Объектно-ориентированное программирование

Наследование шаблонов. Исключительные ситуации.

Наследование шаблонов

Наследование шаблонов

```
class CompanyA {
public:
...
void sendClearText(const std::string& msg);
void sendEncryptedText(const std::string& msg);
...
};
```

```
class MsgInfo {...};
```

```
template<typename Company> class MsgSender {

public:
...                                  // конструктор, деструктор и т. п.

void sendClear(const MsgInfo& info)
{
   std::string msg;
   //создать msg из info
   Company c;
   c.sendClearText(msg);
}

void sendSecret(const MsgInfo& info) // аналогично sendClear,
{...}                                // но вызывает c.sendEncrypted

};
```

Наследование шаблонов

```
template <typename Company>
class LoggingMsgSender: public MsgSender<Company> {
public:
...
void sendClearMsg(const MsgInfo& info)
{
//записать в протокол перед отправкой;

sendClear(info); // вызвать функцию из базового класса
                 // этот код не будет компилироваться!

//записать в протокол после отправки;
}
...
};
```

Наследование шаблонов

- Когда компилятор встречает определение шаблона класса, который наследуется от другого шаблона он не знает, какому классу тот наследует.
- Понятно, что наследование идет от шаблона, но параметр шаблона не известен до момента конкретизации.
- Не зная значения параметра шаблона, невозможно понять, как выглядит класс. В частности, не существует способа узнать, есть ли в нем какая-либо определенная функция.

Наследование шаблонов

```
class CompanyZ { // этот класс не представляет
public:          // функции sendCleartext
...
void sendEncrypted(const std::string& msg);
...
};

template <> // полная специализация MsgSender;
class MsgSender <CompanyZ> { // отличается от общего шаблона

public: // только отсутствием функции
...     // sendCleartext

void sendSecret(const MsgInfo& info)
{...}

};
```

Возможные варианты вызова базовых методов

- Самый распространенный вариант: использовать this

```
template <typename Company>
class LoggingMsgSender: public MsgSender<Company> {
public:
...
void sendClearMsg(const MsgInfo& info)
{
// записать в протокол перед отправкой

this->sendClear(info); // порядок! Предполагается, что
                       // sendClear будет унаследована

// записать в протокол после отправки
}
...
};
```

Возможные варианты вызова базовых методов

- Второй вариант: использовать `using`

```
template <typename Company>
class LoggingMsgSender: public MsgSender<Company> {
public:

using MsgSender<Company>::sendClear; // сообщает компилятору, что
...                                  // sendClear есть в базовом классе

void sendClearMsg(const MsgInfo& info)
{
...
sendClear(info); // нормально, предполагается, что
...              // sendClear будет унаследована
}
...
};
```

Возможные варианты вызова базовых методов

- Третий вариант: явное указание базовой функции

```
template <typename Company>
class LoggingMsgSender: public MsgSender<Company> {
pubilc:
...
void sendClearMsg(const MsgInfo& info)
{
...
MsgSender<Company>::sendClear(info);
    // нормально, предполагается, что
... // sendClear будет унаследована
}
...
};
```

- НО! если вызываемая функция виртуальна, то явная квалификация отключает динамическое связывание.

Наследование шаблонов

- С точки зрения видимости имен, все три подхода эквивалентны: они обещают компилятору, что любая специализация шаблона базового класса будет поддерживать интерфейс, предоставленный общим шаблоном.
- Такое обещание – это все, что необходимо компилятору во время синтаксического анализа производного шаблонного класса, но если данное обещание не будет выполнено, истина всплывет позже (не скомпилируется).

Обработка исключительных ситуаций

Исключения

- Обработка исключений – это механизм, позволяющий двум независимо разработанным программным компонентам взаимодействовать в аномальной ситуации, называемой *исключением*.
- *Исключение* – это аномальное поведение во время выполнения, которое программа может обнаружить, например:
  - Сбой ОС или железа: Не сработавший системный вызов, мусор при возврате из системной функции
  - отсутствие нужных нам ресурсов: память, файловые дескрипторы и др.
  - ошибка в логике программы: удаление несуществующего элемента, вызов метода с неправильными параметрами и др.

Исключения

- Исключения можно разделить на ожидаемые и неожиданные:
  - С ожидаемыми – как-то работаем (сообщаем пользователю что не смогли открыть файл и продолжаем работу)
  - С неожиданными – не работаем и падаем (лучше перезапуститься, чем испортить файлы невалидными данными)

Обработка исключений

- Каким образом обрабатывались исключительные ситуации до введения механизма исключений?

```
int rc = f();
if (rc == 0) {
  ...
} else {
  ...code that handles
     the error...
}
```

- С использованием механизма исключений?

```
try {
  f();
  ...
} catch (std::exception& e) {
  ...code that
     handles the error...
}
```

Обработка исключений

- Каким образом исключение должно передаваться по иерархии вызовов функций? Предположим, f1->f2->f3->...->f10.
- С использованием исключений:

```
void f1() {
  try { f2(); }
   catch (some_exception& e) {
     ...code that handles the error… }
}

 void f2() { ...; f3(); ...; }
 void f3() { ...; f4(); ...; }
 void f4() { ...; f5(); ...; }
 void f5() { ...; f6(); ...; }
 void f6() { ...; f7(); ...; }
 void f7() { ...; f8(); ...; }
 void f8() { ...; f9(); ...; }
 void f9() { ...; f10(); ...; }

 void f10() {
   if (...some error condition...)
     throw some_exception(); }
```

Без использования обработки исключений

- Без использования обработки исключений:

Обработка исключений в C++

- В C++ исключения реализуются посредством классов.
- Для инициации исключения используется инструкция `throw`.

```
// инструкция является вызовом конструктора
throw popOnEmpty();
```

- Эта инструкция создает объект-исключение типа «popOnEmpty».

THROW В РЕАЛЬНЫХ УСЛОВИЯХ

```
#include "stackExcp.h"
void iStack::pop( int &top_value )
{
   if ( empty() )
      throw popOnEmpty();

   top_value = _stack[ --_top ];

   cout << "iStack::pop(): "<<top_value " endl;
}
```

Перехват исключений в C++

- Инструкции, которые могут возбуждать исключения, должны быть заключены в `try`-*блок*.
- Такой блок начинается с ключевого слова `try`, за которым идет последовательность инструкций, заключенная в фигурные скобки, а после этого – список обработчиков, называемых `catch`-*предложениями*.
- `try`-блок группирует инструкции программы и ассоциирует с ними обработчики исключений.

CATCH В РЕАЛЬНЫХ УСЛОВИЯХ

```
try
{    // try-блок для исключений popOnEmpty
 if ( ix % 10  == 0 ) {
    int dummy;
    stack.pop( dummy );
    stack.display();
    }
}
catch ( popOnEmpty ) { ... }
```

Обработка исключительных ситуаций в C++

Обработка исключительных ситуаций

- Инструкции, которые могут возбуждать исключения, должны быть заключены в `try`-*блок*.
- Такой блок начинается с ключевого слова `try`, за которым идет последовательность инструкций, заключенная в фигурные скобки, а после этого – список обработчиков, называемых `catch`-*предложениями*.
- `try`-блок группирует инструкции программы и ассоциирует с ними обработчики исключений.

Блок catch

- В языке C++ исключения обрабатываются в предложениях `catch`. Когда какая-то инструкция внутри try-блока возбуждает исключение, то просматривается список последующих предложений `catch` в поисках такого, который может его обработать
- Catch-обработчик состоит из трех частей:
  - ключевого слова `catch`,
  - объявления одного типа или одного объекта, заключенного в круглые скобки (оно называется *объявлением исключения*),
  - и составной инструкции.

Пример блока catch

```
catch ( pushOnFull ) {
   cerr << "trying to push value on a full stack\n";
   return;
}
catch ( popOnEmpty ) {
   cerr << "trying to pop a value on an empty stack\n";
   return;
}
catch ( const char * ) {
   cerr << “some const char* exception\n";
   return;
}
catch ( ... ) {
   cerr << “some const char* exception\n";
   return;
}
```

Блок catch

- Для обработки исключения выбирается тот обработчик, для которого типы в объявлении исключения и в возбужденном исключении совпадают (или могут быть приведены друг к другу)
- После завершения обработчика выполнение приложения возобновляется с инструкции, идущей за последним `catch`-обработчиком в списке.

Обработчик catch

- Обработчик исключительной ситуации считается найденным, если тип в блоке `catch`
  - Тот же, что и в `throw` (T, const T, T&, const T&)
  - Или является открытым базовым классом для класса исключения, указанного в `throw`
  - Или является указателем, который может быть преобразован к типу указателя в `throw`
  - Или в качестве параметра `catch` вместо класса указано «...», что данный блок перехватывает и обрабатывает любой тип исключений

```
catch ( ... ) {
   cerr << "I’m your last chance!!!\n”;
}
```

Объявление исключения

- Объявлением исключения в catch-обработчике могут быть объявления типа или объекта.
- Объявление типа исключения позволяет идентифицировать исключительную ситуацию, но не позволяет получить дополнительную информацию о возникшем исключении.

```
catch ( pushOnFull ) {
   cerr << "trying to push value on a full stack\n";
   return;
}
```

Объект-исключение

- Если необходимо получить значение или как-то манипулировать объектом, созданным в выражении `throw` создается объект-исключеиние.
- При перехвате объектов-исключений (или ссылок на объекты исключения) инструкции внутри `catch`-обработчика могут обращаться к информации, сохраненной в объекте выражением `throw`.

Генерация объекта-исключения


```
// новый класс исключения:
// он сохраняет значение, которое не удалось поместить в стек
class pushOnFull {
public:
   pushOnFull( int i ) : _value( i ) { }
   int value { return _value; }
private:
   int _value;
};

void iStack::push( int value )
{
   if ( full() )
      throw pushOnFull( value );
   // ...
}
```

Перехват и обработка объекта-исключения

- В объявлении исключения в `catch`-обработчике появляется определение ссылки на объект eObj, с помощью которого вызывается функция-член value() класса pushOnFull

```
catch ( pushOnFull &eObj ) {
   cerr << "trying to push value << "eObj.value()
        << "on a full stack\n";
}
```