



南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

C/C++ Program Design

Lab 7, function overloading & function
template

廖琪梅, 王大兴



Functions Overloading & Template

- Function overloading
- Function template
- Recursive function
- Pointers to functions



Inline Function

C++ provides **inline functions** to help reduce function-call overhead(to avoid a function call). You should place the qualifier **inline** before return type in the **function prototype**.

Default Arguments

Default arguments must be specified in the **function prototype** and must be **rightmost(trailing)**.



Function Overloading

Function overloading is used to create several functions of the same name that perform similar tasks, but of different data types. The C++ compiler selects the proper function to call by examining the number, types and order of the arguments.

- 1.the same function name
- 2.different parameter list



Three ways to overload functions:

1. Number of parameters

```
int add(int, int);  
int add(int, int, int);
```

2. Data type of parameters

```
int add(int, int);  
float add(float, float);
```

3. Sequence of data type of parameters

```
float add(float, int);  
float add(int, float);
```

A function with default arguments omitted might be called identically to another overloaded function, which causes a compilation error.

Use caution when overloading functions with default parameters, because this may cause ambiguity.

Note: The same function signature but different return type is not a valid function overloading example. This will throw compilation error.

```
int add(int, int);  
float add(int, int);
```



Function Templates

The syntax of templates:

```
template <typename T>    // This is the template parameter declaration
T Max(T x, T y)
{
    return (x > y? x : y);
}
```

- Starts with the keyword **template**
- You can also use keyword **class** instead of **typename**
- **T** is a template argument that accepts different data types

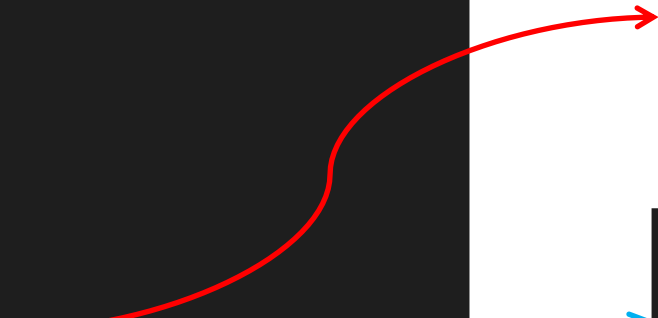


compile internally generates and adds right code respectively.


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

int main()
{
    cout << "Max int = " << Max<int>(3,7) << endl;
    cout << "Max char = " << Max<char>('g','e') << endl;
    cout << "Max double = " << Max<double>(3.1,7.9) << endl;

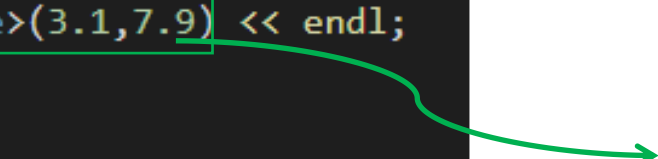
    return 0;
}
```



```
int Max(int x, int y)
{
    return (x > y? x : y);
}
```



```
char Max(char x, char y)
{
    return (x > y? x : y);
}
```



```
double Max(double x, double y)
{
    return (x > y? x : y);
}
```

output:

```
Max int = 7
Max char = g
Max double = 7.9
```



When you declare or define a function template, **template <typename T>** or **template <class T>** can not be omitted.

When you declare or define several function templates, every function must include **template <typename T>** or **template <class T>** before function header.

Template functions can also be overloaded.

Overloaded template functions

```
template <typename T>
void Swap(T &a, T &b)
{
    T temp;
    temp = a;
    a = b;
    b = temp;
}

template <typename T>
void Swap(T a[], T b[], int n)
{
    T temp;
    for (int i = 0; i < n; i++)
    {
        temp = a[i];
        a[i] = b[i];
        b[i] = temp;
    }
}

void Show(int a[])
{
    using namespace std;
    cout << a[0] << a[1] << "/";
    cout << a[2] << a[3] << "/";
    for (int i = 4; i < Lim; i++)
        cout << a[i];

    cout << endl;
}
```

Overloaded
template
functions

```
// using overloaded template functions
#include <iostream>

template <typename T> // original template
void Swap(T &a, T &b);

template <typename T> // new template
void Swap(T *a, T *b, int n);

void Show(int a[]);
const int Lim = 8;

int main()
{
    using namespace std;
    int i = 10, j = 20;
    cout << "i, j = " << i << ", " << j << ".\n";
    cout << "Using compiler-generated int swapper:\n";
    Swap(i,j); // matches original template
    cout << "Now i, j = " << i << ", " << j << ".\n";

    int d1[Lim] = {0,7,0,4,1,7,7,6};
    int d2[Lim] = {0,7,2,0,1,9,6,9};
    cout << "Original arrays:\n";
    Show(d1);
    Show(d2);

    Swap(d1,d2,Lim); // matches new template
    cout << "Swapped arrays:\n";
    Show(d1);
    Show(d2);

    return 0;
}
```

Function
prototype

Output:

```
i, j = 10, 20.
Using compiler-generated int swapper:
Now i, j = 20, 10.
Original arrays:
07/04/1776
07/20/1969
Swapped arrays:
07/20/1969
07/04/1776
```



Recursive function

A function that **calls itself** is known as **recursive function**. And, this technique is known as **recursion**. **Recursion** is used to solve various mathematical problems by dividing it into smaller problems.

In recursive function, you must give the **base case** or **stopping condition** to stop the recursive call. Usually, **if statement** is used to indicate the base case.



Disadvantages of Recursion:

- **Recursive programs are generally slower than nonrecursive programs.** Because it needs to make a function call so the program must save all its current state and retrieve them again later. This consumes more time making recursive programs slower.
- **Recursive programs requires more memory to hold intermediate states in a stack.** Non recursive programs don't have any intermediate states, hence they don't require any extra memory.



Pointer to Function(Function Pointer)

Normally, a function pointer is used as a parameter. When you invoke the function, the corresponding argument is the function name.

Note: Do not omit the **()** of the pointer when you declare a function pointer.

Example:

`int findmax(int, int);` ← Declaring a function

`int (*funptr)(int,int);` ← Declaring a pointer to a function

`funptr = findmax;` ← Assigning the address of a function to the pointer

`int max = funptr(3,5);` ← Calling the function by the pointer



Example:
Compute the definite integral, suppose
calculate the following definite integrals

$$\int_a^b f(x)dx = (b-a)/2 * (f(a) + f(b))$$

$$\int_0^1 x^2 dx$$

$$\int_1^2 \sin x / x dx$$

```
#include <iostream>
#include <cmath>
using namespace std;
double calc(double (*funp)(double), double a, double b);
double f1(double x1);
double f2(double x2);

int main()
{
    double result;
    double (*funp)(double);

    result = calc(f1, a: 0.0, b: 1.0);
    cout<<"1: result= " << result << endl;
    funp = f2;
    result = calc(funp, a: 1.0, b: 2.0);
    cout<<"2: result= " << result << endl;

    return 0;
}
```

function pointer as a parameter

Declaring a function pointer

Calling the function by function name

Assigning the address of function f2 to the pointer

Calling the function by function pointer



$$\int_a^b f(x)dx = (b-a)/2 * (f(a) + f(b))$$

```
double calc ( double (*funp)(double), double a, double b )
{
    double z;
    z = (b-a) / 2 * ( (*funp)(a) + (*funp)(b) );
    return ( z );
}

double f1 ( double x )
{
    return (x * x);
}

double f2 ( double x )
{
    return (sin(x) / x);
}
```

$$\int_0^1 x^2 dx$$

$$\int_1^2 \sin x / x dx$$

Output:

1: result= 0.5

2: result= 0.64806



qsort() in general utilities library stdlib.h

The quick sort method is one of the most effective sorting algorithms. **qsort()** function sorts an array of data object.

void qsort(void *base, size_t nmemb, size_t size, int(*compar)(const void *, const void *));

void *base: pointer to the beginning of the array to be sorted, it permits any data pointer type to be typecast to a pointer-to-void.

size_t nmemb: number of items to be sorted.

size_t size: the size of the data object, for example, if you want to sort an array of double, you would sizeof(double).

int (*compar)(const void *, const void *): a pointer to a function that returns an **int** and take two arguments, each of which is a pointer to type const void. These two pointers point to the items being compared.



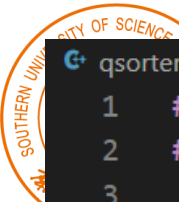
```
qsorter.cpp > showarray(const double [], int)
1 // using qsort() to sort groups of numbers
2 #include <iostream>
3 #include <stdlib.h>
4
5 #define NUM 10
6 void fillarray(double ar[], int n);
7 void showarray(const double ar[], int n);
8 int mycomp(const void *p1, const void *p2);
9
10 int main()
11 {
12     double vals[NUM];
13
14     fillarray(vals, NUM);
15     std::cout << "Random list:\n";
16     showarray(vals, NUM);
17
18     qsort(vals, NUM, sizeof(double), mycomp);
19     std::cout << "\nSorted list:" << std::endl;
20     showarray(vals, NUM);
21
22     return 0;
23 }
24
```

```
25 void fillarray(double ar[], int n)
26 {
27     for(int i = 0; i < n; i++)
28         ar[i] = (double)rand() / ((double)rand() + 0.1);
29 }
30
31 void showarray(const double ar[], int n)
32 {
33     for(int i = 0; i < n; i++)
34         std::cout << ar[i] << " ";
35     std::cout << std::endl;
36 }
37
38 int mycomp(const void *p1, const void *p2)
39 {
40     // need to use pointers to double to access values
41     const double *pd1 = (const double *) p1;
42     const double *pd2 = (const double *) p2;
43
44     if(*pd1 < *pd2)
45         return -1;
46     else if(*pd1 > *pd2)
47         return 1;
48     else
49         return 0;
50 }
```

give the sorting rule

```
Random list:
2.13039 0.980787 4.61474 0.436358 0.501426 0.759134 0.710526 1.03933 0.88626 0.233295

Sorted list:
0.233295 0.436358 0.501426 0.710526 0.759134 0.88626 0.980787 1.03933 2.13039 4.61474
```

```

qsorter2.cpp > main()
1  #include <iostream>
2  #include <cstring>
3
4  using namespace std;
5  #define SIZE 5
6
7  struct student
8  {
9      char name[20];
10     int age;
11 };
12
13 void display(const student *s,int n);
14 int mycomp(const void *p1, const void *p2);
15
16 int main()
17 {
18     student stu[SIZE] = {"Alice",19},{"Bob",20},{"Alice",16},{"Leo",20},{"Billy",19}};
19
20     cout << "Original students:\n";
21     display(stu,SIZE);
22
23     qsort(stu,SIZE,sizeof(student),mycomp);
24     cout << "\nSorted students:" << endl;
25     display(stu,SIZE);
26
27     return 0;
28 }

```

```

30 void display(const student *s,int n)
31 {
32     for(int i = 0; i < n; i++)
33     {
34         cout << "Name: " << s[i].name << ", age: " << s[i].age << endl;
35     }
36 }

```

```

Original students:
Name: Alice, age: 19
Name: Bob, age: 20
Name: Alice, age: 16
Name: Leo, age: 20
Name: Billy, age: 19

Sorted students:
Name: Alice, age: 16
Name: Alice, age: 19
Name: Billy, age: 19
Name: Bob, age: 20
Name: Leo, age: 20

```



```
38  int mycomp(const void *p1, const void *p2)
39  {
40      // need to use pointers to struct student to access values
41      const student *ps1 = (const student *) p1;
42      const student *ps2 = (const student *) p2;
43
44      int res;
45      res = strcmp(ps1->name, ps2->name);
46      if(res != 0)
47          return res;
48      else
49      {
50          if(ps1->age < ps2->age)
51              return -1;
52          else if(ps1->age > ps2->age)
53              return 1;
54          else
55              return 0;
56      }
57
58 }
```

If the name is the same, sort by age





Exercise 1

Define a default arguments function to display a square of any character.

void displaySquare(int side, char filledCharacter);

For example, if *side* is 5, *filledCharacter* is '#', the function displays as follows:

```
#####  
#####  
#####  
#####  
#####
```

In default case, *side* is 4, *filledCharacter* is '*'.

Write a test program to call the displaySquare function using default arguments and non-default arguments respectively and show the result.



Exercise 2

Overload a function `int vabs(int * p, int n)` which can calculate the sum of the absolute values of the elements in an array, the array can be `int`, `float` and `double`.

Should `n` be `int` or `size_t`? what's the difference?



Exercise 3

Write a program that uses a function template called ***minimum*** to determine the smaller of two arguments. Test the program using integer, character and floating-point number arguments.