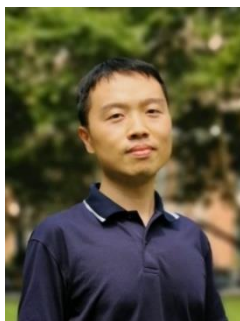




# 程序设计II



张书航 助理教授

电子邮件: [zhangsh52@mail.sysu.edu.cn](mailto:zhangsh52@mail.sysu.edu.cn)

个人主页: [shuhangz.github.io](http://shuhangz.github.io)



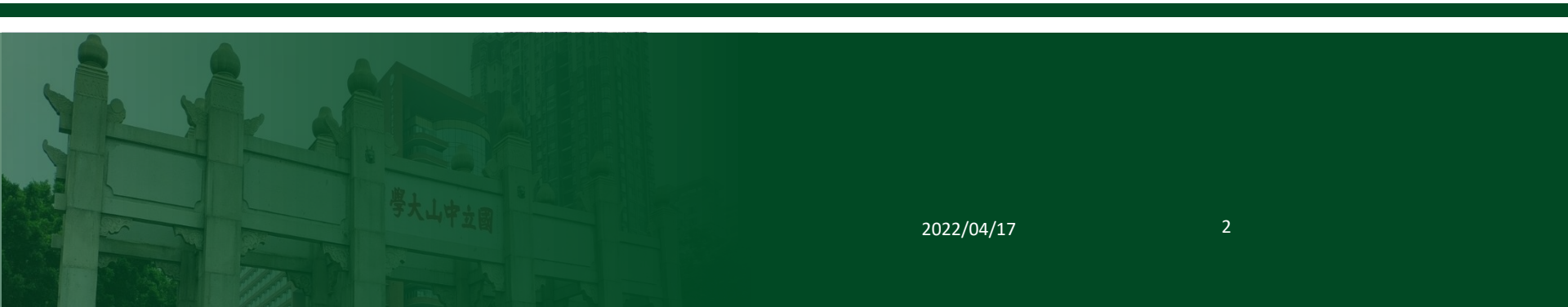
李同文 助理教授

电子邮件: [litw8@mail.sysu.edu.cn](mailto:litw8@mail.sysu.edu.cn)



# C++ 内容简介

## 程序设计II



**继承**--C语言的精髓

*高效性、灵活性;*

**扩充**--面向对象机制

*适合开发大型软件;*

**弥补**--C语言的缺点

***类型更安全(编译器查错能力更强);***  
***支持代码重用;***



# C++在非面向对象方面的特性

C++标准库、头文件、类型转换、常数、bool、enum、结构体、字符串、引用、inline函数、默认参数等

**本章知识点太多？**

许多知识点目前只需要了解，会在后面的课程中详细讨论



- C++程序由称为**类**和**函数**的片段组成。
  - 可以对每一个你需要的片段进行编程，以形成一个C++程序
  - 相反，大多数C++程序员利用C++**标准库**中现有的丰富的类和函数集合
- 因此，学习C++的“世界”其实有两个部分。
  - 第一是学习C++语言本身
  - 第二是学习如何使用C++标准库中的类和函数
- “**不要重复造轮子**”

- C++程序由称为**类**和**函数**的片段组成。
  - 可以对每一个你需要的片段进行编程，以形成一个C++程序
  - 相反，大多数C++程序员利用**C++标准库**中现有的丰富的类和函数集合
- 因此，学习C++的“世界”分。
  - 第一是学习C++语言本身
  - 第二是学习如何使用C++
- “不要重复造轮子”



- 不要重复造轮子
  - 软件重用 (software reuse) 是面向对象编程的核心
  - 团队协作编程: 编程像搭积木一样, 用好标准库, 以及自己、合作者编写的类和函数
- 自己写函数和类
  - 好处: 你会知道它们是如何工作的, 可以检查C++代码。
  - 缺点: 在设计、开发和维护新的函数和类时, 要耗费大量的时间和复杂的精力, 来保证其正确和有效地运行。



## Software Engineering Observation 15.1

*Use a “building-block” approach to create programs. Avoid reinventing the wheel. Use existing pieces wherever possible. Called **software reuse**, this practice is central to object-oriented programming.*



## Software Engineering Observation 15.2

*When programming in C++, you typically will use the following building blocks: classes and functions from the C++ Standard Library, classes and functions you and your colleagues create and classes and functions from various popular third-party libraries.*

# C++ 常用<头文件>

C++风格的引用只有尖括号，没有扩展名。

C++ Standard  
Library header  
file

Explanation

<iostream>

Contains function prototypes for the C++ standard input and output functions, introduced in Section 15.3, and is covered in more detail in Chapter 21, Stream Input/Output: A Deeper Look.

<iomanip>

Contains function prototypes for stream manipulators that format streams of data. This header is first used in Section 15.15 and is discussed in more detail in Chapter 21.

<cmath>

Contains function prototypes for the math library functions.

<cstdlib>

Contains function prototypes for conversions of numbers to text, text to numbers, memory allocation, random numbers and various other utility functions. Portions of the header are covered in Chapter 18, Operator Overloading; Class string and Chapter 22, Exception Handling: A Deeper Look.

<ctime>

Contains function prototypes and types for manipulating the time and date.

**Fig. 15.2** | C++ Standard Library header files. (Part I of 4.)





C++风格的引用只有尖括号，没有扩展名。

include加不加尖括号？

- `#include <file>` — system include files
- `#include "file"` — local include files

将file的内容复制到当前文件中

```
1 #include "some_file.hpp"
2 // We can use contents of file "some_file.hpp" now.
3 int main() { return 0; }
```

<cmath>

Contains function prototypes for the math library functions.

<cstdlib>

Contains function prototypes for conversions of numbers to text, text to numbers, memory allocation, random numbers and various other utility functions. Portions of the header are covered in Chapter 18, Operator Overloading; Class string and Chapter 22, Exception Handling: A Deeper Look.

<ctime>

Contains function prototypes and types for manipulating the time and date.

**Fig. 15.2** | C++ Standard Library header files. (Part I of 4.)



## C++ Standard Library header file

## Explanation

<code>&lt;array&gt;</code> , <code>&lt;vector&gt;</code> , <code>&lt;list&gt;</code> , <code>&lt;forward_list&gt;</code> , <code>&lt;deque&gt;</code> , <code>&lt;queue&gt;</code> , <code>&lt;stack&gt;</code> , <code>&lt;map&gt;</code> , <code>&lt;unordered_map&gt;</code> , <code>&lt;unordered_set&gt;</code> , <code>&lt;set&gt;</code> , <code>&lt;bitset&gt;</code>	These headers contain classes that implement the C++ Standard Library containers. Containers store data during a program's execution. The <code>&lt;vector&gt;</code> header is first introduced in Section 15.15.
<code>&lt;cctype&gt;</code>	Contains function prototypes for functions that test characters for certain properties (such as whether the character is a digit or a punctuation), and function prototypes for functions that can be used to convert lowercase letters to uppercase letters and vice versa.
<code>&lt;cstring&gt;</code>	Contains function prototypes for C-style string-processing functions. This header is used in Chapter 18.
<code>&lt;typeinfo&gt;</code>	Contains classes for runtime type identification (determining data types at execution time). This header is discussed in Section 20.8.

**Fig. 15.2** | C++ Standard Library header files. (Part 2 of 4.)



## C++ Standard Library header file

## Explanation

<code>&lt;exception&gt;</code> , <code>&lt;stdexcept&gt;</code>	These headers contain classes that are used for exception handling (discussed in Chapter 22).
<code>&lt;memory&gt;</code>	Contains classes and functions used by the C++ Standard Library to allocate memory to the C++ Standard Library containers. This header is used in Chapter 22.
<code>&lt;fstream&gt;</code>	Contains function prototypes for functions that perform input from and output to files on disk.
<code>&lt;string&gt;</code>	Contains the definition of class <code>string</code> from the C++ Standard Library.
<code>&lt;sstream&gt;</code>	Contains function prototypes for functions that perform input from strings in memory and output to strings in memory.
<code>&lt;functional&gt;</code>	Contains classes and functions used by C++ Standard Library algorithms.
<code>&lt;iterator&gt;</code>	Contains classes for accessing data in the C++ Standard Library containers.
<code>&lt;algorithm&gt;</code>	Contains functions for manipulating data in C++ Standard Library containers.
<code>&lt;cassert&gt;</code>	Contains macros for adding diagnostics that aid program debugging.

**Fig. 15.2** | C++ Standard Library header files. (Part 3 of 4.)



## C++ Standard Library header file

### Explanation

<code>&lt;cmath&gt;</code>	Contains the floating-point size limits of the system.
<code>&lt;climits&gt;</code>	Contains the integral size limits of the system.
<code>&lt;cstdio&gt;</code>	Contains function prototypes for the C's standard I/O functions.
<code>&lt;locale&gt;</code>	Contains classes and functions normally used by stream processing to process data in the natural form for different languages (e.g., monetary formats, sorting strings, character presentation, etc.).
<code>&lt;limits&gt;</code>	Contains classes for defining the numerical data type limits on each computer platform.
<code>&lt;utility&gt;</code>	Contains classes and functions that are used by many C++ Standard Library headers.

**Fig. 15.2** | C++ Standard Library header files. (Part 4 of 4.)



- 创建自定义的头文件。
  - 自定义的头文件应该以.h结尾。
  - 可以通过使用#include预处理指令来包含程序员定义的头文件。
  - 例如，通过在程序的开头放置指令#include "square.h"，可以在程序中包含头文件square.h。

用来冻结所修饰类型的标识符，使其不能更改单元的内容。

定义形式：

T const 或 const T

如：`int const a=5;`  
等价于 `const int a=5;`  
`cout << a ;`  
`b = a +10;`  
`a = 2; X`

## 1. 用于常量定义:

### 整型常量

```
int const N=5;
```

类似于变量定义

- (1) `int a[N]; //正确`
- (2) 修饰符, 派生新类型
- (3) 分配存储单元存值, 表达式或值符合类型限制
- (4) 使用与普通变量一样  
系统严格的类型检查

### 符号常量

```
#define N 5
```

符号常量大写字母

```
int a[N]; //正确
```

没有类型

没有存储单元  
内容任意

预编译时完成字符替换  
不安全型

(机械替换易出错)



```
const int*  
int const *  
int* const
```

- **const**默认作用于其左边的东西，如果左边没有东西，则作用于其右边：
  - `const` applies to the thing left of it. If there is nothing on the left then it applies to the thing right of it.



## • `const int*`

- `const`只有右边有东西，所以`const`修饰`int`成为常量整型，然后`*`再作用于常量整型。
- 所以这是a pointer to a constant integer (指向一个整型，不可通过该指针改变其指向的内容，但可改变指针本身所指向的地址)

## • `int const *`

- `const`左边有东西，所以`const`作用于`int`，`*`再作用于`int const`所以这还是 a pointer to a constant integer (同上)

## • `int* const`

- `const`作用于指针 (不可改变指向的地址)，所以这是a constant pointer to an integer，可以通过指针改变其所指向的内容但只能指向该地址，不可指向别的地址。

- **const int\* const**
  - a constant pointer to a constant integer, 不可改变指针本身所指向的地址也不可  
通过指针改变其指向的内容。
- **int const \* const**
  - a constant pointer to a constant integer
- 从代码可读性易维护性出发, 推荐把 const 写在右边

```
char const *str="asdf";
```

```
str[1]='k'; X
```

指针指向的对象不能通过这个指针来修改，可是仍然可以通过原来的声明修改

```
str="qwerty"; ✓
```

```
char* const str="asdf";
```

```
str[1]='k'; ✓
```

可修改指针，不修改指向的内容

```
str="qwerty"; X
```

```
const char* const str="abcde"; //初始化后不能改变
```

- 为什么要用内联函数?
  - 从软件工程的角度来看，用函数写程序较好，但是函数调用涉及到执行时间的开销。
  - C++提供了内联函数(inline functions)来帮助减少函数调用的开销，尤其是对于小函数。
- 如何使用?
  - 在函数定义中，将限定符inline放置在函数的返回类型之前。编译器就地生成函数代码的副本，以避免函数调用。

# 内联函数(inline functions)



• 为  
•  
•  
• 如  
•

```
1 // Fig. 15.3: fig15_03.cpp
2 // inline function that calculates the volume of a cube.
3 #include <iostream>
4 using std::cout;
5 using std::cin;
6 using std::endl;
7
8 // Definition of inline function cube. Definition of function appears
9 // before function is called, so a function prototype is not required.
10 // First line of function definition acts as the prototype.
11 inline double cube( const double side )
12 {
13     return side * side * side; // calculate the cube of side
14 }
15
16 int main()
17 {
18     double sideValue; // stores value entered by user
19 }
```

**Fig. 15.3** | inline function that calculates the volume of a cube. (Part 1 of 2.)

子，  
来  
函数。  
数  
马的



- 既然内联函数能减少调用开销，所有函数都内联吧！
  - 错！
  - Inline的代价：在程序中插入了多个函数代码的副本（会使程序变大），而不是每次调用函数时都有一个函数的副本
  - 不过，编译器可以忽略内联限定符，通常除了最小的函数外，其他函数都会忽略
- Tips
  - 适用于函数代码少、频繁调用的情况；
  - 函数体内不能有循环语句、switch语句；
  - 代码不要太多（最好不要超过5行）

- 内联函数的认定还有多条**限制**：
  - 内联函数中不可含有循环；
  - 内联函数中不可含有switch语句；
  - 内联函数中不可含有静态变量；
  - 内联函数不可为递归函数；
  - 内联函数中不可含有异常处理。

# 内联函数(inline functions)



- 使用内联函数 `cube` (第11-14行) 计算边长为 `side` 的立方体的体积。
- 函数 `cube` 中, 关键字 `const` 告诉编译器, 该函数不修改变量 `side`。保证了计算时, `side` 的值不会被函数改变。
- ! 函数 `cube` 需要先定义, 后使用
- 所以, 可重用的内联函数通常被放在头文件中, 这样它们的定义就可以包含在每个使用它们的源文件中。

```
1 // Fig. 15.3: fig15_03.cpp
2 // inline function that calculates the volume of a cube.
3 #include <iostream>
4 using std::cout;
5 using std::cin;
6 using std::endl;
7
8 // Definition of inline function cube. Definition of function appears
9 // before function is called, so a function prototype is not required.
10 // First line of function definition acts as the prototype.
11 inline double cube( const double side )
12 {
13     return side * side * side; // calculate the cube of side
14 }
15
16 int main()
17 {
18     double sideValue; // stores value entered by user
19 }
```

Fig. 15.3 | inline function that calculates the volume of a cube. (Part 1 of 2.)

```
20 for ( int i = 1; i <= 3; i++ )
21 {
22     cout << "\nEnter the side length of your cube: ";
23     cin >> sideValue; // read value from user
24
25     // calculate cube of sideValue and display result
26     cout << "Volume of cube with side "
27         << sideValue << " is " << cube( sideValue ) << endl;
28 }
29 }
```

```
Enter the side length of your cube: 1.0
Volume of cube with side 1 is 1
```

```
Enter the side length of your cube: 2.3
Volume of cube with side 2.3 is 12.167
```

```
Enter the side length of your cube: 5.4
Volume of cube with side 5.4 is 157.464
```

Fig. 15.3 | inline function that calculates the volume of a cube. (Part 2 of 2.)



# 内联函数(inline functions)



- 有了4–6行的using, 22行以后的std::cout就可以省略成cout
- 4–6行还用更简便的写法
  - using namespace std;
  - 就可以用std命名空间下的所有命名(names)

```
1 // Fig. 15.3: fig15_03.cpp
2 // inline function that calculates the volume of a cube.
3 #include <iostream>
4 using std::cout;
5 using std::cin;
6 using std::endl;
7
8 // Definition of inline function cube. Definition of function appears
9 // before function is called, so a function prototype is not required.
10 // First line of function definition acts as the prototype.
11 inline double cube( const double side )
12 {
13     return side * side * side; // calculate the cube of side
14 }
15
16 int main()
17 {
18     double sideValue; // stores value entered by user
19
```

Fig. 15.3 | inline function that calculates the volume of a cube. (Part 1 of 2.)

```
20     for ( int i = 1; i <= 3; i++ )
21     {
22         cout << "\nEnter the side length of your cube: ";
23         cin >> sideValue; // read value from user
24
25         // calculate cube of sideValue and display result
26         cout << "Volume of cube with side "
27             << sideValue << " is " << cube( sideValue ) << endl;
28     }
29 }
```

```
Enter the side length of your cube: 1.0
Volume of cube with side 1 is 1

Enter the side length of your cube: 2.3
Volume of cube with side 2.3 is 12.167

Enter the side length of your cube: 5.4
Volume of cube with side 5.4 is 157.464
```

Fig. 15.3 | inline function that calculates the volume of a cube. (Part 2 of 2.)

[命名空间详解:](https://www.cplusplus.com/doc/oldtutorial/namespaces/)  
<https://www.cplusplus.com/doc/oldtutorial/namespaces/>

# 内联函数(inline functions)



- 后面的程序就会用 namespace 了
- 20行的for语句和C语言是一样的
- C++ 也有布尔类型：  
**true**和**false**.
- true=1, false=0
- 非布尔类型转换至布尔类型时，非零值转换为**true**，0值或null指针转换为**false**

```
1 // Fig. 15.3: fig15_03.cpp
2 // inline function that calculates the volume of a cube.
3 #include <iostream>
4 using std::cout;
5 using std::cin;
6 using std::endl;
7
8 // Definition of inline function cube. Definition of function appears
9 // before function is called, so a function prototype is not required.
10 // First line of function definition acts as the prototype.
11 inline double cube( const double side )
12 {
13     return side * side * side; // calculate the cube of side
14 }
15
16 int main()
17 {
18     double sideValue; // stores value entered by user
19
```

Fig. 15.3 | inline function that calculates the volume of a cube. (Part 1 of 2.)

```
20     for ( int i = 1; i <= 3; i++ )
21     {
22         cout << "\nEnter the side length of your cube: ";
23         cin >> sideValue; // read value from user
24
25         // calculate cube of sideValue and display result
26         cout << "Volume of cube with side "
27             << sideValue << " is " << cube( sideValue ) << endl;
28     }
29 }
```

```
Enter the side length of your cube: 1.0
Volume of cube with side 1 is 1

Enter the side length of your cube: 2.3
Volume of cube with side 2.3 is 12.167

Enter the side length of your cube: 5.4
Volume of cube with side 5.4 is 157.464
```

Fig. 15.3 | inline function that calculates the volume of a cube. (Part 2 of 2.)

# C++ 关键词(keywords)



## C++ Keywords

*Keywords common to the C and C++ programming languages*

auto	break	case	char	const
continue	default	do	double	else
enum	extern	float	for	goto
if	int	long	register	return
short	signed	sizeof	static	struct
switch	typedef	union	unsigned	void
volatile	while			

*C++-only keywords*

and	and_eq	asm	bitand	bitor
bool	catch	class	compl	const_cast
delete	dynamic_cast	explicit	export	false
friend	inline	mutable	namespace	new
not	not_eq	operator	or	or_eq
private	protected	public	reinterpret_cast	static_cast
template	this	throw	true	try
typeid	typename	using	virtual	wchar_t
xor	xor_eq			

**Fig. 15.4** | C++ keywords. (Part 1 of 2.)



- (复习) 什么是关键词?

Keywords are pre-defined words in a C++ compiler. Each keyword is meant to perform a specific function in a C++ program. Since keywords are referred names for compiler, **they can't be used as variable name.**

## C++ Keywords

### *C++11 keywords*

<code>alignas</code>	<code>alignof</code>	<code>char16_t</code>	<code>char32_t</code>	<code>constexpr</code>
<code>decltype</code>	<code>noexcept</code>	<code>nullptr</code>	<code>static_assert</code>	<code>thread_local</code>



**Fig. 15.4** | C++ keywords. (Part 2 of 2.)

- C语言中定义了指针后，动态分配与释放空间用 `malloc()`和`free()`函数

[例] `int *p;`

```
    p=(int *) malloc (个数 * sizeof (int ));  
    ..... //使用动态对象
```

```
    free( p );
```

- C++中用`new`和`delete`,形式简单。

```
[例]    int *p;  
        p=new int ;  
        *p=10;  
        cout<< *p ;  
        delete p;
```

- 动态申请内存操作符 `new`

`new` 类型名T (初值列表)

功能：在程序执行期间，申请用于存放T类型对象 `sizeof(T)` 的内存空间，并依初值列表赋以初值。

结果值：成功：T类型的指针，指向新分配的内存。  
失败：0 (NULL)

- `delete` 指针P

功能：释放指针P所指向的内存。P必须是new操作的返回值。

# //例如:



```
#include <iostream>
using namespace std;
int main()
{
    int *p;
    p=new int;
    *p=10;
    cout<<*p;
    delete p;
    return 0;
}
```



- new和delete的优点：
  - (1) new可以自动计算所要分配内存的的大小，不必使用sizeof()计算所需字节数，从而减少了发生错误的可能性。
  - (2) new能自动返回正确的指针类型，不必对返回指针进行类型转换。
  - (3) 可以用new将分配的对象初始化。
  - (4) new和delete都可以被重载，允许建立自定义的分配系统。



## •说明:

(1) C++可用new动态建立数组

```
char* string = new char[25];
```

(2) C++允许初始化新分配的对象, **但数组除外**

```
float *thingPtr=new float(3.14159);
```

```
char* pChar=new char('a');
```

# //例如:



```
#include <iostream>
using namespace std;
int main()
{
    int *p;
    p=new int(99); //动态分配内存并初始化为99
    cout<<*p;
    delete p;
    return 0;
}
```



(3) C++用delete [ ]释放动态分配的**数组空间**

```
delete [ ] string;
```

```
delete [ ] arrayInt;
```

(4) new对内存分配是否成功进行检查。

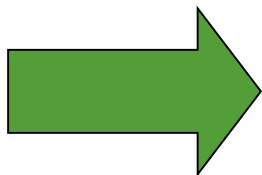
失败: 0 (NULL)

# //例如:



```
#include <iostream>
using namespace std;
int main()
{
    int *p;
    p=new int;
    if(!p)
    {
        cout<<"allocation failure\n";
        return 1;
    }
    *p=20;
    cout<<*p;
    delete p;
    return 0;
}
```



类型转换  系统自动进行  
强制类型转换

**格式**      **(类型名) 表达式**      **//C风格**  
                 **类型名 (表达式)**      **//C++风格**

**说明**

表达式应该用括号括起来。  
在强制类型转换时，只得到所需类型的中间变量，原来变量的类型不发生变化

例 (double)b; 将b转换成double  
(int)(x+y); 将x+y值转换成整型  
(float)(7%5); 将7%5值转换成实型

例

```
int main()
{
    float x;
    int i;
    x = 3.6;
    i = int(x);
    cout << "x=" << x << ", "
         << "i=" << i;
    return 0;
}
```

**运行结果** x=3.6 , i=3

**C++与C不同之处是声明函数原型，以保证实参和形参类型一致（编译器检查）。**

**语法形式：**

**返回类型 函数名(参数表列);**

**函数说明是一个语句，所以要以分号结束。C++中，函数说明也称为函数原型。**

## 1、函数原型的语法形式：

<类型> <函数名>(<参数表>);

例：

```
long Area(int,int);
```

```
long Area(int length,int width);
```

参数表中的参数  
名称可以省略



```
#include <iostream>
using namespace std;
void write(char *s); //函数原型的说明
int main()
{
    write("Hello,world!");
    return 0;
}
void write(char *s)
{ cout<<s;}
```

- 函数参数传递的方式：**按值传递** (pass-by-value) 和 **按引用传递** (pass-by-reference)
- **引用形参** (reference parameters)
  - C++提供的按引用传参的方式之一。另一种？
  - 允许被调函数直接访问、修改主调函数的数据
  - 可以提高性能，避免复制参数的开销，但降低了安全性
  - 如何即提高性能又保证安全性？后面会讲

## • 引用形参的实现

- 在函数原型中加入 & 符号

- `void swap(int& a, int& b) {...}`

- a是对int数据类型的引用

## • 引用作为变量别名

- 引用(&)是标识符的别名,例如:

```
int i,j;
```

```
int &ri=i;
```

```
//建立一个int型的引用ri,并将其
```

```
//初始化为变量i的一个别名
```

- 声明一个引用时, 必须同时对它进行初始化, 使它指向一个已存在的对象。
- 一旦一个引用被初始化后, 就不能改为指向其它对象。

# 引用调用示例



```
#include <iostream>
using namespace std;
void Swap(int &x, int &y);
int main()
{
    int a = 1, b = 2;
    cout << "a=" << a << " b="
<< b << endl;

    Swap(a, b);
    cout << "a=" << a << " b="
<< b << endl;
}

return 0;
}

void Swap(int &x, int &y)
{
    int t;
    t = x;
    x = y;
    y = t;
}
```

运行结果:

a=1 b=2

a=2 b=1



- 按引用传参另一种方式：按指针传递 (pass-by-pointer)

```
#include <iostream>
using namespace std;
void Swap(int *x, int *y); //函数定义, 形参为指针
int main()
{
    int a = 1, b = 2;
    cout << "a=" << a << " b="
<< b << endl;
    Swap(&a, &b);
    cout << "a=" << a << " b="
<< b << endl;
    return 0;
}
void Swap(int *x, int *y)
{
    int t;
    t = *x;
    *x = *y;
    *y = t;
}
```

运行结果:  
a=1 b=2  
a=2 b=1

详解:  
<https://www.geeksforgeeks.org/passing-by-pointer-vs-passing-by-reference-in-c/>



# 三种传参方式对比

```
#include <iostream>
using namespace std;
void test(int xa, int &xb, int *xc)
{
    xa=100; xb=200; *xc=300;
}
int main( )
{
    int a = 10, b=20, c=30;
    test(a, b, &c);
    cout<<"a="<<a<<" , b="<<b<<" , c="<<c<<endl;
    return 0;
}
```

程序输出结果  
是什么?

a=10, b=200, c=300



- 按引用传递只在**函数原型**中以&标出，但在函数调用时不会体现，容易出错
  - 在C++中，函数原型是**必需的**，是告诉编译器这个函数是存在的，让编译器知道这个函数的相关信息。
  - 函数原型不要求提供形参名，有类型列表就可以了
  - 避免使用函数原型的方法是，在首次使用函数定义之前定义它
- 使用常引用形参(const &)可以高效地传递大型对象，也不用担心使用时出错修改变量的值
- 三种传参方法到底如何用？
  - 按指针传递可修改的实参\*
  - 按值传递小型不可修改实参
  - 按常引用传递大型不可修改实参&

# 练习：分析程序的运行结果



```
#include <iostream>
using namespace std;
void f(int &m, int n) {
    int temp;
    temp = m;
    m = n;
    n = temp;
}
```

```
int main() {
    int a = 5, b = 10;
    f(a, b);
    cout << "a=" << a << "b=" << b << endl;
    return 0;
}
```

运行结果: a=10      b=10





# 练习：分析程序的运行结果



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

int& f(int& i) {
    i += 10;
    return i;
}

int main() {
    int k = 0;
    int& m = f(k);
    cout << k << endl;
    m = 20;
    cout << k << endl;
    return 0;
}
```

运行结果: 10  
20



# 引用作为变量别名



```
1 // Fig. 15.6: fig15_06.cpp
2 // Initializing and using a reference.
3 #include <iostream>
4 using namespace std;
5
6 int main()
7 {
8     int x = 3;
9     int &y = x; // y refers to (is an alias for) x
10
11     cout << "x = " << x << endl << "y = " << y << endl;
12     y = 7; // actually modifies x
13     cout << "x = " << x << endl << "y = " << y << endl;
14 }
```

```
x = 3
y = 3
x = 7
y = 7
```

- y是x的别名



# 引用作为变量别名



```
1 // Fig. 15.7: fig15_07.cpp
2 // Uninitialized reference is a syntax error.
3 #include <iostream>
4 using namespace std;
5
6 int main()
7 {
8     int x = 3;
9     int &y; // Error: y must be initialized
10
11     cout << "x = " << x << endl << "y = " << y << endl;
12     y = 7;
13     cout << "x = " << x << endl << "y = " << y << endl;
14 }
```

*Microsoft Visual C++ compiler error message:*

```
fig15_07.cpp(9) : error C2530: 'y' :
  references must be initialized
```

*GNU C++ compiler error message:*

```
fig15_07.cpp:9: error: 'y' declared as a reference but not initialized
```

*Xcode LLVM compiler error message:*

```
Declaration of reference variable 'y' requires an initializer
```

<http://www.dooccn.com/cpp/>



## What if ??

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

```
int main() {
    int x=3;
    int &y=x;
    int &z=x;
    z=6;
    cout << x << endl;
    cout << y << endl;
    cout << z << endl;
    return 0;
}
```

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

```
int main() {
    int x=3;
    int &y=x;
    int &z=y;
    z=6;
    cout << x << endl;
    cout << y << endl;
    cout << z << endl;
    return 0;
}
```

输出是什么？ ?

<http://www.dooccn.com/cpp/>



- 函数可以返回引用，但一般不推荐使用

```
//返回引用
const string &shorterString(const string &s1,const
string &s2)
{
    return s1.size()<s2.size()?s1:s2;
}
```

```
//禁止返回局部对象的引用
const string &mainip(const string &s)
{
    string ret=s;
    return ret;
}
```

[https://blog.csdn.net/qq\\_33266987/article/details/53516977](https://blog.csdn.net/qq_33266987/article/details/53516977)



- 和C一样，C++允许无参数的函数
- 在C++中，形参列表留空或者写void都可以：
  - `void print();`
  - `void print( void );`
- 以上两个函数原型是等价的

- 函数在声明时可以预先给出默认的形参值，调用时如给出实参，则采用实参值，否则采用预先给出的默认形参值。
- 注意事项
  - 默认实参必须是参数列表中最右边的参数
  - 默认值可以是**任何表达式**，如常量、全局变量、函数调用等
  - 也可用于内联函数中

# 默认实参(default arguments)



- 例子：求盒子体积
  - 第7行函数原型中长宽高默认为1
  - 第12行调用boxVolume函数没有指定实参
  - .....
  - 第24行指定了所有实参
- 注意：
  - 默认实参必须在最右侧，否则产生语法错误
  - 有人认为显式指定所有实参可增加可读性

```
1 // Fig. 15.8: fig15_08.cpp
2 // Using default arguments.
3 #include <iostream>
4 using namespace std;
5
6 // function prototype that specifies default arguments
7 int boxVolume( int length = 1, int width = 1, int height = 1 );
8
9 int main()
10 {
11     // no arguments--use default values for all dimensions
12     cout << "The default box volume is: " << boxVolume();
13
14     // specify length; default width and height
15     cout << "\n\nThe volume of a box with length 10,\n"
16           << "width 1 and height 1 is: " << boxVolume( 10 );
17
18     // specify length and width; default height
19     cout << "\n\nThe volume of a box with length 10,\n"
20           << "width 5 and height 1 is: " << boxVolume( 10, 5 );
21
22     // specify all arguments
23     cout << "\n\nThe volume of a box with length 10,\n"
24           << "width 5 and height 2 is: " << boxVolume( 10, 5, 2 )
25           << endl;
26 }
27
28 // function boxVolume calculates the volume of a box
29 int boxVolume( int length, int width, int height )
30 {
31     return length * width * height;
32 }
```

The default box volume is: 1

The volume of a box with length 10,  
width 1 and height 1 is: 10

The volume of a box with length 10,  
width 5 and height 1 is: 50

The volume of a box with length 10,  
width 5 and height 2 is: 100



- 重申一遍
  - 默认实参值必须从右向左顺序声明，并且在默认实参值的右面不能有非默认实参值的参数。因为调用时实参取代形参是从左向右的顺序。
  - 例：

```
int add(int x, int y=5, int z=6); //正确
```

```
int add(int x=1, int y=5, int z); //错误
```

```
int add(int x=1, int y, int z=6); //错误
```

# 练习1--大家来找茬



- 以下关于带默认参数值的程序代码段是否正确？

```
1: int f(int i = 0, int j);
```

```
2: int f(int i = 10); //函数声明
```

```
    int f(int i = 10) //函数定义
```

```
{
```

```
    return i*10;
```

```
}
```

1. 错

2. 错。函数原型中声明了默认实参，函数定义中不能再声明

3. 错。函数调用缺少参数j。

```
3: int f(int i, int j, int k = 0); //函数声明
```

```
    f(1); //函数调用
```



# 练习2

- 如果有的话，指出下面哪些函数声明是错误的？

(a) `int ff( int a, int b = 0, int c = 0 );`

(b) `void init( int ht = 24, int wd, char bckgrnd );`

- 假设有如下函数声明和调用,指出哪些调用是不合法的?哪些是合法的但可能不符合程序员的原意?

`void init( int ht, int wd = 80, char bckgrnd = ' ' );`

(a) `init();`

(b) `init( 24, 10 );`

(c) `init( 14, '*' );`

# 一元作用域运算符(::)



- 问题：局部变量和全局变量可以重名吗？
  - 可以，全局变量前需要带有一元作用域运算符 (::)

```
1 // Fig. 15.9: fig15_09.cpp
2 // Using the unary scope resolution operator.
3 #include <iostream>
4 using namespace std;
5
6 int number = 7; // global variable named number
7
8 int main()
9 {
10     double number = 10.5; // local variable named number
11
12     // display values of local and global variables
13     cout << "Local double value of number = " << number
14         << "\nGlobal int value of number = " << ::number << endl;
15 }
```

```
Local double value of number = 10.5
Global int value of number = 7
```

**Fig. 15.9** | Using the unary scope resolution operator.



## • 注意

- 使用::访问外层块中的局部变量是错误的
- 当然，不同的变量使用不同的名字是最好的，例如：count，global\_count

```
#include <iostream>
using namespace std;
int k=1;
int main() {
    for (int i=0; i<10; i++)
    {
        int k=0;
        ::k++;

    }
    cout << "test\n";
    return 0;
}
```

```
#include <iostream>
using namespace std;
int main() {
    int k=1;
    for (int i=0; i<10; i++)
    {
        int k=0;
        ::k++;

    }
    cout << "test\n";
    return 0;
}
```

- C++允许功能相近的函数在相同的作用域内以相同函数名声明，从而形成**重载**，只要这些函数具有不同的形参类型或个数。
- 例如：

```
int add(int x, int y);
```

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

} 形参类型不同

```
int add(int x, int y);
```

```
int add(int x, int y, int z);
```

} 形参个数不同

- 再举个例子，重载函数 square
  - 使用重载的square函数分别计算int和double类型的平方

```
1 // Fig. 15.10: fig15_10.cpp
2 // Overloaded square functions.
3 #include <iostream>
4 using namespace std;
5
6 // function square for int values
7 int square( int x )
8 {
9     cout << "square of integer " << x << " is ";
10    return x * x;
11 }
12
13 // function square for double values
14 double square( double y )
15 {
16    cout << "square of double " << y << " is ";
17    return y * y;
18 }
19
20 int main()
21 {
22    cout << square( 7 ); // calls int version
23    cout << endl;
24    cout << square( 7.5 ); // calls double version
25    cout << endl;
26 }
```

```
square of integer 7 is 49
square of double 7.5 is 56.25
```

**Fig. 15.10** | Overloaded square functions. (Part 2 of 2.)

- 重载函数的形参必须不同：个数不同或类型不同。
- 编译程序将根据实参和形参的类型及个数的最佳匹配来选择调用哪一个函数。

```
int add(int x,int y);  
int add(int a,int b);
```



编译器不以形参名来区分

```
int add(int x,int y);  
void add(int x,int y);
```



编译器不以返回值来区分

- 不要将不同功能的函数声明为重载函数，以免出现调用结果的误解、混淆。这样不好：

```
int add(int x,int y)  
{ return x+y; }
```



```
float add(float x,float y)  
{ return x-y; }
```



- 编写三个名为add的重载函数，分别实现两整数相加、两实数相加和两个复数相加的功能。

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

struct complex
{
    double real;  double imaginary;
};

int add(int m, int n)
{
    return m+n;
}

double add(double x, double y)
{
    return x+y;
}

complex add(complex c1, complex c2)
{
    complex c;
    c.real=c1.real+c2.real;
    c.imaginary=c1.imaginary+c2.im
    aginary;
    return c;
}
```



```
int main(void)
{
    int m, n; double x, y;
    complex c1, c2, c3;
    cin>>m>>n;
    cout<<add(m,n)<<endl;
    cin>>x>>y;
    cout<<add(x,y)<<endl;
    cin>>c1.real>>c1.imaginary;
    cin>>c2.real>>c2.imaginary;
    c3=add(c1,c2);
    cout<<c3.real<<', '<<c3.imaginary<<"\n";
    return 0;
}
```

输入:

3 5

2.3 6.8

12.3 54.2

125.1 35.7

输出:

8

9.1

137.4,89.9)



- **函数重载**用于对不同数据类型，用不同程序逻辑，执行相似操作
- 如果各种数据类型的程序执行的操作完全相同，那么可以用**函数模板**来实现
- 函数模板可以用来创建一个通用功能的函数，以支持多种不同形参，进一步简化重载函数的函数体设计

- 声明方法

- `template <typename 标识符>`
- `//或 template <class 标识符>`
- 函数声明

- 示例：求绝对值函数的模板

```
#include <iostream>
using namespace std;
template<typename T>
T abs(T x)
{
    return x<0 ? -x:x;
}
```

```
int main()
{
    int n=-5;
    double d=-5.5;
    cout<<abs(n)<<endl;
    cout<<abs(d)<<endl;
    return 0;
}
```

- 模板名字T可以指代任意合理数据类型
- 编译器从调用abs () 时实参的类型, 推导出函数模板的类型参数。例如, 对于调用表达式abs(n), 由于实参n为int型, 所以推导出模板中类型参数T为int。

```
#include <iostream>
using namespace std;
template<typename T>
T abs(T x)
{
    return x<0 ? -x:x;
}
```

```
int main()
{
    int n=-5;
    double d=-5.5;
    cout<<abs(n)<<endl;
    cout<<abs(d)<<endl;
    return 0;
}
```

- 模板名字T可以指代任意合理数据类型
- 编译器从调用abs () 时实参的类型, 推导出函数模板的类型参数。例如, 对于调用表达式abs(n), 由于实参n为int型, 所以推导出模板中类型参数T为int。

```
#include <iostream>
using namespace std;
int abs(int x)
{
    return x<0 ? -x:x;
}

int main()
{
    int n=-5;
    double d=-5.5;
    cout<<abs(n)<<endl;
    cout<<abs(d)<<endl;
    return 0;
}
```

## 书上的例子

```
1 // Fig. 15.12: maximum.h
2 // Function template maximum header file.
3
4 template < class T > // or template< typename T >
5 T maximum( T value1, T value2, T value3 )
6 {
7     T maximumValue = value1; // assume value1 is maximum
8
9     // determine whether value2 is greater than maximumValue
10    if ( value2 > maximumValue )
11        maximumValue = value2;
12
13    // determine whether value3 is greater than maximumValue
14    if ( value3 > maximumValue )
15        maximumValue = value3;
16
17    return maximumValue;
18 }
```

**Fig. 15.12** | Function template maximum header file.



# 函数模板(function template)



- 18, 28和38行分别进行int, double 和 char的大小比较
- 此时函数模板变成什么?

```
1 // Fig. 15.13: fig15_13.cpp
2 // Demonstrating function template maximum.
3 #include <iostream>
4 using namespace std;
5
6 #include "maximum.h" // include definition of function template maximum
7
8 int main()
9 {
10 // demonstrate maximum with int values
11 int int1, int2, int3;
12
13 cout << "Input three integer values: ";
14 cin >> int1 >> int2 >> int3;
15
16 // invoke int version of maximum
17 cout << "The maximum integer value is: "
18     << maximum( int1, int2, int3 );
19
20 // demonstrate maximum with double values
21 double double1, double2, double3;
22
```

```
    "\n\nInput three double values: ";
    double1 >> double2 >> double3;
    // invoke double version of maximum
    cout << "The maximum double value is: "
        << maximum( double1, double2, double3 );
    // demonstrate maximum with char values
    char char1, char2, char3;
    "\n\nInput three characters: ";
    char1 >> char2 >> char3;
    // invoke char version of maximum
    cout << "The maximum character value is: "
        << maximum( char1, char2, char3 ) << endl;
```

```
Input three integer values: 1 2 3
The maximum integer value is: 3
```

```
Input three double values: 3.3 2.2 1.1
The maximum double value is: 3.3
```

```
Input three characters: A C B
The maximum character value is: C
```

Fig. 15.13 | Demonstrating function template maximum. (Part 3 of 3.)

Demonstrating function template maximum. (Part 2 of 3.)



## C++在非面向对象方面的扩充

1. C++标准库STL简介
2. C++的头文件
3. `const`修饰符
4. 内联函数`Inline functions`
5. C++关键词
6. `new`和`delete`
7. 强制类型转换
8. 函数原型
9. 引用和引用形参
10. 空形参列表
11. 默认实参`default arguments`
12. 一元作用域运算符`::`
13. 函数重载`function overloading`
14. 函数模板`function templates`



感谢



- 本课程PPT部分内容收集自网上公开资料，如有侵权，请邮件联系我们
- 如您是承担了类似课程的教师，需要原始pptx文件，请邮件联系张书航或李同文
- 联系方式在首页