Вычислительная техника и компьютерное моделирование в физике

Лекция 6

Зинчик Александр Адольфович

zinchik_alex@mail.ru

Обычно имеет смысл давать разным функциям разные имена. Если же несколько функций выполняет одно и то же действие над объектами разных типов, то удобнее дать одинаковые имена всем этим функциям.

Перегрузкой имени называется его использование для обозначения разных операций над разными типами. Собственно уже для основных операций C++ применяется перегрузка.

Действительно: для операций сложения есть только одно имя +, но оно используется для сложения и целых чисел, и чисел с плавающей точкой, и указателей.

Такой подход легко можно распространить на операции, определенные пользователем, т.е. на функции.

Например:

```
void print(int);          // печать целого
void print(const char*)   // печать строки символов
```

При вызове функции с именем f компилятор должен разобраться, какую именно функцию f следует вызывать.

Для этого сравниваются типы фактических параметров, указанные в вызове, с типами формальных параметров всех описаний функций с именем f.

В результате вызывается та функция, у которой формальные параметры наилучшим образом сопоставились с параметрами вызова, или выдается ошибка если такой функции не нашлось.

Например:

```
void print(double);
void print(long);
void f()
{
print(1L); // print(long)
print(1.0); // print(double)
print(1); // ошибка, неоднозначность: что
   //вызывать

      // print(long(1)) или print(double(1)) ?

}
```

Правила сопоставления применяются в следующем порядке по убыванию их приоритета:

1. **Точное сопоставление:** сопоставление произошло без всяких преобразований типа или только с неизбежными преобразованиями (например, имени массива в указатель, имени функции в указатель на функцию и типа T в const T).

2. **Сопоставление с использованием** стандартных целочисленных преобразований, определенных в (т.е. char в int, short в int и их беззнаковых двойников в int), а также преобразований float в double.

3. Сопоставление с использованием стандартных преобразований, (например, int в double, derived* в base*, unsigned в int).

4. Сопоставление с использованием пользовательских преобразований

Пусть имеются такие описания функции print:

- void print(int);
- void print(const char*);
- void print(double);
- void print(long);
- void print(char);

Тогда результаты следующих вызовов print() будут такими:

```
void h(char c, int i, short s, float f)
{
print(c); // точное сопоставление: вызывается
 //print(char)
print(i); // точное сопоставление: вызывается
 //print(int)
print(s); // стандартное целочисленное
 //преобразование:
        // вызывается print(int)
print(f); // стандартное преобразование:
         // вызывается print(double)
```

```
print('a'); // точное сопоставление:
   //вызывается print(char)
print(49); // точное сопоставление:
   //вызывается print(int)
print(0); // точное сопоставление: вызывается
     //print(int)
print("a"); // точное сопоставление:
            // вызывается print(const char*)
}
```

- Обращение print(0) приводит к вызову print(int), так как 0 имеет тип int.
- Обращение print('a') приводит к вызову print(char), т.к. 'a' - типа char.
- На разрешение неопределенности при перегрузке не влияет порядок описаний рассматриваемых функций, а типы возвращаемых функциями значений вообще не учитываются.

Например:

```
int pow(int, int);
double pow(double, double); // из <math.h>
complex pow(double, complex); // из <complex.h>
complex pow(complex, int);
complex pow(complex, double);
complex pow(complex, complex);
```

```
void k(complex z)
  {
  int i = pow(2,2);   // вызывается pow(int,int)
  double d = pow(2.0,2);   // вызывается
pow(double,double)
  complex z2 = pow(2,z);   // вызывается
pow(double,complex)
  complex z3 = pow(z,2);   // вызывается pow(complex,int)
  complex z4 = pow(z,z);   // вызывается
pow(complex,complex)
  }
```

Если найдены два сопоставления по самому приоритетному правилу, то вызов считается неоднозначным, а значит ошибочным.

Эти правила сопоставления параметров работают с учетом правил преобразований числовых типов для C и C++.

Неоднозначность вызова

Неоднозначность может появиться при:

- преобразовании типа;
- использовании параметров-ссылок;
- использовании аргументов по умолчанию.

Пример неоднозначности при преобразовании типа:

```
#include <iostream.h>
float f(float i){
   printf( "function float f(float i)/n“);
   return i;
}
double f(double i){
    printf( "function double f(double i)" );
   return i*2;
}
```

```
int main(){
    float x = 10.09;
    double y = 10.09;
     printf(“%f \n”, f(x));    // Вызывается f(float)
     printf(“%f \n”, f(y) );  // Вызывается f(double)
    /*    cout << f(10) << endl; Неоднозначность — как
    преобразовать 10: во  float  или  double? */
    return 0;
}
```

Для устранения этой неоднозначности требуется явное приведение типа для константы 10.

Пример неоднозначности при использовании аргументов по умолчанию:

```
#include <iostream.h>
int f(int a){return a;}
int f(int a, int b = 1){return a * b;}
int main(){
    printf("%d \n", f(10, 2)); // Вызывается  f(int, int)
   /* printf("%d \n", f(10));      Неоднозначность —
   что вызывается:  f(int, int) или f(int) ? */
   return 0;
}
```

Правила описания перегруженных функций.

- Перегруженные функции должны находиться *в одной области видимости*, иначе произойдет сокрытие аналогично одинаковым именам переменных во вложенных блоках.
- Перегруженные функции могут иметь *параметры по умолчанию*, при этом значения одного и того же параметра в разных функциях должны совпадать. В различных вариантах перегруженных функций может быть различное количество параметров по умолчанию.
- Функции не могут быть перегружены, если описание их параметров отличается только *модификатором* const *или использованием ссылки* (например, int и const int или int и int&).

Шаблоны функций

- Многие алгоритмы не зависят от типов данных, с которыми они работают (классический пример — сортировка).
- Естественно желание параметризовать алгоритм таким образом, чтобы его можно было использовать для различных типов данных.
- Первое, что может прийти в голову — передать информацию о типе в качестве параметра (например, одним параметром в функцию передается указатель на данные, а другим — длина элемента данных в байтах).

- Использование дополнительного параметра означает генерацию дополнительного кода, что снижает эффективность программы, особенно при рекурсивных вызовах и вызовах во внутренних циклах; кроме того, отсутствует возможность контроля типов.

- Другим решением будет написание для работы с различными типами данных нескольких перегруженных функций, но в таком случае в программе будет несколько одинаковых по логике функций, и для каждого нового типа придется вводить новую.

- В C++ есть мощное средство параметризации — шаблоны. Существуют шаблоны функций и шаблоны классов .
- С помощью шаблона функции можно определить алгоритм, который будет применяться к данным различных типов, а конкретный тип данных передается функции в виде параметра на этапе компиляции.
- Компилятор автоматически генерирует правильный код, соответствующий переданному типу. Таким образом, создается функция, которая автоматически перегружает сама себя и при этом не содержит накладных расходов, связанных с параметризацией.

Формат простейшей функции-шаблона:

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

Вместо слова Type может использоваться произвольное имя.

В общем случае шаблон функции может содержать несколько параметров, каждый из которых может быть не только типом, но и просто переменной, например:

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

Например, функция, сортирующая методом выбора массив из n элементов любого типа, в виде шаблона может выглядеть так:

```
template <class Type>
   void sort_vybor(Type *b, int n){
   Type a;   //буферная переменная для
   //обмена элементов

   for (int i = 0; i<n-1; i++){
    int imin = i;
    for (int j = i + 1; j<n; j++)
        if (b[j] < b[imin]) imin = j;
    a = b[i]; b[i] = b[imin]; b[imin] = a;
   }
}
```

Главная функция программы, вызывающей эту функцию-шаблон, может иметь вид:

```
#include <iostream.h>
template <class Type> void sort_vybor(Type *b, int n);
int main(){
   const int n = 20;
   int i, b[n];
   for (i = 0; i<n; i++) cin >> b[i];
   sort_vybor(b, n);      // Сортировка целочисленного массива
```

```
for (i = 0; i<n; i++) cout << b[i] << ' ';
    cout << endl;
    double a[] = {0.22, 117, -0.08, 0.21, 42.5};
    sort_vybor(a, 5); // Сортировка массива
                      //вещественных чисел
    for (i = 0; i<5; i++) cout << a[i] << ' ';
    return 0;
}
```

- Первый же вызов функции, который использует конкретный тип данных, приводит к созданию компилятором кода для соответствующей версии функции.
- Этот процесс называется инстанцированием шаблона (instantiation). Конкретный тип для инстанцирования либо определяется компилятором автоматически, исходя из типов параметров при вызове функции, либо задается явным образом.
- При повторном вызове с тем же типом данных код заново не генерируется. На месте параметра шаблона, являющегося не типом, а переменной, должно указываться константное выражение.

Пример явного задания аргументов шаблона при вызове:

```
template<class X, class Y, class Z> void f(Y, Z);
void g(){
    f<int, char*, double>("Vasia", 3.0);
    f<int, char*>("Vasia", 3.0);      // Z определяется
                                      //как double
    f<int>("Vasia", 3.0);             // Y определяется как
                                      //char*, а Z определяется как double
    // f("Vasia", 3.0);               ошибка: X
                                      //определить невозможно
}
```

- Чтобы применить функцию-шаблон к типу данных, определенному пользователем (структуре или классу), требуется перегрузить операции для этого типа данных, используемые в функции

- Как и обычные функции, шаблоны функций могут быть перегружены как с помощью шаблонов, так и обычными функциями.

Рекурсивные функции

- Рекурсивной называется функция, которая вызывает саму себя. Такая рекурсия называется *прямой*.
- Существует еще *косвенная* рекурсия, когда две или более функций вызывают друг друга. Если функция вызывает себя, в стеке создается копия значений ее параметров, как и при вызове обычной функции, после чего управление передается первому исполняемому оператору функции.
- При повторном вызове этот процесс повторяется.

- Для завершения вычислений каждая рекурсивная функция должна содержать хотя бы одну нерекурсивную ветвь алгоритма, заканчивающуюся оператором возврата.
- При завершении функции соответствующая часть стека освобождается, и управление передается вызывающей функции, выполнение которой продолжается с точки, следующей за рекурсивным вызовом.

- Классическим примером рекурсивной функции вычисление факториала. Для того чтобы получить значение факториала числа n, требуется умножить на n факториал числа (n-1).

- Известно также, что 0!=1 и 1!=1.

```
long fact(long n)
{
   if (n==0 || n==1) return 1;
   return (n * fact(n - 1));
}
```

- Рекурсивные функции чаще всего применяют для компактной реализации рекурсивных алгоритмов, а также для работы со структурами данных, описанными рекурсивно, например, с двоичными деревьями.
- Любую рекурсивную функцию можно реализовать без применения рекурсии, для этого программист должен обеспечить хранение всех необходимых данных самостоятельно.
- Достоинством рекурсии является компактная запись, а недостатками — расход времени и памяти на повторные вызовы функции и передачу ей копий параметров, и, главное, опасность переполнения стека.