ШАГ компьютерная АКАДЕМИЯ

# ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ С

# Урок №9

## Перегрузка функций и шаблоны функций

## Содержание

# 1. Встраивание

## Ключевое слово inline

В прошлом уроке мы познакомились с понятием функции. И выяснили, что как только программа встречает вызов функции, она сразу же обращается к телу данной функции и выполняет его. Этот процесс существенно сокращает код программы, но при этом увеличивает время ее выполнения за счет постоянных обращений к описанию конкретной вызванной функции. Но, бывает и не так. Некоторые функции в языке С можно определить с использованием специального служебного слова ***inline***.

Данный спецификатор позволяет определить функцию как встраиваемую, то есть подставляемую в текст программы в местах обращения к этой функции. Например, следующая функция определена как подставляемая:

```
inline float module(float x = 0, float y = 0)
{
    return sqrt(x * x + y * y);
}
```

Обрабатывая каждый вызов встраиваемой функции ***module***, компилятор подставляет на место ее вызова — в текст программы — код операторов тела функции. Тем самым при многократных вызовах подставляемой функции, размер программы может увеличиться, однако исключаются временные затраты на обращение к вызы-

ваемой функции и возврат из нее в основную функцию программы.

***Примечание:*** *Наиболее эффективно использовать подставляемые функции в тех случаях, когда тело функции состоит всего из нескольких операторов.*

Случается так, что компилятор не может определить функцию, как встраиваемую и просто игнорирует ключевое слово ***inline***. Перечислим причины, которые приводят к такому результату:

1. Слишком большой размер функции.
2. Функция является рекурсивной (с этим понятием вы познакомитесь в следующих уроках)
3. Функция повторяется в одном и том же выражении несколько раз
4. Функция содержит цикл, ***switch*** или ***if***.

Как видите — всё просто, но inline-функции не единственный способ встраивания. Об этом расскажет следующая тема урока.

## Раскрытие макроса

Помимо вызова функции, для встраивания в программу повторяющегося фрагмента используют, так называемое, раскрытие макроса. Для этих целей применяется директива препроцессора ***#define***, со следующим синтаксисом:

```
#define Имя_макроса(Параметры) (Выражение)

#include <iostream>
#define SQR(X) ((X) * (X))
```

```
#define CUBE(X) (SQR(X)*(X))
#define ABS(X) (((X) < 0)? -(X) : X)

using namespace std;
void main()
{
    y = SQR(t + 8) - CUBE(t - 8);
    cout <<sqrt(ABS(y));
}
```

1. С помощью директивы ***#define*** объявляются три макроса ***sqr(x)***, ***cube(x)*** и ***abs(x)***.
2. В функции main происходит вызов вышеописанных макросов по имени.
3. Препроцессор раскрывает макрос (т.е. подставляет на место вызова выражение из директивы ***#define***) и передает получившийся текст компилятору.
4. После встраивания выражение в ***main*** выглядит для программы таким образом:

```
y = ((t+8) * (t+8)) - ((((t-8)) * (t-8)) * (t-8));
cout << sqrt(((y < 0)? -(y) : y));
```

***Примечание:*** *Следует обратить внимание на использование скобок при объявлении макроса. С помощью них мы избегаем ошибок в последовательности вычислений. Например:*

```
#define SQR(X) X * X
y = SQR(t + 8); //раскроет макрос t+8*t+8
```

В примере при вызове макроса SQR сначала выполнится умножение ***8*** на ***t***, а потом к результату прибавится значение переменной ***t*** и восьмерка, хотя очевидно, что нашей целью было получение квадрата суммы ***t+8***.

Теперь вы полностью знакомы с понятием встраивания и можете использовать его в своих программах.

# 2. Перегрузка функций

Каждый раз, когда мы изучаем, новую тему, нам важно узнать, в чем заключается применение наших знаний на практике. Цель перегрузки функций состоит в том, чтобы несколько функций обладая одним именем, по-разному выполнялись и возвращали разные значения при обращении к ним с разными по типам и количеству фактическими параметрами.

Например, может потребоваться функция, возвращающая максимальное из значений элементов одномерного массива, передаваемого ей в качестве параметра. Массивы, используемые как фактические параметры, могут содержать элементы разных типов, но пользователь функции не должен беспокоиться о типе результата. Функция всегда должна возвращать значение того же типа, что и тип массива — фактического параметра.

Для реализации перегрузки функций необходимо для каждого имени определить, сколько разных функций связано с ним, т.е. сколько вариантов вызовов допустимы при обращении к ним. Предположим, что функция выбора максимального значения элемента из массива должна работать для массивов типа ***int***, ***long***, ***float***, ***double***. В этом случае придется написать четыре разных варианта функции с одним и тем же именем. В нашем примере эта задача решена следующим образом:

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

```
long max_element(int n, int array[])
// Функция для массивов с элементами типа int.
{
    int value = array[0];
    for (int i = 1; i < n; i++)
        value = value > array [i] ? value : array [i];
    cout << "\nFor (int)    : ";
    return long(value);
}

long max_element(int n, long array[])
// Функция для массивов с элементами типа long.
{
    long value = array[0];
    for (int i = 1; i < n; i++)
        value = value > array[i] ? value : array[i];
        cout << "\nFor (long) : ";
        return value;
}

double max_element(int n, float array[])
// Функция для массивов с элементами типа float.
{
    float value = array[0];
    for (int i = 1; i < n; i++)
        value = value > array[i] ? value : array[i];
    cout << "\nFor (float)  : ";
    return double(value);
}
double max_element(int n, double array[])
// Функция для массивов с элементами типа double.
{
    double value = array[0];
    for (int i = 1; i < n; i++)
        value = value > array[i] ? value : array[i];
    cout << "\nFor (double) : ";
    return value;
}
```

```
void main()
{
    int x[] = { 10, 20, 30, 40, 50, 60 };
    long f[] = { 12L, 44L, 5L, 22L, 37L, 30L };
    float y[] = { 0.1, 0.2, 0.3, 0.4, 0.5, 0.6 };
    double z[] = { 0.01, 0.02, 0.03, 0.04, 0.05, 0.06 };
    cout << "max_elem(6,x) = " << max_element(6,x);
    cout << "max_elem(6,f) = " << max_element(6,f);
    cout << "max_elem(6,y) = " << max_element(6,y);
    cout << "max elem(6,z) = " << max_element(6,z);
}
```

1. В программе мы показали независимость перегруженных функций от типа возвращаемого значения. Две функции, обрабатывающие целые массивы (***int***, ***long***), возвращают значение одного типа ***long***. Две функции, обрабатывающие вещественные массивы (***double***, ***float***), обе возвращают значение типа ***double***.
2. Распознавание перегруженных функций при вызове выполняется по их параметрам. Перегруженные функции должны иметь одинаковые имена, но спецификации их параметров должны различаться по количеству и (или) по типам, и (или) по расположению.

   ***ВНИМАНИЕ!!!*** *При использовании перегруженных функций нужно с осторожностью задавать начальные значения их параметров. Предположим, мы следующим образом определили перегруженную функцию умножения разного количества параметров:*

```
double multy (double x) {
    return x * x * x;
}

double multy (double x, double y) {
    return x * y * y;
}

double multy (double x, double y, double z)
{
    return x * y * z;
}
```

Каждое из следующих обращений к функции ***multy()*** будет однозначно идентифицировано и правильно обработано:

```
multy (0.4)
    multy (4.0, 12.3)
    multy (0.1, 1.2, 6.4)
```

Однако, добавление в программу функции с начальными значениями параметров:

```
double multy (double a = 1.0, double b  = 1.0,
              double c = 1.0, double d = 1.0)
{
    return a * b + c * d;
}
```

Навсегда запутает любой компилятор при попытках обработать, например, такой вызов, что приведет к ошибке на этапе компиляции. Будьте внимательны!!!

```
multy(0.1, 1.2);
```

Следующий раздел урока расскажет вам об альтернативном решении, позволяющем создать универсальную функцию.

# 3. Шаблоны функций

Шаблоны функций в языке С позволяют создать общее определение функции, применяемой для различных типов данных.

В прошлой теме, чтобы использовать одну и ту же функцию с различными типами данных мы создавали отдельную перегруженную версию этой функции для каждого типа. Например:

```
int Abs(int N){
    return N < 0 ? -N : N;
}

double Abs(double N){
    return N < 0. ? -N : N;
}
```

Теперь, используя шаблон мы сможем реализовать единственное описание, обрабатывающее значения любого типа:

```
template <typename T> T Abs (T N)
{
    return N < 0 ? -N : N;
}
```

Теперь — обсудим то, что у нас получилось.

1. Идентификатор ***T*** является ***параметром типа.*** Именно он определяет тип параметра, передаваемого в момент вызова функции.

2. Допустим, программа вызывает функцию ***Abs*** и передает ей значения типа ***int***:

```
cout << "Result - 5 = " << Abs(-5);
```

3. В данном случае, компилятор автоматически создает версию функции, где вместо ***T*** подставляется тип ***int***.
4. Теперь функция будет выглядеть так:

```
int Abs (int N)
{
    return N < 0 ? -N : N;
}
```

5. Следует отметить, что компилятор создаст версии функции для любого вызова, с любым типом данных. Такой процесс носит название — ***создание экземпляра*** шаблона функции.

## Основные принципы и понятия при работе с шаблоном

Теперь, после поверхностного знакомства — мы рассмотрим все особенности работы шаблонов:

1. При определении шаблона используются два спецификатора: ***template*** и ***typename***.
2. На место параметра типа T можно подставить любое корректное имя.
3. В угловые скобки можно записывать больше одного параметра типа.
4. ***Параметр функции*** — это значение, передаваемое в функцию при выполнении программы.

5. ***Параметр типа*** — указывает тип аргумента, передаваемого в функцию, и обрабатывается только при компиляции.

## Процесс компиляции шаблона

1. Определение шаблона не вызывает генерацию кода компилятором самостоятельно. Последний создает код функции только в момент её вызова и генерирует при этом соответствующую версию функции.
2. Следующий вызов с теми же типами данных параметров не спровоцирует генерацию дополнительной копии функции, а вызовет ее уже существующую копию.
3. Компилятор создает новую версию функции, только если тип переданного параметра не совпадает ни с одним из предыдущих вызовов.

### *Пример работы с шаблоном*

```
template <typename T> T Max (T A, T B)
{
  return A > B ? A : B;
}
```

1. Шаблон генерирует множество функций, возвращающих большее из двух значений с одинаковым типом данных.
2. Оба параметра определены как параметры типа ***T*** и при вызове функции передаваемые параметры должны быть строго одного типа. В данном случае возможны такие вызовы функции:

```
cout << "Большее из 10 и 5 = " << Max(10, 5) << "\n";
cout << "Большее из 'A' и 'B' = " << Max('A', 'B') << "\n";
cout << "Большее из 3.5 и 5.1 = " << Max(3.5, 5.1) << "\n";
```

А такой вызов приведет к ошибке:

```
cout << "Большее из 10 и 5.55 = " << Max(10, 5.55);
// ОШИБКА!
```

Компилятор не сможет преобразовать параметр ***int*** в ***double***. Решением проблемы передачи разных параметров является такой шаблон:

```
template <typename T1, typename T2> T2 Max(T1 A , T2 B)
{
    return A > B ? A : B;
}
```

В этом случае ***T1*** обозначает тип значения, передаваемого в качестве первого параметра, а ***T2*** — второго.

> ***ВНИМАНИЕ!!!*** *Каждый параметр типа, встречающийся внутри угловых скобок, должен ОБЯЗАТЕЛЬНО появляться в списке параметров функции. В противном случае произойдет ошибка на этапе компиляции.*

```
template <typename T1, typename T2> T1 Max(T1 A , T1 B)
{
    return A > B ? A : B;
}
// ОШИБКА! список параметров должен включать T2 как
// параметр типа.
```

Переопределение шаблонов функций

1. Каждая версия функции, генерируемая с помощью шаблона, содержит один и тот же фрагмент кода.
2. Однако, для отдельных параметров типа можно обеспечить особую реализацию кода, т.е. определить обычную функцию с тем же именем, что и шаблон.
3. Обычная функция переопределит шаблон. Если компилятор находит типы переданных параметров соответствующие спецификации обычной функции, то он вызовет ее, и не создает функцию по шаблону.

# 4. Домашнее задание

1. Написать шаблон функции для поиска среднего арифметического значений массива.
2. Написать перегруженные шаблоны функций для нахождения корней линейного ***(a*x + b = 0)*** и квадратного ***(a*x2+b*x + c = 0)*** уравнений. Замечание: в функции передаются коэффициенты уравнений.
3. Написать функцию, которая принимает в качестве параметров вещественное число и количество знаков после десятичной точки, которые должны остаться. Задачей функции является округление вышеуказанного вещественного числа с заданной точностью


# Урок №9

# Перегрузка функций и шаблоны функций