Лекция 8. Шаблоны в языке С++

**Шаблоны функций**

**Шаблон** - это средство языка, предназначенное для создания обобщённых алгоритмов без привязки к некоторым параметрам. Точное значение параметра определяется на этапе компиляции. Таким образом, создаётся функция, которая автоматически перегружает себя.

Функции-шаблоны имеют следующий вид:

template <class список\_параметров>

тип имя\_функции(список\_параметров)

{

//тело функции

}

**class** используется для пользовательских типов данных;

**typename** используется для встроенных типов данных;

Функционально они не различаются.

Пример. Написать программу, находящую минимальное значение из двух элементов.

#include <iostream>

using namespace std;

template <class tp>//Шаблон функции

tp minx(tp x, tp y)

{

return (x<y)? x:y;

}

int main(){

int x=4, y=3;

cout<<minx(x, y)<<endl;//Выводит 3

double a=4.563, b=9.74;

cout<<minx(a, b)<<endl;//Выводит 4.563

char h='s', f='a';

cout<<minx(h, f)<<endl;//Выводит 'a'

//double e=4.563; char u='a';

//cout<<minx(e, h)<<endl;

//Ошибка: параметр "tp" неоднозначен

return 0;

}

Для функций-шаблонов автоматическое преобразование типов не выполняется. Шаблон может быть описан следующим образом:

1) Шаблон с одним параметром.

template <class параметр>

void f1(параметр имя\_переменной)

{

//тело функции

}

В заголовке функции могут быть использованы параметры других типов.

2) Шаблон с несколькими параметрами.

template <class параметр1, class параметр2>

void f1(параметр1 имя\_переменной1, параметр2 имя\_переменной2)

{

//тело функции

}

Порядок следования параметров в шаблоне значения не имеет, в отличие от порядка следования параметров в функции.

Пример. Найти среднее арифметическое значение элементов массива.

#include <iostream>

using namespace std;

template <typename tmas, typename tsz>

double sred(const tmas \*mas, tsz n)

{

double s=0;

for(int i=0; i<n; i++) s+=mas[i];

return s/n;

}

int main()

{

const int n1=5;

int mas1[n1]={1, 2, 3, 4, 5};

const long n2=3;

double mas2[n2]={10, 20, 30};

cout<<"Average mas1 = "<<sred(mas1, n1)<<endl;

cout<<"Average mas2 = "<<sred(mas2, n2)<<endl;

return 0;

}

Объявление шаблона не приводит к генерации кода. Сама генерация (реализация шаблона) происходит в месте вызова функции. В этом случае генерируется код с используемым при данном вызове типом аргументов. При повторном вызове с аналогичными аргументами генерация кода не происходит. Если встречается другой тип аргументов, то происходит ещё одна реализация шаблонов. Использование шаблонов вместо перегруженных функций не уменьшает размер программы (исполняемого модуля), но сокращает текст программы и увеличивает её надёжность. В списке параметров функции-шаблона обязательно должны присутствовать типы, перечисленные в списке параметров шаблона. Список параметров шаблона не может быть пустым. Кроме параметров типов допускается использование параметров констант. В этом случае в место имени параметра подставляется значение константы из шаблона. Для определения типа и значения константы исследуются фактические аргументы, переданные при вызове функций. Процесс определения типов и значения аргументов шаблонов по известным фактическим аргументам называется **выведением аргументов шаблонов**.

Пример. Найти сумму элементов массива.

#include <iostream>

using namespace std;

template <typename tp, int n>

tp sum(tp(&mas) [n])

{

tp s=0;

for(int i=0; i<n; i++) s+=mas[i];

return s;

}

int main()

{

int mas1[]={1, 2, 3, 4, 5};

cout<<sum(mas1)<<endl;

double mas2[3]={2.7, 7.88, 12.5};

cout<<sum(mas2)<<endl;

return 0;

}

Параметром функции в шаблоне sum является ссылка на массив элементов типа tp.

Для сопоставления с формальным параметром функции фактический аргумент также должен быть значением, представляющим тип массива. При выводе аргумента шаблона принимается во внимание, что, например mas1, является параметром типа указатель на массив элементов типа int.

**Алгоритм выведения аргументов шаблона**:

1) По очереди исследуется каждый фактический аргумент функции для того, чтобы выяснить, присутствует ли в соответствующем формальным параметре какой-нибудь параметр шаблона.

2) Если параметр шаблона найден, то путём анализа типа фактического аргумента выводится соответствующий аргумент шаблона.

3) Если один и тот же параметр найден в нескольких формальных параметрах, то аргумент шаблона должен быть одинаков.

Недостаток данного подхода в том, что, например, для предыдущего примера, даже при использовании одного типа массива для различного количества аргументов генерируются различные функции.

Разрешено явное задание аргументов шаблона.

Пример. Явное задание аргументов шаблона.

#include <iostream>

using namespace std;

template <class tx, class ty, class tz>

void fun(ty, tz) {}

int main()

{

fun <int, char, double> (1,'2');//tx - int, ty - char, tz - double

fun <int, char> (1, '2');//tx - int, ty (1) - char, tz ('2') - char

fun <int> (1, '2');//tx - int, ty - int, tz - char

//fun('1', '2');//Ошибка: не удаётся составить аргумент шаблона для "tx"

return 0;

}

Если некоторые типы фактических аргументов шаблона имеют отличную от других реализацию, то можно использовать явную специализацию.

Пример. Найти частное двух чисел.

#include <iostream>

using namespace std;

template <class tp1, class tp2>

double chast(tp1 x, tp2 y)

{

if(y==0) exit(1);

return x/y;

}

template <>

double chast <int, int> (int x, int y)

{

if(y==0) exit(1);

return static\_cast <double> (x)/y;

}

int main()

{

cout<<chast(2, 3.4)<<endl;

cout<<chast(1, 3)<<endl;

return 0;

}

Такое решение не является переопределением функции.

При наличии нескольких одноименных функций поиск функций для выполнения осуществляется в следующей последовательности:

1) Поиск не шаблонной функции с совпадающим списком параметров.

2) Поиск шаблонной функции с точным соответствием списка параметров.

3) Поиск функции по условию совпадения списка параметров после разрешённых преобразований типов.

Анализ компилятором шаблонов функций не предполагает преобразование типов.

**Шаблоны классов**

template <class параметр>

class имя\_класса

{

//тело функции

}

При включении шаблона класса в программу сам класс не генерируется до тех пор, пока не будет создан экземпляр шаблонного класса с указанным конкретным типом.

**Создание экземпляра класса:**

имя\_класса <тип> имя;

или

имя\_класса <тип> имя = new имя\_класса <тип>;

Пример. Написать класс, хранящий массив, позволяющий добавлять элементы в конец массива и извлекать элементы из конца массива.

#include <iostream>

using namespace std;

template <class tp>

class ShCls

{

tp mas[100];

int k;

public:

ShCls():k(0) {}

void push(tp x) {mas[k++]=x;}

tp pop() {return mas [k--];}

};

int main()

{

ShCls <int> a;//Создание шаблонной функции

ShCls <double> b;//Создание шаблонной функции

a.push(1);//Ввод данных

a.push(4);

cout<<a.pop()<<" "<<a.pop()<<endl;

b.push(15.34);

b.push(48.3);

cout<<b.pop()<<" "<<b.pop()<<endl;

return 0;

}

Если методы класса описываются вне класса, то обязательно добавление template перед заголовком каждого метода.

Например, функция выглядела бы таким образом:

template <class tp>

tp ShCls<tp>::pop()

{

return mas[k--];

}

Пример. Написать класс для работы со стеком.

template <class st>

struct tstk//Объявление стека tstk <st>

{

st inf;

tstk\* a;

};

template <class tp>

class StkCl

{

tstk <tp> \*sp;//Указатель на вершину стека

public:

StkCl():sp(NULL) {}

void push(tp inf);//Добавить данные ~StkCl();

};

template <class tp>

void StkCl <tp>::push(tp inf)

{

tstk <tp> \*spt=new tstk <tp>;

spt->inf=inf;

spt->a=sp;

sp=spt;

}

template <class tp>//Вывод и освобождение памяти

StkCl <tp>::~StkCl()

{

tstk<tp> \*spt;

while (sp!=NULL){

spt=sp;

cout<<sp->inf<<endl;

sp=sp->a;

delete spt;

}

return;}//return можно не писать

int main()

{

StkCl <double> s;

s.push(101.1);

s.push(202.2);

StkCl<char> ch;

ch.push('a');

ch.push('b')

ch.push('c);

return 0;

}

Каждая версия класса, создаваемая по шаблону, содержит одинаковый код с изменяемыми параметрами шаблона. Для класса разрешена полная и частичная специализация.

Для специальной реализации отдельных методов шаблона используются полное либо частичное переопределение (**специализация**) шаблона класса:

template <>

class имя\_класса

<имя\_специализированного\_типа>

{}

Пример. Использование специализации.

#include <iostream>

#include <string>

using namespace std;

template <class tp, int n>// Шаблон класса

class Cls{

protected:

tp \*mas;

public:

Cls();

~Cls();

};

template <class tp, int n>//Конструктор

Cls <tp, n>::Cls(){

mas=new tp[n];//Создание массива элементов типа tp

cout<<endl<<"Array of"<<n<<"elements of the size"<<sizeof(tp);

}

template <class tp, int n>//Деструктор

Cls <tp, n>::~Cls(){

cout<<endl<<"Deallocation of"<<n<<"elements of the size"<<sizeof(tp);

delete[] mas;

}

//Частичная специализация класса

Cls <double, 12>::Cls(){

mas=new tp[n];

cout<<endl<<"Spartial specialization: array of"<<n<<"elements of the size"<<sizeof(tp);

}

template <>

Cls<double, 12>::Cls(){

mas=new double[12];

cout<<endl<<"Spartial specialization: array of"<<12<<"elements of the size"<<sizeof(double);

}

//Полная специализация класса

const int k=20;

template <>

Cls <char, k>(){

protected:

char \*mas;

public:

Cls <char, k>(){

mas=new char[k];

mas[0]='\0';

cout<<endl<<"Full specialization: array of"<<k<<"elements of char";

}

~Cls(){

cout<<endl<<"Full specialization: deallocation of"<<k<<"elements of char";

delete[] mas;

}

};

int main()

{

Cls <int, 10>x1;//Без специализации

//Array of 10 elements of the size 4

Cls <double, 12>x2;//Частичная специализация

//Spartial specialization: array of 12 elements of the size 8

Cls <int, 12>x3;//Без специализации

//Array of 12 elements of the size 4

Cls <char, 20>x4;//Полная специализация

//Full specialization: array of 20 elements of char

Cls <char, 30>x5;//Без специализации

//Array of 30 elements of the size 1

cout<<endl;

//Deallocation of 30 elements of the size 1

//Full specialization: deallocation of 20 elements of char

//Deallocation of 12 elements of the size 4

//Deallocation of 12 elements of the size 8

//Deallocation of 10 elements of the size 4

return 0;

}

Для настройки шаблона класса можно использовать функциональные объекты. **Функциональным** называется объект, для которого переопределёна операция вызова функции: operator().

Пример. Использование функционального оператора.

#include <iostream>

using namespace std;

struct sum{

int operator() (int x, int y)

{return x+y;}

};

int main(){

sum s;

cout<<s(3, 4)<<endl;//Выводит 7

cout<<sum()(5, 6)<<endl;//Выводит 11

return 0;}

Пример. Создать шаблонный класс для выбора минимального или максимального полей класса. Стратегия выбора должна передаваться как параметр шаблона.

#include <iostream>

using namespace std;

template <class tp>

struct Less{

bool operator()(const tp&x, const tp&y)

{

return x<y;

}

};

template <class tp>

struct More{

bool operator()(tpx, tpy)

{

return x>y;

}

};

template <class tp, class Compare>

class Cls{

tp a, b;

public:

Cls(tp x, tp y):a(x), b(y){}

void OutSelect()

{

cout<<(Compare()(a, b)?a:b)<<endl;

}

};

int main(){

Cls <int, Less<int> >f1(22, 6);

f1.OutSelect();//Выводит 6

Cls <double, More<double> >f2(24.2, 7.3);

f2.OutSelect();//Выводит 24.2

return 0;

}

В шаблонах класса допускается использование статических элементов данных, дружественных функций и вложенных классов.

Достоинства шаблонов:

1) Высокая эффективность работы с различными типами объектов.

2) Безопасное использование различных типов.

Недостатки:

1) Увеличение размера исполняемого модуля.

2) Реализация шаблона может оказаться оптимальной только для некоторого ограниченного набора типов.