**C++数据结构**

参考网站：

<https://zh.cppreference.com/w/cpp/container/stack>

<http://www.cplusplus.com/reference/stack/stack/?kw=stack>

## vector

**C++ 简介**

C++ 是一种静态类型的、编译式的、通用的、大小写敏感的、不规则的编程语言，支持过程化编程、面向对象编程和泛型编程。

C++ 被认为是一种**中级**语言，它综合了高级语言和低级语言的特点。

C++ 是由 Bjarne Stroustrup 于 1979 年在新泽西州美利山贝尔实验室开始设计开发的。C++ 进一步扩充和完善了 C 语言，最初命名为带类的C，后来在 1983 年更名为 C++。

C++ 是 C 的一个超集，事实上，任何合法的 C 程序都是合法的 C++ 程序。

**注意：**使用静态类型的编程语言是在编译时执行类型检查，而不是在运行时执行类型检查。

**面向对象程序设计**

C++ 完全支持面向对象的程序设计，包括面向对象开发的四大特性：

* 封装
* 抽象
* 继承
* 多态

**标准库**

标准的 C++ 由三个重要部分组成：

* 核心语言，提供了所有构件块，包括变量、数据类型和常量，等等。
* C++ 标准库，提供了大量的函数，用于操作文件、字符串等。
* 标准模板库（STL），提供了大量的方法，用于操作数据结构等。

**ANSI 标准**

ANSI 标准是为了确保 C++ 的便携性 —— 您所编写的代码在 Mac、UNIX、Windows、Alpha 计算机上都能通过编译。

由于 ANSI 标准已稳定使用了很长的时间，所有主要的 C++ 编译器的制造商都支持 ANSI 标准。

**学习 C++**

学习 C++，关键是要理解概念，而不应过于深究语言的技术细节。

学习程序设计语言的目的是为了成为一个更好的程序员，也就是说，是为了能更有效率地设计和实现新系统，以及维护旧系统。

C++ 支持多种编程风格。您可以使用 Fortran、C、Smalltalk 等任意一种语言的编程风格来编写代码。每种风格都能有效地保证运行时间效率和空间效率。

**C++ 的使用**

基本上每个应用程序领域的程序员都有使用 C++。

C++ 通常用于编写设备驱动程序和其他要求实时性的直接操作硬件的软件。

C++ 广泛用于教学和研究。

任何一个使用苹果电脑或 Windows PC 机的用户都在间接地使用 C++，因为这些系统的主要用户接口是使用 C++ 编写的。

**标准化**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **发布时间** | **文档** | **通称** | **备注** |  |
| 2015 | ISO/IEC TS 19570:2015 | - | 用于并行计算的扩展 |  |
| 2015 | ISO/IEC TS 18822:2015 | - | 文件系统 |  |
| 2014 | ISO/IEC 14882:2014 | C++14 | 第四个C++标准 |  |
| 2011 | ISO/IEC TR 24733:2011 | - | 十进制浮点数扩展 |  |
| 2011 | ISO/IEC 14882:2011 | C++11 | 第三个C++标准 |  |
| 2010 | ISO/IEC TR 29124:2010 | - | 数学函数扩展 |  |
| 2007 | ISO/IEC TR 19768:2007 | C++TR1 | C++技术报告：库扩展 |  |
| 2006 | ISO/IEC TR 18015:2006 | - | C++性能技术报告 |  |
| 2003 | ISO/IEC 14882:2003 | C++03 | 第二个C++标准 |  |
| 1998 | ISO/IEC 14882:1998 | C++98 | 第一个C++标准 |  |

## C++中 "\n" 与 endl 的区别是什么?

"\n" 表示内容为一个回车符的字符串。std::endl 是流操作子，输出的作用和输出 "\n" 类似，但可能略有区别。

std::endl 输出一个换行符，并立即刷新缓冲区。

例如:

std::cout << std::endl;

相当于:

std::cout << '\n' << std::flush;

或者

std::cout << '\n'; std::fflush(stdout);

由于流操作符 << 的重载，对于 '\n' 和 "\n"，输出效果相同。

对于有输出缓冲的流（例如cout、clog），如果不手动进行缓冲区刷新操作，将在缓冲区满后自动刷新输出。不过对于 cout 来说（相对于文件输出流等），缓冲一般体现得并不明显。但是必要情况下使用 endl 代替 '\n' 一般是个好习惯。

对于无缓冲的流（例如标准错误输出流cerr），刷新是不必要的，可以直接使用 '\n'。

## 在 C++ 中，分别使用.h 和.cpp 来定义一个类。

**.h** 中存放类的声明，函数原型（放在类的声明中）。

**.cpp** 存放函数体。

也就是说，一个存放声明(declaration)，一个存放定义（definition)。

如果我们在一个头文件里声明了一个函数，当我们需要定义这个函数（这个定义是唯一的，也就是只能定义一次），或者需要使用这个函数时，我们在 cpp 中需要 include 这个头文件。

同样地，如果我们在一个头文件里声明了一个类，当我们需要定义类里的成员函数，或者我们需要使用这个类时，我们在 cpp 中需要 include 这个头文件。

对于类的设计者来说，头文件就像他们和类的使用者的一个合同，编译器会强化这一合同，它会要求你在使用这些类里的函数或结构时必须要声明。

## 头文件格式

**<>** 先去系统目录中找头文件，如果没有在到当前目录下找。所以像标准的头文件 stdio.h、stdlib.h 等用这个方法。

**" "** 首先在当前目录下寻找，如果找不到，再到系统目录中寻找。 这个用于 include 自定义的头文件，让系统优先使用当前目录中定义的。

## 如何显示多行文本

如果想显示多行文本，如下：

#include <iostream>

using namespace std;

int main()

{

cout<<"...............\n"

<<"Hello, world!\n"

    <<"Welcome to c++\n"

<<"...............\n";

    return 0;

}

不用一直这样 cout 多行插入。

## 避免使用using namespace std

真正的开发过程中， 尽量避免使用 **using namespace std;** 等直接引入整个命名空间，否则会因为命名空间污染导致很多不必要的问题， 比如自己写的某个函数，名称正好和 **std** 中的一样， 编译器会不知道使用哪一个， 引起编译报错， 建议使用:

std::cout << "Hello World" << std::endl;

等直接由命名空间组合起来的全称。

**C++ 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。

**C++ 程序结构**

让我们看一段简单的代码，可以输出单词 *Hello World*。

**实例**

#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; }

接下来我们讲解一下上面这段程序：

* 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 **<iostream>**。
* 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。

**编译 & 执行 C++ 程序**

接下来让我们看看如何把源代码保存在一个文件中，以及如何编译并运行它。下面是简单的步骤：

* 打开一个文本编辑器，添加上述代码。
* 保存文件为 hello.cpp。
* 打开命令提示符，进入到保存文件所在的目录。
* 键入 'g++ hello.cpp '，输入回车，编译代码。如果代码中没有错误，命令提示符会跳到下一行，并生成 a.out 可执行文件。
* 现在，键入 ' a.out' 来运行程序。
* 您可以看到屏幕上显示 ' Hello World '。

$ g++ hello.cpp

$ ./a.out

Hello World

请确保您的路径中已包含 g++ 编译器，并确保在包含源文件 hello.cpp 的目录中运行它。

您也可以使用 makefile 来编译 C/C++ 程序。

**C++ 中的分号 & 语句块**

在 C++ 中，分号是语句结束符。也就是说，每个语句必须以分号结束。它表明一个逻辑实体的结束。

例如，下面是三个不同的语句：

x = y; y = y+1; add(x, y);

语句块是一组使用大括号括起来的按逻辑连接的语句。例如：

{ cout << "Hello World"; // 输出 Hello World return 0; }

C++ 不以行末作为结束符的标识，因此，您可以在一行上放置多个语句。例如：

x = y; y = y+1; add(x, y);

等同于

x = y; y = y+1; add(x, y);

**C++ 标识符**

C++ 标识符是用来标识变量、函数、类、模块，或任何其他用户自定义项目的名称。一个标识符以字母 A-Z 或 a-z 或下划线 \_ 开始，后跟零个或多个字母、下划线和数字（0-9）。

C++ 标识符内不允许出现标点字符，比如 @、& 和 %。C++ 是区分大小写的编程语言。因此，在 C++ 中，**Manpower** 和 **manpower** 是两个不同的标识符。

下面列出几个有效的标识符：

mohd zara abc move\_name a\_123

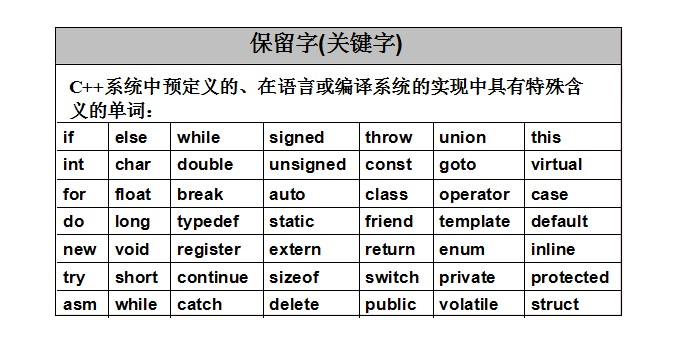
myname50 \_temp j a23b9 retVal

**C++ 关键字**

下表列出了 C++ 中的保留字。这些保留字不能作为常量名、变量名或其他标识符名称。

|  |  |  |  |
| --- | --- | --- | --- |
| asm | else | new | this |
| auto | enum | operator | throw |
| bool | explicit | private | true |
| break | export | protected | try |
| case | extern | public | typedef |
| catch | false | register | typeid |
| char | float | reinterpret\_cast | typename |
| class | for | return | union |
| const | friend | short | unsigned |
| const\_cast | goto | signed | using |
| continue | if | sizeof | virtual |
| default | inline | static | void |
| delete | int | static\_cast | volatile |
| do | long | struct | wchar\_t |
| double | mutable | switch | while |
| dynamic\_cast | namespace | template |  |

完整关键字介绍可查阅：[C++ 的关键字（保留字）完整介绍](https://www.runoob.com/w3cnote/cpp-keyword-intro.html)



**1. asm**

asm (指令字符串)：允许在 C++ 程序中嵌入汇编代码。

**2. auto**

auto（自动，automatic）是存储类型标识符，表明变量"自动"具有本地范围，块范围的变量声明（如for循环体内的变量声明）默认为auto存储类型。

**3. bool**

bool（布尔）类型，C++ 中的基本数据结构，其值可选为 true（真）或者 false（假）。C++ 中的 bool 类型可以和 int 混用，具体来说就是 0 代表 false，非 0 代表 true。bool 类型常用于条件判断和函数返回值。

**4. break**

break（中断、跳出），用在switch语句或者循环语句中。程序遇到 break 后，即跳过该程序段，继续后面的语句执行。

**5. case**

用于 switch 语句中，用于判断不同的条件类型。

**6. catch**

catch 和 try 语句一起用于异常处理。

**7. char**

char（字符，character）类型，C++ 中的基本数据结构，其值一般为 0~255 的 int。这 256 个字符对应着 256 个 ASCII 码。char 类型的数据需要用单引号 **'** 括起来。

**8.class**

class（类）是 C++ 面向对象设计的基础。使用 class 关键字声明一个类。

**9. const**

const（常量的，constant）所修饰的对象或变量不能被改变，修饰函数时，该函数不能改变在该函数外面声明的变量也不能调用任何非const函数。在函数的声明与定义时都要加上const，放在函数参数列表的最后一个括号后。在 C++ 中，用 const 声明一个变量，意味着该变量就是一个带类型的常量，可以代替 #define，且比 #define 多一个类型信息，且它执行内链接，可放在头文件中声明；但在 C 中，其声明则必须放在源文件（即 .C 文件）中，在 C 中 const 声明一个变量，除了不能改变其值外，它仍是一具变量。如:

const double pi(3.14159);

或

const double pi = 3.14159;

**10. const\_cast用法：**

const\_cast<type\_id> (expression)

该运算符用来修改类型的 const 或 volatile 属性。除了 const 或 volatile 修饰之外， type\_id 和 expression 的类型是一样的。常量指针被转化成非常量指针，并且仍然指向原来的对象；常量引用被转换成非常量引用，并且仍然指向原来的对象；常量对象被转换成非常量对象。

**11. continue**

continue（继续）关键字用于循环结构。它使程序跳过代码段后部的部分，与 break 不同的是，continue 不是进入代码段后的部分执行，而是重新开始新的循环。因而它是"继续循环"之意，不是 break（跳出）。

**12. default**

default（默认、缺省）用于 switch 语句。当 switch 所有的 case 都不满足时，将进入 default 执行。default 只能放在 switch 语句所有的 case 之后，并且是可选的。

**13. delete**

delete（删除）释放程序动态申请的内存空间。delete 后面通常是一个指针或者数组 []，并且只能 delete 通过 new 关键字申请的指针，否则会发生段错误。

**14. do**

do-while是一类循环结构。与while循环不同，do-while循环保证至少要进入循环体一次。

**15. double**

double（双精度）类型，C++ 中的基本数据结构，以双精度形式存储一个浮点数。

**16. dynamic\_cast**

dynamic\_cast（动态转换），允许在运行时刻进行类型转换，从而使程序能够在一个类层次结构安全地转换类型。dynamic\_cast 提供了两种转换方式，把基类指针转换成派生类指针，或者把指向基类的左值转换成派生类的引用。

**17. else**

else 紧跟在 if 后面，用于对 if 不成立的情况的选择。

**18. enum**

enum（枚举）类型，给出一系列固定的值，只能在这里面进行选择一个。

**19. explicit**

explicit（显式的）的作用是"禁止单参数构造函数"被用于自动型别转换，其中比较典型的例子就是容器类型。在这种类型的构造函数中你可以将初始长度作为参数传递给构造函数。

**20. export**

为了访问其他编译单元（如另一代码文件）中的变量或对象，对普通类型（包括基本数据类、结构和类），可以利用关键字 extern，来使用这些变量或对象时；但是对模板类型，则必须在定义这些模板类对象和模板函数时，使用标准 C++ 新增加的关键字 export（导出）。

**21. extern**

extern（外部的）声明变量或函数为外部链接，即该变量或函数名在其它文件中可见。被其修饰的变量（外部变量）是静态分配空间的，即程序开始时分配，结束时释放。用其声明的变量或函数应该在别的文件或同一文件的其它地方定义（实现）。在文件内声明一个变量或函数默认为可被外部使用。在 C++ 中，还可用来指定使用另一语言进行链接，这时需要与特定的转换符一起使用。目前仅支持 **C** 转换标记，来支持 C 编译器链接。使用这种情况有两种形式：

extern "C" 声明语句

extern "C" { 声明语句块 }

**22. false**

false（假的），C++ 的基本数据结构 bool 类型的值之一。等同于 int 的 0 值。

**23. float**

float（浮点数），C++ 中的基本数据结构，精度小于 double。

**24. for**

for 是 C++ 中的循环结构之一。

**25. friend**

friend（友元）声明友元关系。友元可以访问与其有 friend 关系的类中的 private/protected 成员，通过友元直接访问类中的 private/protected 成员的主要目的是提高效率。友元包括友元函数和友元类。

**26. goto**

goto（转到），用于无条件跳转到某一标号处开始执行。

**27. if**

if（如果），C++ 中的条件语句之一，可以根据后面的 bool 类型的值选择进入一个分支执行。

**28. inline**

inline（内联）函数的定义将在编译时在调用处展开。inline 函数一般由短小的语句组成，可以提高程序效率。

**29. int**

int（整型，integer），C++ 中的基本数据结构，用于表示整数，精度小于 long。

**30. long**

long（长整型，long integer），C++ 中的基本数据结构，用于表示长整数。

**31. mutable**

mutable（易变的）是 C++ 中一个不常用的关键字。只能用于类的非静态和非常量数据成员。由于一个对象的状态由该对象的非静态数据成员决定，所以随着数据成员的改变，对像的状态也会随之发生变化。如果一个类的成员函数被声明为 const 类型，表示该函数不会改变对象的状态，也就是该函数不会修改类的非静态数据成员。但是有些时候需要在该类函数中对类的数据成员进行赋值，这个时候就需要用到 mutable 关键字。

**32. namespace**

namespace（命名空间）用于在逻辑上组织类，是一种比类大的结构。

**33. new**

new（新建）用于新建一个对象。new 运算符总是返回一个指针。由 new 创建

**34. operator**

operator（操作符）用于操作符重载。这是 C++ 中的一种特殊的函数。

**35. private**

private（私有的），C++ 中的访问控制符。被标明为 private 的字段只能在本类以及友元中访问。

**36. protected**

protected（受保护的），C++ 中的访问控制符。被标明为 protected 的字段只能在本类以及其继承类和友元中访问。

**37. public**

public（公有的），C++ 中的访问控制符。被标明为 public 的字段可以在任何类

**38.register**

register（寄存器）声明的变量称着寄存器变量，在可能的情况下会直接存放在机器的寄存器中；但对 32 位编译器不起作用，当 global optimizations（全局优化）开的时候，它会做出选择是否放在自己的寄存器中；不过其它与 register 关键字有关的其它符号都对32位编译器有效。

**39. reinterpret\_cast**

用法：

reinpreter\_cast<type-id> (expression)

type-id 必须是一个指针、引用、算术类型、函数指针或者成员指针。它可以把一个指针转换成一个整数，也可以把一个整数转换成一个指针（先把一个指针转换成一个整数，在把该整数转换成原类型的指针，还可以得到原先的指针值）。

**40. return**

return（返回）用于在函数中返回值。程序在执行到 return 语句后立即返回，return 后面的语句无法执行到。

**41. short**

short（短整型，short integer），C++ 中的基本数据结构，用于表示整数，精度小于 int。

**42. signed**

signed（有符号），表明该类型是有符号数，和 unsigned 相反。数字类型（整型和浮点型）都可以用 signed 修饰。但默认就是 signed，所以一般不会显式使用。

**43. sizeof**

由于 C++ 每种类型的大小都是由编译器自行决定的，为了增加可移植性，可以用 sizeof 运算符获得该数据类型占用的字节数。

**44. static**

static（静态的）静态变量作用范围在一个文件内，程序开始时分配空间，结束时释放空间，默认初始化为 0，使用时可改变其值。静态变量或静态函数，只有本文件内的代码才可访问它，它的名字（变量名或函数名）在其它文件中不可见。因此也称为"文件作用域"。在 C++ 类的成员变量被声明为 static（称为静态成员变量），意味着它被该类的所有实例所共享，也就是说当某个类的实例修改了该静态成员变量，其修改值为该类的其它所有实例所见；而类的静态成员函数也只能访问静态成员（变量或函数）。类的静态成员变量必须在声明它的文件范围内进行初始化才能使用，private 类型的也不例外。

**45. static\_cast**

用法：

static\_cast < type-id > ( expression )

该运算符把 expression 转换为 type-id 类型，但没有运行时类型检查来保证转换的安全性。它主要有如下几种用法：

* ① 用于类层次结构中基类和子类之间指针或引用的转换。进行上行转换（把子类的指针或引用转换成基类表示）是安全的；进行下行转换（把基类指针或引用转换成子类表示）时，由于没有动态类型检查，所以是不安全的。
* ② 用于基本数据类型之间的转换，如把 int 转换成 char，把 int 转换成 enum。这种转换的安全性也要开发人员来保证。
* ③ 把空指针转换成目标类型的空指针。
* ④ 把任何类型的表达式转换成void类?

**注意** static\_cast 不能转换掉 expression 的 const、volitale、或者 \_\_unaligned 属性。

**46. struct**

struct（结构）类型，类似于 class 关键字，与 C 语言兼容（class 关键字是不与 C 语言兼容的），可以实现面向对象程序设计。

**47. switch**

switch（转换）类似于 if-else-if 语句，是一种多分枝语句。它提供了一种简洁的书写，并且能够生成效率更好的代码。但是，switch 后面的判断只能是int（char也可以，但char本质上也是一种int类型）。switch 语句最后的 default 分支是可选的。

**48. template**

template（模板），C++ 中泛型机制的实现。

**49. this**

this 返回调用者本身的指针。

**50. throw**

throw（抛出）用于实现 C++ 的异常处理机制，可以通过 throw 关键字"抛出"一个异常。

**51. true**

true（真的），C++ 的基本数据结构 bool 类型的值之一。等同于 int 的非 0 值。

**52. try**

try（尝试）用于实现 C++ 的异常处理机制。可以在 try 中调用可能抛出异常的函数，然后在 try 后面的 catch 中捕获并进行处理。

**53. typedef**

typedef（类型定义，type define），其格式为：

typedef 类型 定义名;

类型说明定义了一个数据类型的新名字而不是定义一种新的数据类型。定义名表示这个类型的新名字。

**54. typeid**

指出指针或引用指向的对象的实际派生类型。

**55. typename**

typename（类型名字）关键字告诉编译器把一个特殊的名字解释成一个类型。在下列情况下必须对一个 name 使用 typename 关键字：

* 1． 一个唯一的name（可以作为类型理解），它嵌套在另一个类型中的。
* 2． 依赖于一个模板参数，就是说：模板参数在某种程度上包含这个name。当模板参数使编译器在指认一个类型时产生了误解。

**56. union**

union（联合），类似于 enum。不同的是 enum 实质上是 int 类型的，而 union 可以用于所有类型，并且其占用空间是随着实际类型大小变化的。

**57. unsigned**

unsigned（无符号），表明该类型是无符号数，和 signed 相反。

**58. using**

表明使用 namespace。

**59. virtual**

virtual（虚的），C++ 中用来实现多态机制。

**60. void**

void（空的），可以作为函数返回值，表明不返回任何数据；可以作为参数，表明没有参数传入（C++中不是必须的）；可以作为指针使用。

**61. volatile**

volatile（不稳定的）限定一个对象可被外部进程（操作系统、硬件或并发线程等）改变，声明时的语法如下：

int volatile nVint;

这样的声明是不能达到最高效的，因为它们的值随时会改变，系统在需要时会经常读写这个对象的值。因此常用于像中断处理程序之类的异步进程进行内存单元访问。

**62. wchar\_t**

wchar\_t 是宽字符类型，每个 wchar\_t 类型占 2 个字节，16 位宽。汉字的表示就要用到 wchar\_t。

**三字符组**

三字符组就是用于表示另一个字符的三个字符序列，又称为三字符序列。三字符序列总是以两个问号开头。

三字符序列不太常见，但 C++ 标准允许把某些字符指定为三字符序列。以前为了表示键盘上没有的字符，这是必不可少的一种方法。

三字符序列可以出现在任何地方，包括字符串、字符序列、注释和预处理指令。

下面列出了最常用的三字符序列：

|  |  |
| --- | --- |
| **三字符组** | **替换** |
| ??= | # |
| ??/ | \ |
| ??' | ^ |
| ??( | [ |
| ??) | ] |
| ??! | | |
| ??< | { |
| ??> | } |
| ??- | ~ |

如果希望在源程序中有两个连续的问号，且不希望被预处理器替换，这种情况出现在字符常量、字符串字面值或者是程序注释中，可选办法是用字符串的自动连接："...?""?..."或者转义序列："...?\?..."。

从Microsoft Visual C++ 2010版开始，该编译器默认不再自动替换三字符组。如果需要使用三字符组替换（如为了兼容古老的软件代码），需要设置编译器命令行选项/Zc:trigraphs

g++仍默认支持三字符组，但会给出编译警告。

**C++ 中的空格**

只包含空格的行，被称为空白行，可能带有注释，C++ 编译器会完全忽略它。

在 C++ 中，空格用于描述空白符、制表符、换行符和注释。空格分隔语句的各个部分，让编译器能识别语句中的某个元素（比如 int）在哪里结束，下一个元素在哪里开始。因此，在下面的语句中：

int age;

在这里，int 和 age 之间必须至少有一个空格字符（通常是一个空白符），这样编译器才能够区分它们。另一方面，在下面的语句中：

fruit = apples + oranges; // 获取水果的总数

fruit 和 =，或者 = 和 apples 之间的空格字符不是必需的，但是为了增强可读性，您可以根据需要适当增加一些空格。

**C++ 注释**

程序的注释是解释性语句，您可以在 C++ 代码中包含注释，这将提高源代码的可读性。所有的编程语言都允许某种形式的注释。

C++ 支持单行注释和多行注释。注释中的所有字符会被 C++ 编译器忽略。

C++ 注释以 /\* 开始，以 \*/ 终止。例如：

/\* 这是注释 \*/ /\* C++ 注释也可以 \* 跨行 \*/

注释也能以 // 开始，直到行末为止。例如：

**实例**

#include <iostream>

using namespace std;

int main()

{

cout << "Hello World"; // 输出 Hello World

return 0;

}

当上面的代码被编译时，编译器会忽略 **// 输出 Hello World**，最后会产生以下结果：

Hello World

在 /\* 和 \*/ 注释内部，// 字符没有特殊的含义。在 // 注释内，/\* 和 \*/ 字符也没有特殊的含义。因此，您可以在一种注释内嵌套另一种注释。例如：

/\* 用于输出 Hello World 的注释 cout << "Hello World"; // 输出 Hello World \*/

## 注意

块注释符（/\*...\*/）是不可以嵌套使用的。

此外，我们还可以使用 **#if 0 ... #endif** 来实现注释，且可以实现嵌套，格式为：

#if 0

code

#endif

你可以把 **#if 0** 改成 **#if 1** 来执行 **code** 的代码。

这种形式对程序调试也可以帮助，测试时使用 **#if 1** 来执行测试代码，发布后使用 **#if 0** 来屏蔽测试代码。

**#if** 后可以是任意的条件语句。

下面的代码如果 condition 条件为 true 执行 code1 ，否则执行 code2。

#if condition

code1

#else

code2

#endif

**C++ 数据类型**

使用编程语言进行编程时，需要用到各种变量来存储各种信息。变量保留的是它所存储的值的内存位置。这意味着，当您创建一个变量时，就会在内存中保留一些空间。

您可能需要存储各种数据类型（比如字符型、宽字符型、整型、浮点型、双浮点型、布尔型等）的信息，操作系统会根据变量的数据类型，来分配内存和决定在保留内存中存储什么。

**基本的内置类型**

C++ 为程序员提供了种类丰富的内置数据类型和用户自定义的数据类型。下表列出了七种基本的 C++ 数据类型：

|  |  |
| --- | --- |
| **类型** | **关键字** |
| 布尔型 | bool |
| 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 |

其实 wchar\_t 是这样来的：

typedef short int wchar\_t;

所以 wchar\_t 实际上的空间是和 short int 一样。

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

下表显示了各种变量类型在内存中存储值时需要占用的内存，以及该类型的变量所能存储的最大值和最小值。

**注意：**不同系统会有所差异。

|  |  |  |
| --- | --- | --- |
| **类型** | **位** | **范围** |
| char | 1 个字节 | -128 到 127 或者 0 到 255 |
| unsigned char | 1 个字节 | 0 到 255 |
| signed char | 1 个字节 | -128 到 127 |
| int | 4 个字节 | -2147483648 到 2147483647 |
| unsigned int | 4 个字节 | 0 到 4294967295 |
| signed int | 4 个字节 | -2147483648 到 2147483647 |
| short int | 2 个字节 | -32768 到 32767 |
| unsigned short int | 2 个字节 | 0 到 65,535 |
| signed short int | 2 个字节 | -32768 到 32767 |
| long int | 8 个字节 | -9,223,372,036,854,775,808 到 9,223,372,036,854,775,807 |
| signed long int | 8 个字节 | -9,223,372,036,854,775,808 到 9,223,372,036,854,775,807 |
| unsigned long int | 8 个字节 | 0 to 18,446,744,073,709,551,615 |
| float | 4 个字节 | +/- 3.4e +/- 38 (~7 个数字) |
| double | 8 个字节 | +/- 1.7e +/- 308 (~15 个数字) |
| long double | 16 个字节 | +/- 1.7e +/- 308 (~15 个数字) |
| wchar\_t | 2 或 4 个字节 | 1 个宽字符 |

从上表可得知，变量的大小会根据编译器和所使用的电脑而有所不同。

下面实例会输出您电脑上各种数据类型的大小。

**实例**

#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。

当上面的代码被编译和执行时，它会产生以下的结果，结果会根据所使用的计算机而有所不同：

type:         \*\*\*\*\*\*\*\*\*\*\*\*size\*\*\*\*\*\*\*\*\*\*\*\*\*\*

bool:         所占字节数：1    最大值：1        最小值：0

char:         所占字节数：1    最大值：        最小值：?

signed char:     所占字节数：1    最大值：        最小值：?

unsigned char:     所占字节数：1    最大值：?        最小值：

wchar\_t:     所占字节数：4    最大值：2147483647        最小值：-2147483648

short:         所占字节数：2    最大值：32767        最小值：-32768

int:         所占字节数：4    最大值：2147483647    最小值：-2147483648

unsigned:     所占字节数：4    最大值：4294967295    最小值：0

long:         所占字节数：8    最大值：9223372036854775807    最小值：-9223372036854775808

unsigned long:     所占字节数：8    最大值：18446744073709551615    最小值：0

double:     所占字节数：8    最大值：1.79769e+308    最小值：2.22507e-308

long double:     所占字节数：16    最大值：1.18973e+4932    最小值：3.3621e-4932

float:         所占字节数：4    最大值：3.40282e+38    最小值：1.17549e-38

size\_t:     所占字节数：8    最大值：18446744073709551615    最小值：0

string:     所占字节数：24

type:         \*\*\*\*\*\*\*\*\*\*\*\*size\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**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。

**C++ 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 | 通常是一个字符（八位）。这是一个整数类型。 |
| int | 对机器而言，整数的最自然的大小。 |
| float | 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。 |
| double | 双精度浮点值。双精度是1位符号，11位指数，52位小数。 |
| void | 表示类型的缺失。 |
| wchar\_t | 宽字符类型。 |

C++ 也允许定义各种其他类型的变量，比如**枚举、指针、数组、引用、数据结构、类**等等，这将会在后续的章节中进行讲解。

下面我们将讲解如何定义、声明和使用各种类型的变量。

**C++ 中的变量定义**

变量定义就是告诉编译器在何处创建变量的存储，以及如何创建变量的存储。变量定义指定一个数据类型，并包含了该类型的一个或多个变量的列表，如下所示：

type variable\_list;

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：

type variable\_name = value;

下面列举几个实例：

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。

**实例**

尝试下面的实例，其中，变量在头部就已经被声明，但它们是在主函数内被定义和初始化的：

#include <iostream>

using namespace std;

// 变量声明

extern int a, b;

extern int c;

extern float f;

int main()

{

// 变量定义

int a, b;

int c;

float f;

// 实际初始化

a = 10;

b = 20;

c = a + b;

cout << c << endl;

f = 70.0 / 3.0;

cout << f << endl;

return 0;

}

当上面的代码被编译和执行时，它会产生下列结果：

30

23.3333

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;

}

**C++ 中的左值（Lvalues）和右值（Rvalues）**

C++ 中有两种类型的表达式：

* **左值（lvalue）：**指向内存位置的表达式被称为左值（lvalue）表达式。左值可以出现在赋值号的左边或右边。
* **右值（rvalue）：**术语右值（rvalue）指的是存储在内存中某些地址的数值。右值是不能对其进行赋值的表达式，也就是说，右值可以出现在赋值号的右边，但不能出现在赋值号的左边。

变量是左值，因此可以出现在赋值号的左边。数值型的字面值是右值，因此不能被赋值，不能出现在赋值号的左边。下面是一个有效的语句：

int g = 20;

但是下面这个就不是一个有效的语句，会生成编译时错误：

10 = 20;

**C++ 变量作用域**

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。

我们将在后续的章节中学习什么是函数和参数。本章我们先来讲解什么是局部变量和全局变量。

**局部变量**

在函数或一个代码块内部声明的变量，称为局部变量。它们只能被函数内部或者代码块内部的语句使用。下面的实例使用了局部变量：

**实例**

#include <iostream>

using namespace std;

int main()

{

// 局部变量声明

int a, b;

int c;

// 实际初始化

a = 10;

b = 20;

c = a + b;

cout << c;

return 0;

}

**全局变量**

在所有函数外部定义的变量（通常是在程序的头部），称为全局变量。全局变量的值在程序的整个生命周期内都是有效的。

全局变量可以被任何函数访问。也就是说，全局变量一旦声明，在整个程序中都是可用的。下面的实例使用了全局变量和局部变量：

**实例**

#include <iostream>

using namespace std;

// 全局变量声明

int g;

int main()

{

// 局部变量声明

int a, b;

// 实际初始化

a = 10;

b = 20;

g = a + b;

cout << g;

return 0;

}

在程序中，局部变量和全局变量的名称可以相同，但是在函数内，局部变量的值会覆盖全局变量的值。下面是一个实例：

**实例**

#include <iostream>

using namespace std;

// 全局变量声明

int g = 20;

int main()

{

// 局部变量声明

int g = 10;

cout << g;

return 0;

}

当上面的代码被编译和执行时，它会产生下列结果：

10

**初始化局部变量和全局变量**

当局部变量被定义时，系统不会对其初始化，您必须自行对其初始化。定义全局变量时，系统会自动初始化为下列值：

|  |  |
| --- | --- |
| **数据类型** | **初始化默认值** |
| int | 0 |
| char | '\0' |
| float | 0 |
| double | 0 |
| pointer | NULL |

正确地初始化变量是一个良好的编程习惯，否则有时候程序可能会产生意想不到的结果。

**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。

下面列举几个整数常量的实例：

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

以下是各种类型的整数常量的实例：

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。

下面列举几个浮点常量的实例：

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

**布尔常量**

布尔常量共有两个，它们都是标准的 C++ 关键字：

* **true** 值代表真。
* **false** 值代表假。

我们不应把 true 的值看成 1，把 false 的值看成 0。

**字符常量**

字符常量是括在单引号中。如果常量以 L（仅当大写时）开头，则表示它是一个宽字符常量（例如 L'x'），此时它必须存储在 **wchar\_t** 类型的变量中。否则，它就是一个窄字符常量（例如 'x'），此时它可以存储在 **char** 类型的简单变量中。

字符常量可以是一个普通的字符（例如 'x'）、一个转义序列（例如 '\t'），或一个通用的字符（例如 '\u02C0'）。

在 C++ 中，有一些特定的字符，当它们前面有反斜杠时，它们就具有特殊的含义，被用来表示如换行符（\n）或制表符（\t）等。下表列出了一些这样的转义序列码：

|  |  |
| --- | --- |
| **转义序列** | **含义** |
| \\ | \ 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 |

下面的实例显示了一些转义序列字符：

**实例**

#include <iostream>

using namespace std;

int main()

{

cout << "Hello\tWorld\n\n";

return 0;

}

当上面的代码被编译和执行时，它会产生下列结果：

Hello World

**字符串常量**

字符串字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。

"hello, dear"

"hello, \

dear"

"hello, " "d" "ear"

**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。

**#define 预处理器**

下面是使用 #define 预处理器定义常量的形式：

#define identifier value

具体请看下面的实例：

**实例**

#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;

}

当上面的代码被编译和执行时，它会产生下列结果：

50

**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;

注：被const关键字修饰的变量为常量，不可作为左值，即不可修改。

具体请看下面的实例：

**实例**

#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;

}

当上面的代码被编译和执行时，它会产生下列结果：

50

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。

**C++ 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;

为了理解 C++ 解释有符号整数和无符号整数修饰符之间的差别，我们来运行一下下面这个短程序：

**实例**

#include <iostream>

using namespace std;

/\*

\* 这个程序演示了有符号整数和无符号整数之间的差别

\*/

int main()

{

short int i; // 有符号短整数

short unsigned int j; // 无符号短整数

j = 50000;

i = j;

cout << i << " " << j;

return 0;

}

当上面的程序运行时，会输出下列结果：

-15536 50000

上述结果中，无符号短整数 50,000 的位模式被解释为有符号短整数 -15,536。

**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 | **const** 类型的对象在程序执行期间不能被修改改变。 |
| volatile | 修饰符 **volatile** 告诉编译器不需要优化volatile声明的变量，让程序可以直接从内存中读取变量。对于一般的变量编译器会对变量进行优化，将内存中的变量值放在寄存器中以加快读写效率。 |
| restrict | 由 **restrict** 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 restrict。 |

**C++ 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++ 17 开始，auto 关键字不再是 C++ 存储类说明符，且 register 关键字被弃用。

**auto 存储类**

自 C++ 11 以来，**auto** 关键字用于两种情况：声明变量时根据初始化表达式自动推断该变量的类型、声明函数时函数返回值的占位符。

C++98标准中auto关键字用于自动变量的声明，但由于使用极少且多余，在C++11中已删除这一用法。

根据初始化表达式自动推断被声明的变量的类型，如：

auto f = 3.14; //double

auto s("hello"); //const char\*

auto z = new auto(9); // int\*

auto x1 = 5, x2 = 5.0, x3 = 'r';//错误，必须是初始化为同一类型

**register 存储类**

**register** 存储类用于定义存储在寄存器中而不是 RAM 中的局部变量。这意味着变量的最大尺寸等于寄存器的大小（通常是一个词），且不能对它应用一元的 '&' 运算符（因为它没有内存位置）。

{ register int miles; }

寄存器只用于需要快速访问的变量，比如计数器。还应注意的是，定义 'register' 并不意味着变量将被存储在寄存器中，它意味着变量可能存储在寄存器中，这取决于硬件和实现的限制。

**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。

**实例**

#include <iostream>

// 函数声明

void func(void);

static int count = 10; /\* 全局变量 \*/

int main()

{

while (count--)

{

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;

}

当上面的代码被编译和执行时，它会产生下列结果：

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

**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

**实例**

#include <iostream>

int count;

extern void write\_extern();

int main()

{

count = 5;

write\_extern();

}

第二个文件：support.cpp

**实例**

#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。现在 ，编译这两个文件，如下所示：

$ g++ main.cpp support.cpp -o write

这会产生 **write** 可执行程序，尝试执行 **write**，它会产生下列结果：

$ ./write

Count is 5

**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

}

**C++ 运算符**

**位运算符**

位运算符作用于位，并逐位执行操作。&、 | 和 ^ 的真值表如下所示：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **p** | **q** | **p & q** | **p | q** | **p ^ q** |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 1 |

假设如果 A = 60，且 B = 13，现在以二进制格式表示，它们如下所示：

A = 0011 1100

B = 0000 1101

-----------------

A&B = 0000 1100

A|B = 0011 1101

A^B = 0011 0001

~A  = 1100 0011

下表显示了 C++ 支持的位运算符。假设变量 A 的值为 60，变量 B 的值为 13，则：

|  |  |  |
| --- | --- | --- |
| **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 |

**实例**

请看下面的实例，了解 C++ 中可用的位运算符。

复制并黏贴下面的 C++ 程序到 test.cpp 文件中，编译并运行程序。

**实例**

#include <iostream>

using namespace std;

int main()

{

unsigned int a = 60; // 60 = 0011 1100

unsigned int b = 13; // 13 = 0000 1101

int c = 0;

c = a & b; // 12 = 0000 1100

cout << "Line 1 - c 的值是 " << c << endl;

c = a | b; // 61 = 0011 1101

cout << "Line 2 - c 的值是 " << c << endl;

c = a ^ b; // 49 = 0011 0001

cout << "Line 3 - c 的值是 " << c << endl;

c = ~a; // -61 = 1100 0011

cout << "Line 4 - c 的值是 " << c << endl;

c = a << 2; // 240 = 1111 0000

cout << "Line 5 - c 的值是 " << c << endl;

c = a >> 2; // 15 = 0000 1111

cout << "Line 6 - c 的值是 " << c << endl;

return 0;

}

当上面的代码被编译和执行时，它会产生以下结果：

Line 1 - c 的值是 12

Line 2 - c 的值是 61

Line 3 - c 的值是 49

Line 4 - c 的值是 -61

Line 5 - c 的值是 240

Line 6 - c 的值是 15

**赋值运算符**

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 ^= 2 等同于 C = C ^ 2 |
| |= | 按位或且赋值运算符 | C |= 2 等同于 C = C | 2 |

**实例**

请看下面的实例，了解 C++ 中可用的赋值运算符。

复制并黏贴下面的 C++ 程序到 test.cpp 文件中，编译并运行程序。

**实例**

#include <iostream>

using namespace std;

int main()

{

int a = 21;

int c;

c = a;

cout << "Line 1 - = 运算符实例，c 的值 = : " << c << endl;

c += a;

cout << "Line 2 - += 运算符实例，c 的值 = : " << c << endl;

c -= a;

cout << "Line 3 - -= 运算符实例，c 的值 = : " << c << endl;

c \*= a;

cout << "Line 4 - \*= 运算符实例，c 的值 = : " << c << endl;

c /= a;

cout << "Line 5 - /= 运算符实例，c 的值 = : " << c << endl;

c = 200;

c %= a;

cout << "Line 6 - %= 运算符实例，c 的值 = : " << c << endl;

c <<= 2;

cout << "Line 7 - <<= 运算符实例，c 的值 = : " << c << endl;

c >>= 2;

cout << "Line 8 - >>= 运算符实例，c 的值 = : " << c << endl;

c &= 2;

cout << "Line 9 - &= 运算符实例，c 的值 = : " << c << endl;

c ^= 2;

cout << "Line 10 - ^= 运算符实例，c 的值 = : " << c << endl;

c |= 2;

cout << "Line 11 - |= 运算符实例，c 的值 = : " << c << endl;

return 0;

}

当上面的代码被编译和执行时，它会产生以下结果：

Line 1 - = 运算符实例，c 的值 = 21

Line 2 - += 运算符实例，c 的值 = 42

Line 3 - -= 运算符实例，c 的值 = 21

Line 4 - \*= 运算符实例，c 的值 = 441

Line 5 - /= 运算符实例，c 的值 = 21

Line 6 - %= 运算符实例，c 的值 = 11

Line 7 - <<= 运算符实例，c 的值 = 44

Line 8 - >>= 运算符实例，c 的值 = 11

Line 9 - &= 运算符实例，c 的值 = 2

Line 10 - ^= 运算符实例，c 的值 = 0

Line 11 - |= 运算符实例，c 的值 = 2

**杂项运算符**

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 | [sizeof 运算符](https://www.runoob.com/cplusplus/cpp-sizeof-operator.html)返回变量的大小。例如，sizeof(a) 将返回 4，其中 a 是整数。 |
| Condition ? X : Y | [条件运算符](https://www.runoob.com/cplusplus/cpp-conditional-operator.html)。如果 Condition 为真 ? 则值为 X : 否则值为 Y。 |
| , | [逗号运算符](https://www.runoob.com/cplusplus/cpp-comma-operator.html)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） | [成员运算符](https://www.runoob.com/cplusplus/cpp-member-operators.html)用于引用类、结构和共用体的成员。 |
| Cast | [强制转换运算符](https://www.runoob.com/cplusplus/cpp-casting-operators.html)把一种数据类型转换为另一种数据类型。例如，int(2.2000) 将返回 2。 |
| & | [指针运算符 &](https://www.runoob.com/cplusplus/cpp-pointer-operators.html) 返回变量的地址。例如 &a; 将给出变量的实际地址。 |
| \* | [指针运算符 \*](https://www.runoob.com/cplusplus/cpp-pointer-operators.html) 指向一个变量。例如，\*var; 将指向变量 var。 |

**C++ 中的运算符优先级**

运算符的优先级确定表达式中项的组合。这会影响到一个表达式如何计算。某些运算符比其他运算符有更高的优先级，例如，乘除运算符具有比加减运算符更高的优先级。

例如 x = 7 + 3 \* 2，在这里，x 被赋值为 13，而不是 20，因为运算符 \* 具有比 + 更高的优先级，所以首先计算乘法 3\*2，然后再加上 7。

下表将按运算符优先级从高到低列出各个运算符，具有较高优先级的运算符出现在表格的上面，具有较低优先级的运算符出现在表格的下面。在表达式中，较高优先级的运算符会优先被计算。

|  |  |  |
| --- | --- | --- |
| **类别** | **运算符** | **结合性** |
| 后缀 | () [] -> . ++ - - | 从左到右 |
| 一元 | + - ! ~ ++ - - (type)\* & sizeof | 从右到左 |
| 乘除 | \* / % | 从左到右 |
| 加减 | + - | 从左到右 |
| 移位 | << >> | 从左到右 |
| 关系 | < <= > >= | 从左到右 |
| 相等 | == != | 从左到右 |
| 位与 AND | & | 从左到右 |
| 位异或 XOR | ^ | 从左到右 |
| 位或 OR | | | 从左到右 |
| 逻辑与 AND | && | 从左到右 |
| 逻辑或 OR | || | 从左到右 |
| 条件 | ?: | 从右到左 |
| 赋值 | = += -= \*= /= %=>>= <<= &= ^= |= | 从右到左 |
| 逗号 | , | 从左到右 |

**实例**

请看下面的实例，了解 C++ 中运算符的优先级。

复制并黏贴下面的 C++ 程序到 test.cpp 文件中，编译并运行程序。

对比有括号和没有括号时的区别，这将产生不同的结果。因为 ()、 /、 \* 和 + 有不同的优先级，高优先级的操作符将优先计算。

**实例**

#include <iostream>

using namespace std;

int main()

{

int a = 20;

int b = 10;

int c = 15;

int d = 5;

int e;

e = (a + b) \* c / d; // ( 30 \* 15 ) / 5

cout << "(a + b) \* c / d 的值是 " << e << endl;

e = ((a + b) \* c) / d; // (30 \* 15 ) / 5

cout << "((a + b) \* c) / d 的值是 " << e << endl;

e = (a + b) \* (c / d); // (30) \* (15/5)

cout << "(a + b) \* (c / d) 的值是 " << e << endl;

e = a + (b \* c) / d; // 20 + (150/5)

cout << "a + (b \* c) / d 的值是 " << e << endl;

return 0;

}

当上面的代码被编译和执行时，它会产生以下结果：

(a + b) \* c / d 的值是 90

((a + b) \* c) / d 的值是 90

(a + b) \* (c / d) 的值是 90

a + (b \* c) / d 的值是 50

**C++ 函数**

**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);

当您在一个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。

**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** |
| [传值调用](https://www.runoob.com/cplusplus/cpp-function-call-by-value.html) | 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。 |
| [指针调用](https://www.runoob.com/cplusplus/cpp-function-call-by-pointer.html) | 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。 |
| [引用调用](https://www.runoob.com/cplusplus/cpp-function-call-by-reference.html) | 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。

**参数的默认值**

当您定义一个函数，您可以为参数列表中后边的每一个参数指定默认值。当调用函数时，如果实际参数的值留空，则使用这个默认值。

这是通过在函数定义中使用赋值运算符来为参数赋值的。调用函数时，如果未传递参数的值，则会使用默认值，如果指定了值，则会忽略默认值，使用传递的值。请看下面的实例：

**实例**

#include <iostream> using namespace std; int sum(int a, int b=20) { int result; result = a + b; return (result); } int main () { // 局部变量声明 int a = 100; int b = 200; int result; // 调用函数来添加值 result = sum(a, b); cout << "Total value is :" << result << endl; // 再次调用函数 result = sum(a); cout << "Total value is :" << result << 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 value is :300

Total value is :120

**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：

[] // 沒有定义任何变量。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(); }();

**C++ 数字**

通常，当我们需要用到数字时，我们会使用原始的数据类型，如 int、short、long、float 和 double 等等。这些用于数字的数据类型，其可能的值和数值范围，我们已经在 C++ 数据类型一章中讨论过。

**C++ 定义数字**

我们已经在之前章节的各种实例中定义过数字。下面是一个 C++ 中定义各种类型数字的综合实例：

**实例**

#include <iostream>

using namespace std;

int main()

{

// 数字定义

short s;

int i;

long l;

float f;

double d;

// 数字赋值

s = 10;

i = 1000;

l = 1000000;

f = 230.47;

d = 30949.374;

// 数字输出

cout << "short s :" << s << endl;

cout << "int i :" << i << endl;

cout << "long l :" << l << endl;

cout << "float f :" << f << endl;

cout << "double d :" << d << endl;

return 0;

}

当上面的代码被编译和执行时，它会产生下列结果：

short s :10

int i :1000

long l :1000000

float f :230.47

double d :30949.4

**C++ 数学运算**

在 C++ 中，除了可以创建各种函数，还包含了各种有用的函数供您使用。这些函数写在标准 C 和 C++ 库中，叫做**内置**函数。您可以在程序中引用这些函数。

C++ 内置了丰富的数学函数，可对各种数字进行运算。下表列出了 C++ 中一些有用的内置的数学函数。

为了利用这些函数，您需要引用数学头文件 **<cmath>**。

|  |  |
| --- | --- |
| **序号** | **函数 & 描述** |
| 1 | **double cos(double);** 该函数返回弧度角（double 型）的余弦。 |
| 2 | **double sin(double);** 该函数返回弧度角（double 型）的正弦。 |
| 3 | **double tan(double);** 该函数返回弧度角（double 型）的正切。 |
| 4 | **double log(double);** 该函数返回参数的自然对数。 |
| 5 | **double pow(double, double);** 假设第一个参数为 x，第二个参数为 y，则该函数返回 x 的 y 次方。 |
| 6 | **double hypot(double, double);** 该函数返回两个参数的平方总和的平方根，也就是说，参数为一个直角三角形的两个直角边，函数会返回斜边的长度。 |
| 7 | **double sqrt(double);** 该函数返回参数的平方根。 |
| 8 | **int abs(int);** 该函数返回整数的绝对值。 |
| 9 | **double fabs(double);** 该函数返回任意一个浮点数的绝对值。 |
| 10 | **double floor(double);** 该函数返回一个小于或等于传入参数的最大整数。 |

下面是一个关于数学运算的简单实例：

**实例**

#include <iostream>

#include <cmath>

using namespace std;

int main()

{

// 数字定义

short s = 10;

int i = -1000;

long l = 100000;

float f = 230.47;

double d = 200.374;

// 数学运算

cout << "sin(d) :" << sin(d) << endl;

cout << "abs(i) :" << abs(i) << endl;

cout << "floor(d) :" << floor(d) << endl;

cout << "sqrt(f) :" << sqrt(f) << endl;

cout << "pow( d, 2) :" << pow(d, 2) << endl;

return 0;

}

当上面的代码被编译和执行时，它会产生下列结果：

sin(d) :-0.634939

abs(i) :1000

floor(d) :200

sqrt(f) :15.1812

pow( d, 2 ) :40149.7

**C++ 随机数**

在许多情况下，需要生成随机数。关于随机数生成器，有两个相关的函数。一个是 **rand()**，该函数只返回一个伪随机数。生成随机数之前必须先调用 **srand()** 函数。

下面是一个关于生成随机数的简单实例。实例中使用了 **time()** 函数来获取系统时间的秒数，通过调用 rand() 函数来生成随机数：

**实例**

#include <iostream>

#include <ctime>

#include <cstdlib>

using namespace std;

int main()

{

int i, j;

// 设置种子

srand((unsigned)time(NULL));

/\* 生成 10 个随机数 \*/

for (i = 0; i < 10; i++)

{

// 生成实际的随机数

j = rand();

cout << "随机数： " << j << endl;

}

return 0;

}

当上面的代码被编译和执行时，它会产生下列结果：

随机数： 1748144778

随机数： 630873888

随机数： 2134540646

随机数： 219404170

随机数： 902129458

随机数： 920445370

随机数： 1319072661

随机数： 257938873

随机数： 1256201101

随机数： 580322989

**C++ 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。

**声明数组**

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];

现在 *balance* 是一个可用的数组，可以容纳 10 个类型为 double 的数字。

**初始化数组**

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};

大括号 { } 之间的值的数目不能大于我们在数组声明时在方括号 [ ] 中指定的元素数目。

如果您省略掉了数组的大小，数组的大小则为初始化时元素的个数。因此，如果：

double balance[] = {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;

上述的语句把数组中第五个元素的值赋为 50.0。所有的数组都是以 0 作为它们第一个元素的索引，也被称为基索引，数组的最后一个索引是数组的总大小减去 1。以下是上面所讨论的数组的的图形表示：



**访问数组元素**

数组元素可以通过数组名称加索引进行访问。元素的索引是放在方括号内，跟在数组名称的后边。例如：

double salary = balance[9];

上面的语句将把数组中第 10 个元素的值赋给 salary 变量。下面的实例使用了上述的三个概念，即，声明数组、数组赋值、访问数组：

**实例**

#include <iostream>

using namespace std;

#include <iomanip>

using std::setw;

int main()

{

int n[10]; // n 是一个包含 10 个整数的数组

// 初始化数组元素

for (int i = 0; i < 10; i++)

{

n[i] = i + 100; // 设置元素 i 为 i + 100

}

cout << "Element" << setw(13) << "Value" << endl;

// 输出数组中每个元素的值

for (int j = 0; j < 10; j++)

{

cout << setw(7) << j << setw(13) << n[j] << endl;

}

return 0;

}

上面的程序使用了 **setw()** 函数来格式化输出。当上面的代码被编译和执行时，它会产生下列结果：

Element Value

0 100

1 101

2 102

3 103

4 104

5 105

6 106

7 107

8 108

9 109

**C++ 多维数组**

[[C++ 数组](https://www.runoob.com/cplusplus/cpp-arrays.html) C++ 数组](https://www.runoob.com/cplusplus/cpp-arrays.html)

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];

**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} , /\* 初始化索引号为 0 的行 \*/

{4, 5, 6, 7} , /\* 初始化索引号为 1 的行 \*/

{8, 9, 10, 11} /\* 初始化索引号为 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;

}

当上面的代码被编译和执行时，它会产生下列结果：

a[0][0]: 0

a[0][1]: 0

a[1][0]: 1

a[1][1]: 2

a[2][0]: 2

a[2][1]: 4

a[3][0]: 3

a[3][1]: 6

a[4][0]: 4

a[4][1]: 8

如上所述，您可以创建任意维度的数组，但是一般情况下，我们创建的数组是一维数组和二维数组。

### vector构建二维数组

如果想从键盘中获取值,来构建二维数组,可以使用如下写法构建:

#include <cstdio>

#include <vector>

#include <algorithm>

using namespace std;

int main()

{

    vector< vector<int> > arry; //写成arry(5) 可理解为设定大小5行

    vector<int> d;        //定义一个一维的数组;

    int i, j, k, n;

    int number;

    scanf("%d", &n );

    /\*可以这样实现对vector二维的初始化，得到的是n行n列的矩阵\*/

    for( i=0; i<n; i++ ){

        for( j=0; j<n; j++ ){

            scanf("%d", &number );

            d.push\_back( number );

        }

        sort( d.begin(), d.end() ); //pai xu xuyao头文件algorithm

        arry.push\_back( d );

        //d.clear();        //清空一维的数组

        d.resize(0);

    }

    /\*遍历输出\*/

    if( arry.empty() )

        printf("0\n");

    else{

        for( i=0; i<arry.size(); i++ ) {

            for( j=0; j<arry[0].size(); j++ ){

                printf("%d ", arry[i][j] );

            }

            printf("\n");

        }

    }

    return 0;

}

**C++ 指向数组的指针**

[[C++ 数组](https://www.runoob.com/cplusplus/cpp-arrays.html) C++ 数组](https://www.runoob.com/cplusplus/cpp-arrays.html)

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。下面的实例演示了上面讨论到的这些概念：

**实例**

#include <iostream>

using namespace std;

int main()

{

// 带有 5 个元素的双精度浮点型数组

double balance[5] = { 1000.0, 2.0, 3.4, 17.0, 50.0 };

double \*p;

p = balance;

// 输出数组中每个元素的值

cout << "使用指针的数组值 " << endl;

for (int i = 0; i < 5; i++)

{

cout << "\*(p + " << i << ") : ";

cout << \*(p + i) << endl;

}

cout << "使用 balance 作为地址的数组值 " << endl;

for (int i = 0; i < 5; i++)

{

cout << "\*(balance + " << i << ") : ";

cout << \*(balance + 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：

使用指针的数组值

\*(p + 0) : 1000

\*(p + 1) : 2

\*(p + 2) : 3.4

\*(p + 3) : 17

\*(p + 4) : 50

使用 balance 作为地址的数组值

\*(balance + 0) : 1000

\*(balance + 1) : 2

\*(balance + 2) : 3.4

\*(balance + 3) : 17

\*(balance + 4) : 50

在上面的实例中，p 是一个指向 double 型的指针，这意味着它可以存储一个 double 类型的变量。一旦我们有了 p 中的地址，**\*p** 将给出存储在 p 中相应地址的值，正如上面实例中所演示的。

## 使用（int\*）获得字符串地址

C++ 中，将 **char \*** 或 **char[]** 传递给 **cout** 进行输出，结果会是整个字符串，如果想要获得字符串的地址（第一个字符的内存地址），可使用以下方法：

强制转化为其他指针（非 **char\***）。可以是 **void \*，int \*，float \*， double \*** 等。**\*** 使用 **&s[0]** 不能输出 **s[0]**（首字符）的地址。因为 **&s[0]** 将返回 **char\***，对于 **char\***（**char** 指针），**cout** 会将其作为字符串来处理，向下查找字符并输出直到字符结束 **\***。

#include <iostream>

using namespace std;

const int MAX = 3;

int main ()

{

char var[MAX] = {'a', 'b', 'c'};

char \*ptr;

// 指针中的数组地址

ptr = var;

for (int i = 0; i < MAX; i++)

{

cout << "Address of var[" << i << "] = ";

cout << (int \*)ptr << endl;

cout << "Value of var[" << i << "] = ";

cout << \*ptr << endl;

// 移动到下一个位置

ptr++;

}

return 0;

}

输出结果：

Address of var[0] = 0x7fffd63b79f9

Value of var[0] = a

Address of var[1] = 0x7fffd63b79fa

Value of var[1] = b

Address of var[2] = 0x7fffd63b79fb

Value of var[2] = c

**C++ 传递数组给函数**

[[C++ 数组](https://www.runoob.com/cplusplus/cpp-arrays.html) C++ 数组](https://www.runoob.com/cplusplus/cpp-arrays.html)

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：

**实例**

#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。

**C++ 从函数返回数组**

[[C++ 数组](https://www.runoob.com/cplusplus/cpp-arrays.html) C++ 数组](https://www.runoob.com/cplusplus/cpp-arrays.html)

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

## 数组就是指针

C++中函数是不能直接返回一个数组的，但是数组其实就是指针，所以可以让函数返回指针来实现:

#include <iostream>

using namespace std;

void MultMatrix(float M[4], float A[4], float B[4])

{

M[0] = A[0]\*B[0] + A[1]\*B[2];

M[1] = A[0]\*B[1] + A[1]\*B[3];

M[2] = A[2]\*B[0] + A[3]\*B[2];

M[3] = A[2]\*B[1] + A[3]\*B[3];

cout << M[0] << " " << M[1] << endl;

cout << M[2] << " " << M[3] << endl;

}

int main()

{

float A[4] = { 1.75, 0.66, 0, 1.75 };

float B[4] = {1, 1, 0, 0};

float \*M = new float[4];

MultMatrix(M, A, B);

cout << M[0] << " " << M[1] << endl;

cout << M[2] << " " << M[3] << endl;

delete[] M;

return 0;

}

## 指向函数的指针

『返回指针的函数』和『指向函数的指针』非常相似，使用时特别注意区分。

返回指针的函数定义：**char \* upper(char \*str)**

指向函数的指针：**char (\*fun)(int int)**

返回指针的函数：

char \* upper(char \*str)

{

char \*dest = str;

while(\*str != NULL)

{

if(\*str >= 'a' && \*str <= 'z')

{

\*str -= 'a' - 'A';

}

str++;

}

return dest;

}

指向函数的指针：

int add(int a,int b)

{

return a+b;

}

int min()

{

int (\*fun)() = add;

int result = (\*fun)(1,2);

char hello[] = "Hello";

char \*dest = upper(heLLo);

}

**int result = (\*fun)(1,2);** 是指向 add 函数的函数，执行结果 **result=3**。

**char \*dest = upper(heLLo);** 将 hello 字符串中的小写字母全部转换为大写字母。

**C++ 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";

以下是 C/C++ 中定义的字符串的内存表示：



其实，您不需要把 *null* 字符放在字符串常量的末尾。C++ 编译器会在初始化数组时，自动把 '\0' 放在字符串的末尾。让我们尝试输出上面的字符串：

**实例**

#include <iostream>

using namespace std;

int main()

{

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

cout << "Greeting message: ";

cout << greeting << endl;

return 0;

}

当上面的代码被编译和执行时，它会产生下列结果：

Greeting message: 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 | **strcpy(s1, s2);** 复制字符串 s2 到字符串 s1。 |
| 2 | **strcat(s1, s2);** 连接字符串 s2 到字符串 s1 的末尾。 |
| 3 | **strlen(s1);** 返回字符串 s1 的长度。 |
| 4 | **strcmp(s1, s2);** 如果 s1 和 s2 是相同的，则返回 0；如果 s1<s2 则返回值小于 0；如果 s1>s2 则返回值大于 0。 |
| 5 | **strchr(s1, ch);** 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 | **strstr(s1, s2);** 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。 |

下面的实例使用了上述的一些函数：

**实例**

#include <iostream>

#include <cstring>

using namespace std;

int main()

{

char str1[11] = "Hello";

char str2[11] = "World";

char str3[11];

int len;

// 复制 str1 到 str3

strcpy(str3, str1);

cout << "strcpy( str3, str1) : " << str3 << endl;

// 连接 str1 和 str2

strcat(str1, str2);

cout << "strcat( str1, str2): " << str1 << endl;

// 连接后，str1 的总长度

len = strlen(str1);

cout << "strlen(str1) : " << 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：

strcpy( str3, str1) : Hello

strcat( str1, str2): HelloWorld

strlen(str1) : 10

**C++ 中的 String 类**

C++ 标准库提供了 **string** 类类型，支持上述所有的操作，另外还增加了其他更多的功能。我们将学习 C++ 标准库中的这个类，现在让我们先来看看下面这个实例：

现在您可能还无法透彻地理解这个实例，因为到目前为止我们还没有讨论类和对象。所以现在您可以只是粗略地看下这个实例，等理解了面向对象的概念之后再回头来理解这个实例。

**实例**

#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

**C++ 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：

**实例**

#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

通过上面的实例，我们了解了什么是内存地址以及如何访问它。接下来让我们看看什么是指针。

**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 \*/

所有指针的值的实际数据类型，不管是整型、浮点型、字符型，还是其他的数据类型，都是一样的，都是一个代表内存地址的长的十六进制数。不同数据类型的指针之间唯一的不同是，指针所指向的变量或常量的数据类型不同。

**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：

**实例**

#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** |
| [C++ Null 指针](https://www.runoob.com/cplusplus/cpp-null-pointers.html) | 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。 |
| [C++ 指针的算术运算](https://www.runoob.com/cplusplus/cpp-pointer-arithmetic.html) | 可以对指针进行四种算术运算：++、--、+、- |
| [C++ 指针 vs 数组](https://www.runoob.com/cplusplus/cpp-pointers-vs-arrays.html) | 指针和数组之间有着密切的关系。 |
| [C++ 指针数组](https://www.runoob.com/cplusplus/cpp-array-of-pointers.html) | 可以定义用来存储指针的数组。 |
| [C++ 指向指针的指针](https://www.runoob.com/cplusplus/cpp-pointer-to-pointer.html) | C++ 允许指向指针的指针。 |
| [C++ 传递指针给函数](https://www.runoob.com/cplusplus/cpp-passing-pointers-to-functions.html) | 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。 |
| [C++ 从函数返回指针](https://www.runoob.com/cplusplus/cpp-return-pointer-from-functions.html) | 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**

[[C++ 指针](https://www.runoob.com/cplusplus/cpp-pointers.html) C++ 指针](https://www.runoob.com/cplusplus/cpp-pointers.html)

在变量声明的时候，如果没有确切的地址可以赋值，为指针变量赋一个 NULL 值是一个良好的编程习惯。赋为 NULL 值的指针被称为**空**指针。

NULL 指针是一个定义在标准库中的值为零的常量。请看下面的程序：

#include <iostream>

using namespace std;

int main ()

{

int \*ptr = NULL;

cout << "ptr 的值是 " << ptr ;

return 0;

}

当上面的代码被编译和执行时，它会产生下列结果：

ptr 的值是 0

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。

## NULL就是值为0的常量

之前一直不知道 NULL 到底是怎么一回事，但是其实我们用 vs2013 可以看到，NULL 的定义就是：

define NULL 0

我们可以用这个程序直观地看到，NULL 的本质。

#include<bits/stdc++.h>

using namespace std;

int main()

{

    cout<<"NULL="<<NULL<<endl;

    return 0;

}

程序运行结果为：

NULL=0

**C++ 指针的算术运算**

[[C++ 指针](https://www.runoob.com/cplusplus/cpp-pointers.html) C++ 指针](https://www.runoob.com/cplusplus/cpp-pointers.html)

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。

**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：

**实例**

#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

**递减一个指针**

同样地，对指针进行递减运算，即把值减去其数据类型的字节数，如下所示：

**实例**

#include <iostream>

using namespace std;

const int MAX = 3;

int main()

{

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

int \*ptr;

// 指针中最后一个元素的地址

ptr = &var[MAX - 1];

for (int i = MAX; i > 0; 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[3] = 0xbfdb70f8

Value of var[3] = 200

Address of var[2] = 0xbfdb70f4

Value of var[2] = 100

Address of var[1] = 0xbfdb70f0

Value of var[1] = 10

**指针的比较**

指针可以用关系运算符进行比较，如 ==、< 和 >。如果 p1 和 p2 指向两个相关的变量，比如同一个数组中的不同元素，则可对 p1 和 p2 进行大小比较。

下面的程序修改了上面的实例，只要变量指针所指向的地址小于或等于数组的最后一个元素的地址 &var[MAX - 1]，则把变量指针进行递增：

**实例**

#include <iostream>

using namespace std;

const int MAX = 3;

int main()

{

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

int \*ptr;

// 指针中第一个元素的地址

ptr = var;

int i = 0;

while (ptr <= &var[MAX - 1])

{

cout << "Address of var[" << i << "] = ";

cout << ptr << endl;

cout << "Value of var[" << i << "] = ";

cout << \*ptr << endl;

// 指向上一个位置

ptr++;

i++;

}

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] = 0xbfce42d0

Value of var[0] = 10

Address of var[1] = 0xbfce42d4

Value of var[1] = 100

Address of var[2] = 0xbfce42d8

Value of var[2] = 200

**C++ 指针 vs 数组**

[[C++ 指针](https://www.runoob.com/cplusplus/cpp-pointers.html) C++ 指针](https://www.runoob.com/cplusplus/cpp-pointers.html)

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：

**实例**

#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：

**实例**

#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。

注：这里虽然不能修改var的值（var不能为左值，因为数组名是一个指针常量），但是想要通过指针遍历数组，可以将var的值赋值给一个指针变量，然后对指针变量进行自增操作即可。

int \*ptr;

// 指针中的数组地址

ptr = var;

**C++ 指针数组**

[[C++ 指针](https://www.runoob.com/cplusplus/cpp-pointers.html) C++ 指针](https://www.runoob.com/cplusplus/cpp-pointers.html)

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：

**实例**

#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：

**实例**

#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

您也可以用一个指向字符的指针数组来存储一个字符串列表，如下：

**实例**

#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

**C++ 指向指针的指针（多级间接寻址）**

[[C++ 指针](https://www.runoob.com/cplusplus/cpp-pointers.html) C++ 指针](https://www.runoob.com/cplusplus/cpp-pointers.html)

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：

**实例**

#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 << "var 值为 :" << var << endl;

cout << "\*ptr 值为:" << \*ptr << endl;

cout << "\*\*pptr 值为:" << \*\*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：

var 值为 :3000

\*ptr 值为:3000

\*\*pptr 值为:3000

## 多级间接寻址的寻址过程

通过查看 var，ptr，\*pptr 的地址可以更加清晰的理解多级间接寻址的寻址过程，从结果可以看出这 3 个的地址都是一样的。

#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 << "var 值为 :" << var << endl;

cout << "\*ptr 值为:" << \*ptr << endl;

cout << "\*\*pptr 值为:" << \*\*pptr << endl;

cout << "var 地址为 :" << &var << endl;

cout << "ptr=&var 值为var的地址:" << ptr << endl;

cout << "ptr地址为:" << &ptr << endl;

cout << "\*pptr=ptr=&var 值为var的地址:" << \*pptr << endl;

cout << "pptr 地址为:" << &pptr << endl;

return 0;

}

运算结果为：

var 值为 :3000

\*ptr 值为:3000

\*\*pptr 值为:3000

var 地址为 :0x7ffeec7a65e8

ptr=&var 值为var的地址:0x7ffeec7a65e8

ptr地址为:0x7ffeec7a65e0

\*pptr=ptr=&var 值为var的地址:0x7ffeec7a65e8

pptr 地址为:0x7ffeec7a65d8

**C++ 传递指针给函数**

[[C++ 指针](https://www.runoob.com/cplusplus/cpp-pointers.html) C++ 指针](https://www.runoob.com/cplusplus/cpp-pointers.html)

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：

**实例**

#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：

**实例**

#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

**C++ 从函数返回指针**

[[C++ 指针](https://www.runoob.com/cplusplus/cpp-pointers.html) C++ 指针](https://www.runoob.com/cplusplus/cpp-pointers.html)

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

**C++ 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。

**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：

**实例**

#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** |
| [把引用作为参数](https://www.runoob.com/cplusplus/passing-parameters-by-references.html) | 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。 |
| [把引用作为返回值](https://www.runoob.com/cplusplus/returning-values-by-reference.html) | 可以从 C++ 函数中返回引用，就像返回其他数据类型一样。 |

**C++ 把引用作为参数**

[[C++ 引用](https://www.runoob.com/cplusplus/cpp-references.html) C++ 引用](https://www.runoob.com/cplusplus/cpp-references.html)

我们已经讨论了如何使用指针来实现引用调用函数。下面的实例使用了引用来实现引用调用函数。

**实例**

#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

## 引用作为函数参数

C++之所以增加引用类型， 主要是把它作为函数参数，以扩充函数传递数据的功能。

**C++ 函数传参：**

(1)将变量名作为实参和形参。这时传给形参的是变量的值，传递是单向的。如果在执行函数期间形参的值发生变化，并不传回给实参。因为在调用函数时，形参和实参不是同一个存储单元。// 同 c

(2) 传递变量的指针。形参是指针变量，实参是一个变量的地址，调用函数时，形参(指针变量)指向实参变量单元。这种通过形参指针可以改变实参的值。// 同 c

(3) C++提供了 传递变量的引用。形参是引用变量，和实参是一个变量，调用函数时，形参(引用变量)指向实参变量单元。这种通过形参引用可以直接改变实参的值。

## 引用作为形参的函数，实参不能是常量

以引用作为参数的函数，可以把变量传入，但不能传入常量。

#include <iostream>

using namespace std;

int hls(int& a1, int& a2, int& b1, int& b2) //定义行列式函数

{

int temp;

temp=a1\*b2-a2\*b1;

return temp;

}

int main()

{

int x1=11; int x2=9;

int y1=15; int y2=14; //定义矩阵

int result; //行列式运算结果

result=hls(x1,x2,y1,y2); //result=hls(11,9,15,14)会报错

cout << result << endl;

return 0;

}

**C++ 把引用作为返回值**

[[C++ 引用](https://www.runoob.com/cplusplus/cpp-references.html) C++ 引用](https://www.runoob.com/cplusplus/cpp-references.html)

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：

**实例**

#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;

}

当上面的代码被编译和执行时，它会产生下列结果：

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

}

## 注意事项

（1）以引用返回函数值，定义函数时需要在函数名前加 **&**

（2）用引用返回一个函数值的最大好处是，在内存中不产生被返回值的副本。

**引用作为返回值，必须遵守以下规则：**

* （1）不能返回局部变量的引用。主要原因是局部变量会在函数返回后被销毁，因此被返回的引用就成为了"无所指"的引用，程序会进入未知状态。
* （2）不能返回函数内部new分配的内存的引用。虽然不存在局部变量的被动销毁问题，可对于这种情况（返回函数内部new分配内存的引用），又面临其它尴尬局面。例如，被函数返回的引用只是作为一 个临时变量出现，而没有被赋予一个实际的变量，那么这个引用所指向的空间（由new分配）就无法释放，造成memory leak。
* （3）可以返回类成员的引用，但最好是const。主要原因是当对象的属性是与某种业务规则（business rule）相关联的时候，其赋值常常与某些其它属性或者对象的状态有关，因此有必要将赋值操作封装在一个业务规则当中。如果其它对象可以获得该属性的非常 量引用（或指针），那么对该属性的单纯赋值就会破坏业务规则的完整性。

## setValue()的解释

* 这一段不是很好理解，贴一个其他人写的作为补充。
* **When a variable is returned by reference, a reference to the variable is passed back to the caller. The caller can then use this reference to continue modifying the variable, which can be useful at times. Return by reference is also fast, which can be useful when returning structs and classes.**
* 我的理解：一个变量被返回其引用的意思是，给了返回变量的一个别名，如教程的例子显示函数给 val[i] 了一个别名。当调用这个函数 setValues(3) 时，返回了这个 val[3] 的别名， 那么我们就可以修改 val[3]，所以可以放到等号的左边。

**C++ 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 <ctime> 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 | [**time\_t time(time\_t \*time);**](https://www.runoob.com/cplusplus/c-function-time.html) 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 | [**char \*ctime(const time\_t \*time);**](https://www.runoob.com/cplusplus/c-function-ctime.html) 该返回一个表示当地时间的字符串指针，字符串形式 *day month year hours:minutes:seconds year\n\0*。 |
| 3 | [**struct tm \*localtime(const time\_t \*time);**](https://www.runoob.com/cplusplus/c-function-localtime.html) 该函数返回一个指向表示本地时间的 **tm** 结构的指针。 |
| 4 | [**clock\_t clock(void);**](https://www.runoob.com/cplusplus/c-function-clock.html) 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 | [**char \* asctime ( const struct tm \* time );**](https://www.runoob.com/cplusplus/c-function-asctime.html) 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：day month date hours:minutes:seconds year\n\0。 |
| 6 | [**struct tm \*gmtime(const time\_t \*time);**](https://www.runoob.com/cplusplus/c-function-gmtime.html) 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 | [**time\_t mktime(struct tm \*time);**](https://www.runoob.com/cplusplus/c-function-mktime.html) 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 | [**double difftime ( time\_t time2, time\_t time1 );**](https://www.runoob.com/cplusplus/c-function-difftime.html) 该函数返回 time1 和 time2 之间相差的秒数。 |
| 9 | [**size\_t strftime();**](https://www.runoob.com/cplusplus/c-function-strftime.html) 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）。

**实例**

~~#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;~~

~~}~~

注：以上为不安全的写法。

例 1 本地时间和格林威治时间安全的写法:

#include <iostream>

#include <ctime>

using namespace std;

int main()

{

    struct tm tmnow;

    char dt[100];

    // 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

    ctime\_s(dt,100,&now);

    cout << "本地日期和时间：" << dt << endl;

    // 把 now 转换为 tm 结构

    gmtime\_s(&tmnow,&now);

    asctime\_s(dt ,&tmnow);

    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。

**实例**

~~#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;~~

~~}~~

注：以上为不安全的写法。

例 2 安全的写法:

#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\_s(<m ,&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

## 以 **20\*\*-\*\*-\*\* \*\*:\*\*:\*\*** 格式输出当前日期

#include <iostream>

#include <ctime>

using namespace std;

string getCurrentDate();

int main()

{

//日期格式化输出

cout << getCurrentDate().c\_str() << endl;

return 0;

}

string getCurrentDate() {

time\_t nowtime;

tm ltm;

nowtime = time(NULL);

char tmp[64];

localtime\_s(&ltm, &nowtime);

strftime(tmp, sizeof(tmp), "%Y-%m-%d %H:%M:%S", &ltm);

return tmp;

}

输出格式类似：

2018-09-19 09:00:58

**C++ 基本的输入输出**

C++ 标准库提供了一组丰富的输入/输出功能，我们将在后续的章节进行介绍。本章将讨论 C++ 编程中最基本和最常见的 I/O 操作。

C++ 的 I/O 发生在流中，流是字节序列。如果字节流是从设备（如键盘、磁盘驱动器、网络连接等）流向内存，这叫做**输入操作**。如果字节流是从内存流向设备（如显示屏、打印机、磁盘驱动器、网络连接等），这叫做**输出操作**。

**I/O 库头文件**

下列的头文件在 C++ 编程中很重要。

|  |  |
| --- | --- |
| **头文件** | **函数和描述** |
| <iostream> | 该文件定义了 **cin、cout、cerr** 和 **clog** 对象，分别对应于标准输入流、标准输出流、非缓冲标准错误流和缓冲标准错误流。 |
| <iomanip> | 该文件通过所谓的参数化的流操纵器（比如 **setw** 和 **setprecision**），来声明对执行标准化 I/O 有用的服务。 |
| <fstream> | 该文件为用户控制的文件处理声明服务。我们将在文件和流的相关章节讨论它的细节。 |

**标准输出流（cout）**

预定义的对象 **cout** 是 **iostream** 类的一个实例。cout 对象"连接"到标准输出设备，通常是显示屏。**cout** 是与流插入运算符 << 结合使用的，如下所示：

**实例**

#include <iostream>

using namespace std;

int main()

{

char str[] = "Hello C++";

cout << "Value of str is : " << str << endl;

}

当上面的代码被编译和执行时，它会产生下列结果：

Value of str is : Hello C++

C++ 编译器根据要输出变量的数据类型，选择合适的流插入运算符来显示值。<< 运算符被重载来输出内置类型（整型、浮点型、double 型、字符串和指针）的数据项。

流插入运算符 << 在一个语句中可以多次使用，如上面实例中所示，**endl** 用于在行末添加一个换行符。

**标准输入流（cin）**

预定义的对象 **cin** 是 **iostream** 类的一个实例。cin 对象附属到标准输入设备，通常是键盘。**cin** 是与流提取运算符 >> 结合使用的，如下所示：

**实例**

#include <iostream>

using namespace std;

int main()

{

char name[50];

cout << "请输入您的名称： ";

cin >> name;

cout << "您的名称是： " << name << endl;

}

当上面的代码被编译和执行时，它会提示用户输入名称。当用户输入一个值，并按回车键，就会看到下列结果：

请输入您的名称： cplusplus

您的名称是： cplusplus

C++ 编译器根据要输入值的数据类型，选择合适的流提取运算符来提取值，并把它存储在给定的变量中。

流提取运算符 >> 在一个语句中可以多次使用，如果要求输入多个数据，可以使用如下语句：

cin >> name >> age;

这相当于下面两个语句：

cin >> name;

cin >> age;

**标准错误流（cerr）**

预定义的对象 **cerr** 是 **iostream** 类的一个实例。cerr 对象附属到标准错误设备，通常也是显示屏，但是 **cerr** 对象是非缓冲的，且每个流插入到 cerr 都会立即输出。

**cerr** 也是与流插入运算符 << 结合使用的，如下所示：

**实例**

#include <iostream>

using namespace std;

int main()

{

char str[] = "Unable to read....";

cerr << "Error message : " << str << endl;

}

当上面的代码被编译和执行时，它会产生下列结果：

Error message : Unable to read....

**标准日志流（clog）**

预定义的对象 **clog** 是 **iostream** 类的一个实例。clog 对象附属到标准错误设备，通常也是显示屏，但是 **clog** 对象是缓冲的。这意味着每个流插入到 clog 都会先存储在缓冲在，直到缓冲填满或者缓冲区刷新时才会输出。

**clog** 也是与流插入运算符 << 结合使用的，如下所示：

**实例**

#include <iostream>

using namespace std;

int main()

{

char str[] = "Unable to read....";

clog << "Error message : " << str << endl;

}

当上面的代码被编译和执行时，它会产生下列结果：

Error message : Unable to read....

通过这些小实例，我们无法区分 cout、cerr 和 clog 的差异，但在编写和执行大型程序时，它们之间的差异就变得非常明显。所以良好的编程实践告诉我们，使用 cerr 流来显示错误消息，而其他的日志消息则使用 clog 流来输出。

## 输入输出流中的函数

输入输出流中的函数（模板）：

#include <iostream>

#include <iomanip>

using namespace std;

int main()

{

    cout<<setiosflags(ios::left|ios::showpoint); // 设左对齐，以一般实数方式显示

    cout.precision(5); // 设置除小数点外有五位有效数字

    cout<<123.456789<<endl;

    cout.width(10); // 设置显示域宽10

    cout.fill('\*'); // 在显示区域空白处用\*填充

    cout<<resetiosflags(ios::left); // 清除状态左对齐

    cout<<setiosflags(ios::right); // 设置右对齐

    cout<<123.456789<<endl;

    cout<<setiosflags(ios::left|ios::fixed); // 设左对齐，以固定小数位显示

    cout.precision(3); // 设置实数显示三位小数

    cout<<999.123456<<endl;

    cout<<resetiosflags(ios::left|ios::fixed); //清除状态左对齐和定点格式

    cout<<setiosflags(ios::left|ios::scientific); //设置左对齐，以科学技术法显示

    cout.precision(3); //设置保留三位小数

    cout<<123.45678<<endl;

    return 0;

}

测试输出结果：

123.46

\*\*\*\*123.46

999.123

1.235e+02

其中 cout.setf 跟 setiosflags 一样，cout.precision 跟 setprecision 一样，cout.unsetf 跟 resetiosflags 一样。

setiosflags(ios::fixed) 固定的浮点显示

setiosflags(ios::scientific) 指数表示

setiosflags(ios::left) 左对齐

setiosflags(ios::right) 右对齐

setiosflags(ios::skipws 忽略前导空白

setiosflags(ios::uppercase) 16进制数大写输出

setiosflags(ios::lowercase) 16进制小写输出

setiosflags(ios::showpoint) 强制显示小数点

setiosflags(ios::showpos) 强制显示符号

cout.setf 常见的标志：

| **标志** | **功能** |
| --- | --- |
| boolalpha | 可以使用单词”true”和”false”进行输入/输出的布尔值. |
| oct | 用八进制格式显示数值. |
| dec | 用十进制格式显示数值. |
| hex | 用十六进制格式显示数值. |
| left | 输出调整为左对齐. |
| right | 输出调整为右对齐. |
| scientific | 用科学记数法显示浮点数. |
| fixed | 用正常的记数方法显示浮点数(与科学计数法相对应). |
| showbase | 输出时显示所有数值的基数. |
| showpoint | 显示小数点和额外的零，即使不需要. |
| showpos | 在非负数值前面显示”＋（正号）”. |
| skipws | 当从一个流进行读取时，跳过空白字符(spaces, tabs, newlines). |
| unitbuf | 在每次插入以后，清空缓冲区. |
| internal | 将填充字符回到符号和数值之间. |
| uppercase | 以大写的形式显示科学记数法中的”e”和十六进制格式的”x”. |

iostream 中定义的操作符：

| **操作符** | **描述** | **输入** | **输出** |
| --- | --- | --- | --- |
| boolalpha | 启用boolalpha标志 | √ | √ |
| dec | 启用dec标志 | √ | √ |
| endl | 输出换行标示，并清空缓冲区 |  | √ |
| ends | 输出空字符 |  | √ |
| fixed | 启用fixed标志 |  | √ |
| flush | 清空流 |  | √ |
| hex | 启用 hex 标志 | √ | √ |
| internal | 启用 internal 标志 |  | √ |
| left | 启用 left 标志 |  | √ |
| noboolalpha | 关闭boolalpha 标志 | √ | √ |
| noshowbase | 关闭showbase 标志 |  | √ |
| noshowpoint | 关闭showpoint 标志 |  | √ |
| noshowpos | 关闭showpos 标志 |  | √ |
| noskipws | 关闭skipws 标志 | √ |  |
| nounitbuf | 关闭unitbuf 标志 |  | √ |
| nouppercase | 关闭uppercase 标志 |  | √ |
| oct | 启用 oct 标志 | √ | √ |
| right | 启用 right 标志 |  | √ |
| scientific | 启用 scientific 标志 |  | √ |
| showbase | 启用 showbase 标志 |  | √ |
| showpoint | 启用 showpoint 标志 |  | √ |
| showpos | 启用 showpos 标志 |  | √ |
| skipws | 启用 skipws 标志 | √ |  |
| unitbuf | 启用 unitbuf 标志 |  | √ |
| uppercase | 启用 uppercase 标志 |  | √ |
| ws | 跳过所有前导空白字符 | √ |  |

iomanip 中定义的操作符：

| **操作符** | **描述** | **输入** | **输出** |
| --- | --- | --- | --- |
| resetiosflags(long f) | 关闭被指定为f的标志 | √ | √ |
| setbase(int base) | 设置数值的基本数为base |  | √ |
| setfill(int ch) | 设置填充字符为ch |  | √ |
| setiosflags(long f) | 启用指定为f的标志 | √ | √ |
| setprecision(int p) | 设置数值的精度(四舍五入) |  | √ |
| setw(int w) | 设置域宽度为w |  | √ |

**C++ 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：

**实例**

#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;

}

实例中定义了结构体类似 Books 及其两个变量 Book1 和 Book2。当上面的代码被编译和执行时，它会产生下列结果：

第一本书标题 : C++ 教程

第一本书作者 : Runoob

第一本书类目 : 编程语言

第一本书 ID : 12345

第二本书标题 : CSS 教程

第二本书作者 : Runoob

第二本书类目 : 前端技术

第二本书 ID : 12346

**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：

**实例**

#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;

}

当上面的代码被编译和执行时，它会产生下列结果：

书标题 : C++ 教程

书作者 : Runoob

书类目 : 编程语言

书 ID : 12345

书标题 : CSS 教程

书作者 : Runoob

书类目 : 前端技术

书 ID : 12346

**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：

**实例**

#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;

}

当上面的代码被编译和执行时，它会产生下列结果：

书标题 : C++ 教程

书作者 : Runoob

书类目 : 编程语言

书 ID : 12345

书标题 : CSS 教程

书作者 : Runoob

书类目 : 前端技术

书 ID : 12346

**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 Books

{

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。

## 类与结构体的区别

类与结构体在 C++ 中只有两点区别，除此这外无任何区别。

* （1）class 中默认的成员访问权限是 private 的，而 struct 中则是 public 的。
* （2）从 class 继承默认是 private 继承，而从 struct 继承默认是 public 继承。
* （3）class 可以定义模板，而 struct 不可以。

**结构体变量和结构体类型的定义**

*分类*[**编程技术**](https://www.runoob.com/w3cnote_genre/code)

**1.结构体类型定义**

定义方式1：

Typedef struct LNode {

int data; // 数据域

struct LNode \*next; // 指针域

} \*LinkList;

定义方式2：

struct LNode {

int data; // 数据域

struct LNode \*next; // 指针域

};

Typedef struct LNode \*LinkList;

以上两个定义方式是等价的，是将 **\*LinkList** 定 义为 **struct LNode** 类型，即 **LinkList** 被定义为一个类型名。这样就可以用 **LinkList** 来定义说明新的变量了，如：

LinkList L;

即将 **L** 定义为 **struct LNode** 类型的指针变量。

**2.结构体类型变量定义**

定义方式1：

struct LNode {

int data; // 数据域

struct LNode \*next; // 指针域

}LnodeA;

定义方式2：

struct LNode {

int data; // 数据域

struct LNode \*next; // 指针域

};

struct LNode LnodeA;

以上两个定义方式也是等价的，这样就将 **LnodeA** 定义为一个 **struct LNode** 类型的变量，即 **LnodeA** 为一个 **struct LNode** 类型的变量名。

### C/C++ 语言中的解释

**结构体定义**

结构体(struct)是由一系列具有相同类型或不同类型的数据构成的数据集合，也叫结构。

**结构体作用**

结构体和其他类型基础数据类型一样，例如 int 类型，char 类型 只不过结构体可以做成你想要的数据类型。以方便日后的使用。

在实际项目中，结构体是大量存在的。研发人员常使用结构体来封装一些属性来组成新的类型。

结构体在函数中的作用不是简便，其最主要的作用就是封装。封装的好处就是可以再次利用。让使用者不必关心这个是什么，只要根据定义使用就可以了。

**结构体的大小与内存对齐**

结构体的大小不是结构体元素单纯相加就行的，因为我们现在主流的计算机使用的都是 32Bit 字长的 CPU，对这类型的 CPU 取 4 个字节的数要比取一个字节要高效，也更方便。所以在结构体中每个成员的首地址都是4的整数倍的话，取数据元素是就会相对更高效，这就是内存对齐的由来。

每个特定平台上的编译器都有自己的默认"对齐系数"(也叫对齐模数)。程序员可以通过预编译命令 **#pragma pack(n)，n=1,2,4,8,1**6 来改变这一系数，其中的n就是你要指定的"对齐系数"。

**规则：**

* 1、数据成员对齐规则：结构(struct)(或联合(union))的数据成员，第一个数据成员放在 offset 为 0 的地方，以后每个数据成员的对齐按照 **#pragma pack**指定的数值和这个数据成员自身长度中，比较小的那个进行。
* 2、结构(或联合)的整体对齐规则：在数据成员完成各自对齐之后，结构(或联合)本身也要进行对齐，对齐将按照< span class="marked">#pragma pack 指定的数值和结构(或联合)最大数据成员长度中，比较小的那个进行。
* 3、结合1、2颗推断：当#pragma pack的n值等于或超过所有数据成员长度的时候，这个n值的大小将不产生任何效果。

**C++ 中的结构体**

在 C 语言中，可以定义结构体类型，将多个相关的变量包装成为一个整体使用。在结构体中的变量，可以是相同、部分相同，或完全不同的数据类型。在 C 语言中，结构体不能包含函数。在面向对象的程序设计中，对象具有状态（属性）和行为，状态保存在成员变量中，行为通过成员方法（函数）来实现。C 语言中的结构体只能描述一个对象的状态，不能描述一个对象的行为。在 C++ 中，考虑到 C 语言到 C++ 语言过渡的连续性，对结构体进行了扩展，C++ 的结构体可以包含函数，这样，C++ 的结构体也具有类的功能，与class 不同的是，结构体包含的函数默认为 public，而不是 private。

C++ 控制台输出例子：

### 实例

#include <cstdlib>

#include <iostream>　　//定义结构体

using namespace std;

struct point

{

//包含两个变量成员

int x;

int y;

};

int main(int argc, char \*argv[])

{

struct point pt;

pt.x = 1;

pt.y = 2;

cout << pt.x << endl << pt.y << endl;

return EXIT\_SUCCESS;

}

**C++中的结构体与类的区别**

类与结构体在 C++ 中只有两点区别，除此这外无任何区别。

* （1）class 中默认的成员访问权限是 private 的，而 struct 中则是 public 的。
* （2）从 class 继承默认是 private 继承，而从 struct 继承默认是 public 继承。

**这两种定义有什么区别?**

typedef struct student

{

    int num;

    struct student \*next;

}student;

struct student

{

    int num;

    struct student \*next;

};

第二个 struct student 是定义了一个 student 结构体，这个明白吧。

第一个是用 typedef 把 struct student 这个结构体类型名字重新定义为 student，也就是说 struct student 和 student 表示同一个事物，都是一个类型的标识符，比如 typedef int zhengshu; 就是你把整型 int 重命名为 zhengshu，下面定义：int i; 和 zhengshu i; 两句就是等价的了。

结构是由基本数据类型构成的、并用一个标识符来命名的各种变量的组合。结构中可以使用不同的数据类型。

**结构说明和结构变量定义**

在 Turbo C 中，结构也是一种数据类型，可以使用结构变量，因此，象其它类型的变量一样，在使用结构变量时要先对其定义。

定义结构变量的一般格式为:

struct 结构名

{

    类型 变量名;

    类型 变量名;

...

} 结构变量;

结构名是结构的标识符不是变量名。

类型为第二节中所讲述的五种数据类型(整型、浮点型、字符型、指针型和无值型)。

构成结构的每一个类型变量称为结构成员，它象数组的元素一样，但数组中元素是以下标来访问的，而结构是按变量名字来访问成员的。

下面举一个例子来说明怎样定义结构变量。

struct string

{

    char name[8];

    int age;

    char sex[4];

    char depart[20];

    float wage1,wage2,wage3;

}person;

这个例子定义了一个结构名为 string 的结构变量 person，如果省略变量名 person，则变成对结构的说明。用已说明的结构名也可定义结构变量。这样定义时上例变成:

struct string

{

    char name[8];

    int age;

    char sex[4];

    char depart[20];

    float wage1,wage2,wage3;

};

struct string person;

如果需要定义多个具有相同形式的结构变量时用这种方法比较方便，它先作结构说明，再用结构名来定义变量。

例如:

struct string Tianyr, Liuqi, ...;

如果省略结构名，则称之为无名结构，这种情况常常出现在函数内部，用这种结构时前面的例子变成:

struct

{

    char name[8];

    int age;

    char sex[4];

    char depart[20];

    float wage1,wage2,wage3;

} Tianyr, Liuqi;

**结构变量的使用**

结构是一个新的数据类型，因此结构变量也可以象其它类型的变量一样赋值、运算，不同的是结构变量以成员作为基本变量。

结构成员的表示方式为:

结构变量.成员名

如果将结构变量.成员名看成一个整体，则这个整体的数据类型与结构中该成员的数据类型相同，这样就可象前面所讲的变量那样使用。

下面这个例子定义了一个结构变量，其中每个成员都从键盘接收数据，然后对结构中的浮点数求和，并显示运算结果。请注意这个例子中不同结构成员的访问。

### 实例

#include <stdio.h>

#include <conio.h>

int main()

{

struct

{

char name[8];

int age;

char sex[4];

char depart[20];

float wage1, wage2, wage3;

}a;

float wage;

char c = 'Y';

while (c == 'Y' || c == 'y')

{

printf(\nName:);

scanf(%s, a.name);

printf(Age:);

scanf(%d, &a.wage);

printf(Sex:);

scanf(%s, a.sex);

printf(Dept:);

scanf(%s, a.depart);

printf(Wage1:);

scanf(%f, &a.wage1);

printf(Wage2:);

scanf(%f, &a.wage2);

printf(Wage3:);

scanf(%f, &a.wage3);

wage = a.wage1 + a.wage2 + a.wage3;

printf(The sum of wage is %6.2f\n, wage);

printf(Continue ? );

c = getche();

}

}

**结构数组和结构指针**

结构是一种新的数据类型， 同样可以有结构数组和结构指针。

**1.结构数组**

结构数组就是具有相同结构类型的变量集合。假如要定义一个班级40个同学的姓名、性别、年龄和住址，可以定义成一个结构数组。如下所示:

struct

{

    char name[8];

    char sex[4];

    int age;

    char addr[40];

}student[40];

也可定义为:

struct string

{

    char name[8];

    char sex[4];

    int age;

    char addr[40];

};

struct string student[40];

需要指出的是结构数组成员的访问是以数组元素为结构变量的, 其形式为:

结构数组元素.成员名

例如:

student[0].name

student[30].age

实际上结构数组相当于一个二维构造, 第一维是结构数组元素, 每个元素是一个结构变量, 第二维是结构成员。

**注意**: 结构数组的成员也可以是数组变量。

例如:

struct a

{

    int m[3][5];

    float f;

    char s[20];

}y[4];

为了访问结构a中结构变量 **y[2]** 的这个变量, 可写成 **y[2].m[1][4]**

**2.结构指针**

结构指针是指向结构的指针。它由一个加在结构变量名前的\* 操作符来定义, 例如用前面已说明的结构定义一个结构指针如下:

struct string

{

    char name[8];

    char sex[4];

    int age;

    char addr[40];

}\*student;

也可省略结构指针名只作结构说明, 然后再用下面的语句定义结构指针。

struct string \*student;

使用结构指针对结构成员的访问，与结构变量对结构成员的访问在表达方式上有所不同。结构指针对结构成员的访问表示为:

结构指针名->结构成员

其中 **->** 是两个符号 **-** 和 **>** 的组合，好象一个箭头指向结构成员。例如要给上面定义的结构中 name 和 age 赋值，可以用下面语句:

strcpy(student->name, Lu G.C);

student->age=18;

实际上，**student->name** 就是 **(\*student).name** 的缩写形式。

需要指出的是结构指针是指向结构的一个指针，即结构中第一个成员的首地址，因此在使用之前应该对结构指针初始化，即分配整个结构长度的字节空间，这可用下面函数完成， 仍以上例来说明如下:

student=(struct string\*)malloc(size of (struct string));

**size of (struct string)** 自动求取 string 结构的字节长度， malloc() 函数定义了一个大小为结构长度的内存区域，然后将其诈地址作为结构指针返回。

**注意:**

* 1. 结构作为一种数据类型，因此定义的结构变量或结构指针变量同样有局部变量和全程变量，视定义的位置而定。
* 2. 结构变量名不是指向该结构的地址，这与数组名的含义不同，因此若需要求结构中第一个成员的首地址应该是 **&[结构变量名]**。
* 3. 结构的复杂形式：嵌套结构

嵌套结构是指在一个结构成员中可以包括其它一个结构，Turbo C 允许这种嵌套。

例如: 下面是一个有嵌套的结构

struct string

{

    char name[8];

    int age;

    struct addr address;

} student;

其中: addr 为另一个结构的结构名, 必须要先进行, 说明, 即

struct addr

{

    char city[20];

    unsigned lon zipcode;

    char tel[14];

}

如果要给 student 结构中成员 address 结构中的 zipcode 赋值, 则可写成:

student.address.zipcode=200001;

每个结构成员名从最外层直到最内层逐个被列出,即嵌套式结构成员的表达方式是:

结构变量名.嵌套结构变量名.结构成员名

其中嵌套结构可以有很多，结构成员名为最内层结构中不是结构的成员名。

## . 与 -> 运算符

**.**（点）运算符和 ->（箭头）运算符用于引用类、结构和共用体的成员: 点运算符应用于实际的对象。箭头运算符与一个指向对象的指针一起使用。

例如，假设有下面的结构：

struct Employee {

char first\_name[16];

int age;

} emp;

**. 点运算符**

下面的代码把值 zara 赋给对象 emp 的 first\_name 成员：

strcpy(emp.first\_name, "zara");

**-> 箭头运算符**

如果 p\_emp 是一个指针，指向类型为 Employee 的对象，则要把值 zara 赋给对象 emp 的 first\_name 成员，需要编写如下代码：

strcpy(p\_emp->first\_name, "zara");

**->** 称为箭头运算符，它是由一个减号加上一个大于号组成。

简而言之，访问结构的成员时使用点运算符，而通过指针访问结构的成员时，则使用箭头运算符。

也就是说，用结构体定义了一个实体，那么这个实体要引用他里面的成员，就用 **.** 操作符，如果用结构体定义的是一个结构指针，那么要引用他里面的成员就用 **->**。

## 结构作为函数参数或返回值

可以将结构作为函数的返回值。实例如下：

#include <stdio.h>

#include <stdlib.h>

struct test{

    int i;

    char c;

    double d;

    float f;

};

struct test set( int a, float b, char c, double d )

{

    struct test t;

    t.i = a;

    t.f = b;

    t.c = c;

    t.d = d;

    return t;

}

void sig( int s );

void print(struct test t2)

{

    printf("int:%d\n",t2.i);

    printf("char:%c\n",t2.c);

    printf("float:%f\n",t2.f);

    printf("double:%lf\n",t2.d);

}

void sig( int s )

{

    printf("捕获信号%d,跳出……",s);

    exit(1);

}

int main( void )

{

    struct test info;

    info = set(2,3.22,'d',4.335);

    print(info);

    return 0;

}

## C11标准定义结构不需要typedef

根据原先 C98 的标准，结构体定义的时候需要使用 typedef。

但是对于更新的 C99 标准和 C11 及以上的标准，typedef 可以省略或者强制省略。

对于定义 struct 的时候，C98 需要添加 struct，但是同时 C99 标准也去掉了。

举个小例子（可能不全）：

typedef struct edge{

int from;

int to;

int dis;

};

struct Edge[101];

//C98

现在的新标准（通用的）会变成：

struct edge{

int from;

int to;

int dis;

};

edge Edge[101];

//C99

同时 C99 的例子中 edge 类型就可以像 int 类型一样赋值、传输、传地址、定义、作为函数类型...

需要的时候还可以重载运算符，重载运算符可以参考这个站的其他内容了。

同时再讲点新的：

对于结构体类型的变量，我们可以限制成员的位数大小。

举个例子：

struct demo{int demoint:1;};

这样就限制了 demoint 成员只占一个 B。

**C++ 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

对象 Box1 和 Box2 都有它们各自的数据成员。

**访问数据成员**

类的对象的公共数据成员可以使用直接成员访问运算符 (.) 来访问。为了更好地理解这些概念，让我们尝试一下下面的实例：

**实例**

#include <iostream>

using namespace std;

class Box

{

public:

double length; // 长度

double breadth; // 宽度

double height; // 高度

};

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.height = 5.0;

Box1.length = 6.0;

Box1.breadth = 7.0;

// box 2 详述

Box2.height = 10.0;

Box2.length = 12.0;

Box2.breadth = 13.0;

// box 1 的体积

volume = Box1.height \* Box1.length \* Box1.breadth;

cout << "Box1 的体积：" << volume << endl;

// box 2 的体积

volume = Box2.height \* Box2.length \* Box2.breadth;

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

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** |
| [类成员函数](https://www.runoob.com/cplusplus/cpp-class-member-functions.html) | 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。 |
| [类访问修饰符](https://www.runoob.com/cplusplus/cpp-class-access-modifiers.html) | 类成员可以被定义为 public、private 或 protected。默认情况下是定义为 private。 |
| [构造函数 & 析构函数](https://www.runoob.com/cplusplus/cpp-constructor-destructor.html) | 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。 |
| [C++ 拷贝构造函数](https://www.runoob.com/cplusplus/cpp-copy-constructor.html) | 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。 |
| [C++ 友元函数](https://www.runoob.com/cplusplus/cpp-friend-functions.html) | **友元函数**可以访问类的 private 和 protected 成员。 |
| [C++ 内联函数](https://www.runoob.com/cplusplus/cpp-inline-functions.html) | 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。 |
| [C++ 中的 this 指针](https://www.runoob.com/cplusplus/cpp-this-pointer.html) | 每个对象都有一个特殊的指针 **this**，它指向对象本身。 |
| [C++ 中指向类的指针](https://www.runoob.com/cplusplus/cpp-pointer-to-class.html) | 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。 |
| [C++ 类的静态成员](https://www.runoob.com/cplusplus/cpp-static-members.html) | 类的数据成员和函数成员都可以被声明为静态的。 |

## C++中struct与class的唯一区别

C++ 中的 struct 对 C 中的 struct 进行了扩充，它已经不再只是一个包含不同数据类型的数据结构了，它已经获取了太多的功能。

struct 能包含成员函数吗？ 能！

struct 能继承吗？ 能！！

struct 能实现多态吗？ 能！！！

既然这些它都能实现，那它和 class 还能有什么区别？

最本质的一个区别就是默认的访问控制，体现在两个方面：

1）默认的继承访问权限。struct是public的，class是private的。

你可以写如下的代码：

struct A

{

char a;

};

struct B : A

{

char b;

};

这个时候 B 是 public 继承 A 的。

如果都将上面的 struct 改成 class，那么 B 是 private 继承 A 的。这就是默认的继承访问权限。

所以我们在平时写类继承的时候，通常会这样写：

struct B : public A

就是为了指明是 public 继承，而不是用默认的 private 继承。

当然，到底默认是 public 继承还是 private 继承，取决于子类而不是基类。

我的意思是，struct 可以继承 class，同样 class 也可以继承 struct，那么默认的继承访问权限是看子类到底是用的 struct 还是 class。如下：

struct A{};

class B : A{}; //private继承

struct C : B{}; //public继承

2）struct 作为数据结构的实现体，它默认的数据访问控制是 public 的，而 class 作为对象的实现体，它默认的成员变量访问控制是 private 的。

注意我上面的用词，我依旧强调 struct 是一种数据结构的实现体，虽然它是可以像 class 一样的用。我依旧将 struct 里的变量叫数据，class 内的变量叫成员，虽然它们并无区别。

其实，到底是用 struct 还是 class，完全看个人的喜好，你可以将你程序里所有的 class 全部替换成 struct，它依旧可以很正常的运行。但我给出的最好建议，还是：当你觉得你要做的更像是一种数据结构的话，那么用 struct，如果你要做的更像是一种对象的话，那么用 class。

当然，我在这里还要强调一点的就是，对于访问控制，应该在程序里明确的指出，而不是依靠默认，这是一个良好的习惯，也让你的代码更具可读性。

说到这里，很多了解的人或许都认为这个话题可以结束了，因为他们知道 struct 和 class 的“唯一”区别就是访问控制。很多文献上也确实只提到这一个区别。

但我上面却没有用“唯一”，而是说的“最本质”，那是因为，它们确实还有另一个区别，虽然那个区别我们平时可能很少涉及。那就是：“class” 这个关键字还用于定义模板参数，就像 “typename”。但关键字 “struct” 不用于定义模板参数。这一点在 Stanley B.Lippman 写的 Inside the C++ Object Model 有过说明。

问题讨论到这里，基本上应该可以结束了。但有人曾说过，他还发现过其他的“区别”，那么，让我们来看看，这到底是不是又一个区别。还是上面所说的，C++ 中的 struct 是对 C 中的 struct 的扩充，既然是扩充，那么它就要兼容过去 C 中 struct 应有的所有特性。例如你可以这样写：

struct A //定义一个struct

{

char c1;

int n2;

double db3;

};

A a={'p',7,3.1415926}; //定义时直接赋值

也就是说 struct 可以在定义的时候用 {} 赋初值。那么问题来了，class 行不行呢？将上面的 struct 改成 class，试试看。报错！噢~于是那人跳出来说，他又找到了一个区别。我们仔细看看，这真的又是一个区别吗？

你试着向上面的 struct 中加入一个构造函数（或虚函数），你会发现什么？

对，struct 也不能用 {} 赋初值了。

的确，以 {} 的方式来赋初值，只是用一个初始化列表来对数据进行按顺序的初始化，如上面如果写成 A a={'p',7}; 则 c1,n2 被初始化，而 db3 没有。这样简单的 copy 操作，只能发生在简单的数据结构上，而不应该放在对象上。加入一个构造函数或是一个虚函数会使 struct 更体现出一种对象的特性，而使此{}操作不再有效。

事实上，是因为加入这样的函数，使得类的内部结构发生了变化。而加入一个普通的成员函数呢？你会发现{}依旧可用。其实你可以将普通的函数理解成对数据结构的一种算法，这并不打破它数据结构的特性。

那么，看到这里，我们发现即使是 struct 想用 {} 来赋初值，它也必须满足很多的约束条件，这些条件实际上就是让 struct 更体现出一种数据机构而不是类的特性。

那为什么我们在上面仅仅将 struct 改成 class，{} 就不能用了呢？

其实问题恰巧是我们之前所讲的——访问控制！你看看，我们忘记了什么？对，将 struct 改成 class 的时候，访问控制由 public 变为 private 了，那当然就不能用 {} 来赋初值了。加上一个 public，你会发现，class 也是能用 {} 的，和 struct 毫无区别！！！

做个总结，从上面的区别，我们可以看出，struct 更适合看成是一个数据结构的实现体，class 更适合看成是一个对象的实现体。

## 类外使用指针访问类内部的私有成员

在类的外面，其实也可以用指针访问类内部的私有成员，例如：

#include <iostream>

using namespace std;

class a // 定义了类a

{

    long a0; // 定义私有成员 a0

    public:

    a(long b)

    {

        a0=b;

    }

    void geta()

    {

        cout<<a0<<endl;

    }

};

int main()

{

    a b(5); // 定义对象b，并给 b 中的 a0 赋初值

    long \*p;

    p=(long\*)&b; // 令指针 p 指向 b 中前 4 个字节，在这里相当于指向 a0

    b.geta(); // 用内部函数访问 a0

    cout<<\*p<<endl; // 在外部直接访问 a0

    \*p=8; // 在外部改变 a0 的值

    b.geta(); // 输出改变后的结果

    cout<<\*p<<endl;

    return 0;

}

需要注意的是，使用这种方法虽然可以用于基于类的多态原则的一些程序开发，但违反了类的封装原则，在使用指针的类中也极不安全，所以不建议使用。

# C++最常迷惑的问题

## const指针与指向const的指针

### 指向const的指针

**指向const的指针**：指针指向的内容是不能被修改的。有两种写法：

const int\* p; （推荐）

int const\* p;

理解：这一种可以理解为p是一个指针，它指向的内容是const int类型。p本身不用初始化它可以指向任何标识符，但它指向的内容是不能被改变的。

注：const指针中const总是出现在\*号的左边。

### const指针

**const指针**：指针本身的值是不能修改的。只有一种写法：

int a = 3;

int\* const p = 一个地址。

\*p = 4;

理解：p是一个指针，这个指针是指向int的const指针，它指向的值是可以被改变的。但是因为指针本身的值是不能被修改的所以它必须被初始化。

注：const指针中const总是出现在\*号的右边。

### 指向const的const指针

**指向const的const指针**：指针本身的值和指针指向的内容都是不能被修改的。两种写法：

const int\* const p = 一个地址；

int const\* const p = 一个地址；

理解：本质上就是以上两种的综合，不再细讲。

# VisualStudio2017使用

## 代码格式化

ctrl + A；  
ctrl + K + F;