Тихонов Дмитрий ВМК-22

Конспекты по ООП

Оглавление

[Конспект: Работа с аргументами командной строки в программах на языке C++ 2](#_Toc147517020)

[Конспект: Тернарный оператор 5](#_Toc147517021)

[Конспект: Оператор swicth case 6](#_Toc147517022)

[Конспект: Пространства имён 8](#_Toc147517023)

[Конспект: Значения аргументов функции по умолчанию 13](#_Toc147517024)

[Конспект: Эллипс 15](#_Toc147517025)

[Конспект: Устройство памяти программы. Где хранятся: глобальные переменные, локальные переменные, аргументы функций, динамические переменные 17](#_Toc147517026)

[Конспект: Обработка исключительных ситуаций 19](#_Toc147517027)

# Конспект: Работа с аргументами командной строки в программах на языке C++

Почему мы должны использовать аргументы командной строки?

Аргументы командной строки очень полезны, если вы хотите передать любые входные строки в свою основную программу из командной строки.

Эти аргументы передаются в качестве параметров функции main(). Рассмотрим, как их можно эффективно использовать.

Часто нам очень удобно напрямую вводить данные в нашу программу. Одним из распространенных способов является использование scanf() или getchar() и т.п. для ожидания ввода данных пользователем.

Но эти вызовы тратят много времени на ожидание и требуют, чтобы пользователь вводил данные вручную.

Мы можем сэкономить много времени, просто передав эти данные в нашу основную программу.

Формат будет чем-то вроде:

./<Исполняемый файл> <Аргумент1> <Аргумент2>

Программа автоматически сохранит эти аргументы командной строки в специальных переменных, из которых мы можем получить к ним прямой доступ.

Для этого потребуется только один раз ввести их при запуске программы. Давайте посмотрим, как мы можем использовать их в таком случае.

**Специальные переменные**

Программа передаст аргументы командной строки в функцию main().

Реклама

В С/С++ функция main() принимает два дополнительных параметра для этих аргументов.

* argc -> Количество аргументов (Argument count). Дает количество аргументов, которые мы передаем (включая имя программы).
* argv -> Argument vector. Это массив строк char\*. Это сами значения аргументов.

Таким образом, argv[0] - это название самой программы, а argv[1] … argv[argc-1] - это все наши аргументы командной строки.

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

Чтобы увидеть это в действии, давайте рассмотрим пример.

**Пример использования**

Давайте рассмотрим программу, которая объединяет две строки, заданные в качестве входных данных.

Мы передадим в нашу программу два аргумента командной строки, поэтому общее argc должно быть 3 (включая имя программы).

Мы можем написать нашу программу так:

#include <iostream>

#include <string>

using namespace std;

string concat\_strings(string s1, string s2) {

return s1 + s2;

}

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

{

setlocale(LC\_ALL,"Russian");

cout << "Вы ввели " << argc

<< " аргумента:" << "\n";

if (argc != 3) {

cerr << "Программа имеет вид: " << argv[0] << " <Арг1> <Арг2>\n";

return 1;

}

string result = concat\_strings(argv[1], argv[2]);

cout << "Результат: " << result << endl;

return 0;

}

Если исполняемое имя **example.exe**, на компьютере с Windows запускаем исполняемый файл с помощью этой команды:В

./example.exe Введены \_строки

Обратите внимание, что аргументы разделены пробелом. Таким образом у нас аргументы командной строки: «Введены» и «\_строки».

На выходе получаем:

Вы ввели 3 аргумента:

Результат: Введены\_строки

Отлично! Похоже, что всё работает как и ожидалось, так как первый аргумент - название самой программы.

Попробуем теперь запустить программу с 4 аргументами.

.\example.exe Введены три \_строки

На выходе получаем:

Вы ввели 4 аргумента:

Программа имеет вид: C:\Users\Дмитрий\Desktop\ОПП\example.exe <Арг1> <Арг2>

Действительно, программа выдает нам правильное сообщение об ошибке!

# Конспект: Тернарный оператор

Используется он обычно в тех случаях, если условие и код, который надо выполнить, в результате проверки условия, очень простые. К примеру, спросить у пользователя хочет он продолжить работать в программе или же хочет выйти из неё.

Синтаксис такой:

Условие ? команда\_1 : команды\_2;

Вначале надо  записать необходимое нам условие и за ним поставить знак вопроса **?** .  Далее, в этой же строке, после знака вопроса пишем первую простую команду (код), которая будет выполняться, если условие вернет истину (**true**). После этой команды ставим двоеточие  **:** и пишем вторую команду (код). Эта вторая команда после двоеточия, выполнится только в том случае, если условие возвращает ложь (**false**).

Пример:

С помощью тернарного оператора, можно определить минимальное и максимальное число из двух значений, которые введет пользователь.

#include <iostream>

using namespace std;

int main()

{

setlocale(LC\_ALL, "Russian");

int firstDigit = 0;

int secondDigit = 0;

int max = 0;

int min = 0;

cout << "Введите первое число:";

cin >> firstDigit;

cout << "Введите второе число:";

cin >> secondDigit;

// используя тернарный оператор, определяем максимум

// и сразу записываем его в переменную max

max = (firstDigit > secondDigit) ? firstDigit : secondDigit;

// так же определяем и записываем min

min = (firstDigit < secondDigit) ? firstDigit : secondDigit;

cout << "Максимум = " << max << endl;

cout << "Минимум = " << min<< endl;

return 0;

}

# Конспект: Оператор swicth case

Очень часто в процессе написания программы требуется писать длинные if-else конструкции, например, когда мы получаем какой-либо ключ от пользователя; если вы пишете игру, то придется проверять на какую кнопку нажал игрок (вправо, влево, пробел и т.д.).

Конструкция switch-case — это удобная замена длинной if-else конструкции, которая сравнивает переменную с несколькими константными значениями, например int или char.

### Синтаксис

|  |
| --- |
| switch ( <переменная> ) {  case значение1:    Выполнить если <переменная> == значение1    break;  case значение2:    Выполнить если <переменная> == значение2    break;  ...  default:    выполнить, если ни один вариант не подошел    break;  } |

Переменная в скобках сравнивается со значениями, описанными после ключевого слова case. После двоеточия находится код, который будет выполнен в случае если переменная оказалась равной текущему значению. break необходим для того, чтобы прервать выполнение switch. Рассмотрим пример, где нет break:

|  |
| --- |
| int a=1;  switch(a)  {      case 1:          a++;      case 2:          a++;      case 3:          a++;  }  cout<<"a= "<<a; |

Данная программа выведет a = 4.

Значения для сравнения, описанные после case, могут быть только константами, поэтому следующий вариант использования switch-case — неверен:

|  |
| --- |
| int a = 10;  int b = 10;  int c = 20;  switch ( a ) {  case b:    // Code    break;  case c:    // Code    break;  default:    // Code    break;  } |

При попытке скомпилировать данную программу, вы получите подобное сообщение:

|  |
| --- |
| test.cpp:9: error: 'b' cannot appear in a constant-expression |

Блок default — необязателен, но он полезен для обработки исключительных ситуации.

Следующая программа демонстрирует один из возможных вариантов использования switch-case:

|  |
| --- |
| #include <iostream>  using namespace std;  void playgame()  {      cout << "Play game called";  }  void loadgame()  {      cout << "Load game called";  }  void playmultiplayer()  {      cout << "Play multiplayer game called";  }  int main()  {    int input;    cout<<"1. Play game\n";    cout<<"2. Load game\n";    cout<<"3. Play multiplayer\n";    cout<<"4. Exit\n";    cout<<"Selection: ";    cin>> input;    switch ( input ) {    case 1:      playgame();      break;    case 2:      loadgame();      break;    case 3:      playmultiplayer();      break;    case 4:      cout<<"Thank you for playing!\n";      break;    default:      cout<<"Error, bad input, quitting\n";      break;    }    cin.get();  } |

Эта программа показывает простой способ обработки вводимых пользователем данных.

Минус данной программы в том, что она дает только одну попытку, без права на ошибку. Это легко исправить, заключив весь блок switch-case в цикл. И break внутри switch-case прервёт только выполнение блока switch, но не прервет цикл

# Конспект: Пространства имён

**Пространство имен** позволяет сгруппировать функционал в отдельные контейнеры. Пространство имен представляет блок кода, который содержит набор компонентов (функций, классов и т.д.) и имеет некоторое имя, которое прикрепляется к каждому компоненту из этого пространства имен. Полное имя каждого компонента — это имя пространства имен, за которым следует оператор **::** (оператор области видимости или scope operator) и имя компонента. Примером может служить оператор cout, который предназначен для вывода строки на консоль и который определен в пространстве имен **std**. Соответственно чтобы обратиться к этому оператору, применяется выражение std::cout.

### Глобальное пространство имен

Если пространство имен не указано, то по умолчанию применяется **глобальное пространство имен**. применяется по умолчанию, если пространство имен не было определено. Все имена в глобальном пространстве имен такие же, как вы их объявляете, без прикрепления имени пространства имен. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | #include <iostream>    void print(const std::string&);    const std::string message{"hello"};  int main()  {      print(message);  }    void print(const std::string& text)  {      std::cout << text << std::endl;  } |

Здесь определены функции print и main и константа message и не используется никакого пространства имен. Поэтому фактически функции print и main и константа message определены в глобальном пространстве имен. В принципе для обращения к ним также можно использовать оператор **::**, только без названия пространства имен, хотя это и избыточно:

|  |  |
| --- | --- |
| 1  2  3  4 | int main()  {      ::print(::message);  } |

Функция main должна быть определена в глобальном пространстве имен.

### Определение пространства имен

Для определения пространства имен применяется ключевое слово **namespace**, за которым идет название имени пространства имен:

|  |  |
| --- | --- |
| 1  2  3  4 | namespace имя\_пространства\_имен  {      // код пространства имен  } |

После имени пространства имен идет блок кода, в который собственно помещаются компоненты пространства имен - функции, классы и т.д.

Например, определим пространство имен, которое назовем hello:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | #include <iostream>    namespace hello  {      const std::string message{"hello work"};      void print(const std::string& text)      {          std::cout << text << std::endl;      }  }    int main()  {      hello::print(hello::message);   // hello work  } |

Здесь в пространстве hello определена функция print и константа message. И чтобы обратиться к этим компонентам вне пространства имен hello, надо использовать его имя:

|  |  |
| --- | --- |
| 1 | hello::print(hello::message); |

Внутри пространства имен к его компонентам можно обращаться без имени пространства имен:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | namespace hello  {      const std::string message{"hello work"};      void print(const std::string& text)      {          std::cout << text << std::endl;      }      void print\_default()      {          std::cout << message << std::endl;      }  } |

### Вложенные пространства имен

Одно пространство имен может содержать другие пространства:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | #include <iostream>    namespace console  {      namespace messages      {          const std::string hello{"hello"};          const std::string welcome{"Welcome"};          const std::string goodbye{"Good bye"};      }      void print(const std::string& text)      {          std::cout << text << std::endl;      }      void print\_default()      {          std::cout << messages::hello << std::endl;      }  }    int main()  {      console::print(console::messages::hello);  } |

Здесь в пространстве console определено вложенное пространство messages, которое содержит ряд констант. Для обращения к компонентам вложенного пространства имен также надо использовать его имя:

|  |  |
| --- | --- |
| 1  2  3  4 | void print\_default()  {      std::cout << messages::hello << std::endl;  } |

Вне пространства имен console для обращения к подобным констанстам надо указывать всю цепь пространств имен:

int main()

{

    console::print(console::messages::hello); // console { messages { hello}}

}

### Директива using

Директива **using** позволяет ссылаться на любой компонент пространства имен без использования его имени:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | #include <iostream>    namespace console  {      const std::string message{"hello"};      void print(const std::string& text)      {          std::cout << text << std::endl;      }  }    using namespace console;    // подключаем все компоненты пространства console    int main()  {      print(message); // указывать пространство имен не требуется  } |

Здесь подключаем все компоненты пространства имен console в глобальное пространство имен. И после этого указывать имя этого пространства для обращения к его компонентам не требуется:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | using namespace console;    // подключаем все компоненты пространства console    int main()  {      print(message); // указывать пространство имен не требуется  } |

Однако такое подключение может привести к нежелательным последствиям, если в глобальном пространстве имен определяются компоненты с теми же именами (например, переменная message). В этом случае мы можем подключить только отдельные компоненты:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | #include <iostream>    namespace console  {      const std::string message{"hello"};      void print(const std::string& text)      {          std::cout << text << std::endl;      }  }    using console::print;    // подключаем только функцию print    int main()  {      print(console::message); // надо указывать пространство для message  } |

Здесь подключаем только функцию print:

|  |
| --- |
| using console::print; |

Поэтому для обращения к ней не надо указывать имя пространства имен. А для обращения к любым другим компонентам надо.

### Псевдонимы пространств

Если название пространства длинное, то для него можно определить псевдоним:

|  |
| --- |
| namespace псевдоним = название пространства имен; |

Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | #include <iostream>    namespace console  {      namespace messages      {          const std::string message{"hello"};      }      void print(const std::string& text)      {          std::cout << text << std::endl;      }  }    namespace mes = console::messages;    int main()  {      console::print(mes::message); // обращаемся к message через псевдоним mes  } |

В данном случае для пространства console::messages устанавливается псевдоним mes.

# Конспект: Значения аргументов функции по умолчанию

Аргумент по умолчанию – это такой аргумент функции, который программист может не указывать при вызове функции. Аргумент по умолчанию добавляется компилятором автоматически.

Чтобы использовать аргументы по умолчанию в функции, эта функция должна быть соответствующим образом объявлена. Аргументы по умолчанию объявляются в прототипе функции.

**Общая форма** объявления функции, которая содержит аргументы по умолчанию:

**returned\_type** FunName(**type1** v1 = val1, **type2** v2 = val2, ..., **typeN** vN = valN)

{

// ...

}

где

* ***returned*\_*type***– тип возвращаемый функцией;
* ***FunName***– имя функции;
* ***type1***, ***type2***, …, ***typeN***– типы переменных ***v1*, *v2*, …, *vN*;**
* ***val2***, …, ***valN***– значения, которое присваиваются по умолчанию переменным ***v2***, …, ***vN***. В этом случае типы и присваиваемые значения должны быть совместимыми.

**Пример:** Объявляется функция Inc(),получающая два параметра. Функция возвращает сумму значений первого и второго параметров. Второй параметр функции есть параметром (аргументом) по умолчанию, которому присваивается значение 1.

Код:

#include "stdafx.h"

#include <iostream>

using namespace std;

// функция, содержащая аргумент по умолчанию

int Inc(int value, int step = 1)

{

return value + step;

}

int \_tmain(int argc, \_TCHAR\* argv[])

{

//

int v1, v2;

v1 = 5;

// использование аргумента по умолчанию

v2 = Inc(v1); // v2 = 5 + 1 = 6

v1 = 5;

v2 = Inc(v1, 3); // v2 = 5 + 3 = 8

cout << v2 << endl;

return 0;

}

Использование аргументов по умолчанию дает следующие преимущества:

* сокращается листинг программного кода за счет избежания написания лишних функций, которые выполняют ту же работу только с другими значениями аргументов;
* обеспечивается простой, естественный и эффективный стиль программирования;
* в некоторых случаях аргументы по умолчанию есть сокращенной формой перегрузки функции. Это, в свою очередь улучшает читабельность программного кода и упрощает вызов функции.

Во время объявления функции, аргументы по умолчанию задаются только один раз. Здесь возможны два случая. Первый, если функция имеет прототип и реализацию. Второй случай, если функция не имеет прототипа, а имеет только реализацию.

В первом случае (с прототипом), аргумент по умолчанию задается только в прототипе функции (а не в реализации). Во втором случае, аргумент по умолчанию задается в реализации функции, поскольку нет прототипа функции.

**Пример.** Заданы две функции:

* функция GetAreaCircle(), которая вычисляет площадь круга. Эта функция имеет прототип;
* функция GetVolumeSphere(), которая вычисляет объем шара. Эта функция не имеет прототипа.

В приведенном ниже коде демонстрируется применение функций, которые содержат аргументы по умолчанию. Важно: аргументы по умолчанию объявляются только один раз. В функции GetAreaCircle() аргументы по умолчанию объявляются в прототипе, поскольку эта функция имеет прототип. В функции GetVolumeSphere() аргументы по умолчанию объявляются в реализации, так как функция не имеет прототипа:

#include "stdafx.h"

#include <iostream>

using namespace std;

const double Pi = 3.1415;

// объявление аргументов по умолчанию в прототипе функции

double GetAreaCircle(double r = 1);

// функция GetVolumeSphere() не имеет прототипа,

// поэтому аргумент по умолчанию задается непосредственно в реализации

double GetVolumeSphere(double r = 1)

{

return 4.0/3 \* Pi\*r\*r\*r;

}

int \_tmain(int argc, \_TCHAR\* argv[])

{

double area;

double volume;

area = GetAreaCircle(); // area = 3.1415

volume = GetVolumeSphere(); // volume = 4.18867

area = GetAreaCircle(2); // area = 12.566

volume = GetVolumeSphere(2); // volume = 33.5093

cout << area << endl;

cout << volume << endl;

return 0;

}

// реализация функции - аргументы по умолчанию уже заданы в прототипе

double GetAreaCircle(double r)

{

return Pi \* r \* r;

}

**ВАЖНО**

Аргументы (параметры) по умолчанию должны следовать после объявления обычных аргументов. Если объявить аргументы по умолчанию первыми, а за ними объявить обычные аргументы, то это будет ошибка.

# Конспект: Эллипс

Язык C (и, по наследству, C++) допускает определение функций, принимающих переменное количество параметров. Это довольно низкоуровневый механизм, и не существует стандартного способа определить, сколько параметров было помещено в стек вызовов вызывающей функцией или какого они типа, поэтому обычно эта информация в некоторой форме передаётся первым параметром. Примерами таких функций являются стандартные функции printf и scanf. Общий синтаксис заголовка функции с переменным числом параметров имеет следующий вид:

Тип-результатаИмя-функции*(*Обязательные-параметры*, ...)*

Обязательные параметры объявляются так же, как в обычных функциях. Формально C++ (но не C) позволяет объявлять функции с переменным числом параметров, которые не принимают обязательных параметров, но определить такую функцию в рамках стандарта языка не получится.

При вызове функции с переменным числом параметров необходимо указать значения всех обязательных параметров и далее можно указать произвольное (от нуля до некоторого максимума, зафиксированного компилятором) количество дополнительных параметров, которые могут иметь разный тип. При этом при вызове значения типа float автоматически приводятся к double, значения целочисленных типов меньшей ширины, чем int (например, bool, char, short), приводятся к int.

Для извлечения дополнительных параметров из стека вызовов стандартом предусмотрен следующий набор определений, размещённый в заголовочном файле [cstdarg](http://en.cppreference.com/w/cpp/header/cstdarg):

* **va\_list** — тип данных, описывающий состояние стека;
* **va\_start**(args, last) — макрос, инициализирующий переменную args типа va\_list, last — имя последнего из обязательных параметров. Данный макрос можно применить лишь однажды за вызов функции (иначе [UB](https://teccxx.neocities.org/mx1/ub.html)).
* **va\_copy**(args1, args2) [C99, C++11] — макрос, выполняющий копирование args2 в args1 (оба параметра — переменные типа типа va\_list). Копирование позволяет повторно пройти по параметрам.
* **va\_arg**(args, T) — макрос, извлекающий следующий (в соответствии с состоянием, хранящимся в переменной args) параметр типа T из стека вызовов.
* **va\_end**(args) — макрос, корректно завершающий извлечение параметров из стека. Этот макрос должен быть применён к каждой переменной типа va\_list, которая до того была инициализирована с помощью va\_start или va\_copy (иначе UB).

Пример реализации функции с переменным числом параметров типа double:

*///* Длина вектора с количеством компонент comps.

*///* Компоненты вектора передаются параметрами функции.

double vec\_len(int comps, ...)

{

std::va\_list args;

va\_start(args, comps);

double s = 0.0;

**while** (comps-- > 0)

{

const double arg = va\_arg(args, double);

s += arg \* arg;

}

va\_end(args);

**return** std::sqrt(s);

}

# Конспект: Устройство памяти программы. Где хранятся: глобальные переменные, локальные переменные, аргументы функций, динамические переменные

Работа вычислителя.

Внутри вычислителя есть память, в которой находится программа. Каждая следующая вызванная функция/программа обычно расположена в памяти с большим адресом, и является первой на освобождение (очищение).  
Память под программу рассчитывается на этапе КОМПИЛИРОВАНИЯ и ЛИНКОВКИ и выделяется в процессе выполнения программы.

**Следовательно**

* внутри оперативной памяти идет разбиение (фрагментация) на зоны для каждой программы.
* при вызове дополнительных программ или функций также выделяется место в памяти
* при выходе из функции/ программы место очищается.

Скомпилированная программа имеет тот объем памяти, что ему задали  
Данная память, рассчитанная на этапе компиляции/линковки и выделенная в ходе запуска программы называется **стеком (stack)**.  
Следует еще раз акцентировать внимание, что программы запускаются по принципу LIFO (Last In, First Out),  то есть последний добавленный в стек кусок памяти будет первым в очереди на вывод из стека.  
Так как мы не знаем сколько нам ещё понадобится памяти, ведь она уже выделена в стеке  
Для этого можно использовать остальную память – она называется **куча (heap)**. В ходе работы программы мы можем **динамически** (по своему хотению, в любой момент времени, при наличии свободного места) выделять дополнительную память и работать через ее адреса.  
Эта дополнительная память, выделенная в куче и с которой работают из стека называется **динамической памятью.**   
**Примечание:** На самом деле **не всегда** стек и куча расположены в одной зоне памяти, физически для более быстрого выполнения стек может быть перенесен в область **кэш (cash)** процессора (эта память обычно существенно быстрее оперативной), но за счет адресации и работы ide данные.  
   
Важно отметить, что разработчик сам выделяет память в куче, чем может ограничить работу других программ, в C++ компилятор не отслеживает выделение памяти пользователем и указатели/ссылки на нее, поэтому если указатель будет утерян (удален), то область в куче так и останется выделенной до перезапуска программы/ перезапуска вычислителя.  
Данный механизм потери памяти из-за потери указателя называется утечками leaks.  
Если утечек будет много — память закончится и код не сможет выполняться.

Память, которую использует программа делится на три вида:

1. Статическая память (static memory)
   * хранит глобальные переменные и константы;
   * размер определяется при компиляции.
2. Стек (stack)
   * хранит локальные переменные, аргументы функций и промежуточные значения вычислений;
   * размер определяется при запуске программы (обычно выделяется 4 Мб).
3. Куча (heap)
   * динамически распределяемая память;
   * ОС выделяет память по частям (по мере необходимости).

Динамически распределяемую память следует использовать в случае если мы заранее (на момент написания программы) не знаем сколько памяти нам понадобится (например, размер массива зависит от того, что введет пользователь во время работы программы) и при работе с большими объемами данных (например, массив из 1 000 000 int`ов не поместится на стеке).

**Глобальная переменная** — переменная, определённая вне функций и доступная из разных функций.

**Локальная переменная** — переменная, определённая внутри некоторой функции и доступная только из неё.

**Аргументы функций** — значения, передаваемые параметрам функции при ее вызове, называются аргументами.

**Динамическая переменная** — переменная в программе, место в оперативной памяти под которую выделяется во время выполнения программы.

# Конспект: Обработка исключительных ситуаций

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. Исключение представлякт временный объект любого типа, который используется для сигнализации об ошибке. Цель объекта-исключения состоит в том, чтобы передать информацию из точки, в которой произошла ошибка, в код, который должен ее обработать. Если исключение не обработано, то при его возникновении программа прекращает свою работу.

Например, в следующей программе происходит деление чисел:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | #include <iostream>    double divide(int a, int b)  {      return a / b;  }    int main()  {      int x{500};      int y{};      double z {divide(x, y)};        std::cout << z << std::endl;      std::cout << "The End..." << std::endl;  } |

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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | double divide(int a, int b)  {      if (b)          return a / b;      else          std::cout << "Error! b must not be equal to 0" << std::endl;  } |

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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | double divide(int a, int b)  {      if (b)          return a / b;      throw "Division by zero!";  } |

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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | try  {      инструкции, которые могут вызвать исключение  }  catch(объявление\_исключения)  {      обработка исключения  } |

В блок кода 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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | #include <iostream>  double divide(int a, int b)  {      if (b)          return a / b;      throw "Division by zero!";  }  int main()  {      int x{500};      int y{};      try      {          double z {divide(x, y)};          std::cout << z << std::endl;      }      catch (...)      {          std::cout << "Error!" << std::endl;      }      std::cout << "The End..." << std::endl;  } |

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)};**

При выполнении этой строки 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 блока catch:

Error!

The End...

Однако в данном случае мы только знаем, что произошла какая-то ошибка, а какая именно, неизвестно. Поэтому через параметр в блоке catch мы можем получить то сообщение, которое передается оператору throw:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | #include >iostream<    double divide(int a, int b)  {      if (b)          return a / b;      throw "Division by zero!";  }    int main()  {      int x{500};      int y{};        try      {          double z {divide(x, y)};          std::cout << z << std::endl;      }      catch (const char\* error\_message)      {          std::cout << error\_message << std::endl;      }      std::cout << "The End..." << std::endl;  } |

С помощью параметра const char\* error\_message получаем сообщение, которое предано оператору throw, и выводим это сообщение на консоль. Почему здесь мы получаем сообщение об ошибке в виде типа const char\*? Потому что после оператора throw идет строковый литерал, который представляет как раз тип const char\*. И в этом случае консольный вывод будет выглядеть следующим образом:

Division by zero!

The End...

Таким образом, мы можем узнать суть возникшего исключения. Подобным образом мы можем передавать информацию об исключении через любые типы, например, std::string:

|  |  |
| --- | --- |
| 1 | throw std::string{"Division by zero!!"}; |

Тогда в блоке **catch** мы можем получить эту информацию в виде объекта std::string:

|  |  |
| --- | --- |
| 1  2  3  4 | catch (std::string error\_message)  {      std::cout << error\_message << std::endl;  } |

Если же исключение не обработано, то вызывается функция **std::terminate()** (из модуля <exception> стандартной библиотеки C++), которая, в свою очередь, по умолчанию вызывает другую функцию - **std::abort()** (из <cstdlib>), которая собственно и завершает программу.

Существует очень много функций и в стандартной библиотеке С++, и в каких-то сторонних библиотеках. И может возникнуть вопрос, какие из них вызывать в конструкции try-catch, чтобы не столкнуться с необработанным исключением и аварийным завершением программы. В этом случае может помочь прежде всего документация по функции (при ее наличии). Другой сигнал - ключевое слово **noexcept**, которое при использовании в заголовке функции указывает, что эта функция никогда не будет генерировать исключения. Например:

|  |  |
| --- | --- |
| 1 | void print(int argument) noexcept; |

Здесь указываем, что функция print() никогда не вызовет исключение. Таким образом, встретив функцию с подобным ключевым словом, можно ожидать, что она не вызовет исключения. И соответственно нет необходимости помещать ее вызов в конструкцию try-catch.