Модульное программирование

Перегрузка функций, шаблоны функций, директивы препроцесора

Перегрузка функций

Перегрузка функций

Часто бывает удобно, чтобы функции, реализующие один и тот же алгоритм для различных типов данных, имели одно и тоже имя. Метафора имени распространяется на все типы данных.

Перегрузка функций

Использование нескольких функций с одним и тем же именем, но с различными типами параметров, называется *перегрузкой функций.*

Компилятор самостоятельно определяет, какую именно функцию требуется вызвать, по типу фактических параметров.

Этот процесс называется разрешением перегрузки.

Перегрузка функций

Тип возвращаемого результата в разрешении не участвует.

Рассмотрим примеры:

```
int max(int, int);

int max(int, *char);

long max(char *, int);

char *max(char *, char *);
```

Перегрузка функций

При вызове функции компилятор выбирает соответствующий типу фактических параметров вариант функции.

Если точного соответствия не найдено, выполняется преобразования в соответствие со стандартом языка, например, bool и char в int, float в double, etc. Далее выполняются преобразования, заданные пользователем, а также поиск соответствий за счет параметров по умолчанию.

Перегрузка функций

Если ни одного соответствия не найдено, выдается диагностическое сообщение об ошибке.

Неоднозначность может появиться в следующих случаях:

- при преобразовании типов;
- при использовании параметров-ссылок;
- при использовании аргументов по умолчанию;
- при использовании модификатора const перед именем параметра.

Перегрузка функций

Следующие функции нельзя считать перегруженными:

```
int max(int, int);
double max(int, int);
long max(int, const int);
int max(int, int &);
```

Перегрузка функций

Следующий пример показывает неоднозначность при наличии параметров по умолчанию.

```
int f(int a){return a;}
int f(int a, int b=1){return a*b;}
cout << f(10,2);  // вызывается f(int a, int b=1)
cout << f(10);// неоднозначность
```

Перегрузка функций

Неоднозначность возникает при неопределенности преобразований, например,

```
float f(float);
double f(double);
f(10);     // к какому типу преобразовывать, к типу float или double ?
```

Перегрузка функций

- функции не могут быть перегруженными, если описание их параметров отличается только модификатором const или использованием ссылки.

С перегрузкой функций мы встретимся при перегрузке операций в классах.

Перегрузку функций часто называют слабой формой полиморфизма.

Шаблоны функций

***Шаблоны функций**

В C++ есть мощное средство параметризации алгоритма – шаблоны функций. С помощью него можно определять алгоритм, который применим к различным типам данных, а конкретный тип данных передается функции в виде параметров на этапе компиляции.

Шаблонные функции автоматически перегружают самих себя.

Шаблоны функций

Общий формат объявления шаблонной функции:

```
template<class Type> заголовок
{
   // тело функции
}
```

<class Type> - список параметров шаблона.

Слово class в списке заменимо на слово typename.

Шаблоны функций

В общем случае список шаблона может содержать несколько типов, например,

```
template<class A, class B, int i> void f()
{
   // ….
}
```

Шаблоны функций

В качестве первого примера шаблона функции вспомним функцию swap:

```
template<typename Type>
void swap(Type &arg_1, Type &arg_2)
{
    Type temp;
    temp = arg_1;
    arg_1 = arg_2;
    arg_2 = temp;
}
```

Шаблоны функций

Вызов этой функции может быть осуществлен двумя способами:

- как вызов обычной функции без спецификации параметров шаблона

```
int a=10, b=20;
swap(a,b);
double x=3.45, y=-5.88;
swap(x,y);
```

Шаблоны функций

- со спецификацией параметров шаблона

```
int a=120, b=-45;
swap<int>(a,b);
```

Обратите внимание на то, что после имени функции перед списком фактических параметров в угловых скобках указывается список «фактических» типов параметров шаблона. Если предполагается несколько типов, в списке их нужно перечислить через запятую.

Шаблоны функций

В качестве параметров шаблона функции могут выступать как стандартные типы, так и типы, определенные пользователем. Однако здесь необходимо учитывать особенность, что в типе данных, определенных пользователем (структура, класс), необходимо перегрузить те операции, которые использует шаблонная функция.

Шаблоны функций

Рассмотрим пример

```
class Test
{
protected:
    float test;
public:
    Test(){};
    Test(float ):test(t){};
    Test operator *(const Test &t)
    { return this->test*t.test; }
    friend ostream &operator <<(ostream &, const Test &);
};
```

Шаблоны функций

```
ostream &operator <<(ostream &out, const Test &t)
{
    out << t.test;
    return out;
}
// шаблон функции умножения
template<typename Type> Type
mult(Type arg_1, Type arg_2)
{ return arg_1*arg_2; }
```

Шаблоны функций

```
// вызов функции для пользовательского типа
Test tst_1(3.4F), tst_2(-4.55F);
cout << mult(tst_1,tst_2) << endl;
```

Как говорилось ранее, можно явно указать тип передаваемого параметра

```
cout << mult<Test>(tst_1,tst_2) << endl;
```

Шаблоны функций

Этот шаблон может быть использован и при работе со стандартными типами данных:

```
float f_1 =1.11F, f_2=3.33F;
cout << mult<float>(f_1,f_2) << endl;
```

Большую возможность предоставляют шаблонные классы, которые позволяют параметризировать не только алгоритм, но и типы данных, с которыми работают эти алгоритмы.

Шаблоны функций

Рассмотрим пример несложного шаблонного класса.

```
template<typename Type> class Test
{
protected:
    Type test;
public:
    Test(){};
    Test(Type t):test(t){};
    Test operator *(const Test<Type> &t);
    friend ostream &operator << <>(ostream &, const
    Test<Type> &);
};
```

Шаблоны функций

Реализация составных и дружественных функций шаблонного класса:

```
template<typename Type>
ostream &operator <<(ostream &out, const Test<Type> &t)
{
    out << t.test;
    return out;
}

template<typename Type>
Test<Type> Test<Type>:: operator *(const Test<Type> &t)
{
    return this->test*t.test;
}
```

Шаблоны функций

Объявление объектов шаблонных классов и их использование:

```
Test<int> tst_int_1=10, tst_int_2=200;
cout << tst_int_1*tst_int_2 << endl;

Test<double> tst_double_1(77.84),
    tst_double_2(4.1);
cout << tst_double_1*tst_double_2 << endl;
```

Шаблоны функций

Функция в качестве возвращаемого результата

Мы знаем, что функция не может вернуть в качестве результата массив и другую функцию, но указатели на них вернуть может. Рассмотрим простой пример возврата указателя на функцию.

Шаблоны функций

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

double mult(double d)
{
   return d*2;
}

typedef double (*PF)(double);
```

Шаблоны функций

```
PF func()
{
    return mult;
}

int main()
{
    PF ptr_fun = func();
    cout << ptr_fun(55.6478) << endl;
    return 0;
}
```

Функция main

Функция, которой передается управление после запуска (вызова) программы на исполнение, должна иметь имя main. Она может возвращать результат в вызвавшую ее систему (операционную систему) и принимать параметры из внешнего окружения.

Функция main

Стандарт предусматривает два формата функции:

тип main()

{ // ……. }

тип main(int argc, char *argv[])

{ // ……..}

Тип результата int. Возвращаемое значение – 0.

Функция main

Пример вызова функции main с параметрами из командной строки:

d:\CPP\program_main.exe one two three <enter>

Директивы препроцессора

*** Директивы препроцессора**

Препроцессором называется первая фаза компиляции. Инструкции (команды) препроцессора называются директивами. Они должны начинаться с символа ‘#’.

Директива include

Директива include<имя_файла> осуществляет подстановку указанного файла в точку, где она записана.

Директивы препроцессора

Поиск указанного файла начинается со стандартного каталога include, имеющегося в любой реализации языка. После чего осуществляется поиск в текущем каталоге.

Пользователь может заставить препроцессор искать включаемый файл с текущего каталога, если имя укажет в двойных кавычках: #include”name_file.h”

Директивы препроцессора

Заголовочные файлы обычно имеют расширение h и могут содержать:

- определение типов, встроенных функций, шаблоны, перечисления;
- объявления (прототипы) функций, данных, имен, шаблонов;
- пространства имен;
- директивы препроцессора;
- комментарии.

Директивы препроцессора

В заголовочных файлах не должно быть определение функций и данных. Их принято выносить в файлы реализации. Это не требования языка, это рекомендация.

При указании имен файлов стандартной библиотеки расширение можно опускать. Для большинства старых версий файлов, заимствованных от языка C, в языке C++ есть аналоги файлов без расширения, например,

stdlib.h и cstdlib, stdio.h и cstdio, и т.д.

Директивы препроцессора

#include”file_name.h”
file_name_main.cpp
Файл основной программы, содержащий функцию main

file_name.h
Заголовочный файл(ы)

file_name_1.cpp
file_name_2.cpp
…….
file_name_n.cpp
Файлы реализации

Директивы препроцессора

Директива #define

Директива define определяет подстановку в тексте программы. Она используется для определения:

- символических констант:

#define имя текст_подстановки,

Например,

#define PI 3.14

В любом контексте символьная константа PI будет интерпретироваться как число 3.14.

Директивы препроцессора

- макросов, которые выглядят как функции, но реализуются подстановкой из текста в текст программы:

#define имя(параметры) текст_подсановки

Например,

#define sqr(x) (x*x)

Использование макросов вносит свои сложности в программы, в частности, особенности передачи аргументов.

Директивы препроцессора

Например, для описанного макроса вызов
cout << sqr(y+1) << endl;

приведет к получению числа 6, для правильного ответа нужно вызвать следующим образом

cout << sqr((y+1)) << endl;

Макросы и символические константы заимствованы из языка С, в С++ они не получили широкого применения.

Директивы препроцессора

- символов, управляющих условной трансляцией. Они используются совместно с директивами #ifdef и #ifndef.

Общий формат:

#define имя

Например,

```
#define VERSION 1
#define h_file “head_file.h”
```

Директивы препроцессора

Имена, объявляемые через директиву define рекомендуется писать прописными символами, чтобы зрительно отличать их от других программных объектов (переменных, функций).

Директивы препроцессора

Директивы условной трансляции

Директивы условной трансляции #if, #ifdef, #ifndef применяются для того, чтобы исключить компиляцию отдельных частей программы. Это бывает полезно при отладке или при поддержке нескольких версий программ для различных платформ.

Директивы препроцессора

Формат директивы #if:

```
   #if константное_выражение
.....
    [ #elif константное_выражение ]
.....
    [ #elif константное_выражение ]
.....
    [ #else ]
   #endif
```

Директивы препроцессора

Исключаемые блоки могут содержать как описания, так и исполняемые операторы. Пример условно исключения различных версий заголовочного файла:

```
#ifdef VERSION == 1
  #define INCLFILE “vers_1.h”
#elif VERSION == 2
  #define INCLFILE “vers_2.h”
#else
  #define INCLFILE “vers_N.h”
#endif
#include INCFILE
```

Директивы препроцессора

В константных выражениях может использоваться проверка, определена ли константа с помощью директивы difine, например:

```
#if defined(__BORLANDC__) &&
__BORLAND__ ==0530        // BC5.3
typedef istream_iterator<int, ptrdiff_t> istream_it;
#else
typedef istream_iterator<int> istream_iter;
```

Директивы препроцессора

И еще одно применение директив условной трансляции – временное комментирование фрагмента кода. Иногда используется в целях отладки.

Предопределенные макросы

В C++ определено некоторое количество макросов , предназначенных в основном для того, чтобы выдавать информацию о версии программы или месте возникновения ошибки.

Предопределенные макросы

Например, макрос __splusplus определен, если программа компилируется в среде C++.

```
#ifdef __cplusplus
   cout << " C++ " << endl;
   #else  cout << " no C++ " << endl;
#endif
```

Это макрос использовался в период перехода от C к C++.

Предопределенные макросы

Другие макросы:

- __DATE__ - содержит строку с текущей датой (месяц, день, год), например,

```
cout << __DATE__ << endl;
```

- __FILE__ - содержит строку с полным именем текущего файла, например,

```
cout << __FILE__ << endl;
```

Предопределенные макросы

- __LINE__ - текущая строка исходного текста;
- __TIME__ - текущее время.

Предопределенные макросы