы сортировки продолжают работать с базами данных. Снова приходят пользователи, осуществляющие поиск, а база данных уже отсортирована, исходя из той или иной цели поиска.

В этой статье приведены примеры реализации стандартных алгоритмов сортировки.

**Сортировка выбором (Selection sort)**

Для того, чтобы отсортировать массив в порядке возрастания, следует на каждой итерации найти элемент с наибольшим значением. С ним нужно поменять местами последний элемент. Следующий элемент с наибольшим значением становится уже на предпоследнее место. Так должно происходить, пока элементы, находящиеся на первых местах в массивe, не окажутся в надлежащем порядке.

void SortAlgo::selectionSort(int data[], int lenD)

{

int j = 0;

int tmp = 0;

for(int i=0; i<lenD; i++){

j = i;

for(int k = i; k<lenD; k++){

if(data[j]>data[k]){

j = k;

}

}

tmp = data[i];

data[i] = data[j];

data[j] = tmp;

}

}

**Пузырьковая сортировка (Bubble sort)**

При пузырьковой сортировке сравниваются соседние элементы и меняются местами, если следующий элемент меньше предыдущего. Требуется несколько проходов по данным. Во время первого прохода сраваются первые два элемента в массиве. Если они не в порядке, они меняются местами и затем сравнивается элементы в следующей паре. При том же условии они так же меняются местами. Таким образом сортировка происходит в каждом цикле пока не будет достигнут конец массива.

void SortAlgo::bubbleSort(int data[], int lenD)

{

int tmp = 0;

for(int i = 0;i<lenD;i++){

for(int j = (lenD-1);j>=(i+1);j--){

if(data[j]<data[j-1]){

tmp = data[j];

data[j]=data[j-1];

data[j-1]=tmp;

}

}

}

}

**Сортировка вставками (Insertion sort)**

При сортировке вставками массив разбивается на две области: упорядоченную и и неупорядоченную. Изначально весь массив является неупорядоченной областью. При первом проходе первый элемент из неупорядоченной области изымается и помещается в правильном положении в упорядоченной области.

На каждом проходе размер упорядоченной области возрастает на 1, а размер неупорядоченной области сокращается на 1.

Основной цикл работает в интервале от 1 до N-1. На j-й итерации элемент [i] вставлен в правильное положение в упорядоченной области. Это сделано путем сдвига всех элементов упорядоченной области, которые больше, чем [i], на одну позицию вправо. [i] вставляется в интервал между теми элементами, которые меньше [i], и теми, которые больше [i].

void SortAlgo::insertionSort(int data[], int lenD)

{

int key = 0;

int i = 0;

for(int j = 1;j<lenD;j++){

key = data[j];

i = j-1;

while(i>=0 && data[i]>key){

data[i+1] = data[i];

i = i-1;

data[i+1]=key;

}

}

}

**Сортировка слиянием (Merge sort)**

При рекурсивной сортировке слиянием массив сначала разбивается на мелкие кусочки - на первом этапе - на состоящие из одного элемента. Затем эти кусочки объединяются в более крупные кусочки - по два элемента и элементы при этом сравниваются, а в результате в новом кусочке меньший элемент занимает место слева, а больший - справа. Далее происходит слияние в ещё более крупные кусочки и так до конца алгоритма, когда все кусочки будут объединены в один, уже отсортированный массив. Если есть интерес, есть статья о рекурсивных функциях.

void SortAlgo::mergeSort(int data[], int lenD)

{

if(lenD>1){

int middle = lenD/2;

int rem = lenD-middle;

int\* L = new int[middle];

int\* R = new int[rem];

for(int i=0;i<lenD;i++){

if(i<middle){

L[i] = data[i];

}

else{

R[i-middle] = data[i];

}

}

mergeSort(L,middle);

mergeSort(R,rem);

merge(data, lenD, L, middle, R, rem);

}

}

void SortAlgo::merge(int merged[], int lenD, int L[], int lenL, int R[], int lenR){

int i = 0;

int j = 0;

while(i<lenL||j<lenR){

if (i<lenL & j<lenR){

if(L[i]<=R[j]){

merged[i+j] = L[i];

i++;

}

else{

merged[i+j] = R[j];

j++;

}

}

else if(i<lenL){

merged[i+j] = L[i];

i++;

}

else if(j<lenR){

merged[i+j] = R[j];

j++;

}

}

}

**Быстрая сортировка (Quick sort)**

Быстрая сортировка использует алгоритм "разделяй и властвуй". Она начинается с разбиения исходного массива на две области. Эти части находятся слева и справа от отмеченного элемента, называемого опорным. В конце процесса одна часть будет содержать элементы меньшие, чем опорный, а другая часть будет содержать элементы больше опорного.

void SortAlgo::quickSort(int\* data, int const len)

{

int const lenD = len;

int pivot = 0;

int ind = lenD/2;

int i,j = 0,k = 0;

if(lenD>1){

int\* L = new int[lenD];

int\* R = new int[lenD];

pivot = data[ind];

for(i=0;i<lenD;i++){

if(i!=ind){

if(data[i]<pivot){

L[j] = data[i];

j++;

}

else{

R[k] = data[i];

k++;

}

}

}

quickSort(L,j);

quickSort(R,k);

for(int cnt=0;cnt<lenD;cnt++){

if(cnt<j){

data[cnt] = L[cnt];;

}

else if(cnt==j){

data[cnt] = pivot;

}

else{

data[cnt] = R[cnt-(j+1)];

}

}

}

}

**Структуры**

Структура в языке C++ представляет собой производный тип данных, который представляет какую-то определенную сущность, также как и класс. Нередко структуры применителько к С++ также называют классами. И в реальности различия между ними не такие большие.

Для определения структуры применяется ключевое слово **struct**, а сам формат определения выглядит следующим образом:

struct имя\_структуры

{

    компоненты\_структуры

};

*Имя\_структуры* представляет произвольный идентификатор, к которому применяются те же правила, что и при наименовании переменных.

После имени структуры в фигурных скобках помещаются *Компоненты\_структуры*, которые представляют набор описаний объектов и функций, которые составляют структуру.

Например, определим простейшую структуру:

#include <iostream>

#include <string>

struct person

{

    int age;

    std::string name;

};

int main()

{

    person tom;

    tom.name = "Tom";

    tom.age = 34;

    std::cout << "Name: " << tom.name << "\tAge: " <<

tom.age << std::endl;

    return 0;

}

|  |
| --- |
|  |

Здесь определена структура person, которая имеет два элемента: age (представляет тип int) и name (представляет тип string).

После определения структуры мы можем ее использовать. Для начала мы можем определить объект структуры - по сути обычную переменную, которая будет представлять выше созданный тип. Также после создания переменной структуры можно обращаться к ее элементам - получать их значения или, наоборот, присваивать им новые значения. Для обращения к элементам структуры используется операция "точка":

имя\_переменной\_структуры.имя\_элемента

По сути структура похожа на класс, то есть с помощью структур также можно определять сущности для использования в программе. В то же время все члены структуры, для которых не используется спецификатор доступа (public, private), по умолчанию являются открытыми (public). Тогда как в классе все его члены, для которых не указан спецификатор доступа, являются закрытыми (private).

Кроме того мы можем инициализировать структуру, присвоив ее переменным значения с помощью синтаксиса инициализации:

person tom = { 34, "Tom" };

Инициализация структур аналогична инициализации массивов: в фигурных скобках передаются значения для элементов структуры по порядку. Так как в структуре person первым определено свойство, которое представляет тип int - число, то в фигурных скобках вначале идет число. И так далее для всех элементов структуры по порядку.

При этом любой класс мы можем представить в виде структуры и наоборот. Возьмем, к примеру, следующий класс:

class Person

{

public:

    Person(std::string n, int a)

    {

        name = n; age = a;

    }

    void move()

    {

        std::cout << name << " is moving" << std::endl;

    }

    void setAge(int a)

    {

        if (a > 0 && a < 100) age = a;

    }

    std::string getName()

    {

        return name;

    }

    int getAge()

    {

        return age;

    }

private:

    std::string name;

    int age;

};

|  |
| --- |
|  |

Данный класс определяет сущность человека и содержит ряд приватных и публичных переменных и функции. Вместо класса для определения той же сущности мы могли бы использовать структуру:

#include <iostream>

#include <string>

struct user

{

public:

    user(std::string n, int a)

    {

        name = n; age = a;

    }

    void move()

    {

        std::cout << name << " is moving" << std::endl;

    }

    void setAge(int a)

    {

        if (a > 0 && a < 100) age = a;

    }

    std::string getName()

    {

        return name;

    }

    int getAge()

    {

        return age;

    }

private:

    std::string name;

    int age;

};

int main()

{

    user tom("Tom", 22);

    std::cout << "Name: " << tom.getName() << "\tAge: "

<< tom.getAge() << std::endl;

    tom.setAge(31);

    std::cout << "Name: " << tom.getName() << "\tAge: "

<< tom.getAge() << std::endl;

    return 0;

}

И в плане конечного результата программы мы не увидели бы никакой разницы.

Когда использовать структуры? Как правило, структуры используются для описания таких данных, которые имеют только набор публичных атрибутов - открытых переменных. Например, как та же структура person, которая была определена в начале статьи. Иногда подобные сущности еще называют **аггрегатными классами** (aggregate classes).