ООП 2021-22 весна
Лекция 1

Препроцессор (макросы). Обработка исключений. <exception>

oopCpp@yandex.ru

Препроцессор

Страуструп пишет:

"С++ создавался с целью избавить автора и его друзей от необходимости программировать на ассемблере, С или других современных языках такого уровня.

Главной задачей было придумать язык, на котором удобно писать хорошие программы и с которым программисту **приятно** работать.

С++ никогда не проектировался на бумаге. Его проектирование, документирование и реализация выполнялись одновременно".

Реализация языка C++ включает **препроцессор** с возможностями макроподстановки,
условной трансляции
включения указанных файлов.

Для передачи заданий препроцессору служат строки, начинающиеся с символа **#** (перед ним могут идти пробелы и символы горизонтальной табуляции). Такие строки называются командами, и их синтаксис определяется независимо от остального языка. Команды могут находиться в любом месте программы, и их действие продолжается (независимо от правил областей видимости C++) до конца данной единицы трансляции (файла).

Команду препроцессора, как и любую строку, можно продолжить на следующей строке входного текста, поместив символ обратной дробной черты (**backslash character**) непосредственно перед символом конца продолжаемой строки. Препроцессор до того, как входная строка будет разбита на лексемы, удаляет символы обратной дробной черты и конца строки. Символ обратной дробной черты не должен быть последним символом входного файла.

Фазы препроцессорной обработки

По определению существует несколько фаз препроцессорной обработки. В конкретной реализации фазы могут сливаться, но результат все равно должен быть таким, как будто были выполнены все фазы.

- замена необходимых символов
- удаление определенных символов (например: обратная дробная черта, следующий за ней символ конца строки)
- разбитие входного текста на лексемы препроцессора и последовательность пробелов
- выполнение команд препроцессора, и макроподстановки
- в символьных константах и строках литералов комбинации специальных символов заменяются на свои эквиваленты
- сливаются соседние строки литералов
- результат препроцессорной обработки подвергается синтаксическому и семантическому анализу

Макроопределение и макроподстановка

Команда вида

#define идентификатор строка-лексем

называется макроопределением. Она указывает препроцессору, что надо произвести замену всех последующих вхождений идентификатора на заданную последовательность лексем, называемую строкой замены. Обобщенные пробелы, окружающие эту последовательность лексем, отбрасываются. Например, при определении

```
#define VALUE  8

char array [VALUE ] [VALUE ];
```

после макроподстановки примет вид

```
char array [8][8];
```

Определенный таким способом идентификатор можно переопределить с помощью другой команды #define, но при условии, что строки замены в обоих определениях совпадают.

Команда вида

идентификатор (идентификатор , … , идентификатор) строка-лексем

называется макроопределением с параметрами или "функциональным" макроопределением. В нем недопустимы пробелы между первым идентификатором и символом (. Определенный таким способом идентификатор можно переопределить с помощью другого функционального макроопределения, но при условии, что во втором определении то же число и те же наименования параметров, что и в первом, а обе строки замены совпадают.

Последующие вхождения идентификатора, определенного в функциональном макроопределении, если за ним следуют символ (, последовательность лексем, разделенных запятыми, и символ), заменяются на строку лексем

Пусть есть 2 макроопределения

```
#define index_mask 0XFF00
#define extract( word,mask)   word & mask
```

Тогда макровызов

```
index = extract( packed_data,index_mask);
```

после подстановки примет вид

```
index = packed_data & 0XFF00;
```

Для обоих видов макроопределений строка замены проверяется на наличие других макроопределений

Операция

Если непосредственно перед параметром в строке замены идет лексема #, то при подстановке параметр и операция # будут заменены на строку литералов, содержащую имя соответствующего параметра макровызова. В символьной константе или строке литералов, входящих в параметр, перед каждым вхождением \ или " вставляется символ \.

Например, если есть макроопределения

```
#define  path(logid,cmd) "/usr/" #logid "/bin/" #cmd
```

то макровызов

```
char* mytool=path(joe,readmail);
```

приведет к такому результату:

```
char* mytool="/usr/" "joe" "/bin/" "readmail";
```

После конкатенации соседних строк:

```
char* mytool="/usr/joe/bin/readmail";
```

Операция

Если в строке замены между двумя лексемами, одна из которых представляет параметр макроопределения, появляется операция **##,** то сама операция **##** и окружающие ее обобщенные пробелы удаляются. Таким образом, результат операции ## состоит в конкатенации.

Пример:

```
#define inherit(basenum)  public  Pubbase ## basenum, \
 private Privbase ## basenum
```

тогда

```
class D: inherit(1) { };
```

приведет к такому результату:

```
class D: public Pubbase1, Privbase1 { };
```

Макроопределение, которое в строке замены соседствует с ##, не подлежит подстановке, однако, результат конкатенации может использоваться для подстановки.

Пример:

```
#define concat(a)   a ## ball
#define base B
#define baseball sport
```

Тогда макровызов

concat(base)

даст в результате

sport

а вовсе не

baseball

Область видимости макроимен и конструкция #undef

После появления макроопределения идентификатор из него считается определенным и остается в текущей области видимости (независимо от правил областей видимости в C++) до конца единицы трансляции или пока его определение не будет отменено с помощью команды **#undef**.

Команда #undef имеет вид:

#undef идентификатор

Она заставляет препроцессор "забыть" макроопределение с этим идентификатором. Если указанный идентификатор не является определенным в данный момент макроименем, то команда #undef игнорируется.

Включение файлов

Управляющая строка вида:

#include <имяфайла>

приводит к замене данной строки на содержимое файла с указанным именем. Поиск указанного файла проходит в определенной последовательности частей архива системы и определяется реализацией.

Аналогично, управляющая строка вида:

#include "имяфайла"

приводит к замене данной строки на содержимое файла с указанным именем. Поиск этого файла начинается в особых (системных) частях архива, указанных в начале последовательности поиска. Если там он не найден, то поиск файла идет по всей последовательности, как если бы управляющая строка имела вид:

#include <имяфайла>

В имени файла, ограниченном символами < и > нельзя использовать символы конца строки или >. Если в таком имени появится один из символов ', \, или ", а также последовательность символов /* или //, то результат считается неопределенным.

В имени файла, ограниченном парой символов " нельзя использовать символы конца строки или ", хотя символ > допустим. Если в таком имени появится символ ' или \ или последовательность /* или //, то результат считается неопределенным.

Управление строками и команда error

Для удобства написания программ, порождающих текст на С++, введена управляющая строка вида:

#line константа "имяфайла" opt

Она задает значение предопределенному макроимени __LINE__, которое используется в диагностических сообщениях или при символической отладке; а именно: номер следующей строки входного текста считается равным заданной константе, которая должна быть десятичным целым числом. Если задано "имяфайла", то значение макроимени __FILE__ становится равным имени указанного файла. Если оно не задано, __FILE__ не меняет своего значения.

Макроопределения в этой управляющей строке раскрываются до выполнения самой команды.

Строка вида:

#error строка-лексем

заставляет реализацию выдать диагностическое сообщение, состоящее из заданной последовательности лексем препроцессора.

Обработка ошибок

Совершение ошибки

Программисты часто допускают ошибки, которые приводят к ненормальным состояниям, называемым ошибками. В целом, эти ошибки бывают трех типов: 1) синтаксические ошибки, 2) логические ошибки и 3) ошибки времени выполнения.

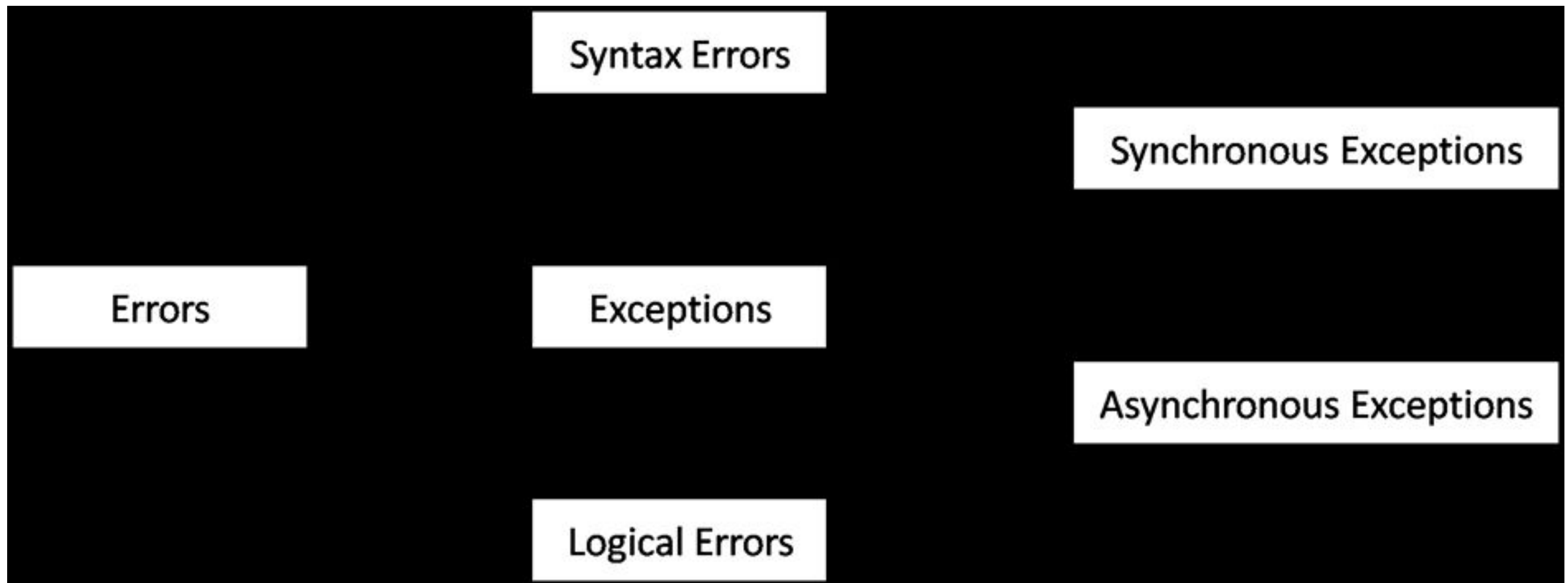
Синтаксические ошибки являются наиболее частым типом ошибок. Например, если мы забываем поставить точку с запятой, это синтаксическая ошибка.

Логические ошибки возникают, когда программист выполняет ошибку в логике программы. Например, в дополнение к двум числам программы, если программисты ставят минус вместо плюса, это является логической ошибкой. Их очень трудно обнаружить.

Последняя категория ошибок - это **ошибки, возникающие при выполнении программы**. Эти ошибки известны как ошибки времени выполнения или **исключения**. Исключения бывают двух типов: 1) синхронные и 2) асинхронные.

Синхронные исключения - это исключения во время выполнения, которые возникают из-за кода, написанного программистом, и они могут быть обработаны программистом. **Асинхронные** исключения - это исключения во время выполнения, возникающие из-за кода вне программы.

Например, если в оперативной памяти нет доступной памяти, это приведет к ошибке **out of memory**, которая является асинхронным исключением. Такие асинхронные исключения не могут быть обработаны. Ошибки в программах можно проиллюстрировать следующим образом:



Обработка синхронных исключений называется **обработкой исключений**. Обработка исключений связана с обнаружением исключений во время выполнения и сообщением об этом пользователю для принятия соответствующих мер.

C++ предоставляет элементы **try**, **catch** и **throw** для обработки исключений.

Блок try содержит код, который может вызывать исключения. Синтаксис блока try выглядит следующим образом:

```
try
{
    //Code
    ...
}
```

Блок catch содержит код для обработки исключений, которые могут возникнуть в блоке try. Синтаксис блока catch выглядит следующим образом:

```
catch (Exception-type)
{
    //Code to handle exception
    ...
}
```

Предложение **throw** используется для создания исключения, которое будет поймано блоком catch. Синтаксис предложения throw выглядит следующим образом:

throw exception;

```
////////////////////////////////
#include<iostream>
using namespace std;
int main()
{
    int a, b;
    cout<<"Enter a and b values: ";
    cin>>> >a>>> >b;
    try    {
     if(b = = 0) // обрабатывает исключение деления на ноль:
            throw b;
     else
            cout<<"Result of a/b = "<<(a/b);
    }
    catch(int b)     {
     cout<<"b cannot be zero";
    }
    return 0;
 }
```

Вывод:

Enter a and b : 10 0

b cannot be zero

Если возникает исключение и нет блока catch для обработки этого исключения, будет выполнена функция terminate (), которая в свою очередь вызывает функция abort () и выполнение программы резко прекращается.

Блок **try** может быть связан с несколькими блоками **catch**. Несколько операторов catch могут следовать за блоком try для обработки нескольких исключений. Первый блок catch, соответствующий типу исключения, будет выполнен, и элемент управления перейдет к следующему оператору после всех доступных блоков catch.

За блоком **try** должен следовать по крайней мере один блок **catch**. Если нет блока catch, соответствующего исключению, то будет выполняться функция terminate ().

Пример

```
#include<iostream>
using namespace std;
int main() {
    int a, b;
    cout<<"Enter a and b values: ";
    cin>>> >a>>> >b;
    try    {
      if(b = = 0)
            throw b;
      else if(b < 0)
            throw "b cannot be negative";
      else
            cout<<"Result of a/b = "<<(a/b);
    }
    catch (int  b)    {
      cout<<"b cannot be zero";
    }
    catch(const char* msg)        {
      cout<<msg;
    }
    return 0;
}
```

Input and output :

First Run:

Enter a and b values: 10 0

b cannot be zero

Second Run:

Enter a and b values: 10 -5

b cannot be negative

Поймать Все Исключения

При написании программ, если программист не знает, какое исключение может вызвать программа, можно использовать блок **catch all**. Он используется, чтобы поймать любой вид исключения.

Синтаксис блока catch all выглядит следующим образом:

```
catch(...)
{
    //Code to handle exception
    ...
}
```

При написании нескольких операторов catch следует позаботиться о том, чтобы блок catch all был записан как последний блок в последовательности блоков catch.

Если он записан первым в последовательности, другие блоки catch никогда не будут выполнены.

Пример

```
#include<iostream>
using namespace std;
int main() {
     int a, b;
     cout<<"Enter a and b values: ";
     cin>>a>>b;
     try    {
       if(b == 0)
             throw b;
       else if( b < 0)
             throw "b cannot be negative";
       else
             cout<<"Result of a/b = "<<(a/b);
     }
     catch (int b)    {
       cout<<"b cannot be zero";
     }
     catch(...)  {
       cout<<"Unkown exception in program";
     }
     return 0;
}
```

Input and output :

Enter a and b: 10-5

Unkown exception in program

Переосмысление исключения

В C++, если функция или вложенный try-блок не хотят обрабатывать исключение, он может повторно создать это исключение для функции или внешнего try-блока, чтобы обработать это исключение. Синтаксис для переосмысления и исключения выглядит следующим образом:

```
throw;
```

Следующий пример демонстрирует переосмысление и исключение из внешнего блока try-catch.

В программе мы увидим, что исключение возникает во внутреннем блоке try.

Блок catch all улавливает исключение и передает его во внешний блок try-catch, где оно будет обработано.

```
#include<iostream>
using namespace std;
int main() {
    int a, b;
    cout<<"Enter a and b values: ";
    cin>>> >a>>> >b;
    try    {
      try         {
            if( b = = 0)
                  throw b;
            else if( b < 0)
                  throw "b cannot be negative";
            else
                  cout<<"Result of a/b = "<<(a/b);
      }
      catch( int b)        {
            cout<<"b cannot be zero";
      }
      catch(...)           {
            throw;
      }
    }
    catch( const char* msg)      {      cout<<msg;   }
     return 0; }
```

Input and output :
Enter a and b values: 10 -2
b cannot be negative

Выбрасывание исключений в определении функции

Функция может объявить, какой тип исключений она может выдавать. Синтаксис для объявления исключений, которые выбрасывает функция, выглядит следующим образом:

```
return-type  function-name ( params-list)  throw ( type1, type2, ...)
{
    //Function body
    ...
}
```

Следующий пример демонстрирует выбрасывание исключений в определении функции.

В программе функция sum () может выдавать исключение типа int. Таким образом, вызывающая функция должна предоставить блок catch для исключения типа int:

```
#include<iostream>
using namespace std;
void sum() throw (int) {
    int a, b;
    cout<<"Enter a and b values: ";
    cin>>> >a>>> >b;
    if(a= =0 |/ b= =0)
     throw 1;
    else
     cout<<"Sum is: "<<(a+b);
}
int main() {
    try    {
     sum();
    }
    catch (int)       {
     cout<<"a or b cannot be zero";
    }
    return 0;
}
```

Input and output :

Enter a and b values: 5 0
a or b cannot be zero

Выбрасывание исключений типа

Вместо того, чтобы выбрасывать исключения из заранее определенных типов, таких как int, float, char и т. д., мы можем создавать классы и выбрасывать эти типы классов в качестве исключений.

Пустые классы особенно полезны при обработке исключений.

Следующая программа демонстрирует выбрасывание типов классов в качестве исключений.

В программе ZeroError-это пустой класс, созданный для обработки исключений:

```
#include<iostream>
using namespace std;
class ZeroError { };
void sum() {
     int a, b;
     cout<<"Enter a and b values: ";
     cin>>a>>b;
     if( a==0 || b==0)
       throw  ZeroError();
     else
       cout<<"Sum is: "<<(a+b);
}
int main() {
     try    {
       sum();
     }
     catch( ZeroError e) { cout<<"a or b cannot be zero";}
     return 0;
}
```

Input and output:
Enter a and b values: 0 8
a or b cannot be zero

Обработка исключений и наследование

При наследовании, выбрасывая исключения производных классов, следует позаботиться о том, чтобы блоки catch с базовым типом **записывались после** блока catch с производным типом. В противном случае блок catch с базовым типом также ловит исключения производных типов классов.

В первом примере, даже если вызванное исключение имеет производный тип, оно ловит его блоком catch базового типа. Чтобы избежать этого, во втором примере, мы должны написать блок catch базового типа после производного.

```
#include<iostream>
using namespace std;
struct Base { };
struct Derived : public Base { };
int main() {
    try
    {
      throw Derived();
    }
    catch(Base b) // Базовый ловит раньше
    {
      cout<<"Base object caught";
    }
    catch(Derived d)
    {
      cout<<"Derived object caught";
    }
    return 0;
}
```

Output :

Base object caught

```
#include<iostream>
using namespace std;
struct Base { };
struct Derived : public Base { };
int main() {
    try
    {
      throw Derived();
    }
    catch(Derived d) // Производный ловит раньше
    {
      cout<<"Derived object caught";
    }
    catch(Base b)
    {
      cout<<"Base object caught";
    }
    return 0;
}
```

Output :

Derived object caught

Исключения в конструкторах и деструкторах

Вполне возможно, что исключения могут возникать в конструкторе или деструкторе. Если в конструкторе возникает исключение, память может быть выделена для некоторых элементов данных и не сможет быть выделена для других. Это может привести к проблеме утечки памяти, поскольку программа останавливается, а память для элементов данных остается не освобожденной в оперативной памяти.

Аналогично, когда в деструкторе возникает исключение, память не сможет быть своевременно освобождена, что снова приведет к проблеме утечки памяти.

Поэтому надо обеспечить обработку исключений внутри конструктора и деструктора, чтобы избежать таких проблем.

```
#include<iostream>
using namespace std;
class Divide
{
    private:
     int *x;
     int *y;
    public:
     Divide()
     {
        x = new int();
        y = new int();
        cout<<"Enter two numbers: ";
        cin>>*x>>*y;
        try
        {
            if(*y == 0)
            {
                throw *x;
            }
        }
```

```
        catch(int)
        {
            delete x;
            delete y;
            cout<<"Second number cannot be zero!"<<endl;
            throw;
        }
    }
    ~Divide()
    {
        try
        {
            delete x;
            delete y;
        }
        catch(...)
        {
            cout<<"Error while deallocating memory"<<endl;
        }
    }
    float division()  {   return (float)*x / *y; }
};
```

```
int main()
{
    try
    {
      Divide d;
      float res = d.division();
      cout<<"Result of division is: "<<res;

    }
    catch(...)
    {
      cout<<"Unkown exception!"<<endl;
    }
    return 0;
}
```

Input and output :

Enter two numbers: 5 0
Second number cannot be zero!
Unkown exception!

Преимущества обработки исключений

Обработка исключений помогает программистам создавать надежные системы. (по материалам из https://www.startertutorials.com/blog)

Обработка исключений отделяет код обработки исключений от основной логики программы.

Исключения можно обрабатывать за пределами обычного кода, вызывая исключения из определения функции или вызывая его повторно.

Функции могут обрабатывать только те исключения, которые они выбирают, т. е. функция может создавать множество исключений, но может выбрать обработку только некоторых из них.

Программа с обработкой исключений не остановится внезапно. Она изящно завершается, выдавая соответствующее сообщение.

////////////////////////////////////

Правда, я лично работаю без исключений.)

std::exception

Базовый класс для стандартных исключений.

Все объекты, создаваемые компонентами стандартной библиотеки, являются производными от этого класса.

Таким образом, все стандартные исключения могут быть перехвачены путем передачи объекта этого типа по ссылке.

```
class exception {
public:
  exception () noexcept;
  exception (const exception&) noexcept;
  exception& operator= (const exception&) noexcept;
  virtual ~exception();
  virtual const char* what() const noexcept;
}
```

Пример

```
#include <iostream>
#include <exception>
struct ooops : std::exception {
  const char* what() const noexcept {return "Ura!\n";}
};

int main () {
  ooops e;
  std::exception* p = &e;
  try {
      throw e;       // throwing (выброс) copy-constructs
       }
     catch (std::exception& ex) {
      std::cout << ex.what();
  }
  try {
      throw *p;      // throwing copy-constructs: std::exception(*p)
  } catch (std::exception& ex) {
      std::cout << ex.what();
  }
  return 0;
}
```

Possible output:
Ura!
exception

Имеющиеся исключения

Все исключения стандартной библиотеки наследуются от std::exception.

На данный момент существуют следующие виды исключений:

logic_error
invalid_argument
domain_error
length_error
out_of_range
future_error (C++11)
runtime_error
range_error
overflow_error
underflow_error
system_error (C++11)
ios_base::failure (начиная с C++11)
bad_typeid
bad_cast

bad_weak_ptr (C++11)
bad_function_call (C++11)
bad_alloc
bad_array_new_length (C++11)
bad_exception
ios_base::failure (до C++11)

std::bad_exception

Исключение определено в заголовочном файле <exception>

std::bad_exception - это тип исключения в C++, которое выполняется в следующих ситуациях:

- Если нарушается динамическая спецификация исключений
- Если std::exception_ptr хранит копию пойманного исключения, и при этом конструктор копирования объекта исключения поймал current_exception, тогда генерируется исключение захваченных исключений.

Пример

```
#include <iostream>
#include <exception>
#include <stdexcept>

void my_unexp() { throw; }

void test() throw( std::bad_exception)
{
    throw std::runtime_error("test");
}

int main()
{
    std::set_unexpected(my_unexp);
    try {
         test();
    } catch(const std::bad_exception& e)
    {
        std::cerr << "Caught " << e.what() << '\n';
    }
}
```