**Билет 1.**  
*Система контроля версий:*  
Наверное стоит начать с определения системы контроля версий (СКВ) — это система, регистрирующая изменения в одном или нескольких файлах с тем, чтобы в дальнейшем была возможность вернуться к определённым старым версиям этих файлов.  
В последнее время файлы являются конечным результатом для многих профессий (для примера, писательскую деятельность, научные работы и, конечно, разработку программного обеспечения). Тратится много времени и сил на разработку и поддержку этих файлов и никто не хочет, чтобы пришлось тратить еще больше времени и сил на восстановление данных потерянных в результате каких-либо изменений.  
Представим, что программист разрабатывает проект состоящий из одного небольшого файла (кстати, пример вполне реальный, не синтетический, встречался в реальной жизни). После выпуска первой версии проекта перед ним встает непростой выбор: необходимо исправлять проблемы о которых сообщают пользователи первой версии и, в тоже время, разрабатывать что-то новое для второй. Даже если надо просто исправлять возникающие проблемы, то велика вероятность, что после какого-либо изменения проект перестает работать, и надо определить, что было изменено, чтобы было проще локализовать проблему. Также желательно вести какой-то журнал внесенных изменений и исправлений, чтобы не делать несколько раз одну и ту же работу.  
В простейшем случае вышеприведенную проблему можно решить хранением нескольких копий файлов, например, один для исправления ошибок в первой версии проекта и второй для новых изменений. Так как изменения обычно не очень большие по сравнению с размером файла, то можно хранить только измененные строки используя утилиту diff и позже объединять их с помощью утилиты patch. Но что если проект состоит из нескольких тысяч файлов и над ним работает сотня человек? Если в этом случае использовать метод с хранением отдельных копий файлов (или даже только изменений) то проект застопорится очень быстро. В последующих лекциях, для примеров я буду использовать исходные коды программ, но на самом деле под версионный контроль можно поместить файлы практически любого типа.  
Если вы графический или веб-дизайнер и хотели бы хранить каждую версию изображения или макета — а этого вам наверняка хочется — то пользоваться системой контроля версий будет очень мудрым решением. СКВ даёт возможность возвращать отдельные файлы к прежнему виду, возвращать к прежнему состоянию весь проект, просматривать происходящие со временем изменения, определять, кто последним вносил изменения во внезапно переставший работать модуль, кто и когда внёс в код какую-то ошибку, и многое другое. Вообще, если, пользуясь СКВ, вы всё испортите или потеряете файлы, всё можно будет легко восстановить. Вдобавок, накладные расходы за всё, что вы получаете, будут очень маленькими.

**Билет 2.***Структура программы:*  
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:

|  |  |
| --- | --- |
| 1 | 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:

|  |  |
| --- | --- |
| 1  2  3  4 | {      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 **main** фиксировано и для всех программ на Си всегда одинаково.  
Функция также является блоком кода, поэтому ее тело обрамляется фигурными скобками, между которыми определяется набор инструкций.  
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 подключить 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.  
*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:

|  |  |
| --- | --- |
| 1 | тип\_переменной имя\_переменной; |

Простейшее определение переменной:

|  |  |
| --- | --- |
| 1 | **int** age; |

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

|  |  |
| --- | --- |
| 1  2 | **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**.   
Также нельзя объявить больше одной переменной с одним и тем же именем, например:

|  |  |
| --- | --- |
| 1  2 | **int** age;  **int** age; |

**Билет 3.** *Типы данных:*  
Подобное определение вызовет ошибку на этапе компиляции.  
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) и **false** (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\_t** и **char32\_t**.  
Определим несколько переменных:

|  |  |
| --- | --- |
| 1  2 | **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.

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | **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**:

|  |  |
| --- | --- |
| 1  2  3 | **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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | #include <iostream>    **int** main()  {  **long** **double** number = 2;      std::cout << "sizeof(number) =" << **sizeof**(number);  **return** 0;  } |

### 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:

|  |  |
| --- | --- |
| 1 | auto number = 5; |

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

|  |  |
| --- | --- |
| 1 | auto number; |

**Билет 4.**

Метод, наверное, подразумевается как функция передаваемые в нее аргументы это то, что указывается в скобках int test(int a, string b, float c) возвращаемые аргументы это то, что стоит перед названием функции(метода)

**Билет 5.***Инкапсуляция:*  
По умолчанию, в классе (class) данные и методы приватные (private); они могут быть прочитаны и изменены только классом, к которому принадлежат. Уровень доступа может быть изменен при помощи соответствующих ключевых слов, которые предоставляет Си++.

В Си++ доступно несколько спецификаторов, и они изменяют доступ к данным следующим образом:

публичные (public) данные — доступны всем;  
защищенные (protected) — доступны только классу и дочерним классам;  
приватные (private) —доступны только классу которому они принадлежат.  
Для краткости, только два уровня (приватный и публичный) будут освещены в примерах.  
Преимущество №1:**Инкапсулированные классы проще в использовании и уменьшают сложность ваших программ.**  
С полностью инкапсулированным классом вам нужно знать только то, какие методы являются доступными для использования, какие аргументы они принимают и какие значения возвращают. Не нужно знать, как класс реализован изнутри.   
Преимущество №2:**Инкапсулированные классы помогают защитить ваши данные и предотвращают их неправильное использование.**  
[**Глобальные переменные**](https://ravesli.com/urok-49-globalnye-peremennye/) опасны, так как нет строгого контроля над тем, кто имеет к ним доступ и как их используют. Классы с открытыми членами имеют ту же проблему, только в меньших масштабах.   
Преимущество №3:**Инкапсулированные классы легче изменить.**  
Преимущество №4**: С инкапсулированными классами легче проводить отладку.**И, наконец, инкапсуляция помогает проводить [отладку программ](https://ravesli.com/urok-26-otladka-programm-stepping-i-breakpoints/), когда что-то идет не по плану. Часто причиной неправильной работы программы является некорректное значение одной из переменных. Если каждый объект имеет прямой доступ к переменной, то отследить часть кода, которая изменила переменную, может быть довольно-таки трудно. Однако, если для изменения значения нужно вызывать один и тот же метод, вы можете просто использовать точку останова для этого метода и посмотреть, как каждый вызывающий объект изменяет значение, пока не увидите что-то странное.

**Билет 6.***Цикл for:*Цикл for имеет следующее формальное определение:

|  |  |
| --- | --- |
| 1  2  3  4 | **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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | #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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | **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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | #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;  } |

**Билет 7.***Статическая типизация и преобразования типов:*С++ является статически типизированным языком программирования. То есть если мы определили для переменной какой-то тип данных, то в последующем мы этот тип изменить не сможем. Соответственно переменная может получить значения только того типа, который она представляет. Однако нередко возникает необходимость присвоить переменной значения каких-то других типов. И в этом случае применяются преобразования типов.  
Ряд преобразований компилятор может производить неявно, то есть автоматически. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | #include <iostream>    **int** main()  {  **int** code = 'g';  **char** letter = 103;      std::cout << letter << " in ASCII is " << code << "\n";  **return** 0;  } |

В данном случае числовой переменной типа int присваивается символ 'g'. Этот символ будет автоматически преобразовываться в число. По факту переменная получит числовой код этого символа в таблице ASCII.  
Переменной letter, наоборот, присваивается число, хотя эта переменная представляет тип char. Это число будет преобразовываться в символ, и в итоге переменная letter будет хранить символ, чей код в таблице ASCII равен 103, то есть символ 'g'.  
Результатом этой программы будет следующий консольный вывод:

g in ASCII is 103

Как выполняются преобразования:  
Переменной типа **bool** присваивается значение другого типа. В этом случае переменная получает **false**, если значение равно 0. Во всех остальных случаях переменная получает **true**.

|  |  |
| --- | --- |
| 1  2  3  4 | **bool** a = 1;     // true  **bool** b = 0;     // false  **bool** c = 'g'; // true  **bool** d = 3.4;   // true |

Числовой или символьной переменной присваивается значение типа **bool**. В этом случае переменная получает 1, если значений равно **true**, либо получает 0, если присваиваемое значение равно **false**.

|  |  |
| --- | --- |
| 1  2 | **int** c = **true**;       // 1  **double** d = **false**;   // 0 |

Целочисленной переменной присваивается дробное число. В этом случае дробная часть после запятой отбрасывается.

|  |  |
| --- | --- |
| 1  2 | **int** a = 3.4;        // 3  **int** b = 3.6;        // 3 |

Переменной, которая представляет тип с плавающей точкой, присваивается целое число. В этом случае если целое число содержит больше битов, чем может вместить тип переменной, то часть информации усекается.

|  |  |
| --- | --- |
| 1  2 | **float** a = 35005;                // 35005  **double** b = 3500500000033;       // 3.5005e+012 |

Переменной беззнакового типа (unsigned) присваивается значение не из его диапазона. В этом случае результатом будет остаток от деления по модулю. Например, тип **unsigned char** может хранить значения от 0 до 255. Если присвоить ему значение вне этого диапазона, то компилятор присвоит ему остаток от деления по модулю 256 (так как тип unsigned char может хранить 256 значений). Так, при присвоении значения -1 переменная типа unsigned char получит 256 - |-1/256| = 255

|  |  |
| --- | --- |
| 1  2  3 | unsigned **char** a = -5;           // 251  unsigned **short** b = -3500;       // 62036 ()  unsigned **int** c = -50000000;     // 4244967296 |

Переменной знакового типа (signed) присваивается значение не из его диапазона. В этом случае результат не определен. Программа может работать нормально, выдавая адекватный результат, а может работать некорректно.

**Билет 8.**  
*Наследование:*Наследование (inheritance) представляет один из ключевых аспектов объектно-ориентированного программирования, который позволяет наследовать функциональность одного класса или базового класса (base class) в другом - производном классе (derived class).  
Зачем нужно наследование? Рассмотрим небольшую ситуацию, допустим, у нас есть классы, которые представляют человека и работника предприятия:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | class Person  {  public:      std::string name;       //  имя      int age;                // возраст      void display()      {          std::cout << "Name: " << name << "\tAge: " << age << std::endl;      }  };  class Employee  {  public:      std::string name;       // имя      int age;                // возраст      std::string company;    // компания      void display()      {          std::cout << "Name: " << name << "\tAge: " << age << std::endl;      }  }; |

В данном случае класс Employee фактически содержит функционал класса Person: свойства name и age и функцию display. И было бы не совсем правильно повторять функциональность одного класса в другом классе, тем более что, по сути, сотрудник предприятия в любом случае является человеком. Поэтому в этом случае лучше использовать механизм наследования.   
Для установки отношения наследования после название класса ставится двоеточие, затем идет название класса, от которого мы хотим унаследовать функциональность. В этом отношении класс Person еще будет называться базовым классом, а Employee - производным классом.  
Перед названием базового класса также можно указать спецификатор доступа, как в данном случае используется спецификатор public, который позволяет использовать в производном классе все открытые члены базового класса. Если мы не используем модификатор доступа, то класс Employee ничего не будет знать о переменных name и age и функции display.  
После установки наследования мы можем убрать из класса Employee те переменные, которые уже определены в классе Person.   
Класс, который наследует данные, называется подклассом (subclass), производным классом (derived class) или дочерним классом (child). Класс, от которого наследуются данные или методы, называется суперклассом (super class), базовым классом (base class) или родительским классом (parent). Термины “родительский” и “дочерний” чрезвычайно полезны для понимания наследования. Как ребенок получает характеристики своих родителей, производный класс получает методы и переменные базового класса.

Наследование полезно, поскольку оно позволяет структурировать и повторно использовать код, что, в свою очередь, может значительно ускорить процесс разработки. Несмотря на это, наследование следует использовать с осторожностью, поскольку большинство изменений в суперклассе затронут все подклассы, что может привести к непредвиденным последствиям.

В этом примере, метод turn\_on() и переменная serial\_number не были объявлены или определены в подклассе Computer. Однако их можно использовать, поскольку они унаследованы от базового класса.

*Важное примечание*: приватные переменные и методы не могут быть унаследованы.  
Запрет наследования  
Иногда наследование от класса может быть нежелательно. И с помощью спецификатора final мы можем запретить наследование.  
После этого мы не сможем унаследовать другие классы от класса User. И, например, если мы попробуем написать, как в случае ниже, то мы столкнемся с ошибкой.

**Билет 9.**

*Операторы ветвления:*Условный оператор if... else  
Условный оператор реализует структуру выбора. Он имеет такой вид:  
if (условие) оператор1[ else оператор 2]  
Если условие оценивается как истинное (ненулевое), выполняется onepaтop1, если как ложное (нулевое), выполняется onepaтop2. Простейший пример:  
if (а > b)  
max\_ab = a;  
else  
max\_ab = b;  
Как было сказано чуть выше, вместо одиночного оператора всегда можно подставить блок из нескольких операторов, заключенный в фигурные скобки. Другими словами, возможна следующая синтаксическая форма:  
if (условие)  
{операторы\_блока\_if}  
else  
{операторы\_блока\_еlsе}  
В случае, когда при ложности условия не нужно выполнять никаких действий, а требуется только пропустить операторы блока if, ключевое слово else и соответствующий ему оператор (блок) могут отсутствовать.

!    – логическое НЕ (инверсия);  
<    – меньше;  
<=    – меньше или равно;  
>    – больше;  
>=    – больше или равно;  
= =    – равно;  
!=    – логическое И;  
&&    – логическое И;  
||    – логическое ИЛИ.  
Условия операторов if содержат проверку кода нажатой клавиши на равенство одному из допустимых символов. Если код клавиши не соответствует никакой команде, выводится сообщение об ошибке.

**Билет 10.***Перезагрузка функций:*C++ позволяет определять несколько функций с одинаковым именем в одной области. Эти функции называются *перегруженными* функциями. Перегруженные функции позволяют указать другую семантику для функции в зависимости от типов и числа аргументов.  
Например, print функция, которая принимает аргумент, std::string может выполнять задачи, отличные от тех, которые принимают аргумент типа **double** . Перегрузка позволяет избежать использования таких имен, как print\_string или print\_double . Во время компиляции компилятор выбирает, какую перегрузку следует использовать в зависимости от типа аргументов, передаваемых вызывающим объектом. При вызове print(42.0)void print(double d) функции будет вызвана функция. При вызове метода print("hello world")void print(std::string) будет вызвана перегрузка.  
Можно перегружать как функции-члены, так и функции, не являющиеся членами. В следующей таблице указаны компоненты объявления функций, используемые языком C++ для различения групп функций с одинаковым именем в одной области.

**Билет 11.**  
*Ссылки:*  
Ссылка в С++ — это альтернативное имя объекта.  
Ссылку можно понимать как безопасный вариант указателя. При этом ссылки имеют особенности, отличающие их от указателей:  
При объявлении ссылка обязательно на уже существующий объект данного типа. Ссылка не может ссылаться "ни на что".  
Ссылка от её объявления до её исчезновения указывает на один и тот же адрес.  
При обращении к ссылке разыменование происходит автоматически.  
Адрес ссылки — это адрес исходного объекта, на который она указывает.  
Объявление ссылок очень похоже на объявление указателей, только вместо звёздочки \* пишется амперсанд &.  
При объявлении ссылка обязана быть инициализирована.

**Билет 12.**  
*Массив:*  
Массив представляет набор однотипных данных. Формальное определение массива выглядит следующим образом:

|  |  |
| --- | --- |
| 1 | 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:

|  |  |
| --- | --- |
| 1 | **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:

|  |  |
| --- | --- |
| 1 | **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:

|  |  |
| --- | --- |
| 1 | **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:

|  |  |
| --- | --- |
| 1 | **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:

|  |  |
| --- | --- |
| 1  2 | **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:

|  |  |
| --- | --- |
| 1  2  3 | **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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | #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:

|  |  |
| --- | --- |
| 1  2 | **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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | #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 работы с коллекциями, в том числе с массивами. Эта форма имеет следующее формальное определение:

|  |  |
| --- | --- |
| 1  2  3  4 | **for**(тип переменная : коллекция)  {      инструкции;  } |

Используем эту форму для перебора массива:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | #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:

|  |  |
| --- | --- |
| 1  2 | **for**(auto number : numbers)      std::cout << number << std::endl; |

### Многомерные 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:

|  |  |
| --- | --- |
| 1 | **int** numbers[3][2]; |

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

|  |  |
| --- | --- |
| 1 | **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:

|  |  |
| --- | --- |
| 1 | 2 |
| 4 | 5 |
| 7 | 8 |

Также при инициализации можно опускать фигурные скобки:

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

Возможна также инициализация не всех элементов, а только некоторых:

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

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

|  |  |
| --- | --- |
| 1  2  3  4 | **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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | #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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | #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.

### **Билет 13.** *Стек и кучи:* Стек Стек — это область оперативной памяти, которая создаётся для каждого потока. Он работает в порядке LIFO (Last In, First Out),  то есть последний добавленный в стек кусок памяти будет первым в очереди на вывод из стека. Каждый раз, когда функция объявляет новую переменную, она добавляется в стек, а когда эта переменная пропадает из области видимости (например, когда функция заканчивается), она автоматически удаляется из стека. Когда стековая переменная освобождается, эта область памяти становится доступной для других стековых переменных. Из-за такой природы стека управление памятью оказывается весьма логичным и простым для выполнения на ЦП; это приводит к высокой скорости, в особенности потому, что время цикла обновления байта стека очень мало, т.е. этот байт скорее всего привязан к кэшу процессора. Тем не менее, у такой строгой формы управления есть и недостатки. Размер стека — это фиксированная величина, и превышение лимита выделенной на стеке памяти приведёт к переполнению стека. Размер задаётся при создании потока, и у каждой переменной есть максимальный размер, зависящий от типа данных. Это позволяет ограничивать размер некоторых переменных (например, целочисленных), и вынуждает заранее объявлять размер более сложных типов данных (например, массивов), поскольку стек не позволит им изменить его. Кроме того, переменные, расположенные на стеке, всегда являются локальными. В итоге стек позволяет управлять памятью наиболее эффективным образом — но если вам нужно использовать динамические структуры данных или глобальные переменные, то стоит обратить внимание на кучу. Куча Куча — это хранилище памяти, также расположенное в ОЗУ, которое допускает динамическое выделение памяти и не работает по принципу стека: это просто склад для ваших переменных. Когда вы выделяете в куче участок памяти для хранения переменной, к ней можно обратиться не только в потоке, но и во всем приложении. Именно так определяются глобальные переменные. По завершении приложения все выделенные участки памяти освобождаются. Размер кучи задаётся при запуске приложения, но, в отличие от стека, он ограничен лишь физически, и это позволяет создавать динамические переменные. Вы взаимодействуете с кучей посредством ссылок, обычно называемых указателями — это переменные, чьи значения являются адресами других переменных. Создавая указатель, вы указываете на местоположение памяти в куче, что задаёт начальное значение переменной и говорит программе, где получить доступ к этому значению. Из-за динамической природы кучи ЦП не принимает участия в контроле над ней; в языках без сборщика мусора (C, C++) разработчику нужно вручную освобождать участки памяти, которые больше не нужны. Если этого не делать, могут возникнуть утечки и фрагментация памяти, что существенно замедлит работу кучи. В сравнении со стеком, куча работает медленнее, поскольку переменные разбросаны по памяти, а не сидят на верхушке стека. Некорректное управление памятью в куче приводит к замедлению её работы; тем не менее, это не уменьшает её важности — если вам нужно работать с динамическими или глобальными переменными, пользуйтесь кучей.

**Билет 14.**  
*Конструктор:*  
В классе Person определен конструктор

|  |  |
| --- | --- |
| 1  2  3  4 | Person(string n, int a)  {      name = n; age = a;  } |

По сути, конструктор представляет функцию, которая может принимать параметры и которая должна называться по имени класса. В данном случае конструктор принимает два параметра и передает их значения полям name и age.  
Если в классе определены конструкторы, то при создании объекта этого класса необходимо вызвать один из его конструкторов.  
Вызов конструктора получает значения для параметров и возвращает объект класса:

|  |  |
| --- | --- |
| 1 | Person person = Person("Tom", 22); |

После этого вызова у объекта person для поля name будет определено значение "Tom", а для поля age - значение 22. Вполедствии мы также сможем обращаться к этим полям и переустанавливать их значения.  
Тажке можно использовать сокращенную форму инициализации:

|  |  |
| --- | --- |
| 1 | Person person("Tom", 22); |

По сути, она будет эквивалетна предыдущей.

*Конструктор* — функция, предназначенная для инициализации объектов класса. Рассмотрим класс date:

class date  
{  
  int day, month, year;  
public:  
  set(int, int, int);  
};

## Билет 15. *Деструктор:* ****Деструктор**** — это специальный тип [****метода****](https://ravesli.com/urok-113-klassy-obekty-i-metody-klassov/) класса, который выполняется при удалении объекта класса. В то время как [****конструкторы****](https://ravesli.com/urok-116-konstruktory/) предназначены для инициализации класса, деструкторы предназначены для очистки памяти после него. Когда объект автоматически выходит из области видимости или [****динамически выделенный****](https://ravesli.com/urok-85-dinamicheskoe-vydelenie-pamyati-operatory-new-i-delete/) объект явно удаляется с помощью ключевого слова delete, вызывается деструктор класса (если он существует) для выполнения необходимой очистки до того, как объект будет удален из памяти. Для простых классов (тех, которые только инициализируют значения обычных переменных-членов) деструктор не нужен, так как C++ автоматически выполнит очистку самостоятельно. Однако, если объект класса содержит любые ресурсы (например, динамически выделенную память или файл/базу данных), или, если вам необходимо выполнить какие-либо действия до того, как объект будет уничтожен, деструктор является идеальным решением, поскольку он производит последние действия с объектом перед его окончательным уничтожением. Имена деструкторов Так же, как и конструкторы, деструкторы имеют свои правила, которые касаются их имен: деструктор должен иметь то же имя, что и класс, со знаком тильда (~) в самом начале; деструктор не может принимать аргументы; деструктор не имеет типа возврата. Из второго правила вытекает еще одно правило: для каждого класса может существовать только один деструктор, так как нет возможности [****перегрузить****](https://ravesli.com/urok-102-peregruzka-funktsij/) деструкторы, как функции, и отличаться друг от друга аргументами они не могут.

Билет 16. ***Функции:***Функции — это блоки кода, выполняющие определенные операции. Если требуется, функция может определять входные параметры, позволяющие вызывающим объектам передавать ей аргументы. При необходимости функция также может возвращать значение как выходное. Функции полезны для инкапсуляции основных операций в едином блоке, который может многократно использоваться. В идеальном случае имя этого блока должно четко описывать назначение функции. Следующая функция принимает два целых числа от вызывающего объекта и возвращает их сумму; a и b — это Параметры типа.  
C++Копировать  
int sum(int a, int b)  
{  
 return a + b;  
}  
Функцию можно вызывать или вызыватьиз любого числа мест в программе. Значения, передаваемые в функцию, являются аргументами, типы которых должны быть совместимы с типами параметров в определении функции.  
int main()  
{  
 int i = sum(10, 32);  
 int j = sum(i, 66);  
 cout << "The value of j is" << j << endl;   
}  
Длина функции практически не ограничена, однако для максимальной эффективности кода целесообразно использовать функции, каждая из которых выполняет одиночную, четко определенную задачу. Сложные алгоритмы лучше разбивать на более короткие и простые для понимания функции, если это возможно.  
Функции, определенные в области видимости класса, называются функциями-членами. В C++, в отличие от других языков, функции можно также определять в области видимости пространства имен (включая неявное глобальное пространство имен). Такие функции называются бесплатными функциями или функциями, не являющимися членами. Они широко используются в стандартной библиотеке.  
Функции могут быть перегружены, а это значит, что разные версии функции могут совместно использовать одно и то же имя, если они отличаются числом и (или) типом формальных параметров. Дополнительные сведения см. в разделе [перегрузка функций](https://docs.microsoft.com/ru-ru/cpp/cpp/function-overloading?view=msvc-170).

**Билет 17.**  
*Возврат массива:*У функции возвращаемое значение должно указываться как масив допустим у тебя динамический масив он обозначается int\* array функция будет иметь вид int\* function(..){ int\* array = new int[5]; - здесь создали динамич. массив return array; - здесь его вернули } то есть результат функции – массив.

**Билет 18.**  
*Массив:  
Билет номер 12.*

**Билет 19:***Указатели:*  
Указатель – переменная, значением которой является адрес ячейки памяти. То есть указатель ссылается на блок данных из области памяти, причём на самое его начало. Указатель может ссылаться на переменную или функцию. Для этого нужно знать адрес переменной или функции. Так вот, чтобы узнать адрес конкретной переменной в С++ существует унарная операция взятия адреса &. Такая операция извлекает адрес объявленных переменных, для того, чтобы его присвоить указателю.  
Указатели используются для передачи по ссылке данных, что намного ускоряет процесс обработки этих данных (в том случае, если объём данных большой), так как их не надо копировать, как при передаче по значению, то есть, используя имя переменной. В основном указатели используются для организации динамического распределения памяти, например при объявлении массива, не надо будет его ограничивать в размере. Ведь программист заранее не может знать, какого размера нужен массив тому или иному пользователю, в таком случае используется динамическое выделение памяти под массив. Любой указатель необходимо объявить перед использованием, как и любую переменную.

|  |  |
| --- | --- |
| 1  2 | //объявление указателя  /\*тип данных\*/  \* /\*имя указателя\*/; |

Принцип объявления указателей такой же, как и принцип объявления переменных. Отличие заключается только в том, что перед именем ставится символ звёздочки \*. Визуально указатели отличаются от переменных только одним символом. При объявлении указателей компилятор выделяет несколько байт памяти, в зависимости от типа данных, отводимых для хранения некоторой информации в памяти. Чтобы получить значение, записанное в некоторой области, на которое ссылается указатель нужно воспользоваться операцией разыменования указателя \*. Необходимо поставить звёздочку перед именем и получим доступ к значению указателя. Разработаем программу, которая будет использовать указатели.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | // pointer1.cpp: определяет точку входа для консольного приложения.    #include "stdafx.h"  #include <iostream>  **using** **namespace** std;    **int** main(**int** argc, **char**\* argv[])  {  **int** var = 123; // инициализация переменной var числом 123  **int** \*ptrvar = &var; // указатель на переменную var (присвоили адрес переменной указателю)      cout << "&var    = " << &var << endl;// адрес переменной var содержащийся в памяти, извлечённый операцией взятия адреса      cout << "ptrvar  = " << ptrvar << endl;// адрес переменной var, является значением указателя ptrvar      cout << "var     = " << var << endl; // значение в переменной var      cout << "\*ptrvar = " << \*ptrvar << endl; // вывод значения содержащегося в переменной var через указатель, операцией разименования указателя  **system**("pause");  **return** 0;  } |

## Билет 20: *Операции с указателями:* Указатели поддерживают ряд операций: присваивание, получение адреса указателя, получение значения по указателю, некоторые арифметические операции и операции сравнения. Присваивание Указателю можно присвоить либо адрес объекта того же типа, либо значение другого указателя. Присвоение указателю адреса уже рассматривалось в прошлой теме. Для получения адреса объекта используется операция &:

|  |  |
| --- | --- |
| 1  2 | **int** a = 10;  **int** \*pa = &a;   // указатель pa хранит адрес переменной a |

При этом указатель и переменная должны иметь один и тот же тип, в данном случае это тип int.  
Присвоение указателю другого указателя:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | #include <iostream>  **using** std::cout;  **using** std::endl;    **int** main()  {  **int** a = 10;  **int** b = 2;    **int** \*pa = &a;  **int** \*pb = &b;        cout << "Variable a: address=" << pa << "\t value=" << \*pa << endl;      cout << "Variable b: address=" << pb << "\t value=" << \*pb << endl;        pa = pb;    // теперь указатель pa хранит адрес переменной b      cout << "Variable b: address=" << pa << "\t value=" << \*pa << endl;    **return** 0;  } |

Когда указателю присваивается другой указатель, то фактически первый указатель начинает также указывать на тот же адрес, на который указывает второй указатель.  
Нулевые указатели  
Нулевой указатель (null pointer) - это указатель, который не указывает ни на какой объект. Если мы не хотим, чтобы указатель указывал на какой-то конкретный адрес, то можно присвоить ему условное нулевое значение. Для создания нулевого указателя можно применять различные способы:

|  |  |
| --- | --- |
| 1  2  3 | **int** \*p1 = nullptr;  **int** \*p2 = NULL;  **int** \*p3 = 0; |

### Ссылки на указатели Так как ссылка не является объектом, то нельзя определить указатель на ссылку, однако можно определить ссылку на указатель. Через подобную ссылку можно изменять значение, на которое указывает указатель или изменять адрес самого указателя:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | #include <iostream>    **int** main()  {  **int** a = 10;  **int** b = 6;    **int** \*p = 0;     // указатель  **int** \*&pRef = p;     // ссылка на указатель      pRef = &a;          // через ссылку указателю p присваивается адрес переменной a      std::cout << "p value=" << \*p << std::endl;   // 10      \*pRef = 70;         // изменяем значение по адресу, на который указывает указатель      std::cout << "a value=" << a << std::endl;    // 70        pRef = &b;          // изменяем адрес, на который указывает указатель      std::cout << "p value=" << \*p << std::endl;   // 6    **return** 0;  } |

### Разыменование указателя Операция разыменования указателя представляет выражение в виде \*имя\_указателя. Эта операция позволяет получить объект по адресу, который хранится в указателе.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | #include <iostream>  **using** std::cout;  **using** std::endl;    **int** main()  {  **int** a = 10;    **int** \*pa = &a;  **int** \*pb = pa;        \*pa = 25;        cout << "Value on pointer pa: " << \*pa << endl;  // 25      cout << "Value on pointer pb: " << \*pb << endl;  // 25      cout << "Value of variable a: " << a << endl;    // 25    **return** 0;  } |

Через выражение \*pa мы можем получить значение по адресу, который хранится в указателе pa, а через выражение типа \*pa = значение вложить по этому адресу новое значение.  
И так как в данном случае указатель pa указывает на переменную a, то при изменении значения по адресу, на который указывает указатель, также изменится и значение переменной a.

**Билет 21.**  
*Динамический объект:*В C++ можно использовать различные типы объектов, которые различаются по использованию памяти. Так, глобальные объекты создаются при запуске программы и освобождаются при ее завершении. Локальные автоматические объекты создаются в блоке кода и удаляются, когда этот блок кода завершает работу. Локальные статические объекты создаются перед их первым использованием и освобождаются при завершении программы.  
Глобальные, а также статические локальные объекты помещаются в статической памяти, а локальные автоматические объекты размещаются в стеке. Объекты в статической памяти и стеке создаются и удаляются компилятором. Статическая память очищается при завершении программы, а объекты из стека существуют, пока выполняется блок, в котором они определены.  
В дополнение к этим типам в C++ можно создавать **динамические объекты**. Продолжительность их жизни не зависит от того, где они созданы. Динамические объекты существуют, пока не будут удалены явным образом. Динамические объекты размещаются в **динамической памяти** (free store).  
Для управления динамическими объектами применяются операторы **new** и **delete**.  
Оператор **new** выделяет место в динамической памяти для объекта и возвращает указатель на этот объект.  
Оператор **delete** получает указатель на динамический объект и удаляет его из памяти.  
Выделение памяти  
Создание динамического объекта:

|  |  |
| --- | --- |
| 1 | **int** \*ptr = **new** **int**; |

Оператор new создает новый объект типа int в динамической памяти и возвращает указатель на него. Значение такого объекта неопределено.  
Также можно инициализировать объект при создании:

|  |  |
| --- | --- |
| 1  2  3  4  5 | **int** \*p1 = **new** **int**();    // значение по умолчанию - 0  std::cout << "p1: " << \*p1 << std::endl;  // 0    **int** \*p2 = **new** **int**(12);  std::cout << "p2: " << \*p2 << std::endl;   // 12 |

### Освобождение памяти

### Динамические объекты будут существовать пока не будут явным образом удалены. И после завершения использования динамических объектов следует освободить их память с помощью оператора **delete**:

|  |  |
| --- | --- |
| 1  2  3 | **int** \*p1 = **new** **int**(12);  std::cout << "p1: " << \*p1 << std::endl;  // 12  **delete** p1; |

Особенно это надо учитывать, если динамический объект создается в одной части кода, а используется в другой. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | #include <iostream>    **int**\* createPtr(**int** value)  {  **int** \*ptr = **new** **int**(value);  **return** ptr;  }  **void** usePtr()  {  **int** \*p1 = createPtr(10);      std::cout << \*p1 << std::endl;  // 10  **delete** p1;  // объект надо освободить  }  **int** main()  {      usePtr();    **return** 0;  } |

В функции usePtr получаем из функции createPtr указатель на динамический объект. Однако после выполнения функции usePtr этот объект автоматически не удаляется из памяти (как это происходит в случае с локальными автоматическими объектами). Поэтому его надо явным образом удалить, использовав оператор delete.  
Использование объекта по указателю после его удаления или повторное применение оператора delete к указателю могут привести к непредсказуемым результатам:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | **int** \*p1 = **new** **int**(12);  std::cout << \*p1 << std::endl;  // 12  **delete** p1;    // ошибочные сценарии  std::cout << \*p1 << std::endl;  // объект по указателю p1 уже удален!  **delete** p1;  // объект по указателю p1 уже удален! |

Поэтому следует удалять объект только один раз.  
Также нередко имеет место ситуация, когда на один и тот же динамический объект указывают сразу несколько указателей. Если оператор delete применен к одному из указателей, то память объекта освобождается, и по второму указателю этот объект мы использовать уже не сможем. Если же после этого ко второму указателю применить оператор delete, то динамическая память может быть нарушена.  
В то же время недопустимость указателей после применения к ним оператора delete не означает, что эти указатели мы в принципе не сможем использовать. Мы сможем их использовать, если присвоим им адрес другого объекта:

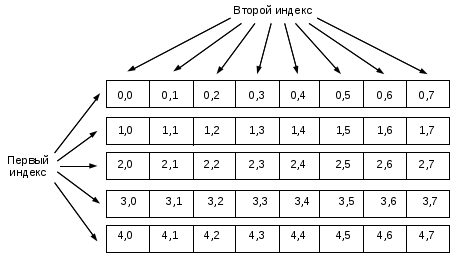
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | #include <iostream>    **int** main()  {  **int** \*p1 = **new** **int**(12);  **int** \*p2 = p1;  **delete** p1;      // адреса в p1 и p2 недопустимы        p1 = **new** **int**(11);   // p1 указывает на новый объект      std::cout << \*p1 << std::endl;  // 11  **delete** p1;    **return** 0;  } |

Здесь после удаления объекта, на который указывает p1, этому указателю передается адрес другого объекта в динамической памяти. Соответственно мы также можем использовать указатель p1. В то же время адрес в указателе p2 по-прежнему будет недействительным.

**Билет 22.**  
*Двумерные массив:*  
С позволяет создавать многомерные массивы. Простейшим видом многомерного массива является двумерный массив. Двумерный массив — это массив одномерных массивов. Двумерный массив объявляется следующим образом:  
*тип имя\_массива[размер второго измерения][размер первого измерения];*  
Следовательно, для объявления двумерного массива целых с размером 10 на 20 следует написать:  
int d[10] [20] ;  
Посмотрим внимательно на это объявление. В противоположность другим компьютерным языкам, где размерности массива отделяются запятой, С помещает каждую размерность в отдельные скобки.

Для доступа к элементу с индексами 3, 5 массива d следует использовать  
d[3] [5]  
В следующем примере в двумерный массив заносятся числа от 1 до 12, после чего массив выводится на экран.  
#include <stdio.h>  
int main(void)  
{  
int t,i, num[3][4];  
/\* загрузка чисел \*/  
for(t=0; t<3; ++t)  
for (i=0; i<4; ++i)  
num[t][i] = (t\*4)+i+1;  
  
/\* вывод чисел \*/  
for (t=0; t<3; ++t)  
{  
for (i=0; i<4; ++i)  
printf("%d  ",num[t][i]);  
printf ("\n");  
}  
return 0;  
}  
В данном примере num[0][0] имеет значение 1, num[0][1] имеет значение 2, num[0][2] - 3 и так далее. num[2][3] имеет значение 12.  
Двумерные массивы сохраняются в виде матрицы, где первый индекс отвечает за строку, а второй -за столбец. Это означает, что правый индекс изменяется быстрее левого, если двигаться по массиву в порядке расположения элементов в памяти. На рис. показано графическое представление двумерного массива в памяти. Левый индекс можно рассматривать как указатель на строку.  
Число байт в памяти, требуемых для размещения двумерного массива, вычисляется следующим  образом:  
*число байт = размер второго измерения \* размер первого измерения \* sizeof (базовый тип)*  
Предполагая наличие в системе 2-байтных целых, целочисленный массив с размерностями 10 на 5 будет занимать 10 \* 5 \* 2, то есть 100 байт.  
Когда двумерный массив используется как аргумент функции, передается указатель на первый элемент. Функция, получающая двумерный массив, должна, как минимум, определять размер первого измерения, поскольку компилятору необходимо знать длину каждой строки для корректной  индексации массива. Например, функция, получающая двумерный целочисленный массив с размерностями 5, 10, будет объявляться следующим образом:  
void func1 (int х[] [10])  
{  
...  
}  
Можно определить размер второго измерения, но это не обязательно. Компилятору нужно знать размер первого измерения для правильного выполнения операторов типа  
х [2] [4]  
в функции. Если длина строки не известна, невозможно узнать, где начинается следующая строка.  
Краткая программа, приведенная ниже, использует двумерный массив для хранения оценок каждого студента в классах учителя. Программа предполагает, что учитель имеет три класса и в каждом классе может быть максимум 30 студентов. Обратим внимание, как осуществляется доступ к grade из каждой функции:  
  
#include <conio.h>  
#include <ctype.h>  
#include <stdio.h>  
#include <stdlib.h>  
#define CLASSES 3  
#define GRADES 30  
int grade[CLASSES][GRADES];  
void disp\_grades(int g[] [GRADES]), enter\_grades(void);  
int get\_grade(int num);  
int main(void) /\* программа для хранения оценок \*/  
{  
char ch;  
for (;;) {  
do {  
printf("(E)nter grades\n");  
printf("(R)eport grades\n");  
printf("(Q) uit\n");  
ch = toupper(getche());  
}  
while(ch!='E' && ch!='R' && ch!='Q');  
switch(ch)  
{  
case 'E':  
enter\_grades();  
break;  
case 'R':  
disp\_grades(grade);  
break;  
case 'Q':  
return 0;  
}  
}  
}  
  
/\* ввод каждой оценки студентов \*/  
void enter\_grades(void)  
{  
int t, i;  
for(t=0; t<CLASSES; t++) {  
printf("Class # %d:\n", t+1);  
for(i=0; i<GRADES; ++i)  
grade[t][i] = get\_grade (i);  
}  
}  
  
/\* реальный ввод оценки \*/  
int get\_grade(int num) {  
char s [80];  
printf("enter grade for student # %d:\n", num+1);  
gets (s);  
return(atoi(s));  
}  
  
/\* вывод оценок класса \*/  
void disp\_grades(int g[] [GRADES])  
{  
int t, i;  
for(t=0; t<CLASSES; ++t)  
{  
printf("Class # %d:\n", t+1);  
for(i=0; i<GRADES; ++i)  
printf("grade for student #%d is %d\n", i+1, g[t][i]);  
}  
}

|  |
| --- |
|  |
| Рисунок. Размещение двумерного массива в памяти |



**Билет 23.**  
*Передача массива в функцию:*  
Здесь рассмотрена операция по передаче массивов в качестве аргументов функции, поскольку существуют исключения из стандартного правила передачи по значению.  
Когда массив используется в качестве аргумента функции, передается только адрес массива, а не копия всего массива. При вызове функции с именем массива в функцию передается указатель на первый элемент массива. (Надо помнить, что в С имена массивов без индекса - это указатели на первый элемент массива.) Параметр должен иметь тип, совместимый с указателем. Имеется три способа объявления параметра, предназначенного для получения указателя на массив. Во-первых, он может быть объявлен как массив, как показано ниже:  
#include <stdio.h>  
void display(int num[10]);  
int main (void) /\* вывод чисел \*/  
{  
int  t [10], i;  
for (i=0; i<10; ++i) t[i]=i;  
display(t);  
return 0;  
}  
  
void display(int num[10])  
{  
int i;  
for (i=0; i<10; i++) printf ("%d", num[i]);  
}  
Хотя параметр num объявляется как целочисленный массив из десяти элементов, С автоматически преобразует его к целочисленному указателю, поскольку не существует параметра, который мог бы на самом деле принять весь массив. Передается только указатель на массив, поэтому должен быть параметр, способный принять его.  
Следующий способ состоит в объявлении параметра для указания на безразмерный массив, как показано ниже:  
void display(int num[])  
{  
int i;  
for (i=0; i<10; i++) printf("%d ", num[i]);  
}  
где num объявлен как целочисленный массив неизвестного размера. Поскольку С не предоставляет проверку границ массива, настоящий размер массива не имеет никакого отношения к параметру (но, естественно, не к программе). Данный метод объявления также определяет num как целочисленный указатель.  
Последний способ, которым может быть объявлен num, - это наиболее типичный способ, применяемый при написании профессиональных программ, - через указатель, как показано ниже:  
void display(int \*num)  
{  
int i;  
for (i=0; i<10; i++) printf ("%d ", num[i]);  
}  
Он допустим, поскольку любой указатель может быть индексирован с использованием [], если он является массивом. (На самом деле массивы и указатели очень тесно связаны друг с другом.) Все три метода объявления параметра приводят к одинаковому результату - указателю. С другой стороны, элемент массива используется как аргумент, трактуемый как и другие простые переменные. Например, программа может быть написана без передачи всего массива:  
#include <stdio.h>  
void display(int num);  
int main(void) /\* вывод чисел \*/  
{  
int t[10], i;  
for (i=0; i<10; ++i) t[i] = i;  
for (i=0; i<10; i++) display(t[i]);  
return 0;  
}  
void display(int num)  
{  
printf ("%d ", num);  
}  
Как можно видеть, в display() передается параметр типа int. Не имеет значения, что display() вызывается с элементом массива в качестве параметра, поскольку передается только одно значение.  
Важно понять, что при использовании массива в качестве аргумента функции происходит передача в функцию его адреса. Это означает, что код внутри функции действует и может изменять настоящее значение массива, используемого при вызове. Например, рассмотрим функцию print\_upper(), выводящую строку прописными буквами:  
#include <stdio.h>  
#include <ctype.h>  
void print\_upper(char \*string);  
int main(void) /\* вывод строки в верхнем регистре \*/  
{  
char s[80];  
gets (s);  
print\_upper(s) ;  
return 0;  
}  
void print\_upper(char \*string)  
{  
register int t;  
for(t=0; string[t]; ++t)  
{  
string[t] = toupper(string[t]);  
printf("%c", string[t]);  
}  
}  
После вызова print upper() происходит изменение содержимого массива s в main(). Если это не нужно, следует переписать программу следующим образом:  
# include <stdio.h>  
#include <ctype.h>  
void print upper(char \*string);  
int main(void) /\* вывод строки в верхнем регистре \*/  
{  
char s[80];  
gets(s);  
print\_upper(s);  
return 0;  
}  
void print\_upper(char \*string)  
{  
register int t;  
for(t=0; string[t]; ++t)  
printf ("%c",  toupper (string[t]));  
}  
В данной версии содержимое массива s остается неизменным, поскольку значения не меняются.  
Классический пример передачи массивов в функции находится в стандартной библиотечной функции gets(). Хотя gets() из библиотеки Borland С++ гораздо сложнее, функция, показанная в данном примере, содержит основную идею работы. Для того, чтобы избежать путаницы и не вызвать стандартную функцию, данная функция называется xgets().  
  
/\* простейшая версия стандартной библиотечной функции gets() \*/  
void xgets (char \*s)  
{  
register char ch;  
register int t;  
for(t=0; t<79; )  
{  
ch = getche();  
switch(ch)  
{  
case ' \r':  
s[t] = '\0'; /\* null завершает строку \*/  
return;  
case '\b':  
if(t>0) t-;  
break;  
default:  
s[t] = ch;  
t++;  
}  
}  
s[79] = ' \0';  
}  
Функция xgets() должна вызываться с указателем на символ. Это может быть имя символьного массива, который по определению является указателем на символ. xgets() организует цикл for от 0 до 79. Таким образом предотвращается ввод больших строк с клавиатуры. Если набирается более 80 символов, функция завершает работу. Поскольку C не имеет проверки границ массива, следует убедиться, что массив, передаваемый в xgets(), может принять, по крайней мере, 80 символов. По мере набора символов на клавиатуре они вводятся в строку. Если набирается забой, счетчик t уменьшится на 1. При нажатии на ввод помещается нулевой символ в конец строки, то есть строка оканчивается. Поскольку массив, используемый при вызове xgets(), модифицируется, после возврата он будет содержать набранные символы.