Объектно-ориентированное программирование на алгоритмическом языке C++

МИРЭА, Институт Информационных технологий,
кафедра Вычислительной техники

Автор: доцент, канд. физ.-мат. наук,
Путуридзе Зураб Шотаевич

Реализация сигналов и обработчиков

Для компактной и выразительной реализации кода, можно воспользоваться параметризированным макроопределением препроцессора. Например:

1. Для получения указателя на метод сигнала объекта

```
#define SIGNAL_D( signal_f ) ( TYPE_SIGNAL ) ( & signal_f )
```

2. Для получения указателя на метод обработчика объекта

```
#define HENDLER_D( hendler_f ) ( TYPE_HANDLER ) ( & hendler_f )
```

Реализация сигналов и обработчиков

Для определения указателей на метод сигнала и метод обработчика, можно воспользоваться определением новых типов данных:

1. Указатель на метод сигнала объекта

```
typedef void ( cl_base :: * TYPE_SIGNAL ) ( string & );
```

2. Указатель на метод обработчика объекта

```
typedef void ( cl_base :: * TYPE_HANDLER ) ( string );
```

Необходимо задать пространство имен базового класса cl_base :: - это определяет, что указатель относиться к методу класса.

Сигналы и обработчики

Описание взаимодействия объектов посредством сигналов и обработчиков

№	ob_s	signal_* (string &)	ob_h	hendler_* (string)
1	ob_1	signal_1 (string &)	ob_2	hendler_summ (string)
2	ob_1	signal_1 (string &)	ob_3	hendler_cn (string)
3	ob_4	signal_2 (string &)	ob_3	hendler_2 (string)
4	ob_3	signal_3 (string &)	ob_4	hendler_1 (string)
5	ob_4	signal_4 (string &)	ob_5	hendler_1 (string)

```
void «имя сигнала»      ( string & s_text );
void «имя обработчика» ( string   s_text );
```

Реализация сигналов и обработчиков

Тогда в заголовочной части базового класса можно определить структуру:

```
struct o_sh                         // Структура задания одной связи
{
  TYPE_SIGNAL  p_signal;            // Указатель на метод сигнала
  cl_base    * p_cl_base;           // Указатель на второй объект
  TYPE_HANDLER p_hendler;           // Указатель на метод обработчика
}
```

А для хранения установленных связей можно объявить вектор:

```
vector < o_sh * > connects;
```

Реализация сигналов и обработчиков

В заголовочной части базового класса можно добавить:

```
public:
  . . . . .
  void set_connect    ( TYPE_SIGNAL   p_signal,
                        cl_base     * p_object, TYPE_HANDLER   p_ob_hendler );
  void delete_connect ( TYPE_SIGNAL   p_signal,
                        cl_base     * p_object, TYPE_HANDLER   p_ob_hendler );
  void emit_signal    ( TYPE_SIGNAL   p_signal, string       & s_command    );
```

Реализация сигналов и обработчиков

Например, реализация метода установки связи может иметь вид:

```
void cl_base :: set_connect ( TYPE_SIGNAL    p_signal,
                              cl_base      * p_object,
                              TYPE_HANDLER   p_ob_hendler )
{
    o_sh  * p_value;
    //-------------------------------------------------
    // Цикл для исключения повторного установления связи
    for ( unsigned int i = 0; i < connects.size ( ); i ++ )
       {
          if ( connects [ i ] -> p_signal  == p_signal     &&
               connects [ i ] -> p_cl_base == p_object     &&
               connects [ i ] -> p_hendler == p_ob_hendler    )
             {
                   return;
             }
       }

    p_value =  new o_sh ( );

    p_value -> p_signal  = p_signal;
    p_value -> p_cl_base = p_object;
    p_value -> p_hendler = p_ob_hendler;

    connects.push_back ( p_value );
}
```

Использование сигналов и обработчиков

В части реализации алгоритма вызов метода установки связи может иметь вид:

```
p_signal_object -> set_connect ( SIGNAL_D  ( cl_3 :: signal_1 ),
                                 ( ( cl_3 * ) p_hendler_object ),
                                 HENDLER_D ( cl_3 :: hendler_1 ) );
. . . . .
```

В части реализации алгоритма вызов метода выдачи сигнала может иметь вид:

```
p_signal_object -> emit_signal ( SIGNAL_D ( cl_application :: signal_1 ),
                                 s_text );
. . . . .
```

Новые имена типов данных

Ключевое слово typedef – определяет новое наименование для существующих типов данных.

typedef тип новое_имя;

Новое имя дополнение а не замена существующего типа.

```
typedef float  balance;
. . . . .
balance  bal_1;
typedef balance balance_week;
```

Динамическая идентификация типа

Класс type_info описывает тип объекта.
Заголовочный файл <typeinfo>
Для получения типа объекта во время выполнения программы используется функция – typeid (объект)

typeid возвращает ссылку на объект типа type_info

```
bool operator== ( const type_info & объект );
bool operator!= ( const type_info & объект );
const char * name ( );
```

Пример 1 динамической идентификации

```
#include <iostream>
#include <typeinfo>
using namespace std;
class myclass {
  // ...
};
int main() {
  int      k, j;
  float    d;
  myclass ob;
  cout << "The type of k  is: " << typeid( k).name() << endl;
  cout << "The type of d  is: " << typeid( d).name() << endl;
  cout << "The type of ob is: " << typeid(ob).name() << "\n\n";
  if(typeid(k) == typeid(j))
    cout << "The types of k and j are the same\n";
  if(typeid(k) != typeid(d))
    cout << "The types of k and d are not the same\n";
  return 0;
}
```

Пример 1 динамической идентификации

```
The type of k  is: i
The type of d  is: f
The type of ob is: 7myclass

The types of k and j are the same
The types of k and d are not the same
```

Пример 2 динамической идентификации

```
#include <iostream>
#include <typeinfo>
using namespace std;

class Base {
  virtual void f() { }
};

class Derived1: public Base { };
class Derived2: public Base { };
```

Пример 2 динамической идентификации

```
int main()
{
  Base     * p, baseob;
  Derived1 ob1;
  Derived2 ob2;

  p = & baseob;
  cout << "p is pointing to an object of type ";
  cout << typeid ( * p ).name ( ) << endl;

  p = & ob1;
  cout << "p is pointing to an object of type ";
  cout << typeid ( * p ).name ( ) << endl;

  p = & ob2;
  cout << "p is pointing to an object of type ";
  cout << typeid ( * p ).name ( ) << endl;

  return 0;
}
```

Пример 2 динамической идентификации

```
p is pointing to an object of type 4Base
p is pointing to an object of type 8Derived1
p is pointing to an object of type 8Derived2
```

Если закомментировать `virtual void f() { }`

```
p is pointing to an object of type 4Base
p is pointing to an object of type 4Base
p is pointing to an object of type 4Base
```

Если указатель нулевой, typeid генерирует исключение типа bad_typeid

Операторы приведения типов

```
( целевой_тип )
dynamic_cast      < целевой_тип > ( выражение )
const_cast        < целевой_тип > ( выражение )
reinterpret_cast  < целевой_тип > ( выражение )
static_cast       < целевой_тил > ( выражение )
```

dynamic_cast

Оператор dynamic_cast выполняет операцию приведения полиморфных классов (типов) во время выполнения программы.

Преобразует указатель (ссылку) одного типа в указатель (в ссылку) другого типа.

Неудачное приведение указателя возвращает 0

Неудачное приведение ссылки генерирует исключение типа bad_cast

Пример 1 dynamic_cast

```
#include <iostream>
using namespace std;

class Base {
  virtual void f() { }
};

class Derived1: public Base { };

int main()
{
  Base     * p_b,  ob_b;
  Derived1 * p_d1, ob_d1;

  p_b = & ob_d1;
  p_d1 = dynamic_cast < Derived1 * > ( p_b );

  if ( p_d1 ) cout << "Cast OK" << endl;
  return 0;
}
```

Пример 2 dynamic_cast

```
#include <iostream>
using namespace std;

class Base {
  virtual void f() { }
};

class Derived1: public Base { };

int main()
{
  Base     * p_b,  ob_b;
  Derived1 * p_d1, ob_d1;

  p_b = & ob_b;
  p_d1 = dynamic_cast < Derived1 * > ( p_b );

  if ( ! p_d1 ) cout << "Cast error" << endl;
  return 0;
}
```

const_cast

Оператор const_cast переопределяет модификаторы const и/или volatile.

Спецификатор const предотвращает модификацию переменной при выполнении программы

Спецификатор volatile информирует транслятор о том, что переменная может быть изменена внешними (по отношению к программе) факторами

Используется для удаления признака постоянства.

Пример 4 const_cast

```
#include <iostream>
using namespace std;
void f( const int * p ) {
   int * v;
   // ......
    v = const_cast < int * > ( p );
   *v = 100;
}
int main()
{
   int x = 99;
   cout << "x before call: " << x << endl;
   f ( & x );
   cout << "x after call: "  << x << endl;
   return 0;
}
```

```
x before call: 99
x after call: 100
```

static_cast

Оператор static_cast выполняет операцию неполиморфного приведения типов.

Можно использовать для любого стандартного преобразования.

При работе программы проверки на допустимость не выполняются.

Пример 5 static_cast

```
#include <iostream>
using namespace std;

int main()
{
   int   i;
   float f;

   f = 199.22F;

   i = static_cast < int > ( f );

   cout << i;

   return 0;
}
```

```
199
```

reinterpret_cast

Оператор reinterpret_cast выполняет фундаментальное изменение типа.

Преобразует один тип в принципиально другой.

Например, для преобразования указателя в целое значение и целого значения – в указатель.

Пример 6 reinterpret_cast

```
#include <iostream>
using namespace std;

int main()
{
  int    i;
  char * p = "This is a string";

  // cast pointer to integer

  i = reinterpret_cast < int > ( p );

  cout << i;

  return 0;
}
```

```
4210789
```

Потоки

Поток – последовательность байтов, который связан с физическим устройством.

Поток – логический интерфейс, который связан с физическим устройством.

Одно устройство компьютера может “выглядеть” подобно любому другому.

Связь потока с устройством устанавливается по операции открытия.

Отсоединение потока от устройства выполняется по операции закрытия.

Устройство $\Leftrightarrow$ Поток (интерфейс) $\Leftrightarrow$ Оперативная память

Потоки

Два типа потоков: текстовый и двоичный.

Текстовый поток используется для ввода-вывода символов.

Допускается преобразование символов.

Двоичный поток может использоваться с данными любого типа.

Преобразование символов не выполняется, существует взаимно-однозначное соответствие.

Текущая позиция – это место на устройстве (в файле), с которого будет выполняться следующая операция доступа (ввода-вывода)

Встроенные объекты потоков

Встроенные объекты потоков:

cin – объект входного потока.

cout – объект выходного потока.

cerr – вывод информации об ошибках, объект выходного потока, не буферизованный.

clog – вывод информации об ошибках, объект выходного потока, буферизованный.

Встроенные объекты потоков открываются автоматический.

По умолчанию устройство вывода – консоль.

По умолчанию устройство ввода – клавиатура.

Двухбайтовые встроенные потоки

Двухбайтовые встроенные потоки:

wcin – объект входного потока.

wcout – объект выходного потока.

wcerr – вывод информации об ошибках, объект выходного потока, не буферизованный.

wclog – вывод информации об ошибках, объект выходного потока, буферизованный.

Встроенные объекты потоков открываются автоматический.

Библиотека – <iostream>

Оператор << – вводит информацию в объект поток.

Оператор >> – выводит информацию из объекта потока.

Форматированный ввод-вывод

Два способа:

1. Методы класса ios.
2. Посредством манипуляторов.

Флаги форматирования (перечисление fmtflags)

adjustfield	floatfield	rights	skipws
basetfield	hex	scientific	unitbuf
boolalpha	internal	showbase	uppercase
dec	left	showpoint	
fixed	oct	showpos	

Методы класса ios

Метод setf (fmtflags) – установка флага форматирования.

Метод unsetf (fmtflags) – очистка флага форматирования.

skipws – ведущие “пробельные” символы отбрасываются.

left – выводимые данные выравниваются по левому краю.

right – выводимые данные выравниваются по правому краю.

oct – вывод в восьмеричном представлении.

hex – вывод в шестнадцатеричном представлении.

dec – вывод в десятичном представлении.

showbase – вывод основания системы счисления (1F – 0x1F).

Пример методов класса ios

```
#include <iostream>
using namespace std;
int main ( ) {
     cout.setf ( ios :: showpos );
     cout.setf ( ios :: scientific );
     cout << 123 << " " << 123.23 << " ";
     return 0;
}
```

```
+123 +1.232300e+002
```

Манипуляторы ввода-вывода

Манипуляторы позволяют встраивать инструкции форматирования в выражении ввода-вывода

В основном совпадают с наименованиями флагов.

Для манипуляторов с аргументами надо включить в программу

```
#include  <iomanip>
```

Пример манипуляторов ввода-вывода

```
#include <iostream>
#include <iomanip>
using namespace std;
int main ( ) {
    cout << setprecision ( 6 ) << 1234.5678 << endl;
    cout << setw ( 20 ) << "right" << endl;
    cout << setw ( 20 ) << left    << "right";
    return 0;
}

1234.57
                right
right
```

Файловый ввод-вывод

Для файлового ввода-вывода надо включить в программу

#include <fstream>

Определение файловых потоков.

```
ifstream  in;     // входной поток
ofstream  out;    // выходной поток
fstream   both;   // поток ввода-вывода
```

Открыть файл посредством метода open ( )

Пример файлового ввода-вывода

```
#include <iostream>
#include <iomanip>
#include <fstream>
using namespace std;
int main ( ) {
    ofstream out ( "test.txt" );
    if ( ! out ) {
        cout << "File not open!" << endl;
        return 1;
    }
    out << setprecision ( 6 ) << 1234.567 << endl;
    out << setw ( 20 )          << "right"  << endl;
    out << setw ( 20 )          << left     << "right";
    ut.close ( );
    return 0;
}
```