Объектно-ориентированное программирование на алгоритмическом языке C++

МИРЭА, Институт Информационных технологий,
кафедра Вычислительной техники

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

Шаблоны классов

```
template < class Ttype > class «имя класса»
{
   // описание класса
};
```

где - Ttype представляет имя типа, который будет задан при создании объекта класса, используя следующий формат:

«имя класса» < тип > «имя объекта»;

Здесь «тип» определяет имя типа данных, который будет обрабатываться объектом класса. Методы члены класса, автоматически являются обобщенными.

Пример шаблона класса

```
// Демонстрация класса очереди queue.
#include <iostream>
using namespace std;

const int SIZE = 100;

// Описание класса queue.
template < class QType > class queue {
  QType q [ SIZE ];
  int   sloc, rloc;
public:
  queue ( )  { sloc = rloc = 0; }
  void  qput ( QType i );
  QType qget ( );
};
```

Пример шаблона класса

```
template < class QType > void queue < QType > :: qput ( QType i ) {
    if ( sloc == SIZE ) {
        cout << "Очередь заполнен.\n";
        return;
    }
    sloc ++;
    q [ sloc ] = i;
}

template < class QType > QType queue < QType > :: qget ( ) {
    if ( rloc == sloc ) {
        cout << "Очередь пуст.\n";
        return 0;
    }
    rloc ++;
    return q [ rloc ];
}
```

Пример шаблона класса

```
int main ( )  {
    queue < int > a, b;  // две очереди для целых
чисел
    a.qput ( 10 );       b.qput ( 19 );
    a.qput ( 20 );       b.qput (  1 );
    cout << a.qget ( ) << " ";
    cout << a.qget ( ) << " ";
    cout << b.qget ( ) << " ";
    cout << b.qget ( ) << "\n";
    queue < double > c, d; // две очереди для
действительных чисел
    c.qput ( 10.12 );   d.qput ( 19.99  );
    c.qput ( -20.0 );   d.qput (  0.986 );
    cout << c.qget ( ) << " ";
    cout << c.qget ( ) << " ";
    cout << d.qget ( ) << " ";
    cout << d.qget ( ) << "\n";
    return 0;
}
```

Использование параметров, не являющихся типами

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

// Здесь, int size это параметр не являющийся типом.
template < class AType, int size > class atype
{
    AType a [ size ]; // параметр size передает длину массива
public:
    atype ( ) {
        register int i;
        for ( i = 0; i < size; i++ ) a [ i ] = i;
    }
    AType & operator[] ( int i );
};

// Обеспечение контроля границ.
template < class AType, int size >
AType & atype < AType, size > :: operator[] ( int i )
{
    if ( i < 0 || i > size - 1 ) {
        cout << "\nЗначение индекса ";
        cout << i << " за пределами границ массива.\n";
        exit ( 1 );
    }
    return a [ i ];
}
```

Использование параметров, не являющихся типами

```
int main ( )  {
    Atype < int,    10 > intob;    // массив целых чисел
    Atype < double, 15 > doubleob; // массив действительных
                                   //    чисел
    int i;

    cout << "Целочисленный массив: ";
    for ( i = 0; i < 10; i++ ) intob [ i ] = i;
    for ( i = 0; i < 10; i++ ) cout << intob [ i ] << "  ";
    cout << '\n';

    cout << "Действительный массив: ";
    for ( i = 0; i < 15; i++ ) doubleob [ i ] = ( double ) i/3;
    for ( i = 0; i < 15; i++ ) cout << doubleob [ i ] << "  ";
    cout << '\n';

    intob [ 12 ] = 100; // Ошибка времени выполнения

    return 0;
}
```

Обработка исключительных ситуаций

Обработка исключений – это системные средства, с помощью которых программа может справиться с ошибками времени выполнения.

Основные инструкции:

try – блок, в блоке размещается фрагмент кода реализации, в котором ожидается возникновение ошибки времени выполнения.

catch – блок, предназначен для перехвата исключении.

throw – генерирует исключение, которое перехватывается catch инструкцией.

Синтаксис инструкций

```
try {
    // фрагмент кода реализации
}
catch ( type1 arg ) {
    // обработчик исключения type1
}
catch ( type2 arg ) {
    // обработчик исключения type2
}
// . . . . .

catch ( . . . ) {
    // обработчик исключения любого типа
}
```

Инструкция throw

В try блоке

throw выражение определенного типа;

В catch блоке

throw; // повторное генерирование исключения

Пример 1

```
int main() {
    cout << "начало\n";
try { // начало блока try
    cout << "Внутри блока try\n";
    throw 10;      // возбуждение ошибки
    cout << "Эта инструкция выполнена не будет";
}
catch ( int i ) { // перехват ошибки
    cout << "перехвачена ошибка номер: ";
    cout << i << "\n";
}
    cout << "конец";
    return 0;
}
```

Ответ примера 1

```
начало
Внутри блока try
Перехвачена ошибка номер: 10
конец
```

Пример 2

```
int main() {
    cout << "начало\n";
try {                  // начало блока try
    cout << "Внутри блока try\n";
    throw 10;          // возбуждение ошибки
    cout << "Эта инструкция выполнена не будет";
}
catch ( double i ) { // Эта инструкция не будет работать
                     // с исключительной ситуацией целого типа
    cout << "перехвачена ошибка номер: ";
    cout << i << "\n";
}
    cout << "конец";
    return 0;
}
```

Ответ примера 2

```
начало
Внутри блока try
Abnormal program termination
```

Пример 3

```
void Xtest ( int test ) {
    cout << "Внутри функции Xtest, test равно: " << test << "\n";
    if ( test ) throw test;
}
int main () {
    cout << "начало\n";
    try { // начало блока try
        cout << "Внутри блока try\n";
        Xtest ( 0 );
        Xtest ( 1 );
        Xtest ( 2 );
    }
    catch ( int i ) { // перехват ошибки
        cout << "перехвачена ошибка номер: ";
        cout << i << "\n";
    }
    cout << "конец";
    return 0;
}
```

Ответ примера 3

```
начало
Внутри блока try
Внутри функции Xtest, test равно: 0
Внутри функции Xtest, test равно: 1
перехвачена ошибка номер: 1
конец
```

Пример 4

```
void Xhandler ( int test ) {
    try {
        if ( test ) throw test;
    }
    catch ( int i ) {
        cout << "перехвачена ошибка номер : " << i << ' \n ';
    }
}
int main ( ) {
    cout << "начало\n";
    Xhandler ( 1 );
    Xhandler ( 2 );
    Xhandler ( 0 );
    Xhandler ( 3 );
    cout << "конец";
    return 0;
}
```

Ответ примера 4

начало

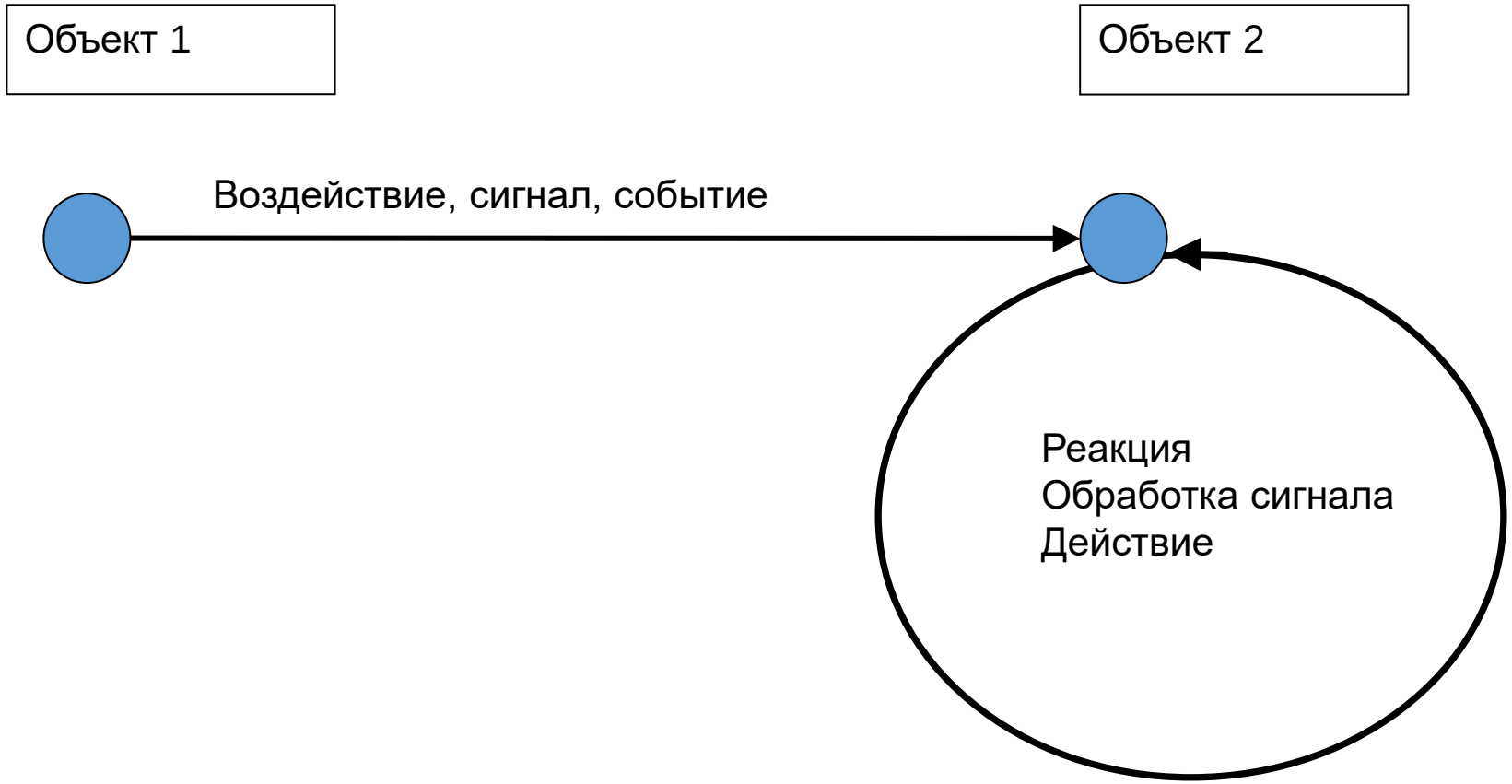
перехвачена ошибка номер: 1

перехвачена ошибка номер: 2

перехвачена ошибка номер: 3

конец

Схема взаимодействия объектов



Сигналы и обработчики

Описание взаимодействия объектов посредством сигналов и обработчиков

№	Объект 1	Сигнал и сообщение	Объект 2	Обработчик и сообщение
1	ob_1	Сигнал о начале вычисления суммы. Массив чисел.	ob_2	Вычисление суммы
2	ob_4	Сигнал для переименования. Новое имя.	ob_3	Переименование объекта
3	ob_2	Вычисление суммы выполнено. Результат вычисления	ob_3	Получение результата вычисления суммы

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

Методы сигналов и обработчиков принадлежат производным объектам (реализованы в производных классах).

Методы для:
1. **Установки связи между сигналом и обработчиком.**
2. **Разрыва связи между сигналом и обработчиком.**
3. **Выдачи сигнала объектом.**

Необходимо реализовать в базовом классе.

Реализация сигналов и обработчиков

В каждой связи задействованы:

1. **Объект выдающий сигнал (первый объект).**
2. **Сигнал первого объекта.**
3. **Объект получающий сигнал (второй объект).**
4. **Обработчик второго объекта.**

Установка связи и выдача сигнала осуществляется от первого объекта. По этому, необходима структура, которая будет содержать:

1. **Указатель на метод сигнала первого объекта.**
2. **Указатель на второй объект.**
3. **Указатель на метод обработчика второго объекта.**

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.