# Шаблоны (templates)

С помощью шаблонов можно создавать родовые (обобщенные) функции и родовые классы. В родовой функции или классе *тип данных*, с которыми функция или класс работают, *задается в качестве параметра*. То есть можно использовать одну и ту же функцию или класс с несколькими различными типами данных без перепрограммирования.

Родовые функции определяют базовый набор операций, которые будут применяться к разным типам данным. Родовая функция оперирует с тем типом данных, который она получает в качестве параметра. Таким образом, компилятор автоматически генерирует правильный код для фактически используемого при выполнении функции типа данных. (То есть, родовая функция фактически автоматически перегружается сама).

**Замечание**: Понятия *родовой функции*, *шаблонной функции* и *обобщенной функции* эквивалентны. То же касается и классов. Конкретная версия обобщенной функции, создаваемая компилятором, называется *специализацией* или *генерируемой функцией*.

Родовая функция создается с помощью ключевого слова **template** (шаблон).

Формат создания обобщенной функции:

template <class Ф\_тип>

возвр\_знач имя\_ф(список\_параметров)

{

// тело функции

}

Здесь вместо Ф\_тип указывается некоторое имя типа используемых функцией данных. Это *фиктивное имя*, которое компилятор автоматически заменит именем реального типа данных при создании конкретной версии функции.

**Пример**: // *обмен данных*

#include <iostream>

using namespace std;

template <class X> void swaping(X &a, X &b)

{

X c;

c = a;

a = b;

b = c;

}

int main()

{

double i = 5.5, j = 7.7;

char a = 'a', b = 'z';

cout << "i=" << i << "\tj=" << j << endl;

cout << "a=" << a << "\tb=" << b << endl;

swaping(i, j);

swaping(a, b);

cout << "i=" << i << "\tj=" << j << endl;

cout << "a=" << a << "\tb=" << b << endl;

system("pause");

return 0;

}

**Замечание**: В операторе **template** можно определить несколько обобщенных типов, элементы которых разделены запятыми.

**Пример**:

…

template <class X, class Y> void myfunc(X a, Y b)

{

// тело функции

}

Перегрузка родовой (шаблонной) функции и явная перегрузка

Для того чтобы перегрузить обобщенную функцию, необходимо создать еще одну функцию, отличающуюся от остальных списком параметров или заданием явного типа.

**Пример**:

#include <iostream>

using namespace std;

// Первая версия шаблонной функции myfunc()

template <class X> X myfunc(X &a)

{

return a += a;

}

// Явная перегрузка функции myfunc()

float myfunc(float &a)

{

return a \*= a;

}

// Вторая версия шаблонной функции myfunc()

template <class X, class Y> X myfunc(X &c, Y &m)

{

return c \*= m;

}

int main()

{

int i = 10;

double d = 1.3;

cout << "i=" << i << "\td=" << d << endl;

myfunc(i);

myfunc(d);

cout << "i=" << i << "\td=" << d << endl;

float f = 1.5;

cout << "f=" << f << endl;

myfunc(f);

cout << "f=" << f << endl;

int j = 5;

double k = 2.5;

cout << "j=" << j << "\tk=" << k << endl;

myfunc(k, j);

cout << "j=" << j << "\tk=" << k << endl;

system("pause");

return 0;

}

Результаты выполнения программы:

i=10 d=1.3

i=20 d=2.6

f=1.5

f=2.25

j=5 k=2.5

j=5 k=12.5

**Родовые (обобщенные) классы**

В родовых классах определяются алгоритмы, а фактические типы данных определяются после.

Функции-члены родового класса сами автоматически становятся родовыми. Для них не обязательно явно задавать ключевое слово template.

В C++ существует библиотека стандартных шаблонов (STL – Standard Template Library).

Основная форма объявления родового класса:

template <class Ф\_тип> class имя\_класса{

…

}

Создание конкретного экземпляра класса

имя\_класса <тип > объект;

Здесь <тип> – имя типа данных, с которым будет оперировать класс.

Рассмотрим пример создания обобщенного класса, в котором используются два обобщенных типа данных и одна явная специализация класса.

**Пример**:

#include <iostream>

using namespace std;

template <class X, class Y> class myclass

{

X i;

Y j;

public:

myclass(X a, Y b) { i = a; j = b; }

void show() { cout << i << ' ' << j << endl; }

};

template <> class myclass <float, float>

{

float i;

float j;

public:

myclass(float a, float b) { i = a; j = b; }

void show() { cout << "i\*j=" << i \* j << endl; }

};

int main()

{

myclass<int, double> ob1(10, 0.23);

myclass<char, const char \*> ob2('X', "Template");

myclass<float, float> ob3(5.2, 1.1);

ob1.show();

ob2.show();

ob3.show();

system("pause");

return 0;

}

Результаты выполнения программы:

10 0.23

X Template

i\*j=5.72

**Комментарий**: Для варианта, когда явно передаются типы (в нашем случае float,float), используется формат явной специализации класса: **template <> class myclass <float,float>.**

***Задание:*** *Написать класс* ***matrix*** *c использованием template.*

# Обработка исключительных ситуаций

C++ обеспечивает встроенный механизм обработки ошибок, называемый обработкой исключительных ситуаций.

Обработка исключительных ситуаций в C++ организуется с помощью трех ключевых слов: **try, catch, throw**.

Инструкции программы, во время выполнения которых необходимо обеспечить обработку исключительных ситуаций, располагаются в блоке try. Если исключительная ситуация (ошибка) имеет место внутри блока try, она возбуждается (ключевое слово throw), перехватывается (ключевое слово catch) и обрабатывается.

Инструкция catch располагается непосредственно за блоком try, возбуждающим исключительную ситуацию.

Блок **try** должен содержать ту часть программы, в которой надо отследить ошибки. Если исключительная ситуация перехвачена, аргумент arg получает ее значение.

***Основная форма:***

try{

// блок возбуждения исключительной ситуации

throw исключительная\_ситуация;

}

catch (type1 arg) {

// блок перехвата исключительной ситуаци

}

…

catch (typeN arg) {

// блок перехвата исключительной ситуаци

}

Если возбуждается исключительная ситуация, для которой нет соответствующей инструкции **catch**, то вызывается стандартная библиотечная функция **terminate()**, которая вызывает функцию **abort().**

**Пример**:// *Исключительные ситуации в главной функции*

#include <iostream>

using namespace std;

int main()

{

setlocale(LC\_ALL, "Russian");

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 (int): ";

cout << i << "\n";

}

cout << "Конец" << endl;

system("pause");

return 0;

}

**Пример**:// *Исключительные ситуации в вызываемой функции*

#include <iostream>

using namespace std;

void Err(int);

int main()

{

setlocale(LC\_ALL, "Russian");

cout << "Начало\n";

Err(10);

Err(0);

Err(15);

cout << "Конец" << endl;

system("pause");

return 0;

}

void Err(int test)

{

try {

if (test) throw test;

else

throw "Аргуменет равен нулю";

}

catch (int i)

{

cout << "Перехвачена ошибка:" << "i=" << i << endl;

}

catch (char const \* str)

{

cout << "Перехвачена строка:" << str << endl;

}

}

Дополнительные варианты использования исключительных ситуаций:

1. перехватывать все исключительные ситуации, независимо от их типа.

catch(…) {

//обработка всех исключительных ситуаций

}

1. Запретить возбуждать какую-либо исключительную ситуацию.

возвр\_тип имя\_ф(список\_аргументов) throw(список\_типов)

{

// …

}

В список\_типов аргументов через запятую перечислить те типы данных исключительных ситуаций, которые могут быть возбуждены функцией.

Список\_типов пустой. Тогда не возбуждается никаких исключительных ситуаций.

**Пример**: // *Обработка исключительной ситуации – деление на ноль*

#include <iostream>

using namespace std;

int main()

{

double a = 1, b;

while (a != 0)

{

cout << "Input a, b (a=0 ->exit)" << endl;

cin >> a >> b;

try {

if (!b) throw b;

cout << "rez=" << a / b << endl;

}

catch (...) {

cout << "ZERO!!!" << endl;

}

}

system("pause");

return 0;

}