



南方科技大学  
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

# C/C++ Program Design

## CS205

**Prof. Shiqi Yu (于仕琪)**

yusq@sustech.edu.cn

<http://faculty.sustech.edu.cn/yusq/>



南方科技大学  
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

# Default Arguments



# Default arguments

- A feature in C++ (not C).
- To call a function without providing one or more trailing arguments.

```
float norm(float x, float y, float z);  
float norm(float x, float y, float z = 0);  
float norm(float x, float y = 0, float z);
```

Parameter-list of a function  
declaration.



```
int main()  
{  
    cout << norm(3.0f) << endl;  
    cout << norm(3.0f, 4.0f) << endl;  
    cout << norm(3.0f, 4.0f, 5.0f) << endl;  
    return 0;  
}
```

```
float norm(float x, float y, float z)  
{  
    return sqrt(x * x + y * y + z * z);  
}
```

default-argument.cpp



南方科技大学  
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

# Function Overloading



# Why to overload?

- C99

Which one do you prefer?

<math.h>

```
double    round (double x);  
float     roundf (float x);  
long double roundl (long double x);
```

- C++11

<cmath>

```
double    round (double x);  
float     round (float x);  
long double round (long double x);
```



# Function overloading

- Which function to choose? The compiler will perform name lookup.
- Argument-dependent lookup, also known as ADL.
- The return type will not be considered in name lookup.

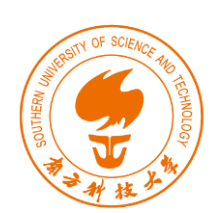
```
int sum(int x, int y)
{
    cout << "sum(int, int) is called" << endl;
    return x + y;
}
float sum(float x, float y)
{
    cout << "sum(float, float) is called" << endl;
    return x + y;
}
double sum(double x, double y)
{
    cout << "sum(double, double) is called" << endl;
    return x + y;
}
```

overload.cpp



南方科技大学  
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

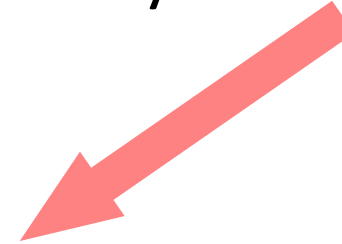
# Function Templates



# Why function templates

- The definitions of some overloaded functions may be similar.

```
int sum(int x, int y)
{
    cout << "sum(int, int) is called" << endl;
    return x + y;
}
float sum(float x, float y)
{
    cout << "sum(float, float) is called" << endl;
    return x + y;
}
double sum(double x, double y)
{
    cout << "sum(double, double) is called" << endl;
    return x + y;
}
```







# Explicit instantiation

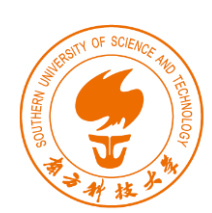
- A function template is not a type, or a function, or any other entity.
- No code is generated from a source file that contains only template definitions.
- The template arguments must be determined, then the compiler can generate an actual function

```
template<typename T>
T sum(T x, T y)
{
    cout << "The input type is " << typeid(T).name() << endl;
    return x + y;
}
// instantiates sum<double>(double, double)
template double sum<double>(double, double);

// instantiates sum<char>(char, char), template argument deduced
template char sum<>(char, char);

// instantiates sum<int>(int, int), template argument deduced
template int sum(int, int);
```

template1.cpp



# Implicit instantiation

- Implicit instantiation occurs when a function template is not explicitly instantiated.

```
template<typename T>
T product(T x, T y)
{
    cout << "The input type is " << typeid(T).name() << endl;
    return x * y;
}
// Implicitly instantiates product<int>(int, int)
cout << "product = " << product<int>(2.2f, 3.0f) << endl;
// Implicitly instantiates product<float>(float, float)
cout << "product = " << product(2.2f, 3.0f) << endl;
```



# Function template specialization

- We have a function template:

```
template<typename T> T sum(T x, T y)
```

- If the input type is Point

```
struct Point
{
    int x;
    int y;
};
```

- But no + operator for Point
- We need to give a special definition for this case.



南方科技大学  
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

# Function Pointers and References



# Function pointers

- `norm_ptr` is a pointer, a function pointer.
- The function should have two `float` parameters, and returns `float`.

```
float norm_l1(float x, float y);
```

```
float norm_l2(float x, float y);
```

```
float (*norm_ptr)(float x, float y);
```

```
norm_ptr = norm_l1; //Pointer norm_ptr is pointing to norm_l1
```

```
norm_ptr = &norm_l2; //Pointer norm_ptr is pointing to norm_l2
```

```
float len1 = norm_ptr(-3.0f, 4.0f); //function invoked
```

```
float len2 = (*norm_ptr)(-3.0f, 4.0f); //function invoked
```



# Function pointers

- A function pointer can be an argument and pass to a function.

<stdlib.h>

```
void qsort( void *ptr, size_t count, size_t size,  
           int (*comp)(const void *, const void *) );
```

- To sort some customized types, such as
  - struct Point
  - struct Person



# Function references

```
float norm_l1(float x, float y); //declaration  
float norm_l2(float x, float y); //declaration  
float (&norm_ref)(float x, float y) = norm_l1; //norm_ref is a function reference
```



南方科技大学  
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

# Recursive Functions





# Recursive Functions

- A simple example

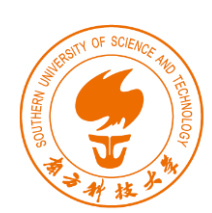
```
int main()
{
    div2(1024.); // call the recursive function
    return 0;
}
```

```
void div2(double val)
{
    cout << "Entering val = " << val << endl;

    if (val > 1.0)
        div2( val / 2); // function calls itself
    else
        cout << "-----" << endl;

    cout << "Leaving val = " << val << endl;
}
```

[recursion.cpp](#)



# Recursive Functions

- Pros.
  - Good at tree traversal
  - Less lines of source code
- Cons.
  - Consume more stack memory
  - May be slow.
  - Difficult to implement and debug