## 18/19. Структура языка программирования: пользовательские типы данных (структуры, перечисления, объединения). Определение, примеры (C++). Массивы данных пользовательского типа. Перегрузка операторов для пользовательских типов / Структура языка программирования: пользовательские типы данных. Определение структуры, примеры (C++). Перегрузка операторов для пользовательских типов.

**Пользовательские типы данных:** типы, создаваемые пользователем, на основе того, что доступно в языке; всегда есть объявление типа.

**С++: struct** addres{ //**структура** - это набор переменных, объединенных общим именем. Она обеспечивает удобный способ организации взаимосвязанных данных.

string street;

int number\_of\_house;};

**struct** rgb\_color{ //**битовое поле** – особый вид полей структуры, используемый для

unsigned red\_value: 3; // плотной упаковки данных.

unsigned green\_value: 3;

unsigned blue\_value: 3;};

**объединение** - это пользовательская переменная, которая может хранить объекты различного типа и размера, для их размещения выделяется одна общая память, размерность определяется размерностью максимального элемента объединения.

**union** bit{ char ch;

int i;};

**enum** coin{ //**перечисление** - набор именованных целых констант

penny, nickel, dime};

**typedef** unsigned int nomer; // определяет новое имя типа данных, новый тип при этом не создается, уже существующий тип получает новое имя

***Массивы данных пользовательского типа:*** это совокупность данных (переменных) пользовательского типа и объединенных под одним именем. Доступ к отдельному элементу массива осуществляется с помощью индекса.

* **Массив struct:**

struct{

int i;

char ch;} S[5];

Обращение: S[0].i;

* **Массив union:**

union{float f;

int i;} U[5];

Обращение: U[1].f;

Перегрузка: возможность перегрузки есть только для пользовательских типов/классов реализация в одной области видимости нескольких различных вариантов применения оператора, имеющих одно и то же имя, но различающихся типами параметров, к которым они применяются

#include <iostream>

struct Vector {

int x;

int y;};

// Перегрузка оператора сложения (+)

Vector operator+(const Vector& v1, const Vector& v2) {

Vector result;

result.x = v1.x + v2.x;

result.y = v1.y + v2.y;

return result;}

int main() {

Vector v1 = {1, 2};

Vector v2 = {3, 4};

Vector v3 = v1 + v2;

std::cout << v3 << std::endl; // Вывод: (4, 6)

В этом примере мы определяем структуру **Vector**, представляющую двумерный вектор с координатами x и y. Затем мы перегружаем операторы сложения (**+**) и вычитания (**-**) для векторов.

Функция **operator+** принимают два вектора в качестве параметров и выполняют соответствующие операции с компонентами векторов. Результат операции возвращается как новый вектор.

## 20. Структура языка программирования: понятие конфликта имен, область видимости переменных, пространства имен, псевдонимы пространства имен.

**Область видимости переменных в C++**: доступность переменных по их идентификатору в разных частях (блоках программы).

Переменная должна быть объявлена до ее использования; переменная объявленная во внутреннем блоке (локальная переменная {…}) не доступна во внешнем; переменная объявленная во внешнем блоке доступна во внутреннем; во внутреннем блоке переменная может быть переобъявлена.

**Пространство имен:** именованная область видимости. Применяется для разрешения конфликтов имен. Имена, объявленные в одном пространстве имен, не будут вступать в конфликт с аналогичными именами, объявленными в другой области.

Примеры**:** namespace, using, псевдонимы пространства имен.

**Псевдоним пространства имен**: Имена пространств имен должны быть уникальными, из-за чего зачастую они получаются не слишком короткими. Если длинное имя оказывается трудночитаемым или его сложно вводить в файле заголовка, где нельзя использовать директиву using, можно создать псевдоним пространства имен, который будет служить в качестве сокращения фактического имени.

Пространства имен помогают предотвратить конфликты имен, но не устранить их полностью. Такой конфликт может произойти, когда одно и то же имя объявляется в двух разных пространствах имен и затем предпринимается попытка сделать видимым оба пространства. В таком случае для указания предполагаемого пространства имен явным образом можно воспользоваться оператором разрешения области видимости (**::**). Оператор разрешения области видимости позволяет указать конкретное пространство имен, из которого должен быть взят идентификатор.

Например, предположим, у нас есть два пространства имен: **Namespace1** и **Namespace2**, и оба пространства имен объявляют функцию с одним и тем же именем **foo**. Чтобы указать, какую именно функцию **foo** мы хотим использовать, мы можем явно указать пространство имен с помощью оператора разрешения области видимости

#include <iostream>

namespace Namespace1 {

void foo() {

std::cout << "Function foo() in Namespace1" << std::endl;}}

namespace Namespace2 {

void foo() {

std::cout << "Function foo() in Namespace2" << std::endl;}}

int main() {

Namespace1::foo(); // Вызов функции foo() из пространства имен Namespace1

Namespace2::foo(); // Вызов функции foo() из пространства имен Namespace2

return 0;}

## 21.Структура языка программирования: определение выражения, его состав, порядок вычисления выражения. Символ окончания последовательности. Примеры (С++).

* 1. ***Выражение*** – объединение литералов, имен (переменных, функций и пр.), операторов и специальных символов, служащих для вычисления выражения или достижения побочных эффектов (например: при применении в выражении функций).
* Выражения состоят из операндов, знаков операций и скобок.
* Операнды выражения задают данные для вычислений.
* Операции задают действия, которые необходимо выполнить.
* Операнд может быть выражением, литералом или переменной.
* Порядок вычисления выражения с операторами одинакового приоритета не определен.

Символ окончания последовательности (в языке программирования C++ – это точка с запятой) определяет точку последовательности

(sequence point), в которой завершились все вычисления и побочные эффекты. Точка последовательности – момент времени, когда побочные эффекты вычисленных выражений уже случились, а побочные эффекты следующих в последовательности выражений еще не начались.

## 22.Структура языка программирования: выражения (lvalue, rvalue, побочные эффекты, точка последовательности, унарные, бинарные и тернарные выражения), константные выражения, укороченное вычисление. Примеры (С++).

*lvalue (locator value)* представляет собой объект, который занимает идентифицируемое место в памяти (например, имеет адрес).

*rvalue* определено путём исключения, говоря, что любое выражение является либо *lvalue*, либо *rvalue*. Таким образом из определения *lvalue* следует, что *rvalue* — это выражение, которое *не представляет* собой объект, который занимает идентифицируемое место в памяти.

**Элементарные примеры**

Предположим, мы имеем дело с переменной целого типа:

**int** **var**;

**var** = 4;

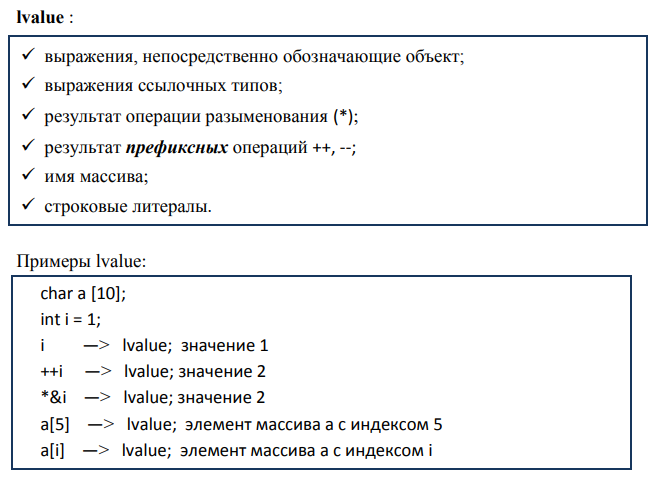
Оператор присваивания ожидает lvalue с левой стороны, и var является lvalue, потому что это объект с идентифицируемым местом в памяти. С другой стороны, следующие выражения приведут к ошибкам:

4 = **var**; // ERROR!

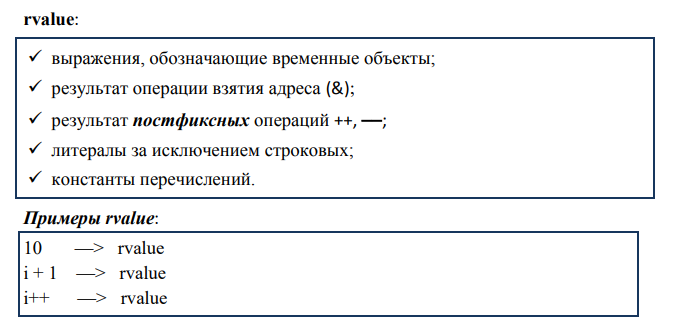
(**var** + 1) = 4; // ERROR!

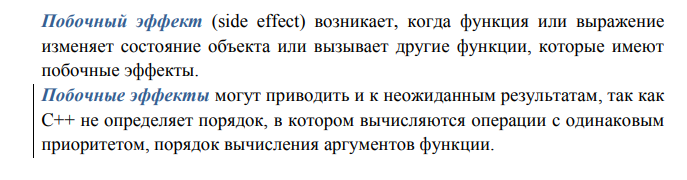
Ни константа 4, ни выражение var+1 не являются lvalue  
(что автоматически их делает rvalue). Они не lvalue, потому что оба являются временным результатом выражений, которые не имеют определённого места в памяти (то есть они могут находится в каких-нибудь временных регистрах на время вычислений). Таким образом, присваивание в данном случае не несёт в себе никакого семантического смысла. Иными словами — некуда присваивать.

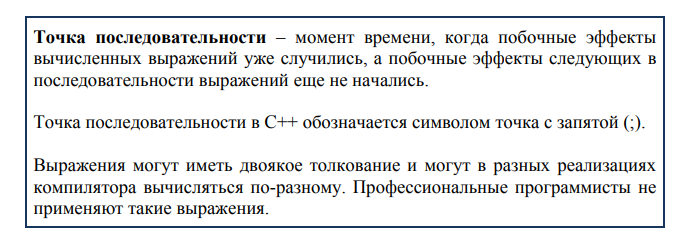
lvalue (именующее выражение) – это ссылка на значение; могут использоваться в левой и правой части оператора присваивания. Имя переменной, ссылка на элемент массива по индексу, вызов функции возвращающей указатель, всегда связаны с областью памяти, адрес которой известен

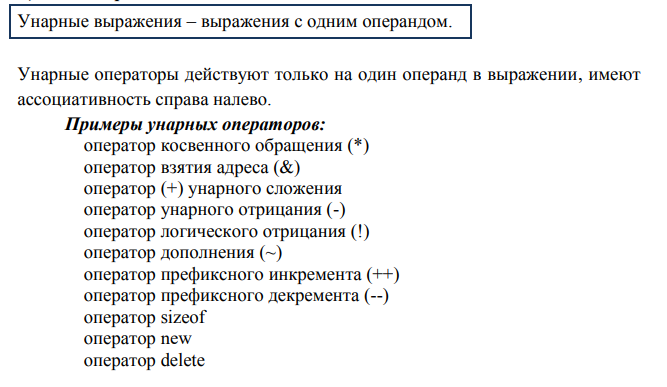


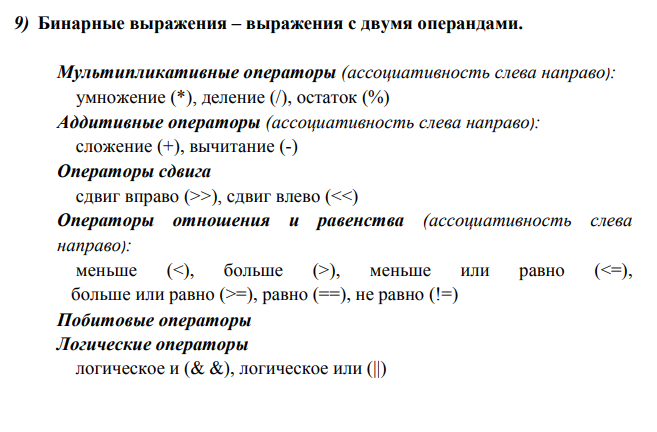
rvalue (значащее выражение) – может использоваться только в правой части оператора присваивания (не связано с адресом, связано только со значением; это могут быть литералы, вызов функции, возвращающей значение).

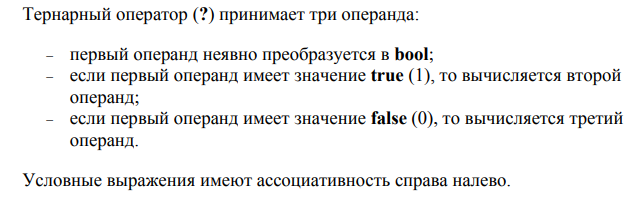












**Укороченное вычисление**

Укороченное вычисление – вычисление, значение которого может быть вычислено по части выражения.

Таблица истинности для логических операторов:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **x** | **y** | **x && y** | **x || y** | **! x** |
| 0 | 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 |

## 23. Структура языка программирования: инструкции языка программирования, объявление, определение, инициализация, простые и составные инструкции, инструкции выбора, инструкции циклов, инструкции переходов, примеры (С++).

**Инструкция (оператор)** — наименьшая автономная часть языка программирования; команда или набор команд. Программа обычно представляет собой последовательность инструкций.

**Объявление** – ***описание некоторой сущности*** (сигнатура функции, определение типа, описание внешней переменной, шаблон и т.п.). Объявление уведомляет компилятор о её существовании и свойствах.

**Определение** – ***реализация некоторой сущности*** (переменная, функция, метод класса и т.п.). При обработке определения компилятор генерирует информацию для объектного модуля – исполняемый код, резервирование памяти под переменную и т.д.

**Простые инструкции** завершаются точкой с запятой. **Составные инструкции**, или блоки ({ }), состоят из одной или более инструкций (каждая из которых сама может быть составной), заключенных в фигурные скобки.

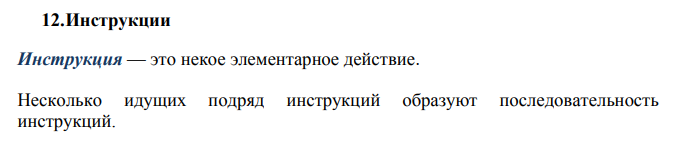
**Инструкции выбора** *if*и *switch*, позволяют выполнять те или иные участки кода в зависимости от выполнения условий.

**Инструкции циклов** используются для многократного повторения фрагментов кода. Цикл *for* мы можем использовать, если знаем точное количество действий (итераций). Когда мы не знаем, сколько итераций должен произвести цикл, нам понадобится цикл *while* или *do…while*. Цикл *do…while* почти ничем не отличается от цикла *while*, за исключением того, что тело цикла гарантированно выполняется хотя бы один раз.

**Инструкции переходов** выполняют немедленную локальную передачу контроля. Оператор *break* завершает выполнение ближайшего внешнего цикла или условного оператора, в котором он находится. Управление передается оператору, который расположен после оператора, при его наличии. Инструкцию *continue* можно располагать только внутри цикла. Она вызывает переход к следующей итерации самого внутреннего содержащего ее цикла. С помощью *return* функция возвращает управление в программу, откуда была вызвана. Если за *return* следует выражение, то его значение возвращается вызвавшей эту функцию программе. Оператор *goto* осуществляет безусловную передачу управления оператору, метка которого задана идентификатором. Метка оператора имеет значение только для оператора *goto*; в остальных случаях метки операторов игнорируются. Повторное объявление меток невозможно.

**Инструкции обработки исключений** используют три ключевых слова: *try*, *catch* и *throw*. Те инструкции программы, где ожидается возможность появления исключительных ситуаций, содержатся в бло­ке *try*. Если в блоке *try* возникает исключение, т. е. ошибка, то генерируется исключение throw. Исклю­чение перехватывается, используя *catch*, и обрабатывается.

## 24. Структура языка программирования: инструкции языка программирования, инструкции обработки исключений, примеры (С++).



**Инструкции выбора** *if*и *switch*, позволяют выполнять те или иные участки кода в зависимости от выполнения условий.

**Инструкции циклов** используются для многократного повторения фрагментов кода. Цикл *for* мы можем использовать, если знаем точное количество действий (итераций). Когда мы не знаем, сколько итераций должен произвести цикл, нам понадобится цикл *while* или *do…while*. Цикл *do…while* почти ничем не отличается от цикла *while*, за исключением того, что тело цикла гарантированно выполняется хотя бы один раз.

**Инструкции переходов** выполняют немедленную локальную передачу контроля. Оператор *break* завершает выполнение ближайшего внешнего цикла или условного оператора, в котором он находится. Управление передается оператору, который расположен после оператора, при его наличии. Инструкцию *continue* можно располагать только внутри цикла. Она вызывает переход к следующей итерации самого внутреннего содержащего ее цикла. С помощью *return* функция возвращает управление в программу, откуда была вызвана. Если за *return* следует выражение, то его значение возвращается вызвавшей эту функцию программе. Оператор *goto* осуществляет безусловную передачу управления оператору, метка которого задана идентификатором. Метка оператора имеет значение только для оператора *goto*; в остальных случаях метки операторов игнорируются. Повторное объявление меток невозможно.

**Инструкции обработки исключений** используют три ключевых слова: *try*, *catch* и *throw*. Те инструкции программы, где ожидается возможность появления исключительных ситуаций, содержатся в бло­ке *try*. Если в блоке *try* возникает исключение, т. е. ошибка, то генерируется исключение throw. Исклю­чение перехватывается, используя *catch*, и обрабатывается.

#include <iostream>

int divide(int a, int b) {

if (b == 0) {

throw std::runtime\_error("Division by zero");}

return a / b;}

int main() {

try {

int result = divide(10, 2);

std::cout << "Result: " << result << std::endl;

result = divide(10, 0);

std::cout << "Result: " << result << std::endl; // Этот код не выполнится

} catch (const std::exception& e) {

std::cout << "Exception caught: " << e.what() << std::endl;}return 0;}

## 25/26. Структура языка программирования: программные конструкции (блоки, функции, процедуры и пр.). Функции: передача параметров по значению и по ссылке, передача переменного числа параметров в функции C++. Примеры/ Структура языка программирования: программные конструкции (блоки, функции, процедуры и пр.). Передача параметров в функцию. Передача параметров по значению и по ссылке, передача значений параметров по умолчанию в функции C++. Примеры.

**Программные конструкции:**

**Блоки**: логически сгруппированный набор идущих подряд инструкций в [исходном коде](https://ru.wikipedia.org/wiki/%D0%98%D1%81%D1%85%D0%BE%D0%B4%D0%BD%D1%8B%D0%B9_%D0%BA%D0%BE%D0%B4) программы, является основой парадигмы [структурного программирования](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)

**Пример С++:**

{

int a = 1;

int b = 2;

int c = a + b;

return c;}

**Функции:** фрагмент программного кода, к которому можно обратиться из другого места программы

**Пример С++:**

void name(char\* text)

{ std::cout<<text<< std::endl;}

**Процедуры:** функции не возвращающие значения, return не обязателен, но желателен для явного указания завершения функции

**Пример С++:**

 void ValidMessage()

 { cout << "Пароль введен верно" << endl;}

**Передача параметров:** передача параметров в функцию происходит через стек. Код, вызывающий функцию, знает, сколько параметров ей передать и каковы значения этих параметров

**Передача параметров по значению:** они при выходе из функции не изменятся

**Пример С++:**

int func(int k){

k\*=2;

return k;}

void main(){

int z=1, y=3, k;

k=func(z)+func(y);

cout<<z<<" "<<y;

}

Z = 1, y = 3

**по ссылке:** при выходе из функции из значения могут измениться

int func(int &k){

k\*=2;

return k;

}

void main(){

int z=1, y=3, k;

k=func(z)+func(y);

cout<<z<<" "<<y;

}

Z = 2, y = 6

**Передача переменного числа параметров в функции С++:**

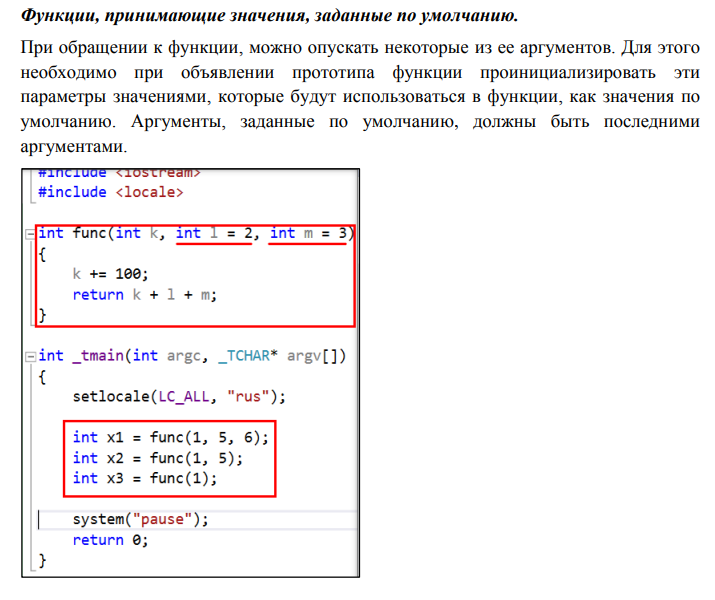
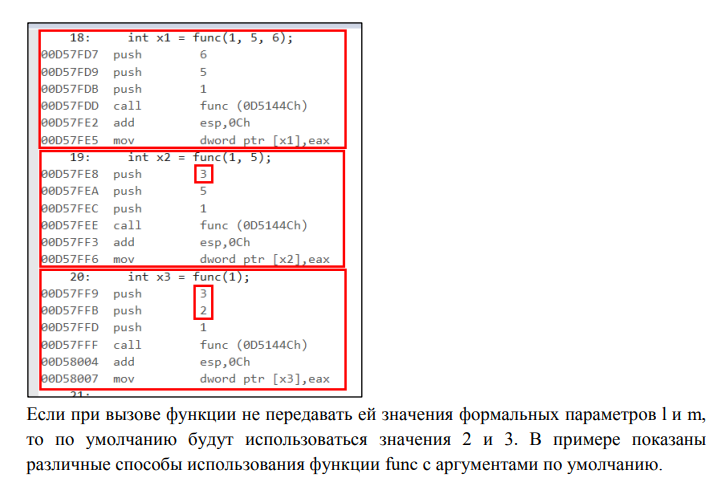
По умолчанию параметры передаются в функцию через стек. Поэтому, технически, нет ограничения на количество передаваемых параметров. Функции с переменным числом параметров объявляются как обычные функции, но вместо недостающих аргументов ставится многоточие. Количество параметров и их типы известны только при вызове функции.

Передать функции параметры можно двумя способами:

* явно передать обязательный аргумент, задающий число параметров;
* добавить в конец списка параметр с уникальным значением, по которому будет определяться конец списка параметров.

Общий принцип работы следующий: в функции для доступа к списку параметров устанавливается указатель, значением которого будет адрес явного параметра в списке, далее изменяется значение этого указателя, чтобы переместиться на следующий параметр.

Передача значений параметров по умолчанию в функции C++:

## 27. Структура языка программирования: программные конструкции (блоки, функции, процедуры и пр.). Передача параметров в функцию. Реализация программных конструкций: лямбда-функции в С++. Примеры.

**Блоки**: логически сгруппированный набор идущих подряд инструкций в [исходном коде](https://ru.wikipedia.org/wiki/%D0%98%D1%81%D1%85%D0%BE%D0%B4%D0%BD%D1%8B%D0%B9_%D0%BA%D0%BE%D0%B4) программы, является основой парадигмы [структурного программирования](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)

**Пример С++:**

{

int a = 1;

int b = 2;

int c = a + b;

return c;}

**Функции:** фрагмент программного кода, к которому можно обратиться из другого места программы

**Пример С++:**

void name(char\* text)

{std::cout<<text<< std::endl;}

**Процедуры:** функции не возвращающие значения, return не обязателен, но желателен для явного указания завершения функции

**Пример С++:**

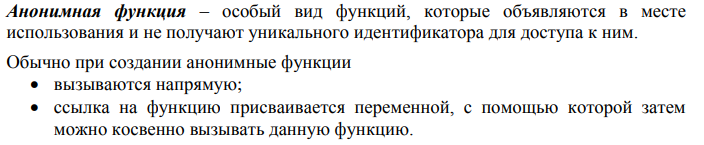
 void ValidMessage()

 {

         cout << "Пароль введен верно" << endl;

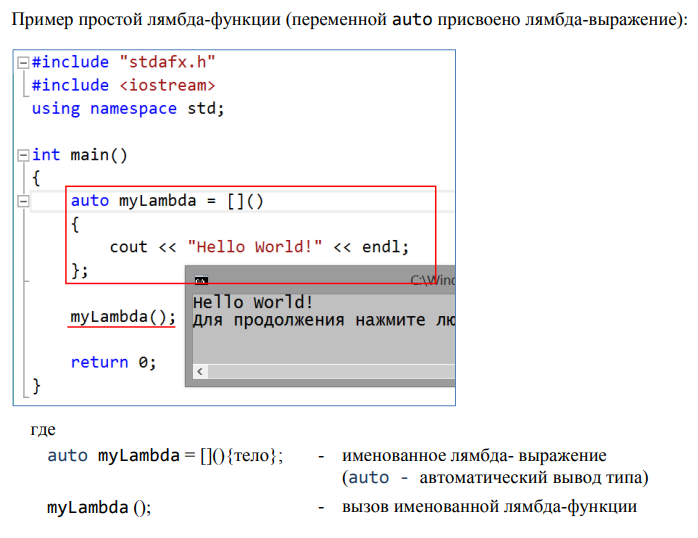
 }

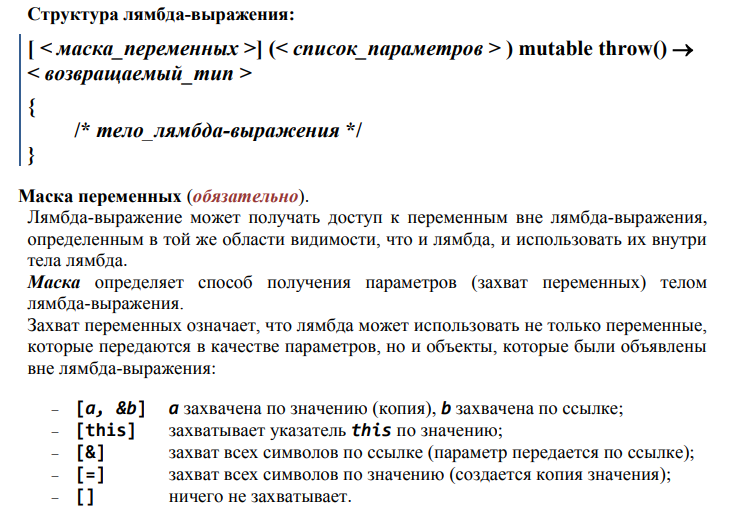
**Передача параметров:** передача параметров в функцию происходит через стек. Код, вызывающий функцию, знает, сколько параметров ей передать и каковы значения этих параметров



Лямбда-функция (также известная как анонимная функция или безымянная функция) в C++ - это компактный способ определения и использования функции без явного имени.

Лямбда-функции обычно используются в ситуациях, когда требуется определить небольшую функцию, которая будет использоваться локально внутри другой функции или контекста. Они удобны в использовании, когда необходимо передать функцию как аргумент в другую функцию





## 28. Структура языка программирования: программные конструкции (блоки, функции, процедуры и пр.). Передача параметров в функцию. Реализация программных конструкций в C++. Перегружаемые функции в C++. Inline-функции в C++. Шаблоны функций С++.

**Блоки**: логически сгруппированный набор идущих подряд инструкций в [исходном коде](https://ru.wikipedia.org/wiki/%D0%98%D1%81%D1%85%D0%BE%D0%B4%D0%BD%D1%8B%D0%B9_%D0%BA%D0%BE%D0%B4) программы, является основой парадигмы [структурного программирования](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)

**Пример С++:**

{

int a = 1;

int b = 2;

int c = a + b;

return c;}

**Функции:** фрагмент программного кода, к которому можно обратиться из другого места программы

**Пример С++:**

void name(char\* text)

{std::cout<<text<< std::endl;}

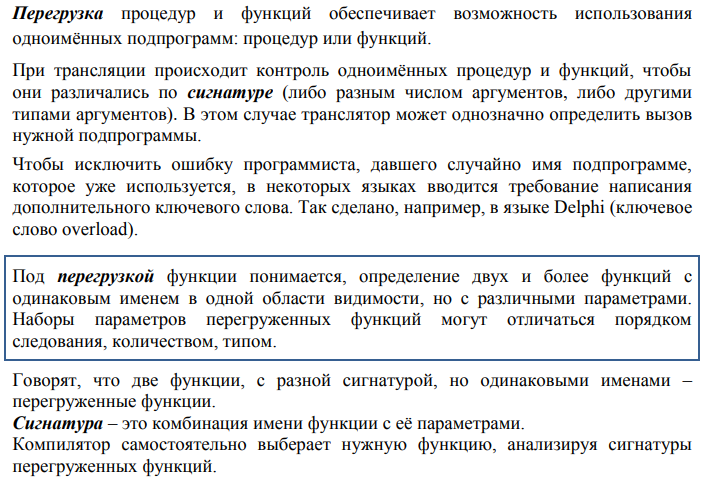
**Процедуры:** функции не возвращающие значения, return не обязателен, но желателен для явного указания завершения функции

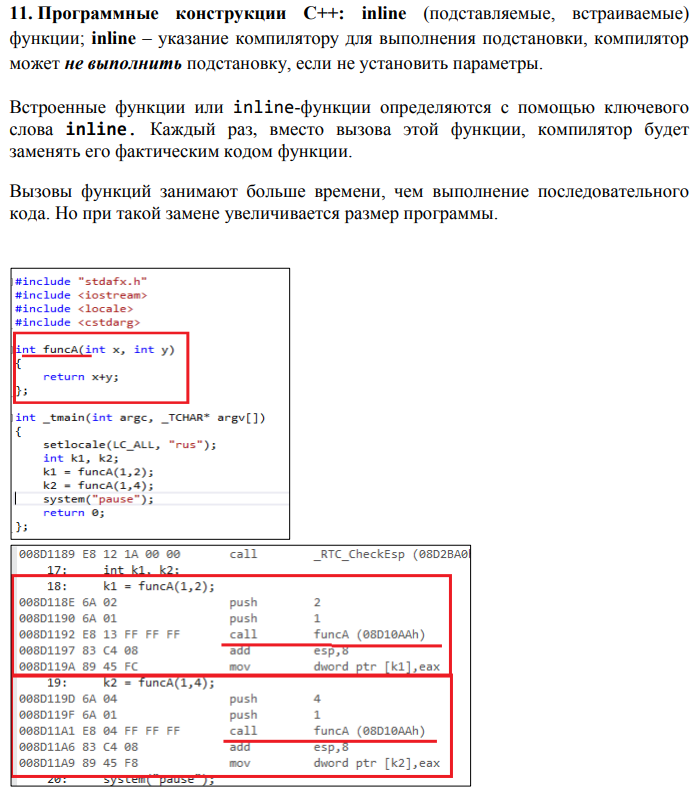
**Пример С++:**

 void ValidMessage()

 {cout << "Пароль введен верно" << endl;}

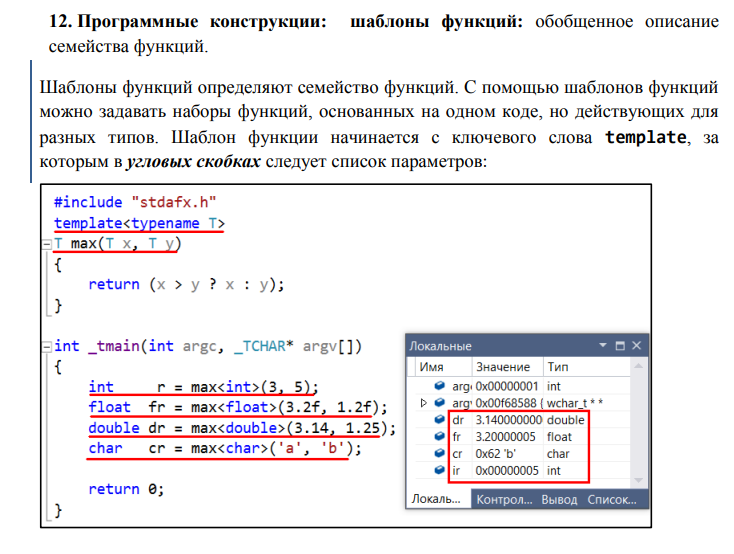
**Передача параметров:** передача параметров в функцию происходит через стек. Код, вызывающий функцию, знает, сколько параметров ей передать и каковы значения этих параметров





inline int sum(int a, int b)

{return a + b;}



## 29. Структура языка программирования: программные конструкции (блоки, функции, процедуры и пр.). Передача параметров в функцию. Реализация программных конструкций в C++: соглашения о вызовах (\_cdecl).

**Блоки**: логически сгруппированный набор идущих подряд инструкций в [исходном коде](https://ru.wikipedia.org/wiki/%D0%98%D1%81%D1%85%D0%BE%D0%B4%D0%BD%D1%8B%D0%B9_%D0%BA%D0%BE%D0%B4) программы, является основой парадигмы [структурного программирования](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)

**Пример С++:**

{

int a = 1;

int b = 2;

int c = a + b;

return c;

}

**Функции:** фрагмент программного кода, к которому можно обратиться из другого места программы

**Пример С++:**

void name(char\* text)

{

std::cout<<text<< std::endl;

}

**Процедуры:** функции не возвращающие значения, return не обязателен, но желателен для явного указания завершения функции

**Пример С++:**

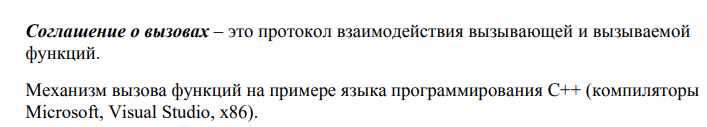
 void ValidMessage()

 {

         cout << "Пароль введен верно" << endl;

 }

**Передача параметров:** передача параметров в функцию происходит через стек. Код, вызывающий функцию, знает, сколько параметров ей передать и каковы значения этих параметров



**\_cdecl:** Параметры функций помещаются в стек, порядок параметров "справа налево". Параметры, размер которых меньше 4-х байт, расширяются до 4-х байт. Адрес возврата кладется в стек поверх параметров.

## 30. Структура языка программирования: программные конструкции (блоки, функции, процедуры и пр.). Передача параметров в функцию. Реализация программных конструкций в C++: соглашения о вызовах (\_stdcall).

**Блоки**: логически сгруппированный набор идущих подряд инструкций в [исходном коде](https://ru.wikipedia.org/wiki/%D0%98%D1%81%D1%85%D0%BE%D0%B4%D0%BD%D1%8B%D0%B9_%D0%BA%D0%BE%D0%B4) программы, является основой парадигмы [структурного программирования](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)

**Пример С++:**

{

int a = 1;

int b = 2;

int c = a + b;

return c;

}

**Функции:** фрагмент программного кода, к которому можно обратиться из другого места программы

**Пример С++:**

void name(char\* text)

{std::cout<<text<< std::endl;}

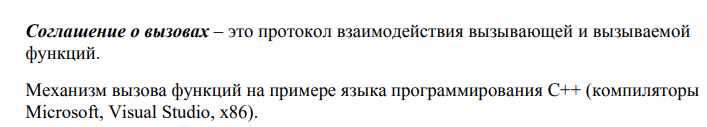
**Процедуры:** функции не возвращающие значения, return не обязателен, но желателен для явного указания завершения функции

**Пример С++:**

 void ValidMessage()

 {cout << "Пароль введен верно" << endl;}

**Передача параметров:** передача параметров в функцию происходит через стек. Код, вызывающий функцию, знает, сколько параметров ей передать и каковы значения этих параметров



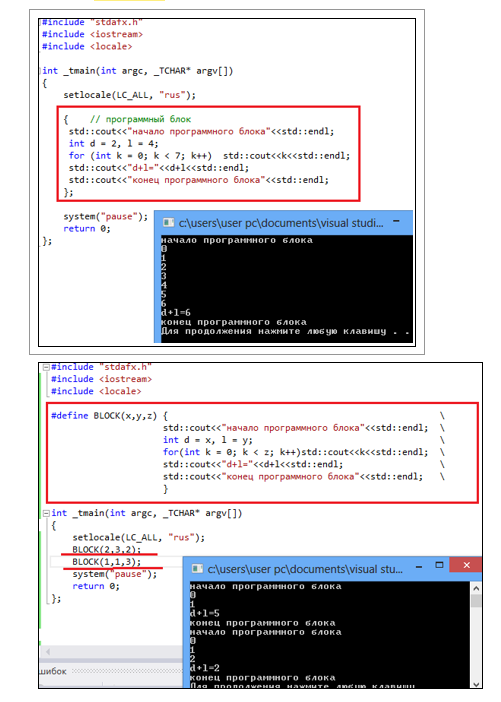
**\_stdcall:** Параметры помещаются в стек, порядок параметров "справа налево". Адрес возврата кладется в стек поверх параметров.

Стек освобождает вызываемый код. В последней инструкции вызываемого кода RET указывается суммарный размер в байтах всех параметров процедуры. Команда RET после извлечения адреса возврата прибавляет к регистру ESP указанное значение. Целостность стека восстанавливается вызываемым кодом.

Возвращаемый параметр передается через регистр EAX.

## 31. Структура языка программирования: программные конструкции (блоки, функции, процедуры и пр.). Передача параметров в функцию. Реализация программных конструкций в C++: соглашения о вызовах (\_fastcall).

Программные блоки представлены:

****

Функции:

***Функция*** – фрагмент программного кода, к которому можно обратиться из другого места программы. Функция должна быть ***объявлена*** и ***определена***.

***Функция*** – подпрограмма, выполняющая какие-либо операции и возвращающая значение.

***Процедура*** – подпрограмма, которая выполняет операции, и не возвращает значения.

**Передача параметров в функцию**

Передача параметров в функцию происходит через стек.

***Код, вызывающий функцию***, знает, сколько параметров ей передать и каковы значения этих параметров. В нашем примере, если код вызывает функцию sum, которая принимает два параметра типа int, то вызывающий код:

* + кладет два параметра в стек с помощью двух инструкций push. В результате этого указатель стека (ESP) **уменьшается на 2\*4** байта (вершина стека сдвигается на **8** байт);
  + выполняет инструкцию call, которая передает управление функции sum. При этом значение ESP **уменьшается еще на 4 байта**, потому что в стек помещается ***адрес точки возврата***.

Соглашение вызовах \_fastcall (не стандартизированный, для внутренних вызовов), параметры передаются через регистры (первые 2 – через регистры, остальные справа налево в стек), стек освобождает вызываемый код, возврат через регистр EAX. В Borland (Delphi) (параметры слева на право.

**Определение. С**оглашение вызовах **\_fastcall:**

Первые два параметра передаются через 2 регистра. Для передачи остальных параметров, используется стек; порядок параметров "справа налево".

Целостность стека восстанавливается вызываемым кодом. Возвращаемый параметр передается через регистр EAX.

**! В компиляторе фирмы Borland параметры передаются через регистры слева направо, если параметров больше двух, остальные помещаются в стек.**

## 32. Структура языка программирования: стандартная библиотека. Реализация стандартной библиотеки STL в C++. Понятие контейнера, итератора и алгоритма. Стандартные функции STL для работы со строками: копирование, сравнение, вычисление длины, поиск символа и подстроки.

В составе языка программирования, как правило, есть обязательный (стандартный) набор функций. Такие функции называют встроенными.

Встраиваться функции могут тремя способами: прямо в код транслятора; находиться в отдельной библиотеке; сочетание первого и второго случаев.

Основные требования к набору средств стандартной библиотеки (Бьёрн Страуструп):

* + ***эффективность***;
  + ***независимость от алгоритмов*** — должна предоставлять возможность задавать алгоритмы в качестве параметров;
  + ***удобство и безопасность***;
  + ***завершённость*;**
  + ***органично сочетаться с языком*;**
  + ***типобезопасность*;**
  + ***поддержка общепринятых стилей программирования***;
  + ***расширяемость*** — способность единообразно работать со встроенными типами данных и с типами, определяемыми пользователем

Подходы к разработке стандартных библиотек языков программирования:

* + должна содержать в себе только те процедуры и функции, которые используются практически всеми и обладают максимальной универсальностью;
  + должна содержать в себе максимально возможное количество типичных алгоритмов, обеспечивать простую работу с большинством объектов (в идеале, со всеми), с которыми может взаимодействовать программа. Пример реализации этого подхода является язык Python с девизом «Batteries included» (батарейки в комплекте).

Стандартная библиотека C++ STL:

библиотека стандартных шаблонов (STL) – набор согласованных обобщённых алгоритмов, контейнеров, средств доступа к их содержимому и различных вспомогательных функций в C++.

STL (Standard Template Library) – стандартная библиотека шаблонов. Библиотека содержит универсальные шаблонные классы и функции, реализующие большое количество распространенных универсальных алгоритмов и структур данных. Т.к. библиотека STL состоит из шаблонных классов, ее алгоритмы и структуры можно применять практически к любым типам данных.

Частью *стандартной библиотеки С++* является библиотека *STL*. Библиотека STL содержит пять основных видов компонентов:

* *контейнер* (*container*): управляет набором объектов в памяти.
* *итератор* (*iterator*): обеспечивает для алгоритма средство доступа к содержимому контейнера.
* *алгоритм* (*algorithm*): определяет вычислительную процедуру.
* *функциональный объект* (*function object*): инкапсулирует функцию в объекте для использования другими компонентами.
* *адаптер* (*adaptor*): адаптирует компонент для обеспечения различного интерфейса.

Функции для работы со строками:

strlen() (от слова length – длина)

strcpy() – копирование

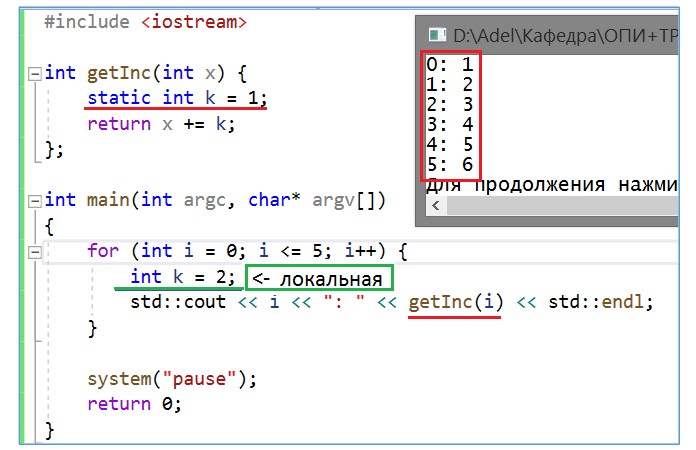
strcmp() - Функция strcmp() сравнивает две строки: s1 и s2. Она возвращает целое число, которое меньше, больше нуля или равно ему, если s1 соответственно меньше, больше или равно s2. Функция strncmp() работает аналогичным образом, но сравнивает только первые n символов строки s1.

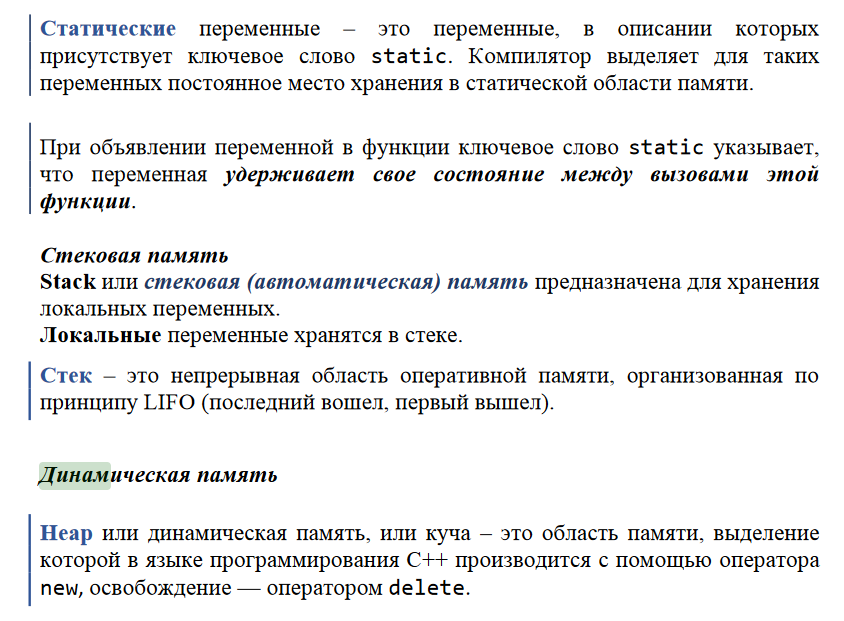
.find() - находит первое вхождение элемента. Он использует оператор = для сравнения.

## 33. **Классы памяти: код, стек, статические данные, динамическая область памяти. Структура** **динамической памяти (Heap) C++. Фрагментация динамической памяти C++.**

Область кода – память, в которой размещается код программы.

Локальная статическая память.

  
  
Переменная с ключевым словом static — это статическая переменная.  
Время ее жизни — постоянное. Область видимости статической переменной ограничена одним файлом, внутри которого она определена, ее можно использовать только после ее объявления.  
Ключевое слово static в языке С/С++ используется для двух различных целей:  
как указание типа памяти: переменная располагается в статической  
памяти; как способ ограничить область видимости переменной рамками одного  
файла (в случае описания переменной вне функции).

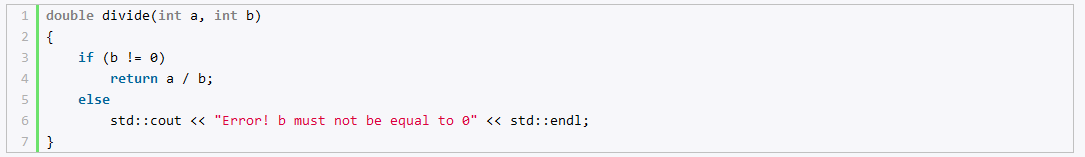
****

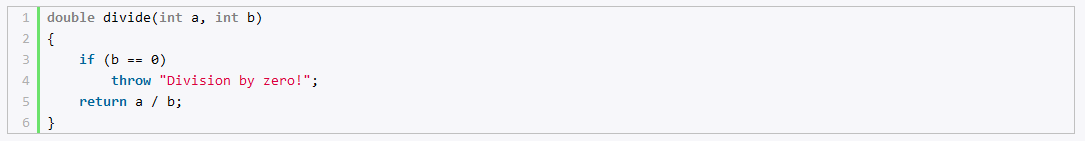
## 34. **Механизм обработки исключений: определение, назначение, применение. Реализация** **обработки исключений в C++. Пример.**

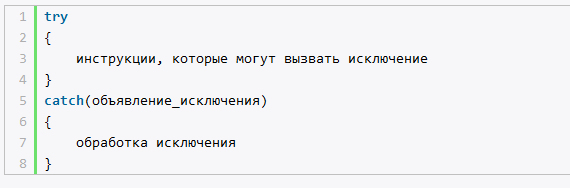
В процессе работы программы могут возникать различные ошибки. Например, при передаче файла по сети оборвется сетевое подключение или будут введены некорректные и недопустимые данные, которые вызовут падение программы. Такие ошибки еще называются исключениями. Если исключение не обработано, то при его возникновении программа прекращает свою работу. Например, в следующей программе происходит деление чисел:

 Эта программа успешно скомпилируется, но при ее выполнении возникнет ошибка, поскольку в коде производится деление на ноль, после чего программа аварийно завершится.

С одной стороны, мы можем в функции divide определить проверку и выполнять деление, если параметр b не равен 0. Однако нам в любом случае надо возвращать из функции divide некоторый результат - некоторое число. То есть мы не можем просто написать:

И в этом случае нам надо известить систему о возникшей ошибке. Для этого используется оператор throw. Оператор throw генерирует исключение. Через оператор throw можно передать информацию об ошибке. Например, функция divide могла бы выглядеть следующим образом:

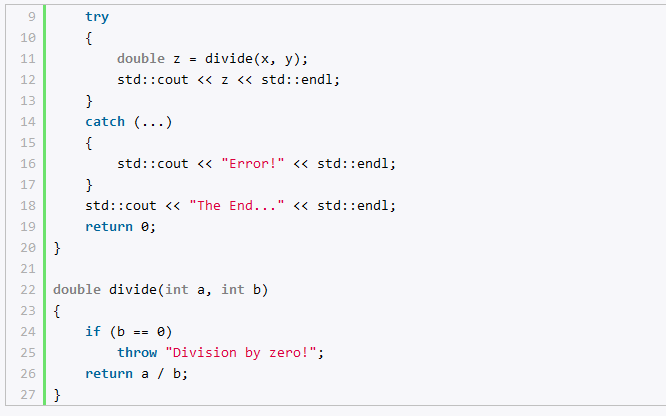
****То есть если параметр b равен 0, то генерируем исключение. Но это исключение еще надо обработать в коде, где будет вызываться функция divide. Для обработки исключений применяется конструкция try...catch. Она имеет следующую форму:

****

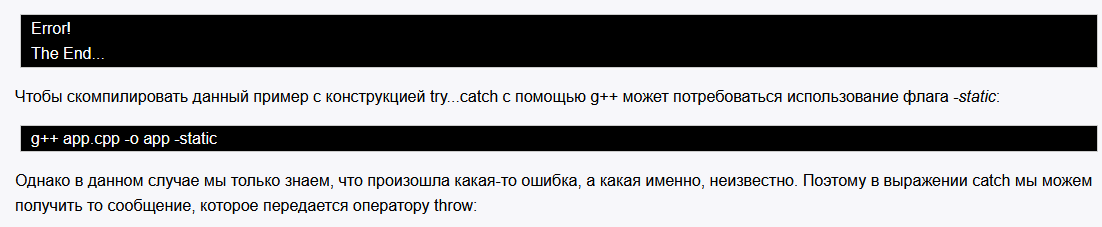
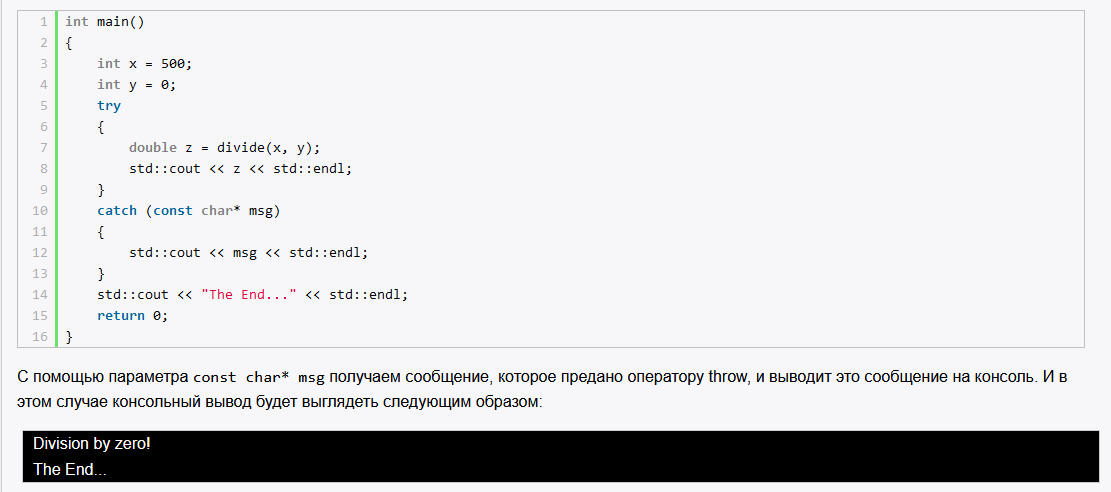
В блок после ключевого слова try помещается код, который потенциально может сгенерировать исключение.

После ключевого слова catch в скобках идет параметр, который передает информацию об исключении. Затем в блоке производится собственно обработка исключения.

Так изменим весь код следующим образом:

****

Код, который потенциально может сгенерировать исключение - вызов функции divide помещается в блок try. В блоке catch идет обработка исключения. Причем многоточие в скобках после оператора catch (catch(...)) позволяет обработать любое исключение. В итоге когда выполнение программы дойдет до строки double z = divide(x, y);, будет сгенерировано исключение, поэтому последующие инструкции из блока try выполняться не будут, а управление перейдет в блок catch, в котором на консоль просто выводится сообщение об ошибке. После выполнения блока catch программа аварийно не завершится, а продолжит свою работу, выполняя операторы после бллока catch:

**** ****

## 35. **Препроцессор: определение, назначение, применение, директивы, выражения, макросы,** **директивы условной компиляции. Примеры на C++.**

Препроцессор С++:  
директивы: #include, #define, #if, #else, #elif, #endif, #ifdef, #ifndef, #error, #line,  
#pragma, #undef; операторы: defined, #, ##.  
**Препроцессор C/С++:** текстовый процессор, который обрабатывает текст исходного файла на первой фазе компиляции.

**Инструкции,** регламентирующие работу компилятора, называются директивами препроцессора.  
**Назначение.**  
директивы препроцессора могут:  
− заменить какие-то лексемы в исходном тексте;  
− вставить содержимое других файлов в указанном месте;  
− подавить компиляцию части файла.  
Директивы препроцессора могут появляться в произвольном месте исходного  
текста, при этом они будут воздействовать только на оставшуюся часть исходного файла.  
**Препроцессор** может быть отдельной программой, или же интегрирован в  
компилятор.  
Входные и выходные данные для препроцессора имеют текстовый формат.  
Препроцессор преобразует текст в соответствии с директивами препроцессора.  
В случае если текст не содержит директив препроцессора, то он остаётся без  
изменений.

Препроцессор можно вызвать отдельно для обработки текста программы без ее компиляции.  
Основные директивы препроцессора:  
#include – вставляет текст из указанного файла;  
#define – задаёт макроопределение (макрос) или символическую  
константу;  
#undef – отменяет предыдущее определение;  
#if – осуществляет условную компиляцию при истинности  
константного выражения;  
#ifdef – осуществляет условную компиляцию при определённости  
символической константы;  
#ifndef – осуществляет условную компиляцию при неопределённости  
символической константы;  
#else – ветка условной компиляции при ложности выражения;  
#elif – ветка условной компиляции, образуемая слиянием else и if;  
#endif – конец ветки условной компиляции;  
#line – препроцессор изменяет номер текущей строки и имя  
компилируемого файла;  
#error – выдача диагностического сообщения;  
#pragma – действие, зависящее от конкретной реализации