

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

Шаблоны классов

ВОПРОСЫ

- ⊙ Что такое vtable?
- ⊙ Каким образом vtable помогает решить проблему ромбовидного наследования?
- ⊙ Какие существуют типы параметров шаблонов?
- ⊙ Что такое инстанцирование шаблона?

ШАБЛОНЫ КЛАССОВ

ШАБЛОН КЛАССА

- ◎ Шаблон класса позволяет задать класс, параметризованный типом данных.
- ◎ Передача классу различных типов данных в качестве параметра создает семейство родственных классов.
- ◎ Наиболее широкое применение шаблоны находят при создании *контейнерных классов*.

КОНТЕЙНЕРНЫЙ КЛАСС

- ◎ Контейнерным называется класс, который предназначен для хранения каким-либо образом организованных данных и работы с ними.
- ◎ Преимущество использования шаблонов состоит в том, что как только алгоритм работы с данными определен и отлажен, он может применяться к любым типам данных без переписывания кода.

СИНТАКСИС ОПИСАНИЯ ШАБЛОНА КЛАССА

```
template <описание_параметров_шаблона> class имя  
{ /* определение класса */ };
```

◎ Например:

```
template <typename Type>  
class Queue {  
public:  
    Queue();  
    ~Queue();  
  
    Type& remove();  
    void add( const Type & );  
    bool is_empty();  
    bool is_full();  
private:  
    // ...  
};
```

СОЗДАНИЕ КЛАССОВ НА ОСНОВЕ ШАБЛОНОВ

- ⊙ Определение методов шаблона вне класса:

```
template <typename T> Queue<T>::add (const T &a)  
{ /*реализация класса*/ }
```

- ⊙ Пример создания классов Очереди (Queue) для целых чисел, комплексных чисел, строк:

```
Queue<int> qi;  
Queue<complex<double>> qc;  
Queue<string> qs;
```

СПЕЦИАЛИЗАЦИЯ ШАБЛОНОВ

- ◎ Генерация конкретного класса из обобщенного определения шаблона называется конкретизацией шаблона.
- ◎ При такой конкретизации `Queue` для объектов типа `int` каждое вхождение параметра `Type` в определении шаблона заменяется на `int`, так что определение класса `Queue` принимает вид.
- ◎ Части `<int>` и `<string>`, следующие за именем `Queue`, называются фактическими аргументами шаблона.

СПЕЦИАЛИЗАЦИЯ ШАБЛОНОВ

```
template <class int>
class Queue {
public:
    Queue() : front( 0 ), back (
0 ) { }
    ~Queue();

    int& remove();
    void add( const int & );
    bool is_empty() const {
        return front == 0;
    }
private:
    QueueItem<int> *front;
    QueueItem<int> *back;
};
```

ЧАСТИЧНАЯ СПЕЦИАЛИЗАЦИЯ ШАБЛОНОВ

- ⊙ Если у шаблона класса есть несколько параметров, то можно специализировать его только для одного или нескольких аргументов, оставляя другие неспециализированными.

```
template <int hi, int wid>
class Screen {
    // ...
};
// частичная специализация шаблона класса Screen
template <int hi>
class Screen<hi, 80> {
public:
    Screen();
    // ...
private:
    string          _screen;
    string::size_type _cursor;
    short           _height;
    // для экранов с 80 колонками используются специальные алгоритмы
};
```

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> {
public:
    ...
    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?
- ⊙ С использованием механизма исключений?

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

```
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.

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.

САТЧ В РЕАЛЬНЫХ УСЛОВИЯХ

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

- ◎ Шаблоны классов позволяют реализовать контейнерные классы
- ◎ Шаблоны классов позволяют провести конкретизацию (в том числе частичную)
- ◎ В процессе конкретизации интерфейс класса может быть изменен
- ◎ Наследование шаблонов – штука не надежная
- ◎ Для вызова метода родительского шаблона лучше всего использовать `this`
- ◎ Обработка исключительных ситуаций методом `try/catch` позволяет несколько улучшить читаемость исходного кода