全面涵盖 C++ 语法精粹，为初学者量身定制的 C++ 学习指南


极客学院出版

# 前言

C++ 是由 Bjarne Stroustrup 于 1979 年开始在贝尔实验室开发的一个中级编程语言。 C++ 可运行在不同的平台，如 Windows，Mac OS 和各种版本的 UNIX。

本文将通过简单实用的方法来带你学习 C++ 编程语言。

## 适用人群

本文是为新手所准备的，可以帮助他们理解从基本到高级的有关 C++ 编程语言的概念。

## 学习前提

在你开始做本文提供的各种类型例子练习之前，我们假设你已经知道什么是计算机程序，什么是计算机编程语言？

# 目录

# 1

# C++ 基础

# 概述

C++ 是静态，可编译，通用，大小写敏感，格式自由的编程语言，它支持程序化，面向对象的，和泛型编程方式。

C++ 被看作是**中间层**语言，因为它同时包含了低级语言和高级语言的特性。

C++ 是于 1979 年在新泽西的茉莉山丘的贝尔实验室由 Bjarne Stroustrup 开发的，它是 C 语言的加强版，最开始它被称作 “C with Classes”，但是后来在 1983 年被更名为 C++。

C++ 是 C 语言的超集，也就是说任何合法的 C 程序它同时也是合法 C++ 程序。

**注意：** 编程语言使用静态类型指的是对于类型检查是在编译的时候进行，而不是在运行期检查。

## 面向对象编程

C++ 完全支持面向对象编程，它包含了面向对象开发的四个特性：

- 封装
- 数据隐藏
- 继承
- 多态

## 标准库

标准的 C++ 包含三个重要的部分：

- 语言的核心部分提供了编程所需的基本构件，比如变量定义，数据类型和字面值等；
- C++ 标准库提供了丰富的函数操作，例如对文件和字符串的操作等；
- 标准模板库(STL)提供了许多的操作数据结构的方法

## ANSI 标准

ANSI 标准试图确保 C++ 的可移植性，也就是说，你所编写的代码利用了微软的编译器编译之后没有错误，那么它在 Mac，NIX，Windows box，或者 Alpha 上同样没有错误。

ANSI 标准到最近基本上还保持不变，并且所有的 C++ 的编译器生成商支持 ANSI 标准。

## 学习 C++

学习 C++ 的时候最重要的事情是关注与它的概念，而不是被语言的具体技术细节给弄晕了。

学习一门编程语言的目的是成为一名更优秀的程序员，也就是说在设计、实现一个新的系统和维护旧的程序时，能够更加高效的工作。

C++ 支持各种编程风格。 你可以按照 Fortran， C, Smalltalk 等任何语言的风格进行编程。每种编程风格均可以实现运行期和空间高效性这一目标。

## C++ 的使用

C++ 被几十万的程序员所使用，它是进行应用程序开发中必不可少的一部分。

C++ 被大量的使用在编写设备驱动程序，和那些有实时性限制，需要直接操作硬件的软件。

不管是使用 Macintosh 或者使用运行 Windows 操作系统的电脑的用户都直接的使用了 C++，因为这些系统的主要用户界面是由 C++ 编写的。

# 开发环境

## 本地开发环境设置

如果你想要本地的 C++ 开发环境，你应该确保如下的两个软件已经安装在你的电脑上：

### 文本编辑器

文本编辑器用来编写程序。举几个编辑器的例子： Windows 的 NotePad，一些操作系统提供的 Edit 命令，Brief，Epsilon，EMACS，和 vim 或者 vi。

文本编辑器的名称和版本在不同的操作系统上可能有差异。例如，Windows 操作系统上可用 NotePad，在 Windows 和 Linux 或者 UNIX 上均可以使用 vim 或者 vi。

你用编辑器创建的文件称为源文件，对于 C++ 而言，这些文件的很明显都是用 .cpp，.cp，或者 .c 为扩展名(后缀名)。

在开始编程之前，请确保你有一个文本编辑器可用，并且你有足够的经验编写 C++ 程序。

### C++ 编译器

C++ 编译器的作用是编译你的源代码，最终将它转换成可执行程序。

大多数 C++ 编译器并不在意你的源代码文件的扩展名，如果你没有指定它的扩展名，许多编译器都会用 .cpp 作为文件的默认扩展名。

最常用并且免费的编译器是 GNU C/C++ 编译器，另外如果你有其他的操作系统，你也可以使用 HP 或者 Solaris 的编译器。

## Installing GNU C/C++ 编译器

### UNIX/Linux 安装

如果你使用 Linux 或者 UNIX 系统，通过在命令行中输入如下的命令检查你的系统是否已经安装了 GCC：

```
$ g++ -v
```

如果已经安装了 GCC， 那么在控制台中应该输出类似如下的信息：

```
Using built-in specs.
Target: i386-redhat-linux
Configured with: ../configure --prefix=/usr .......
Thread model: posix
gcc version 4.1.2 20080704 (Red Hat 4.1.2-46)
```

如果 GCC 没有安装，那么你需要自己手动安装，你可以从 http://gcc.gnu.org/install/ 这里获得更详细的说明。

### Mac OS X 安装

如果你使用 Mac OS X 系统，那么得到 GCC 最方便的方式是从苹果的网站上下载 Xcode 开发环境， 按照如下的链接中的说明进行安装。

Xcode 现在可用的链接是： developer.apple.com/technologies/tools/

### Windows 安装

为了在 Windows 上安装 GCC， 你需要安装 MinGW。为了安装 MinGW，你可以访问 MinGW 的主页 www.mingw.org

接着访问 MinGW 下载页那个链接。下载最新版的 MinGW 安装程序，这个程序的名称应该是类似于 `MinGW-<版本号>.exe` 这样的形式。

在安装 MinGW 的时候，最低限度，你必须要安装 gcc-core，gcc-g++，binutils，和 MinGW 运行时环境，当然你也可以选择更多进行安装。

添加你安装 MinGW 的子目录 bin 的路径到你的 PATH 环境变量中，这样你就可以通过在命令行中输入 MinGW 提供的工具的名称来使用这些工具。

当安装完成之后，你可以在 Windows 的命令行中运行 GNU 提供的几个工具，类似于：gcc，g++，ar，ranlib，dlltool 等。

# 基本语法

C++ 程序,它可以被定义为一个对象集合，彼此之间通过调用对方的方法进行通信。现在让我们简要地看看类，对象，方法和实例变量的意思。

- 对象： 对象具有状态和行为。例子:一只狗拥有的状态——颜色、名称、品种以及行为——摇尾巴,吠叫,吃。对象是类的一个实例。
- 类：一个类可以定义为一个模板/复写纸，它描述的是它所支持类型对象的行为或者状态。
- 方法：一个方法基本上是一个行为。一个类可以包含许多方法。在这些方法中，你可以编写你的业务逻辑代码，你可以进行数据操作。
- 实例变量：每个对象都有其独特的实例变量集合。通过给这些实例变量进行赋值，从而这个对象的状态被创建了。

## 程序结构

首先让我们看一个简单的示例代码，这个程序会输出 Hello World。

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

// main() is where program execution begins.

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

让我们来看下以上程序的各个部分：

- C++ 语言定义了几个头，其中包含的信息对你的程序要么是必要的或要么是有用的。对于这个程序,头部 `<iostream>` 是必要的。
- `using namespace std;` 这一行告诉编译器使用 std 命名空间。命名空间是 C++ 中一个相对较新的特性。
- 下一行 `//main() is where program execution begins` 是一个在 C++ 可以用来表示注释的语句。单可以使用 // 进行单行注释。
- `int main()` 这一行是程序的主方法，也是程序执行的入口。

- 下一行 `cout << "This is my first C++ program"` 会将 `”This is my first C++ program”` 显示在屏幕上。
- 下一行 `return 0;` 会终止 `main()` 函数的执行，并将 0 返回给调用进程。

## 编译和执行 C++ 程序

接着让我们看看如何保存源文件，编译和执行程序。请按照如下的步骤执行：

- 打开一个文本编辑器，添加上面的代码；
- 保存该文件，并以 hello.cpp 为该文件名；
- 打开一个命令行程序，接着切换目录为你保存 hello.cpp 这个文件的目录；
- 在命令行中输入 g++ hello.cpp，接着按下回车按键，编译你的程序。如果你的代码中没有错误，在命令行控制窗口中提示符将会跳转到下一行，并将生成一个 a.out 的可执行文件。
- 接着，输入 a.out 来运行你的程序；
- 你将会在你当前的窗口中看到输出 Hello World。

```
$ g++ hello.cpp
$ ./a.out
Hello World
```

确保 g++ 在你的路径和你目录包含文件 hello.cpp 运行它。

你可以编译 C/C++ 程序使用 makefile。

## C++ 中的分号和语句块

在 C++ 中分号是终止符。也就是说，每一个独立语句都必须以分号结束。它表明了一个逻辑实体的结束。

例如，如下是三个不同的语句：

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

一个语句块是一组逻辑上连接的语句，它们被左右花括号包围住。例如：

```
    {
        cout << "Hello World"; // prints Hello World
```

```
    return 0;
}
```

C++ 并不是把一行的结束作为一个终结符。由于这个原因，它也就不在意你在一行中哪里放置你的语句。例如：

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

这个也等同于：

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

## C++ 标识符

一个 C++ 标识符是用来标识一个变量、函数、类、模块，或任何其他用户定义的项目。一个标识符以字母 A 到 Z 或者 a 到 z 或下划线(_)开始，接着后面是零个或多个字母，下划线，数字(0 - 9)。

C++ 不允许用一些特殊符号作为标识符，如 `@、 $、%。C++` 是一种区分大小写的编程语言。因此， `Manpower` 和 `manpower` 在 C++ 中是两个不同的标识符。

如下是一些合法的标识符的例子：

```
mohd       zara    abc    move_name  a_123
myname50   _temp   j      a23b9      retVal
```

## C++ 关键字

如下的列表展示了 C++ 中的关键字。这些关键字不能用作常量或变量或其他标识符的名称。

| | | | |
|---|---|---|---|
| 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 | |

## 三元符

有些字符有其他的表示含义，称为三元字符序列。三元字符是由三个字符组成，当表示一个字符序列并且该序列总是以两个问号开始。

三元符在任何地方出现时都会被转义，包括在字符串字面值和字符字面值，注释，和预处理器指令中。

以下是最常用的三元符序列:

| Trigraph | Replacement |
|---|---|
| ??= | # |
| ??/ | \ |
| ??' | ^ |
| ??( | [ |
| ??) | ] |
| ??! | \| |
| ??< | { |
| ??> | } |
| ??- | ~ |

所有的编译器不支持三元符，也不建议使用，因为它们很容易让人混淆。

## C++ 中的空格

只包含空格，可能有注释的行被称为一个空行，并且 C++ 编译器会完全忽略它。

空格是 C++ 中使用的术语来描述空格、制表符、换行符和注释。空格将一部分注释与其他的部分分隔开，从而使编译器能够识别一个元素在语句中的位置，例如 int 的结束，和下一个元素的开始位置。因此，在下面的语句中，

```
    int age;
```

在 int 和 age 之间必须有至少一个空格字符(通常是一个空格)，这样编译器才能够区分它们。另一方面，在如下语句中

```
    fruit = apples + oranges: //Get the total fruit
```

在 fruit 和 = 之间，或者 = 和 apples 之间是不需要空格的，尽管这样做不会有什么影响。如果为了使程序阅读起来更方便，你可以在它们之间添加空格。

# 注释

程序注释是说明性语句，您可以在你编写的 C++ 代码中添加注释，从而帮助任何人阅读源代码。所有的编程语言都支持某种形式的注释。

C++ 支持单行和多行注释。在注释中可以使用所有的字符，并且它们都将会被 C++ 编译器忽略。

C++ 注释以 `/*` 开始和以 `*/` 结束。例如:

```
/* This is a comment */

/* C++ comments can  also
* span multiple lines
 */
```

注释还可以以 // 开始，一直到行的结束位置。例如:

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

main()
{
    cout << "Hello World"; // prints Hello World

    return 0;
}
```

上面的代码编译时,将忽略 //prints Hello World，最终可执行程序将产生以下结果:

```
Hello World
```

`/*` 和 `*/` 注释， `//` 字符没有特殊的意义。在 `//` 注释的内部， `/*` 和 `*/` 没有特殊的意义。因此，您可以在在一种注释中嵌套另外一种注释。例如：

```
/* Comment out printing of Hello World:

cout << "Hello World"; // prints Hello World

*/
```

# 数据类型

当在使用任何编程语言编程时，您需要使用各种不同的变量来存储不同的信息。变量只是让内存预留位置来存储一些值。这意味着，当你创建一个变量时在内存中会保留一些空间给该变量。

你可能喜欢使用像字符类型，宽字符，整数，浮点数，双精度浮点数，布尔类型等各种数据类型来存储信息。操作系统基于变量的数据类型来分配内存空间和决定什么可以存储在该保留内存区域。

## 基本内置类型

C++ 给程序员提供了一系列丰富的内置类型以及用户定义的数据类型。下表列出了七个基本的 C++ 数据类型:

| 类型 | 关键字 |
| --- | --- |
| Boolean | bool |
| Character | char |
| Integer | int |
| Floating point | float |
| Double floating point | double |
| Valueless | void |
| Wide character | wchar_t |

上面的几个基本类型可以使用一个或多个如下的修饰符这来修饰:

- signed
- unsigned
- short
- long

下面的表显示了变量类型，该类型的值在内存中需要多少内存空间来存储，该类型能够表示的最大值和最小值。

| 类型 | 长度 | 范围 |
| --- | --- | --- |
| char | 1byte | -127 - 127 或者 0 - 255 |
| unsigned char | 1byte | 0 - 255 |
| signed char | 1byte | -127 - 127 |
| int | 4bytes | -2147483648 - 2147483647 |
| unsigned int | 4bytes | 0 - 4294967295 |
| signed int | 4bytes | -2147483648 - 2147483647 |

| | | |
|---|---|---|
| short int | 2bytes | -32768 - 32767 |
| unsigned short int | Range | 0 - 65,535 |
| signed short int | Range | -32768 - 32767 |
| long int | 4bytes | -2,147,483,647 - 2,147,483,647 |
| signed long int | 4bytes | 和 long int 一样 |
| unsigned long int | 4bytes | 0 - 4,294,967,295 |
| float | 4bytes | +/- 3.4e +/- 38 (~7 位数字) |
| double | 8bytes | +/- 1.7e +/- 308 (~15 位数字) |
| long double | 8bytes | +/- 1.7e +/- 308 (~15 位数字) |
| wchar_t | 2 或者 4 bytes | 1 个宽字符 |

变量类型占用空间的实际大小可能和上表展示的有些不同,这取决于您所使用的编译器和电脑操作系统。

下面示例将输出各种数据类型在您的计算机上实际占用的内存空间大小。

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

int main()
{
    cout << "Size of char : " << sizeof(char) << endl;
    cout << "Size of int : " << sizeof(int) << endl;
    cout << "Size of short int : " << sizeof(short int) << endl;
    cout << "Size of long int : " << sizeof(long int) << endl;
    cout << "Size of float : " << sizeof(float) << endl;
    cout << "Size of double : " << sizeof(double) << endl;
    cout << "Size of wchar_t : " << sizeof(wchar_t) << endl;
    return 0;
}
```

这个示例使用 endl，这个表示在每一行之后插入一个换行符，<< 操作符被用来将多个值输出到屏幕上。我们也使用 sizeof() 函数来获得各种数据类型的存储大小。

上面的代码编译和执行时,它会产生以下结果，这个可能会随着机器的不同而显示不同的结果:

```
Size of char : 1
Size of int : 4
Size of short int : 2
Size of long int : 4
Size of float : 4
Size of double : 8
Size of wchar_t : 4
```

## typedef 声明

你可以使用 typedef 为已经存在的数据类型起一个新的名称。下面是一个使用 typedef 定义一个新类型的简单语法方式:

```
typedef type newname;
```

例如,下面告诉编译器，feet 是 int 的另一个名字:

```
typedef int feet;
```

现在，下面的声明是完全合法和创建一个整数变量称为 distance:

```
feet distance;
```

## 枚举类型

枚举类型声明了一个可选的类型名和一组值为零的或多个标识符的类型。每个枚举器是一个常数，它的类型是枚举。

创建一个枚举类型需要使用关键字 enum。枚举类型的一般形式是:

```
enum enum-name { list of names} var-list;
```

在这里，enum-name 是枚举类型名。list of name 是由逗号分隔开的。

例如，下面的代码定义了一个称为 color 的颜色枚举类型，并且变量 c 的是一个 color 类型的枚举变量。最后， c 被赋值为 “blue”。

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

默认情况下,枚举类型花括号中第一个变量被赋值为 0，第二个变量赋值为 1，第三个赋值为 2，依此类推。但是你也可以给某个指定变量一个初始值。例如,在下面的枚举类型， green 将会被初始化为 5。

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

这里，blue 的值将会变成 6，因为每个变量都会比它前一个变量大 1。

# 变量类型

变量名给我们提供了在程序中我们可以使用的空间信息。每个变量在 C++ 中都有一个特定的类型，它决定变量在内存中的大小和布局；在该内存空间可以存放取值范围；和一组可以应用于该变量的操作。

一个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始。大写和小写的字母是不同，因为 C++ 是区分大小写的:

C++ 中有以下基本变量类型，我们在上一节已经介绍过:

| 类型 | 描述 |
| --- | --- |
| bool | 存储的值为 true 或者 false |
| char | 通常是 8 位(一位字节)。它是个整数类型。 |
| int | 机器中最常用的整数类型。 |
| float | 单精度浮点类型的值。 |
| double | 双精度浮点类型的值。 |
| void | 代表着缺失类型。 |
| wchar_t | 一个宽字符类型。 |

C++ 还允许定义各种其他类型的变量，我们将在后续章节将会介绍的，比如 Enumeration，指针，数组，引用，数据结构和类。

下一节将介绍如何定义，声明和使用各种类型的变量。

### C++ 中变量的定义

一个变量定义意味着告诉编译器，存储在哪里以及需要多少的存储空间。变量定义要指定数据类型，同时包含该类型的一个或多个变量的列表：

```
type variable_list;
```

在这里，type 必须是一个有效的 C++ 数据类型，包括 char，w_char，int，float，double，bool 或者任何用户自定义的对象等。variable_list 要包含一个或多个由逗号分隔的标识符。如下是一些有效的声明示例：

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

int i,j,k; 这一行同时声明和定义了变量 i，j 和 k，这条语句指示编译器创建类型为 int，名称为 i，j 和 k 的变量。

变量在声明的同时可以进行初始化(分配一个初始值)。初始化由一个等号后面跟着一个常数表达式如下:

```
type variable_name = value;
```

如下是一些示例：

```
extern int d = 3, f = 5;    // declaration of d and f.
int d = 3, f = 5;           // definition and initializing d and f.
byte z = 22;                // definition and initializes z.
char x = 'x';               // the variable x has the value 'x'.
```

没有初始化的定义：静态类型的变量会隐式地被初始化为 NULL(所有位值为0)，而其他的所有变量的初始值是未知的。

### C++ 中变量声明

变量声明为编译器提供保证，对于给定的类型和名称的变量是唯一的，从而编译器在进一步进行编译变量时不需要变量的完整细节。变量声明只是在编译时有意义的，因为编译器在进行程序的链接时需要变量声明的信息。

当你使用多个文件，并且你自己定义的变量放在其中一个文件里，变量的声明将对程序的链接很有用。您可以使用 extern 关键字来声明一个放在任何位置的变量。虽然你在你的 C++ 程序中你可以声明一个变量多次，但它在一个文件中，一个函数或一块代码中只能定义一次。

### 示例

试试下面的例子，一个变量已经在顶部进行了声明，但同时它也在 main 函数中被定义了:

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

// Variable declaration:
extern int a, b;
extern int c;
extern float f;

int main ()
{
    // Variable definition:
    int a, b;
```

```
    int c;
    float f;

    // actual initialization
    a = 10;
    b = 20;
    c = a + b;

    cout << c << endl ;

    f = 70.0/3.0;
    cout << f << endl ;

    return 0;
}
```

上面的代码编译和执行后，它产生以下结果：

```
30
23.3333
```

相同的概念也可以应用于函数声明，当你对一个函数进行声明时，它的定义可以在其他任何地方。例如：

```
// function declaration
int func();

int main()
{
    // function call
    int i = func();
}

// function definition
int func()
{
    return 0;
}
```

### 左值和右值

C++ 中有两种表达式：

- 左值：指向内存位置的表达式称为左值表达式。一个左值可能出现在赋值语句的左边或右边。

- 右值：右值是指一个数据值存储在某个内存地址中。一个右值是一个表达式，它不能被赋值，这意味着一个右值可能出现在赋值语句的右边，而不是左边。

变量是左值,因此可能会出现在赋值语句的左边。数字字面值是右值，因此不能被赋值，不能出现赋值语句的在左边。下面是一个有效的语句:

```
int g = 20;
```

但如下却不是一个有效的赋值语句，会产生编译期错误:

```
10 = 20;
```

# 变量作用域

变量作用域就是指变量在程序中能够操作的区域，通常按照在程序中不同地方声明可以分为如下三类：

1. 局部变量：在函数中或者由花括号括起来的代码块中声明。
2. 形式变量：在函数的参数中声明，也叫形参。
3. 全局变量：在所有函数之外声明的变量。

关于函数及函数参数的内容我们会在后续章节中学习。这里我们首先学习关于局部变量和全部变量的相关内容。

## 局部变量

在函数或代码块内部声明的变量称为局部变量。他们在函数体内声明后仅能被其声明的所在函数体内部的后续语句操作。局部变量不能被函数外部访问到。下面就是使用局部变量的例子。

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

int main ()
{
  // Local variable declaration:
  int a, b;
  int c;

  // actual initialization
  a = 10;
  b = 20;
  c = a + b;

  cout << c;

  return 0;
}
```

## 全局变量

全局变量通常会被声明定义在所有函数体的外面，大部分情况下是在程序的最上方定义。全部变量的生命周期就是进程从开始到程序执行结束的整个过程。

全局变量可以被任何函数访问。也就是说，全局变量一旦被声明，将在程序的整个生命周期内都是有效的。下面是使用全局和局部变量的例子：

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

// Global variable declaration:
int g;

int main ()
{
  // Local variable declaration:
  int a, b;

  // actual initialization
  a = 10;
  b = 20;
  g = a + b;

  cout << g;

  return 0;
}
```

程序中的局部变量和全局变量可以有相同的变量名称。但是在局部变量所在函数体内如，使用变量名仅能访问到局部变量。比如：

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

// Global variable declaration:
int g = 20;

int main ()
{
  // Local variable declaration:
  int g = 10;

  cout << g;

  return 0;
}
```

上述例子中的程序被编译并执行后，将显示如下结果：

```
10
```

## 局部变量和全局变量的初始化

局部变量被定义后，默认情况下，系统并不会对该变量进行初始化，所以需要程序员进行初始化操作。与之不同的是，全局变量定义后会被编译系统自动初始化，具体初始化的值如下：

| 数据类型 | 默认初始化的值 |
| --- | --- |
| int | 0 |
| char | '\0' |
| float | 0 |
| double | 0 |
| pointer | NULL |

适如其分的给变量初始化是一个很好的编程习惯，否则，有时程序会出现很多意想不到的错误。

# 常量

常量就是程序中不能被修改的固定的值，也称为字面值。

常量可以是任何基础数据类型，比如整型，浮点型，字符型，字符串和布尔类型。

通常情况下常量被认为是定义后不能更改的变量。

## 整型常量

整数常量可以是十进制、八进制或十六进制常量。对于十六进制可以用 ox 或 OX 前缀来标示，可以用 0 前缀来标示八进制，十进制默认不需要前缀标志。

整数常量也可以与 U 和 L 后缀一起搭配使用，相应的来表示无符号和长整型。后缀对大小写不敏感。

下面是整型常量的一些例子：

```
212 // 合法
215u// 合法
0xFeeL  // 合法
078 // 不合法：8不是合法的八进制
032UU   // 不合法：U后缀不能重复使用
```

下面试其他的关于不同类型的整型常量：

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

## 浮点常量

浮点常量由整数部分、小数点、小数部分和一个指数部分组成。可以使用十进制或者指数的形式来表示浮点常量。

使用指数表示时，必须包含小数点,指数,或两者同时使用。相对应的，使用十进制形式表示，必须包含整数部分，小数部分，或两者兼而有之。有符号指数可以用 e 或 E 来表示。

下面是一些浮点常量的一些例子：

```
3.14159    // 合法
314159E-5L// 合法
510E  // 不合法：指数部分不完整
210f  // 不合法：没有整数部分
.e55  // 不合法：无整数或小数部分
```

## 布尔常量

C++ 中只有两个布尔类型的常量，他们都是标准 C++ 中的关键词。

1. true ：代表肯定的
2. false ：代表否定的

需要注意的是不可以将true看做是1，也不可以将false看做是0.

## 字符常量

字符常量应该用单引号括起来。如果字符常量以 L 开头（仅大写），那么它将是一个宽字符常量且需要使用 wchar_t 类型的变量来存储。否则，它将是个窄字符常量且存储在 char 类型的变量中。

字符常量可以还是一个普通字符（比如 'x'）、转义字符（'\t'）或者是一个通用字符（比如 '\u02C0'）.

C++ 中有一些使用反斜杠标志的字符，他们具有特殊的意义，且可以用于表示换行（\n）或者 tab 键（\t）。下面是一些转义字符的列表：

| 转义字符 | 意义 |
| --- | --- |
| \\ | \字符 |
| \' | '字符 |
| \'' | ''字符 |
| \? | ?问号字符 |
| \a | 警告 |
| \b | 退格 |
| \f | 换页 |
| \n | 换行 |
| \r | 回车 |
| \t | 横向tab |
| \ooo | 八进制的一到三位数 |

| | |
|---|---|
| \xhh... | 一位或多位的十六进制数 |

下面是一些展示转义字符的例子：

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

int main()
{
   cout << "Hello\tWorld\n\n";
   return 0;
}
```

当上述代码先编译再执行后，会出现如下结果：

```
Hello   World
```

## 字符串常量

字符串常量需要使用双引号括起来。字符串常量跟字符常量一样，包括：普通字符、转义字符和通用字符。

可以使用空格将一行字符分割成多个行。

如下是一些字符串常量的例子。所有的三种形式是相同的字符串。

```
"hello, dear"

"hello, \

dear"

"hello, " "d" "ear"
```

## 默认常量

如下为 C++ 中的两个简单的默认常量。

1. #define：预处理
2. const：关键字

如下是详细的例子：

```
#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;
}
```

当上述代码先编译再执行后，会出现如下结果：

```
50
```

## const 关键字

可以使用const前缀来声明特殊类型的常量，比如：

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

下面是比较具体的例子：

```
#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;
}
```

当上述代码先编译再执行后，会出现如下结果：

```
50
```

> 注意：在程序首部定义常量是一个良好的编程习惯。

## 修饰符的类型

C++ 允许 char、int 和 double 类型的数据可以在其前面使用修饰符。修饰符用于更改数据变量的意义以实现变量可以更加精准的运用到其所应用的环境中。

数据类型的修饰符如下：

1. signed：有符号类型
2. unsigned：无符号类型
3. long：长整型
4. short：短整型

signed、unsigned、long 和 short 可以应用到整型基础类型。此外，signed 和 unsigned 可以应用到char类型，long 可以应用到 double 类型。

signed 和 unsigned 也可以作为 long 或 short 修饰符的前缀。比如，unsigned long int.

C++ 也允许使用简化字符的方式来声明 unsigned、short 或 long 整数。程序员可以仅使用 unsigned、short 或 long 而不使用 int 来定义整型变量。这里的 int 就被简化掉了。比如，下面的两句程序均实现对 unsigned 整型变量的声明的功能。

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

为了理解 C++ 中 signed 和 unsigned 整数修饰符的不同。可以尝试着运行下面的程序：

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

/* This program shows the difference between
 * signed and unsigned integers.
*/
int main()
{
   short int i;   // a signed short integer
   short unsigned int j;  // an unsigned short integer

   j = 50000;

   i = j;
   cout << i << " " << j;
```

```
    return 0;
}
```

上述程序执行结果如下：

```
-15536 50000
```

上述结果的背后原因是，unsigned 短整型变量的值为 5000，当时 short 类型时，就是-15536 了。这和值表示范围有关系。

## C++ 中的类型限定符

类型限定符提供了关于变量保存值更丰富的信息:

| 限定符 | 意义 |
| --- | --- |
| const | const 类型修饰的对象在起运行周期内不可被改变 |
| volatile | volatile 修饰符用于提示编译器，程序中某个变量值的改变可能不是程序显式修改的 |
| restrict | restrict 限定符修饰的指针意味着所有修改该指针所指向内容的操作全部都是基于该指针的。仅在 C99 标准中增加了这个修饰符。 |

# 存储类型

存储类型定义了变量或函数的作用范围及生命周期。这些说明符也声明了他们的修改方式的类型。有如下几种存储类型：

1. auto
2. register
3. static
4. extern
5. mutable

## auto 存储类型

auto存储类型是所有局部变量的默认存储类型。

```
{
   int mount;
   auto int month;
}
```

上面的例子中定义了两个相同存储类型的变量，auto 仅能运用于函数内的局部变量。

## register 存储类型

register 存储类型用于定义存储于寄存器中的变量而不是内存中。这意味着该变量的最大尺寸将是寄存器的大小（通常是一个字），并且不能使用 '&' 寻址运算符进行操作（因为它没有内存地址）。

```
{
   register int  miles;
}
```

register类型应该仅应用于需要快速访问的变量，比如计数器。需要注意的是，定义 register 类型的变量并不意味着该变量一定就存储在寄存器中，这仅仅意味着需要按照硬件以及具体实现的限制来判定到底是不是存储在寄存器中。

## static 存储类型

static 存储类型的变量意味着该变量将会从始至终地存活在程序的整个生命周期内，而不会随着每次访问到它所在的代码块时就建立该变量，离开代码块时就销毁该变量。因此，局部变量静态化可以使他们在函数调用时仍保有其值。

static 修饰符也可以应用于全局变量。 当全局变量使用该修饰符后， 该全局变量就被限制在其声明的文件内。

在 C++中，当 static 应用于类的数据成员时，它所起到的作用是多个该类的成员变量都是指的同一个变量。

```
#include <iostream>

// Function declaration
void func(void);

static int count = 10; /* Global variable */

main()
{
while(count--)
{
   func();
}
return 0;
}
// Function definition
void func( void )
{
static int i = 5; // local static variable
i++;
std::cout << "i is " << i ;
std::cout << " and count is " << count << std::endl;
}
```

当上述代码被编译后执行，其结果如下：

```
i is 6 and count is 9
i is 7 and count is 8
i is 8 and count is 7
i is 9 and count is 6
i is 10 and count is 5
i is 11 and count is 4
i is 12 and count is 3
```

```
i is 13 and count is 2
i is 14 and count is 1
i is 15 and count is 0
```

## extern 存储类型

**extern** 存储类型用于使全局变量的引用对所有程序文件可见。如果前面已经定义了一个变量名，那么就不能再使用 extern 来声明同一变量名的变量了。

当你有多个程序文件且需要定义一个可以在其他文件用可以访问到的变量或函数时，就可以在其他文件中使用 *extern* 声明该变量或函数的引用。

extern 修饰符通常被应用于多个文件中需要共享相同的全局变量或函数的情况。一个例子如下：

第一个文件：main.CPP

```
#include <iostream>

int count ;
extern void write_extern();

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

第二个文件：support.cpp

```
#include <iostream>

extern int count;

void write_extern(void)
{
   std::cout << "Count is " << count << std::endl;
}
```

这里的 *extern* 关键用于声明 count 变量已经在其他文件中定义了。按照下面的方式来编译：

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

这样会生出一个 write 可执行文件，运行它并看他的结果：

```
$./write
5
```

## mutable 存储类型

**mutable** 修饰符应用于类对象，将会在后续章节中详细讨论。它允许对象的成员可以覆盖常量。也即是说，mutable 成员可以被 const 成员函数所修改。

# 操作符

操作符就是告诉编译器来执行数学或逻辑运算操作的符号。C++ 有非常丰富的内置操作符。提供如下几类的操作符：

1. 数学运算操作符
2. 关系运算操作符
3. 逻辑运算操作符
4. 位运算操作符
5. 赋值运算操作符
6. 复合运算操作符

下面的章节将一一介绍数学、关系、逻辑、位运算、赋值和其他操作符。

## 数学运算操作符

下面的就是 C++ 语言所支持的数学运算操作符：

假设变量 A 存储 10，变量 B 存储 20，那么：

| 运算符 | 描述 | 例子 |
| --- | --- | --- |
| + | 两个运算数相加 | A + B = 30 |
| - | 第一个运算数减去第二个运算数 | A - B = -10 |
| * | 运算数相乘 | A * B = 200 |
| / | 分子除以分母 | B / A = 2 |
| % | 模数运算符，整除后的余数 | B % A = 0 |
| ++ | 增量运算符，整数值逐次加1 | A++ = 11 |
| -- | 减量运算符，整数值逐次减1 | A-- = 9 |

## 关系运算符

下面的就是 C++语言所支持的关系运算操作符：

假设变量 A 存储 10，变量 B 存储 20，那么：

| 运算符 | 描述 | 例子 |
| --- | --- | --- |

| | | |
|---|---|---|
| == | 检查两个运算数的值是否相等，如果是，则结果为 true | A == B 为 false |
| != | 检查两个运算数的值是否相等，如果不相等，则结果为 true | A != B 为 true |
| > | 检查左边运算数是否大于右边运算数，如果是，则结果为 true | A > B 为 false |
| < | 检查左边运算数是否小于右边运算数，如果是，则结果为 true | A < B 为 true |
| >= | 检查左边运算数是否大于或者等于右边运算数，如果是，则结果为 true | A >= B 为 false |
| <= | 检查左边运算数是否小于或者等于运算数，如果是，则结果为 true | A <= B 为 true |

## 逻辑运算符

下面的就是 C++语言所支持的逻辑运算操作符：

假设变量 A 存储 1，变量 B 存储 0，那么：

| 运算符 | 描述 | 例子 |
|---|---|---|
| && | 称为逻辑与运算符。如果两个运算数都非零，则结果为 true。 | A && B 为 true |
| \|\| | 称为逻辑或运算符。如果两个运算数中任何一个非零，则结果为 true。 | A \|\| B 为 true |
| ! | 称为逻辑非运算符。用于改变运算数的逻辑状态。如果逻辑状态为 true，则通过逻辑非运算符可以使逻辑状态变为 false | !(A && B) 为 false |

## 位运算符

位操作运算是按位来进行操作的。与、或、非和异或的真值表如下：

| 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 |

假设变量 A 存储 60，变量 B 存储 13，那么：

下面的就是 C++ 语言所支持的位运算操作符：

假设变量 A 存储 60，变量 B 存储 13，那么：

```
A = 0011 1100

B = 0000 1101

A&B = 0000 1100
```

```
A|B = 0011 1101

`A^B` = 0011 0001

~A  = 1100 0011
```

| 运算符 | 描述 | 例子 |
| --- | --- | --- |
| & | 称为按位与运算符。它对整型参数的每一个二进制位进行布尔与操作。 | A & B = 12 . |
| \| | 称为按位或运算符。它对整型参数的每一个二进制位进行布尔或操作。 | A \| B = 61. |
| ^ | 称为按位异或运算符。它对整型参数的每一个二进制位进行布尔异或操作。异或运算是指第一个参数或者第二个参数为true，并且不包括两个参数都为 true 的情况，则结果为true。 | (A ^ B) = 49. |
| ~ | 称为按位非运算符。它是一个单运算符，对运算数的所有二进制位进行取反操作。 | ~B = -61 . |
| << | 称为按位左移运算符。它把第一个运算数的所有二进制位向左移动第二个运算数指定的位数，而新的二进制位补0。将一个数向左移动一个二进制位相当于将该数乘以2，向左移动两个二进制位相当于将该数乘以4，以此类推。 | A << 1 =120. |
| >> | 称为按位右移运算符。它把第一个运算数的所有二进制位向右移动第二个运算数指定的位数。为了保持运算结果的符号不变，左边二进制位补 0 或 1 取决于原参数的符号位。如果第一个运算数是正的，运算结果最高位补 0；如果第一个运算数是负的，运算结果最高位补 1。 | A >> 1 = 15. |
| >>> | 称为 0 补最高位无符号右移运算符。这个运算符与>>运算符相像，除了位移后左边总是补0. | A >>> = 15. |

## 赋值运算符

下面的就是C++语言所支持的赋值运算操作符：

| 运算符 | 描述 | 例子 |
| --- | --- | --- |
| = | 简单赋值运算符，将右边运算数的值赋给左边运算数 | C = A + B 将 A+B 的值赋给 C |
| += | 加等赋值运算符，将右边运算符与左边运算符相加并将运算结果赋给左边运算数 | C += A 相当于 C = C + A |
| -= | 减等赋值运算符，将左边运算数减去右边运算数并将运算结果赋给左边运算数 | C -= A 相当于C = C - A |
| *= | 乘等赋值运算符，将右边运算数乘以左边运算数并将运算结果赋给左边运算数 | C *= A 相当于C = C * A |
| /= | 除等赋值运算符， 将左边运算数除以右边运算数并将运算结果赋值给左边运算数 | C /= A 相当于 C = C / A |
| %= | 模等赋值运算符，用两个运算数做取模运算并将运算结果赋值给左边运算数 | C %= A 相当于 C = C % A |
| <<= | 左移且赋值运算符 | C <<= 2 相当于 C = C << 2 |

| | | |
|---|---|---|
| >>= | 右移且赋值运算符 | C >>= 2 相当于 C = C >> 2 |
| &= | 位与且赋值运算符 | C &= 2 相当于 C = C & 2 |
| ^= | 位异或且赋值运算符 | C ^= A 相当于 C = C ^ A |
| \|= | 位或且赋值运算符 | C \|= A 相当于 C = C \| A |

## 复合运算符

下面是 C++ 支持的其他运算符：

| 操作符 | 描述 |
|---|---|
| sizeof | sizeof 操作符返回变量占用内存的大小。比如，sizeof（a），a 是一个整数时，返回 4 |
| 条件 ? X : Y | 条件操作符：如果条件判断为true，则返回 X，否则返回 Y. |
| , | 逗号操作符：可以使操作顺序执行。整体逗号表达式的值就是逗号最后表达式的返回结果。 |
| .(点) 和 ->(箭头) | 成员操作符：用于获取类、结构体或联合体成员的引用。 |
| 转换 | 转换操作符：可以将数值类型转成其他类型，比如，int（2.2000）将返回 2. |
| & | 取地操作符：可以返回一个变量的地址。比如， &a 将会返回变量实际的内存地址。 |
| * | 指针操作符：指向一个变量。比如 *var 意味着指向变量 var. |

## C++ 中操作符优先级

运算符优先级确定表达式中项的分组。这会影响如何表达一个表达式。某些操作符比其他有更高的优先级,例如,乘法运算符的优先级高于加法操作符。

比如 x=7+3*2，这里x的值是13而不是20，因为乘法优先级比加法高。所以应该先执行3乘2，然后再加7.

操作符的优先级如下表，上方的优先级比下方高。较高优先级的操作符优先进行计算。

| 分类 | 操作符 | 操作顺序 |
|---|---|---|
| 后缀运算 | () [] -> . ++ - - | 从左到右 |
| 一元运算 | + - ! ~ ++ - - (type)* & sizeof | 从右到左 |
| 乘法 | * / % | 从左到右 |
| 加法 | + - | 从左到右 |
| 移位 | << >> | 从左到右 |
| 相等 | == != | 从左到右 |
| 比较 | < <= > >= | 从左到右 |
| 位与 | & | 从左到右 |
| 异或 | ^ | 从左到右 |
| 位或 | \| | 从左到右 |

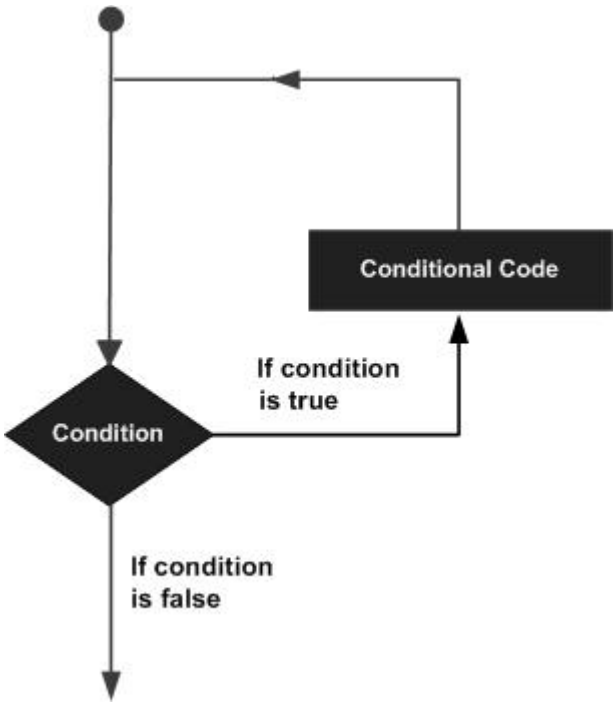
| | | |
|---|---|---|
| 逻辑与 | && | 从左到右 |
| \|\| | 逻辑或 | 从左到右 |
| 条件 | ?: | 从左到右 |
| 赋值 | = += -= *= /= %=>>= <<= &= ^= \|= | 从右到左 |
| 逗号 | , | 从左到右 |

# 循环的类型

程序员都会遇到需要多次执行同一代码段的情况。一般情况下，代码会顺序执行：函数中的第一句代码首先会被执行，后面的语句依次执行。

编程语言往往可以提供多种控制结构来实现更复杂的程序执行流程。

循环语句以执行单个语句或一组语句。下面是大部分编程语言中循环语句的一般模式：




C++ 语言支持下方的循环类型来满足循环的需求。点击链接查看详细情况。

| 循环类型 | 描述 |
| --- | --- |
| while 循环 | 当给定条件为 true 时，执行循环体。在每次执行循环体前都检查条件是否为 true。 |
| for 循环 | 按照条件执行循环体，可以简化循环体的结构 |
| switch 语句 | 一个 switch 语句允许一个变量针对多个不同的值分别进行验证是否满足条件。 |
| do…while 循环 | 与 loop 循环类似，不同的是在循环体后检查条件 |
| 嵌套循环 | 可以嵌套 for 或 loop 循环来多次执行循环体 |

## 循环控制语句

循环控制语句可以改变原有循环执行顺序。当循环体执行结束后，其范围内定义的对象都会被销毁。

C++ 语言支持下方的循环控制语句。点击链接查看详细情况。

| 循环语句 | 描述 |
| --- | --- |
| break | 终止当前 loop 或 switch 代码块，并且跳出后执行后续代码。 |
| continue | 跳出当前循环体，检测循环执行条件 |
| goto | 跳转到指定的代码标签处，不建议在程序中大量使用该功能。 |

## 无穷循环

如果循环条件无法变为 false 的话，那么该训话那就是无穷循环。for 循环就是实现无穷循环。for 循环条件中的三个表达式并不是必须的，所以只要将条件判断语句置空就可以实现无穷循环。

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

int main ()
{

   for( ; ; )
   {
  printf("This loop will run forever.\n");
   }

   return 0;
}
```

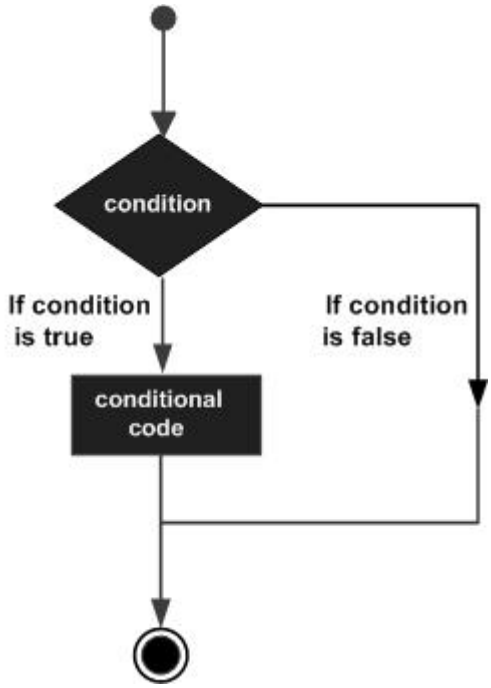
当条件判断语句置空后，就默认是 true. 程序员也可以保留初始化和递增表达式，但是 C++ 工程师一般都是使用 for（;;）来表达无穷循环。

**注意：**可以通过 Ctrl + C 的方式来终止无穷循环。

# 决策语句

决策结构需要程序员指定一个或多个可以被程序评估或测试的条件，以及一个语句或者是当条件被确定为真时可以用来执行的语句，和当条件被确定为假时，可以选择用来执行的其他语句。

以下是在大多数编程语言中找到的一个典型的决策结构的通用格式。




C++ 编程语言提供以下类型的决策语句。单击以下链接来查看它们的细节。

| 语句 | 描述 |
| --- | --- |
| if 语句 | 一个 if 语句由一个布尔表达式及紧随其后的一个或多个语句组成 |
| if…else 语句 | 一个 if 语句当执行的布尔表达式为假时，可以在后面伴随着一个可选的 else 语句 |
| switch 语句 | 一个 switch 语句允许一个变量针对多个不同的值分别进行验证是否满足条件。 |
| 嵌套的 if 语句 | 你可以使一个if或者 else if 语句嵌套在另一个 if 或 else if 语句中。 |
| 嵌套的 switch 语句 | 你可以一个 switch 语句中嵌套使用另一个 switch 语句中。 |

## ？：运算符

我们在前一章中有可以用来替换 if...else 语句的覆盖条件语句 `？：` 。它具有以下基本形式：

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

其中 Exp1 ， Exp2 和 Exp3 是表达式。注意冒号的使用和它的位置。

？ 表达式值的计算方式为：首先计算 Exp1 的值。如果 Exp1 为真，则计算 Exp2 的值作为整个 ？ 表达式的值，如果 Exp1 为假，则计算 Exp3 的值，并且将其作为这个表达式的值。

# 函数

函数是执行某个任务的一组语句。每个 C++ 程序至少具有一个功能，即 main()，所有最琐碎的程序可以定义额外的功能。

你可以将代码分成单独的功能。怎样在不同的功能之间分配你的代码取决于你，但从逻辑上讲，这个划分通常是每个函数执行一个特定的任务。

一个函数 **声明** 用来告诉编译器函数的名字，返回值类型，和参数。一个函数的 **定义** 提供了函数的实际函数体。

C++ 标准库提供了众多的可以被您的程序直接调用的内置功能。例如，用 strcat() 函数来连接两个字符串，用 memcpy() 函数可以将一个内存位置的内容复制到另一个内存位置，还有更多的函数。

我们知道一个函数有着各种各样的名字诸如一种方法或子程序或一个程序等。

## 定义一个函数

C++ 函数的定义的一般形式如下所示：

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

一个 C++ 函数的定义由一个函数头和一个函数体组成。以下是一个函数的所有部分：

- **返回值类型：** 一个函数可能返回一个值。 return-type 是该函数返回的值的数据类型。有些函数执行所需的操作不返回一个值，在这种情况下，return-type 是关键字 void.
- **函数名称：** 这是函数实际的名字。函数名称和参数列表一起构成了这个函数的签名。
- **参数：** 一个参数就像一个占位符。当调用参数时，你将把值传递给该参数。这个值称为实际参数或参数。参数列表是指一个函数的类型，顺序和参数的数目。参数是可选的，那就是指，一个函数可能不包含参数。
- **函数体：** 函数体包含定义函数做什么的一系列语句。

## 例子

下面是一个叫做 max() 的函数的源代码。这个函数包含 num1 和 num2 两个参数，并返回两者之间的最大值：

```
// function returning the max between two numbers

int max(int num1, int num2)
{
   // local variable declaration
   int result;

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

   return result;
}
```

## 函数声明

一个函数**声明**告诉编译器函数的名字和如何调用这个函数。函数的实际实体可以被单独定义。

函数声明有以下几个部分：

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

对于上述定义的 `max()` 函数，以下是这个函数的声明：

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

参数的名称在函数声明中并不重要，但是参数的类型是必须有的，所以以下也是有效的声明：

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

当你在一个源文件中定义了一个函数并且在另一个文件中调用了该函数时，函数的声明是必需的。在这种情况下，你应该在调用该函数的文件的顶部声明这个函数。

## 调用函数

当你创建一个 C++ 函数时，你给出这个函数应该做什么的一个定义。若要使用一个函数，你将需要调用或唤起这个函数。

当一个程序调用一个函数时，程序控制转到所调用的函数。被调用的函数执行定义的任务，当执行到返回语句或函数执行到该函数结尾的右大括号时，它将程序控制返回给主程序。

要调用一个函数，你只需要传递所需的参数以及函数名，如果函数返回一个值，然后你可以存储返回的值。举个例子：

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

// function declaration
int max(int num1, int num2);

int main ()
{
   // local variable declaration:
   int a = 100;
   int b = 200;
   int ret;

   // calling a function to get max value.
   ret = max(a, b);

   cout << "Max value is : " << ret << endl;

   return 0;
}

// function returning the max between two numbers
int max(int num1, int num2)
{
   // local variable declaration
   int result;

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

   return result;
}
```

我将 max() 函数跟在 main() 函数后面然后编译源代码。当运行最终可执行文件时，它将产生以下结果：

```
Max value is : 200
```

## 函数参数

如果一个函数要使用参数，它必须声明接受参数值的变量。这些变量被称为函数的**形参**。

像其他本地变量一样，这些形参在函数中，在进入函数时创建，在退出函数时销毁。

当调用函数时，这里有两种方式可以将参数传递给函数。

| 调用类型 | 描述 |
| --- | --- |
| 通过值调用 | 这个方法将参数的实际值复制到该函数的形参。在这种情况下，在函数中对该参数做出的更改对这个参数没有影响。 |
| 通过指针调用 | 这个方法将参数的地址复制到形参中。在函数的内部，这个地址用来在调用中访问这个实参。这意味着，对参数所做出的更改会影响参数。 |
| 通过引用调用 | 这个方法将参数的引用复制到形参中。在函数的内部，这个引用用来在调用中访问这个形参。这意味着，对参数所作出的更改会影响参数。 |

默认情况下， C++ 使用**通过值调用**来传递参数。一般情况下，这意味着，在函数内部的代码不能改变用来调用函数的参数的值，在上面提到的例子中，当调用 max() 函数时使用了。

## 参数的默认值

当你定义一个函数时，你可以为每个最后的参数指定一个默认值。当调用函数时，如果对 应的参数为空时，将使用此值。

这是在函数定义中通过使用赋值运算符和在函数定义中进行参数赋值完成的。如果当调用 该函数时，该参数的值没有被传递，则使用默认给定的值，但如果指定了一个值，这个默 认值被忽略，而使用传递的值。请考虑下面的示例：

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

int sum(int a, int b=20)
{
  int result;

  result = a + b;

  return (result);
}

int main ()
```

```
{
   // local variable declaration:
   int a = 100;
   int b = 200;
   int result;

   // calling a function to add the values.
   result = sum(a, b);
   cout << "Total value is :" << result << endl;

   // calling a function again as follows.
   result = sum(a);
   cout << "Total value is :" << result << endl;

   return 0;
}
```

当上面的代码被编译和执行后，它将产生以下结果：

```
Total value is :300
Total value is :120
```

# 数字

当我们使用数字时，通常我们使用原始数据类型，例如 int，short，long，float 和 double 等。数字数据类型，它们的可能值和取值范围在讨论 C++ 数据类型时已经解释了。

## 在 C++ 程序中定义数字

在之前的章节中，我们定义了多种数字类型。下面的例子定义了多种数字类型：

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

int main ()
{
   // number definition:
   short  s;
   inti;
   long   l;
   float  f;
   double d;

   // number assignments;
   s = 10;
   i = 1000;
   l = 1000000;
   f = 230.47;
   d = 30949.374;

   // number printing;
   cout << "short  s :" << s << endl;
   cout << "inti :" << i << endl;
   cout << "long   l :" << l << endl;
   cout << "float  f :" << f << endl;
   cout << "double d :" << d << endl;

   return 0;
}
```

当上述代码被编译执行时，它将产生下面的结果：

```
short  s :10
inti :1000
long   l :1000000
float  f :230.47
double d :30949.4
```

## C++ 中的数学运算

您除了可以创建各种各样的函数外，C++ 中还包括一些你可以使用的已经存在的函数。这些函数在标准的 C 和 C++ 库中都可以使用，被成为**内置**函数。这些函数都可以包含在您的程序中，然后使用。

C++ 具有一套丰富的数学运算，可以运用在各种各样的数字上。下表中列出了一些在 C++ 中可以使用的内置数学函数。

要利用这些函数，你需要包括 `<cmath>` 这个数学头文件。

| 序号 | 函数和功能 |
|---|---|
| 1 | double cos(double)；这个函数输入一个角度（作为一个双精度）并返回余弦值。 |
| 2 | double sin(double)；这个函数输入一个角度（作为一个双精度）并返回正弦值。 |
| 3 | double tan(double)；这个函数输入一个角度（作为一个双精度）并返回正切值。 |
| 4 | double log(double)；这个函数输入一个数字并返回该数字的自然对数。 |
| 5 | double pow(double,double)；第一个参数是你想要增长的数字，第二个参数是你想要将它增长的倍数。 |
| 6 | double hypot(double,double)；如果你向该函数传递一个直角三角形的两个边的长度，它将返回你的直角三角形的斜边长度。 |
| 7 | double sprt(double)；您向该函数传递一个参数，它将给你这个数字的平方根。 |
| 8 | int abs(int)；这个函数返回传递给该函数的整数的绝对值。 |
| 9 | double fabs(double)；这个函数返回传递给该函数的任何十进制数的绝对值。 |
| 10 | double floor(double)；查找小于或者等于输入参数的整数。 |

以下是一个用来说明少数数学运算的简单的示例：

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

int main ()
{
   // number definition:
   short  s = 10;
   inti = -1000;
   long   l = 100000;
```

```
   float  f = 230.47;
   double d = 200.374;

   // mathematical operations;
   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;
}
```

当上述代码被编译执行时，它将产生如下结果：

```
sign(d) :-0.634939
abs(i)  :1000
floor(d) :200
sqrt(f) :15.1812
pow( d, 2 ) :40149.7
```

## C++ 中的随机数字

在很多情况下，您会希望生成一个随机数字。这里有两个你需要知道的随机数字的生成函数。第一个是 rand()，这个函数只返回一个伪随机数。要解决这个问题的方式是，首先调用 srand() 函数。

以下是一个用来产生几个随机数字的简单例子。这个示例使用 time() 函数来获取在您的系统时间中的秒数来随机的生成 rand() 函数：

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

using namespace std;

int main ()
{
   int i,j;

   // set the seed
   srand( (unsigned)time( NULL ) );

   /* generate 10  random numbers. */
   for( i = 0; i < 10; i++ )
```

```
    {
   // generate actual random number
   j= rand();
   cout <<" Random Number : " << j << endl;
    }

    return 0;
}
```

当上述代码被编译执行时，它将产生下面的结果：

```
Random Number : 1748144778
Random Number : 630873888
Random Number : 2134540646
Random Number : 219404170
Random Number : 902129458
Random Number : 920445370
Random Number : 1319072661
Random Number : 257938873
Random Number : 1256201101
Random Number : 580322989
```

# 数组

C++ 提供了一种数据结构，**数组**，它存储了一个固定大小的相同类型的有序集合的元素。一个数组通常被用来存储大量的数据，但往往将数组看作是一个相同类型的变量的集合。

不是声明单独的变量，例如数字 0，数字 1,...,和数字 99，你可以声明一个数组变量，例如 numbers,并且使用 number[0]，number[1] 和 ...,number[99] 来表示单个变量。数组中的特定元素是通过索引访问的。

所有的数组组成连续的内存位置。最低地址对应于第一个元素，最高地址对应最后一个元素。

## 声明数组

要在 C++ 中声明一个数组，程序员需要指定元素的类型和一个数组需要的元素的数目，如下所示：

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

这就是所谓的一维数组。其中 **arraySize** 必须是一个大于零的整数常量，而且 **type** 可以是任何有效的 C++ 数据类型。例如，若要声称一个 double 类型的包含 10 个元素的 balance 数组，需要使用如下语句。

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

## 初始化数组

你可以一个接着一个或者是使用一个单独的语句来初始化一个 C++ 数组，如下所示：

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

大括号 { } 之间的值的大小不能比我们为数组声明时的方括号 [ ] 中的数目大。以下是一个指定数组单个元素的例子。

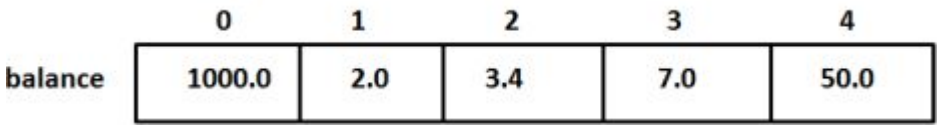
如果你省略了数组的大小，我们将创建一个足够容纳初始化的一个数组。如下代码：

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

就像之前的示例一样，你将创建一个完全相同的数组。

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

上面的语句将数组中的第五个元素定义为 50.0，第四个数组索引即第五个元素，即最后一个元素，因为所有的数组都使用 0 作为它们第一个元素的索引，称为基索引，下面是我们之前讨论的同一个数组的图案表示：




## 访问数组元素

一个元素通过索引数组名称被访问。这是通过将元素的索引放置在数组名字后面的方括号中来完成的，举例：

```
double salary = balance[9];
```

上面的语句将第 10 个元素从数组中取出来，并将该值赋给 salary 变量。以下是一个例子，即使用所有上述提到的三个概念，即声明，初始化和访问数组：

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

#include <iomanip>
using std::setw;

int main ()
{
   int n[ 10 ]; // n is an array of 10 integers

   // initialize elements of array n to 0
   for ( int i = 0; i < 10; i++ )
   {
         n[ i ] = i + 100; // set element at location i to i + 100
   }
   cout << "Element" << setw( 13 ) << "Value" << endl;

   // output each array element's value
   for ( int j = 0; j < 10; j++ )
   {
          cout << setw( 7 )<< j << setw( 13 ) << n[ j ] << endl;
   }

   return 0;
}
```

这个程序使用 setw() 函数来格式化输出。当上述代码编译执行时，它将生成以下结果：

```
ElementValue
  0  100
```

```
1  101
2  102
3  103
4  104
5  105
6  106
7  107
8  108
9  109
```

## 详细的 C++ 数组

数组对 C++ 非常重要,而且应该需要大量的详细信息。这里有以下几个 C++ 程序员应该清楚的重要的概念。

| 概念 | 描述 |
| --- | --- |
| 多维数组 | C++ 支持多维数组。多维数组最简单的形式是二维数组。 |
| 指向数组的指针 | 你可以通过简单的指定数组的名字来生成这个数组的第一个元素的指针，并不需要任何索引的。 |
| 将数组传递给函数 | 你可以不使用索引直接通过指定数组的名字将一个指向数组的指针传递给函数。 |
| 从函数中返回数组 | C++ 允许函数返回一个数组。 |

# 字符串

C++ 提供了以下两种类型的字符串表示形式：

- C 样式字符串
- 用标准 C++ 介绍的标准字符串类型

## C 样式字符串

C 样式字符串源于 C 语言，在 C++ 中仍然被支持。这个串实际是一个字符的一维数组，这个数组以一个空字符 '\0' 结束。因此以 null 结尾的字符串包含由字符组成的字符串，此字符串后跟着一个 null。

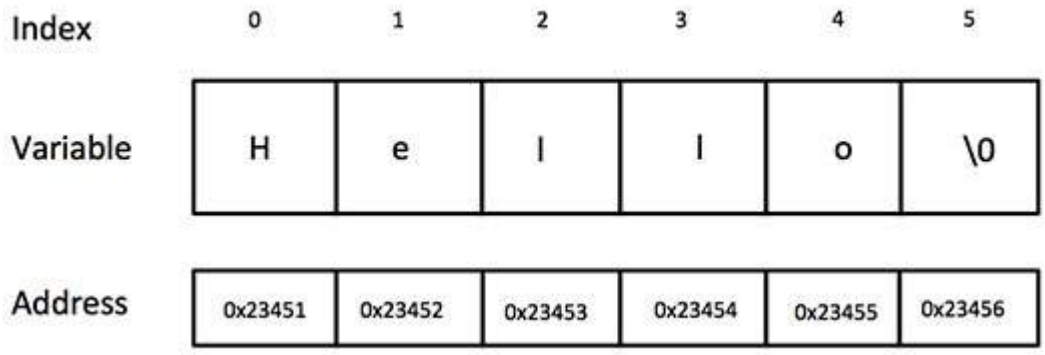
接下来声明和初始化创建一个字符串，这个字符串组成一个单词 "hello"。为了包含数组末尾的空字符，包含该字符串的字符数组的大小应该比单词 "hello" 中的字符的数目多一个。

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

如果你遵循数组初始化的规则，你可以像下面一样书写上面的语句：

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

以下是在 C/C++ 中定义上面的字符串的内存演示：




实际上，你不需要在字符串常量的末尾放置一个空字符。 C++ 编译器在初始化数组时自动在串的末尾放置一个 '\0'。让我们尝试打印上述字符串：

```
#include <iostream>

using namespace std;
```

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

   cout << "Greeting message: ";
   cout << greeting << endl;

   return 0;
}
```

当上述代码被编译执行时，它将产生如下结果：

```
Greeting message: Hello
```

C++ 支持广泛的函数来操作以空字符终止的字符串：

| SN | 函数和功能 |
|---|---|
| 1 | strcpy(s1,s2)；将字符串 s2 复制到字符串 s1 中。 |
| 2 | strcat(s1,s2)；将字符串 s2 串联到字符串 s1 的结尾。 |
| 3 | strlen(s1)；返回字符串 s1 的长度。 |
| 4 | strcmp(s1,s2)；如果 s1 和 s2 相同，返回 0；如果 s1≥s2，返回大于 0 的数；如果 s1 |
| 5 | strchr(s1,ch)；返回在字符串 s1 中指向第一次出现字符 ch 的指针。 |
| 6 | strstr(s1,s2)；返回在字符串 s1 中指向第一次出现字符串 s2 的指针。 |

以下是应用了一些上述函数的示例：

```
#include <iostream>
#include <cstring>

using namespace std;

int main ()
{
   char str1[10] = "Hello";
   char str2[10] = "World";
   char str3[10];
   int  len ;

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

   // concatenates str1 and str2
   strcat( str1, str2);
   cout << "strcat( str1, str2): " << str1 << endl;
```

```
   // total lenghth of str1 after concatenation
   len = strlen(str1);
   cout << "strlen(str1) : " << len << endl;

   return 0;
}
```

当上述代码被编译执行时，它将产生如下结果：

```
strcpy( str3, str1) : Hello
strcat( str1, str2): HelloWorld
strlen(str1) : 10
```

## C++ 中的字符串类

标准的 C++ 库中提供了一个 string 类，它支持上面提到的所有的操作，另外它还有更多的功能。我们会研究 C++ 标准库中的这个类，但现在我们先检查以下示例：

在这点上，你可能还没有明白这个例子，因为到目前为止我们还没有讨论类和对象。所以直到你理解了面向对象的概念你才可以继续进行下去。

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

using namespace std;

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

   // copy str1 into str3
   str3 = str1;
   cout << "str3 : " << str3 << endl;

   // concatenates str1 and str2
   str3 = str1 + str2;
   cout << "str1 + str2 : " << str3 << endl;

   // total lenghth of str3 after concatenation
   len = str3.size();
```

```
    cout << "str3.size() :  " << len << endl;

    return 0;
}
```

当上述代码被编译执行时，它将产生如下结果：

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

# 指针

C++ 指针学起来非常容易和有趣。一些 C++ 的任务用指针执行非常容易，诸如动态分配内存的 C++ 工作，如果没有指针将无法执行。

如你所知，每个变量有一个内存位置，每个内存位置都有它的地址定义，这个地址定义可以使用表示内存中地址的和运算符 (&) 进行访问。下面我们将打印定义的变量的地址：

```
#include <iostream>

using namespace std;

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

   cout << "Address of var1 variable: ";
   cout << &var1 << endl;

   cout << "Address of var2 variable: ";
   cout << &var2 << endl;

   return 0;
}
```

当上述代码被编译执行时，它将产生如下结果:

```
Address of var1 variable: 0xbfebd5c0
Address of var2 variable: 0xbfebd5b6
```

## 指针是什么？

**指针**是一个变量，它的值是另一个变量的地址。像任何变量或常量一样，你必须在使用它之前声明一个指针。

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++ 类型，var-name 是指针变量的名称。用来声明一个指针的星号与你用于乘法的星号是一样的。然而，在这个语句中，这个星号用来指定一个变量作为一个指针。以下是有效的指针声明：

```
int*ip;// pointer to an integer
double *dp;// pointer to a double
float  *fp;// pointer to a float
char   *ch // pointer to character
```

所有指针的值的实际的数据类型，或者是整数，浮点数，字符或者是其他，同样的一个长十六进制数表示一个内存地址。不同的数据类型的指针之间的唯一区别在于指针指向的变量或常量的数据类型。

## 在 C++ 中使用指针

这里有几个我们将非常频繁的使用指针的重要的操作。

(a) 我们定义一个指针变量；

(b) 将一个变量的地址分配给指针；

(c) 最后使用在指针变量中的地址来访问这个值。

这是通过使用一元运算符 * 完成的，返回位于通过运算符指定地址的变量的值。以下示例使用这些操作：

```
#include <iostream>

using namespace std;

int main ()
{
   int  var = 20;   // actual variable declaration.
   int  *ip;// pointer variable

   ip = &var;   // store address of var in pointer variable

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

   // print the address stored in ip pointer variable
   cout << "Address stored in ip variable: ";
   cout << ip << endl;

   // access the value at the address available in pointer
```

```
   cout << "Value of *ip variable: ";
   cout << *ip << endl;

   return 0;
}
```

当上述代码被编译执行时，它将产生如下结果：

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

## 详细的 C++ 指针

指针有很多但是很容易的概念，它们对 C++ 的编程非常重要。这里有一些 C++ 程序员应该必须清楚的重要的指针概念：

| 概念 | 描述 |
| --- | --- |
| C++ 空指针 | C++ 支持空指针，它是在几个标准库中被定义值为零的一个常量。 |
| C++ 指针算法 | 这里有四种可用于指针的算术运算符：++,--,+,- |
| C++ 指针与数组 | 指针与数组之间有着密切的关系。让我们看一看？ |
| C++ 指针数组 | 你可以定义保存大量指针的数组。 |
| C++ 指针的指针 | C++ 允许您有指针的指针，等等。 |
| 将指针传递给函数 | 通过引用或通过地址传递参数，两种方法都可以使被调用的函数在调用函数中传递的参数发生更改。 |
| 从函数返回指针 | C++ 允许函数返回一个指向本地变量的指针，同样也允许返回指向静态变量和动态分配内存的指针。 |

# 引用

引用变量是一个别名，即已经存在的变量的另一个名称（引用就是某一个变量的别名）。一旦用一个变量初始化引用，变量名称和引用名称都可以用来指示变量。

## C++ 引用 VS 指针

引用与指针非常容易混淆，但引用和指针有三个主要区别：

- 空引用不可能存在。你必须始终能够假定一个引用被连接到一个合法的存储块。
- 一旦一个引用被初始化为一个对象，它就不能改变去指示另一个对象。指针可以随时改变指向另一个不同的对象。
- 引用必须在它被创建时就初始化。指针可以在任何时候初始化。

## 在 c++ 中创建引用

考虑到一个变量名是一个附加到该变量在内存中的位置的标签。你可以认为一个引用是附加到该内存位置的第二个标签。因此,您可以通过原始变量名或引用来访问变量的内容。例如，我们假设有下面的例子：

```
int i = 17;
```

我们可以为 i 声明引用变量，如下所示。

```
int& r = i;
```

在这些声明中将 & 理解为引用（reference）。因此，第一个声明理解为 “ r 是一个整数引用，初始化为 i ”和第二声明理解为 “ s 是一个双引用，初始化为 d ”。下面的例子使用了 int 和 double 引用：

```
#include <iostream>

using namespace std;

int main ()
{
   // declare simple variables
   inti;
   double d;
```

```
    // declare reference variables
    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;
}
```

将上面的代码放在一起编译、执行，执行结果如下 ：

```
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。以下是与 c++ 引用有关的两个重要的方面，一个 c++ 程序员应该明确了解：

| 内容 | 描述 |
| --- | --- |
| 引用作为参数 | c++ 支持引用作为函数参数传递，它比直接传递参数更安全。 |
| 引用作为返回值 | 可以从一个 c++ 函数返回引用，就像返回任何其他数据类型。 |

# 日期和时间

C++ 标准库没有提供一个合适的日期类型。C++ 从 C 中继承了针对日期和时间的结构和功能，为了访问与日期和时间相关的功能和结构，需要在 C++ 程序中包括 <ctime> 头文件。

这里有四个与时间相关的类型：**clock_t、time_t、size_t** 和 **tm**。clock_t，size_t 和 time_t 类型能够以某种类型的整数表示系统时间和日期。

结构类型 **tm** 以 C 结构体的形式支持日期和时间，有以下元素：

```
struct tm {
  int tm_sec;   // seconds of minutes from 0 to 61
  int tm_min;   // minutes of hour from 0 to 59
  int tm_hour;  // hours of day from 0 to 24
  int tm_mday;  // day of month from 1 to 31
  int tm_mon;   // month of year from 0 to 11
  int tm_year;  // year since 1900
  int tm_wday;  // days since sunday
  int tm_yday;  // days since January 1st
  int tm_isdst; // hours of daylight savings time
}
```

以下是我们在 C 或 C++ 中处理日期和时间时使用的一些重要的函数。所有这些函数都是标准 C 和 C++ 库的一部分，你可以使用下面给出的 C++ 标准库引用查看它们的使用细节。

| 序号 | 功能与目的 |
|---|---|
| 1 | **time_t time(time_t *time);**<br>这将返回当前系统的日历时间，以自 1970 年 1 月 1 日开始系统运行秒数的形式。如果系统没有时间，返回 1。 |
| 2 | **char *ctime(const time_t *time);**<br>这返回一个指向字符串的指针，字符串形式为 *day month year hours:minutes:seconds year\n\0*。 |
| 3 | **struct tm *localtime(const time_t *time);**<br>这将返回一个指向 tm 结构体的指针，tm 结构体代表当地时间。 |
| 4 | **clock_t clock(void);**<br>这将返回一个与被调用程序运行时间的总和接近的值。如果时间无效，返回 1。 |
| 5 | **char * asctime ( const struct tm * time );**<br>这将返回一个指向字符串的指针，该字符串包含的信息以如下结构体存储，结构体形式如下:*day month year hours:minutes:seconds year\n\0* |
| 6 | **struct tm *gmtime(const time_t *time);**<br>它返回一个指向时间的指针，该时间是 tm 结构的。时间用协调世界时(UTC)表示，在本质上是格林威治标准时间(GMT)。 |

| | |
|---|---|
| 7 | time_t mktime(struct tm *time);<br>返回日历时间，时间以参数中指出的结构形式表示。 |
| 8 | double difftime ( time_t time2, time_t time1 );<br>这个函数计算秒 time1 和 time2 之间的差异。 |
| 9 | size_t strftime();<br>这个函数可以用于以一种特定格式来格式化日期和时间。 |

## 当前的日期和时间

考虑你想要取得当前系统的日期和时间，作为当地时间或作为一个协调世界时（UTC）。下面是一个实现相同目的的示例：

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

using namespace std;

int main( )
{
   // current date/time based on current system
   time_t now = time(0);

   // convert now to string form
   char* dt = ctime(&now);

   cout << "The local date and time is: " << dt << endl;

   // convert now to tm struct for UTC
   tm *gmtm = gmtime(&now);
   dt = asctime(gmtm);
   cout << "The UTC date and time is:"<< dt << endl;
}
```

编译和执行上面的代码，执行结果如下：

```
The local date and time is: Sat Jan  8 20:07:41 2011

The UTC date and time is:Sun Jan  9 03:07:41 2011
```

## 使用结构体 tm 格式化时间：

无论是在 C 还是在 C++ 中，tm 结构体在处理日期和时间时都是非常重要的。如前所述，该结构以一种 C 语言结构体的形式支持日期和时间。大部分与时间相关的函数使用 tm 结构。下面是一个例子，它使用了各种各样日期和时间的相关函数和 tm 结构：

在本章中使用结构体，基于一个假设：你已经对 C 语言的结构体有了基本的了解，并且知道如何使用箭头操作符：-> 访问结构体成员。

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

using namespace std;

int main( )
{
   // current date/time based on current system
   time_t now = time(0);

   cout << "Number of sec since January 1,1970:" << now << endl;

   tm *ltm = localtime(&now);

   // print various components of tm structure.
   cout << "Year: "<< 1900 + ltm->tm_year << endl;
   cout << "Month: "<< 1 + ltm->tm_mon<< endl;
   cout << "Day: "<<  ltm->tm_mday << endl;
   cout << "Time: "<< 1 + ltm->tm_hour << ":";
   cout << 1 + ltm->tm_min << ":";
   cout << 1 + ltm->tm_sec << endl;
}
```

编译和执行上面的代码，执行结果如下：

```
Number of sec since January 1, 1970:1294548238
Year: 2011
Month: 1
Day: 8
Time: 22: 44:59
```

# 基本输入输出

C++ 标准库提供了一组广泛的的输入/输出功能，我们将在随后的章节中展开。本章将讨论 C++ 编程所需的最基本和最常见的 I/O 操作。

C++ I/O发生在流中，流是一种字节序列。如果字节流从一个设备（如键盘、磁盘驱动器,或网络连接等）到主内存，这称为输入操作（input operation）。如果字节从主内存流向一个设备（如显示屏，打印机，磁盘驱动器，或网络连接等），这就是所谓的输出操作（output operation）。

## I/O 库头文件

下边列出对于 C++ 程序重要的头文件：

| 头文件 | 功能和描述 |
| --- | --- |
| <iostream> | 这个头文件定义了 cin、cout、cerr 和 clog 对象，它们分别对应于标准输入流，标准输出流，无缓冲标准错误流和有缓冲标准错误流。 |
| <iomanip> | 这个头文件声明了用于执行格式化 I/O 操作的一系列服务，即所谓的参数化流操作，如 setw 和 set precision。 |
| <fstream> | 这个头文件声明了基于用户控制的文件处理服务。我们将在文件和流相关的章节更详细讨论关于它的内容。 |

## 标准输出流

预定义的对象 cout 是 ostream 类的一个实例。cout 对象通常连接到标准输出设备，一般是显示屏。cout 和流插入运算符联合使用，流插入运算符写为 `<<`，即两个表示小于的符号，如以下示例所示。

```
#include <iostream>

using namespace std;

int main( )
{
   char str[] = "Hello C++";

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

编译和执行上面的代码，运行结果如下：

```
Value of str is : Hello C++
```

C++ 编译器也决定了输出的变量数据类型并选择适当的流插入运算符来显示值。`<<` 操作符重载了多种输出数据项（包括各种内置类型：整数、浮点数、双精度浮点数、字符串和指针类型的值）。

插入运算符`<<`在一个语句中可能不止一次被使用，如上所示，endl 写在结束的时候用于添加一个新行。

## 标准输入流

预定义对象 cin 是 istream 类的一个实例。cin 对象通常用于标准输入设备，一般是键盘。cin 和流提取运算符联合使用，流提取符号写为`>>`即两个表示大于的符号，如以下示例所示。

```
#include <iostream>

using namespace std;

int main( )
{
   char name[50];

   cout << "Please enter your name: ";
   cin >> name;
   cout << "Your name is: " << name << endl;

}
```

编译和执行上面的代码，它会提示输入一个名称。输入一个值，然后回车，运行结果如下：

```
Please enter your name: cplusplus
Your name is: cplusplus
```

C++ 编译器也决定输入值的数据类型和选择适当的流提取运算符提取值并将其存储在给定的变量。

流提取操作符`>>`可以在一个声明中不止一次使用。请求多个数据时，可采用以下书写形式：

```
cin >> name >> age;
```

上述表达式等价于下面两个声明语句：

```
cin >> name;
cin >> age;
```

## 标准错误流

预定义对象 cerr 是 ostream 类的一个实例。cerr 对象通常附加到标准错误设备，这一般也是一个显示屏，但对象 cerr 是无缓存的，每当有流插入到 cerr 会导致其输出立即出现。

cerr 也与流插入操作符一起使用，如以下示例所示。

```
#include <iostream>

using namespace std;

int main( )
{
   char str[] = "Unable to read....";

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

编译和执行上面的代码，运行结果如下：

```
Error message : Unable to read....
```

## 标准日志流

预定义对象 clog 是 ostream 类的一个实例。clog 对象通常附加到标准错误设备，这一般也是一个显示屏，但对象 clog 是有缓冲的。这意味着每个插入的 clog 会暂存在缓冲区中，直到缓冲区满或者缓冲区刷新才会产生一次输出。

clog 也与流插入操作符一起使用，如以下示例所示。

```
#include <iostream>

using namespace std;

int main( )
{
   char str[] = "Unable to read....";

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

编译和执行上面的代码，运行结果如下：

```
    Error message : Unable to read....
```

在这些小例子中，你可能无法看出 cout,cerr,clog 的区别，但在编写、执行大项目时，差异就变得显而易见了。所以这是很好的应用实践：使用 cerr 流显示错误消息，而使用 clog 显示其他日志信息。

# 结构体

C/C++ 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，叫做结构，但结构体（structure）是一种用户定义的数据类型，允许你将不同类型的数据项放在一起。

结构用来表示一条记录。假设你想在图书馆中找一本书，您可能需要查找每本书的以下属性：

- Title
- Author
- Subject
- Book ID

## 定义一个结构体

定义一个结构体，您必须使用结构体声明。结构体语句为您的程序定义了一个新的数据类型，拥有一个以上的成员。

结构体声明的格式是这样的:

```
struct [structure tag]
{
   member definition;
   member definition;
   ...
   member definition;
} [one or more structure variables];
```

structure tag 是可选的，每个成员的定义都是一个正常的变量定义，如 int i；或者 float f；或者任何其他有效的变量定义。在结构的定义结束时，在结构体定义结尾处的（“；”）符号之前可以指定一个或多个结构变量，但它是可选的。这是声明书结构体的方式:

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

## 访问结构体成员

访问一个结构体的任何成员，我们使用 member access operator（成员访问操作符）：(.) 来访问结构体成员。成员访问操作符编码为结构变量名和我们要访问结构成员之间的一个点符号。使用关键字 struct 来定义结构类型的变量。下面是例子解释怎样使用结构体：

```
#include <iostream>
#include <cstring>

using namespace std;

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

int main( )
{
   struct Books Book1;// Declare Book1 of type Book
   struct Books Book2;// Declare Book2 of type Book

   // book 1 specification
   strcpy( Book1.title, "Learn C++ Programming");
   strcpy( Book1.author, "Chand Miyan");
   strcpy( Book1.subject, "C++ Programming");
   Book1.book_id = 6495407;

   // book 2 specification
   strcpy( Book2.title, "Telecom Billing");
   strcpy( Book2.author, "Yakit Singha");
   strcpy( Book2.subject, "Telecom");
   Book2.book_id = 6495700;

   // Print Book1 info
   cout << "Book 1 title : " << Book1.title <<endl;
   cout << "Book 1 author : " << Book1.author <<endl;
   cout << "Book 1 subject : " << Book1.subject <<endl;
   cout << "Book 1 id : " << Book1.book_id <<endl;

   // Print Book2 info
```

```
   cout << "Book 2 title : " << Book2.title <<endl;
   cout << "Book 2 author : " << Book2.author <<endl;
   cout << "Book 2 subject : " << Book2.subject <<endl;
   cout << "Book 2 id : " << Book2.book_id <<endl;

   return 0;
}
```

编译和执行上面的代码，执行结果如下：

```
Book 1 title : Learn C++ Programming
Book 1 author : Chand Miyan
Book 1 subject : C++ Programming
Book 1 id : 6495407
Book 2 title : Telecom Billing
Book 2 author : Yakit Singha
Book 2 subject : Telecom
Book 2 id : 6495700
```

## 结构体作为函数参数

你可以将结构体作为函数参数传递，其使用方式和将其他任何变量或指针作为参数传递非常相似。你可以以同样的方式访问结构变量，就如在上面的例子中显示的一样：

```
#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( )
{
   struct Books Book1;// Declare Book1 of type Book
   struct Books Book2;// Declare Book2 of type Book

   // book 1 specification
```

```
   strcpy( Book1.title, "Learn C++ Programming");
   strcpy( Book1.author, "Chand Miyan");
   strcpy( Book1.subject, "C++ Programming");
   Book1.book_id = 6495407;

   // book 2 specification
   strcpy( Book2.title, "Telecom Billing");
   strcpy( Book2.author, "Yakit Singha");
   strcpy( Book2.subject, "Telecom");
   Book2.book_id = 6495700;

   // Print Book1 info
   printBook( Book1 );

   // Print Book2 info
   printBook( Book2 );

   return 0;
}
void printBook( struct Books book )
{
   cout << "Book title : " << book.title <<endl;
   cout << "Book author : " << book.author <<endl;
   cout << "Book subject : " << book.subject <<endl;
   cout << "Book id : " << book.book_id <<endl;
}
```

编译和执行上面的代码，执行结果如下：

```
Book title : Learn C++ Programming
Book author : Chand Miyan
Book subject : C++ Programming
Book id : 6495407
Book title : Telecom Billing
Book author : Yakit Singha
Book subject : Telecom
Book id : 6495700
```

## 结构体指针

您可以定义结构体指针，以一种定义指向其他变量的指针非常相似的方式，如下所示：

```
struct Books *struct_pointer;
```

现在，您可以用上面定义的指针变量存储一个结构变量的地址。找到一个结构变量的地址，把操作符 & 置于结构体名称的前面，如下所示：

```
struct_pointer = &Book1;
```

为了通过一个指向结构的指针访问结构体成员，必须使用 -> 操作符，如下所示：

```
struct_pointer->title;
```

让我们使用结构指针重写上面的例子，希望这将帮助你更容易理解这个概念：

```
#include <iostream>
#include <cstring>

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

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

int main( )
{
   struct Books Book1;// Declare Book1 of type Book
   struct Books Book2;// Declare Book2 of type Book

   // Book 1 specification
   strcpy( Book1.title, "Learn C++ Programming");
   strcpy( Book1.author, "Chand Miyan");
   strcpy( Book1.subject, "C++ Programming");
   Book1.book_id = 6495407;

   // Book 2 specification
   strcpy( Book2.title, "Telecom Billing");
   strcpy( Book2.author, "Yakit Singha");
   strcpy( Book2.subject, "Telecom");
   Book2.book_id = 6495700;

   // Print Book1 info, passing address of structure
   printBook( &Book1 );
```

```
   // Print Book1 info, passing address of structure
   printBook( &Book2 );

   return 0;
}
// This function accept pointer to structure as parameter.
void printBook( struct Books *book )
{
   cout << "Book title : " << book->title <<endl;
   cout << "Book author : " << book->author <<endl;
   cout << "Book subject : " << book->subject <<endl;
   cout << "Book id : " << book->book_id <<endl;
}
```

编译和执行上面的代码，执行结果如下：

```
Book title : Learn C++ Programming
Book author : Chand Miyan
Book subject : C++ Programming
Book id : 6495407
Book title : Telecom Billing
Book author : Yakit Singha
Book subject : Telecom
Book id : 6495700
```

## typedef 关键字

有一个更简单的方法来定义结构体，你可以给你创建的类型起一个别名，例如：

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

现在，可以直接使用 Books 来定义书籍类型的变量，而不使用 struct 关键字。下面是示例：

```
Books Book1, Book2;
```

你也可以在非结构体（non-structs）中使用 typedef 关键字，如下所示：

```
typedef long int *pint32;

pint32 x, y, z;
```

`x,y,z` 是指向长整数类型的指针。

2

# C++ 面向对象

# 类和对象

C++ 编程的主要目的是将面向对象的思想引进到 C 编程语言中，类是 C++ 的核心特征，用来支持面向对象编程，类通常被称为用户定义的类型。

类是用于指定一个对象的形式，它将数据表示和用于处理数据的方法组合成一个整洁的包。一个类的数据和函数统称为类的成员。

## C++ 类的定义

当你定义了一个类，你就定义一个数据类型的蓝图。这实际上没有定义任何数据，它只是定义了类名是什么意思，也就是意味着，一个类的对象包含什么，在这样一个对象上可以执行哪些操作。

定义一个类，以关键字 class 开始，紧随其后的是类名，和类的主体，类的主体由一对大括号封闭。一个类定义必须以分号或者一系类声明结尾。例如，我们通过使用关键字 class 定义 Box 数据类型，如下所示：

```
class Box
{
   public:
  double length;   // Length of a box
  double breadth;  // Breadth of a box
  double height;   // Height of a box
};
```

关键字 public 决定了紧随其后的类的成员的访问属性。一个公共成员可以从类外处于任何一个类对象范围内的地方访问。类对象的范围内的任何地方。您还可以指定一个类的成员为 private 或 public，我们将在一个小节中讨论它们。

## 定义 C++ 对象

一个类为对象提供了蓝图，对象是由类创建而来。我们声明一个类的对象的方式，用声明其他基本类型变量的方式完全相同。以下语句声明 Box 类的两个对象：

```
Box Box1;  // Declare Box1 of type Box
Box Box2;  // Declare Box2 of type Box
```

Box1 和 Box2 对象都分别持有其各自的数据副本。

## 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操作符 `：(.)` 访问。试着执行下面的例子，能更清晰的说明这个问题：

```
#include <iostream>

using namespace std;

class Box
{
   public:
  double length;   // Length of a box
  double breadth;  // Breadth of a box
  double height;   // Height of a box
};

int main( )
{
   Box Box1;// Declare Box1 of type Box
   Box Box2;// Declare Box2 of type Box
   double volume = 0.0; // Store the volume of a box here

   // box 1 specification
   Box1.height = 5.0;
   Box1.length = 6.0;
   Box1.breadth = 7.0;

   // box 2 specification
   Box2.height = 10.0;
   Box2.length = 12.0;
   Box2.breadth = 13.0;
   // volume of box 1
   volume = Box1.height * Box1.length * Box1.breadth;
   cout << "Volume of Box1 : " << volume <<endl;

   // volume of box 2
   volume = Box2.height * Box2.length * Box2.breadth;
   cout << "Volume of Box2 : " << volume <<endl;
   return 0;
}
```

编译和执行上面的代码，执行结果如下：

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

需要特别加以注意的是，不能使用直接成员访问操作符 `：(.)` 直接访问私有成员和保护成员。我们将在以后学习如何访问私有成员和保护成员。

## 类与对象的细节

到目前为止，我们已经对 C++ 类和对象有了最基本的了解。还还有更多的与 C++ 类和对象相关的有趣的概念，我们将在下面列出的各个小节中讨论它们：

| 内容 | 描述 |
| --- | --- |
| 类成员函数 | 类的成员函数是一个函数，像其他变量一样，成员函数在类中有其定义和原型。 |
| 类的访问修饰符 | 一个类成员可以被定义为公共，私有或保护。默认情况下成员将被假定为私有。 |
| 构造函数和析构函数 | 一个类的构造函数是一种特殊的函数，在创建一个类的新对象时调用它。析构函数也是一个特殊的函数，当创建对象被删除时调用它。 |
| C++ 拷贝构造函数 | 拷贝构造函数是一个构造函数，它创建一个对象并用之前已经创建好的一个同类的对象对其进行初始化。 |
| C++ 友函数 | 一个友（friend）函数允许完全访问类的私有成员和保护成员。 |
| C++ 内联函数 | 使用一个内联函数，编译器试图用函数体中的代码替换调用函数的地方的函数名，从而达到消除函数调用时的时间开销的目的。 |
| C++ 中的 this 指针 | 每个对象都有一个特殊的指针 this，它指向对象本身。 |
| 指向 C++ 类的指针 | 类指针和一个指向结构的指针是以完全相同的方式实现的。事实上一个类就是一个在其中包含了函数的结构体。 |
| 类的静态成员 | 类的数据成员和函数成员都可以被声明为静态的。 |

# 继承

在面向对象编程中最重要的概念之一就是继承。继承允许我们根据一个类来定义另一个类，这使得创建和维护一个应用程序更加的容易。这也提供了一个重用代码功能和快速实现的机会。

当创建一个类，不是写全新的数据成员和成员函数的时候，程序员可以指定新类，这个类可以继承现有类的成员。这个现有的类称为基类，这个新类称为派生类。

继承的概念其实是一种关系。例如，哺乳动物是动物，狗是哺乳动物，因此狗是动物等等。

## 基类和派生类

一个类可以继承多个类，这就意味着它可以从多个基类中继承数据和函数。为了定义一个派生类，我们可以使用一个类继承列表来指定基类。一个类继承列表指定一个或多个基类，类继承列表形式如下：

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

在这里 access-specifier 是 public 、 protected 或者 private ，base-class 是之前定义的类的名称。如果不使用 access-specifier ，那么在默认情况下它是私有的。

考虑一个基类的 shape 和其派生类 Rectangle 的继承情况如下：

```
#include <iostream>

using namespace std;

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

```
};

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

int main(void)
{
   Rectangle Rect;

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

   // Print the area of the object.
   cout << "Total area: " << Rect.getArea() << endl;

   return 0;
}
```

上面的代码编译和执行时，它产生以下结果：

```
Total area: 35
```

## 访问控制和继承

一个派生类可以访问所有它的基类的非公有类型的成员。因此不希望被派生类的成员函数访问的基类成员应该在基类中声明为私有类型。

我们可以根据谁能访问它们总结出不同的访问类型，如下表格中所示：

| 访问权限 | public | protected | private |
| --- | --- | --- | --- |
| 同一个类 | 是 | 是 | 是 |
| 派生类 | 是 | 是 | 否 |
| 类外成员 | 是 | 否 | 否 |

派生类继承了基类的所有方法，以下情况除外：

- 基类的构造函数、析构函数和拷贝构造函数。

- 基类的重载操作符。
- 基类的友元函数。

## 继承方式

当从一个基类派生一个子类的时候,公共基类可以通过 public ，protected ，或者 private 方式被继承。继承方式被 access-specifier 指定，正如上面解释的。

我们几乎不使用protected或私有private 继承，但public继承是常用的。在使用不同类型的继承的时候,应用规则如下：

- public 继承：当从一个公有基类派生一个类的时候，基类的公有成员成为派生类的公有成员；基类的保护成员成为派生类的保护成员。一个基类的私有成员不能被派生类直接访问，但可以通过调用基类的公有和保护成员访问基类的私有成员。
- protected 继承：当从一个受保护的基类派生子类的时候,基类的公有和保护成员成为派生类的保护成员。
- private 继承：当从一个私有的基类派生子类的时候,基类的公有和保护成员成为派生类的私有成员。

## 多继承

一个C++类可以继承多个类的成员，多继承语法如下：

```
class derived-class: access baseA, access baseB....
```

在这里 access 是 public ，protected ，或者是 private ，并且每一个基类将有一个 access 类型，他们将由逗号分隔开，如上所示。让我们试试下面的例子：

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

// Base class Shape
class Shape
{
   public:
  void setWidth(int w)
  {
 width = w;
  }
  void setHeight(int h)
  {
```

```
 height = h;
  }
   protected:
  int width;
  int height;
};

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

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

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

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

   area = Rect.getArea();

   // Print the area of the object.
   cout << "Total area: " << Rect.getArea() << endl;

   // Print the total cost of painting
   cout << "Total paint cost: $" << Rect.getCost(area) << endl;

   return 0;
}
```

上面的代码编译和执行时，它产生以下结果：

```
Total area: 35
Total paint cost: $2450
```

# 重载

C++ 允许在同一范围内对一个函数名或一个操作符指定多个定义，分别被称为函数重载和操作符重载。

重载声明是在同一的范围内对先前已经声明的相同函数名的声明，除非这两个声明有不同的参数和明显不同的定义（实现方式）。

当你调用一个重载的函数或操作符时，编译器通过比较用来调用函数或操作符的指定的参数类型来确定使用最合适的定义。选择最合适的重载函数或操作符的过程被称为重载决议。

## C++ 中的函数重载

你可以在同一范围内对同一函数名有多个定义。函数的定义必须满足参数类型不同或参数的数量不同或两者都不相同。你不能重载只有返回类型不同的函数声明。

下面是一个相同的函数 print() 函数被用来打印不同的数据类型的例子：

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

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

  void print(double  f) {
cout << "Printing float: " << f << endl;
  }

  void print(char* c) {
cout << "Printing character: " << c << endl;
  }
};

int main(void)
{
   printData pd;

   // Call print to print integer
```

```
   pd.print(5);
   // Call print to print float
   pd.print(500.263);
   // Call print to print character
   pd.print("Hello C++");

   return 0;
}
```

上面的代码编译和执行时，它产生以下结果：

```
Printing int: 5
Printing float: 500.263
Printing character: Hello C++
```

## C++ 中的运算符重载

你可以重新定义或重载的大部分 C++ 已有的操作符。因此，程序员可以像使用用户自定义类型一样使用操作符。

重载操作符是一类函数，它们就是对已有的运算符重新进行定义，赋予其另一种功能，以适应不同的数据类型。像任何其它函数,重载运算符也有返回类型和参数列表。

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

声明加法运算符可以用来使两个 Box 对象相加并返回最终 Box 对象。大多数重载运算符可以被定义为普通非成员函数或类成员函数。如果我们把上面的函数定义为一个类的非成员函数,那么我们就必须为每个操作数传两个参数如下：

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

下面是通过使用成员函数来展示运算符重载的概念的示例。这里一个对象作为一个参数被传递，通过访问这个对象可以获得参数的属性，将调用这个操作符的对象可以通过使用 this 操作符获得，下面这个例子展示了这一点：

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

class Box
{
   public:
```

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

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

  void setHeight( double hei )
  {
  height = hei;
  }
  // Overload + operator to add two Box objects.
  Box operator+(const Box& b)
  {
 Box box;
 box.length = this->length + b.length;
 box.breadth = this->breadth + b.breadth;
 box.height = this->height + b.height;
 return box;
  }
   private:
  double length;  // Length of a box
  double breadth; // Breadth of a box
  double height;  // Height of a box
};
// Main function for the program
int main( )
{
   Box Box1;// Declare Box1 of type Box
   Box Box2;// Declare Box2 of type Box
   Box Box3;// Declare Box3 of type Box
   double volume = 0.0; // Store the volume of a box here

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

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

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

    // volume of box 2
    volume = Box2.getVolume();
    cout << "Volume of Box2 : " << volume <<endl;

    // Add two object as follows:
    Box3 = Box1 + Box2;

    // volume of box 3
    volume = Box3.getVolume();
    cout << "Volume of Box3 : " << volume <<endl;

    return 0;
}
```

上面的代码编译和执行时，它产生以下结果：

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

## 可重载/不可重载的运算符

下面这张表列举了可以重载的运算符：

| | | | | | |
|---|---|---|---|---|---|
| + | - | * | / | % | ^ |
| & | \| | ~ | ! | , | = |
| < | > | <= | >= | ++ | -- |
| << | >> | == | != | && | \|\| |
| += | -= | /= | %= | ^= | &= |
| \|= | *= | <<= | >>= | [ ] | () |
| -> | ->* | new | new[ ] | delete | delete[ ] |

下面这张表列举了不可以重载的运算符：

| | | | |
|---|---|---|---|
| :: | .* | . | ?: |

## 运算符重载例子

这里有各种操作符重载的例子来帮助你理解这一概念。

| 序号 | 运算符和例子 |
|---|---|
| 1 | 一元运算符重载 |
| 2 | 二元运算符重载 |
| 3 | 关系运算符重载 |
| 4 | 输入/输出运算符重载 |
| 5 | ++ 和 -- 运算符重载 |
| 6 | 赋值运算符重载 |
| 7 | 函数 call() 运算符重载 |
| 8 | 下标[ ]运算符重载 |
| 9 | 类成员获取运算符 -< 重载 |

# 多态

多态性意味着有多种形式。通常，多态发生在类之间存在层级关系且这些类有继承关系的时候。

C++ 多态性是指不同的对象发送同一个消息，不同对象对应同一消息产生不同行为。

考虑下面的例子，一个基类派生了其他的两类：

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

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

```
};
// Main function for the program
int main( )
{
   Shape *shape;
   Rectangle rec(10,7);
   Triangle  tri(10,5);

   // store the address of Rectangle
   shape = &rec;
   // call rectangle area.
   shape->area();

   // store the address of Triangle
   shape = &tri;
   // call triangle area.
   shape->area();

   return 0;
}
```

上面的代码编译和执行时，它产生以下结果：

```
Parent class area
Parent class area
```

输出结果不正确的原因是对函数 `area()` 的调用被编译器设置了一次，即在基类中定义的版本，这被称为对函数调用的静态分辨或者静态链接，静态链接就是在程序被执行之前函数调用是确定的。这有时也被称为早期绑定，因为函数 `area()` 在编译程序期间是固定的。

但是现在，让我们对程序做略微修改，并在 Shape 类中 `area()` 的声明之前加关键字 **virtual** ，它看起来像这样：

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

```
    return 0;
    }
};
```

这轻微的修改后，前面的示例代码编译和执行时，它会产生以下结果：

```
Rectangle class area
Triangle class area
```

这一次，编译器关注的是指针的内容而不是它的类型。因此，由于三角形和矩形类对象的地址被存储在形状类中，各自的 `area()` 函数可以被调用。

正如你所看到的，每个子类都有一个对 `area()` 函数的实现。通常多态就是这样使用的。你有不同的类，它们都有一个的相同名字的函数，甚至有相同的参数，但是对这个函数有不同的实现。

## 虚函数

基类中的虚函数是一个使用关键字 **virtual** 声明的函数。派生类中已经对函数进行定义的情况下，定义一个基类的虚函数，就是要告诉编译器我们不想对这个函数进行静态链接。

我们所希望的是根据调用函数的对象的类型对程序中在任何给定指针中被调用的函数的选择。这种操作被称为动态链接，或者后期绑定。

## 纯虚函数

可能你想把虚函数包括在基类中，以便它可以在派生类中根据该类的对象对函数进行重新定义，但在许多情况下，在基类中不能对虚函数给出有意义的实现。

我们可以改变基类中的虚函数 `area()` 如下：

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

`area() = 0` 就是告诉编译器上面的函数没有函数体。上面的虚函数就被称为纯虚函数。

# 数据抽象

数据抽象是指对外只提供基本信息并且隐藏他们的背景细节，即只呈现程序中所需的信息而没有提供细节。

数据抽象是一种编程(和设计)技术，依赖于接口和实现的分离。

让我们举一个现实生活中的例子。一个电视，你可以打开和关闭，改变频道，调整音量，并添加外部组件，比如扬声器，录像机，以及 DVD 播放器。但是你不知道它的内部细节，也就是说，你不知道它如何通过无线技术或者通过电缆接收信号，如何转化信号，以及最后将信号显示在屏幕上。

因此，我们可以说电视机实现了外部接口与内部实现的清晰分离，你可以无需知道它的内部具体实现，就可以通过其外部接口比如电源按钮，更换频道，音量控制。

现在，如果我们谈论的是 C++ 编程， C++ 类提供了大量数据抽象的例子。他们提供了大量的针对外部世界的公有函数来满足对象的功能或者操作对象数据，即外部函数不知道类在内部是如何实现的。

例如，你的程序可以在不知道函数实际使用什么算法来对给定的值进行排序的情况下调用 `sort()` 函数。事实上，排序功能的底层实现可以在不同版本之间变化，只要接口保持不变，你的函数调用将仍然起作用。

在 C++ 中，我们使用类来定义自己的抽象数据类型( ADT )。您可以使用类 ostream 的 cout 对象对流数据进行标准输出如下：

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

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

在这里，你不需要了解 cout 如何在用户的屏幕上显示文本。你只需要知道的公共接口和的 cout 底层实现是可以自由改变的。

## 访问标号实施抽象

在 C++ 中，我们使用访问标号定义抽象接口类。一个类可以包含零个或多个访问标签：

- 成员定义了一个公有标号，程序的所有部分都可以访问这个公共标号。类型的数据抽象视图由其公有成员定义。

- 使用类的代码不可以访问带有私有标号的成员。对于使用类的代码，私有部分隐藏了类的实现细节。

一个访问标号可以出现的次数通常是没有限制的。每个访问标号指定了随后的成员定义的访问级别。这个指定的访问级别持续有效，知道遇到下一个访问标号或看到类定义提的右花括号为止。

## 数据抽象的好处

数据抽象提供了两个重要的优势：

- 避免内部出现无意的，可能破坏对象状态的用户级错误。
- 随着时间的推移类实现可能会根据需求或缺陷报告来做出修改，但是这种修改无需改变用户级代码。

通过只在类的私有部分定义数据成员，类作者可以自由的对数据进行更改。如果实现更改，只需要检查类的代码看看这个改变可能造成什么影响。如果数据是公开的，那么任何可以直接访问旧的数据成员的函数都可能遭到破坏。

## 数据抽象举例

任何一个用公有和私有成员实现一个类的 C++ 程序都是数据抽象的一个例子。考虑下面的例子：

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

class Adder{
   public:
  // constructor
  Adder(int i = 0)
  {
total = i;
  }
  // interface to outside world
  void addNum(int number)
  {
  total += number;
  }
  // interface to outside world
  int getTotal()
  {
  return total;
  };
   private:
```

```
  // hidden data from outside world
  int total;
};
int main( )
{
   Adder a;

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

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

编译和执行上面的代码时，它产生以下结果：

```
Total 60
```

上面的类实现了把数字加起来，并且返回总和。公有成员 addNum 和 getTotal 是对外的接口，用户需要知道他们才能使用的类。私有成员 total 是用户不需要知道的，但是它是为保证程序正常运行类必要的。

## 设计策略

抽象使代码分离成接口和实现。所以在设计你的组件的时候，你必须保持接口独立于实现，因此，你才能做到在改变底层实现时，界面将保持不变。

在这种情况下，无论任何程序使用这些接口，他们不会受到影响，只需要重新编译最新的实现。

# 数据封装

所有的 C++ 程序是由以下两个基本要素组成：

- 程序语句(代码)：这是程序执行行为的一部分，他们被称为函数。
- 程序数据：数据是受程序函数影响的信息。

封装是一个面向对象编程的概念，它将数据和操作数据的函数结合在一起，并使其免受外部干扰和误用。数据封装是**数据隐藏**的重要面向对象编程概念。

数据封装是一种将数据和使用数据的函数结合在一起的机制；数据抽象是一种只将接口公开并且向用户隐藏实现细节的机制。

C++ 支持封装的属性和通过创建用户定义类型实现的数据隐藏，这称为类。我们已经研究过，一个类可以包含私有、保护和公有成员。默认情况下，所有定义在一个类中的成员是私有的。例如：

```
class Box
{
   public:
  double getVolume(void)
  {
 return length * breadth * height;
  }
   private:
  double length;  // Length of a box
  double breadth; // Breadth of a box
  double height;  // Height of a box
};
```

变量 length 、breadth 和 height 都是私有的。这意味着只有 box 类的其他成员可以访问它们，而程序的任何其它的部分不能访问它们。这是一个封装的实现方式。

要想使类的某个部分成为共有的(即访问您的程序的其他部分)，你必须在 public 关键字后声明它们。公有说明符后定义的所有变量或函数可以被程序中的其它函数访问。

使一个类成为其它类的友元类就可以获得实现细节，降低封装。这个思想就是获得尽可能多的每个类的对其它类隐藏的细节。

## 数据封装的例子

任何一个实现有公有和私有成员的类的 C++ 程序都是一个数据封装和数据抽象的例子。考虑下面的例子：

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

class Adder{
   public:
  // constructor
  Adder(int i = 0)
  {
total = i;
  }
  // interface to outside world
  void addNum(int number)
  {
  total += number;
  }
  // interface to outside world
  int getTotal()
  {
  return total;
  };
   private:
  // hidden data from outside world
  int total;
};
int main( )
{
   Adder a;

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

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

编译和执行上面的代码时，它产生以下结果：

```
Total 60
```

上面的类实现了把数字加起来，并且返回总和。公有成员 addNum 和 getTotal 是对外的接口，用户需要知道他们才能使用的类。私有成员 total 是用户不需要知道的，但是它是为保证程序正常运行类所必要的。

## 设计策略

经过一段痛苦的经历,我们大多数人已经学会了使类成员在默认情况下是私有的,除非我们真的需要使它们变成公有的。这就是一个好的的封装。

这个知识被频繁的应用于数据成员,它同样适用于所有成员，包括虚函数。

# 接口（抽象类）

接口可以用来描述一个 C++ 类的行为或功能,但是并不需要对这个类进行实现。

C++ 接口是通过抽象类来实现的，这些抽象类不应与数据抽象混淆，数据抽象的概念：概念结构是对现实世界的一种抽象，从实际的人、物、事和概念中抽取所关心的共同特性，忽略非本质的细节，把这些特性用各种概念精确地加以描述，这些概念组成了某种模型。

一个抽象类的声明里至少要有一个函数作为纯虚函数。在函数形参表后面写上 “= 0” 以指定纯虚函数：

```
class Box
{
   public:
  // pure virtual function
  virtual double getVolume() = 0;
   private:
  double length;  // Length of a box
  double breadth; // Breadth of a box
  double height;  // Height of a box
};
```

建立抽象类（通常被称为一个ABC）的目的是提供一个适当的并且其他类可以继承的基类。抽象类不能实例化对象并且只能作为一个接口使用。试图实例化一个抽象类的对象会导致编译错误。

因此,如果一个抽象类的子类的需要实例化,它必须实现所有的虚函数,这意味着它支持抽象类的接口声明。如果在派生类中未能覆盖一个纯虚函数,然后试图实例化该类的对象,会导致一个编译错误。

可用于实例化对象的类被称为具体类。

## 抽象类样例

考虑下面的例子,父类为基类提供了一个接口来实现函数 `getArea()`：

```
#include <iostream>

using namespace std;

// Base class
class Shape
{
public:
```

```
   // pure virtual function providing interface framework.
   virtual int getArea() = 0;
   void setWidth(int w)
   {
  width = w;
   }
   void setHeight(int h)
   {
  height = h;
   }
protected:
   int width;
   int height;
};

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

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

   Rect.setWidth(5);
   Rect.setHeight(7);
   // Print the area of the object.
   cout << "Total Rectangle area: " << Rect.getArea() << endl;

   Tri.setWidth(5);
   Tri.setHeight(7);
```

```
    // Print the area of the object.
    cout << "Total Triangle area: " << Tri.getArea() << endl;

    return 0;
}
```

上面的代码编译和执行后,将产生以下结果:

```
Total Rectangle area: 35
Total Triangle area: 17
```

## 设计策略

一个面向对象的系统可能使用一个抽象基类提供一个普遍的和适合所有的外部应用程序的标准化接口。然后，通过继承的抽象基类，形成派生类。

功能(即，公共函数)，即由外部应用程序提供的，作为抽象基类里面的纯虚函数。 那些纯虚函数是由派生类实现的，派生类对应于应用的特定类型。

即使在系统已经定义之后， 这种架构还允许添加新的应用程序到系统中。

# 3

# C++ 进阶

# 文件和流

到目前为止,我们一直在使用 iostream 标准库,它提供了 cin 及 cout方法，分别用于读取标准输入以及写入标准输出。

本教程将教你如何从文件中读取和写入。这需要另一个称为 fstream 的标准 C++ 库,它定义了三个新的数据类型:

| 数据类型 | 描述 |
| --- | --- |
| ofstream | 这个数据类型表示输出文件流，用于创建文件以及将信息写入文件。 |
| ifstream | 这个数据类型表示输入文件流,用于从文件读取信息。 |
| fstream | 这个数据类型通常表示该文件流,兼备有 ofstream 和 ifstream 功能，这意味着它可以创建文件,编写文件,以及读文件。 |

使用 C++ 执行文件处理时，头文件 `<iostream>` 和 `<fstream>` 必须包含在你的 C++ 源文件里面。

## 打开文件

需要对一个文件进行读写操作时必须先打开该文件。 ofstream 或 fstream 对象可以用来打开一个文件并且写入； ifstream 对象用于以读入打开一个文件。

下面是 `open()` 函数的标准语法,它是 fstream，ifstream，ofstream 对象的成员。

```
    void open(const char *filename, ios::openmode mode) ;
```

在这里,第一个参数指定文件的名称和打开位置,open()成员函数的第二个参数定义了文件应该以哪种模式被打开。

| 模式标志 | 描述 |
| --- | --- |
| ios::app | 追加模式。所有输出文件附加到结尾。 |
| ios::ate | 为输出打开一个文件并将读/写控制移动到文件的末尾。 |
| ios::in | 打开一个文件去读。 |
| ios::out | 打开一个文件去写。 |
| ios::trunc | 如果文件已经存在,打开该文件前文件中的内容将被清空。 。 |

您可以通过逻辑运算将两个或更多的这些值组合到一起。例如, 如果你想以写方式打开一个文件, 并且想在其打开之前清空内容,以防它已经存在的话,使用一下语法规则:

```
ofstream outfile;
outfile.open("file.dat", ios::out | ios::trunc );
```

同样,你可以以读入和写入目的打开一个文件如下:

```
fstream  afile;
afile.open("file.dat", ios::out | ios::in );
```

## 关闭文件

一个 C++ 程序终止时它会自动关闭所有流,释放所有分配的内存并关闭所有打开的文件。但在终止之前，程序员应该关闭所有打开的程序文件始终是一个很好的实习惯。

下面是标准的 `close()` 函数语法,它是一个 fstream，ifstream,以及 ofstream对象的成员。

```
void close();
```

## 写文件

在使用 C++ 编程时,你通过程序使用流插入操作符 `(< <)` 将信息写入文件，使用流插入操作符 `(< <)` 就像你使用键盘输入将信息输出到屏幕上。唯一的区别是，你使用一个 ofstream 或 fstream 对象而不是 cout。

## 读文件

您使用留提取符（`>>`）将文件中的信息读入程序就像你使用该运营商从键盘输入信息。唯一的区别是,你使用一个 ifstream 或 fstream 对象而不是 cin 的对象。

## 读取与写入样例

下面是一段 C++ 程序，以读取和写入方式打开一个文件。将用户输入的信息写入文件后以 `afile.dat` 命名文件。程序从文件中读取信息并输出在屏幕上:

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

int main ()
{
```

```
    char data[100];

    // open a file in write mode.
    ofstream outfile;
    outfile.open("afile.dat");

    cout << "Writing to the file" << endl;
    cout << "Enter your name: ";
    cin.getline(data, 100);

    // write inputted data into the file.
    outfile << data << endl;

    cout << "Enter your age: ";
    cin >> data;
    cin.ignore();

    // again write inputted data into the file.
    outfile << data << endl;

    // close the opened file.
    outfile.close();

    // open a file in read mode.
    ifstream infile;
    infile.open("afile.dat");

    cout << "Reading from the file" << endl;
    infile >> data;

    // write the data at the screen.
    cout << data << endl;

    // again read the data from the file and display it.
    infile >> data;
    cout << data << endl;

    // close the opened file.
    infile.close();

    return 0;
}
```

当上面的代码被编译并执行，将产生如下的样本和输出：

```
$./a.out
Writing to the file
Enter your name: Zara
Enter your age: 9
Reading from the file
Zara
9
```

上面的例子利用 cin 对象额外的功能,如利用 getline()函数来从外部读取线,用 ignor()函数忽略先前读取语句留下的额外字符。

## 文件位置指针

istream 和 ostream 是用来重新定位文件位置的指针成员函数。这些成员函数有 seekg (“seek get”) istream 和 seekp 上 ostream (“seek put”)。

seekg 和 seekp 通常的参数是一个长整型。可以指定第二个参数为寻找方向。寻找方向可以 `ios: :beg` (默认)定位相对于流的开始， `io: :scur` 定位相对于当前位置的流或 `ios::end` 定位相对于流的结束。

文件位置指针是一个整数值,它指定文件的位置作为一个从文件的开始位置的字节数。

文件位置指针是一个整数值,它指定文件的位置。定位 “get” 文件位置指针的一些示例如下:

```
// position to the nth byte of fileObject (assumes ios::beg)
fileObject.seekg( n );

// position n bytes forward in fileObject
fileObject.seekg( n, ios::cur );

// position n bytes back from end of fileObject
fileObject.seekg( n, ios::end );

// position at end of fileObject
fileObject.seekg( 0, ios::end );
```

# 异常处理

异常是一个程序执行过程中出现的问题。C++ 异常是对程序运行过程中产生的例外情况作出的响应,比如试图除以零。

异常提供一种方法将程序控制从一个程序的一部分转移到另一部分。C++ 异常处理是建立在三个关键词：尝试,捕获和抛出之上的。

- throw：程序运行出现问题时抛出异常。这是使用一个 throw 关键字实现的。
- catch：程序用异常处理器在你想要处理问题的地方捕获异常。catch 关键字显示异常的捕获。
- try：一个 try 块标识一个可能会产生异常的代码块。紧随其后的是一个或多个 catch 块。

假设一个代码块将产生一个异常,结合使用 try 和 catch 关键词的方法捕获了一个异常。一个 try / catch 块放置在可能生成一个异常的代码周围。在一个 try / catch 块里面的代码被称为保护代码, try / catch 的语法规则如下：

```
try
{
   // protected code
}catch( ExceptionName e1 )
{
   // catch block
}catch( ExceptionName e2 )
{
   // catch block
}catch( ExceptionName eN )
{
   // catch block
}
```

你可以列出多个捕捉语句捕获不同类型的异常, 以防你的 try 代码块在不同的情况下产生了不止一个异常。

## 抛出异常

异常可以在代码块的任何地方使用抛出语句抛出。把语句的操作数确定类型的异常,可以是任何表达式,表达式的结果的类型决定了类型的异常抛出。

下面是一个例子, 在除以零条件发生时，抛出异常：

```
double division(int a, int b)
{
   if( b == 0 )
   {
  throw "Division by zero condition!";
   }
   return (a/b);
}
```

## 捕获异常

try 块后的 catch 块可以捕获任何异常。您可以指定你需要捕获何种类型的异常,这是由出现在关键字 catch 后边的括号中的异常声明确定的。

```
try
{
   // protected code
}catch( ExceptionName e )
{
  // code to handle ExceptionName exception
}
```

上面的代码将捕获到一个 ExceptionName 类型的异常。如果您想要指定一个 catch 块可以应该处理任何在 try 代码中产生的异常，你必须将一个省略号…放在 catch 后的括号中，异常声明如下：

```
try
{
   // protected code
}catch(...)
{
  // code to handle any exception
}
```

下面是一个例子，这个例子抛出会除零异常, 我们在 catch 块里面捕获它

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

double division(int a, int b)
{
   if( b == 0 )
   {
  throw "Division by zero condition!";
   }
```

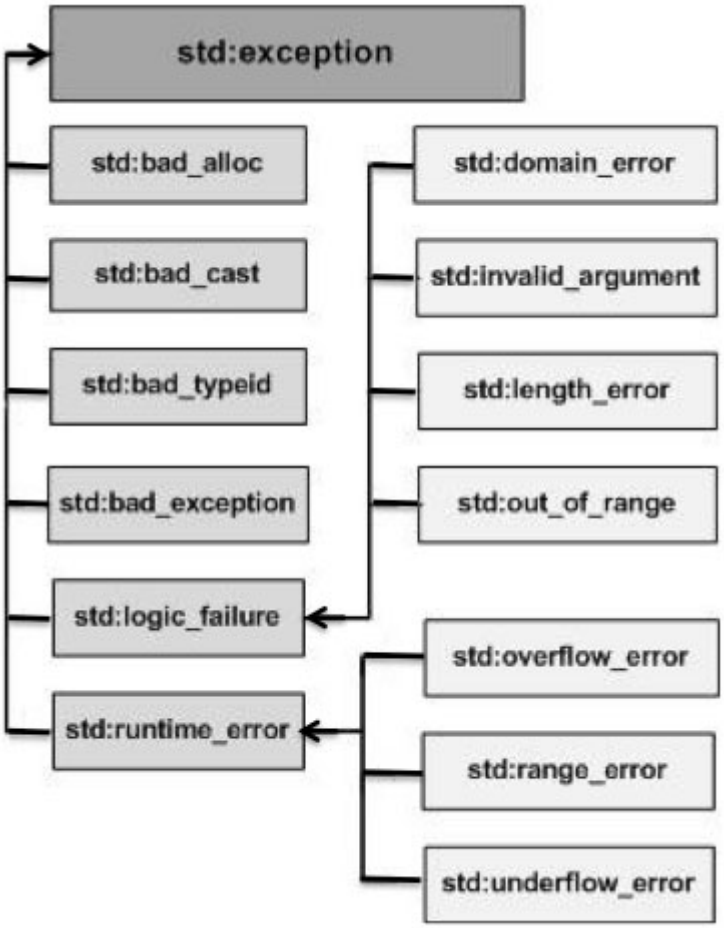
```
   return (a/b);
}

int main ()
{
   int x = 50;
   int y = 0;
   double z = 0;

   try {
 z = division(x, y);
 cout << z << endl;
   }catch (const char* msg) {
 cerr << msg << endl;
   }

   return 0;
}
```

因为上例中提出了一个 `const char *` 类型的异常，所以捕捉这个异常时,我们必须在 catch 块中使用 `const char *` 。如果我们编译和运行上面的代码，这将产生以下结果：

```
Division by zero condition!
```

## C++ 标准异常

C++ 在 `<exception>` 中提供了一系列标准的异常，我们可以用在我们的程序中。这些异常使用父-子分层结构展示如下：




这是对上面提到的层次结构中每个异常的描述：

| 异常 | 描述 |
| --- | --- |
| **std::exception** | 异常和所有标准 C++ 异常的父类 |
| std::bad_alloc | 该异常可能会在使用 new 关键字时抛出。 |
| std::bad_cast | 该异常可以由 dynamic_cast 抛出。 |
| std::bad_exception | 这是一个在 C++ 程序中处理意想不到的异常的有效手段。 |
| std::bad_typeid | 该异常可以由 typeid 抛出。 |
| **std::logic_error** | 理论上可以通过阅读代码发现的异常。 |
| std::domain_error | 这是一个在数学无效域被使用时抛出的异常。 |
| std::invalid_argument | 参数非法时会抛出的异常。 |
| std::length_error | 太大的 std::string 被创造时，抛出异常。 |
| std::out_of_range | 可以抛出该异常的方法例如 std::vector 和 std::bitset < >::operator[] ()。 |
| **std::runtime_error** | 理论上不能通过读代码检测到的异常。 |
| std::overflow_errorr | 如果出现数字溢出，则抛出该异常 |
| std::range_error | 当你试图存储一个超过范围的值的时候，会抛出该异常。 |
| std::underflow_error | 如果出现数学下溢时，抛出该异常。 |

## 定义新异常

你可以采用继承及重写异常类来。下面是示例，显示如何使用 `std::exception` 类以标准的方式实现自己的异常：

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

struct MyException : public exception
{
  const char * what () const throw ()
  {
return "C++ Exception";
  }
};

int main()
{
  try
  {
throw MyException();
  }
  catch(MyException& e)
  {
std::cout << "MyException caught" << std::endl;
std::cout << e.what() << std::endl;
  }
  catch(std::exception& e)
  {
//Other errors
  }
}
```

这将产生如下的结果：

```
MyException caught
C++
```

这里，`what()` 是一个异常类提供的公共方法，所有子异常类都覆盖了该方法。这将返回一个异常的原因。

# 动态内存

很好地理解动态内存到底如何在 C++ 中发挥作用是成为一个好的 C++ 程序员所必需的。 C++ 程序中的内存分为两个部分：

- 栈：所有函数内部声明的变量会占用栈的内存。
- 堆：这是程序中未使用的内存，可以在程序运行时动态地分配内存。

很多时候，你事先不知道你在一个定义的变量中需要多少内存来存储特定的信息以及在程序运行时所需内存的大小。

你可以在运行时为指定类型的变量分配堆内存，并且可以使用 C++ 中特殊操作符返回分配空间的地址。这个操作符被称为 new 操作符。

如果你不再需要动态分配内存了，你可以使用 delete 操作符来释放之前用 new 操作符分配的内存。

## new 和 delete 操作符

下面是使用 new 操作符为任意数据类型动态地分配内存的通用的语法。

```
new data-type;
```

这里， data-type 可以是任何内置数据类型，包括数组或任何用户定义的数据类型包括类或结构。让我们先看看内置的数据类型。例如，我们可以定义一个 double 类型的指针然后在程序执行时请求分配内存。我们可以使用 new 操作符来完成它，程序语句如下：

```
double* pvalue  = NULL; // Pointer initialized with null
pvalue  = new double;   // Request memory for the variable
```

如果自由存储区都已经被占用，内存可能就不能被成功分配。因此检查 new 操作符是否返回空指针是一种很好的做法，并且要采取适当的措施如下：

```
double* pvalue  = NULL;
if( !(pvalue  = new double ))
{
   cout << "Error: out of memory." <<endl;
   exit(1);

}
```

C 语言中的 malloc() 函数在C++中仍然存在，但是建议避免使用 malloc() 函数。相对于 malloc() 函数 new 操作符的主要优势是 new 操作符不仅分配内存，它还可以构造对象，而这正是 C++ 的主要目的。

在任何时候，当你觉得一个变量已经不再需要动态分配，你可以用 delete 操作符来释放它在自由存储区所占用的内存，如下：

```
delete pvalue;// Release memory pointed to by pvalue
```

让我们把理解一下这些概念，并且用下面的例子来说明 new 和 delete 是如何起作用的：

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

int main ()
{
   double* pvalue  = NULL; // Pointer initialized with null
   pvalue  = new double;   // Request memory for the variable

   *pvalue = 29494.99; // Store value at allocated address
   cout << "Value of pvalue : " << *pvalue << endl;

   delete pvalue; // free up the memory.

   return 0;
}
```

如果我们编译和运行上面的代码，这将产生以下结果：

```
Value of pvalue : 29495
```

## 数组的动态内存分配

考虑到你想要为字符数组分配内存，即 20 个字符的字符串。使用与上面相同的语法我们可以动态地分配内存，如下所示。

```
char* pvalue  = NULL;   // Pointer initialized with null
pvalue  = new char[20]; // Request memory for the variable
```

应该像这样删除我们刚刚创建的数组声明：

```
delete [] pvalue;// Delete array pointed to by pvalue
```

学习过 new 操作符的类似通用语法，你可以为一个多维数组分配内存如下：

```
double** pvalue  = NULL; // Pointer initialized with null
pvalue  、= new double [3][4]; // Allocate memory for a 3x4 array
```

然而,释放多维数组内存的语法仍然同上:

```
delete [] pvalue;// Delete array pointed to by pvalue
```

## 对象的动态内存分配

对象与简单的数据类型并无不同。例如，考虑下面的代码，我们将使用一个对象数组来解释这个概念：

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

class Box
{
   public:
  Box() {
 cout << "Constructor called!" <<endl;
  }
  ~Box() {
 cout << "Destructor called!" <<endl;
  }
};

int main( )
{
   Box* myBoxArray = new Box[4];

   delete [] myBoxArray; // Delete array

   return 0;
}
```

如果你为四个 Box 对象数组分配内存，一个简单的构造函数将被调用四次，同样的删除这些对象时，析构函数也被调用相同的次数。

如果我们编译和运行上面的代码，这将产生以下结果：

```
Constructor called!
Constructor called!
Constructor called!
Constructor called!
Destructor called!
```

```
Destructor called!
Destructor called!
Destructor called!
```

# 命名空间

考虑一个情况，在同一个班有两个同名的人，都叫 Zara 。每当我们需要区分他们的时候，除了它们的名字我们肯定会使用一些额外的信息，就像如果他们住在不同的区域或他们的母亲或父亲的名字，等等。

同样的情况会出现在你的 C++ 应用程序中。例如，你可能会编写一些代码，有一个名为 xyz() 的函数，在另一个库中也有同样的函数 xyz() 。现在编译器不知道在你的代码中指定的是哪个版本的 xyz() 函数。

namespace就是用来克服这个困难的，而且作为附加信息来区分在不同的库中具有相同名称的函数，类、变量等。使用命名空间，你可以定义名字已经定义的上下文。从本质上讲，一个命名空间定义了一个范围。

## 定义命名空间

一个命名空间的定义由关键字 namespace 加它的名称组成，如下所示：

```
namespace namespace_name {
       // code declarations
}
```

调用任何函数或变量的命名空间启用版本，前面加上命名空间名字如下：

```
name::code;  // code could be variable or function.
```

让我们看看命名空间如何限定实体（包括变量和函数）使用范围：

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

// first name space
namespace first_space{
   void func(){
  cout << "Inside first_space" << endl;
   }
}
// second name space
namespace second_space{
   void func(){
  cout << "Inside second_space" << endl;
   }
}
int main ()
```

```
{

   // Calls function from first name space.
   first_space::func();

   // Calls function from second name space.
   second_space::func();

   return 0;
}
```

如果我们编译和运行上面的代码，这将产生以下结果：

```
Inside first_space
Inside second_space
```

## using 指令

你可以通过使用 using namespace 指令来避免在头部添加命名空间。这个指令告诉编译器，随后代码要使用指定命名空间中的名称。因此名称空间隐含下面的代码：

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

// first name space
namespace first_space{
   void func(){
  cout << "Inside first_space" << endl;
   }
}
// second name space
namespace second_space{
   void func(){
  cout << "Inside second_space" << endl;
   }
}
using namespace first_space;
int main ()
{

   // This calls function from first name space.
   func();
```

```
   return 0;
}
```

如果我们编译和运行上面的代码，这将产生以下结果：

```
Inside first_space
```

using 指令也可以用来指一个名称空间中的特定的项目。例如，如果你打算只是用 std 名称空间的一部分cout，你可以进行如下操作：

```
using std::cout;
```

后续的代码可以调用 cout 而不用在前面加上命名空间名字，但命名空间中的其他项目仍需要作如下说明：

```
#include <iostream>
using std::cout;

int main ()
{

   cout << "std::endl is used with std!" << std::endl;

   return 0;
}
```

如果我们编译和运行上面的代码，这将产生以下结果：

```
std::endl is used with std!
```

using 指令引入的名字遵循正常的检测规则。这个名字相对于从 using 指令的指针到范围的结束是可见的，并且在这个范围中指令可以被找到。定义在外部范围的有相同名字的实体是被隐藏的。

## 不连续的命名空间

一个命名空间可以被分别定义为若干个不同部分，因此命名空间是由这些部分的合集组成。这些被分开定义的命名空间可以分散在多个文件中。一个命名空间的分离的部分可以分散在多个文件。

所以，如果命名空间的一部分需要定义在另一个文件中的名字，仍然必须声明这个名字。定义以下命名空间实现定义一种新的命名空间或添加新的元素到一个现有的命名空间：

```
namespace namespace_name {
   // code declarations
}
```

## 嵌套的命名空间

命名空间可以被嵌套，你可以在一个命名空间内定义另一个命名空间，如下：

```
namespace namespace_name1 {
   // code declarations
   namespace namespace_name2 {
  // code declarations
   }
}
```

在上面的语句中如果你使用的是 namespace _ name1 ，那么它将使 namespace _ name2 的元素在整个范围内可用，如下：

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

// first name space
namespace first_space{
   void func(){
  cout << "Inside first_space" << endl;
   }
   // second name space
   namespace second_space{
  void func(){
 cout << "Inside second_space" << endl;
  }
   }
}
using namespace first_space::second_space;
int main ()
{

   // This calls function from second name space.
   func();

   return 0;
}
```

如果我们编译和运行上面的代码，这将产生以下结果：

```
Inside second_space
```

# 模板

模板是泛型编程的基础。泛型编程就是以独立于任何特定类型的方式编写代码。

模板是创建泛型类或函数的蓝图或公式。

使用模板的概念开发的库容器，像迭代器和算法都是泛型编程的例子。

每个容器都有一个单一定义，例如 **vector**，但我们也可以定义许多不同类型的 vector，如 `vector <int>` 或 `vector <string>` 。

你也可以使用模板定义函数和类，让我们看看是怎么做的：

## 函数模板

模板函数定义的一般形式如下所示：

```
template <class type> ret-type func-name(parameter list)
{
   // body of function
}
```

这里的 **type** 是函数使用的数据类型的占位符名称。 这个名称可以在函数定义内使用。

下面是一个返回两个值中的最大值的函数模板例子：

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

using namespace std;

template <typename T>
inline T const& Max (T const& a, T const& b)
{
return a < b ? b:a;
}
int main ()
{

int i = 39;
int j = 20;
cout << "Max(i, j): " << Max(i, j) << endl;
```

```
double f1 = 13.5;
double f2 = 20.7;
cout << "Max(f1, f2): " << Max(f1, f2) << endl;

string s1 = "Hello";
string s2 = "World";
cout << "Max(s1, s2): " << Max(s1, s2) << endl;

   return 0;
}
```

如果我们编译并运行上述代码，将会产生以下结果：

```
Max(i, j): 39
Max(f1, f2): 20.7
Max(s1, s2): World
```

## 类模板

就像我们可以定义函数模板一样，我们也可以定义类模板。

模板类定义的一般形式如下所示：

```
template <class type> class class-name {
.
.
.
}
```

这里的 type 是一个类型的占位符名称，当类实例化的时候，此类型会被指定。 你可以用一个逗号隔开的列表定义多个泛型数据类型。

以下是一个定义 `Stack<>` 类并实现泛型方法来压入和弹出堆栈元素的例子：

```
#include <iostream>
#include <vector>
#include <cstdlib>
#include <string>
#include <stdexcept>

using namespace std;

template <class T>
```

```
class Stack {
  private:
vector<T> elems; // elements

  public:
void push(T const&);  // push element
void pop();   // pop element
T top() const;// return top element
bool empty() const{   // return true if empty.
return elems.empty();
}
};

template <class T>
void Stack<T>::push (T const& elem)
{
// append copy of passed element
elems.push_back(elem);
}

template <class T>
void Stack<T>::pop ()
{
if (elems.empty()) {
throw out_of_range("Stack<>::pop(): empty stack");
}
    // remove last element
elems.pop_back();
}

template <class T>
T Stack<T>::top () const
{
if (elems.empty()) {
throw out_of_range("Stack<>::top(): empty stack");
}
    // return copy of last element
return elems.back();
}

int main()
{
try {
Stack<int> intStack;  // stack of ints
Stack<string> stringStack;// stack of strings
```

```
// manipulate int stack
intStack.push(7);
cout << intStack.top() <<endl;

// manipulate string stack
stringStack.push("hello");
cout << stringStack.top() << std::endl;
stringStack.pop();
stringStack.pop();
}
catch (exception const& ex) {
cerr << "Exception: " << ex.what() <<endl;
return -1;
}
}
```

如果我们编译并运行上述代码，将会产生以下结果：

```
7
hello
Exception: Stack<>::pop(): empty stack
```

# 预处理器

预处理器是指令，在实际编译开始之前，预处理器会给编译器指令来预处理需要编译的信息。

所有的预处理指令以 # 开头，可能在预处理指令行之前出现的只有空白字符。预处理指令是不是 C++ 语句，所以它们不会以分号( ; )结束。

你已经在所有的例子中都看到了 **#include** 指令。这个宏用于将头文件包含到源文件。

还有许多 C++ 支持的预处理指令，如 `#include` , `#define` , `#if` , `#else` , `#line` 等等。下面我们来看一些重要的指令：

## #define 处理器

`#define` 处理指令用来创建符号常量。这个符号常量被称为**宏**。指令的一般形式是：

```
#define macro-name replacement-text
```

当这一行出现在一个文件中时，在程序被编译之前，该文件中的所有后续出现的 macro-name 将会被 replacement-text 替换掉。 例如：

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

#define PI 3.14159

int main ()
{

cout << "Value of PI :" << PI << endl;

return 0;
}
```

现在，让我们对这个代码进行预处理来看一看结果。假设我们有源代码文件，那么我们用 -E 选项编译它并重定向结果到 test.p。现在，如果你查看 test.p，会发现里面有大量的信息，并且你会在底部发现值已经被替换了，如下所示：

```
$gcc -E test.cpp > test.p

...
```

```
int main ()
{

cout << "Value of PI :" << 3.14159 << endl;

return 0;
}
```

## 函数宏

你可以用 `#define` 定义一个带有如下参数的宏：

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

#define MIN(a,b) (((a)<(b)) ? a : b)

int main ()
{
   int i, j;
   i = 100;
   j = 30;
   cout <<"The minimum is " << MIN(i, j) << endl;

return 0;
}
```

如果我们编译并运行上述代码，将会产生以下结果：

```
The minimum is 30
```

## 条件编译

有几种不同的指令，其可用于有选择的编译程序源代码的一部分。这个过程被称为条件编译。

条件预处理器结构很像 if 选择结构。思考下面的预处理代码：

```
#ifndef NULL
   #define NULL 0
#endif
```

你可以编译一个用于调试的程序，并且可以用单个宏打开或关闭调试开关，如下所示：

```
#ifdef DEBUG
   cerr <<"Variable x = " << x << endl;
#endif
```

如果符号常量 DEBUG 定义在 `#ifdef DEBUG` 指令之前，那么程序中的 cerr 语句会被编译。你可以使用 `#if 0` 语句注释掉程序的一部分，如下所示：

```
#if 0
   code prevented from compiling
#endif
```

我们来用下面的例子试一下：

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

#define MIN(a,b) (((a)<(b)) ? a : b)

int main ()
{
   int i, j;
   i = 100;
   j = 30;
#ifdef DEBUG
   cerr <<"Trace: Inside main function" << endl;
#endif

#if 0
   /* This is commented part */
   cout << MKSTR(HELLO C++) << endl;
#endif

   cout <<"The minimum is " << MIN(i, j) << endl;

#ifdef DEBUG
   cerr <<"Trace: Coming out of main function" << endl;
#endif
return 0;
}
```

如果我们编译并运行上述代码，将会产生以下结果：

```
Trace: Inside main function
The minimum is 30
Trace: Coming out of main function
```

## # 和 ## 操作符

在 C++ 和 ANSI/ISO C 中 # 和 ## 预处理器操作符是可用的。

# 操作符会将要替代的文本符号转换成用双引号引起来的字符串。

思考下面的宏定义：

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

#define MKSTR( x ) #x

int main ()
{
cout << MKSTR(HELLO C++) << endl;

return 0;
}
```

如果我们编译并运行上述代码，将会产生以下结果：

```
HELLO C++
```

让我们来看看它是如何工作的。

这很容易理解，C++ 预处理器将下面一行代码：

```
cout << MKSTR(HELLO C++) << endl;
```

转变成了下面这一行的代码：

```
cout << "HELLO C++" << endl;
```

## 操作符是用来连接两个符号的。

例子如下所示：

```
#define CONCAT( x, y )  x ## y
```

当 CONCAT 出现在程序中时，它的参数会被连接起来，并用其来取代宏。例如，CONCAT(HELLO, C++)，在程序中会 “HELLO C++” 替代。例子如下所示。

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

```
#define concat(a, b) a ## b
int main()
{
   int xy = 100;

   cout << concat(x, y);
   return 0;
}
```

如果我们编译并运行上述代码，将会产生以下结果：

```
100
```

让我们来看看它是如何工作的。

这很容易理解，C++ 预处理器将下面一行代码：

```
cout << concat(x, y);
```

转换成了下面这一行的代码：

```
 cout << xy;
```

## C++ 预定义的宏

C++ 提供了许多预定义的宏，如下所示：

| 宏 | 描述 |
| --- | --- |
| __LINE__ | 编译过后，其包含了当前程序行在程序内的行号 |
| __FILE__ | 编译过后，其包含了当前程序的程序名 |
| __DATE__ | 其包含了由源文件转换为目标代码的日期，该日期是格式为 月/日/年 的字符串文本 |
| __TIME__ | 其包含了源文件编译的时间，该时间是 时:分:秒 形式的字符串文本 |

我们来看一个展示上面宏的例子：

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

int main ()
{
cout << "Value of __LINE__ : " << __LINE__ << endl;
cout << "Value of __FILE__ : " << __FILE__ << endl;
cout << "Value of __DATE__ : " << __DATE__ << endl;
```

```
    cout << "Value of __TIME__ : " << __TIME__ << endl;

    return 0;
    }
```

如果我们编译并运行上述代码，将会产生以下结果：

```
    Value of __LINE__ : 6
    Value of __FILE__ : test.cpp
    Value of __DATE__ : Feb 28 2011
    Value of __TIME__ : 18:52:48
```

# 信号处理

信号是由操作系统传递到进程的中断，它可以提前终止一个程序。在 UNIX，LINUX，Mac OS X 或 Windows 系统上，你可以通过按 Ctrl+C 产生一个中断。

有的信号不能被程序捕获到，但是下面列出的信号，你可以在程序中捕捉它们，并且可以基于这些信号进行相应的操作。这些信号定义在 C++ 头文件 `<csignal>` 中。

| 信号 | 描述 |
| --- | --- |
| SIGABRT | 程序的异常终止，例如调用 abort |
| SIGFPE | 一个错误的算术运算，例如除以零或运算结果溢出。 |
| SIGILL | 检测到非法指令。 |
| SIGINT | 接收到交互注意信号。 |
| SIGSEGV | 一个非法的存储访问。 |
| SIGTERM | 发送给程序的终止请求信号。 |

## signal() 函数

C++ 信号处理库提供 **signal** 函数来捕获意外事件。以下是 `signal()` 函数的语法：

```
void (*signal (int sig, void (*func)(int)))(int);
```

简单来说，这个函数接收两个参数：第一个参数是一个整数，表示信号号码；第二个参数是一个指向信号处理函数的指针。

让我们用 `signal()` 函数写一个简单的 C++ 程序，用它来捕捉 SIGINT 信号。不管你想在程序中捕获什么信号，你必须使用 **signal** 函数注册该信号，并将其与信号处理程序相关联。 例子如下所示：

```
#include <iostream>
#include <csignal>

using namespace std;

void signalHandler( int signum )
{
cout << "Interrupt signal (" << signum << ") received.\n";

// cleanup and close up stuff here
// terminate program
```

```
   exit(signum);

}

int main ()
{
// register signal SIGINT and signal handler
signal(SIGINT, signalHandler);

while(1){
   cout << "Going to sleep...." << endl;
   sleep(1);
}

return 0;
}
```

当上述代码编译和执行后，将会产生以下的结果：

```
Going to sleep....
Going to sleep....
Going to sleep....
```

现在，按 Ctrl+C 来中断这个程序，你会看到程序将捕获的信号，并会通过打印展示出来，如下所示：

```
Going to sleep....
Going to sleep....
Going to sleep....
Interrupt signal (2) received.
```

## raise() 函数

您可以通过 raise() 函数生成信号，它用一个整数的信号编号作为参数，语法如下所示。

```
int raise (signal sig);
```

这里的 sig 是要发送的信号编号，这些信号是：SIGINT，SIGABRT，SIGFPE，SIGILL，SIGSEGV，SIGTERM，SIGHUP。以下是我们使用 raise() 函数从程序内部发出一个信号的例子：

```
#include <iostream>
#include <csignal>

using namespace std;

void signalHandler( int signum )
```

```
{
cout << "Interrupt signal (" << signum << ") received.\n";

// cleanup and close up stuff here
// terminate program

   exit(signum);

}

int main ()
{
int i = 0;
// register signal SIGINT and signal handler
signal(SIGINT, signalHandler);

while(++i){
   cout << "Going to sleep...." << endl;
   if( i == 3 ){
  raise( SIGINT);
   }
   sleep(1);
}

return 0;
}
```

当上述代码编译和执行后，将会产生以下的结果，并且这些结果会自动出现：

```
Going to sleep....
Going to sleep....
Going to sleep....
Interrupt signal (2) received.
```

# 多线程

多线程是多任务处理的一种特殊形式，而多任务处理是一种让你的电脑能并发运行两个或两个以上程序的特性。一般有两种类型的多任务处理：基于进程的和基于线程的。

基于进程的多任务处理是并发执行的程序。基于线程的多任务处理是并发执行的程序的一部分。

多线程程序包含了可以并发运行的两个或更多个程序部分。这样程序中的每个部分称为一个线程，并且每个线程都定义了一个单独的执行路径。

C++ 不包含对多线程应用程序的任何嵌入式支持。相反，它完全依赖于操作系统来提供此项功能。

本教程假设您正在使用的是 Linux 操作系统，我们将要使用 POSIX 编写 C++ 多线程程序。 POSIX 线程，或称 Pthreads，它提供了在许多类 Unix 的 POSIX 系统（如 FreeBSD，NetBSD，GNU/Linux，Mac OS X 和 Solaris）中可用的 API。

## 创建线程

我们使用下面的函数来创建一个 POSIX 线程：

```
#include <pthread.h>
pthread_create (thread, attr, start_routine, arg)
```

这里的 pthread_create 创建了一个新线程，并使其可执行。这个函数可以在代码中的任意位置调用任意次。

下面是详细的参数说明：

| 参数 | 描述 |
| --- | --- |
| thread | 新线程的不透明、唯一的标识符，它由子函数返回。 |
| attr | 一个不透明的属性对象,可用于设置线程属性。你可以指定一个线程的属性对象，默认值为 NULL。 |
| start_routine | C++ 例程，线程一旦创建将会被执行。 |
| arg | 一个传递给 start_routine 的参数。它必须传递一个 void 类型指针的引用。如果没有参数传递，默认值为 NULL。 |

一个进程可创建的最大线程数是依赖实现决定的。线程一旦创建，它们之间是对等的，而且也有可能创建其它的线程。线程之间没有隐含的层次或依赖关系。

## 终止线程

我们使用下面的函数来终止一个 POSIX 线程：

```
#include <pthread.h>
pthread_exit (status)
```

此处的 pthread_exit 用于显式的退出一个线程。通常在线程已完成了其工作，并且没有存在的必要的时候，调用 `pthread_exit()` 函数。

如果 `main()` 在其创建的线程之前终止，并且使用了 `pthread_exit()` 来退出线程，那么其线程将会继续执行。否则，当 `main()` 终止后，这些线程将会自动终止。

## 例子

下面简单的样例代码，用 `pthread_create()` 函数创建了 5 个线程。每个线程均打印 “Hello World!”，然后调用 `pthread_exit()` 函数终止了线程。

```
#include <iostream>
#include <cstdlib>
#include <pthread.h>

using namespace std;

#define NUM_THREADS 5

void *PrintHello(void *threadid)
{
   long tid;
   tid = (long)threadid;
   cout << "Hello World! Thread ID, " << tid << endl;
   pthread_exit(NULL);
}

int main ()
{
   pthread_t threads[NUM_THREADS];
   int rc;
   int i;
   for( i=0; i < NUM_THREADS; i++ ){
  cout << "main() : creating thread, " << i << endl;
```

```
      rc = pthread_create(&threads[i], NULL,
      PrintHello, (void *)i);
      if (rc){
     cout << "Error:unable to create thread," << rc << endl;
     exit(-1);
      }
       }
       pthread_exit(NULL);
    }
```

使用 `-lpthread` 库编译上面的程序，如下所示：

```
$gcc test.cpp -lpthread
```

现在执行上面的程序，将会产生如下的结果：

```
main() : creating thread, 0
main() : creating thread, 1
main() : creating thread, 2
main() : creating thread, 3
main() : creating thread, 4
Hello World! Thread ID, 0
Hello World! Thread ID, 1
Hello World! Thread ID, 2
Hello World! Thread ID, 3
Hello World! Thread ID, 4
```

## 传递参数给线程

下面的例子展示了如何通过一个结构体传递多个参数。你可以在一个线程回调中传递任何数据类型，这是因为它指向 void 类型。

下面的例子解释了这一点：

```
#include <iostream>
#include <cstdlib>
#include <pthread.h>

using namespace std;

#define NUM_THREADS 5

struct thread_data{
   int  thread_id;
```

```
   char *message;
};

void *PrintHello(void *threadarg)
{
   struct thread_data *my_data;

   my_data = (struct thread_data *) threadarg;

   cout << "Thread ID : " << my_data->thread_id ;
   cout << " Message : " << my_data->message << endl;

   pthread_exit(NULL);
}

int main ()
{
   pthread_t threads[NUM_THREADS];
   struct thread_data td[NUM_THREADS];
   int rc;
   int i;

   for( i=0; i < NUM_THREADS; i++ ){
  cout <<"main() : creating thread, " << i << endl;
  td[i].thread_id = i;
  td[i].message = "This is message";
  rc = pthread_create(&threads[i], NULL,
  PrintHello, (void *)&td[i]);
  if (rc){
 cout << "Error:unable to create thread," << rc << endl;
 exit(-1);
  }
   }
   pthread_exit(NULL);
}
```

当上述代码编译和执行后，将会有以下的结果：

```
main() : creating thread, 0
main() : creating thread, 1
main() : creating thread, 2
main() : creating thread, 3
main() : creating thread, 4
Thread ID : 3 Message : This is message
Thread ID : 2 Message : This is message
Thread ID : 0 Message : This is message
```

```
Thread ID : 1 Message : This is message
Thread ID : 4 Message : This is message
```

## 连接和分离线程

下面的两个函数，我们可以用它们来连接或分离线程：

```
pthread_join (threadid, status)
pthread_detach (threadid)
```

`pthread_join ()` 子例程会阻塞调用它的线程，一直等到其指定的 threadid 的线程结束为止。当一个线程创建后，它的属性决定了它是否是可连接的或可分离的。只有创建时属性为可连接的线程才可以连接。如果创建的是一个可分离的线程，那么它永远不能连接。

下面的例子演示了如何使用 pthread_join 函数来等待一个线程结束。

```
#include <iostream>
#include <cstdlib>
#include <pthread.h>
#include <unistd.h>

using namespace std;

#define NUM_THREADS 5

void *wait(void *t)
{
   int i;
   long tid;

   tid = (long)t;

   sleep(1);
   cout << "Sleeping in thread " << endl;
   cout << "Thread with id : " << tid << "  ...exiting " << endl;
   pthread_exit(NULL);
}

int main ()
{
   int rc;
   int i;
   pthread_t threads[NUM_THREADS];
   pthread_attr_t attr;
```

```
   void *status;

   // Initialize and set thread joinable
   pthread_attr_init(&attr);
   pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

   for( i=0; i < NUM_THREADS; i++ ){
  cout << "main() : creating thread, " << i << endl;
  rc = pthread_create(&threads[i], NULL, wait, &amp;&amp;i );
  if (rc){
 cout << "Error:unable to create thread," << rc << endl;
 exit(-1);
  }
   }

   // free attribute and wait for the other threads
   pthread_attr_destroy(&attr);
   for( i=0; i < NUM_THREADS; i++ ){
  rc = pthread_join(threads[i], &status);
  if (rc){
 cout << "Error:unable to join," << rc << endl;
 exit(-1);
  }
  cout << "Main: completed thread id :" << i ;
  cout << "  exiting with status :" << status << endl;
   }

   cout << "Main: program exiting." << endl;
   pthread_exit(NULL);
}
```

当上述代码编译和执行后，将产生以下的结果：

```
main() : creating thread, 0
main() : creating thread, 1
main() : creating thread, 2
main() : creating thread, 3
main() : creating thread, 4
Sleeping in thread
Thread with id : 0 .... exiting
Sleeping in thread
Thread with id : 1 .... exiting
Sleeping in thread
Thread with id : 2 .... exiting
Sleeping in thread
Thread with id : 3 .... exiting
```

```
Sleeping in thread
Thread with id : 4 .... exiting
Main: completed thread id :0  exiting with status :0
Main: completed thread id :1  exiting with status :0
Main: completed thread id :2  exiting with status :0
Main: completed thread id :3  exiting with status :0
Main: completed thread id :4  exiting with status :0
Main: program exiting.
```

# Web 编程

## 什么是 CGI ？

- 通用网关接口（Common Gateway Interface），简称 CGI，是一组定义如何将信息在 Web 服务器和一个自定义脚本之间交换的标准。
- 目前 CGI 的规范是由 NCSA 维护的；NCSA 定义 CGI 如下：
- 通用网关接口（CGI），是外部网关程序与信息服务器（例如，HTTP 服务器）之间的接口标准。
- 目前应用的版本是 CGI /1.1；CGI/1.2 版本正在开发中。

## 网页浏览

要了解 CGI 的概念，那我们先来看看，当我们点击超链接来浏览特定的网页或 URL 时会发生什么。

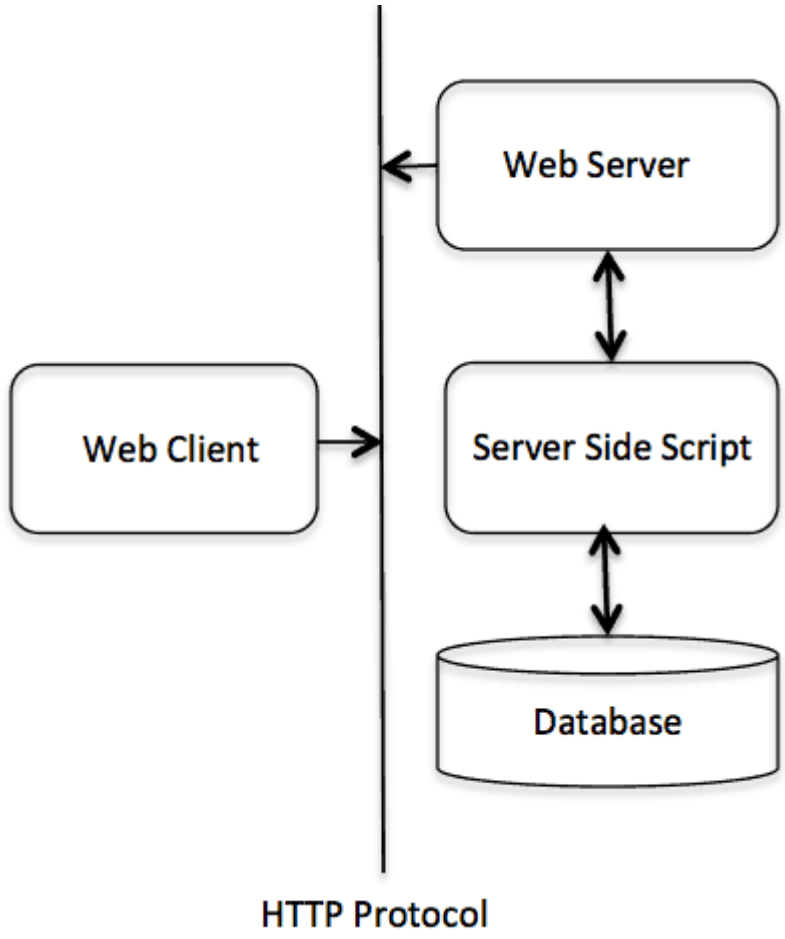
- 您的浏览器连接到 HTTP Web 服务器，并且请求 URL ，文件名。
- Web 服务器将会解析 URL，并且会查找文件名。如果找到所请求的文件，那么 Web 服务器会将该文件发送给浏览器，否则它会发送一个错误信息表明您请求了一个错误的文件。
- Web 浏览器的响应来自于 Web 服务器，并且浏览器要显示所接收到的文件或者错误消息的响应。

然而，也可以以这样的方式来设置 HTTP 服务器：每当在某个目录中的文件被请求时，该文件并不被发送回；代替的是，这个请求作为一个程序来被执行，由该程序产生输出，并被发送回给浏览器来显示。

公共网关接口（CGI）是一个能使应用程序（称为 CGI 程序或 CGI 脚本）与 Web 服务器和客户端进行交互的标准协议。这些 CGI 程序可以使用 Python，Perl，Shell，C 或 C++ 等语言来编写。

## CGI 架构图

下图简单的展示了 CGI 程序的一个简单架构:




## Web 服务器配置

在你进行 CGI 编程之前确保您的 Web 服务器支持 CGI，并且它被配置为能够处理 CGI 程序。 HTTP 服务器执行的所有 CGI 程序被保存在一个预先配置的目录里。这个目录称为 CGI 目录，一般其命名为 `/var/www/cgi-bin` 。尽管 CGI 文件是 C++ 可执行的，但是按照惯例他们会有扩展名 .cgi.

默认情况下，Apache Web 服务器配置为在 `/var/www/cgi-bin` 运行 CGI 程序。如果你想指定其他目录运行 CGI 脚本，你可以在 httpd.conf 配置文件中修改下面部分的内容:

```
<Directory "/var/www/cgi-bin">
   AllowOverride None
   Options ExecCGI
   Order allow,deny
   Allow from all
</Directory>

<Directory "/var/www/cgi-bin">
```

```
    Options All
    </Directory>
```

在这里，我们假定你有 Web 服务器，启动并运行成功，并且你可以运行任何其他语言（比如 Perl 或 Shell 等）所写的 CGI 程序。

## 第一个 CGI 程序

思考下面的 C++ 程序内容:

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

int main ()
{

   cout << "Content-type:text/html\r\n\r\n";
   cout << "<html>\n";
   cout << "<head>\n";
   cout << "<title>Hello World - First CGI Program</title>\n";
   cout << "</head>\n";
   cout << "<body>\n";
   cout << "<h2>Hello World! This is my first CGI program</h2>\n";
   cout << "</body>\n";
   cout << "</html>\n";

   return 0;
}
```

编译上面的代码，并命名可执行文件为 cplusplus.cgi. 该文件被保存在 /var/WWW/cgi-bin 目录里，并且它有上面的内容。在运行 CGI 程序之前，请确保你已使用 chmod755 cplusplus.cgi UNIX 命令来改变文件的权限，使文件能够可执行。现在，如果你点击 cplusplus.cgi ，那么这将产生以下的输出：

```
Hello World! This is my first CGI program
```

以上的 C++ 程序是一个简单的将输出写入 STDOUT 文件（即屏幕）的程序。它有一个可用的重要和额外的功能是，第一行要输出 Content-type:text/html\r\n\r\n. 这一行被发送回浏览器，并指定在浏览器屏幕上显示出来的内容类型。 现在，你理解了 CGI 的基本概念，然后你就可以使用 Python 写许多复杂的 CGI 程序。一个 C++ 的 CGI 程序可以与任何其他外部系统(例如 RDBMS )交互，从而来进行信息的交换。

## HTTP 报头

Content-type:text/html\r\n\r\n 是发送到浏览器的 HTTP 报头的一部分，它用来帮助浏览器解析文本内容。下面的表格展示了 HTTP 报头

```
HTTP Field Name: Field Content

For Example
Content-type: text/html\r\n\r\n
```

还有其他一些你可能经常会在 CGI 编程中使用的重要HTTP报头。

| 报头 | 描述 |
| --- | --- |
| Content-type: | 一个MIME字符串格式的文件会被返回。例如：Content-type:text/htm |
| Expires: Date | Date：页面信息失效的日期。这运行在浏览器端，由浏览器决定何时需要刷新页面。一个有效的日期字符串格式应该是这样的： 01 Jan 1998 12:00:00 GMT. |
| Location: URL | 这里的 URL 是应当返回的 URL，而不是请求的 URL.你可以使用此来重定向一个请求到任何文件。 |
| Last-modified: Date | 资源最近修改的日期。 |
| Content-length: N | 返回数据的长度，以字节为单位存储。浏览器使用这个值来预估计文件的下载时间。 |
| Set-Cookie: String | 通过字符串 string 设置 cookie |

## CGI 环境变量

所有 CGI 程序可使用下列环境变量。这些变量在编写 CGI 程序时，发挥着重要的作用。

| 变量名 | 描述 |
| --- | --- |
| CONTENT_TYPE | 内容的数据类型。当客户端发送连接内容到服务器时使用。例如文件上传等。 |
| CONTENT_LENGTH | 查询信息的长度。它仅适用于 POST 请求。 |
| HTTP_COOKIE | 以键值对的形式返回设置的 cookies 。 |
| HTTP_USER_AGENT | 用户代理请求头字段包含发起请求的用户代理的有关信息，包括 Web 浏览器的名称。 |
| PATH_INFO | CGI 脚本的路径信息。 |
| QUERY_STRING | 与发送 GET 请求一同发送的 URL 编码信息。 |
| REMOTE_ADDR | 发出请求的远程主机的 IP 地址。这可用于记录日志或认证。 |
| REMOTE_HOST | 发出请求的主机的完全资格名称。如果该信息不可用，则REMOTE_ADDR 可用于获取 IR 地址。 |
| REQUEST_METHOD | 用于请求的方法。最常用的方法是 GET 和 POST. |

| | |
|---|---|
| SCRIPT_FILENAME | CGI 脚本的完全路径。 |
| SCRIPT_NAME | CGI 脚本的名称。 |
| SERVER_NAME | 服务器的主机名或是 IP 地址。 |
| SERVER_SOFTWARE | 服务器正在运行软件的名称和版本。 |

这里提供一个小 CGI 程序来列出所有的 CGI 变量。点击链接 Get Environment 来查看结果。

```
#include <iostream>
#include <stdlib.h>
using namespace std;

const string ENV[ 24 ] = {
"COMSPEC", "DOCUMENT_ROOT", "GATEWAY_INTERFACE",
"HTTP_ACCEPT", "HTTP_ACCEPT_ENCODING",
"HTTP_ACCEPT_LANGUAGE", "HTTP_CONNECTION",
"HTTP_HOST", "HTTP_USER_AGENT", "PATH",
"QUERY_STRING", "REMOTE_ADDR", "REMOTE_PORT",
"REQUEST_METHOD", "REQUEST_URI", "SCRIPT_FILENAME",
"SCRIPT_NAME", "SERVER_ADDR", "SERVER_ADMIN",
"SERVER_NAME","SERVER_PORT","SERVER_PROTOCOL",
"SERVER_SIGNATURE","SERVER_SOFTWARE" };

int main ()
{

   cout << "Content-type:text/html\r\n\r\n";
   cout << "<html>\n";
   cout << "<head>\n";
   cout << "<title>CGI Envrionment Variables</title>\n";
   cout << "</head>\n";
   cout << "<body>\n";
   cout << "<table border = \"0\" cellspacing = \"2\">";

   for ( int i = 0; i < 24; i++ )
   {
   cout << "<tr><td>" << ENV[ i ] << "</td><td>";
   // attempt to retrieve value of environment variable
   char *value = getenv( ENV[ i ].c_str() );
   if ( value != 0 ){
 cout << value;
   }else{
 cout << "Environment variable does not exist.";
   }
   cout << "</td></tr>\n";
   }
```

```
    cout << "</table><\n";
    cout << "</body>\n";
    cout << "</html>\n";

    return 0;
}
```

## C++ CGI 库

对于真正的应用来说，你需要你的CGI程序来完成许多操作。现在有一个用 C++ 编写的 CGI 程序库，你可以从 ftp://ftp.gnu.org/gnu/cgicc/ 下载并按照以下步骤来安装此库：

```
$tar xzf cgicc-X.X.X.tar.gz
$cd cgicc-X.X.X/
$./configure --prefix=/usr
$make
$make install
```

你可以在 C++ CGI Lib Documentation 上查看相关文档。

## GET 和 POST 方法

你一定遇到过很多将信息数据从浏览器发送到 Web 服务器，并最终发送到 CGI 程序的情况。浏览器最常使用 GET 和 POST 两种方法将信息传送给 Web 服务器。

## 利用 GET 方法传递数据

GET 方法将编码的用户信息附加到页面请求后面，页面和编码信息用字符？分割开。如下所示：

```
http://www.test.com/cgi-bin/cpp.cgi?key1=value1&key2=value2
```

GET 方法是将信息数据从浏览器传递到 Web 服务器的默认方法。在浏览器的地址栏里面，此方法会产生一个长长的字符串。如果你有密码或其他敏感信息要传递给服务器，千万不要使用 GET 方法。 GET 方法有大小限制，在请求字符串里面，最多可以有 1024 个字符。

当使用 GET 方法时，信息数据使用 HTTP 报头 QUERY_STRING 传递，同时你的 CGI 程序通过访问 QUERY_STRING 环境变量来获取信息数据。

你可以通过简单地串联键值对和任何 URL 来传递信息数据，或者可以使用 HTML<FORM> 标签，通过利用 GET 方法来传递信息。

## 简单的 URL 例子： GET 方法

下面是一个使用 GET 方法传递两个值到 hello_get.py 程序的简单 URL。

/cgi-bin/cpp_get.cgi?first_name=ZARA&last_name=ALI

下面是一个生成 cpp_get.cgi CGI 程序的程序，它会处理来自 Web 浏览器的输入。我们使用了 C++ CGI 库，这使得它非常容易访问传递的信息：

```
#include <iostream>
#include <vector>
#include <string>
#include <stdio.h>
#include <stdlib.h>

#include <cgicc/CgiDefs.h>
#include <cgicc/Cgicc.h>
#include <cgicc/HTTPHTMLHeader.h>
#include <cgicc/HTMLClasses.h>

using namespace std;
using namespace cgicc;

int main ()
{
   Cgicc formData;

   cout << "Content-type:text/html\r\n\r\n";
   cout << "<html>\n";
   cout << "<head>\n";
   cout << "<title>Using GET and POST Methods</title>\n";
   cout << "</head>\n";
   cout << "<body>\n";

   form_iterator fi = formData.getElement("first_name");
   if( !fi->isEmpty() && fi != (*formData).end()) {
  cout << "First name: " << **fi << endl;
   }else{
  cout << "No text entered for first name" << endl;
   }
   cout << "<br/>\n";
   fi = formData.getElement("last_name");
   if( !fi->isEmpty() &&fi != (*formData).end()) {
```

```
   cout << "Last name: " << **fi << endl;
    }else{
   cout << "No text entered for last name" << endl;
    }
    cout << "<br/>\n";

    cout << "</body>\n";
    cout << "</html>\n";

    return 0;
}
```

现在，编译上面的程序，如下所示：

```
$g++ -o cpp_get.cgi cpp_get.cpp -lcgicc
```

它将会产生 `cpp_get.cgi`，并把它放在你的 CGI 目录，并尝试使用下面的链接访问：

/cgi-bin/cpp_get.cgi?first_name=ZARA&last_name=ALI

这将会产生如下的结果：

```
First name: ZARA
Last name: ALI
```

## 简单的 FORM 例子： GET 方法

下面是一个使用 HTML FORM 和提交按钮来传递两个值的简单例子。我们将使用同一个 CGI 脚本 cpp_get.cgi 来处理此输入。

```
<form action="/cgi-bin/cpp_get.cgi" method="get">
First Name: <input type="text" name="first_name">  <br />

Last Name: <input type="text" name="last_name" />
<input type="submit" value="Submit" />
</form>
```

## 利用 POST 方法传递数据

将信息数据传递给 CGI 程序，一般比较可靠的方法是 POST 方法。它会以与 GET 方法完全相同的方式将数据信息打包，但不是将其作为在 URL 中？后面的一个文本字符串来发送，而是将它作为一个单独的消息发送。此消息以标准输入的形式的发送到 CGI 脚本。

同样，利用同一个 cpp_get.cgi 程序来处理 POST 方法的输入。我们用与上面一样的例子，利用 HTML FORM 和提交按钮来传递两个值，但是这一次用 POST 方法，如下所示：

```
<form action="/cgi-bin/cpp_get.cgi" method="post">
First Name: <input type="text" name="first_name"><br />
Last Name: <input type="text" name="last_name" />

<input type="submit" value="Submit" />
</form>
```

## 传递复选框数据到 CGI 程序

当有不止一个选项需要被选择的时候，要使用复选框。

下面是一个有两个复选框的 HTML 表单样例代码。

```
<form action="/cgi-bin/cpp_checkbox.cgi"
 method="POST"
 target="_blank">
<input type="checkbox" name="maths" value="on" /> Maths
<input type="checkbox" name="physics" value="on" /> Physics
<input type="submit" value="Select Subject" />
</form>
```

下面 C++ 程序会产生 `cpp_checkbox.cgi` 脚本来处理由 Web 浏览器提供的复选框按钮的输入。

```
#include <iostream>
#include <vector>
#include <string>
#include <stdio.h>
#include <stdlib.h>

#include <cgicc/CgiDefs.h>
#include <cgicc/Cgicc.h>
#include <cgicc/HTTPHTMLHeader.h>
#include <cgicc/HTMLClasses.h>

using namespace std;
using namespace cgicc;

int main ()
{
   Cgicc formData;
   bool maths_flag, physics_flag;
```

```
   cout << "Content-type:text/html\r\n\r\n";
   cout << "<html>\n";
   cout << "<head>\n";
   cout << "<title>Checkbox Data to CGI</title>\n";
   cout << "</head>\n";
   cout << "<body>\n";

   maths_flag = formData.queryCheckbox("maths");
   if( maths_flag ) {
  cout << "Maths Flag: ON " << endl;
   }else{
  cout << "Maths Flag: OFF " << endl;
   }
   cout << "<br/>\n";

   physics_flag = formData.queryCheckbox("physics");
   if( physics_flag ) {
  cout << "Physics Flag: ON " << endl;
   }else{
  cout << "Physics Flag: OFF " << endl;
   }
   cout << "<br/>\n";
   cout << "</body>\n";
   cout << "</html>\n";

   return 0;
}
```

## 传递单选按钮数据到 CGI 程序

当仅需要选择一个选项时，使用单选按钮。

下面是一个有两个单选按钮的 HTML 表单样例代码。

```
<form action="/cgi-bin/cpp_radiobutton.cgi"
 method="post"
 target="_blank">
<input type="radio" name="subject" value="maths"
checked="checked"/> Maths
<input type="radio" name="subject" value="physics" /> Physics
<input type="submit" value="Select Subject" />
</form>
```

下面的 C++ 程序会产生 cpp_checkbox.cgi 脚本来处理由 Web 浏览器提供的单选按钮的输入。

```
#include <iostream>
#include <vector>
#include <string>
#include <stdio.h>
#include <stdlib.h>

#include <cgicc/CgiDefs.h>
#include <cgicc/Cgicc.h>
#include <cgicc/HTTPHTMLHeader.h>
#include <cgicc/HTMLClasses.h>

using namespace std;
using namespace cgicc;

int main ()
{
   Cgicc formData;

   cout << "Content-type:text/html\r\n\r\n";
   cout << "<html>\n";
   cout << "<head>\n";
   cout << "<title>Radio Button Data to CGI</title>\n";
   cout << "</head>\n";
   cout << "<body>\n";

   form_iterator fi = formData.getElement("subject");
   if( !fi->isEmpty() && fi != (*formData).end()) {
  cout << "Radio box selected: " << **fi << endl;
   }

   cout << "<br/>\n";
   cout << "</body>\n";
   cout << "</html>\n";

   return 0;
}
```

## 传递文本域数据到 CGI 程序

当有多行文本需要传递到 CGI 程序时，要使用文本域元素。

下面是一个有文本域框的 HTML 表单样例代码。

```
<form action="/cgi-bin/cpp_textarea.cgi"
 method="post"
 target="_blank">
<textarea name="textcontent" cols="40" rows="4">
Type your text here...
</textarea>
<input type="submit" value="Submit" />
</form>
```

下面的 C++ 程序会产生 cpp_checkbox.cgi 脚本来处理由 Web 浏览器提供的文本域的输入。

```
#include <iostream>
#include <vector>
#include <string>
#include <stdio.h>
#include <stdlib.h>

#include <cgicc/CgiDefs.h>
#include <cgicc/Cgicc.h>
#include <cgicc/HTTPHTMLHeader.h>
#include <cgicc/HTMLClasses.h>

using namespace std;
using namespace cgicc;

int main ()
{
   Cgicc formData;

   cout << "Content-type:text/html\r\n\r\n";
   cout << "<html>\n";
   cout << "<head>\n";
   cout << "<title>Text Area Data to CGI</title>\n";
   cout << "</head>\n";
   cout << "<body>\n";

   form_iterator fi = formData.getElement("textcontent");
   if( !fi->isEmpty() && fi != (*formData).end()) {
  cout << "Text Content: " << **fi << endl;
   }else{
  cout << "No text entered" << endl;
   }
```

```
   cout << "<br/>\n";
   cout << "</body>\n";
   cout << "</html>\n";

   return 0;
}
```

## 传递下拉框数据到 CGI 程序

当有多个可选项，但是只能选择一个或者两个选项时，需要使用下拉框。

下面是一个有一个下拉框的 HTML 表单样例代码。

```
<form action="/cgi-bin/cpp_dropdown.cgi"
   method="post" target="_blank">
<select name="dropdown">
<option value="Maths" selected>Maths</option>
<option value="Physics">Physics</option>
</select>
<input type="submit" value="Submit"/>
</form>
```

下面的 C++ 程序会产生 cpp_checkbox.cgi 脚本来处理由 Web 浏览器提供的下拉框的输入。

```
#include <iostream>
#include <vector>
#include <string>
#include <stdio.h>
#include <stdlib.h>

#include <cgicc/CgiDefs.h>
#include <cgicc/Cgicc.h>
#include <cgicc/HTTPHTMLHeader.h>
#include <cgicc/HTMLClasses.h>

using namespace std;
using namespace cgicc;

int main ()
{
   Cgicc formData;

   cout << "Content-type:text/html\r\n\r\n";
   cout << "<html>\n";
```

```
    cout << "<head>\n";
    cout << "<title>Drop Down Box Data to CGI</title>\n";
    cout << "</head>\n";
    cout << "<body>\n";

    form_iterator fi = formData.getElement("dropdown");
    if( !fi->isEmpty() && fi != (*formData).end()) {
   cout << "Value Selected: " << **fi << endl;
    }

    cout << "<br/>\n";
    cout << "</body>\n";
    cout << "</html>\n";

    return 0;
}
```

## 在 CGI 中使用 Cookies

HTTP 协议是一个无状态协议。但是对于一个商业网站，它需要保持不同的页面间的会话信息。例如，一个用户的注册需要完成许多的页面，那么将如何保持所有网页之间的用户的会话信息将是一个问题。

在许多情况下，使用 cookies 是记录和跟踪，喜好，所购物品，佣金或其他能提供更好访问体验的信息或是网站的统计数据的最有效的方法。

## 它是如何工作的

你的服务器发送一些 cookie 格式的数据给访问者的浏览器。该浏览器可能接受了该 cookie. 如果是这样，它以纯文本的方式记录在访问者的硬盘驱动器上。现在，当访问者访问你的网站上的其他页面时，该 cookie 可用于检索。一旦检索到，你的服务器就会知道/记起已存储的信息。

Cookies 是一个记录了5可变长度字段的纯文本数据：

- Expires：cookie 将失效的日期。 如果此字段为空，那么当访问者退出浏览器后 cookie 将会失效。
- Domain ：网站的域名。
- Path ：设置 cookie 的目录或网页的路径。如果你想要从任何目录或页面检索 cookie ，此字段将设置为空。

- Secure ：如果该字段包含单词 "secure"，那么该 cookie 仅可由一个安全的服务器检索到。如果该字段为空，将不存在这样的限制。
- Name=Value：cookie 将以键值对的形式设置和检索。

## 设置 Cookies

将 cookies 发送到浏览器是非常容易的。这些 cookie 会设置在 HTTP报头的 Content-type 字段之前，并与其一起发送出去。假设你要将 UserID 和 Password 设置为 cookie. cookie 的设置如下所示

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

int main ()
{

   cout << "Set-Cookie:UserID=XYZ;\r\n";
   cout << "Set-Cookie:Password=XYZ123;\r\n";
   cout << "Set-Cookie:Domain=www.tutorialspoint.com;\r\n";
   cout << "Set-Cookie:Path=/perl;\n";
   cout << "Content-type:text/html\r\n\r\n";

   cout << "<html>\n";
   cout << "<head>\n";
   cout << "<title>Cookies in CGI</title>\n";
   cout << "</head>\n";
   cout << "<body>\n";

   cout << "Setting cookies" << endl;

   cout << "<br/>\n";
   cout << "</body>\n";
   cout << "</html>\n";

   return 0;
}
```

从这个例子中,你一定要了解如何设置 cookie。 我们使用 HTTP 报头的 Set-Cookie 字段设置 cookie。

cookies 的属性，如 Expires， Domain 和 Path 是可选设置项。值得注意的是，cookies 设置在魔力代码行 Content-type:text/html\r\n\r\n 之前。

编译上面的程序将产生 setcookies.cgi，并尝试使用下面的链接来设置 cookie。它会在你的电脑上设置四个 cookie：

/cgi-bin/setcookies.cgi

## 检索 Cookie

检索所有设置的 cookie 是非常容易的。 cookie 存储在 CGI 环境变量的 HTTP_COOKIE 字段，它们的形式如下所示：

```
key1=value1;key2=value2;key3=value3....
```

下面是一个如何检索 cookie 的例子。

```
#include <iostream>
#include <vector>
#include <string>
#include <stdio.h>
#include <stdlib.h>

#include <cgicc/CgiDefs.h>
#include <cgicc/Cgicc.h>
#include <cgicc/HTTPHTMLHeader.h>
#include <cgicc/HTMLClasses.h>

using namespace std;
using namespace cgicc;

int main ()
{
   Cgicc cgi;
   const_cookie_iterator cci;

   cout << "Content-type:text/html\r\n\r\n";
   cout << "<html>\n";
   cout << "<head>\n";
   cout << "<title>Cookies in CGI</title>\n";
   cout << "</head>\n";
   cout << "<body>\n";
   cout << "<table border = \"0\" cellspacing = \"2\">";

   // get environment variables
   const CgiEnvironment& env = cgi.getEnvironment();
```

```
   for( cci = env.getCookieList().begin();
cci != env.getCookieList().end();
++cci )
   {
  cout << "<tr><td>" << cci->getName() << "</td><td>";
  cout << cci->getValue();
  cout << "</td></tr>\n";
   }
   cout << "</table><\n";

   cout << "<br/>\n";
   cout << "</body>\n";
   cout << "</html>\n";

   return 0;
}
```

现在，编译上面的程序将产生 getcookies.cgi，并且试图列出你电脑上所有可用的 cookies：

/cgi-bin/getcookies.cgi

这将会列出在上一节设置的四个 cookies 和其它你电脑上设置的 cookies。

```
UserID XYZ
Password XYZ123
Domain www.tutorialspoint.com
Path /perl
```

## 文件上传

要上传一个文件，HTML 表单必须将 enctype 属性设置为 **multipart/form-data**。 type 为 file 的 input 标签将会产生一个“选择文件”按钮。

```
<html>
<body>
   <form enctype="multipart/form-data"
action="/cgi-bin/cpp_uploadfile.cgi"
method="post">
   <p>File: <input type="file" name="userfile" /></p>
   <p><input type="submit" value="Upload" /></p>
   </form>
</body>
</html>
```

**注意：**对上面的例子，我们已经禁止了向我们的服务器上传文件。但是你可以在你自己的服务上实验以上的代码。

下面是一个用来处理文件上传的脚本 **cpp_uploadfile.cpp **：

```
#include <iostream>
#include <vector>
#include <string>
#include <stdio.h>
#include <stdlib.h>

#include <cgicc/CgiDefs.h>
#include <cgicc/Cgicc.h>
#include <cgicc/HTTPHTMLHeader.h>
#include <cgicc/HTMLClasses.h>

using namespace std;
using namespace cgicc;

int main ()
{
   Cgicc cgi;

   cout << "Content-type:text/html\r\n\r\n";
   cout << "<html>\n";
   cout << "<head>\n";
   cout << "<title>File Upload in CGI</title>\n";
   cout << "</head>\n";
   cout << "<body>\n";

   // get list of files to be uploaded
   const_file_iterator file = cgi.getFile("userfile");
   if(file != cgi.getFiles().end()) {
  // send data type at cout.
  cout << HTTPContentHeader(file->getDataType());
  // write content at cout.
  file->writeToStream(cout);
   }
   cout << "<File uploaded successfully>\n";
   cout << "</body>\n";
   cout << "</html>\n";

   return 0;
}
```

上面的例子是在 cout 流中写内容，但是你可以打开你的文件流，并将上传文件的内容保存到一个预定位置的文件中。