

对c++中常用的一些基本知识做一下简单的整理，只需要在看代码的时候可以搜索到对应的用法即可

具体文章来源于：<http://www.runoob.com/cplusplus/cpp-tutorial.html>

# C++基础知识

## 基本语法

C++ 程序可以定义为对象的集合，这些对象通过调用彼此的方法进行交互。现在让我们简要地看一下什么是类、对象，方法、即时变量。

- **对象** - 对象具有状态和行为。例如：一只狗的状态 - 颜色、名称、品种，行为 - 摇动、叫唤、吃。对象是类的实例。
- **类** - 类可以定义为描述对象行为/状态的模板/蓝图。
- **方法** - 从基本上说，一个方法表示一种行为。一个类可以包含多个方法。可以在方法中写入逻辑、操作数据以及执行所有的动作。
- **即时变量** - 每个对象都有其独特的即时变量。对象的状态是由这些即时变量的值创建的。

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

// main() 是程序开始执行的地方

int main()
{
    cout << "Hello World"; // 输出 Hello World
    return 0;
}
```

- C++ 语言定义了一些头文件，这些头文件包含了程序中必需的或有用的信息。上面这段程序中，包含了头文件。
- 下一行 **using namespace std;** 告诉编译器使用 std 命名空间。命名空间是 C++ 中一个相对新的概念。
- 下一行 **// main()** 是程序开始执行的地方 是一个单行注释。单行注释以 // 开头，在行末结束。
- 下一行 **int main()** 是主函数，程序从这里开始执行。
- 下一行 **cout << "Hello World";** 会在屏幕上显示消息 "Hello World"。
- 下一行 **return 0;** 终止 main() 函数，并向调用进程返回值 0。

## C++中的注释

使用/\* ---\*/或者是//

数据类型

布尔型	<b>bool</b>
字符型	char
整型	int
浮点型	float
双浮点型	double
无类型	void
宽字符型	wchar_t

一些基本类型可以使用一个或多个类型修饰符进行修饰：

- signed
- unsigned
- short
- long

示例

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

int main()
{
    cout << "type: \t\t" << "*****size*****"<< endl;
    cout << "bool: \t\t" << "所占字节数: " << sizeof(bool);
    cout << "\t最大值: " << (numeric_limits<bool>::max)();
    cout << "\t\t最小值: " << (numeric_limits<bool>::min)() << endl;
    cout << "char: \t\t" << "所占字节数: " << sizeof(char);
    cout << "\t最大值: " << (numeric_limits<char>::max)();
    cout << "\t\t最小值: " << (numeric_limits<char>::min)() << endl;
    cout << "signed char: \t" << "所占字节数: " << sizeof(signed char);
    cout << "\t最大值: " << (numeric_limits<signed char>::max)();
    cout << "\t\t最小值: " << (numeric_limits<signed char>::min)() << endl;
    cout << "unsigned char: \t" << "所占字节数: " << sizeof(unsigned char);
    cout << "\t最大值: " << (numeric_limits<unsigned char>::max)();
    cout << "\t\t最小值: " << (numeric_limits<unsigned char>::min)() << endl;
    cout << "wchar_t: \t" << "所占字节数: " << sizeof(wchar_t);
    cout << "\t最大值: " << (numeric_limits<wchar_t>::max)();
    cout << "\t\t最小值: " << (numeric_limits<wchar_t>::min)() << endl;
    cout << "short: \t\t" << "所占字节数: " << sizeof(short);
    cout << "\t最大值: " << (numeric_limits<short>::max)();
    cout << "\t\t最小值: " << (numeric_limits<short>::min)() << endl;
    cout << "int: \t\t" << "所占字节数: " << sizeof(int);
    cout << "\t最大值: " << (numeric_limits<int>::max)();
    cout << "\t最小值: " << (numeric_limits<int>::min)() << endl;
```

```

cout << "unsigned: \t" << "所占字节数: " << sizeof(unsigned);
cout << "\t最大值: " << (numeric_limits<unsigned>::max)();
cout << "\t最小值: " << (numeric_limits<unsigned>::min)() << endl;
cout << "long: \t\t" << "所占字节数: " << sizeof(long);
cout << "\t最大值: " << (numeric_limits<long>::max)();
cout << "\t最小值: " << (numeric_limits<long>::min)() << endl;
cout << "unsigned long: \t" << "所占字节数: " << sizeof(unsigned long);
cout << "\t最大值: " << (numeric_limits<unsigned long>::max)();
cout << "\t最小值: " << (numeric_limits<unsigned long>::min)() << endl;
cout << "double: \t" << "所占字节数: " << sizeof(double);
cout << "\t最大值: " << (numeric_limits<double>::max)();
cout << "\t最小值: " << (numeric_limits<double>::min)() << endl;
cout << "long double: \t" << "所占字节数: " << sizeof(long double);
cout << "\t最大值: " << (numeric_limits<long double>::max)();
cout << "\t最小值: " << (numeric_limits<long double>::min)() << endl;
cout << "float: \t\t" << "所占字节数: " << sizeof(float);
cout << "\t最大值: " << (numeric_limits<float>::max)();
cout << "\t最小值: " << (numeric_limits<float>::min)() << endl;
cout << "size_t: \t" << "所占字节数: " << sizeof(size_t);
cout << "\t最大值: " << (numeric_limits<size_t>::max)();
cout << "\t最小值: " << (numeric_limits<size_t>::min)() << endl;
cout << "string: \t" << "所占字节数: " << sizeof(string) << endl;
// << "\t最大值: " << (numeric_limits<string>::max)() << "\t最小值: " <<
(numeric_limits<string>::min)() << endl;
cout << "type: \t\t" << "*****size*****" << endl;
return 0;
}

```

本实例使用了 **endl**，这将在每一行后插入一个换行符，<< 运算符用于向屏幕传多个值。我们也使用 **sizeof()** 函数来获取各种数据类型的大小。

## typedef 声明

您可以使用 **typedef** 为一个已有的类型取一个新的名字。下面是使用 **typedef** 定义一个新类型的语法：

```
typedef type newname;
```

例如，下面的语句会告诉编译器，**feet** 是 **int** 的另一个名称：

```
typedef int feet;
```

现在，下面的声明是完全合法的，它创建了一个整型变量 **distance**：

```
feet distance;
```

## 枚举类型

枚举类型(enumeration)是C++中的一种派生数据类型，它是由用户定义的若干枚举常量的集合。

如果一个变量只有几种可能的值，可以定义为枚举(enumeration)类型。所谓"枚举"是指将变量的值一一列举出来，变量的值只能在列举出来的值的范围内。

创建枚举，需要使用关键字 **enum**。枚举类型的一般形式为：

```
enum 枚举名{  
    标识符 [= 整型常数],  
    标识符 [= 整型常数],  
    ...  
    标识符 [= 整型常数]  
} 枚举变量;
```

如果枚举没有初始化, 即省掉"=整型常数"时, 则从第一个标识符开始。

例如，下面的代码定义了一个颜色枚举，变量 `c` 的类型为 `color`。最后，`c` 被赋值为 "blue"。

```
enum color { red, green, blue } c;  
c = blue;
```

默认情况下，第一个名称的值为 0，第二个名称的值为 1，第三个名称的值为 2，以此类推。但是，您也可以给名称赋予一个特殊的值，只需要添加一个初始值即可。例如，在下面的枚举中，**green** 的值为 5。

```
enum color { red, green=5, blue };
```

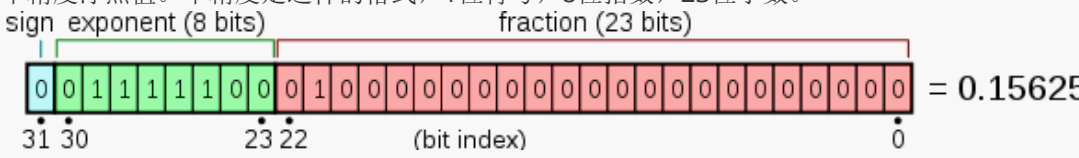
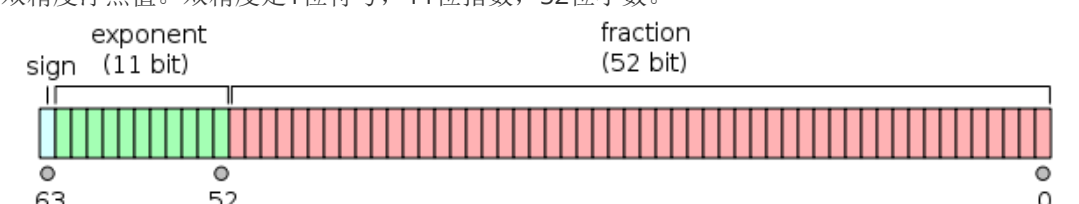
在这里，**blue** 的值为 6，因为默认情况下，每个名称都会比它前面一个名称大 1，但 **red** 的值依然为 0。

## 变量类型

变量其实只不过是程序可操作的存储区的名称。C++ 中每个变量都有指定的类型，类型决定了变量存储的大小和布局，该范围内的值都可以存储在内存中，运算符可应用于变量上。

变量的名称可以由字母、数字和下划线字符组成。它必须以字母或下划线开头。大写字母和小写字母是不同的，因为 C++ 是大小写敏感的。

基于前一章讲解的基本类型，有以下几种基本的变量类型，将在下一章中进行讲解：

类型	描述
bool	存储值 true 或 false。
char	通常是一个八位字节（一个字符）。这是一个整数类型。
int	对机器而言，整数的最自然的大小。
float	<p>单精度浮点值。单精度是这样的格式，1位符号，8位指数，23位小数。</p> 
double	<p>双精度浮点值。双精度是1位符号，11位指数，52位小数。</p> 
void	表示类型的缺失。
wchar_t	宽字符类型。

在这里，**type** 必须是一个有效的 C++ 数据类型，可以是 char、wchar\_t、int、float、double、bool 或任何用户自定义的对象，**variable\_list** 可以由一个或多个标识符名称组成，多个标识符之间用逗号分隔。下面列出几个有效的声明：

```
int    i, j, k;
char   c, ch;
float  f, salary;
double d;
```

行 **int i, j, k;** 声明并定义了变量 i、j 和 k，这指示编译器创建类型为 int 的名为 i、j、k 的变量。

变量可以在声明的时候被初始化（指定一个初始值）。初始化器由一个等号，后跟一个常量表达式组成，如下所示：

```
extern int d = 3, f = 5;    // d 和 f 的声明
int d = 3, f = 5;         // 定义并初始化 d 和 f
byte z = 22;              // 定义并初始化 z
char x = 'x';             // 变量 x 的值为 'x'
```

不带初始化的定义：带有静态存储持续时间的变量会被隐式初始化为 NULL（所有字节的值都是 0），其他所有变量的初始值是未定义的。

**C++ 中的变量声明**

变量声明向编译器保证变量以给定的类型和名称存在，这样编译器在不需要知道变量完整细节的情况下也能继续进一步的编译。变量声明只在编译时有它的意义，在程序连接时编译器需要实际的变量声明。

当您使用多个文件且只在其中一个文件中定义变量时（定义变量的文件在程序连接时是可用的），变量声明就显得非常有用。您可以使用 **extern** 关键字在任何地方声明一个变量。虽然您可以在 C++ 程序中多次声明一个变量，但变量只能在某个文件、函数或代码块中被定义一次。

```
#include <iostream>
using namespace std;
// 变量声明
extern int a, b;
extern int c;
extern float f;
```

同样的，在函数声明时，提供一个函数名，而函数的实际定义则可以在任何地方进行。例如：

```
// 函数声明
int func();
int main()
{
    // 函数调用
    int i = func();
}
// 函数定义
int func()
{
    return 0;
}
```

作用域是程序的一个区域，一般来说有三个地方可以定义变量：

- 在函数或一个代码块内部声明的变量，称为局部变量。
- 在函数参数的定义中声明的变量，称为形式参数。
- 在所有函数外部声明的变量，称为全局变量。

## C++ 常量

常量是固定值，在程序执行期间不会改变。这些固定的值，又叫做字面量。

常量可以是任何的基本数据类型，可分为整型数字、浮点数字、字符、字符串和布尔值。

常量就像是常规的变量，只不过常量的值在定义后不能进行修改。

### 整数常量

整数常量可以是十进制、八进制或十六进制的常量。前缀指定基数：0x 或 0X 表示十六进制，0 表示八进制，不带前缀则默认表示十进制。

整数常量也可以带一个后缀，后缀是 U 和 L 的组合，U 表示无符号整数（unsigned），L 表示长整数（long）。后缀可以是大写，也可以是小写，U 和 L 的顺序任意。

212 // 合法的 215u // 合法的 0xFeeL // 合法的 078 // 非法的：8 不是八进制的数字 032UU // 非法的：不能重复后缀

85 // 十进制 0213 // 八进制 0x4b // 十六进制 30 // 整数 30u // 无符号整数 30l // 长整数 30ul // 无符号长整数

浮点常量

浮点常量由整数部分、小数点、小数部分和指数部分组成。您可以使用小数形式或者指数形式来表示浮点常量。

当使用小数形式表示时，必须包含整数部分、小数部分，或同时包含两者。当使用指数形式表示时，必须包含小数点、指数，或同时包含两者。带符号的指数是用 e 或 E 引入的。

3.14159 // 合法的 314159E-5L // 合法的 510E // 非法的：不完整的指数 210f // 非法的：没有小数或指数 .e55 // 非法的：缺少整数或分数

字符常量

\\	\\ 字符
\'	\' 字符
\"	\" 字符
\\?	? 字符
\\a	警报铃声
\\b	退格键
\\f	换页符
\\n	换行符
\\r	回车
\\t	水平制表符
\\v	垂直制表符
\\ooo	一到三位的八进制数
\\xhh...	一个或多个数字的十六进制数

字符串常量

字符串面值或常量是括在双引号 "" 中的。一个字符串包含类似于字符常量的字符：普通的字符、转义序列和通用的字符。

您可以使用空格做分隔符，把一个很长的字符串常量进行分行。

下面的实例显示了一些字符串常量。下面这三种形式所显示的字符串是相同的。

定义常量

在 C++ 中，有两种简单的定义常量的方式：

- 使用 #define 预处理器。

- 使用 **const** 关键字。

下面是使用 **#define** 预处理器定义常量的形式：

```
#define identifier value
```

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

#define LENGTH 10
#define WIDTH 5
#define NEWLINE '\n'

int main()
{

    int area;

    area = LENGTH * WIDTH;
    cout << area;
    cout << NEWLINE;
    return 0;
}
```

## **const** 关键字

您可以使用 **const** 前缀声明指定类型的常量，如下所示：

```
const type variable = value;
```

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

int main()
{
    const int LENGTH = 10;
    const int WIDTH = 5;
    const char NEWLINE = '\n';
    int area;

    area = LENGTH * WIDTH;
    cout << area;
    cout << NEWLINE;
    return 0;
}
```

请注意，把常量定义为大写字母形式，是一个很好的编程实践。

## 修饰符类型

---



C++ 允许在 **char**、**int** 和 **double** 数据类型前放置修饰符。修饰符用于改变基本类型的含义，所以它更能满足各种情境的需求。

下面列出了数据类型修饰符：

- signed
- unsigned
- long
- short

修饰符 **signed**、**unsigned**、**long** 和 **short** 可应用于整型，**signed** 和 **unsigned** 可应用于字符型，**long** 可应用于双精度型。

修饰符 **signed** 和 **unsigned** 也可以作为 **long** 或 **short** 修饰符的前缀。例如：**unsigned long int**。

C++ 允许使用速记符号来声明无符号短整数或无符号长整数。您可以不写 **int**，只写单词 **unsigned**、**short** 或 **unsigned**、**long**，**int** 是隐含的。例如，下面的两个语句都声明了无符号整型变量。

```
unsigned x;  
unsigned int y;
```

C++ 中的类型限定符

类型限定符提供了变量的额外信息。

限定符	含义
const	<b>const</b> 类型的对象在程序执行期间不能被修改改变。
volatile	修饰符 <b>volatile</b> 告诉编译器，变量的值可能以程序未明确指定的方式被改变。
restrict	由 <b>restrict</b> 修饰的指针是唯一一种访问它所指向的对象的方式。只有 C99 增加了新的类型限定符 <b>restrict</b> 。

# 存储类

存储类定义 C++ 程序中变量/函数的范围（可见性）和生命周期。这些说明符放置在它们所修饰的类型之前。下面列出 C++ 程序中可用的存储类：

- auto
- register
- static
- extern
- mutable
- thread\_local (C++11)

从 C++ 11 开始，auto 关键字不再是 C++ 存储类说明符，且 register 关键字被弃用。

# static 存储类

**static** 存储类指示编译器在程序的生命周期内保持局部变量的存在，而不需要在每次它进入和离开作用域时进行创建和销毁。因此，使用 **static** 修饰局部变量可以在函数调用之间保持局部变量的值。

**static** 修饰符也可以应用于全局变量。当 **static** 修饰全局变量时，会使变量的作用域限制在声明它的文件内。

在 C++ 中，当 **static** 用在类数据成员上时，会导致仅有一个该成员的副本被类的所有对象共享。

```
#include <iostream>

// 函数声明
void func(void);

static int count = 10; /* 全局变量 */

int main()
{
    while(count-->0)
    {
        func();
    }
    return 0;
}

// 函数定义
void func( void )
{
    static int i = 5; // 局部静态变量
    i++;
    std::cout << "变量 i 为 " << i ;
    std::cout << " , 变量 count 为 " << count << std::endl;
}
```

```
变量 i 为 6 , 变量 count 为 9
变量 i 为 7 , 变量 count 为 8
变量 i 为 8 , 变量 count 为 7
变量 i 为 9 , 变量 count 为 6
变量 i 为 10 , 变量 count 为 5
变量 i 为 11 , 变量 count 为 4
变量 i 为 12 , 变量 count 为 3
变量 i 为 13 , 变量 count 为 2
变量 i 为 14 , 变量 count 为 1
变量 i 为 15 , 变量 count 为 0
```

可以说是在局部变量退出时，将其保存下来。

## extern 存储类

**extern** 存储类用于提供一个全局变量的引用，全局变量对所有的程序文件都是可见的。当您使用 'extern' 时，对于无法初始化的变量，会把变量名指向一个之前定义过的存储位置。

当您有多个文件且定义了一个可以在其他文件中使用的全局变量或函数时，可以在其他文件中使用 **extern** 来得到已定义的变量或函数的引用。可以这么理解，**extern** 是用来在另一个文件中声明一个全局变量或函数。

**extern** 修饰符通常用于当有两个或多个文件共享相同的全局变量或函数的时候，如下所示：

第一个文件：main.cpp

```
#include <iostream>

int count ;
extern void write_extern();

int main()
{
    count = 5;
    write_extern();
}
```

第二个文件：support.cpp

```
#include <iostream>
extern int count;
void write_extern(void)
{
    std::cout << "Count is " << count << std::endl;
}
```

在这里，第二个文件中的 **extern** 关键字用于声明已经在第一个文件 main.cpp 中定义的 count。

也就是说，使用外来的东西，该程序就不用管了，一般都是几个程序一起运行的时候有用。

### **mutable** 存储类

**mutable** 说明符仅适用于类的对象，这将在本教程的最后进行讲解。它允许对象的成员替代常量。也就是说，mutable 成员可以通过 const 成员函数修改。

### **thread\_local** 存储类

使用 **thread\_local** 说明符声明的变量仅可在它在其上创建的线程上访问。变量在创建线程时创建，并在销毁线程时销毁。每个线程都有其自己的变量副本。

**thread\_local** 说明符可以与 **static** 或 **extern** 合并。

可以将 **thread\_local** 仅应用于数据声明和定义，**thread\_local** 不能用于函数声明或定义。

以下演示了可以被声明为 **thread\_local** 的变量：

```
thread_local int x; // 命名空间下的全局变量
class X
{
    static thread_local std::string s; // 类的static成员变量
};
static thread_local std::string X::s; // X::s 是需要定义的

void foo()
{
    thread_local std::vector<int> v; // 本地变量
}
```

## 运算符

运算符是一种告诉编译器执行特定的数学或逻辑操作的符号。C++ 内置了丰富的运算符，并提供了以下类型的运算符：

- 算术运算符
- 关系运算符
- 逻辑运算符
- 位运算符
- 赋值运算符
- 杂项运算符

运算符	描述	实例
+	把两个操作数相加	A + B 将得到 30
-	从第一个操作数中减去第二个操作数	A - B 将得到 -10
*	把两个操作数相乘	A * B 将得到 200
/	分子除以分母	B / A 将得到 2
%	取模运算符，整除后的余数	B % A 将得到 0
++	<a href="#">自增运算符</a> ，整数值增加 1	A++ 将得到 11
--	<a href="#">自减运算符</a> ，整数值减少 1	A-- 将得到 9

运算符	描述	实例
==	检查两个操作数的值是否相等，如果相等则条件为真。	(A == B) 不为真。
!=	检查两个操作数的值是否相等，如果不相等则条件为真。	(A != B) 为真。
>	检查左操作数的值是否大于右操作数的值，如果是则条件为真。	(A > B) 不为真。
<	检查左操作数的值是否小于右操作数的值，如果是则条件为真。	(A < B) 为真。
>=	检查左操作数的值是否大于或等于右操作数的值，如果是则条件为真。	(A >= B) 不为真。
<=	检查左操作数的值是否小于或等于右操作数的值，如果是则条件为真。	(A <= B) 为真。

运算符	描述	实例
&&	称为逻辑与运算符。如果两个操作数都非零，则条件为真。	(A && B) 为假。
	称为逻辑或运算符。如果两个操作数中有任意一个非零，则条件为真。	(A    B) 为真。
!	称为逻辑非运算符。用来逆转操作数的逻辑状态。如果条件为真则逻辑非运算符将使其为假。	!(A && B) 为真。

运算符	描述	实例
&	如果同时存在于两个操作数中，二进制 AND 运算符复制一位到结果中。	(A & B) 将得到 12，即为 0000 1100
	如果存在于任一操作数中，二进制 OR 运算符复制一位到结果中。	(A   B) 将得到 61，即为 0011 1101
^	如果存在于其中一个操作数中但不同时存在于两个操作数中，二进制异或运算符复制一位到结果中。	(A ^ B) 将得到 49，即为 0011 0001
~	二进制补码运算符是一元运算符，具有"翻转"位效果，即0变成1，1变成0。	(~A) 将得到 -61，即为 1100 0011，一个有符号二进制数的补码形式。
<<	二进制左移运算符。左操作数的值向左移动右操作数指定的位数。	A << 2 将得到 240，即为 1111 0000
>>	二进制右移运算符。左操作数的值向右移动右操作数指定的位数。	A >> 2 将得到 15，即为 0000 1111

下表列出了 C++ 支持的赋值运算符：

运算符	描述	实例
=	简单的赋值运算符，把右边操作数的值赋给左边操作数	$C = A + B$ 将把 $A + B$ 的值赋给 $C$
+=	加且赋值运算符，把右边操作数加上左边操作数的结果赋值给左边操作数	$C += A$ 相当于 $C = C + A$
-=	减且赋值运算符，把左边操作数减去右边操作数的结果赋值给左边操作数	$C -= A$ 相当于 $C = C - A$
*=	乘且赋值运算符，把右边操作数乘以左边操作数的结果赋值给左边操作数	$C *= A$ 相当于 $C = C * A$
/=	除且赋值运算符，把左边操作数除以右边操作数的结果赋值给左边操作数	$C /= A$ 相当于 $C = C / A$
%=	求模且赋值运算符，求两个操作数的模赋值给左边操作数	$C \% = A$ 相当于 $C = C \% A$
<<=	左移且赋值运算符	$C <<= 2$ 等同于 $C = C << 2$
>>=	右移且赋值运算符	$C >>= 2$ 等同于 $C = C >> 2$
&=	按位与且赋值运算符	$C \&= 2$ 等同于 $C = C \& 2$
^=	按位异或且赋值运算符	$C \wedge= 2$ 等同于 $C = C \wedge 2$
=	按位或且赋值运算符	$C  = 2$ 等同于 $C = C   2$

下表列出了 C++ 支持的其他一些重要的运算符。

运算符	描述
sizeof	<a href="#">sizeof 运算符</a> 返回变量的大小。例如， <code>sizeof(a)</code> 将返回 4，其中 <code>a</code> 是整数。
Condition ? X : Y	<a href="#">条件运算符</a> 。如果 Condition 为真？则值为 X；否则值为 Y。
,	<a href="#">逗号运算符</a> 会顺序执行一系列运算。整个逗号表达式的值是以逗号分隔的列表中的最后一个表达式的值。
.（点）和 ->（箭头）	<a href="#">成员运算符</a> 用于引用类、结构和共用体的成员。
Cast	<a href="#">强制转换运算符</a> 把一种数据类型转换为另一种数据类型。例如， <code>int(2.2000)</code> 将返回 2。
&	<a href="#">指针运算符 &amp;</a> 返回变量的地址。例如 <code>&amp;a</code> ；将给出变量的实际地址。
*	<a href="#">指针运算符 *</a> 指向一个变量。例如， <code>*var</code> ；将指向变量 <code>var</code> 。

# 循环类型

C++ 编程语言提供了以下几种循环类型。点击链接查看每个类型的细节。

循环类型	描述
<a href="#">while 循环</a>	当给定条件为真时，重复语句或语句组。它会在执行循环主体之前测试条件。
<a href="#">for 循环</a>	多次执行一个语句序列，简化管理循环变量的代码。
<a href="#">do...while 循环</a>	除了它是在循环主体结尾测试条件外，其他与 <b>while</b> 语句类似。
<a href="#">嵌套循环</a>	您可以在 <b>while</b> 、 <b>for</b> 或 <b>do..while</b> 循环内使用一个或多个循环。

## 循环控制语句

循环控制语句更改执行的正常序列。当执行离开一个范围时，所有在该范围中创建的自动对象都会被销毁。

C++ 提供了下列的控制语句。点击链接查看每个语句的细节。

控制语句	描述
<a href="#">break 语句</a>	终止 <b>loop</b> 或 <b>switch</b> 语句，程序流将继续执行紧接着 <b>loop</b> 或 <b>switch</b> 的下一条语句。
<a href="#">continue 语句</a>	引起循环跳过主体的剩余部分，立即重新开始测试条件。
<a href="#">goto 语句</a>	将控制转移到被标记的语句。但是不建议在程序中使用 <b>goto</b> 语句。

注意：您可以按 **Ctrl + C** 键终止一个无限循环。

# 判断语句

C++ 编程语言提供了以下类型的判断语句。点击链接查看每个语句的细节。

语句	描述
<a href="#">if 语句</a>	一个 <b>if</b> 语句 由一个布尔表达式后跟一个或多个语句组成。
<a href="#">if...else 语句</a>	一个 <b>if</b> 语句 后可跟一个可选的 <b>else</b> 语句， <b>else</b> 语句在布尔表达式为假时执行。
<a href="#">嵌套 if 语句</a>	您可以在一个 <b>if</b> 或 <b>else if</b> 语句内使用另一个 <b>if</b> 或 <b>else if</b> 语句。
<a href="#">switch 语句</a>	一个 <b>switch</b> 语句允许测试一个变量等于多个值时的情况。
<a href="#">嵌套 switch 语句</a>	您可以在一个 <b>switch</b> 语句内使用另一个 <b>switch</b> 语句。

## ?: 运算符

我们已经在前面的章节中讲解了 [条件运算符 ?:](#)，可以用来替代 **if...else** 语句。它的一般形式如下：

```
Exp1 ? Exp2 : Exp3;
```

其中，Exp1、Exp2 和 Exp3 是表达式。请注意，冒号的使用和位置。

? 表达式的值是由 Exp1 决定的。如果 Exp1 为真，则计算 Exp2 的值，结果即为整个 ? 表达式的值。如果 Exp1 为假，则计算 Exp3 的值，结果即为整个 ? 表达式的值。

```
var = (y < 10) ? 30 : 40;
```

## 函数

函数是一组一起执行一个任务的语句。每个 C++ 程序都至少有一个函数，即主函数 **main()**，所有简单的程序都可以定义其他额外的函数。

您可以把代码划分到不同的函数中。如何划分代码到不同的函数中是由您来决定的，但在逻辑上，划分通常是根据每个函数执行一个特定的任务来进行的。

函数声明告诉编译器函数的名称、返回类型和参数。函数定义提供了函数的实际主体。

C++ 标准库提供了大量的程序可以调用的内置函数。例如，函数 **strcat()** 用来连接两个字符串，函数 **memcpy()** 用来复制内存到另一个位置。

函数还有很多叫法，比如方法、子例程或程序，等等。

### 定义函数

C++ 中的函数定义的一般形式如下：

```
return_type function_name( parameter list )
{
    body of the function
}
```

在 C++ 中，函数由一个函数头和一个函数主体组成。下面列出一个函数的所有组成部分：

- **返回类型**：一个函数可以返回一个值。**return\_type** 是函数返回的值的数据类型。有些函数执行所需的操作而不返回值，在这种情况下，**return\_type** 是关键字 **void**。
- **函数名称**：这是函数的实际名称。函数名和参数列表一起构成了函数签名。
- **参数**：参数就像是占位符。当函数被调用时，您向参数传递一个值，这个值被称为实际参数。参数列表包括函数参数的类型、顺序、数量。参数是可选的，也就是说，函数可能不包含参数。
- **函数主体**：函数主体包含一组定义函数执行任务的语句。

以下是 **max()** 函数的源代码。该函数有两个参数 num1 和 num2，会返回这两个数中较大的那个数：

```
// 函数返回两个数中较大的那个数

int max(int num1, int num2)
{
    // 局部变量声明
    int result;
```



```
    if (num1 > num2)
        result = num1;
    else
        result = num2;

    return result;
}
```

## 函数声明

函数声明会告诉编译器函数名称及如何调用函数。函数的实际主体可以单独定义。

函数声明包括以下几个部分：

```
return_type function_name( parameter list );
```

针对上面定义的函数 `max()`，以下是函数声明：

```
int max(int num1, int num2);
```

在函数声明中，参数的名称并不重要，只有参数的类型是必需的，因此下面也是有效的声明：

```
int max(int, int);
```

当您在源文件中定义函数且在另一个文件中调用函数时，函数声明是必需的。在这种情况下，您应该在调用函数的文件顶部声明函数。

## 调用函数

创建 C++ 函数时，会定义函数做什么，然后通过调用函数来完成已定义的任务。

当程序调用函数时，程序控制权会转移给被调用的函数。被调用的函数执行已定义的任务，当函数的返回语句被执行时，或到达函数的结束括号时，会把程序控制权交还给主程序。

调用函数时，传递所需参数，如果函数返回一个值，则可以存储返回值

## 函数参数

如果函数要使用参数，则必须声明接受参数值的变量。这些变量称为函数的**形式参数**。

形式参数就像函数内的其他局部变量，在进入函数时被创建，退出函数时被销毁。

当调用函数时，有两种向函数传递参数的方式：

调用类型	描述
<a href="#">传值调用</a>	该方法把参数的实际值复制给函数的形式参数。在这种情况下，修改函数内的形式参数对实际参数没有影响。
<a href="#">指针调用</a>	该方法把参数的地址复制给形式参数。在函数内，该地址用于访问调用中要用到的实际参数。这意味着，修改形式参数会影响实际参数。
<a href="#">引用调用</a>	该方法把参数的引用复制给形式参数。在函数内，该引用用于访问调用中要用到的实际参数。这意味着，修改形式参数会影响实际参数。

默认情况下，C++ 使用 **传值调用** 来传递参数。一般来说，这意味着函数内的代码不能改变用于调用函数的参数。之前提到的实例，调用 `max()` 函数时，使用了相同的方法。

## 传值调用

向函数传递参数的 **传值调用** 方法，把参数的实际值复制给函数的形式参数。在这种情况下，修改函数内的形式参数不会影响实际参数。

默认情况下，C++ 使用 *传值调用* 方法来传递参数。一般来说，这意味着函数内的代码不会改变用于调用函数的实际参数。函数 **`swap()`** 定义如下：

```
// 函数定义
void swap(int x, int y)
{
    int temp;

    temp = x; /* 保存 x 的值 */
    x = y;    /* 把 y 赋值给 x */
    y = temp; /* 把 x 赋值给 y */

    return;
}
```

现在，让我们通过传递实际参数来调用函数 **`swap()`**：

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

// 函数声明
void swap(int x, int y);

int main ()
{
    // 局部变量声明
    int a = 100;
    int b = 200;
```

```

cout << "交换前, a 的值: " << a << endl;
cout << "交换前, b 的值: " << b << endl;

// 调用函数来交换值
swap(a, b);

cout << "交换后, a 的值: " << a << endl;
cout << "交换后, b 的值: " << b << endl;

return 0;
}

```

当上面的代码被编译和执行时，它会产生下列结果：

```

交换前, a 的值: 100
交换前, b 的值: 200
交换后, a 的值: 100
交换后, b 的值: 200

```

上面的实例表明了，虽然在函数内改变了 **a** 和 **b** 的值，但是实际上 **a** 和 **b** 的值没有发生变化。

## 指针调用

向函数传递参数的**指针调用**方法，把参数的地址复制给形式参数。在函数内，该地址用于访问调用中要用到的实际参数。这意味着，修改形式参数会影响实际参数。

按指针传递值，参数指针被传递给函数，就像传递其他值给函数一样。因此相应地，在下面的函数 **swap()** 中，您需要声明函数参数为指针类型，该函数用于交换参数所指向的两个整数变量的值。

```

// 函数定义
void swap(int *x, int *y)
{
    int temp;
    temp = *x;    /* 保存地址 x 的值 */
    *x = *y;      /* 把 y 赋值给 x */
    *y = temp;    /* 把 x 赋值给 y */

    return;
}

```

如需了解 C++ 中指针的更多细节，请访问 [C++ 指针](#) 章节。

现在，让我们通过指针传值来调用函数 **swap()**：

```

#include <iostream>
using namespace std;

// 函数声明
void swap(int *x, int *y);

int main ()
{

```

```
// 局部变量声明
int a = 100;
int b = 200;

cout << "交换前, a 的值: " << a << endl;
cout << "交换前, b 的值: " << b << endl;

/* 调用函数来交换值
 * &a 表示指向 a 的指针, 即变量 a 的地址
 * &b 表示指向 b 的指针, 即变量 b 的地址
 */
swap(&a, &b);

cout << "交换后, a 的值: " << a << endl;
cout << "交换后, b 的值: " << b << endl;

return 0;
}
```

当上面的代码被编译和执行时, 它会产生下列结果:

```
交换前, a 的值: 100
交换前, b 的值: 200
交换后, a 的值: 200
交换后, b 的值: 100
```

## 引用调用

向函数传递参数的引用调用方法, 把引用的地址复制给形式参数。在函数内, 该引用用于访问调用中要用到的实际参数。这意味着, 修改形式参数会影响实际参数。

按引用传递值, 参数引用被传递给函数, 就像传递其他值给函数一样。因此相应地, 在下面的函数 **swap()** 中, 您需要声明函数参数为引用类型, 该函数用于交换参数所指向的两个整数变量的值。

```
// 函数定义
void swap(int &x, int &y)
{
    int temp;
    temp = x; /* 保存地址 x 的值 */
    x = y;    /* 把 y 赋值给 x */
    y = temp; /* 把 x 赋值给 y */

    return;
}
```

现在, 让我们通过引用传值来调用函数 **swap()**:

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

// 函数声明
```

```

void swap(int &x, int &y);

int main ()
{
    // 局部变量声明
    int a = 100;
    int b = 200;

    cout << "交换前, a 的值: " << a << endl;
    cout << "交换前, b 的值: " << b << endl;

    /* 调用函数来交换值 */
    swap(a, b);

    cout << "交换后, a 的值: " << a << endl;
    cout << "交换后, b 的值: " << b << endl;

    return 0;
}

```

当上面的代码被编译和执行时，它会产生下列结果：

```

交换前, a 的值: 100
交换前, b 的值: 200
交换后, a 的值: 200
交换后, b 的值: 100

```

## Lambda 函数与表达式

C++11 提供了对匿名函数的支持,称为 Lambda 函数(也叫 Lambda 表达式)。

Lambda 表达式把函数看作对象。Lambda 表达式可以像对象一样使用，比如可以将它们赋给变量和作为参数传递，还可以像函数一样对其求值。

Lambda 表达式本质上与函数声明非常类似。Lambda 表达式具体形式如下：

```
[capture](parameters)->return-type{body}
```

例如：

```
[](int x, int y){ return x < y ; }
```

如果没有返回值可以表示为：

```
[capture](parameters){body}
```

例如：

```
[]{ ++global_x; }
```

在一个更为复杂的例子中，返回类型可以被明确的指定如下：

```
[](int x, int y) -> int { int z = x + y; return z + x; }
```

本例中，一个临时的参数 `z` 被创建用来存储中间结果。如同一般的函数，`z` 的值不会保留到下一次该不具名函数再次被调用时。

如果 `lambda` 函数没有传回值（例如 `void`），其返回类型可被完全忽略。

在 `Lambda` 表达式内可以访问当前作用域的变量，这是 `Lambda` 表达式的闭包（`Closure`）行为。与 `JavaScript` 闭包不同，`C++` 变量传递有传值和传引用的区别。可以通过前面的 `[]` 来指定：

```
[]          // 没有定义任何变量。使用未定义变量会引发错误。
[x, &y]      // x以传值方式传入（默认），y以引用方式传入。
[&]         // 任何被使用到的外部变量都隐式地以引用方式加以引用。
[=]         // 任何被使用到的外部变量都隐式地以传值方式加以引用。
[&, x]      // x显式地以传值方式加以引用。其余变量以引用方式加以引用。
[=, &z]     // z显式地以引用方式加以引用。其余变量以传值方式加以引用。
```

另外有一点需要注意。对于 `[=]` 或 `[&]` 的形式，`lambda` 表达式可以直接使用 `this` 指针。但是，对于 `[]` 的形式，如果要使用 `this` 指针，必须显式传入：

```
[this]() { this->someFunc(); }();
```

## 数组

`C++` 支持数组数据结构，它可以存储一个固定大小的相同类型元素的顺序集合。数组是用来存储一系列数据，但它往往被认为是一系列相同类型的变量。

数组的声明并不是声明一个个单独的变量，比如 `number0`、`number1`、...、`number99`，而是声明一个数组变量，比如 `numbers`，然后使用 `numbers[0]`、`numbers[1]`、...、`numbers[99]` 来代表一个个单独的变量。数组中的特定元素可以通过索引访问。

所有的数组都是由连续的内存位置组成。最低的地址对应第一个元素，最高的地址对应最后一个元素。

在 `C++` 中要声明一个数组，需要指定元素的类型和元素的数量，如下所示：

```
type arrayName [ arraySize ];
```

这叫做一维数组。**`arraySize`** 必须是一个大于零的整数常量，**`type`** 可以是任意有效的 `C++` 数据类型。例如，要声明一个类型为 `double` 的包含 10 个元素的数组 **`balance`**，声明语句如下：

```
double balance[10];
```

在 `C++` 中，您可以逐个初始化数组，也可以使用一个初始化语句，如下所示：

```
double balance[5] = {1000.0, 2.0, 3.4, 7.0, 50.0};
```

您将创建一个数组，它与前一个实例中所创建的数组是完全相同的。下面是一个为数组中某个元素赋值的实例：

```
balance[4] = 50.0;
```

C++ 中数组详解

在 C++ 中，数组是非常重要的，我们需要了解更多有关数组的细节。下面列出了 C++ 程序员必须清楚的一些与数组相关的重要概念：

概念	描述
<a href="#">多维数组</a>	C++ 支持多维数组。多维数组最简单的形式是二维数组。
<a href="#">指向数组的指针</a>	您可以通过指定不带索引的数组名称来生成一个指向数组中第一个元素的指针。
<a href="#">传递数组给函数</a>	您可以通过指定不带索引的数组名称来给函数传递一个指向数组的指针。
<a href="#">从函数返回数组</a>	C++ 允许从函数返回数组。

多维数组

C++ 支持多维数组。多维数组声明的一般形式如下：

```
type name[size1][size2]...[sizeN];
```

例如，下面的声明创建了一个三维 5 . 10 . 4 整型数组：

```
int threedim[5][10][4]; // 5*10*4
```

二维数组

多维数组最简单的形式是二维数组。一个二维数组，在本质上，是一个一维数组的列表。声明一个 x 行 y 列的二维整型数组，形式如下：

```
type arrayName [ x ][ y ];
```

其中，**type** 可以是任意有效的 C++ 数据类型，**arrayName** 是一个有效的 C++ 标识符。

一个二维数组可以被认为是一个带有 x 行和 y 列的表格。下面是一个二维数组，包含 3 行和 4 列：

因此，数组中的每个元素是使用形式为 **a[i,j]** 的元素名称来标识的，其中 **a** 是数组名称，i 和 j 是唯一标识 **a** 中每个元素的下标。

初始化二维数组

多维数组可以通过在括号内为每行指定值来进行初始化。下面是一个带有 3 行 4 列的数组。

```
int a[3][4] = {
    {0, 1, 2, 3} , /* 初始化索引为 0 的行 */
    {4, 5, 6, 7} , /* 初始化索引为 1 的行 */
    {8, 9, 10, 11} /* 初始化索引为 2 的行 */
};
```

内部嵌套的括号是可选的，下面的初始化与上面是等同的：

```
int a[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
```

### 访问二维数组元素

二维数组中的元素是通过使用下标（即数组的行索引和列索引）来访问的。例如：

```
int val = a[2][3];
```

上面的语句将获取数组中第 3 行第 4 个元素。您可以通过上面的示意图来进行验证。让我们来看看下面的程序，我们将使用嵌套循环来处理二维数组：

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

int main ()
{
    // 一个带有 5 行 2 列的数组
    int a[5][2] = { {0,0}, {1,2}, {2,4}, {3,6},{4,8}};

    // 输出数组中每个元素的值
    for ( int i = 0; i < 5; i++ )
        for ( int j = 0; j < 2; j++ )
        {
            cout << "a[" << i << "][" << j << "]: ";
            cout << a[i][j]<< endl;
        }

    return 0;
}
```

## 指向数组的指针

您可以先跳过本章，等了解了 C++ 指针的概念之后，再来学习本章的内容。

如果您对 C++ 指针的概念有所了解，那么就可以开始本章的学习。数组名是一个指向数组中第一个元素的常量指针。因此，在下面的声明中：

```
double balance[50];
```

**balance** 是一个指向 **&balance[0]** 的指针，即数组 **balance** 的第一个元素的地址。因此，下面的程序片段把 **p** 赋值为 **balance** 的第一个元素的地址：

```
double *p;
double balance[10];

p = balance;
```

使用数组名作为常量指针是合法的，反之亦然。因此，**\*(balance + 4)** 是一种访问 **balance[4]** 数据的合法方式。

一旦您把第一个元素的地址存储在 **p** 中，您就可以使用 **p**、**(p+1)**、**\*(p+2)** 等来访问数组元素。



## 传递数组给函数

C++ 中您可以通过指定不带索引的数组名来传递一个指向数组的指针。

C++ 传数组给一个函数，数组类型自动转换为指针类型，因而传的实际是地址。

如果您想要在函数中传递一个一维数组作为参数，您必须以下面三种方式来声明函数形式参数，这三种声明方式的结果是一样的，因为每种方式都会告诉编译器将要接收一个整型指针。同样地，您也可以传递一个多维数组作为形式参数。

### 方式 1

形式参数是一个指针：

```
void myFunction(int *param)
{
    .
    .
    .
}
```

### 方式 2

形式参数是一个已定义大小的数组：

```
void myFunction(int param[10])
{
    .
    .
    .
}
```

### 方式 3

形式参数是一个未定义大小的数组：

```
void myFunction(int param[])
{
    .
    .
    .
}
```

### 实例

现在，让我们来看下面这个函数，它把数组作为参数，同时还传递了另一个参数，根据所传的参数，会返回数组中各元素的平均值：

```
double getAverage(int arr[], int size)
{
    int    i, sum = 0;
    double avg;
```

```

    for (i = 0; i < size; ++i)
    {
        sum += arr[i];
    }

    avg = double(sum) / size;

    return avg;
}

```

现在，让我们调用上面的函数，如下所示：

```

#include <iostream>
using namespace std;

// 函数声明
double getAverage(int arr[], int size);

int main ()
{
    // 带有 5 个元素的整型数组
    int balance[5] = {1000, 2, 3, 17, 50};
    double avg;

    // 传递一个指向数组的指针作为参数
    avg = getAverage( balance, 5 );

    // 输出返回值
    cout << "平均值是: " << avg << endl;

    return 0;
}

```

当上面的代码被编译和执行时，它会产生下列结果：

```

平均值是:  214.4

```

您可以看到，就函数而言，数组的长度是无关紧要的，因为 C++ 不会对形式参数执行边界检查

## 从函数返回数组

C++ 不允许返回一个完整的数组作为函数的参数。但是，您可以通过指定不带索引的数组名来返回一个指向数组的指针。

如果您想要从函数返回一个一维数组，您必须声明一个返回指针的函数，如下：

```

int * myFunction()
{
    ...
}

```

另外，C++ 不支持在函数外返回局部变量的地址，除非定义局部变量为 **static** 变量。

现在，让我们来看下面的函数，它会生成 10 个随机数，并使用数组来返回它们，具体如下：

实例

```
#include <iostream>
#include <cstdlib>
#include <ctime>

using namespace std;

// 要生成和返回随机数的函数
int * getRandom( )
{
    static int  r[10];

    // 设置种子
    srand( (unsigned)time( NULL ) );
    for (int i = 0; i < 10; ++i)
    {
        r[i] = rand();
        cout << r[i] << endl;
    }

    return r;
}

// 要调用上面定义函数的主函数
int main ()
{
    // 一个指向整数的指针
    int *p;

    p = getRandom();
    for ( int i = 0; i < 10; i++ )
    {
        cout << *(p + i) << " : ";
        cout << *(p + i) << endl;
    }

    return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
624723190
1468735695
807113585
976495677
613357504
```

```
1377296355
1530315259
1778906708
1820354158
667126415
*(p + 0) : 624723190
*(p + 1) : 1468735695
*(p + 2) : 807113585
*(p + 3) : 976495677
*(p + 4) : 613357504
*(p + 5) : 1377296355
*(p + 6) : 1530315259
*(p + 7) : 1778906708
*(p + 8) : 1820354158
*(p + 9) : 667126415
```

## 字符串

C++ 提供了以下两种类型的字符串表示形式：

- C 风格字符串
- C++ 引入的 `string` 类类型

### C 风格字符串

C 风格的字符串起源于 C 语言，并在 C++ 中继续得到支持。字符串实际上是使用 **null** 字符 `'\0'` 终止的一维字符数组。因此，一个以 null 结尾的字符串，包含了组成字符串的字符。

下面的声明和初始化创建了一个 "Hello" 字符串。由于在数组的末尾存储了空字符，所以字符数组的大小比单词 "Hello" 的字符数多一个。

```
char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

依据数组初始化规则，您可以把上面的语句写成以下语句：

```
char greeting[] = "Hello";
```

C++ 中有大量的函数用来操作以 null 结尾的字符串：supports a wide range of functions that manipulate null-terminated strings:

序号	函数 & 目的
1	<b>strcpy(s1, s2);</b> 复制字符串 s2 到字符串 s1。
2	<b>strcat(s1, s2);</b> 连接字符串 s2 到字符串 s1 的末尾。
3	<b>strlen(s1);</b> 返回字符串 s1 的长度。
4	<b>strcmp(s1, s2);</b> 如果 s1 和 s2 是相同的，则返回 0；如果 s1s2 则返回值大于 0。
5	<b>strchr(s1, ch);</b> 返回一个指针，指向字符串 s1 中字符 ch 的第一次出现的位置。
6	<b>strstr(s1, s2);</b> 返回一个指针，指向字符串 s1 中字符串 s2 的第一次出现的位置。

## String 类

```
#include <iostream>
#include <string>

using namespace std;

int main ()
{
    string str1 = "Hello";
    string str2 = "World";
    string str3;
    int len ;

    // 复制 str1 到 str3
    str3 = str1;
    cout << "str3 : " << str3 << endl;

    // 连接 str1 和 str2
    str3 = str1 + str2;
    cout << "str1 + str2 : " << str3 << endl;

    // 连接后，str3 的总长度
    len = str3.size();
    cout << "str3.size() : " << len << endl;

    return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
str3 : Hello
str1 + str2 : HelloWorld
str3.size() : 10
```

这个就和matlab里面是一样的，里面不仅仅包含你的输入，还包含了其中的很多信息，比如长度，类型等，后面再说。

## 指针

学习 C++ 的指针既简单又有趣。通过指针，可以简化一些 C++ 编程任务的执行，还有一些任务，如动态内存分配，没有指针是无法执行的。所以，想要成为一名优秀的 C++ 程序员，学习指针是很有必要的。

正如您所知道的，每一个变量都有一个内存位置，每一个内存位置都定义了可使用连字号（&）运算符访问的地址，它表示了在内存中的一个地址。请看下面的实例，它将输出定义的变量地址：

```
#include <iostream>

using namespace std;

int main ()
{
    int  var1;
    char var2[10];

    cout << "var1 变量的地址:  ";
    cout << &var1 << endl;

    cout << "var2 变量的地址:  ";
    cout << &var2 << endl;

    return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
var1 变量的地址:  0xbfebd5c0
var2 变量的地址:  0xbfebd5b6
```

什么是指针？

指针是一个变量，其值为另一个变量的地址，即，内存位置的直接地址。就像其他变量或常量一样，您必须在使用指针存储其他变量地址之前，对其进行声明。指针变量声明的一般形式为：

```
type *var-name;
```

在这里，**type** 是指针的基类型，它必须是一个有效的 C++ 数据类型，**var-name** 是指针变量的名称。用来声明指针的星号 \* 与乘法中使用的星号是相同的。但是，在这个语句中，星号是用来指定一个变量是指针。以下是有效的指针声明：

```
int    *ip;    /* 一个整型的指针 */
double *dp;    /* 一个 double 型的指针 */
float  *fp;    /* 一个浮点型的指针 */
char   *ch;    /* 一个字符型的指针 */
```

## C++ 中使用指针

使用指针时会频繁进行以下几个操作：定义一个指针变量、把变量地址赋值给指针、访问指针变量中可用地址的值。这些是通过使用一元运算符 **\*** 来返回位于操作数所指定地址的变量的值。下面的实例涉及到了这些操作：

```
#include <iostream>

using namespace std;

int main ()
{
    int  var = 20;    // 实际变量的声明
    int  *ip;         // 指针变量的声明

    ip = &var;        // 在指针变量中存储 var 的地址

    cout << "Value of var variable: ";
    cout << var << endl;

    // 输出在指针变量中存储的地址
    cout << "Address stored in ip variable: ";
    cout << ip << endl;

    // 访问指针中地址的值
    cout << "Value of *ip variable: ";
    cout << *ip << endl;

    return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Value of var variable: 20
Address stored in ip variable: 0xbfc601ac
Value of *ip variable: 20
```

## C++ 指针详解

在 C++ 中，有很多指针相关的概念，这些概念都很简单，但是都很重要。下面列出了 C++ 程序员必须清楚的一些与指针相关的重要概念：

概念	描述
<a href="#">C++ Null 指针</a>	C++ 支持空指针。NULL 指针是一个定义在标准库中的值为零的常量。
<a href="#">C++ 指针的算术运算</a>	可以对指针进行四种算术运算：++、--、+、-
<a href="#">C++ 指针 vs 数组</a>	指针和数组之间有着密切的关系。
<a href="#">C++ 指针数组</a>	可以定义用来存储指针的数组。
<a href="#">C++ 指向指针的指针</a>	C++ 允许指向指针的指针。
<a href="#">C++ 传递指针给函数</a>	通过引用或地址传递参数，使传递的参数在调用函数中被改变。
<a href="#">C++ 从函数返回指针</a>	C++ 允许函数返回指针到局部变量、静态变量和动态内存分配。

## C++ Null 指针

在大多数的操作系统上，程序不允许访问地址为 0 的内存，因为该内存是操作系统保留的。然而，内存地址 0 有特别重要的意义，它表明该指针不指向一个可访问的内存位置。但按照惯例，如果指针包含空值（零值），则假定它不指向任何东西。

如需检查一个空指针，您可以使用 if 语句，如下所示：

```
if(ptr)      /* 如果 ptr 非空，则完成 */
if(!ptr)     /* 如果 ptr 为空，则完成 */
```

因此，如果所有未使用的指针都被赋予空值，同时避免使用空指针，就可以防止误用一个未初始化的指针。很多时候，未初始化的变量存有一些垃圾值，导致程序难以调试。

## 指针的算术运算

指针是一个用数值表示的地址。因此，您可以对指针执行算术运算。可以对指针进行四种算术运算：++、--、+、-。

假设 **ptr** 是一个指向地址 1000 的整型指针，是一个 32 位的整数，让我们对该指针执行下列的算术运算：

```
ptr++
```

在执行完上述的运算之后，**ptr** 将指向位置 1004，因为 **ptr** 每增加一次，它都将指向下一个整数位置，即当前位置往后移 4 个字节。这个运算会在不影响内存位置中实际值的情况下，移动指针到下一个内存位置。如果 **ptr** 指向一个地址为 1000 的字符，上面的运算会导致指针指向位置 1001，因为下一个字符位置是在 1001。

## 指针 vs 数组

指针和数组是密切相关的。事实上，指针和数组在很多情况下是可以互换的。例如，一个指向数组开头的指针，可以通过使用指针的算术运算或数组索引来访问数组。请看下面的程序：

```
#include <iostream>
```



```

using namespace std;
const int MAX = 3;

int main ()
{
    int var[MAX] = {10, 100, 200};
    int *ptr;

    // 指针中的数组地址
    ptr = var;
    for (int i = 0; i < MAX; i++)
    {
        cout << "var[" << i << "]的内存地址为 ";
        cout << ptr << endl;

        cout << "var[" << i << "] 的值为 ";
        cout << *ptr << endl;

        // 移动到下一个位置
        ptr++;
    }
    return 0;
}

```

当上面的代码被编译和执行时，它会产生下列结果：

```

var[0]的内存地址为 0x7fff59707adc
var[0] 的值为 10
var[1]的内存地址为 0x7fff59707ae0
var[1] 的值为 100
var[2]的内存地址为 0x7fff59707ae4
var[2] 的值为 200

```

然而，指针和数组并不是完全互换的。例如，请看下面的程序：

```

#include <iostream>

using namespace std;
const int MAX = 3;

int main ()
{
    int var[MAX] = {10, 100, 200};

    for (int i = 0; i < MAX; i++)
    {
        *var = i;    // 这是正确的语法
        var++;       // 这是不正确的
    }

    return 0;
}

```

```
}
```

把指针运算符 `*` 应用到 `var` 上是完全可以的，但修改 `var` 的值是非法的。这是因为 `var` 是一个指向数组开头的常量，不能作为左值。

由于一个数组名对应一个指针常量，只要不改变数组的值，仍然可以用指针形式的表达式。例如，下面是一个有效的语句，把 `var[2]` 赋值为 500：

```
*(var + 2) = 500;
```

上面的语句是有效的，且能成功编译，因为 `var` 未改变。

## 指针数组

在我们讲解指针数组的概念之前，先让我们来看一个实例，它用到了一个由 3 个整数组成的数组：

```
#include <iostream>

using namespace std;
const int MAX = 3;

int main ()
{
    int var[MAX] = {10, 100, 200};

    for (int i = 0; i < MAX; i++)
    {
        cout << "Value of var[" << i << "] = ";
        cout << var[i] << endl;
    }
    return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Value of var[0] = 10
Value of var[1] = 100
Value of var[2] = 200
```

可能有一种情况，我们想要让数组存储指向 `int` 或 `char` 或其他数据类型的指针。下面是一个指向整数的指针数组的声明：

```
int *ptr[MAX];
```

在这里，把 `ptr` 声明为一个数组，由 `MAX` 个整数指针组成。因此，`ptr` 中的每个元素，都是一个指向 `int` 值的指针。下面的实例用到了三个整数，它们将存储在一个指针数组中，如下所示：

```
#include <iostream>
```

```

using namespace std;
const int MAX = 3;

int main ()
{
    int var[MAX] = {10, 100, 200};
    int *ptr[MAX];

    for (int i = 0; i < MAX; i++)
    {
        ptr[i] = &var[i]; // 赋值为整数的地址
    }
    for (int i = 0; i < MAX; i++)
    {
        cout << "Value of var[" << i << "] = ";
        cout << *ptr[i] << endl;
    }
    return 0;
}

```

当上面的代码被编译和执行时，它会产生下列结果：

```

Value of var[0] = 10
Value of var[1] = 100
Value of var[2] = 200

```

您也可以用指向字符的指针数组来存储一个字符串列表，如下：

```

#include <iostream>

using namespace std;
const int MAX = 4;

int main ()
{
    const char *names[MAX] = {
        "Zara Ali",
        "Hina Ali",
        "Nuha Ali",
        "Sara Ali",
    };

    for (int i = 0; i < MAX; i++)
    {
        cout << "Value of names[" << i << "] = ";
        cout << names[i] << endl;
    }
    return 0;
}

```

当上面的代码被编译和执行时，它会产生下列结果：

```
Value of names[0] = Zara Ali
Value of names[1] = Hina Ali
Value of names[2] = Nuha Ali
Value of names[3] = Sara Ali
```

## 指向指针的指针（多级间接寻址）

指向指针的指针是一种多级间接寻址的形式，或者说是一个指针链。通常，一个指针包含一个变量的地址。当我们定义一个指向指针的指针时，第一个指针包含了第二个指针的地址，第二个指针指向包含实际值的位置。

一个指向指针的指针变量必须如下声明，即在变量名前放置两个星号。例如，下面声明了一个指向 `int` 类型指针的指针：

```
int **var;
```

当一个目标值被一个指针间接指向到另一个指针时，访问这个值需要使用两个星号运算符，如下面实例所示：

```
#include <iostream>

using namespace std;

int main ()
{
    int  var;
    int  *ptr;
    int  **pptr;

    var = 3000;

    // 获取 var 的地址
    ptr = &var;

    // 使用运算符 & 获取 ptr 的地址
    pptr = &ptr;

    // 使用 pptr 获取值
    cout << "Value of var : " << var << endl;
    cout << "Value available at *ptr : " << *ptr << endl;
    cout << "Value available at **pptr : " << **pptr << endl;

    return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Value of var = 3000
Value available at *ptr = 3000
Value available at **pptr = 3000
```

## 传递指针给函数

C++ 允许您传递指针给函数，只需要简单地声明函数参数为指针类型即可。

下面的实例中，我们传递一个无符号的 long 型指针给函数，并在函数内改变这个值：

```
#include <iostream>
#include <ctime>

using namespace std;
void getSeconds(unsigned long *par);

int main ()
{
    unsigned long sec;

    getSeconds( &sec );

    // 输出实际值
    cout << "Number of seconds :" << sec << endl;

    return 0;
}

void getSeconds(unsigned long *par)
{
    // 获取当前的秒数
    *par = time( NULL );
    return;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Number of seconds :1294450468
```

能接受指针作为参数的函数，也能接受数组作为参数，如下所示：

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

// 函数声明
double getAverage(int *arr, int size);

int main ()
{
    // 带有 5 个元素的整型数组
    int balance[5] = {1000, 2, 3, 17, 50};
    double avg;

    // 传递一个指向数组的指针作为参数
    avg = getAverage( balance, 5 );
```

```

    // 输出返回值
    cout << "Average value is: " << avg << endl;

    return 0;
}

double getAverage(int *arr, int size)
{
    int    i, sum = 0;
    double avg;

    for (i = 0; i < size; ++i)
    {
        sum += arr[i];
    }

    avg = double(sum) / size;

    return avg;
}

```

当上面的代码被编译和执行时，它会产生下列结果：

```
Average value is: 214.4
```

## 从函数返回指针

在上一章中，我们已经了解了 C++ 中如何从函数返回数组，类似地，C++ 允许您从函数返回指针。为了做到这点，您必须声明一个返回指针的函数，如下所示：

```

int * myFunction()
{
    .
    .
    .
}

```

另外，C++ 不支持在函数外返回局部变量的地址，除非定义局部变量为 **static** 变量。

现在，让我们来看下面的函数，它会生成 10 个随机数，并使用表示指针的数组名（即第一个数组元素的地址）来返回它们，具体如下：

```

#include <iostream>
#include <ctime>
#include <cstdlib>

using namespace std;

// 要生成和返回随机数的函数
int * getRandom( )

```

```

{
    static int  r[10];

    // 设置种子
    srand( (unsigned)time( NULL ) );
    for (int i = 0; i < 10; ++i)
    {
        r[i] = rand();
        cout << r[i] << endl;
    }

    return r;
}

// 要调用上面定义函数的主函数
int main ()
{
    // 一个指向整数的指针
    int *p;

    p = getRandom();
    for ( int i = 0; i < 10; i++ )
    {
        cout << "(p + " << i << " ) : ";
        cout << *(p + i) << endl;
    }

    return 0;
}

```

当上面的代码被编译和执行时，它会产生下列结果：

```

624723190
1468735695
807113585
976495677
613357504
1377296355
1530315259
1778906708
1820354158
667126415
*(p + 0) : 624723190
*(p + 1) : 1468735695
*(p + 2) : 807113585
*(p + 3) : 976495677
*(p + 4) : 613357504
*(p + 5) : 1377296355
*(p + 6) : 1530315259
*(p + 7) : 1778906708
*(p + 8) : 1820354158
*(p + 9) : 667126415

```

## 递增一个指针

我们喜欢在程序中使用指针代替数组，因为变量指针可以递增，而数组不能递增，因为数组是一个常量指针。下面的程序递增变量指针，以便顺序访问数组中的每一个元素：

```
#include <iostream>

using namespace std;
const int MAX = 3;

int main ()
{
    int  var[MAX] = {10, 100, 200};
    int  *ptr;

    // 指针中的数组地址
    ptr = var;
    for (int i = 0; i < MAX; i++)
    {
        cout << "Address of var[" << i << "] = ";
        cout << ptr << endl;

        cout << "Value of var[" << i << "] = ";
        cout << *ptr << endl;

        // 移动到下一个位置
        ptr++;
    }
    return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Address of var[0] = 0xbfa088b0
Value of var[0] = 10
Address of var[1] = 0xbfa088b4
Value of var[1] = 100
Address of var[2] = 0xbfa088b8
Value of var[2] = 200
```

## 引用

引用变量是一个别名，也就是说，它是某个已存在变量的另一个名字。一旦把引用初始化为某个变量，就可以使用该引用名称或变量名称来指向变量。

简单的说，就是定义一个变量，完全的代替另一个变量，然后将其放到函数里面运行。

### C++ 引用 vs 指针

引用很容易与指针混淆，它们之间有三个主要的不同：

- 不存在空引用。引用必须连接到一块合法的内存。



- 一旦引用被初始化为一个对象，就不能被指向到另一个对象。指针可以在任何时候指向到另一个对象。
- 引用必须在创建时被初始化。指针可以在任何时间被初始化。

## C++ 中创建引用

试想变量名称是变量附属在内存位置中的标签，您可以把引用当成是变量附属在内存位置中的第二个标签。因此，您可以通过原始变量名称或引用来访问变量的内容。例如：

```
int i = 17;
```

我们可以为 `i` 声明引用变量，如下所示：

```
int& r = i;  
double& s = d;
```

在这些声明中，`&` 读作引用。因此，第一个声明可以读作 "r 是一个初始化为 `i` 的整型引用"，第二个声明可以读作 "s 是一个初始化为 `d` 的 `double` 型引用"。下面的实例使用了 `int` 和 `double` 引用：

```
#include <iostream>  
  
using namespace std;  
  
int main ()  
{  
    // 声明简单的变量  
    int i;  
    double d;  
  
    // 声明引用变量  
    int& r = i;  
    double& s = d;  
  
    i = 5;  
    cout << "Value of i : " << i << endl;  
    cout << "Value of i reference : " << r << endl;  
  
    d = 11.7;  
    cout << "Value of d : " << d << endl;  
    cout << "Value of d reference : " << s << endl;  
  
    return 0;  
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Value of i : 5  
Value of i reference : 5  
Value of d : 11.7  
Value of d reference : 11.7
```

引用通常用于函数参数列表和函数返回值。下面列出了 C++ 程序员必须清楚的两个与 C++ 引用相关的重要概念：

概念	描述
<a href="#">把引用作为参数</a>	C++ 支持把引用作为参数传给函数，这比传一般的参数更安全。
<a href="#">把引用作为返回值</a>	可以从 C++ 函数中返回引用，就像返回其他数据类型一样。

## 把引用作为参数

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

// 函数声明
void swap(int& x, int& y);

int main ()
{
    // 局部变量声明
    int a = 100;
    int b = 200;

    cout << "交换前, a 的值: " << a << endl;
    cout << "交换前, b 的值: " << b << endl;

    /* 调用函数来交换值 */
    swap(a, b);

    cout << "交换后, a 的值: " << a << endl;
    cout << "交换后, b 的值: " << b << endl;

    return 0;
}

// 函数定义
void swap(int& x, int& y)
{
    int temp;
    temp = x; /* 保存地址 x 的值 */
    x = y;    /* 把 y 赋值给 x */
    y = temp; /* 把 x 赋值给 y */

    return;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
交换前, a 的值: 100
交换前, b 的值: 200
交换后, a 的值: 200
交换后, b 的值: 100
```

## 把引用作为返回值

通过使用引用来替代指针，会使 C++ 程序更容易阅读和维护。C++ 函数可以返回一个引用，方式与返回一个指针类似。

当函数返回一个引用时，则返回一个指向返回值的隐式指针。这样，函数就可以放在赋值语句的左边。例如，请看下面这个简单的程序：

```
#include <iostream>

using namespace std;

double vals[] = {10.1, 12.6, 33.1, 24.1, 50.0};

double& setValues( int i )
{
    return vals[i];    // 返回第 i 个元素的引用
}

// 要调用上面定义函数的主函数
int main ()
{

    cout << "改变前的值" << endl;
    for ( int i = 0; i < 5; i++ )
    {
        cout << "vals[" << i << "] = ";
        cout << vals[i] << endl;
    }

    setValues(1) = 20.23; // 改变第 2 个元素
    setValues(3) = 70.8;  // 改变第 4 个元素

    cout << "改变后的值" << endl;
    for ( int i = 0; i < 5; i++ )
    {
        cout << "vals[" << i << "] = ";
        cout << vals[i] << endl;
    }
    return 0;
}
```

```
改变前的值
vals[0] = 10.1
vals[1] = 12.6
vals[2] = 33.1
vals[3] = 24.1
vals[4] = 50
改变后的值
vals[0] = 10.1
vals[1] = 20.23
vals[2] = 33.1
vals[3] = 70.8
vals[4] = 50
```

当返回一个引用时，要注意被引用的对象不能超出作用域。所以返回一个对局部变量的引用是不合法的，但是，可以返回一个对静态变量的引用。

```
int& func() {
    int q;
    //! return q; // 在编译时发生错误
    static int x;
    return x;      // 安全，x 在函数作用域外依然是有效的
}
```

## 日期 & 时间

C++ 标准库没有提供所谓的日期类型。C++ 继承了 C 语言用于日期和时间操作的结构和函数。为了使用日期和时间相关的函数和结构，需要在 C++ 程序中引用 头文件。

有四个与时间相关的类型：**clock\_t**、**time\_t**、**size\_t** 和 **tm**。类型 **clock\_t**、**size\_t** 和 **time\_t** 能够把系统时间和日期表示为某种整数。

结构类型 **tm** 把日期和时间以 C 结构的形式保存，**tm** 结构的定义如下：

```
struct tm {
    int tm_sec;    // 秒，正常范围从 0 到 59，但允许至 61
    int tm_min;    // 分，范围从 0 到 59
    int tm_hour;   // 小时，范围从 0 到 23
    int tm_mday;   // 一月中的第几天，范围从 1 到 31
    int tm_mon;    // 月，范围从 0 到 11
    int tm_year;   // 自 1900 年起的年数
    int tm_wday;   // 一周中的第几天，范围从 0 到 6，从星期日算起
    int tm_yday;   // 一年中的第几天，范围从 0 到 365，从 1 月 1 日算起
    int tm_isdst;  // 夏令时
}
```

下面是 C/C++ 中关于日期和时间的重要函数。所有这些函数都是 C/C++ 标准库的组成部分，您可以在 C++ 标准库中查看一下各个函数的细节。

序号	函数 & 描述
1	<a href="#"><code>time_t time(time_t *time);</code></a> 该函数返回系统的当前日历时间，自 1970 年 1 月 1 日以来经过的秒数。如果系统没有时间，则返回 .1。
2	<a href="#"><code>char *ctime(const time_t *time);</code></a> 该返回一个表示当地时间的字符串指针，字符串形式 <i>day month year hours:minutes:seconds year</i> \n\0。
3	<a href="#"><code>struct tm *localtime(const time_t *time);</code></a> 该函数返回一个指向表示本地时间的 <b>tm</b> 结构的指针。
4	<a href="#"><code>clock_t clock(void);</code></a> 该函数返回程序执行起（一般为程序的开头），处理器时钟所使用的时间。如果时间不可用，则返回 .1。
5	<a href="#"><code>char * asctime ( const struct tm * time );</code></a> 该函数返回一个指向字符串的指针，字符串包含了 time 所指向结构中存储的信息，返回形式为： <i>day month date hours:minutes:seconds year</i> \n\0。
6	<a href="#"><code>struct tm *gmtime(const time_t *time);</code></a> 该函数返回一个指向 time 的指针，time 为 tm 结构，用协调世界时（UTC）也被称为格林尼治标准时间（GMT）表示。
7	<a href="#"><code>time_t mktime(struct tm *time);</code></a> 该函数返回日历时间，相当于 time 所指向结构中存储的时间。
8	<a href="#"><code>double difftime ( time_t time2, time_t time1 );</code></a> 该函数返回 time1 和 time2 之间相差的秒数。
9	<a href="#"><code>size_t strftime();</code></a> 该函数可用于格式化日期和时间为指定的格式。

## 当前日期和时间

下面的实例获取当前系统的日期和时间，包括本地时间和协调世界时（UTC）。

```
#include <iostream>
#include <ctime>

using namespace std;

int main( )
{
    // 基于当前系统的当前日期/时间
    time_t now = time(0);

    // 把 now 转换为字符串形式
    char* dt = ctime(&now);

    cout << "本地日期和时间: " << dt << endl;

    // 把 now 转换为 tm 结构
    tm *gmtm = gmtime(&now);
    dt = asctime(gmtm);
    cout << "UTC 日期和时间: " << dt << endl;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
本地日期和时间: Sat Jan  8 20:07:41 2011
```

```
UTC 日期和时间: Sun Jan  9 03:07:41 2011
```

### 使用结构 **tm** 格式化时间

**tm** 结构在 C/C++ 中处理日期和时间相关的操作时，显得尤为重要。**tm** 结构以 C 结构的形式保存日期和时间。大多数与时间相关的函数都使用了 **tm** 结构。下面的实例使用了 **tm** 结构和各种与日期和时间相关的函数。

在练习使用结构之前，需要对 C 结构有基本的了解，并懂得如何使用箭头 -> 运算符来访问结构成员。

```
#include <iostream>
#include <ctime>

using namespace std;

int main( )
{
    // 基于当前系统的当前日期/时间
    time_t now = time(0);

    cout << "1970 到目前经过秒数:" << now << endl;

    tm *ltm = localtime(&now);

    // 输出 tm 结构的各个组成部分
    cout << "年: "<< 1900 + ltm->tm_year << endl;
    cout << "月: "<< 1 + ltm->tm_mon<< endl;
    cout << "日: "<< ltm->tm_mday << endl;
    cout << "时间: "<< ltm->tm_hour << ":";
    cout << ltm->tm_min << ":";
    cout << ltm->tm_sec << endl;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
1970 到目前时间:1503564157
年: 2017
月: 8
日: 24
时间: 16:42:37
```

## 数据结构

C/C++ 数组允许定义可存储相同类型数据项的变量，但是**结构**是 C++ 中另一种用户自定义的可用的数据类型，它允许您存储不同类型的数据项。

结构用于表示一条记录，假设您想要跟踪图书馆中书本的动态，您可能需要跟踪每本书的下列属性：

- Title：标题

- Author：作者
- Subject：类目
- Book ID：书的 ID

### 定义结构

为了定义结构，您必须使用 **struct** 语句。**struct** 语句定义了一个包含多个成员的新的数据类型，**struct** 语句的格式如下：

```
struct type_name {  
    member_type1 member_name1;  
    member_type2 member_name2;  
    member_type3 member_name3;  
    .  
    .  
} object_names;
```

**type\_name** 是结构体类型的名称，**member\_type1 member\_name1** 是标准的变量定义，比如 **int i;** 或者 **float f;** 或者其他有效的变量定义。在结构定义的末尾，最后一个分号之前，您可以指定一个或多个结构变量，这是可选的。下面是声明一个结构体类型 **Books**，变量为 **book**：

```
struct Books  
{  
    char title[50];  
    char author[50];  
    char subject[100];  
    int book_id;  
} book;
```

### 访问结构成员

为了访问结构的成员，我们使用成员访问运算符（.）。成员访问运算符是结构变量名称和我们要访问的结构成员之间的一个句号。

下面的实例演示了结构的用法：

```
#include <iostream>  
#include <cstring>  
  
using namespace std;  
  
// 声明一个结构体类型 Books  
struct Books  
{  
    char title[50];  
    char author[50];  
    char subject[100];  
    int book_id;  
};  
  
int main( )  
{
```

```

Books Book1;          // 定义结构体类型 Books 的变量 Book1
Books Book2;          // 定义结构体类型 Books 的变量 Book2

// Book1 详述
strcpy( Book1.title, "C++ 教程");
strcpy( Book1.author, "Runoob");
strcpy( Book1.subject, "编程语言");
Book1.book_id = 12345;

// Book2 详述
strcpy( Book2.title, "CSS 教程");
strcpy( Book2.author, "Runoob");
strcpy( Book2.subject, "前端技术");
Book2.book_id = 12346;

// 输出 Book1 信息
cout << "第一本书标题 : " << Book1.title <<endl;
cout << "第一本书作者 : " << Book1.author <<endl;
cout << "第一本书类目 : " << Book1.subject <<endl;
cout << "第一本书 ID : " << Book1.book_id <<endl;

// 输出 Book2 信息
cout << "第二本书标题 : " << Book2.title <<endl;
cout << "第二本书作者 : " << Book2.author <<endl;
cout << "第二本书类目 : " << Book2.subject <<endl;
cout << "第二本书 ID : " << Book2.book_id <<endl;

return 0;
}

```

## 结构作为函数参数

您可以把结构作为函数参数，传参方式与其他类型的变量或指针类似。您可以使用上面实例中的方式来访问结构变量：

```

#include <iostream>
#include <cstring>

using namespace std;
void printBook( struct Books book );

// 声明一个结构体类型 Books
struct Books
{
    char title[50];
    char author[50];
    char subject[100];
    int book_id;
};

int main( )
{
    Books Book1;          // 定义结构体类型 Books 的变量 Book1

```



```

Books Book2;          // 定义结构体类型 Books 的变量 Book2

// Book1 详述
strcpy( Book1.title, "C++ 教程");
strcpy( Book1.author, "Runoob");
strcpy( Book1.subject, "编程语言");
Book1.book_id = 12345;

// Book2 详述
strcpy( Book2.title, "CSS 教程");
strcpy( Book2.author, "Runoob");
strcpy( Book2.subject, "前端技术");
Book2.book_id = 12346;

// 输出 Book1 信息
printBook( Book1 );

// 输出 Book2 信息
printBook( Book2 );

return 0;
}
void printBook( struct Books book )
{
    cout << "书标题 : " << book.title <<endl;
    cout << "书作者 : " << book.author <<endl;
    cout << "书类目 : " << book.subject <<endl;
    cout << "书 ID : " << book.book_id <<endl;
}

```

## 指向结构的指针

您可以定义指向结构的指针，方式与定义指向其他类型变量的指针相似，如下所示：

```
struct Books *struct_pointer;
```

现在，您可以在上述定义的指针变量中存储结构变量的地址。为了查找结构变量的地址，请把 & 运算符放在结构名称的前面，如下所示：

```
struct_pointer = &Book1;
```

为了使用指向该结构的指针访问结构的成员，您必须使用 -> 运算符，如下所示：

```
struct_pointer->title;
```

让我们使用结构指针来重写上面的实例，这将有助于您理解结构指针的概念：

```

#include <iostream>
#include <cstring>

using namespace std;

```

```

void printBook( struct Books *book );

struct Books
{
    char  title[50];
    char  author[50];
    char  subject[100];
    int   book_id;
};

int main( )
{
    Books Book1;          // 定义结构体类型 Books 的变量 Book1
    Books Book2;          // 定义结构体类型 Books 的变量 Book2

    // Book1 详述
    strcpy( Book1.title, "C++ 教程");
    strcpy( Book1.author, "Runoob");
    strcpy( Book1.subject, "编程语言");
    Book1.book_id = 12345;

    // Book2 详述
    strcpy( Book2.title, "CSS 教程");
    strcpy( Book2.author, "Runoob");
    strcpy( Book2.subject, "前端技术");
    Book2.book_id = 12346;

    // 通过传 Book1 的地址来输出 Book1 信息
    printBook( &Book1 );

    // 通过传 Book2 的地址来输出 Book2 信息
    printBook( &Book2 );

    return 0;
}
// 该函数以结构指针作为参数
void printBook( struct Books *book )
{
    cout << "书标题   : " << book->title <<endl;
    cout << "书作者   : " << book->author <<endl;
    cout << "书类目   : " << book->subject <<endl;
    cout << "书 ID    : " << book->book_id <<endl;
}

```

## typedef 关键字

下面是一种更简单的定义结构的方式，您可以为创建的类型取一个"别名"。例如：

```
typedef struct
{
    char   title[50];
    char   author[50];
    char   subject[100];
    int    book_id;
}Books;
```

现在，您可以直接使用 *Books* 来定义 *Books* 类型的变量，而不需要使用 `struct` 关键字。下面是实例：

```
Books Book1, Book2;
```

您可以使用 **typedef** 关键字来定义非结构类型，如下所示：

```
typedef long int *pint32;

pint32 x, y, z;
```

`x, y` 和 `z` 都是指向长整型 `long int` 的指针。

# 面向对象

## 类 & 对象

C++ 在 C 语言的基础上增加了面向对象编程，C++ 支持面向对象程序设计。类是 C++ 的核心特性，通常被称为用户定义的类型。

类用于指定对象的形式，它包含了数据表示法和用于处理数据的方法。类中的数据和方法称为类的成员。函数在一个类中被称为类的成员。

### C++ 类定义

定义一个类，本质上是定义一个数据类型的蓝图。这实际上并没有定义任何数据，但它定义了类的名称意味着什么，也就是说，它定义了类的对象包括了什么，以及可以在这个对象上执行哪些操作。

类定义是以关键字 **class** 开头，后跟类的名称。类的主体是包含在一对花括号中。类定义后必须跟着一个分号或一个声明列表。例如，我们使用关键字 **class** 定义 `Box` 数据类型，如下所示：

```
class Box
{
    public:
        double length;    // 盒子的长度
        double breadth;   // 盒子的宽度
        double height;    // 盒子的高度
};
```

关键字 **public** 确定了类成员的访问属性。在类对象作用域内，公共成员在类的外部是可访问的。您也可以指定类的成员为 **private** 或 **protected**，这个我们稍后会进行讲解。

### 定义 C++ 对象

类提供了对象的蓝图，所以基本上，对象是根据类来创建的。声明类的对象，就像声明基本类型的变量一样。下面的语句声明了类 `Box` 的两个对象：

```
Box Box1;           // 声明 Box1，类型为 Box
Box Box2;           // 声明 Box2，类型为 Box
```

引用和`struct`一样，都是用（.），但是需要注意的是，私有的成员和受保护的成员不能使用直接成员访问运算符（.）来直接访问。我们将在后续的教程中学习如何访问私有成员和受保护的成员。

类 & 对象详解

到目前为止，我们已经对 C++ 的类和对象有了基本的了解。下面的列表中还列出了其他一些 C++ 类和对象相关的概念，可以点击相应的链接进行学习。

概念	描述
<a href="#">类成员函数</a>	类的成员函数是指那些把定义和原型写在类定义内部的函数，就像类定义中的其他变量一样。
<a href="#">类访问修饰符</a>	类成员可以被定义为 <code>public</code> 、 <code>private</code> 或 <code>protected</code> 。默认情况下是定义为 <code>private</code> 。
<a href="#">构造函数 &amp; 析构函数</a>	类的构造函数是一种特殊的函数，在创建一个新的对象时调用。类的析构函数也是一种特殊的函数，在删除所创建的对象时调用。
<a href="#">C++ 拷贝构造函数</a>	拷贝构造函数，是一种特殊的构造函数，它在创建对象时，是使用同一类中之前创建的对象来初始化新创建的对象。
<a href="#">C++ 友元函数</a>	友元函数可以访问类的 <code>private</code> 和 <code>protected</code> 成员。
<a href="#">C++ 内联函数</a>	通过内联函数，编译器试图在调用函数的地方扩展函数体中的代码。
<a href="#">C++ 中的 this 指针</a>	每个对象都有一个特殊的指针 <b>this</b> ，它指向对象本身。
<a href="#">C++ 中指向类的指针</a>	指向类的指针方式如同指向结构的指针。实际上，类可以看成是一个带有函数的结构。
<a href="#">C++ 类的静态成员</a>	类的数据成员和函数成员都可以被声明为静态的。

类成员函数

类的成员函数是指那些把定义和原型写在类定义内部的函数，就像类定义中的其他变量一样。类成员函数是类的一个成员，它可以操作类的任意对象，可以访问对象中的所有成员。

让我们看看之前定义类 `Box`，现在我们要使用成员函数来访问类的成员，而不是直接访问这些类的成员：

```
class Box
{
    public:
        double length;        // 长度
        double breadth;       // 宽度
        double height;        // 高度
        double getVolume(void); // 返回体积
};
```

成员函数可以定义在类定义内部，或者单独使用范围解析运算符 `::` 来定义。在类定义中定义的成员函数把函数声明为内联的，即便没有使用 `inline` 标识符。所以您可以按照如下方式定义 **Volume()** 函数：

```
class Box
{
    public:
        double length;        // 长度
        double breadth;       // 宽度
        double height;        // 高度

        double getVolume(void)
        {
            return length * breadth * height;
        }
};
```

您也可以在类的外部使用范围解析运算符 `::` 定义该函数，如下所示：

```
double Box::getVolume(void)
{
    return length * breadth * height;
}
```

在这里，需要强调一点，在 `::` 运算符之前必须使用类名。调用成员函数是在对象上使用点运算符 (`.`)，这样它就能操作与该对象相关的数据，如下所示：（就就是定义成员函数的时候使用 `::`，后面调用的时候采用 `.`）

```
Box myBox;                // 创建一个对象

myBox.getVolume();        // 调用该对象的成员函数
```

让我们使用上面提到的概念来设置和获取类中不同的成员的值：

```
#include <iostream>

using namespace std;

class Box
{
    public:
        double length;        // 长度
        double breadth;       // 宽度
```

```

        double height;           // 高度

        // 成员函数声明
        double getVolume(void);
        void setLength( double len );
        void setBreadth( double bre );
        void setHeight( double hei );
};

// 成员函数定义
double Box::getVolume(void)
{
    return length * breadth * height;
}

void Box::setLength( double len )
{
    length = len;
}

void Box::setBreadth( double bre )
{
    breadth = bre;
}

void Box::setHeight( double hei )
{
    height = hei;
}

// 程序的主函数
int main( )
{
    Box Box1;           // 声明 Box1, 类型为 Box
    Box Box2;           // 声明 Box2, 类型为 Box
    double volume = 0.0; // 用于存储体积

    // box 1 详述
    Box1.setLength(6.0);
    Box1.setBreadth(7.0);
    Box1.setHeight(5.0);

    // box 2 详述
    Box2.setLength(12.0);
    Box2.setBreadth(13.0);
    Box2.setHeight(10.0);

    // box 1 的体积
    volume = Box1.getVolume();
    cout << "Box1 的体积: " << volume << endl;

    // box 2 的体积
    volume = Box2.getVolume();

```

```
    cout << "Box2 的体积: " << volume << endl;
    return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Box1 的体积:  210
Box2 的体积:  1560
```

## C++ 类访问修饰符

数据封装是面向对象编程的一个重要特点，它防止函数直接访问类类型的内部成员。类成员的访问限制是通过在类主体内部对各个区域标记 **public**、**private**、**protected** 来指定的。关键字 **public**、**private**、**protected** 称为访问修饰符。

一个类可以有多个 **public**、**protected** 或 **private** 标记区域。每个标记区域在下一个标记区域开始之前或者在遇到类主体结束右括号之前都是有效的。成员和类的默认访问修饰符是 **private**。

### 公有（**public**）成员

公有成员在程序中类的外部是可访问的。您可以不使用任何成员函数来设置和获取公有变量的值，如下所示：

```
#include <iostream>

using namespace std;

class Line
{
public:
    double length;
    void setLength( double len );
    double getLength( void );
};

// 成员函数定义
double Line::getLength(void)
{
    return length ;
}

void Line::setLength( double len )
{
    length = len;
}

// 程序的主函数
int main( )
{
    Line line;

    // 设置长度
    line.setLength(6.0);
}
```

```

    cout << "Length of line : " << line.getLength() << endl;

    // 不使用成员函数设置长度
    line.length = 10.0; // OK: 因为 length 是公有的
    cout << "Length of line : " << line.length << endl;
    return 0;
}

```

## 私有（**private**）成员

私有成员变量或函数在类的外部是不可访问的，甚至是不可查看的。只有类和友元函数可以访问私有成员。

默认情况下，类的所有成员都是私有的。例如在下面的类中，**width** 是一个私有成员，这意味着，如果您没有使用任何访问修饰符，类的成员将被假定为私有成员：

```

class Box
{
    double width;
public:
    double length;
    void setWidth( double wid );
    double getWidth( void );
};

```

实际操作中，我们一般会在私有区域定义数据，在公有区域定义相关的函数，以便在类的外部也可以调用这些函数，如下所示：

```

#include <iostream>

using namespace std;

class Box
{
public:
    double length;
    void setWidth( double wid );
    double getWidth( void );

private:
    double width;
};

// 成员函数定义
double Box::getWidth(void)
{
    return width ;
}

void Box::setWidth( double wid )
{
    width = wid;
}

```



```

// 程序的主函数
int main( )
{
    Box box;

    // 不使用成员函数设置长度
    box.length = 10.0; // OK: 因为 length 是公有的
    cout << "Length of box : " << box.length <<endl;

    // 不使用成员函数设置宽度
    // box.width = 10.0; // Error: 因为 width 是私有的
    box.setWidth(10.0); // 使用成员函数设置宽度
    cout << "Width of box : " << box.getWidth() <<endl;

    return 0;
}

```

## 保护（**protected**）成员

保护成员变量或函数与私有成员十分相似，但有一点不同，保护成员在派生类（即子类）中是可访问的。

在下一个章节中，您将学习到派生类和继承的知识。现在您可以看到下面的实例中，我们从父类 **Box** 派生了一个子类 **smallBox**。

下面的实例与前面的实例类似，在这里 **width** 成员可被派生类 **smallBox** 的任何成员函数访问。

```

#include <iostream>
using namespace std;

class Box
{
    protected:
        double width;
};

class SmallBox:Box // SmallBox 是派生类
{
    public:
        void setSmallWidth( double wid );
        double getSmallWidth( void );
};

// 子类的成员函数
double SmallBox::getSmallWidth(void)
{
    return width ;
}

void SmallBox::setSmallWidth( double wid )
{
    width = wid;
}

// 程序的主函数

```

```
int main( )
{
    SmallBox box;

    // 使用成员函数设置宽度
    box.setSmallWidth(5.0);
    cout << "Width of box : "<< box.getSmallWidth() << endl;

    return 0;
}
```

## 继承中的特点

有public, protected, private三种继承方式，它们相应地改变了基类成员的访问属性。

- 1.**public** 继承：基类 public 成员，protected 成员，private 成员的访问属性在派生类中分别变成：public, protected, private
- 2.**protected** 继承：基类 public 成员，protected 成员，private 成员的访问属性在派生类中分别变成：protected, protected, private
- 3.**private** 继承：基类 public 成员，protected 成员，private 成员的访问属性在派生类中分别变成：private, private, private

但无论哪种继承方式，上面两点都没有改变：

- 1.private 成员只能被本类成员（类内）和友元访问，不能被派生类访问；
- 2.protected 成员可以被派生类访问。

## C++ 类构造函数 & 析构函数

### 类的构造函数

(简单说，就是对类的输入使用一个单独的构造函数进行说明，或者初始化里面变量的数据)

类的构造函数是类的一种特殊的成员函数，它会在每次创建类的新对象时执行。

构造函数的名称与类的名称是完全相同的，并且不会返回任何类型，也不会返回 void。构造函数可用于为某些成员变量设置初始值。

下面的实例有助于更好地理解构造函数的概念：

```
#include <iostream>

using namespace std;

class Line
{
public:
    void setLength( double len );
    double getLength( void );
    Line(); // 这是构造函数

private:
    double length;
};
```

```

// 成员函数定义，包括构造函数
Line::Line(void)
{
    cout << "Object is being created" << endl;
}

void Line::setLength( double len )
{
    length = len;
}

double Line::getLength( void )
{
    return length;
}
// 程序的主函数
int main( )
{
    Line line;

    // 设置长度
    line.setLength(6.0);
    cout << "Length of line : " << line.getLength() << endl;

    return 0;
}

```

### 带参数的构造函数

默认的构造函数没有任何参数，但如果需要，构造函数也可以带有参数。这样在创建对象时就会给对象赋初始值，如下面的例子所示：

```

#include <iostream>

using namespace std;

class Line
{
public:
    void setLength( double len );
    double getLength( void );
    Line(double len); // 这是构造函数

private:
    double length;
};

// 成员函数定义，包括构造函数
Line::Line( double len)
{
    cout << "Object is being created, length = " << len << endl;
    length = len;
}

```

```

}

void Line::setLength( double len )
{
    length = len;
}

double Line::getLength( void )
{
    return length;
}
// 程序的主函数
int main( )
{
    Line line(10.0);

    // 获取默认设置的长度
    cout << "Length of line : " << line.getLength() << endl;
    // 再次设置长度
    line.setLength(6.0);
    cout << "Length of line : " << line.getLength() << endl;

    return 0;
}

```

使用初始化列表来初始化字段

使用初始化列表来初始化字段：

```

Line::Line( double len): length(len)
{
    cout << "Object is being created, length = " << len << endl;
}

```

上面的语法等同于如下语法：

```

Line::Line( double len)
{
    cout << "Object is being created, length = " << len << endl;
    length = len;
}

```

假设有一个类 C，具有多个字段 X、Y、Z 等需要进行初始化，同理地，您可以使用上面的语法，只需要在不同的字段使用逗号进行分隔，如下所示：

```

C::C( double a, double b, double c): X(a), Y(b), Z(c)
{
    ....
}

```

类的析构函数

类的析构函数是类的一种特殊的成员函数，它会在每次删除所创建的对象时执行。

析构函数的名称与类的名称是完全相同的，只是在前面加了个波浪号（~）作为前缀，它不会返回任何值，也不能带有任何参数。析构函数有助于在跳出程序（比如关闭文件、释放内存等）前释放资源。

下面的实例有助于更好地理解析构函数的概念：

```
#include <iostream>

using namespace std;

class Line
{
public:
    void setLength( double len );
    double getLength( void );
    Line();    // 这是构造函数声明
    ~Line();   // 这是析构函数声明

private:
    double length;
};

// 成员函数定义，包括构造函数
Line::Line(void)
{
    cout << "Object is being created" << endl;
}
Line::~~Line(void)
{
    cout << "Object is being deleted" << endl;
}

void Line::setLength( double len )
{
    length = len;
}

double Line::getLength( void )
{
    return length;
}

// 程序的主函数
int main( )
{
    Line line;

    // 设置长度
    line.setLength(6.0);
    cout << "Length of line : " << line.getLength() << endl;

    return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Object is being created
Length of line : 6
Object is being deleted
```

（也就是说，构造函数是用来初始化的，里面一开始就会执行，而析构函数是在最后程序跳出时执行，而且不能带初始化参数）

## C++ 拷贝构造函数

拷贝构造函数是一种特殊的构造函数，它在创建对象时，是使用同一类中之前创建的对象来初始化新创建的对象。拷贝构造函数通常用于：

- 通过使用另一个同类型的对象来初始化新创建的对象。
- 复制对象把它作为参数传递给函数。
- 复制对象，并从函数返回这个对象。

如果在类中没有定义拷贝构造函数，编译器会自行定义一个。如果类带有指针变量，并有动态内存分配，则它必须有一个拷贝构造函数。拷贝构造函数的最常见形式如下：

```
classname (const classname &obj) {
    // 构造函数的主体
}
```

在这里，**obj** 是一个对象引用，该对象是用于初始化另一个对象的。

```
#include <iostream>

using namespace std;

class Line
{
public:
    int getLength( void );
    Line( int len );           // 简单的构造函数
    Line( const Line &obj);    // 拷贝构造函数
    ~Line();                  // 析构函数

private:
    int *ptr;
};

// 成员函数定义，包括构造函数
Line::Line(int len)
{
    cout << "调用构造函数" << endl;
    // 为指针分配内存
    ptr = new int;
    *ptr = len;
}
```

```

Line::Line(const Line &obj)
{
    cout << "调用拷贝构造函数并为指针 ptr 分配内存" << endl;
    ptr = new int;
    *ptr = *obj.ptr; // 拷贝值
}

Line::~~Line(void)
{
    cout << "释放内存" << endl;
    delete ptr;
}

int Line::getLength( void )
{
    return *ptr;
}

void display(Line obj)
{
    cout << "line 大小 : " << obj.getLength() << endl;
}

// 程序的主函数
int main( )
{
    Line line(10);

    display(line);

    return 0;
}

```

当上面的代码被编译和执行时，它会产生下列结果：

```

调用构造函数
调用拷贝构造函数并为指针 ptr 分配内存
line 大小 : 10
释放内存
释放内存

```

## C++ 友元函数

类的友元函数是定义在类外部，但有权访问类的所有私有（**private**）成员和保护（**protected**）成员。尽管友元函数的原型有在类的定义中出现过，但是友元函数并不是成员函数。

友元可以是一个函数，该函数被称为友元函数；友元也可以是一个类，该类被称为友元类，在这种情况下，整个类及其所有成员都是友元。

如果要声明函数为一个类的友元，需要在类定义中该函数原型前使用关键字 **friend**，如下所示：

```
class Box
{
    double width;
public:
    double length;
    friend void printWidth( Box box );
    void setWidth( double wid );
};
```

声明类 `ClassTwo` 的所有成员函数作为类 `ClassOne` 的友元，需要在类 `ClassOne` 的定义中放置如下声明：

```
friend class ClassTwo;
```

请看下面的程序：

```
#include <iostream>

using namespace std;

class Box
{
    double width;
public:
    friend void printWidth( Box box );
    void setWidth( double wid );
};

// 成员函数定义
void Box::setWidth( double wid )
{
    width = wid;
}

// 请注意: printWidth() 不是任何类的成员函数
void printWidth( Box box )
{
    /* 因为 printWidth() 是 Box 的友元, 它可以直接访问该类的任何成员 */
    cout << "Width of box : " << box.width << endl;
}

// 程序的主函数
int main( )
{
    Box box;

    // 使用成员函数设置宽度
    box.setWidth(10.0);

    // 使用友元函数输出宽度
    printWidth( box );
}
```



```
    return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Width of box : 10
```

## C++ 内联函数

C++ 内联函数是通常与类一起使用。如果一个函数是内联的，那么在编译时，编译器会把该函数的代码副本放置在每个调用该函数的地方。

对内联函数进行任何修改，都需要重新编译函数的所有客户端，因为编译器需要重新更换一次所有的代码，否则将会继续使用旧的函数。

如果想把一个函数定义为内联函数，则需要在函数名前面放置关键字 **inline**，在调用函数之前需要对函数进行定义。如果已定义的函数多于一行，编译器会忽略 **inline** 限定符。

在类定义中的定义的函数都是内联函数，即使没有使用 **inline** 说明符。

下面是一个实例，使用内联函数来返回两个数中的最大值：

```
#include <iostream>

using namespace std;

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

// 程序的主函数
int main( )
{

    cout << "Max (20,10): " << Max(20,10) << endl;
    cout << "Max (0,200): " << Max(0,200) << endl;
    cout << "Max (100,1010): " << Max(100,1010) << endl;
    return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Max (20,10): 20
Max (0,200): 200
Max (100,1010): 1010
```

## C++ this 指针

在 C++ 中，每一个对象都能通过 **this** 指针来访问自己的地址。**this** 指针是所有成员函数的隐含参数。因此，在成员函数内部，它可以用来指向调用对象。

友元函数没有 **this** 指针，因为友元不是类的成员。只有成员函数才有 **this** 指针。

下面的实例有助于更好地理解 **this** 指针的概念：

```
#include <iostream>

using namespace std;

class Box
{
public:
    // 构造函数定义
    Box(double l=2.0, double b=2.0, double h=2.0)
    {
        cout << "Constructor called." << endl;
        length = l;
        breadth = b;
        height = h;
    }
    double Volume()
    {
        return length * breadth * height;
    }
    int compare(Box box)
    {
        return this->Volume() > box.Volume();
    }
private:
    double length;    // Length of a box
    double breadth;    // Breadth of a box
    double height;    // Height of a box
};

int main(void)
{
    Box Box1(3.3, 1.2, 1.5);    // Declare box1
    Box Box2(8.5, 6.0, 2.0);    // Declare box2

    if(Box1.compare(Box2))
    {
        cout << "Box2 is smaller than Box1" << endl;
    }
    else
    {
        cout << "Box2 is equal to or larger than Box1" << endl;
    }
    return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Constructor called.  
Constructor called.  
Box2 is equal to or larger than Box1
```

## C++ 指向类的指针

一个指向 C++ 类的指针与指向结构的指针类似，访问指向类的指针的成员，需要使用成员访问运算符 `->`，就像访问指向结构的指针一样。与所有的指针一样，您必须在使用指针之前，对指针进行初始化。

下面的实例有助于更好地理解指向类的指针的概念：

```
#include <iostream>  
  
using namespace std;  
  
class Box  
{  
public:  
    // 构造函数定义  
    Box(double l=2.0, double b=2.0, double h=2.0)  
    {  
        cout <<"Constructor called." << endl;  
        length = l;  
        breadth = b;  
        height = h;  
    }  
    double Volume()  
    {  
        return length * breadth * height;  
    }  
private:  
    double length;    // Length of a box  
    double breadth;   // Breadth of a box  
    double height;    // Height of a box  
};  
  
int main(void)  
{  
    Box Box1(3.3, 1.2, 1.5);    // Declare box1  
    Box Box2(8.5, 6.0, 2.0);    // Declare box2  
    Box *ptrBox;                // Declare pointer to a class.  
  
    // 保存第一个对象的地址  
    ptrBox = &Box1;  
  
    // 现在尝试使用成员访问运算符来访问成员  
    cout << "Volume of Box1: " << ptrBox->Volume() << endl;  
  
    // 保存第二个对象的地址  
    ptrBox = &Box2;  
  
    // 现在尝试使用成员访问运算符来访问成员
```

```
    cout << "Volume of Box2: " << ptrBox->Volume() << endl;

    return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Constructor called.
Constructor called.
Volume of Box1: 5.94
Volume of Box2: 102
```

## C++ 类的静态成员

我们可以使用 **static** 关键字来把类成员定义为静态的。当我们声明类的成员为静态时，这意味着无论创建多少个类的对象，静态成员都只有一个副本。

静态成员在类的所有对象中是共享的。如果不存在其他的初始化语句，在创建第一个对象时，所有的静态数据都会被初始化为零。我们不能把静态成员的初始化放置在类的定义中，但是可以在类的外部通过使用范围解析运算符 **::** 来重新声明静态变量从而对它进行初始化，如下面的实例所示。

下面的实例有助于更好地理解静态成员数据的概念：

静态成员变量在类中仅仅是声明，没有定义，所以要在类的外面定义，实际上是给静态成员变量分配内存。如果不加定义就会报错，初始化是赋一个初始值，而定义是分配内存。

```
#include <iostream>

using namespace std;

class Box
{
public:
    static int objectCount;
    // 构造函数定义
    Box(double l=2.0, double b=2.0, double h=2.0)
    {
        cout << "Constructor called." << endl;
        length = l;
        breadth = b;
        height = h;
        // 每次创建对象时增加 1
        objectCount++;
    }
    double Volume()
    {
        return length * breadth * height;
    }
private:
    double length;    // 长度
    double breadth;   // 宽度
    double height;    // 高度
}
```

```
};

// 初始化类 Box 的静态成员 其实是定义并初始化的过程
int Box::objectCount = 0;
//也可这样 定义却不初始化
//int Box::objectCount;
int main(void)
{
    Box Box1(3.3, 1.2, 1.5);    // 声明 box1
    Box Box2(8.5, 6.0, 2.0);    // 声明 box2

    // 输出对象的总数
    cout << "Total objects: " << Box::objectCount << endl;

    return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Constructor called.
Constructor called.
Total objects: 2
```

## 静态成员函数

如果把函数成员声明为静态的，就可以把函数与类的任何特定对象独立开来。静态成员函数即使在类对象不存在的情况下也能被调用，**静态函数**只要使用类名加范围解析运算符 `::` 就可以访问。

静态成员函数只能访问静态成员数据、其他静态成员函数和类外部的其他函数。

静态成员函数有一个类范围，他们不能访问类的 `this` 指针。您可以使用静态成员函数来判断类的某些对象是否已被创建。

静态成员函数与普通成员函数的区别：

- 静态成员函数没有 `this` 指针，只能访问静态成员（包括静态成员变量和静态成员函数）。
- 普通成员函数有 `this` 指针，可以访问类中的任意成员；而静态成员函数没有 `this` 指针。

下面的实例有助于更好地理解静态成员函数的概念：

```
#include <iostream>

using namespace std;

class Box
{
public:
    static int objectCount;
    // 构造函数定义
    Box(double l=2.0, double b=2.0, double h=2.0)
    {
        cout << "Constructor called." << endl;
        length = l;
        breadth = b;
```

```

        height = h;
        // 每次创建对象时增加 1
        objectCount++;
    }
    double Volume()
    {
        return length * breadth * height;
    }
    static int getCount()
    {
        return objectCount;
    }
private:
    double length;    // 长度
    double breadth;   // 宽度
    double height;    // 高度
};

// 初始化类 Box 的静态成员
int Box::objectCount = 0;

int main(void)
{

    // 在创建对象之前输出对象的总数
    cout << "Initial Stage Count: " << Box::getCount() << endl;

    Box Box1(3.3, 1.2, 1.5);    // 声明 box1
    Box Box2(8.5, 6.0, 2.0);    // 声明 box2

    // 在创建对象之后输出对象的总数
    cout << "Final Stage Count: " << Box::getCount() << endl;

    return 0;
}

```

当上面的代码被编译和执行时，它会产生下列结果：

```

Initial Stage Count: 0
Constructor called.
Constructor called.
Final Stage Count: 2

```

## C++ 继承

面向对象程序设计中最重要一个概念是继承。继承允许我们依据另一个类来定义一个类，这使得创建和维护一个应用程序变得更容易。这样做，也达到了重用代码功能和提高执行时间的效果。

当创建一个类时，您不需要重新编写新的数据成员和成员函数，只需指定新建的类继承了一个已有的类的成员即可。这个已有的类称为**基类**，新建的类称为**派生类**。

继承代表了 **is a** 关系。例如，哺乳动物是动物，狗是哺乳动物，因此，狗是动物，等等。

## 基类 & 派生类

一个类可以派生自多个类，这意味着，它可以从多个基类继承数据和函数。定义一个派生类，我们使用一个类派生列表来指定基类。类派生列表以一个或多个基类命名，形式如下：

```
class derived-class: access-specifier base-class
```

其中，访问修饰符 `access-specifier` 是 **public**、**protected** 或 **private** 其中的一个，`base-class` 是之前定义过的某个类的名称。如果未使用访问修饰符 `access-specifier`，则默认为 `private`。

假设有一个基类 **Shape**，**Rectangle** 是它的派生类，如下所示：

```
#include <iostream>

using namespace std;

// 基类
class Shape
{
public:
    void setWidth(int w)
    {
        width = w;
    }
    void setHeight(int h)
    {
        height = h;
    }
protected:
    int width;
    int height;
};

// 派生类
class Rectangle: public Shape
{
public:
    int getArea()
    {
        return (width * height);
    }
};

int main(void)
{
    Rectangle Rect;

    Rect.setWidth(5);
    Rect.setHeight(7);

    // 输出对象的面积
    cout << "Total area: " << Rect.getArea() << endl;
```

```
    return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Total area: 35
```

## 访问控制和继承

派生类可以访问基类中所有的非私有成员。因此基类成员如果不想被派生类的成员函数访问，则应在基类中声明为 `private`。

我们可以根据访问权限总结出不同的访问类型，如下所示：

访问	<b>public</b>	<b>protected</b>	<b>private</b>
同一个类	yes	yes	yes
派生类	yes	yes	no
外部的类	yes	no	no

一个派生类继承了所有的基类方法，但下列情况除外：

- 基类的构造函数、析构函数和拷贝构造函数。
- 基类的重载运算符。
- 基类的友元函数。

### 继承类型

当一个类派生自基类，该基类可以被继承为 **public**、**protected** 或 **private** 几种类型。继承类型是通过上面讲解的访问修饰符 `access-specifier` 来指定的。

我们几乎不使用 **protected** 或 **private** 继承，通常使用 **public** 继承。当使用不同类型的继承时，遵循以下几个规则：

- **公有继承（public）**：当一个类派生自公有基类时，基类的公有成员也是派生类的公有成员，基类的保护成员也是派生类的保护成员，基类的私有成员不能直接被派生类访问，但是可以通过调用基类的公有和保护成员来访问。
- **保护继承（protected）**：当一个类派生自保护基类时，基类的公有和保护成员将成为派生类的保护成员。
- **私有继承（private）**：当一个类派生自私有基类时，基类的公有和保护成员将成为派生类的私有成员。

### 多继承

多继承即一个子类可以有多个父类，它继承了多个父类的特性。

C++ 类可以从多个类继承成员，语法如下：

```
class <派生类名>:<继承方式1><基类名1>,<继承方式2><基类名2>,...
{
    <派生类类体>
};
```



其中，访问修饰符继承方式是 **public**、**protected** 或 **private** 其中的一个，用来修饰每个基类，各个基类之间用逗号分隔，如上所示。现在让我们一起来看看下面的实例：

```
#include <iostream>

using namespace std;

// 基类 Shape
class Shape
{
public:
    void setWidth(int w)
    {
        width = w;
    }
    void setHeight(int h)
    {
        height = h;
    }
protected:
    int width;
    int height;
};

// 基类 PaintCost
class PaintCost
{
public:
    int getCost(int area)
    {
        return area * 70;
    }
};

// 派生类
class Rectangle: public Shape, public PaintCost
{
public:
    int getArea()
    {
        return (width * height);
    }
};

int main(void)
{
    Rectangle Rect;
    int area;

    Rect.setWidth(5);
    Rect.setHeight(7);

    area = Rect.getArea();
}
```

```
// 输出对象的面积
cout << "Total area: " << Rect.getArea() << endl;

// 输出总花费
cout << "Total paint cost: $" << Rect.getCost(area) << endl;

return 0;
}
```

## C++ 重载运算符和重载函数

（简单理解就是：重载函数重新对原函数进行定义，重载运算符在原来的基础上进行运算符操作，一般是相同的类加减乘除，免得重新定义一个结构比较像的类）

C++ 允许在同一作用域中的某个函数和运算符指定多个定义，分别称为**函数重载**和**运算符重载**。

重载声明是指一个与之前已经在该作用域内声明过的函数或方法具有相同名称的声明，但是它们的参数列表和定义（实现）不相同。

当您调用一个**重载函数**或**重载运算符**时，编译器通过把您所使用的参数类型与定义中的参数类型进行比较，决定选用最合适的定义。选择最合适的重载函数或重载运算符的过程，称为**重载决策**。

### C++ 中的函数重载

在同一个作用域内，可以声明几个功能类似的同名函数，但是这些同名函数的形式参数（指参数的个数、类型或者顺序）必须不同。您不能仅通过返回类型的不同来重载函数。

下面的实例中，同名函数 **print()** 被用于输出不同的数据类型：

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

class printData
{
public:
    void print(int i) {
        cout << "整数为: " << i << endl;
    }

    void print(double f) {
        cout << "浮点数为: " << f << endl;
    }

    void print(string c) {
        cout << "字符串为: " << c << endl;
    }
};

int main(void)
{
    printData pd;

    // 输出整数
```

```

    pd.print(5);
    // 输出浮点数
    pd.print(500.263);
    // 输出字符串
    pd.print("Hello C++");

    return 0;
}

```

当上面的代码被编译和执行时，它会产生下列结果：

```

整数为： 5
浮点数为： 500.263
字符串为： Hello C++

```

## C++ 中的运算符重载

您可以重定义或重载大部分 C++ 内置的运算符。这样，您就能使用自定义类型的运算符。

重载的运算符是带有特殊名称的函数，函数名是由关键字 **operator** 和其后要重载的运算符符号构成的。与其他函数一样，重载运算符有一个返回类型和一个参数列表。

```
Box operator+(const Box&);
```

声明加法运算符用于把两个 **Box** 对象相加，返回最终的 **Box** 对象。大多数的重载运算符可被定义为普通的非成员函数或者被定义为类成员函数。如果我们定义上面的函数为类的非成员函数，那么我们需要为每次操作传递两个参数，如下所示：

```
Box operator+(const Box&, const Box&);
```

下面的实例使用成员函数演示了运算符重载的概念。在这里，对象作为参数进行传递，对象的属性使用 **this** 运算符进行访问，如下所示：

```

#include <iostream>
using namespace std;

class Box
{
public:

    double getVolume(void)
    {
        return length * breadth * height;
    }
    void setLength( double len )
    {
        length = len;
    }

    void setBreadth( double bre )
    {

```

```

        breadth = bre;
    }

    void setHeight( double hei )
    {
        height = hei;
    }
    // 重载 + 运算符，用于把两个 Box 对象相加
    Box operator+(const Box& b)
    {
        Box box;
        box.length = this->length + b.length;
        box.breadth = this->breadth + b.breadth;
        box.height = this->height + b.height;
        return box;
    }
private:
    double length;      // 长度
    double breadth;     // 宽度
    double height;      // 高度
};
// 程序的主函数
int main( )
{
    Box Box1;           // 声明 Box1，类型为 Box
    Box Box2;           // 声明 Box2，类型为 Box
    Box Box3;           // 声明 Box3，类型为 Box
    double volume = 0.0; // 把体积存储在该变量中

    // Box1 详述
    Box1.setLength(6.0);
    Box1.setBreadth(7.0);
    Box1.setHeight(5.0);

    // Box2 详述
    Box2.setLength(12.0);
    Box2.setBreadth(13.0);
    Box2.setHeight(10.0);

    // Box1 的体积
    volume = Box1.getVolume();
    cout << "Volume of Box1 : " << volume <<endl;

    // Box2 的体积
    volume = Box2.getVolume();
    cout << "Volume of Box2 : " << volume <<endl;

    // 把两个对象相加，得到 Box3
    Box3 = Box1 + Box2;

    // Box3 的体积
    volume = Box3.getVolume();
    cout << "Volume of Box3 : " << volume <<endl;
}

```

```
    return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Volume of Box1 : 210
Volume of Box2 : 1560
Volume of Box3 : 5400
```

可重载运算符/不可重载运算符

下面是可重载的运算符列表：

双目算术运算符	<b>+(加), -(减), *(乘), /(除), %(取模)</b>
关系运算符	<b>==(等于), !=(不等于), &lt;(小于), &gt;(大于), &lt;=(小于等于), &gt;=(大于等于)</b>
逻辑运算符	<b>  (逻辑或), &amp;&amp;(逻辑与), !(逻辑非)</b>
单目运算符	<b>+(正), -(负), *(指针), &amp;(取地址)</b>
自增自减运算符	<b>++(自增), --(自减)</b>
位运算符	<b>  (按位或), &amp; (按位与), ~(按位取反), ^(按位异或), &lt;&lt; (左移), &gt;&gt; (右移)</b>
赋值运算符	<b>=, +=, -=, *=, /=, %=, &amp;=,  =, ^=, &lt;&lt;=, &gt;&gt;=</b>
空间申请与释放	<b>new, delete, new[ ], delete[]</b>
其他运算符	<b>()(函数调用), -(成员访问), ,(逗号),</b>

下面是不可重载的运算符列表：

- **::**：成员访问运算符
- **, ->**：成员指针访问运算符
- **:::**：域运算符
- **sizeof**：长度运算符
- **?:**：条件运算符
- **#**：预处理符号

运算符重载实例

下面提供了各种运算符重载的实例，帮助您更好地理解重载的概念。

序号	运算符和实例
1	<a href="#">一元运算符重载</a>
2	<a href="#">二元运算符重载</a>
3	<a href="#">关系运算符重载</a>
4	<a href="#">输入/输出运算符重载</a>
5	<a href="#">++ 和 -- 运算符重载</a>
6	<a href="#">赋值运算符重载</a>
7	<a href="#">函数调用运算符 () 重载</a>
8	<a href="#">[] 下标运算符重载</a>
9	<a href="#">类成员访问运算符 -&gt; 重载</a>

## C++ 一元运算符重载

一元运算符只对一个操作数进行操作，下面是一元运算符的实例：

- [递增运算符（++）和递减运算符（--）](#)
- 一元减运算符，即负号（-）
- 逻辑非运算符（!）

一元运算符通常出现在它们所操作的对象的前面，比如 !obj、-obj 和 ++obj，但有时它们也可以作为后缀，比如 obj++ 或 obj--。

下面的实例演示了如何重载一元减运算符（-）。

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

class Distance
{
private:
    int feet;           // 0 到无穷
    int inches;         // 0 到 12
public:
    // 所需的构造函数
    Distance(){
        feet = 0;
        inches = 0;
    }
    Distance(int f, int i){
        feet = f;
        inches = i;
    }
    // 显示距离的方法
```

```

    void displayDistance()
    {
        cout << "F: " << feet << " I:" << inches <<endl;
    }
    // 重载负运算符 ( - )
    Distance operator- ()
    {
        feet = -feet;
        inches = -inches;
        return Distance(feet, inches);
    }
};
int main()
{
    Distance D1(11, 10), D2(-5, 11);

    -D1;                // 取相反数
    D1.displayDistance(); // 距离 D1

    -D2;                // 取相反数
    D2.displayDistance(); // 距离 D2

    return 0;
}

```

当上面的代码被编译和执行时，它会产生下列结果：

```

F: -11 I:-10
F: 5 I:-11

```

## C++ 二元运算符重载

二元运算符需要两个参数，下面是二元运算符的实例。我们平常使用的加运算符（+）、减运算符（-）、乘运算符（\*）和除运算符（/）都属于二元运算符。就像加(+)运算符。

下面的实例演示了如何重载加运算符（+）。类似地，您也可以尝试重载减运算符（-）和除运算符（/）。

```

#include <iostream>
using namespace std;

class Box
{
    double length;    // 长度
    double breadth;   // 宽度
    double height;    // 高度
public:

    double getVolume(void)
    {
        return length * breadth * height;
    }
    void setLength( double len )

```

```

{
    length = len;
}

void setBreadth( double bre )
{
    breadth = bre;
}

void setHeight( double hei )
{
    height = hei;
}
// 重载 + 运算符，用于把两个 Box 对象相加
Box operator+(const Box& b)
{
    Box box;
    box.length = this->length + b.length;
    box.breadth = this->breadth + b.breadth;
    box.height = this->height + b.height;
    return box;
}
};
// 程序的主函数
int main( )
{
    Box Box1;           // 声明 Box1，类型为 Box
    Box Box2;           // 声明 Box2，类型为 Box
    Box Box3;           // 声明 Box3，类型为 Box
    double volume = 0.0; // 把体积存储在该变量中

    // Box1 详述
    Box1.setLength(6.0);
    Box1.setBreadth(7.0);
    Box1.setHeight(5.0);

    // Box2 详述
    Box2.setLength(12.0);
    Box2.setBreadth(13.0);
    Box2.setHeight(10.0);

    // Box1 的体积
    volume = Box1.getVolume();
    cout << "Volume of Box1 : " << volume << endl;

    // Box2 的体积
    volume = Box2.getVolume();
    cout << "Volume of Box2 : " << volume << endl;

    // 把两个对象相加，得到 Box3
    Box3 = Box1 + Box2;

    // Box3 的体积

```



```

    volume = Box3.getVolume();
    cout << "Volume of Box3 : " << volume <<endl;

    return 0;
}

```

当上面的代码被编译和执行时，它会产生下列结果：

```

Volume of Box1 : 210
Volume of Box2 : 1560
Volume of Box3 : 5400

```

## C++ 关系运算符重载

C++ 语言支持各种关系运算符（<、>、<=、>=、== 等等），它们可用于比较 C++ 内置的数据类型。

您可以重载任何一个关系运算符，重载后的关系运算符可用于比较类的对象。

下面的实例演示了如何重载 < 运算符，类似地，您也可以尝试重载其他的运算符。

```

#include <iostream>
using namespace std;

class Distance
{
private:
    int feet;           // 0 到无穷
    int inches;         // 0 到 12
public:
    // 所需的构造函数
    Distance(){
        feet = 0;
        inches = 0;
    }
    Distance(int f, int i){
        feet = f;
        inches = i;
    }
    // 显示距离的方法
    void displayDistance()
    {
        cout << "F: " << feet << " I:" << inches <<endl;
    }
    // 重载负运算符（ - ）
    Distance operator- ()
    {
        feet = -feet;
        inches = -inches;
        return Distance(feet, inches);
    }
    // 重载小于运算符（ < ）
    bool operator <(const Distance& d)
    {

```

```

        if(feet < d.feet)
        {
            return true;
        }
        if(feet == d.feet && inches < d.inches)
        {
            return true;
        }
        return false;
    }
};

int main()
{
    Distance D1(11, 10), D2(5, 11);

    if( D1 < D2 )
    {
        cout << "D1 is less than D2 " << endl;
    }
    else
    {
        cout << "D2 is less than D1 " << endl;
    }
    return 0;
}

```

当上面的代码被编译和执行时，它会产生下列结果：

```
D2 is less than D1
```

## C++ 输入/输出运算符重载

C++ 能够使用流提取运算符 >> 和流插入运算符 << 来输入和输出内置的数据类型。您可以重载流提取运算符和流插入运算符来操作对象等用户自定义的数据类型。

在这里，有一点很重要，我们需要把运算符重载函数声明为类的友元函数，这样我们就能不用创建对象而直接调用函数。

下面的实例演示了如何重载提取运算符 >> 和插入运算符 <<。

```

#include <iostream>
using namespace std;

class Distance
{
private:
    int feet;           // 0 到无穷
    int inches;         // 0 到 12
public:
    // 所需的构造函数
    Distance(){
        feet = 0;
    }
};

```

```

        inches = 0;
    }
    Distance(int f, int i){
        feet = f;
        inches = i;
    }
    friend ostream &operator<<( ostream &output,
                                const Distance &D )
    {
        output << "F : " << D.feet << " I : " << D.inches;
        return output;
    }

    friend istream &operator>>( istream &input, Distance &D )
    {
        input >> D.feet >> D.inches;
        return input;
    }
};

int main()
{
    Distance D1(11, 10), D2(5, 11), D3;

    cout << "Enter the value of object : " << endl;
    cin >> D3;
    cout << "First Distance : " << D1 << endl;
    cout << "Second Distance : " << D2 << endl;
    cout << "Third Distance : " << D3 << endl;

    return 0;
}

```

当上面的代码被编译和执行时，它会产生下列结果：

```

$./a.out
Enter the value of object :
70
10
First Distance : F : 11 I : 10
Second Distance : F : 5 I : 11
Third Distance : F : 70 I : 10

```

## C++ ++ 和 -- 运算符重载

递增运算符（++）和递减运算符（--）是 C++ 语言中两个重要的一元运算符。

下面的实例演示了如何重载递增运算符（++），包括前缀和后缀两种用法。类似地，您也可以尝试重载递减运算符（--）。

```

#include <iostream>
using namespace std;

```

```

class Time
{
private:
    int hours;           // 0 到 23
    int minutes;         // 0 到 59
public:
    // 所需的构造函数
    Time(){
        hours = 0;
        minutes = 0;
    }
    Time(int h, int m){
        hours = h;
        minutes = m;
    }
    // 显示时间的方法
    void displayTime()
    {
        cout << "H: " << hours << " M:" << minutes << endl;
    }
    // 重载前缀递增运算符 ( ++ )
    Time operator++ ()
    {
        ++minutes;       // 对象加 1
        if(minutes >= 60)
        {
            ++hours;
            minutes -= 60;
        }
        return Time(hours, minutes);
    }
    // 重载后缀递增运算符 ( ++ )
    Time operator++( int )
    {
        // 保存原始值
        Time T(hours, minutes);
        // 对象加 1
        ++minutes;
        if(minutes >= 60)
        {
            ++hours;
            minutes -= 60;
        }
        // 返回旧的原始值
        return T;
    }
};

int main()
{
    Time T1(11, 59), T2(10,40);

    ++T1;                // T1 加 1

```

```

    T1.displayTime();           // 显示 T1
    ++T1;                       // T1 再加 1
    T1.displayTime();           // 显示 T1

    T2++;                       // T2 加 1
    T2.displayTime();           // 显示 T2
    T2++;                       // T2 再加 1
    T2.displayTime();           // 显示 T2
    return 0;
}

```

当上面的代码被编译和执行时，它会产生下列结果：

```

H: 12 M:0
H: 12 M:1
H: 10 M:41
H: 10 M:42

```

## C++ 赋值运算符重载

就像其他运算符一样，您可以重载赋值运算符（=），用于创建一个对象，比如拷贝构造函数。

下面的实例演示了如何重载赋值运算符。

```

#include <iostream>
using namespace std;

class Distance
{
private:
    int feet;           // 0 到无穷
    int inches;          // 0 到 12
public:
    // 所需的构造函数
    Distance(){
        feet = 0;
        inches = 0;
    }
    Distance(int f, int i){
        feet = f;
        inches = i;
    }
    void operator=(const Distance &D )
    {
        feet = D.feet;
        inches = D.inches;
    }
    // 显示距离的方法
    void displayDistance()
    {
        cout << "F: " << feet << " I:" << inches << endl;
    }
}

```

```

};
int main()
{
    Distance D1(11, 10), D2(5, 11);

    cout << "First Distance : ";
    D1.displayDistance();
    cout << "Second Distance :";
    D2.displayDistance();

    // 使用赋值运算符
    D1 = D2;
    cout << "First Distance :";
    D1.displayDistance();

    return 0;
}

```

当上面的代码被编译和执行时，它会产生下列结果：

```

First Distance : F: 11 I:10
Second Distance :F: 5 I:11
First Distance :F: 5 I:11

```

## C++ 函数调用运算符 () 重载

函数调用运算符 () 可以被重载用于类的对象。当重载 () 时，您不是创造了一种新的调用函数的方式，相反地，这是创建一个可以传递任意数目参数的运算符函数。

下面的实例演示了如何重载函数调用运算符 ()。

```

#include <iostream>
using namespace std;

class Distance
{
private:
    int feet;           // 0 到无穷
    int inches;         // 0 到 12
public:
    // 所需的构造函数
    Distance(){
        feet = 0;
        inches = 0;
    }
    Distance(int f, int i){
        feet = f;
        inches = i;
    }
    // 重载函数调用运算符
    Distance operator()(int a, int b, int c)

```

```

    {
        Distance D;
        // 进行随机计算
        D.feet = a + c + 10;
        D.inches = b + c + 100 ;
        return D;
    }
    // 显示距离的方法
    void displayDistance()
    {
        cout << "F: " << feet << " I:" << inches << endl;
    }
};
int main()
{
    Distance D1(11, 10), D2;

    cout << "First Distance : ";
    D1.displayDistance();

    D2 = D1(10, 10, 10); // invoke operator()
    cout << "Second Distance :";
    D2.displayDistance();

    return 0;
}

```

当上面的代码被编译和执行时，它会产生下列结果：

```

First Distance : F: 11 I:10
Second Distance :F: 30 I:120

```

## C++ 下标运算符 [] 重载

下标操作符 [] 通常用于访问数组元素。重载该运算符用于增强操作 C++ 数组的功能。

下面的实例演示了如何重载下标运算符 []。

```

#include <iostream>
using namespace std;
const int SIZE = 10;

class safearray
{
private:
    int arr[SIZE];
public:
    safearray()
    {
        register int i;
        for(i = 0; i < SIZE; i++)

```

```

        {
            arr[i] = i;
        }
    }
    int& operator[](int i)
    {
        if( i > SIZE )
        {
            cout << "索引超过最大值" <<endl;
            // 返回第一个元素
            return arr[0];
        }
        return arr[i];
    }
};
int main()
{
    safearray A;

    cout << "A[2] 的值为：" << A[2] <<endl;
    cout << "A[5] 的值为：" << A[5]<<endl;
    cout << "A[12] 的值为：" << A[12]<<endl;

    return 0;
}

```

当上面的代码被编译和执行时，它会产生下列结果：

```

$ g++ -o test test.cpp
$ ./test
A[2] 的值为：2
A[5] 的值为：5
A[12] 的值为：索引超过最大值
0

```

## C++ 类成员访问运算符 -> 重载

类成员访问运算符（->）可以被重载，但它较为麻烦。它被定义用于为一个类赋予"指针"行为。运算符->必须是一个成员函数。如果使用了->运算符，返回类型必须是指针或者是类的对象。

运算符->通常与指针引用运算符\*结合使用，用于实现"智能指针"的功能。这些指针是行为与正常指针相似的对象，唯一不同的是，当您通过指针访问对象时，它们会执行其他的任务。比如，当指针销毁时，或者当指针指向另一个对象时，会自动删除对象。

间接引用运算符->可被定义为一个一元后缀运算符。也就是说，给出一个类：

```

class Ptr{
    //...
    X * operator->();
};

```

类 **Ptr** 的对象可用于访问类 **X** 的成员，使用方式与指针的用法十分相似。例如：



```

void f(Ptr p )
{
    p->m = 10 ; // (p.operator->())->m = 10
}

```

语句 `p->m` 被解释为 `(p.operator->())->m`。同样地，下面的实例演示了如何重载类成员访问运算符 `->`。

```

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

// 假设一个实际的类
class Obj {
    static int i, j;
public:
    void f() const { cout << i++ << endl; }
    void g() const { cout << j++ << endl; }
};

// 静态成员定义
int Obj::i = 10;
int Obj::j = 12;

// 为上面的类实现一个容器
class ObjContainer {
    vector<Obj*> a;
public:
    void add(Obj* obj)
    {
        a.push_back(obj); // 调用向量的标准方法
    }
    friend class SmartPointer;
};

// 实现智能指针，用于访问类 Obj 的成员
class SmartPointer {
    ObjContainer oc;
    int index;
public:
    SmartPointer(ObjContainer& objc)
    {
        oc = objc;
        index = 0;
    }
    // 返回值表示列表结束
    bool operator++() // 前缀版本
    {
        if(index >= oc.a.size()) return false;
        if(oc.a[++index] == 0) return false;
        return true;
    }
    bool operator++(int) // 后缀版本

```

```

    {
        return operator++();
    }
    // 重载运算符 ->
    Obj* operator->() const
    {
        if(!oc.a[index])
        {
            cout << "Zero value";
            return (Obj*)0;
        }
        return oc.a[index];
    }
};

int main() {
    const int sz = 10;
    Obj o[sz];
    ObjContainer oc;
    for(int i = 0; i < sz; i++)
    {
        oc.add(&o[i]);
    }
    SmartPointer sp(oc); // 创建一个迭代器
    do {
        sp->f(); // 智能指针调用
        sp->g();
    } while(sp++);
    return 0;
}

```

当上面的代码被编译和执行时，它会产生下列结果：

```

10
12
11
13
12
14
13
15
14
16
15
17
16
18
17
19
18
20
19
21

```

---

## C++ 多态

---

多态按字面的意思就是多种形态。当类之间存在层次结构，并且类之间是通过继承关联时，就会用到多态。

C++ 多态意味着调用成员函数时，会根据调用函数的对象的类型来执行不同的函数。

下面的实例中，基类 Shape 被派生为两个类，如下所示：

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

class Shape {
protected:
    int width, height;
public:
    Shape( int a=0, int b=0)
    {
        width = a;
        height = b;
    }
    int area()
    {
        cout << "Parent class area :" <<endl;
        return 0;
    }
};

class Rectangle: public Shape{
public:
    Rectangle( int a=0, int b=0):Shape(a, b) { }
    int area ()
    {
        cout << "Rectangle class area :" <<endl;
        return (width * height);
    }
};

class Triangle: public Shape{
public:
    Triangle( int a=0, int b=0):Shape(a, b) { }
    int area ()
    {
        cout << "Triangle class area :" <<endl;
        return (width * height / 2);
    }
};

// 程序的主函数
int main( )
{
    Shape *shape;
    Rectangle rec(10,7);
    Triangle tri(10,5);

    // 存储矩形的地址
```

```

    shape = &rec;
    // 调用矩形的求面积函数 area
    shape->area();

    // 存储三角形的地址
    shape = &tri;
    // 调用三角形的求面积函数 area
    shape->area();

    return 0;
}

```

当上面的代码被编译和执行时，它会产生下列结果：

```

Parent class area
Parent class area

```

导致错误输出的原因是，调用函数 `area()` 被编译器设置为基类中的版本，这就是所谓的静态多态，或静态链接 - 函数调用在程序执行前就准备好了。有时候这也被称为早绑定，因为 `area()` 函数在程序编译期间就已经设置好了。

但现在，让我们对程序稍作修改，在 `Shape` 类中，`area()` 的声明前放置关键字 **virtual**，如下所示：

```

class Shape {
protected:
    int width, height;
public:
    Shape( int a=0, int b=0)
    {
        width = a;
        height = b;
    }
    virtual int area()
    {
        cout << "Parent class area :" <<endl;
        return 0;
    }
};

```

修改后，当编译和执行前面的实例代码时，它会产生以下结果：

```

Rectangle class area
Triangle class area

```

此时，编译器看的是指针的内容，而不是它的类型。因此，由于 `tri` 和 `rec` 类的对象的地址存储在 `*shape` 中，所以会调用各自的 `area()` 函数。

正如您所看到的，每个子类都有一个函数 `area()` 的独立实现。这就是多态的一般使用方式。有了多态，您可以有多个不同的类，都带有同一个名称但具有不同实现的函数，函数的参数甚至可以是相同的。

## 虚函数

**虚函数** 是在基类中使用关键字 **virtual** 声明的函数。在派生类中重新定义基类中定义的虚函数时，会告诉编译器不要静态链接到该函数。

我们想要的是在程序中任意点可以根据所调用的对象类型来选择调用的函数，这种操作被称为**动态链接**，或**后期绑定**。

## 纯虚函数

您可能想要在基类中定义虚函数，以便在派生类中重新定义该函数更好地适用于对象，但是您在基类中又不能对虚函数给出有意义的实现，这个时候就会用到纯虚函数。

我们可以把基类中的虚函数 `area()` 改写如下：

```
class Shape {
protected:
    int width, height;
public:
    Shape( int a=0, int b=0)
    {
        width = a;
        height = b;
    }
    // pure virtual function
    virtual int area() = 0;
};
```

`= 0` 告诉编译器，函数没有主体，上面的虚函数是**纯虚函数**。

- 1、纯虚函数声明如下：`virtual void funtion1()=0`; 纯虚函数一定没有定义，纯虚函数用来规范派生类的行为，即接口。包含纯虚函数的类是抽象类，抽象类不能定义实例，但可以声明指向实现该抽象类的具体类的指针或引用。
- 2、虚函数声明如下：`virtual Return Type FunctionName(Parameter)` 虚函数必须实现，如果不实现，编译器将报错，错误提示为：

```
error LNK****: unresolved external symbol "public: virtual void __thiscall
ClassName::virtualFunctionName(void)"
```

- 3、对于虚函数来说，父类和子类都有各自的版本。由多态方式调用的时候动态绑定。
- 4、实现了纯虚函数的子类，该纯虚函数在子类中就编程了虚函数，子类的子类即孙子类可以覆盖该虚函数，由多态方式调用的时候动态绑定。
- 5、虚函数是C++中用于实现多态(polymorphism)的机制。核心理念就是通过基类访问派生类定义的函数。
- 6、在有动态分配堆上内存的时候，析构函数必须是虚函数，但没有必要是纯虚的。
- 7、友元不是成员函数，只有成员函数才可以是虚拟的，因此友元不能是虚拟函数。但可以通过让友元函数调用虚拟成员函数来解决友元的虚拟问题。
- 8、析构函数应当是虚函数，将调用相应对象类型的析构函数，因此，如果指针指向的是子类对象，将调用子类的析构函数，然后自动调用基类的析构函数。

## C++ 数据抽象

数据抽象是指，只向外界提供关键信息，并隐藏其后台的实现细节，即只表现必要的信息而不呈现细节。

数据抽象是一种依赖于接口和实现分离的编程（设计）技术。

让我们举一个现实生活中的真实例子，比如一台电视机，您可以打开和关闭、切换频道、调整音量、添加外部组件（如喇叭、录像机、DVD 播放器），但是您不知道它的内部实现细节，也就是说，您并不知道它是如何通过缆线接收信号，如何转换信号，并最终显示在屏幕上。

因此，我们可以说电视把它的内部实现和外部接口分离开了，您无需知道它的内部实现原理，直接通过它的外部接口（比如电源按钮、遥控器、声量控制器）就可以操控电视。

现在，让我们言归正传，就 C++ 编程而言，C++ 类为**数据抽象**提供了可能。它们向外界提供了大量用于操作对象数据的公共方法，也就是说，外界实际上并不清楚类的内部实现。

例如，您的程序可以调用 **sort()** 函数，而不需要知道函数中排序数据所用到的算法。实际上，函数排序的底层实现会因库的版本不同而有所差异，只要接口不变，函数调用就可以照常工作。

在 C++ 中，我们使用类来定义我们自己的抽象数据类型（ADT）。您可以使用类 **ostream** 的 **cout** 对象来输出数据到标准输出，如下所示：

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

int main( )
{
    cout << "Hello C++" <<endl;
    return 0;
}
```

在这里，您不需要理解 **cout** 是如何在用户的屏幕上显示文本。您只需要知道公共接口即可，**cout** 的底层实现可以自由改变。

### 访问标签强制抽象

在 C++ 中，我们使用访问标签来定义类的抽象接口。一个类可以包含零个或多个访问标签：

- 使用公共标签定义的成员都可以访问该程序的所有部分。一个类型的数据抽象视图是由它的公共成员来定义的。
- 使用私有标签定义的成员无法访问到使用类的代码。私有部分对使用类型的代码隐藏了实现细节。

访问标签出现的频率没有限制。每个访问标签指定了紧随其后的成员定义的访问级别。指定的访问级别会一直有效，直到遇到下一个访问标签或者遇到类主体的关闭右括号为止。

### 数据抽象的好处

数据抽象有两个重要的优势：

- 类的内部受到保护，不会因无意的用户级错误导致对象状态受损。
- 类实现可能随着时间的推移而发生变化，以便应对不断变化的需求，或者应对那些要求不改变用户级代码的错误报告。

如果只在类的私有部分定义数据成员，编写该类的作者就可以随意更改数据。如果实现发生改变，则只需要检查类的代码，看看这个改变会导致哪些影响。如果数据是公有的，则任何直接访问旧表示形式的数据成员的函数都可能受到影响。

### 数据抽象的实例

C++ 程序中，任何带有公有和私有成员类都可以作为数据抽象的实例。请看下面的实例：

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

class Adder{
public:
    // 构造函数
    Adder(int i = 0)
    {
        total = i;
    }
    // 对外的接口
    void addNum(int number)
    {
        total += number;
    }
    // 对外的接口
    int getTotal()
    {
        return total;
    };
private:
    // 对外隐藏的数据
    int total;
};

int main( )
{
    Adder a;

    a.addNum(10);
    a.addNum(20);
    a.addNum(30);

    cout << "Total " << a.getTotal() << endl;
    return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Total 60
```

上面的类把数字相加，并返回总和。公有成员 **addNum** 和 **getTotal** 是对外的接口，用户需要知道它们以便使用类。私有成员 **total** 是用户不需要了解的，但又是类能正常工作所必需的。

### 设计策略

抽象把代码分离为接口和实现。所以在设计组件时，必须保持接口独立于实现，这样，如果改变底层实现，接口也将保持不变。

在这种情况下，不管任何程序使用接口，接口都不会受到影响，只需要将最新的实现重新编译即可。

# C++ 数据封装

所有的 C++ 程序都有以下两个基本要素：

- 程序语句（代码）：这是程序中执行动作的部分，它们被称为函数。
- 程序数据：数据是程序的信息，会受到程序函数的影响。

封装是面向对象编程中的把数据和操作数据的函数绑定在一起的一个概念，这样能避免受到外界的干扰和误用，从而确保了安全。数据封装引申出了另一个重要的 OOP 概念，即数据隐藏。

数据封装是一种把数据和操作数据的函数捆绑在一起的机制，数据抽象是一种仅向用户暴露接口而把具体的实现细节隐藏起来的机制。

C++ 通过创建类来支持封装和数据隐藏（`public`、`protected`、`private`）。我们已经知道，类包含私有成员（`private`）、保护成员（`protected`）和公有成员（`public`）成员。默认情况下，在类中定义的所有项目都是私有的。例如：

```
class Box
{
    public:
        double getVolume(void)
        {
            return length * breadth * height;
        }
    private:
        double length;    // 长度
        double breadth;   // 宽度
        double height;    // 高度
};
```

变量 `length`、`breadth` 和 `height` 都是私有的（`private`）。这意味着它们只能被 `Box` 类中的其他成员访问，而不能被程序中其他部分访问。这是实现封装的一种方式。

为了使类中的成员变成公有的（即，程序中的其他部分也能访问），必须在这些成员前使用 **public** 关键字进行声明。所有定义在 `public` 标识符后边的变量或函数可以被程序中所有其他的函数访问。

把一个类定义为另一个类的友元类，会暴露实现细节，从而降低了封装性。理想的做法是尽可能地对外隐藏每个类的实现细节。

## 数据封装的实例

C++ 程序中，任何带有公有和私有成员类都可以作为数据封装和数据抽象的实例。请看下面的实例：

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

class Adder{
    public:
        // 构造函数
        Adder(int i = 0)
        {
            total = i;
        }
};
```



```

    // 对外的接口
    void addNum(int number)
    {
        total += number;
    }
    // 对外的接口
    int getTotal()
    {
        return total;
    };
private:
    // 对外隐藏的数据
    int total;
};
int main( )
{
    Adder a;

    a.addNum(10);
    a.addNum(20);
    a.addNum(30);

    cout << "Total " << a.getTotal() << endl;
    return 0;
}

```

当上面的代码被编译和执行时，它会产生下列结果：

```
Total 60
```

上面的类把数字相加，并返回总和。公有成员 **addNum** 和 **getTotal** 是对外的接口，用户需要知道它们以便使用类。私有成员 **total** 是对外隐藏的，用户不需要了解它，但它又是类能正常工作所必需的。

### 设计策略

通常情况下，我们都会设置类成员状态为私有（**private**），除非我们真的需要将其暴露，这样才能保证良好的封装性。

这通常应用于数据成员，但它同样适用于所有成员，包括虚函数。

## C++ 接口（抽象类）

接口描述了类的行为和功能，而不需要完成类的特定实现。

C++ 接口是使用**抽象类**来实现的，抽象类与数据抽象互不混淆，数据抽象是一个把实现细节与相关的数据分离开的概念。

如果类中至少有一个函数被声明为纯虚函数，则这个类就是抽象类。纯虚函数是通过在声明中使用 "**= 0**" 来指定的，如下所示：

```

class Box
{
public:
    // 纯虚函数
    virtual double getVolume() = 0;
private:
    double length;    // 长度
    double breadth;   // 宽度
    double height;    // 高度
};

```

设计**抽象类**（通常称为 **ABC**）的目的，是为了给其他类提供一个可以继承的适当的基类。抽象类不能被用于实例化对象，它只能作为**接口**使用。如果试图实例化一个抽象类的对象，会导致编译错误。

因此，如果一个 **ABC** 的子类需要被实例化，则必须实现每个虚函数，这也意味着 **C++** 支持使用 **ABC** 声明接口。如果没有在派生类中重载纯虚函数，就尝试实例化该类的对象，会导致编译错误。

可用于实例化对象的类被称为**具体类**。

### 抽象类的实例

请看下面的实例，基类 **Shape** 提供了一个接口 **getArea()**，在两个派生类 **Rectangle** 和 **Triangle** 中分别实现了 **getArea()**:

```

#include <iostream>

using namespace std;

// 基类
class Shape
{
public:
    // 提供接口框架的纯虚函数
    virtual int getArea() = 0;
    void setWidth(int w)
    {
        width = w;
    }
    void setHeight(int h)
    {
        height = h;
    }
protected:
    int width;
    int height;
};

// 派生类
class Rectangle: public Shape
{
public:
    int getArea()
    {

```

```

        return (width * height);
    }
};
class Triangle: public Shape
{
public:
    int getArea()
    {
        return (width * height)/2;
    }
};

int main(void)
{
    Rectangle Rect;
    Triangle Tri;

    Rect.setWidth(5);
    Rect.setHeight(7);
    // 输出对象的面积
    cout << "Total Rectangle area: " << Rect.getArea() << endl;

    Tri.setWidth(5);
    Tri.setHeight(7);
    // 输出对象的面积
    cout << "Total Triangle area: " << Tri.getArea() << endl;

    return 0;
}

```

当上面的代码被编译和执行时，它会产生下列结果：

```

Total Rectangle area: 35
Total Triangle area: 17

```

从上面的实例中，我们可以看到一个抽象类是如何定义一个接口 `getArea()`，两个派生类是如何通过不同的计算面积的算法来实现这个相同的函数。

### 设计策略

面向对象的系统可能会使用一个抽象基类为所有的外部应用程序提供一个适当的、通用的、标准化的接口。然后，派生类通过继承抽象基类，就把所有类似的操作都继承下来。

外部应用程序提供的功能（即公有函数）在抽象基类中是以纯虚函数的形式存在的。这些纯虚函数在相应的派生类中被实现。

这个架构也使得新的应用程序可以很容易地被添加到系统中，即使是在系统被定义之后依然可以如此。

## C++ 高级教程

看情况在添加。。。