Пространства имен и исключения

Никитин Михаил Евгеньевич, 2015

Конвертеры

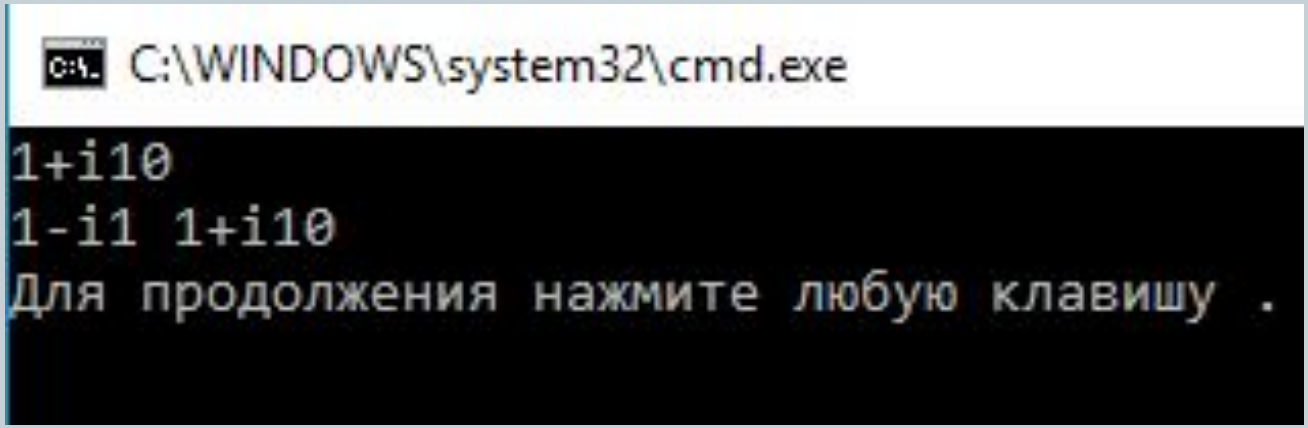

```
struct vect {
    double x,y;
};
class complex {
    double re, im;
public:
    complex(double r = 0, double i = 0) :re(r), im(i) {};
    complex& operator+= (complex);
    complex& operator+= (double);
    complex operator* (complex);
    operator vect();
    double real() const { return re;}
    double imag() const { return im;}
    friend istream& operator>>(istream& in, complex& c);
    };
```

Конвертеры


```
complex::operator vect(){
    vect a;
    a.x = re;
    a.y = im;
    return a;
}
```

Конвертеры


```
int main()
{
    complex a = complex(1, -1);
    complex b = 2.0f+a;
    cin >> b;
    vect d = (vect)b;
    std::cout << a << " " << b << endl;
    return 0;
}
```



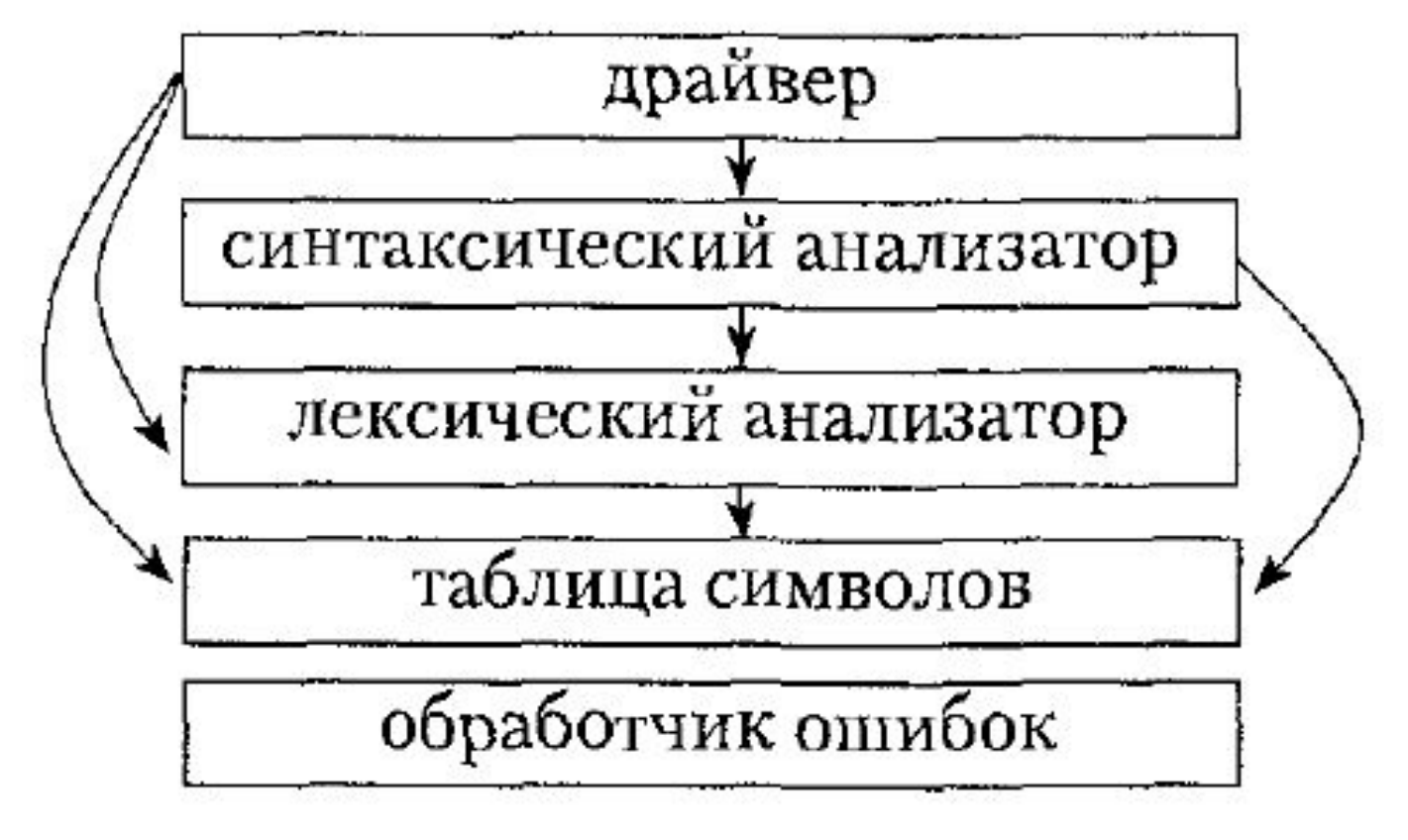
Разбиение на модули и интерфейсы

- Любая реальная программа состоит из некоторого количества отдельных частей.
- Например, даже простая программа, такая как ***Hello, world!***
- ***Задача. Написать программу калькулятора, которая поддерживает четыре стандартных арифметических действия в качестве инфиксных операторов над числами с плавающей точкой. Кроме того, пользователь может определять переменные.***

Калькулятор

- Калькулятор состоит из пяти частей:
  - Обработчика, выполняющего синтаксический анализ;
  - Лексического анализатора, выделяющего лексемы из последовательности символов;
  - Таблицы символов, в которой хранятся пары (строка, значение);
  - Управляющей программы (драйвера) ***main ();***
  - Обработчика ошибок.

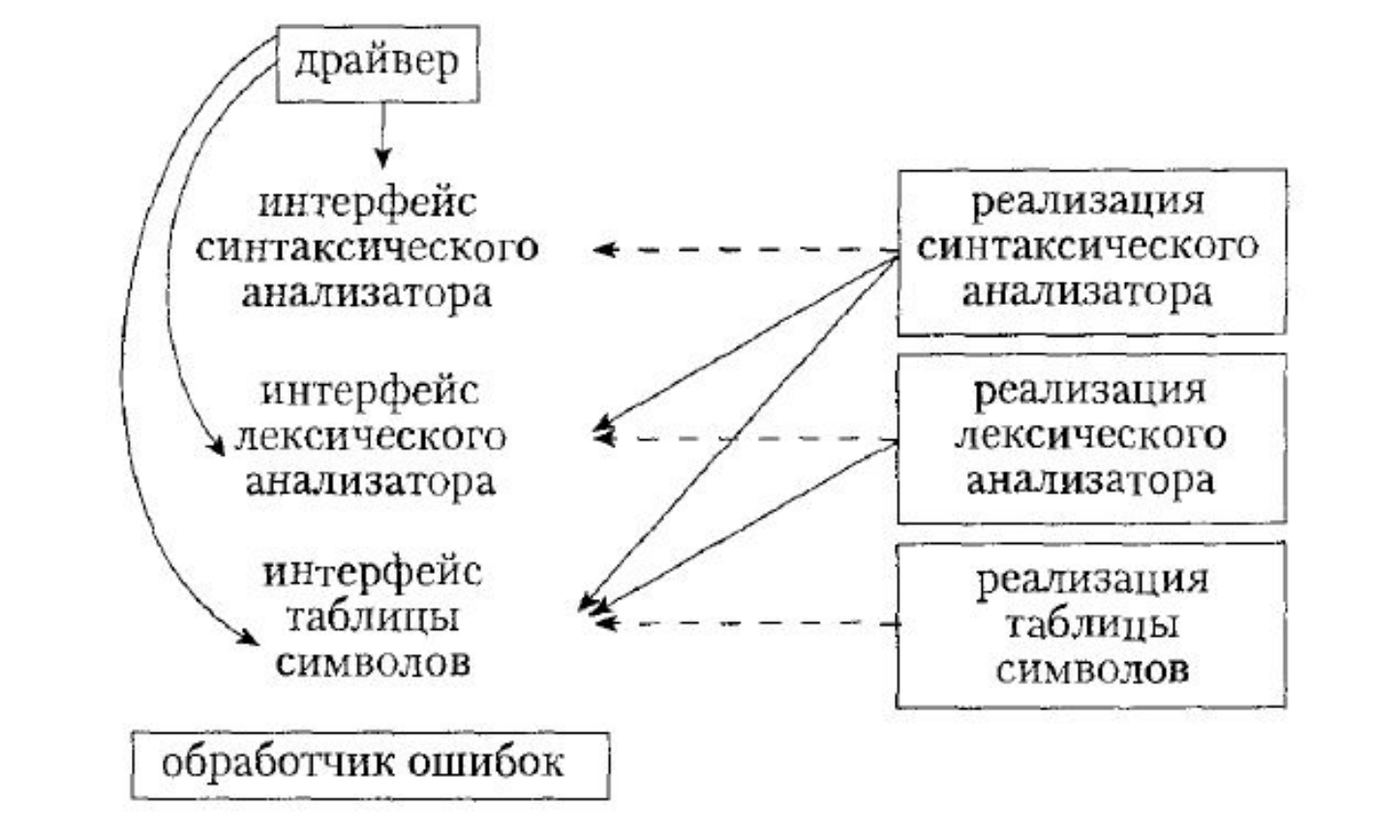
Калькулятор



Калькулятор

- Когда один модуль пользуется другим, ему не надо знать все подробности об используемом модуле.
- В идеале большая часть деталей реализации модуля должна быть неизвестна его пользователям.
- Следовательно, мы проводим различие между модулем и его интерфейсом.

Калькулятор



Калькулятор

- Преимущества такого подхода:
  - код будет «простым»;
  - код будет эффективным;
  - код будет понятным;
  - код будет легким в сопровождении;
  - возможна параллельная разработка.

Калькулятор

- Как реализовать логически эту структуру?
- Как реализовать физически данную структуру?
- Какие для этого есть возможности в C++?

Пространства имен

- Пространство имен является механизмом отражения логического группирования.
- Например, объявления синтаксического анализатора для калькулятора можно поместить в пространство имен ***Parser***

```
namespace Parser {
    double prim(bool get) {/*...*/}
    double term(bool get) {/*...*/}
    double expr(bool get) {/*...*/}
}
```

Лексический анализатор


```
namespace Lexer {
    enum Token_value {
    NAME, NUMBER, END,
    PLUS = '+', MINUS = '-', MUL = '*', DTV = '/',
    PRINT = ',' , ASSIGN = '=', LP = '(', RP = ')'};
    Token_value curr_tok;
    double number_value;
    std::string string_value;
    Token_value get_token() {/*...*/ }
}
```

Пространства имен


```
namespace Parser {
    double prim(bool get) {/*...*/}
    double term(bool get) {/*...*/}
    double expr(bool get) {/*...*/}
}
```

Что плохого в текущем коде?

Пространства имен


```
namespace Parser {
    double prim(bool get);
    double term(bool get);
    double expr(bool get);
}

double Parser::prim(bool get) {/*...*/ }
double Parser::term(bool get) {/* ... */ }
double Parser::expr(bool get) {/*... */ }
```

Пространства имен

- Члены пространства имен объявляются следующим образом:

namespace имя_пространства_имен {

// объявления и определения

}

- Нельзя объявить новый член пространства имен вне его определения. Например:

```
void Parser::logical(bool); // ошибка: в Parser нет logical
()
```

Имена с квалификаторами

- Пространство имен является областью видимости. Поэтому «пространство имен» является фундаментальной и относительно простой концепцией.

```
double Parser::term(bool get){
    double left = prim(get);
    for (;;)
      switch (Lexer::curr_tok) {
      case Lexer::MUL:
      left *= prim(true);
      /* ... */
}
```

Объявления using


```
double Parser::prim(bool get) // обработка первичных выражений
{
   if (get) Lexer::get_token();
   switch (Lexer::curr_tok) {
   case Lexer::NUMBER:
      Lexer::get_token();
      return Lexer::number_value; // константа с плавающей точкой
   case Lexer::NAME:
      double& v = Lexer::table[Lexer::string_value];
      if (Lexer::get_token() == Lexer::ASSIGN) v = expr(true);
      return v;
   case Lexer::MINUS: // унарный минус
      return -prim(true);
      case Lexer::LP:
      double e = expr(true);
      if (Lexer::curr_tok = Lexer::RP) return Error::error("ожидалась )");
      Lexer::get_token();  // пропустить скобки ')'
      return e;
   case Lexer::END:
      return 1;
   default:
      return Error::error("ожидалось первичное выражение");
}
```

Объявления using

- Если имя часто используется вне пределов своего пространства имен, довольно утомительно писать его каждый раз с квалификатором.

Объявления using


```
double Parser::prim(bool get) // обработка первичных выражений
{
   using Lexer::get_token; // использовать get_token из Lexer
   using Lexer::curr_tok; // использовать curr_tok из Lexer
   using Error::error; // использовать error из Error
   if (get) get_token();
   switch (curr_tok) {
   case Lexer::NUMBER:
      get_token();
      return Lexer::number_value; // константа с плавающей точкой
   case Lexer::NAME:
      double& v = Lexer::table[Lexer::string_value];
      if (get_token() == Lexer::ASSIGN) v = expr(true);
      return v;
   case Lexer::MINUS: // унарный минус
      return -prim(true);
      case Lexer::LP:
      double e = expr(true);
      if (curr_tok = Lexer::RP) return Error::error("ожидалась )");
      get_token();  // пропустить скобки ')'
      return e;
   case Lexer::END:
      return 1;
   default:
      return Error::error("ожидалось первичное выражение");
}
```

Объявления using

- Такие синонимы следует делать как можно более локальными во избежание конфликтов имен.
- Можем поместить ***using-объявление*** в определение пространства имен ***Parser***

```
namespace Parser {
    double prim(bool get);
    double term(bool get);
    double expr(bool get);
    using Lexer::get_token; // использовать get_token из
Lexer
    using Lexer::curr_tok; // использовать curr_tok из
Lexer
    using Error::error; // использовать error из Error
}
```

Директивы using

- ***using-директива*** делает доступными имена из пространства имен почти точно так же, как если бы они были объявлены вне своих пространств

```
using namespace Lexer;
using namespace Error;
```

Псевдонимы пространств имен

- Короткие названия пространств имен могут войти в конфликт друг с другом:

```
namespace A {
    //...
}
```

- Однако, длинные названия пространств имен непрактичны при написании реального кода:

```
namespace Russian_Telephone_Book{
    //...
}
```

Псевдонимы пространств имен

- Можно создать короткий псевдоним длинного названия пространства имен:

```
namespace RTB = Russian_Telephone_Book;
RTB::fio s3 = "Григ";
RTB::fio s4 = "Нильсен";
```

Пространство имен

- В идеале, пространство имен должно:
  - выражать логически связанный набор средств;
  - препятствовать доступу пользователей к ненужным им средствам;
  - не требовать значительных дополнительных усилий при использовании.

Обработка ошибок

- При работе программ возникаю ошибки времени выполнения (runtime error), когда дальнейшее нормальное выполнение приложения становится невозможным.
- Причиной ошибок времени выполнения могут быть как ошибки в программе, так и неправильные действия пользователя, неверные данные и т.д.

Обработка ошибок

- Автор библиотеки может обнаружить ошибки времени выполнения, но, в общем случае, не имеет ни малейшего представления, что с ними делать.
- Пользователь библиотеки может знать, как бороться с такими ошибками, но не может их обнаружить — в противном случае, они бы обрабатывались в коде пользователя, и их обнаружение не было бы возложено на библиотеку.

Обработка ошибок

- Обработка ошибок должна быть разделена на две части:
  1. генерация информации о возникновении ошибочной ситуации, которая не может быть разрешена локально;
  2. обработка ошибок, обнаруженных в других местах.

Обработка ошибок

- При обнаружении проблемы, которая не может быть решена локально, функция может:
  1. прекратить выполнение;
  2. возвратить значение, означающее «ошибка»;
  3. возвратить допустимое значение и оставить программу в ненормальном состоянии;
  4. вызвать функцию, предназначенную для обработки «ошибочных» ситуаций.

Обработка ошибок

- Вариант 1 — «прекратить выполнение» — это то, что происходит по умолчанию, когда не перехватывается исключение.
- Вариант 2 — «возвратить значение, сигнализирующее об ошибке» — не всегда выполним, потому что часто нет приемлемого соответствующего значения.

Обработка ошибок

- Вариант 3 — «возвратить допустимое значение и оставить программу в ненормальном состоянии» — имеет тот недостаток, что вызывающая функция может не заметить, что программа находится в ненормальном состоянии.
- Обработка исключений не предназначена для решения проблем, для которых подходит вариант 4 «вызвать функцию обработки ошибок».

Обработка ошибок


```
namespace Error {
    int no_of_errors;
    double error(const char* s)
    {
        std::cerr << "ошибка " << s << std::endl;
        no_of_errors++;
        return 1;
    }
}
```

Обработка ошибок

- Для помощи в решении проблем возникновения и обработки ошибок введено понятие ***исключения.***
- Фундаментальная идея состоит в том, что функция, обнаружившая проблему, но не знающая как ее решить, ***генерирует*** (throw) исключение в надежде, что вызвавшая ее (непосредственно или косвенно) функция сможет решить возникшую проблему.
- Функция, которая хочет решать проблемы данного типа, может указать, что она ***перехватывает*** (catch) такие исключения.

Исключения

- Механизм обработки исключений предоставляет
  - альтернативу традиционным методам в тех случаях, когда они не достаточны, не элегантны и подвержены ошибкам.
  - способ явного отделения кода обработки ошибок от «обычного» кода.
  - более регулярный способ обработки ошибок, упрощая в результате взаимодействие между отдельно написанными фрагментами кода.

Исключения. Блок try-catch

- Простейший формат защищенного блока имеет вид:

Полный формат защищенного блока имеет вид:
try {операторы защищенного блока}
catch-блоки

Catch-блок имеет один из следующих форматов:
catch (тип) {обработчик ошибочной ситуации}
catch (тип идентификатор) {обработчик ошибочной ситуации}
catch (...) {обработчик ошибочной ситуации}

Исключения. Блок try-catch

- Простейший формат защищенного блока :

```
try
    {операторы защищенного блока}
catch(...)
    {обработчик ошибочной ситуации}
```

```
int main()
{
   int x = 0;
   try {
    std::cout << 2 / x; //Здесь произойдет генерация исключения
                // Последующие операторы выполняться не будут
   }
   catch (...) {
    std::cout << "Division by zero" << std::endl;
   }
}
```

- **Project Properties -> C/C++ -> Code Generation -> Modify the Enable C++ Exceptions to "Yes With SEH Exceptions"**

throw

- throw (бросить) - ключевое слово, "создающее" ("возбуждающее") исключение.

```
struct Range_error{
 int i;
 Range_error(int _i) {i = _i;}
};
char to_char(int i)
{
 if(i< 0 ||255 < i)
 throw Range_error(i);
 return i;
}
```