C++基本内容

环境设置

本地环境设置

设置 C++ 语言环境需要确保电脑上有**文本编辑器**和 **C++ 编译器。**

文本编辑器

用于输入程序。

通过**编辑器创建的文件**通常称为**源文件**，源文件包含**程序源代码。**C++ 程序的源文件通常使用扩展名 .cpp、.cp 或 .c。

C++ 编译器

写在源文件中的源代码是人类可读的源。它**需要"编译"**，转为**机器语言**，这样 CPU 可以按给定指令执行程序。

C++ 编译器用于把源代码编译成最终的可执行程序。

大多数的 C++ 编译器并不在乎源文件的扩展名，但是如果您未指定扩展名，则默认使用 .cpp。

基本语法

基本语法

C++ 程序可以定义为**对象的集合**，这些对象通过调用彼此的方法进行交互。

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

程序结构

实例

```
#include<iostream>

using namespace std;

int main()
{
        cout<<"Hello!";
        return 0;
}
```

对于上面这段程序：

C++ 语言定义了一些头文件，这些头文件包含了程序中必需的或有用的信息。上面这段程序中，包含了头文件 **<iostream>**。

下一行 **using namespace std;** 告诉编译器使用 std 命名空间。命名空间是 C++ 中一个相对新的概念。

下一行 **// main() 是程序开始执行的地方** 是一个单行注释。单行注释以 // 开头，在行末结束。

下一行 **int main()** 是主函数，程序从这里开始执行。

下一行 **cout << "Hello World";** 会在屏幕上显示消息 "Hello World"。

下一行 **return 0;** 终止 main( )函数，并向调用进程返回值 0。

分号 & 语句块

分号

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

分号可单独作为一个**空语句**

例如，下面是三个不同的语句：

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

语句块

语句块是一组使用**大括号括起来**的按逻辑连接的语句。例如：

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

C++ 不以行末作为结束符的标识，因此，可以在一行上放置多个语句。例如：

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

等同于

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

标识符

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

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

下面列出几个有效的标识符：

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

注释

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

C++ 支持单行注释和多行注释。注释中的所有字符会被 C++ 编译器忽略。

单行注释

注释以 **//** 开始，直到行末为止。例如：

```
#include<iostream>
using namespace std;
int main()
{
        //这是一个注释
```

```
        cout<<"Hello World!";
        return 0;
}
```

也可以放在语句后面：

```
#include<iostream>
using namespace std;
int main()
{
        cout<<"Hello World!";////这是一个注释
        return 0;
}
```

当上面的代码被编译时，编译器会忽略 **// 这是一个注释** 和 **// 输出 Hello World!**，最后会产生以下结果：

```
Hello World!
```

多行注释

C++ 注释以 **/*** 开始，以 ***/** 终止。例如：

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

int main() {
    /* 这是注释 */

    /* C++ 注释也可以
     * 跨行
     */
    cout << "Hello World!";
    return 0;
}
```

注释嵌套

在 **/*** 和 ***/** 注释内部，**//** 字符没有特殊的含义。

在 **//** 注释内，**/*** 和 ***/** 字符也没有特殊的含义。

因此，您可以在一种注释内嵌套另一种注释。例如：

```
/* 用于输出 Hello World 的注释

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

*/
```

关于头文件

一般而言，头文件会包含**函数原型**、define和const定义的**符号常量**、**结构与类的声明**、**模板声明**、**内联函数定义**。

函数定义和**变量声明**不能放在**头文件**中

如果将函数定义和变量声明放在头文件中，当同属于一个程序的两个文件包含这个头文件编译时，就会出现一个程序有相同的两个函数定义的情况，引起出错。
双引号括起来的头文件指示编译器优先在当前工作目录查找

include**不能包含源代码文件**，会导致多重声明

基本类型

基本类型

C++ 为程序员提供了种类丰富的内置数据类型和用户自定义的数据类型。

数据被分为不同的类型，分别以不同的方式进行存储。

每一种数据类型可以看成由两个集合构成：**值集和运算集**。

值集：描述该数据类型包含**哪些值**

运算集：描述对值集中的值可以**进行哪些运算**。

如整型的值集是由一定范围的整数构成的集合，运算集包括加减乘除余等。

下表列出了六种基本的 C++ 数据类型：

类型	关键字
布尔型	bool
字符型	char/wchar_t(Unicode,2-4字节)
整型	int
浮点型	float
双浮点型	double
无类型	void

一些基本类型可以使用一个或多个类型修饰符进行修饰：

- unsigned
- short
- long

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

注意：long int 8 个字节，int 都是 4 个字节，早期的 C 编译器定义了 long int 占用 4 个字节，int 占用 2 个字节，新版的 C/C++ 标准兼容了早期的这一设定。

类型	位	范围
char	1 个字节	-128 到 127 或者 0 到 255
unsigned char	1 个字节	0 到 255
signed char	1 个字节	-128 到 127
int	4 个字节	-2147483648 到 2147483647
unsigned int	4 个字节	0 到 4294967295
signed int	4 个字节	-2147483648 到 2147483647
short int	2 个字节	-32768 到 32767
unsigned short int	2 个字节	0 到 65,535
signed short int	2 个字节	-32768 到 32767
long int	8 个字节	-9,223,372,036,854,775,808 到 9,223,372,036,854,775,807
signed long int	8 个字节	-9,223,372,036,854,775,808 到 9,223,372,036,854,775,807
unsigned long int	8 个字节	0 到 18,446,744,073,709,551,615
float	4 个字节	精度型占4个字节（32位）内存空间，+/- 3.4e +/- 38 (~7 个数字)
double	8 个字节	双精度型占8 个字节（64位）内存空间，+/- 1.7e +/- 308 (~15 个数字)
long double	16 个字节	长双精度型 16 个字节（128位）内存空间，可提供18-19位有效数字。
wchar_t	2 或 4 个字节	1 个宽字符

类型转换

类型转换是将**一个数据类型**的值转换为**另一种数据类型**的值。

隐式转换（自动转换）

当不同数据类型的数据混合运算时，编译系统自动将其调整为**同一数据类型。**

1. **无条件的隐式类型转换**：char型和short型先转化为int
2. **同一类型的隐式类型转换**：将**精度较低**的数据类型转为**精度较高、范围较大**的类型

   即int→unsigned→long→float→double
3. **赋值时进行的隐式类型转换**：对变量赋值时，表达式计算的**最终结果**会被转换成**被赋值变量的类型**，

   这样的过程可能导致**类型升级或降级。**降级将会带来**精度的损失。**

例如当double赋值给float时，会四舍五入；要求保留几位小数输出时也会四舍五入。

C中的显式类型转换

格式为：

（类型名）数据

1. 从表示范围大的类型强制转换到表示范围小的类型，可能会**丢失精度。**
2. 类型转换不会改变变量**本身的数据类型和值**，只**对其进行的运算有影响。**

C++类型转换符

dynamic_cast

```
dynamic_cast <type-name> argument
```

当dynamic_cast检测到一个**派生类指针或引用**为一个**基类指针或引用赋值**时，它会返回一个**空指针**

const_cast

```
const_cast <type-name> argument
```

当**const_cast**只允许**const**或**volatile**的改变，如果有其他类型转换就会报错。

如允许一个**const int**转为**volatile int**，但是不允许一个**const int**转为**double**

static_cast

```
static_cast <type-name> argument
```

static_cast只允许在**隐式转换**能进行的情况的**显式转换**

reinterpret_cast

```
reinterpret_cast <type-name> argument
```

reinterpret_cast的功能有**指针**和**整数**之间的转换，**不同类型的指针/成员指针/引用**之间的转换

它的意义类似于C中的union，将同一处内存中的数据**作为指针**或**作为数据**解释

常量与变量

常量

常量就像是常规的变量，只不过常量的值在定义后不能进行修改。

字面常量

也称**直接常量**。以字面值的形式直接出现在程序中。

整数常量

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

八进制和**十六进制**常量通常用于表示**长串的二进制数**，如存储地址。

负数转换进制时，**不管符号先化成二进制**，再**按位取反**，再**加1**，再换成**对应的进制**。

整数常量也可以带一个**后缀**，后缀是 **U 和 L** 的组合，U 表示**无符号整数（unsigned）**，L 表示**长整数（long）**。

后缀可以是大写，也可以是小写，U 和 L 的顺序任意。

下面列举几个整数常量的实例：

```
212         // 合法的
215u        // 合法的,一个无符号整数常量
0xFeeL      // 合法的，一个十六进制长整数常量
078         // 非法的：8 不是八进制的数字
032UU       // 非法的：不能重复后缀
```

以下是各种类型的整数常量的实例：

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

浮点常量

浮点常量由**整数部分**、**小数点**、**小数部分**和**指数部分**组成。您可以使用**小数形式**或者**指数形式**来表示浮点常量。

当使用小数形式表示时，必须包含**整数部分**、**小数部分**，或**同时包含两者**。

当小数点前或后的数为0时，0可以省略。如23.即23.0，.23即0.23，都将会被记为浮点数。

当使用**指数形式**表示时， 必须包含**小数点**、**指数**，或**同时包含两者**。类似**科学计数法**，但是**将×10改为e/E**。

如31.4记为0.314E + 2.其中**0.314被称为尾数，b为阶码（必须为整数）。**

e前面不能没有数字，且e后面的数字不能加小括号。

下面列举几个浮点常量的实例：

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

布尔常量

布尔常量有**true**和**false**，**true** 值代表真。**false** 值代表假。不应把 true 的值看成 1，把 false 的值看成 0。事实上，true指代**非零值**

字符常量

字符常量是括在**单引号**中。如果常量以**L**(仅当大写时）开头，则表示它是一个**宽字符常量**（例如 L'x'），此时它必须存储在 **wchar_t** 类型的变量中。

否则，它就是一个**窄字符常量**（例如 'x'），此时它可以存储在 **char** 类型的简单变量中。

小写字母与相应的**大写字母**ASCII码值**相差32**

赋给一个字符常量大于256的数，**整型输出**时会输出余数

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

下表列出了一些这样的转义序列码：

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

字符串常量

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

可以使用 **** 做分隔符，把一个很长的字符串常量进行分行。

下面的实例显示了一些字符串常量：

实例

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

int main() {
    string greeting = "hello, runoob";
 cout << greeting;
 cout << "\n";  // 换行符
  string greeting2 = "hello, <br />                        runoob";
 cout << greeting2;
 return 0;
}
```

```
hello, runoob
hello, runoob
```

定义常量

在 C++ 中，有两种简单的定义常量的方式：

- 使用 **#define** 预处理器。
- 使用 **const** 关键字。

#define 预处理器

下面是使用 **#define** 预处理器定义常量的形式：

```
#define identifier value
```

实例

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

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

int main()
{

   int area;

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

当上面的代码被编译和执行时，它会产生下列结果：

```
50
```

const常量

可以使用 **const** 前缀声明**指定类型的常量**，如下所示：

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

实例

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

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

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

当上面的代码被编译和执行时，它会产生下列结果：

```
50
```

通常建议把常量定义为**大写字母**形式

变量

变量简介

变量实质是程序可操作的**存储区的名称。**C++ 中**每个变量**都有指定的类型，类型决定了**变量存储的大小和布局**，

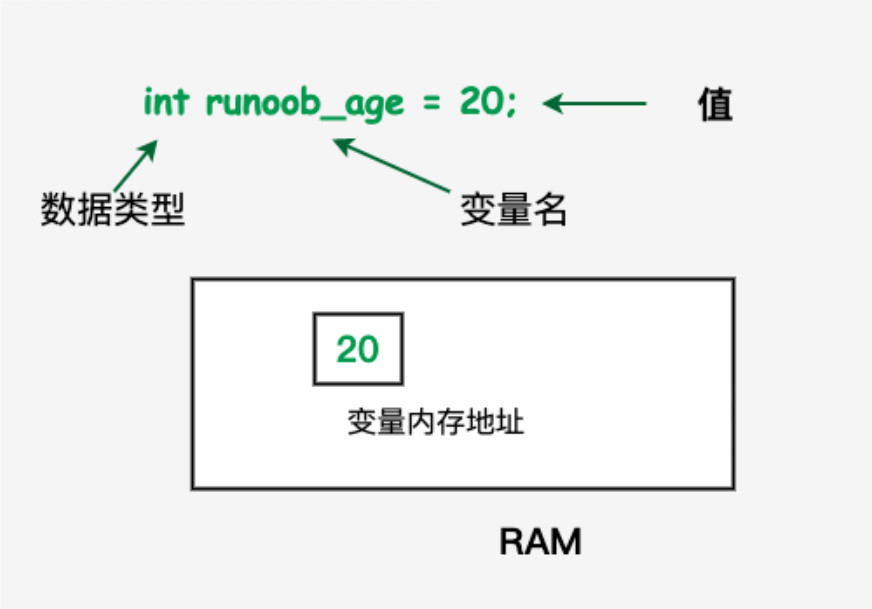
该范围内的值都可以存储在内存中，运算符可应用于变量上。

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

C++ 允许定义复杂类型的变量，比如**枚举、指针、数组、引用、数据结构、类**等等，接下来的变量内容以**基本类型**为例

变量定义

变量定义就是告诉编译器在**何处创建变量的存储**，以及**如何创建变量的存储。**



变量定义指定一个数据类型，并包含了该类型的一个或多个变量的列表，如下所示：

```
type variable_name = value;
```

在这里，**type** 必须是一个有效的 C++ 数据类型，可以是 char、wchar_t、int、float、double、bool 或任何用户自定义的对象，

variable_list 可以由一个或多个标识符名称组成，多个标识符之间用**逗号**分隔。

变量可以在声明的时候被**初始化**（**指定一个初始值**）。**初始化器**由一个等号，后跟一个**常量表达式**组成，如此处被初始化为value

初始化还可以用函数表示法，如int num(13)

不带初始化的定义

带有**静态存储持续时间**的变量会被隐式初始化为 **NULL**（所有字节的值都是 **0**），**其他所有变量**的初始值是**未定义**的，即可能存放一个**随机值**

定义变量的位置

一般来说有三个地方可以定义变量：

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

变量声明

变量声明向编译器保证变量以**给定的类型**和**名称**存在，这样编译器在不需要知道变量完整细节的情况下也能继续进一步的编译。

变量声明只在**编译时**有它的意义，在程序连接时编译器需要**实际的变量声明。**

当使用多个文件且只在**其中一个文件**中定义变量时（定义变量的文件在程序连接时是可用的），变量声明就显得非常有用。可以使用 **extern** 关键字在任何地方**声明一个变量。**

虽然可以在 C++ 程序中**多次声明一个变量**，但变量只能在某个文件、函数或代码块中被**定义一次。**

实例

变量在头部就已经被声明，但它们是在主函数内被定义和初始化的：

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

// 变量声明
extern int a, b;
extern int c;
extern float f;

int main ()
{
  // 变量定义
  int a, b;
  int c;
  float f;

  // 实际初始化
  a = 10;
  b = 20;
```

```
  c = a + b;

  cout << c << endl ;

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

  return 0;
```

当上面的代码被编译和执行时，它会产生下列结果：

```
30
23.3333
```

左值（Lvalues)和右值（Rvalues）

C++ 中有两种类型的表达式：

- **左值（lvalue）：指向内存位置**的表达式被称为**左值（lvalue）表达式。**左值可以出现在**赋值号的左边**或**右边。**
- **右值（rvalue）：**指的是**存储在内存中某些地址的数值。**右值是**不能对其进行赋值的表达式**，

也就是说，右值可以出现在赋值号的右边，但不能出现在赋值号的左边。

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

```
int g = 20;
```

但是下面这个就不是一个有效的语句，会生成编译时错误：

```
10 = 20;
```

变量作用域

如果在**内部作用域**中声明的变量与**外部作用域**中的变量同名，则**内部作用域**中的变量将**覆盖**外部作用域中的变量。

局部变量

在**函数**或**一个代码块内部**声明的变量，称为**局部变量**。它们只能被函数内部或者代码块内部的语句使用。

局部变量在函数每**次被调用时被创建**，在函数执行完后**被销毁**。下面的实例使用了局部变量：

```
#include <iostream>

using namespace std;

int main ()

{

  // 局部变量声明

  int a, b;

  int c;

  // 实际初始化

  a = 10;

  b = 20;

  c = a + b;

  cout << c;

  return 0;

}
```

全局变量

在**所有函数外部定义**的变量（通常是在**程序的头部**），称为全局变量。

全局变量**在程序开始时被创建**，在**程序结束时被销毁**。全局变量的值在程序的整个生命周期内都是有效的。

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

```
#include <iostream>

using namespace std;

// 全局变量声明

int g;

int main ()

{

  // 局部变量声明

  int a, b;

  // 实际初始化

  a = 10;

  b = 20;

  g = a + b;

  cout << g;

  return 0;

}
```

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

```
#include <iostream>

using namespace std;

// 全局变量声明

int g = 20;
```

```
int main ()

{

  // 局部变量声明

  int g = 10;

  cout << g;

  return 0;

}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
10
```

如果要在语句块中使用重名的全局变量，可以使用**域解析符**来表示使用的是一个全局变量

```
::x//全局变量
x//局部变量
```

但不建议用全局变量：

1. **影响函数之间的独立性**
2. 全局变量可能会被**各函数调用的时候改变其值**，出现错误难以排查。

初始化局部变量和全局变量

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

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

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

块作用域

块作用域指的是在**代码块内部**声明的变量，它们只能在**代码块内部**访问。块作用域变量在**代码块每次被执行时**被创建，在**代码块执行**完后被销毁。

```
#include <iostream>

int main() {

    int a = 10;

    {

        int a = 20;  // 块作用域变量

        std::cout << "块变量: " << a << std::endl;

    }

    std::cout << "外部变量: " << a << std::endl;

    return 0;

}
```

以上实例中，内部的代码块中声明了一个名为 a 的变量，它与外部作用域中的变量 a 同名。

内部作用域中的变量 a 将覆盖外部作用域中的变量 a，在内部作用域中访问 a 时输出的是20，而在外部作用域中访问 a 时输出的是 10。

当上面的代码被编译和执行时，它会产生下列结果：

```
块变量: 20
```

```
外部变量: 10
```

类作用域

类作用域指的是在**类内部声明的变量**，在**类内部**声明的变量具有**类作用域**，它们可以被**类的所有成员函数**访问。

类作用域变量的生命周期与**类的生命周期相同。**

```
#include <iostream>

class MyClass {

public:

    static int class_var;  // 类作用域变量

};

int MyClass::class_var = 30;

int main() {

    std::cout << "类变量: " << MyClass::class_var << std::endl;

    return 0;

}
```

以上实例中，MyClass 类中声明了一个名为 class_var 的类作用域变量。

可以使用类名和作用域解析运算符 :: 来访问这个变量。在 main() 函数中访问 class_var 时输出的是 30。

```
类变量: 30
```

命名空间作用域

在**命名空间**处提

文件作用域

是单独的一个**源文件**

在**全局标识符的定义**中加上**static修饰符**，则该全局标识符就成了具有**文件作用域**的**标识符**

它们只能在**定义它们的源文件**中使用。

具有**文件作用域**的标识符如果定义在**使用点之后**，则需要**声明。**

const定义的**全局常量**具有**文件作用域**，**不需要加static。**

```
//file1.cpp
static int y;         //文件作用域
static void f()     //文件作用域
{ ..... }

void g()
{  ... y ...             //OK
   f();                  //OK
}
//file2.cpp
extern int y;
extern void f();

void g()
{  ... y ...     //Error
   f();          //Error
}
```

变量生存期

变量占有内存空间的时间段称为**变量的生存期。**

静态生存期

内存空间**从程序开始执行时**就进行分配，直到程序结束才收回它们的空间。

全局变量具有**静态生存期。**

全局变量、static修饰的局部变量(具有静态生存期)和**常量**均会被分配在静态数据区

自动生存期

内存空间**在程序执行到定义它们的复合语句**（包括函数体）时**才分配**，当定义**它们的复合语句执行结束时**，它们的**空间将被收回。**

局部变量和**函数的参数**具有自动生存期。

局部变量和**函数参数**均会被分配在**栈区**

动态生存期

内存空间在程序中**显式地**用**new**操作或**malloc库函数分配**、用**delete操作**或**free库函数**收回。

动态变量具有**动态生存期。**

动态变量会被分配在**堆区**

变量存储类

存储类定义 C++ 程序中**变量/函数**的**范围（可见性）**和**生命周期。**这些**说明符**放置在**它们所修饰的类型之前。**

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

auto 存储类

自 C++ 11 以来，**auto** 关键字用于两种情况：

- 声明变量时**根据初始化表达式自动推断**该变量的**类型**
- 声明函数时**函数返回值**的**占位符**

C++98标准中auto关键字用于自动变量的声明，但由于使用极少且多余，在 C++17 中已删除这一用法。

如：

```
auto f=3.14;      //double
auto s("hello");  //const char*
auto z = new auto(9); // int*
auto x1 = 5, x2 = 5.0, x3='r';//错误，必须是初始化为同一类型
```

register 存储类

register 存储类用于定义**存储在寄存器中**局部变量。

这意味着**变量的最大尺寸**等于寄存器的大小（通常是一个词），且不能对它应用一元的 **'&' 运算符**（因为它没有内存位置）

```
register int  miles;
```

寄存器只用于**需要快速访问的变量**，比如**计数器。**

定义 'register' 并不意味着变量将被存储在寄存器中，它意味着**变量可能存储在寄存器**中，这**取决于硬件和实现的限制。**

static 存储类

static 存储类指示编译器**在程序的生命周期内保持局部变量的存在**，而不需要在每次它进入和离开作用域时进行**创建和销毁。**

因此，使用 static 修饰局部变量可以**在函数调用**之间保持局部变量的值。

- 在**局部变量**的定义中，static修饰符指定**局部变量**采用**静态存储分配**；
- 在**全局标识符**的定义中，static修饰符把**全局标识符**的作用域改变为**文件作用域。**
- 在**类数据成员的定义中**，static修饰符导致**仅有一个该成员的副本**被类的**所有对象**共享。

```
#include <iostream>
// 函数声明
void func(void);
static int count = 10; /* 全局变量 */
int main()
{
    while(count--)
    {
       func();
    }
    return 0;
}

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

```
    std::cout << " , 变量 count 为 " << count << std::endl;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
变量 i 为 6 , 变量 count 为 9
```

```
变量 i 为 7 , 变量 count 为 8
```

```
变量 i 为 8 , 变量 count 为 7
```

```
变量 i 为 9 , 变量 count 为 6
```

```
变量 i 为 10 , 变量 count 为 5
```

```
变量 i 为 11 , 变量 count 为 4
```

```
变量 i 为 12 , 变量 count 为 3
```

```
变量 i 为 13 , 变量 count 为 2
```

```
变量 i 为 14 , 变量 count 为 1
```

```
变量 i 为 15 , 变量 count 为 0
```

extern 存储类

extern 存储类用于提供一个全局变量的引用，**全局变量对所有的程序文件都是可见的。**

当使用 **'extern'** 时，对于**无法初始化的变量**，变量名将指向一个**之前定义过的存储位置。**

当有**多个文件**且定义了一个可以在其他文件中使用的**全局变量或函数(不能是局部)**时，可以在其他文件中使用 *extern* 来得到**已定义的变量或函数的引用。**

可以这么理解，*extern* 是用来在**另一个文件中声明一个全局变量或函数。**

extern声明变量**不允许同时初始化**，因为这只是一个声明

extern还可以用于提示编译器将源文件**按照C的规则进行编译**，即在源文件前加上**extern "C"**,从而实现**C/C++**的混合编程

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

```
第一个文件：main.cpp

#include <iostream>
int count ;
extern void write_extern();
int main()
{
   count = 5;
   write_extern();
}

第二个文件：support.cpp

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

在这里，第二个文件中的 *extern* 关键字用于声明已经在第一个文件 main.cpp 中定义的 count。现在，编译这两个文件，如下所示：

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

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

```
$ ./write
```

```
Count is 5
```

mutable 存储类

mutable 说明符仅适用于**类的对象**，它允许对象的**成员替代常量。**

也就是说，mutable 成员可以**通过 const 成员函数修改。**

```
class Example
{
        public:
    int get_value() const {
        return value_; // const 关键字表示该成员函数不会修改对象中的数据成员
    }
    void set_value(int value) const {
        value_ = value; // mutable 关键字允许在 const 成员函数中修改成员变量
    }
        private:
    mutable int value_;
};
```

thread_local 存储类

使用 **thread_local** 说明符声明的变量**仅可在它在其上创建的线程上访问。**

变量在**创建线程时创建**，并在**销毁线程时销毁。** 每个**线程**都有其自己的**变量副本。**

thread_local 说明符可以与 static 或 extern 合并。

thread_local **仅应用于数据声明和定义**，thread_local **不能用于函数声明或定义。**

以下演示了可以被声明为 thread_local 的变量：

```
thread_local int x;  // 命名空间下的全局变量

class X
{
    static thread_local std::string s; // 类的static成员变量
};
```

```cpp
static thread_local std::string X::s;  // X::s 是需要定义的

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

变量修饰符

修饰符是用于**改变变量类型的行为的关键字**，它更能满足各种情境的需求。

数据类型修饰符

- signed：表示**变量**可以**存储负数**。对于整型变量来说，signed 可以省略，因为整型变量默认为有符号类型。
- unsigned：表示**变量不能存储负数**。对于整型变量来说，unsigned 可以将**变量范围扩大一倍**。
- short：表示**变量的范围**比 int 更小。short int 可以缩写为 short。
- long：表示**变量的范围**比 int 更大。long int 可以缩写为 long。
- long long：表示**变量的范围**比 long 更大。C++11 中新增的数据类型修饰符。

修饰符 **signed、unsigned、long 和short**可应用于**整型**，**signed** 和 **unsigned** 可应用于**字符型**，**long** 可应用于**双精度**型。

这些修饰符也可以组合使用，修饰符 **signed** 和 **unsigned** 也可以作为 **long** 或 **short** 修饰符的**前缀**。例如：**unsigned long int**。

C++ 允许使用速记符号来声明**无符号短整数**或**无符号长整数**。

可以不写 int，只写单词 **unsigned、short** 或 **long**，**int** 是隐含的。例如，下面的两个语句都声明了无符号整型变量。

```cpp
signed int num1 = -10;           // 定义有符号整型变量 num1，初始值为 -10
unsigned int num2 = 20;          // 定义无符号整型变量 num2，初始值为 20

short int num1 = 10;              // 定义短整型变量 num1，初始值为 10
long int num2 = 100000;          // 定义长整型变量 num2，初始值为 100000

long long int num1 = 10000000000; // 定义长长整型变量 num1，初始值为 10000000000

float num1 = 3.14f;              // 定义单精度浮点数变量 num1，初始值为 3.14
double num2 = 2.71828;           // 定义双精度浮点数变量 num2，初始值为 2.71828
```

```
bool flag = true;                   // 定义布尔类型变量 flag，初始值为 true

char ch1 = 'a';                     // 定义字符类型变量 ch1，初始值为 'a'
wchar_t ch2 = L'你';                // 定义宽字符类型变量 ch2，初始值为 '你'
```

类型限定符

类型限定符提供了变量的额外信息，用于在定义变量或函数时**改变它们的默认行为的关键字。**

限定符	含义
const	**const** 定义常量，表示该变量的值不能被修改。
volatile	修饰符 **volatile** 告诉编译器**每次读取该变量**时不能使用寄存器而都**要访问原始内存地址。**
restrict	由 **restrict** 修饰的指针是**唯一一种访问**它所指向的对象的方式。只有C99增加了新的类型限定符restrict。
mutable	表示**类中的成员变量**可以在const成员函数中被修改。
static	用于定义**静态变量**，表示该变量的作用域**仅限于当前文件或当前函数**内，不会被**其他文件或函数**访问。函数的形参**不允许使用static**
register	用于定义**寄存器变量**，表示该变量被频繁使用，可以存储在CPU的寄存器中，以提高程序的运行效率。

const详解

const 变量

const 变量指的是，此变量的值是**只读**的，不应该被改变。

如果我们在程序中试图修改 const 变量的值，在编译的时候，编译器将给出错误提示。

正因为 const 变量的值在给定以后不能改变，所以 **const 变量必须被初始化。**

```
const int debugLevel =10;
const int logLevel; // 编译错误：未初始化const变量（这个错误是否提示，和所用的编译器有关）
debugLevel =5; // 编译错误：给只读变量赋值
```

在C++中，const 全局变量被用来替换一般常量宏定义。

因为虽然 const 变量的值不能改变，但是依然是变量，使用时依然会进行类型检查，

const全局定义时自带static，即const全局定义是**内部链接(只能在本文件中使用)**的。如果想去除这种**内部链接**，可以加上extern

const结构体变量

```c
struct debugInfo
{
    int debugLevel;
    int line;
};

int main( int argc, char *argv[])
{
    const struct debugInfo debug_const1;  // 编译错误：未初始化只读变量（与编译器实现有关）

    struct debugInfo debug_not_const;

    debug_not_const.debugLevel = 10;
    debug_not_const.line = 5;

    const struct debugInfo debug_const2 = debug_not_const;

    debug_const2.debugLevel = 10;  // 编译错误：不允许修改只读变量
    debug_const2.line = 2;         // 编译错误：不允许修改只读变量

    return 0;
}
```


在C中，const 结构体变量表示**结构体中任何数据域均不允许改变，**且需要另一个结构体变量进行初始化。

const 类对象

const类对象指的是，此类对象**不应该被改变。**

const 类对象与 const 变量并无实质不同，只在于类对象的 “改变” 定义。

类对象的 “**改变**” 定义：**改变任何成员变量的值或调用任何非const成员函数**

```
class CDebugModule
{
    public:
        CDebugModule() {};
        ~CDebugModule() {};
    public:
        int m_debugLevel;

    public:
        void SetDebugLevel(int debugLevel) { m_debugLevel = debugLevel;};
        void PrintDebugLevel(void) { cout << m_debugLevel;};
        void PrintDebugLevel_const(void) const { cout << m_debugLevel;};   // const 类成员函数
};

int main( int argc, char *argv[])
{
    const CDebugModule debug;

    debug.m_debugLevel = 10;        // 编译出错：不能直接改变成员变量
    debug.SetDebugLevel(20);        // 编译出错：不能通过成员函数改变成员变量
    debug.PrintDebugLevel();        // 编译出错：不能调用非 const 成员函数
    debug.PrintDebugLevel_const(); // 正常

    return 0;
}
```


指向 const 变量的指针

指向 const 变量的指针，指的是一个保存着一个 **const 变量的地址**的指针。

由于指针**指向一个 const 变量**，所以通过指针，不能**修改这个 const 变量**的值。

虽然指针**指向的值不能改变**，但是**指针的指向可以改变。**

```
const int debugLevel = 10;
const int logLevel = 10;

const int *p = &debugLevel;
p = &logLevel;     // 正常，指针的指向可以改变

*p = 10;           // 编译错误，指针指向的位置是只读的
```

```
const int debugLevel = 10;          // const 变量
int logLevel = 10;                  // 普通变量

const int *p;
int *q;

p = &logLevel;                      // 我们让指向 const 变量的指针指向一个普通变量
q = (int*)&debugLevel;              // 让指向普通变量的指针，指向一个 const 变量

*q = 5;  // 编译正常
*p = 5;  // 编译出错：位置为只读
```


编译器是通过 “指针的类型” 来判断是否只读的。

当指针的类型为“**指向const变量的指针**”，即使其指向的内容是**非const变量**，也无法通过这个指针改变数据内容。

当指针的类型是“**指向非const变量的指针**”，即使指向的是**const变量**，也可以通过这个指针改变const变量的内容。

const 指针

const指针指的是一个自身内容（指针指向）**不发生改变的指针。**

指针的指向一经确定，不能改变。指针指向的内容，可以改变。

```
int logLevel = 10;
int logId = 0;

int * const p = &logLevel;
int * const q;                  // 编译错误，未初始化 const 变量（这个错误是否报告，和所用的编译器有关）

*p = 5;                         // 正常，const指针指向内容可以改变
p = &logId                      // 编译错误，const指针自身内容（指向）不能改变
```


指针也是一种变量，只不过其内容保存的是地址而已。所以const指针的内容不能改变，也即是它的指向不能改变。

const指针和指向const变量的指针，有一种区分方法是：**从右向左，依次结合，const就近结合。**

比如，int * const p 可以这样进行解读：

1、int * (const p)：变量p 经过 const 修饰，为只读变量。

2、int (* (const p))：（const p 现在作为一个整体） 只读变量p是一个指针。

3、(int (* (const p)))：（同样的 * const p 作为一个整体） 这个只读的指针p，指向一个int型变量。

于是，可以区分出 int * const p 是一个指向 int 型的const指针。

再比如，const int * p 可以这样解读：

1、const int (* p)：变量p是一个指针。

2、（const int）（* p）：（const与就近的 int 结合）这个指针指向 const int 型变量。

所以，const int * p 是一个指向 const 整形变量的指针

指向 const 变量的 const 指针

即**既不允许改变指向变量内容**也**不允许改变指向**的指针

const 变量作为函数参数

将函数参数声明为 const 类型，表示**对于函数来说，这个参数是一个 const 变量。**

也就是说，函数内部不能够改变这个参数的值。

将函数参数是一个指针，把它声明为 “指向 const 变量的指针” ，**允许上层使用 ” 指向 const 变量的指针 “ 或 普通指针 作为参数，调用函数。**

```
// 接收一个int变量，并在函数内部，认为它是只读的
void OutputInt_const( const int a )
{
    a = 5;  // 编译错误：不允许修改只读变量
    printf("a = %d\n", a);
}

// 接收一个int变量，在函数内部，认为它是普通的
void OutputInt_not_const( int a )
{
    a = 5;  // 正常
    printf("a = %d\n", a);
}

// 接收一个 指向const型整形数的指针
void OutputInt_p_const( const int *a )
{
    *a = 5;  // 编译错误：
    printf("*a = %d\n", *a);
}

// 接收一个普通指针
void OutputInt_p_not_const( int *a )
{
    *a = 5;
    printf("*a = %d\n", *a);
}
```

const 返回值

const 型的返回值指的是**函数的返回值**为一个 **const 变量。**

函数返回const返回值，主要用于**函数返回const引用。**

```cpp
#include <string>

using namespace std;

// 返回 const 引用的函数
const string& SetVersion_const(string & versionInfo)
{
    versionInfo = "V0.0.3";
    return versionInfo;
}

// 返回普通引用的函数
string& SetVersion_not_const(string & versionInfo)
{
    versionInfo = "V0.0.3";
    return versionInfo;
}

// 主函数
int main( int argc, char *argv[])
{
    string versionInfo;

    SetVersion_const(versionInfo) = "V0.0.5";       // 编译错误，返回的引用为 const 引用，不允许修改。

    SetVersion_not_const(versionInfo) = "V0.0.5";   // 正常，返回的引用为普通引用，可以修改内容。

    return 0;
}
```


const 引用相当于指向 const 变量的 const 指针，其指向和指向的内容均不允许改变。

在函数返回 const 引用时，不能够通过函数返回的引用对实际对象进行任何修改，即便对象本身不是 const 的。

在本例中，versionInfo 在 main 函数中不是const的。

SetVersion_const 函数返回了一个指向 versionInfo 的 const 引用，不能通过此引用，对 versionInfo 进行修改。

const 成员变量

const 成员变量指的是类中的**成员变量为只读**，不能够被修改（包括在**类外部和类内部**）。

const 成员变量**必须被初始化**（在相关构造函数的**初始化列表中**），**初始化后，不能够被修改。**

静态 const 成员变量需要在类外部单独定义并初始化（可定义在头文件）

```
class CDebugModule
{
public:
    CDebugModule();
    ~CDebugModule();

public:
    const int m_debugLevel;
    static const int m_debugInfo;

};

const int CDebugModule::m_debugInfo = 1;  // 静态常量成员需要在类外进行单独定义和初始化

CDebugModule::CDebugModule()
    : m_debugLevel(10)  // const 成员在构造函数初始化列表中初始化
{
}

CDebugModule::~CDebugModule()
{
}

int main(int argc, char *argv[])
{
    CDebugModule debugModule;

    debugModule.m_debugLevel = 10;  // 编译错误，不能改变只读成员
    CDebugModule::m_debugInfo = 10; // 编译错误，不能改变只读成员

    return 0;
}
```


类对象的实例化过程可以理解为包含以下步骤：首先，开辟整个类对象的内存空间。之后，根据类成员情况，分配各个成员变量的内存空间，并通过构造函数的初始化列表进行初始化。最后，执行构造函数中的代码。由于 const 成员变量必须在定义（分配内存空间）时，就进行初始化。所以需要在够在函数的初始化列表中初始化。const成员在初始化之后，其值就不允许改变了，即便在构造内部也是不允许的。

静态成员变量并不属于某个类对象，而是整个类共有的。静态成员变量可以不依附于某个实例化后的类对象进行访问。那么，静态成员变量的值，应该在任何实例化操作之前，就能够进行改变（否则，只有实例化至少一个对象，才能访问静态成员）。所以，静态成员变量不能够由构造函数进行内存分配，而应该在类外部单独定义，在实例化任何对象之前，就开辟好空间。又由于 const 成员变量 必须初始化，所以静态成员变量必须在定义的时候就初始化。

const 成员函数

const成员函数指的是，此函数**不应该修改任何成员变量。**

确定不修改任何成员变量的函数应该声明为**const成员函数**

传给const成员函数的this指针，是**指向 const 对象 的 const 指针。**

const成员函数，**不能够修改**任何成员变量，**除非成员变量被 mutable 修饰符修饰。**

一个const对象也只能使用它类内的**const成员函数**

```
class CDebugModule
{
public:
    CDebugModule() {};
    ~CDebugModule();

public:
    int m_debugLevel_not_mutable;        // 不带 mutable 修饰的成员变量
    mutable int m_debugLevel_mutable;    // 带 mutable 修饰的成员变量

public:
    void SetDebugLevel_not_const(int debugLevel);   // 非 const 成员函数
    void SetDebugLevel_const(int debugLevel) const; // const 成员函数
};

void CDebugModule::SetDebugLevel_not_const(int debugLevel)
{
    m_debugLevel_not_mutable = debugLevel;
    m_debugLevel_mutable = debugLevel;
    return;
}

void CDebugModule::SetDebugLevel_const(int debugLevel) const
{
    m_debugLevel_not_mutable = debugLevel; // 编译错误，const 成员函数不能修改一般的成员变量
    m_debugLevel_mutable = debugLevel;     // 正常，当成员变量被 mutable 修饰时，const成员函数就能修改了
    return;
}

int main(int argc, char *argv[])
{
    CDebugModule debugModule;

    debugModule.SetDebugLevel_not_const(10);
    debugModule.SetDebugLevel_const(10);

    return 0;
}
```


返回const对象

返回const对象可以避免出现将"n=f1+f2"写成"f1+f2=n"的情况

static详解

下面总结了`static`关键字的不同用法，并为每种用法提供示例代码和运行后的输出结果以及相应的解释。

static变量

static变量在编译时被存储在静态存储区中，在程序的整个运行期间都存在，

如果没有被初始化，static变量自动被初始化为0值

示例代码：

```
#include <iostream>

void foo() {
    static int counter = 0;
    counter++;
    std::cout << "Counter: " << counter << std::endl;
}

int main() {
    foo();  // 输出：Counter: 1
    foo();  // 输出：Counter: 2
    foo();  // 输出：Counter: 3
    return 0;
}
```

静态变量 `counter` 在函数 `foo` 中声明并初始化为0。每次调用 `foo` 函数时，`counter` 的值会递增，并在每次调用后输出。

由于静态变量在程序的整个运行期间都存在，函数运行结束后静态变量会保留其原有的值

static函数(不包括成员函数)

相当于一个**全局函数**

即使在**类内成员访问权限允许**的情况下，static函数仍然不能访问**类内非静态成员**

static类成员

用**static修饰的类内成员**，这个类的所有对象都**共用这一个成员。**

这样的成员需要在**类内声明，在类外定义。**

C++允许static的**int类**和**enum类**在类内声明和定义。

示例：

```
class A
```

```
{     int x,y;

static int shared;//静态数据成员声明

public:

A(){x=y=0;}

void increase_all(){x++;y++;shared++;}//每当这个类的对象调用这个函数，所有这个类的对象的shared都会自增

int sum_all() const {return x+y+shared};

};

int A::shared=0;//静态数据成员定义
```

static成员函数

用static修饰的成员函数，只能**访问类内的静态数据成员。**

也正因此，static修饰的成员函数**不需要this指针**，因为不需要this指针来区别访问的是哪个对象的成员。

设有一个静态函数在类A中，为static void getshared(),有个对象a

访问静态函数的方法有

- getshared(10)//通过**对象访问**
- A::getshared()//通过**类名限制访问**

数学运算

运算符

算术运算符

下表显示了 C++ 支持的算术运算符，假设变量 A 的值为 10，变量 B 的值为 20，则：

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

- 任意**两个整型相除**时，其结果也为**整型**。如5除以2为2.
- 两个数据至少有一个为浮点型，则运算结果为double型。如5.0除以2为2.5
- **求余运算符%**两侧的数据必须均为**整型数据**，其运算结果为**两个数据相除的余数。**
- 两个符号相异的数求余，结果的符号取决于被除数的符号。
- **自增在前**，变量**先自增后应用**；**自增在后**，变量**先应用后自增**

```
#include <iostream>

using namespace std;

int main()
{
   int a = 21;
   int b = 10;
   int c;
   c = a + b;
   cout << "Line 1 - c 的值是 " << c << endl ;
   c = a - b;
   cout << "Line 2 - c 的值是 " << c << endl ;
   c = a * b;
   cout << "Line 3 - c 的值是 " << c << endl ;
   c = a / b;
   cout << "Line 4 - c 的值是 " << c << endl ;
   c = a % b;
   cout << "Line 5 - c 的值是 " << c << endl ;
   int d = 10;   //  测试自增、自减
   c = d++;
   cout << "Line 6 - c 的值是 " << c << endl ;
   d = 10;    // 重新赋值
   c = d--;
```

```
   cout << "Line 7 - c 的值是 " << c << endl ;
   return 0;
}
```

关系运算符

下表显示了 C++ 支持的关系运算符。假设变量 A 的值为 10，变量 B 的值为 20，则：

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

```
#include <iostream>

using namespace std;

int main()
{
   int a = 21;
   int b = 10;
   int c ;
   if( a == b )
   {
      cout << "Line 1 - a 等于 b" << endl ;
   }
   else
   {
      cout << "Line 1 - a 不等于 b" << endl ;
   }
   if ( a < b )
   {
      cout << "Line 2 - a 小于 b" << endl ;
   }
   else
   {
      cout << "Line 2 - a 不小于 b" << endl ;
   }
   if ( a > b )
```

```
   {
      cout << "Line 3 - a 大于 b" << endl ;
   }
   else
   {
      cout << "Line 3 - a 不大于 b" << endl ;
   }
   /* 改变 a 和 b 的值 */

   a = 5;
   b = 20;
   if ( a <= b )
   {
      cout << "Line 4 - a 小于或等于 b" << endl ;
   }
   if ( b >= a )
   {
      cout << "Line 5 - b 大于或等于 a" << endl ;
   }
   return 0;
}
```

当上面的代码被编译和执行时，它会产生以下结果：

```
Line 1 - a 不等于 b

Line 2 - a 不小于 b

Line 3 - a 大于 b

Line 4 - a 小于或等于 b

Line 5 - b 大于或等于 a
```

逻辑运算符

下表显示了 C++ 支持的**关系逻辑运算符**。假设变量 A 的值为 1，变量 B 的值为 0，则：

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

```
#include <iostream>

using namespace std;

int main()
{
   int a = 5;
   int b = 20;
   int c ;
   if ( a && b )
   {
      cout << "Line 1 - 条件为真"<< endl ;
   }
   if ( a || b )
   {
      cout << "Line 2 - 条件为真"<< endl ;
   }

   /* 改变 a 和 b 的值 */
   a = 0;
   b = 10;
   if ( a && b )
   {
      cout << "Line 3 - 条件为真"<< endl ;
   }
   else
   {
      cout << "Line 4 - 条件不为真"<< endl ;
   }
   if ( !(a && b) )
   {
      cout << "Line 5 - 条件为真"<< endl ;
   }
   return 0;
}
```

当上面的代码被编译和执行时，它会产生以下结果：

```
Line 1 - 条件为真

Line 2 - 条件为真

Line 4 - 条件不为真
```

```
Line 5 - 条件为真
```

在计算含有或、与的逻辑表达式时，当从左至右计算时**已能确定总表达式的值**，后面的子表达式**便不计算**。

位运算符

位运算符作用于位，并逐位执行操作。&、 | 和 ^ 的真值表如下所示：

p	q	p & q	p | q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

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

```
A = 0011 1100

B = 0000 1101

A&B = 0000 1100

A|B = 0011 1101

A^B = 0011 0001

~A  = 1100 0011
```

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

运算符	描述	实例
&	按位与操作，按二进制位进行"与"运算。运算规则： 0&0=0; 0&1=0; 1&0=0; 1&1=1;	(A & B)将得到12，即为0000 1100
|	按位或运算符，按二进制位进行"或"运算。运算规则：0|0=0;0|1=1;1|0=1;1|1=1;	按位或运算符，按二进制位进行"或"运算。运算规则： 0
^	异或运算符，按二进制位进行"异或"运算。两位相同记0，否则记1，运算规则： $0^{0=0;}\ 0$1=1; $1^{0=1;}\ 1$1=0;	(A ^ B)将得到49，即为0011 0001
~	取反运算符，按二进制位进行"取反"运算。运算规则： ~1=-2; ~0=-1;	(~A)将得到-61，即为1100 0011，一个有符号二进制数的补码形式。
<<	二进制左移运算符。将一个运算对象的各二进制位全部左移若干位（左边的二进制位丢弃，右边补0）。	A << 2将得到240，即为1111 0000
>>	二进制右移运算符。将一个数的各二进制位全部右移若干位，正数左补0，负数左补1，右边丢弃。	

```
#include <iostream>
```

```
using namespace std;

int main()
{
   unsigned int a = 60;      // 60 = 0011 1100
   unsigned int b = 13;      // 13 = 0000 1101
   int c = 0;
   c = a & b;             // 12 = 0000 1100
   cout << "Line 1 - c 的值是 " << c << endl ;
   c = a | b;             // 61 = 0011 1101
   cout << "Line 2 - c 的值是 " << c << endl ;
   c = a ^ b;             // 49 = 0011 0001
   cout << "Line 3 - c 的值是 " << c << endl ;
   c = ~a;                // -61 = 1100 0011
   cout << "Line 4 - c 的值是 " << c << endl ;
   c = a << 2;            // 240 = 1111 0000
   cout << "Line 5 - c 的值是 " << c << endl ;
   c = a >> 2;            // 15 = 0000 1111
   cout << "Line 6 - c 的值是 " << c << endl ;
   return 0;
}
```

当上面的代码被编译和执行时，它会产生以下结果：

```
```
Line 1 - c 的值是 12
Line 2 - c 的值是 61
Line 3 - c 的值是 49
Line 4 - c 的值是 -61
Line 5 - c 的值是 240
Line 6 - c 的值是 15
```
```

赋值运算符

下表列出了 C++ 支持的赋值运算符：

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

```
#include <iostream>

using namespace std;

int main()
{
   int a = 21;
   int c ;
   c =  a;
   cout << "Line 1 - =  运算符实例，c 的值 = : " <<c<< endl ;
   c +=  a;
   cout << "Line 2 - += 运算符实例，c 的值 = : " <<c<< endl ;
   c -=  a;
   cout << "Line 3 - -= 运算符实例，c 的值 = : " <<c<< endl ;
   c *=  a;
   cout << "Line 4 - *= 运算符实例，c 的值 = : " <<c<< endl ;
   c /=  a;
   cout << "Line 5 - /= 运算符实例，c 的值 = : " <<c<< endl ;
   c  = 200;
   c %=  a;
   cout << "Line 6 - %= 运算符实例，c 的值 = : " <<c<< endl ;
   c <<=  2;
   cout << "Line 7 - <<= 运算符实例，c 的值 = : " <<c<< endl ;
   c >>=  2;
   cout << "Line 8 - >>= 运算符实例，c 的值 = : " <<c<< endl ;
   c &=  2;
   cout << "Line 9 - &= 运算符实例，c 的值 = : " <<c<< endl ;
   c ^=  2;
   cout << "Line 10 - ^= 运算符实例，c 的值 = : " <<c<< endl ;
   c |=  2;
```

```
   cout << "Line 11 - |= 运算符实例，c 的值 = : " <<c<< endl ;
   return 0;
}
```

当上面的代码被编译和执行时，它会产生以下结果：

```
Line 1 - =  运算符实例，c 的值 = 21

Line 2 - += 运算符实例，c 的值 = 42

Line 3 - -= 运算符实例，c 的值 = 21

Line 4 - *= 运算符实例，c 的值 = 441

Line 5 - /= 运算符实例，c 的值 = 21

Line 6 - %= 运算符实例，c 的值 = 11

Line 7 - <<= 运算符实例，c 的值 = 44

Line 8 - >>= 运算符实例，c 的值 = 11

Line 9 - &= 运算符实例，c 的值 = 2

Line 10 - ^= 运算符实例，c 的值 = 0

Line 11 - |= 运算符实例，c 的值 = 2
```

杂项运算符

下表列出了 C++ 支持的其他一些重要的运算符。

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

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

如果 Exp1 为假，则计算 Exp3 的值，结果即为整个？表达式的值。

只计算其中一个表达式，另一个不计算

运算符优先级

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

副作用与序列点

副作用

函数的主要目的是**对表达式求值**，而副作用是**对数据对象或文件**的修改。

序列点

是**程序的进行点**，所有函数的副作用必须在**下一个序列点(下一步程序)之前**完成。

如**赋值运算符**、**自增自减运算符**必须在程序执行**下一条语句之前完成。**

序列点种类

完整表达式

完整表达式并非子表达式，而是**整个表达式。**

逻辑运算符||和&&

逻辑运算符的**两侧**各产生一个**序列点。**

条件运算符

条件运算符中的条件是一个序列点。

逗号运算符

一个逗号运算符就代表一个序列点

数学运算

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

为了利用这些函数，要引用数学头文件 **<cmath>**。

序号	函数 & 描述
1	**double cos(double);/double acos(double);** 该函数返回弧度角（double型）的余弦/反余弦。
2	**double sin(double);/ double asin(double);** 该函数返回弧度角（double型）的正弦/反正弦。
3	**double tan(double);/ double atan(double);** 该函数返回弧度角（double型）的正切/反正切。
4	**double log(double);/ double log10(double);** 该函数返回lnx/lgx。
5	**double pow(double, double);/double pow10(int p)/double exp(double x)** 假设第一个参数为x，第二个参数为y，则该函数返回x的y次方/返回10的p次方/返回e的x次方
6	**double hypot(double, double);** 该函数返回两个参数的平方总和的平方根，也就是说，参数为一个直角三角形的两个直角边，函数会返回斜边的长度。
7	**double sqrt(double);/double cbrt(double)** 该函数返回参数的平方根/立方根。
8	**int abs(int);** 该函数返回整数的绝对值。
9	**double fabs(double);** 该函数返回任意一个浮点数的绝对值。
10	**double floor(double);/double ceil(double)** 取下整/取上整
11	**double round(double)** 四舍五入
12	**double fmax(double,double)/double fmin(double,double)** 求两个参数的最大值/最小值

随机数

随机数生成器与两个函数相关。

一个是 **rand()**，该函数只返回一个伪随机数。

生成随机数之前必须先调用 **srand()** 函数生成种子，在种子相同的情况下，会得到相同的随机数。

要获得**一定范围内的随机数**，可以通过对随机数**求余**得到

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

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

using namespace std;

int main ()
{
   int i,j;

   // 设置种子
   srand( (unsigned)time( NULL ) );
   /* 生成 10 个随机数 */

   for( i = 0; i < 10; i++ )
   {
      // 生成实际的随机数
      j= rand();
      cout <<"随机数： " << j << endl;
   }
   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
随机数： 1748144778

随机数： 630873888

随机数： 2134540646

随机数： 219404170
```

```
随机数： 902129458

随机数： 920445370

随机数： 1319072661

随机数： 257938873

随机数： 1256201101

随机数： 580322989
```

循环

循环类型

while循环

只要给定的条件为真，**while** 循环语句会**重复执行一个目标语句。**

语法

C++ 中 **while** 循环的语法：

```
while(condition)
{

   statement(s);

}
```

在这里，**statement(s)** 可以是一个**单独的语句**，也可以是几个语句组成的代码块。

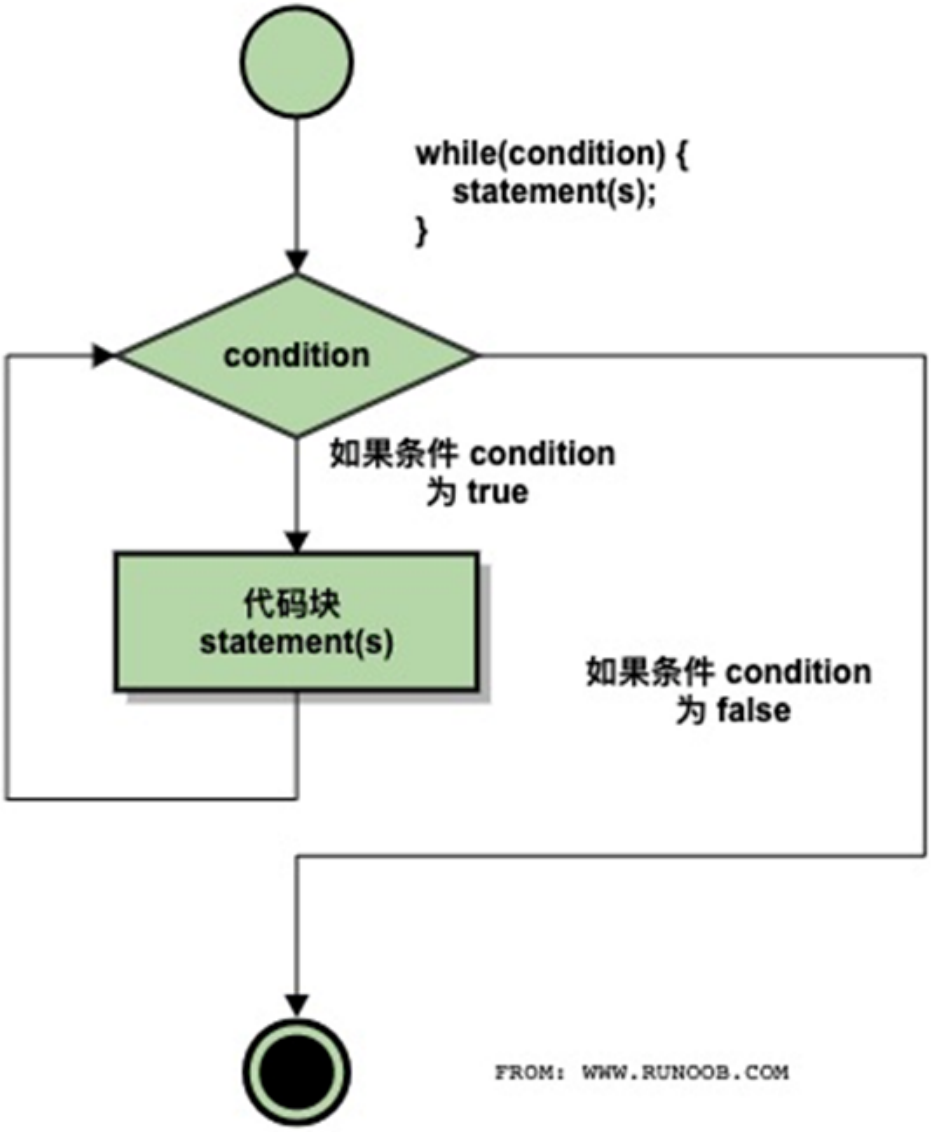
condition 可以是任意的表达式，当为**任意非零值**时都为真。当条件为真时执行循环。

当**条件为假**时，程序流将继续执行紧接着循环的**下一条语句。**

while循环体内**要有改变循环条件的语句**，避免出现**死循环**

如果出现了死循环，可以用**ctrl+c**结束

流程



while 循环可能**一次都不会执行。**当**条件被测试且结果为假**时，会跳过循环主体，直接执行紧接着 while 循环的下一条语句。

```
#include <iostream>

using namespace std;

int main ()

{
```

```
   // 局部变量声明
   int a = 10;
   // while 循环执行
   while( a < 20 )
   {
       cout << "a 的值: " << a << endl;
       a++;
   }
   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
a 的值： 10
a 的值： 11
a 的值： 12
a 的值： 13
a 的值： 14
a 的值： 15
a 的值： 16
a 的值： 17
a 的值： 18
a 的值： 19
```

for循环

for 循环可用于一个**执行特定次数**的循环的重复控制结构。

语法

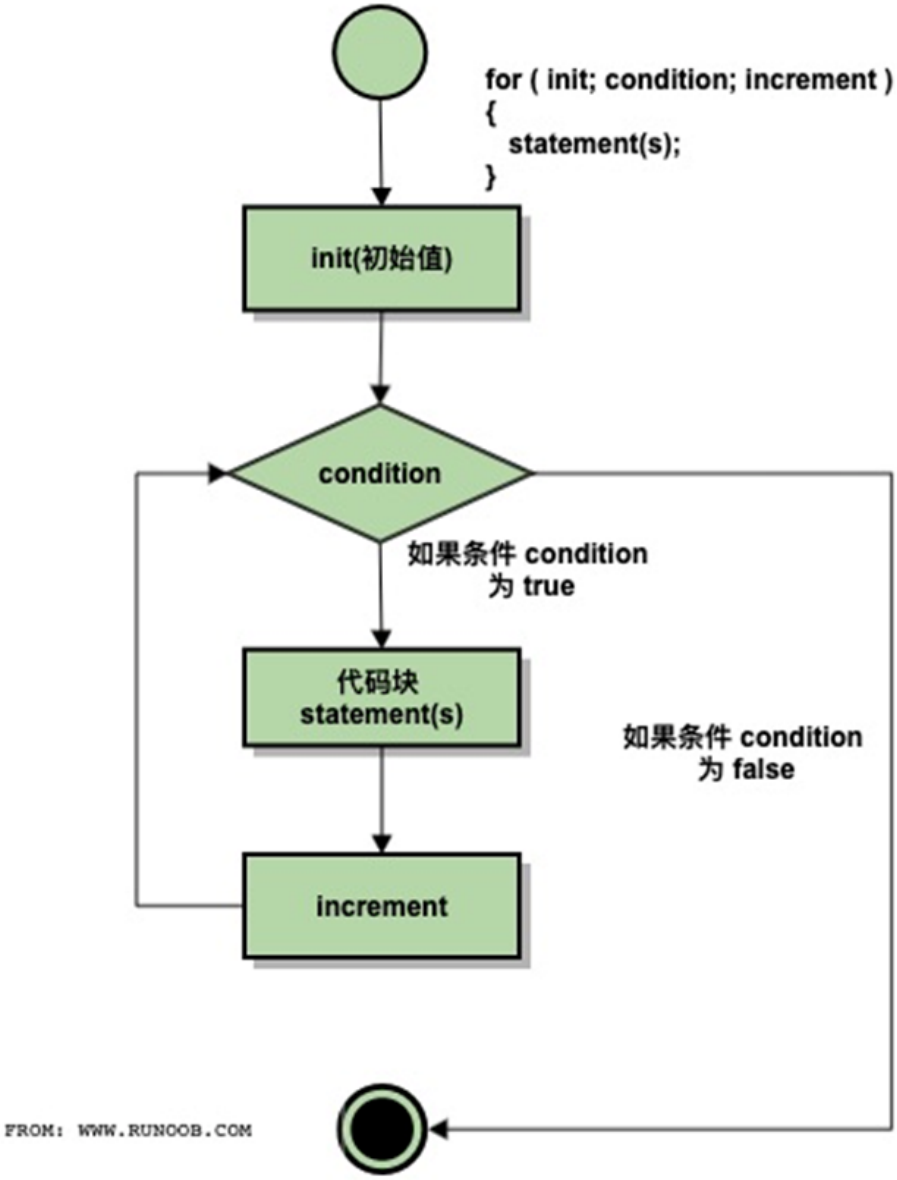
C++ 中 **for** 循环的语法：

```
for ( init; condition; increment )

{

   statement(s);

}
```

下面是 for 循环的控制流：

1. **init (初始化)**会首先被执行，且只会执行一次。这一步**声明并初始化**任何**循环控制变量**。可以不在这里写任何语句，只要**有一个分号**出现即可。
2. 接下来，会判断 **condition(循环条件)**。如果为真，则执行循环主体。如果为假，则不执行循环主体，且控制流会跳转到紧接着 for 循环的下一条语句。若**循环条件留空**，则**默认判断循环条件为真**，需要在循环体内加入终止循环的语句
3. 在执行完 **for 循环主体**后，控制流会跳回上面的 **increment (更新循环条件)**语句。该语句**更新循环控制变量**且可以留空，只要在条件后有一个分号出现即可。
4. **条件再次被判断**。如果为真，则执行循环，这个过程会不断重复（循环主体，然后增加步值，再然后重新判断条件）。在条件变为假时，for 循环终止。

流程图




```
#include <iostream>

using namespace std;

int main ()

{

   // for 循环执行

   for( int a = 10; a < 20; a = a + 1 )

   {
```

```
        cout << "a 的值: " << a << endl;
    }
    return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
a 的值： 10
a 的值： 11
a 的值： 12
a 的值： 13
a 的值： 14
a 的值： 15
a 的值： 16
a 的值： 17
a 的值： 18
a 的值： 19
```

基于范围的for循环(C++11)

for 语句允许简单的范围迭代：

```
int my_array[5] = {1, 2, 3, 4, 5};
// 每个数组元素乘于 2
for (int &x : my_array)
{
    x *= 2;
```

```
    cout << x << endl;
}
// auto 类型也是 C++11 新标准中的，用来自动获取变量的类型
for (auto &x : my_array) {
    x *= 2;
    cout << x << endl;
}
```

上面for述句的第一部分**定义被用来做范围迭代的变量**，就像被声明在一般for循环的变量一样，其**作用域仅只于循环的范围。**

而在":"之后的第二区块，代表**将被迭代的范围。**这个基于范围的for循环常用在**STL**中

```
#include<iostream>
#include<string>
#include<cctype>
using namespace std;
int main()
{
    string str("some string");
    // range for 语句
    for(auto &c : str)
    {
        c = toupper(c);
    }
    cout << str << endl;
```

```
    return 0;

}
```

上面的程序使用Range for语句遍历一个字符串，并将所有字符全部变为大写，然后输出结果为：

```
SOME STRING
```

do...while语句

不像 **for** 和 **while** 循环，它们是在**循环头部**测试循环条件。**do...while** 循环是在**循环的尾部**检查它的条件。

do...while 循环与 while 循环类似，但是 **do...while 循环**会确保**至少执行一次循环**。

语法

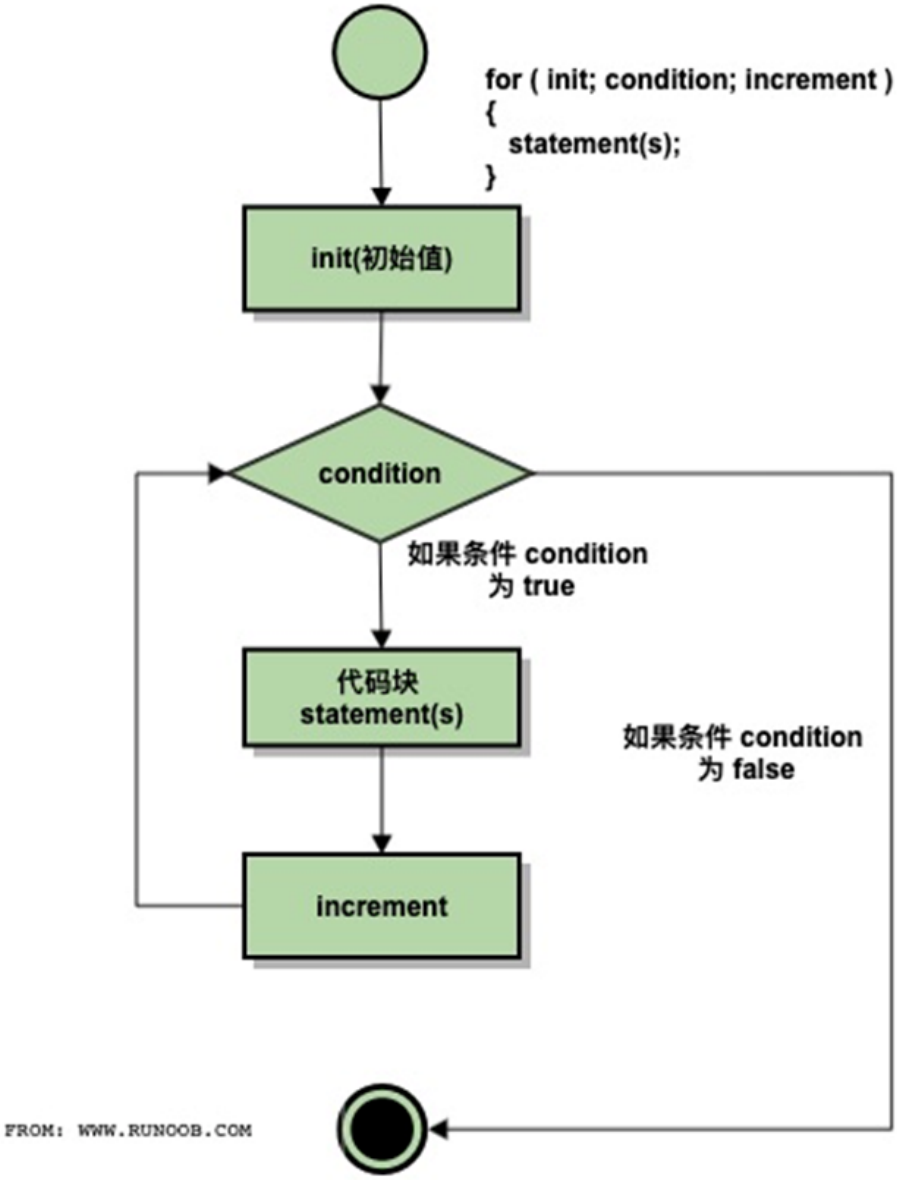
C++ 中 **do...while** 循环的语法：

```
do

{

   statement(s);

}while( condition );
```

条件表达式出现在**循环的尾部**，所以循环中的 statement(s) 会在条件被测试之前**至少执行一次**。

如果条件为真，控制流会跳转回上面的 do，然后重新执行循环中的 statement(s)。这个过程会不断重复，直到给定条件变为假为止。

流程图




```
#include <iostream>

using namespace std;

int main ()

{

   // 局部变量声明

   int a = 10;

   // do 循环执行
```

```
   do
   {
        cout << "a 的值： " << a << endl;
        a = a + 1;
   }while( a < 20 );
   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
a 的值： 10
a 的值： 11
a 的值： 12
a 的值： 13
a 的值： 14
a 的值： 15
a 的值： 16
a 的值： 17
a 的值： 18
a 的值： 19
```

嵌套循环

除上面三种基本的循环外，还可以**嵌套循环**。

循环控制语句

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

break语句

break 语句有以下两种用法：

1. 当 **break** 语句出现在一个循环内时，循环会立即终止，且程序流将继续执行紧接着循环的下一条语句。
2. 它可用于终止 **switch** 语句中的一个 case。

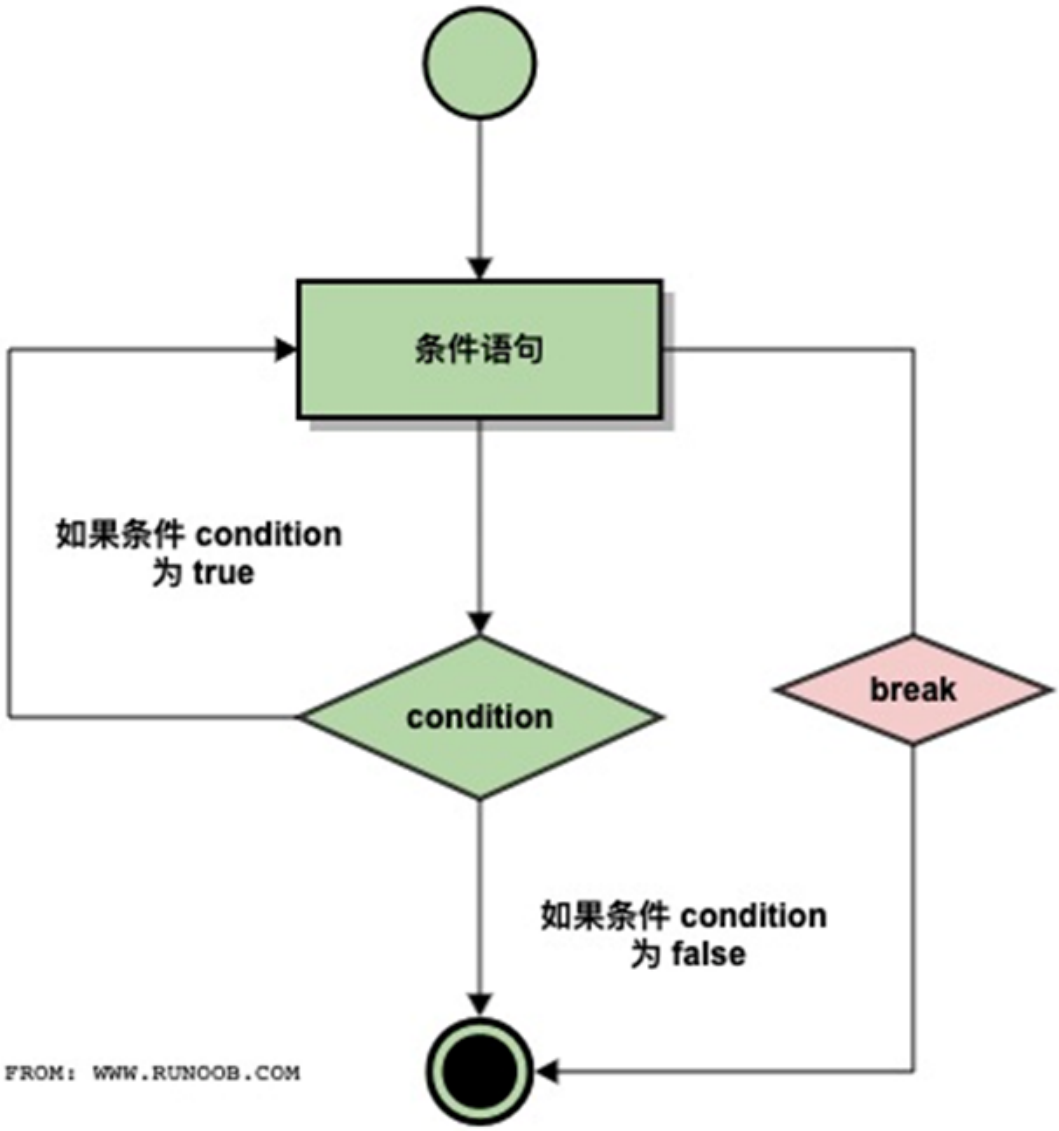
如果break用在嵌套循环，break 语句会停止执行最内层的循环，然后开始执行该块之后的下一行代码。

语法

C++ 中 **break** 语句的语法：

```
break;
```

流程图




```cpp
#include <iostream>

using namespace std;

int main ()

{

   // 局部变量声明

   int a = 10;

   // do 循环执行
```

```
   do

   {

        cout << "a 的值: " << a << endl;

        a = a + 1;

        if( a > 15)

        {

           // 终止循环

           break;

        }

   }while( a < 20 );

   return 0;

}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
a 的值： 10

a 的值： 11

a 的值： 12

a 的值： 13

a 的值： 14

a 的值： 15
```

continue语句

continue会**跳过当前循环**中的代码，**强迫开始下一次循环。**

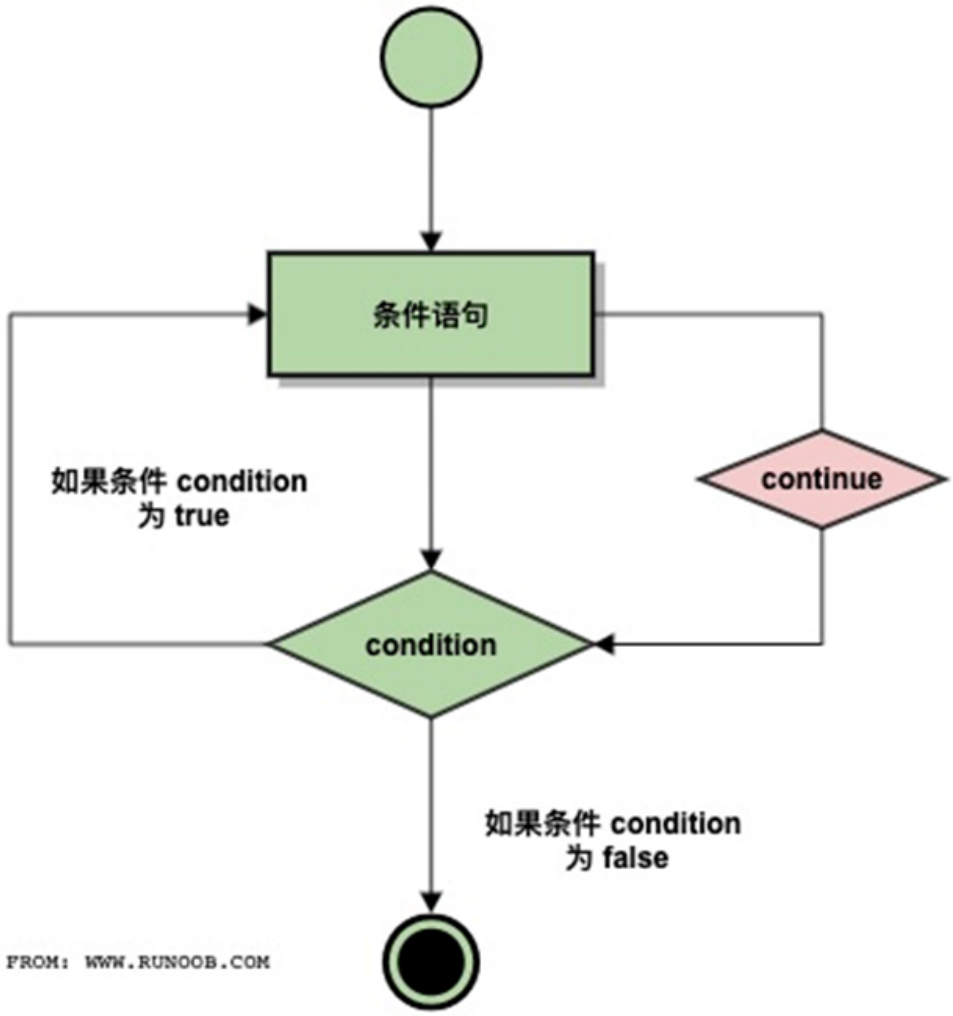
对于 **for** 循环，**continue** 语句会导致**执行条件测试**和**循环增量**部分。对于 **while** 和 **do...while** 循环，**continue** 语句会导致**程序控制回到条件测试**上。

语法

C++ 中 **continue** 语句的语法：

```
continue;
```

流程图




```
#include <iostream>

using namespace std;
```

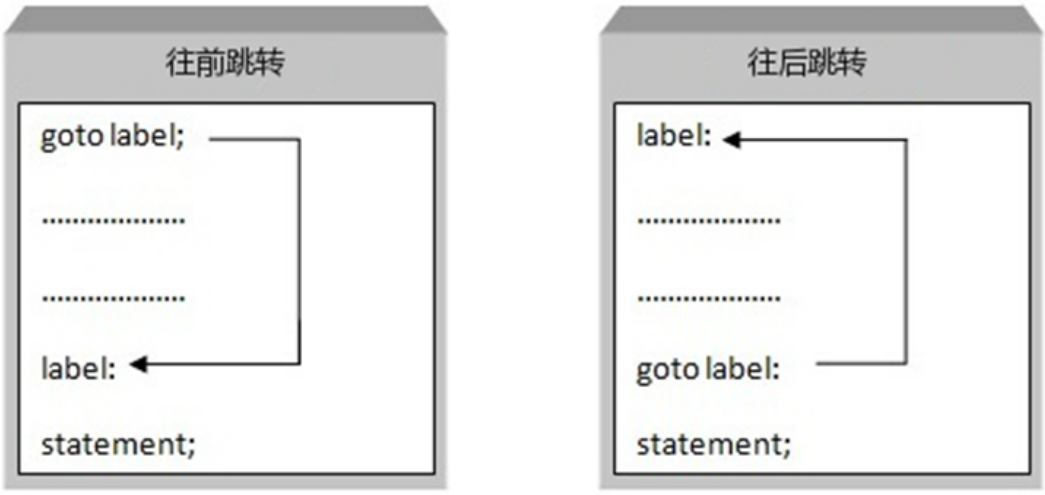
```
int main ()

{

   // 局部变量声明

   int a = 10;

   // do 循环执行

   do

   {

       if( a == 15)

       {

          // 跳过迭代

          a = a + 1;

          continue;

       }

       cout << "a 的值: " << a << endl;

       a = a + 1;

   }while( a < 20 );

   return 0;

}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
a 的值： 10

a 的值： 11

a 的值： 12

a 的值： 13
```

```
a 的值： 14

a 的值： 16

a 的值： 17

a 的值： 18

a 的值： 19
```

goto语句



goto 语句允许把控制**无条件转移**到同一函数内的被标记的语句。

但在任何编程语言中，都不建议使用 goto 语句。

因为它使得程序的控制流难以跟踪，使程序难以理解和难以修改。任何使用 goto 语句的程序可以改写成不需要使用 goto 语句的写法。

goto的跳跃范围仅限于**函数体内，不能跳到函数体之外，也不能从函数体外跳入函数体内。**可以从循环体内到循环体外，但是**不能从循环体外到循环体内。**

语法

C++ 中 **goto** 语句的语法：

```
goto label;

...
```

```
label: statement;
```

在这里，**label** 是**识别被标记语句的标识符**，可以是任何除 C++ 关键字以外的纯文本。标记语句可以是**任何语句**，放置在标识符和冒号（:）后边。

goto 语句一个很好的作用是**退出深嵌套**循环。例如：

```
for(...) {
   for(...) {
      while(...) {
         if(...) goto stop;
         .
         .
         .
      }
   }
}
stop:
cout << "Error in program.\n";
```

消除 **goto** 会导致一些额外的测试被执行。一个简单的 **break** 语句在这里不会起到作用，因为它只会使程序退出最内层循环。

判断

判断结构要求指定**一个或多个要评估或测试**的条件，以及条件为真时要执行的语句（必需的）和条件为假时要执行的语句（可选的）。

if 语句

一个 **if 语句** 由一个**布尔表达式**后跟**一个或多个语句**组成。if语句同时也是可以**嵌套**的

语法

C++ 中 **if** 语句的语法：

```
if(boolean_expression)
{
```

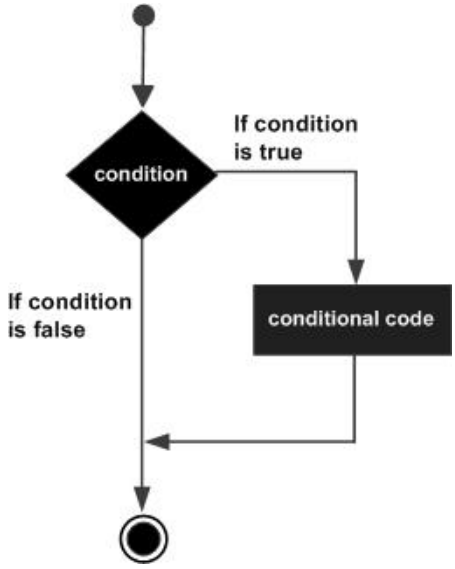
```
   // 如果布尔表达式为真将执行的语句
}
```

如果布尔表达式为 **true**，则 if 语句内的代码块将被执行。

如果布尔表达式为 **false**，则 if 语句结束后的第一组代码（闭括号后）将被执行。

C 语言把任何**非零**和**非空**的值假定为 **true**，把**零**或 **null** 假定为 **false**。

流程图




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

int main ()
{
   // 局部变量声明
   int a = 10;

   // 使用 if 语句检查布尔条件
   if( a < 20 )
   {
       // 如果条件为真，则输出下面的语句
       cout << "a 小于 20" << endl;
   }
   cout << "a 的值是 " << a << endl;

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
a 小于 20
a 的值是 10
```

if...else语句

一个 **if 语句** 后可跟一个可选的 **else 语句**，else 语句在布尔表达式为假时执行。

语法

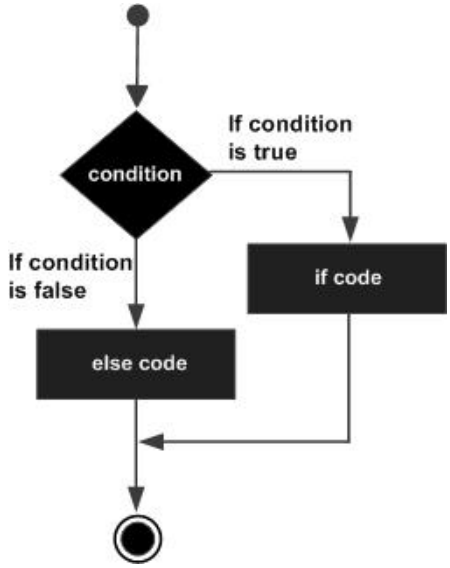
C++ 中 **if...else** 语句的语法：

```
if(boolean_expression)
{
   // 如果布尔表达式为真将执行的语句
}
else
{
   // 如果布尔表达式为假将执行的语句
}
```

如果布尔表达式为 **true**，则执行 **if** 块内的代码。

如果布尔表达式为 **false**，则执行 **else** 块内的代码。

流程图



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

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

   // 检查布尔条件
   if( a < 20 )
   {
       // 如果条件为真，则输出下面的语句
       cout << "a 小于 20" << endl;
   }
   else
   {
       // 如果条件为假，则输出下面的语句
       cout << "a 大于 20" << endl;
   }
   cout << "a 的值是 " << a << endl;

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
a 大于 20
a 的值是 100
```

if...else if...else 语句

一个 **if** 语句后可跟一个可选的 **else if...else** 语句，这可用于测试多种条件。

当使用 if...else if...else 语句时，以下几点需要注意：

- 一个 if 后可跟零个或一个 else，else 必须在所有 else if 之后。
- 一个 if 后可跟零个或多个 else if，else if 必须在 else 之前。
- 一旦某个 else if 匹配成功，其他的 else if 或 else 将不会被测试。

语法

C++ 中的 **if...else if...else** 语句的语法：

```
if(boolean_expression 1)
{
   // 当布尔表达式 1 为真时执行
}
else if( boolean_expression 2)
{
   // 当布尔表达式 2 为真时执行
}
else if( boolean_expression 3)
{
   // 当布尔表达式 3 为真时执行
}
else
{
   // 当上面条件都不为真时执行
}
```

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

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

   // 检查布尔条件
   if( a == 10 )
   {
       // 如果 if 条件为真，则输出下面的语句
       cout << "a 的值是 10" << endl;
   }
   else if( a == 20 )
   {
       // 如果 else if 条件为真，则输出下面的语句
       cout << "a 的值是 20" << endl;
   }
   else if( a == 30 )
   {
       // 如果 else if 条件为真，则输出下面的语句
       cout << "a 的值是 30" << endl;
   }
   else
```

```
   {
       // 如果上面条件都不为真，则输出下面的语句
       cout << "没有匹配的值" << endl;
   }
   cout << "a 的准确值是 " << a << endl;

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
没有匹配的值
a 的准确值是 100
```

switch语句

一个 **switch** 语句允许测试一个变量等于多个值时的情况。每个值称为一个 **case**，且被测试的变量会对每个 **switch case** 进行检查。

语法

C++ 中 **switch** 语句的语法：

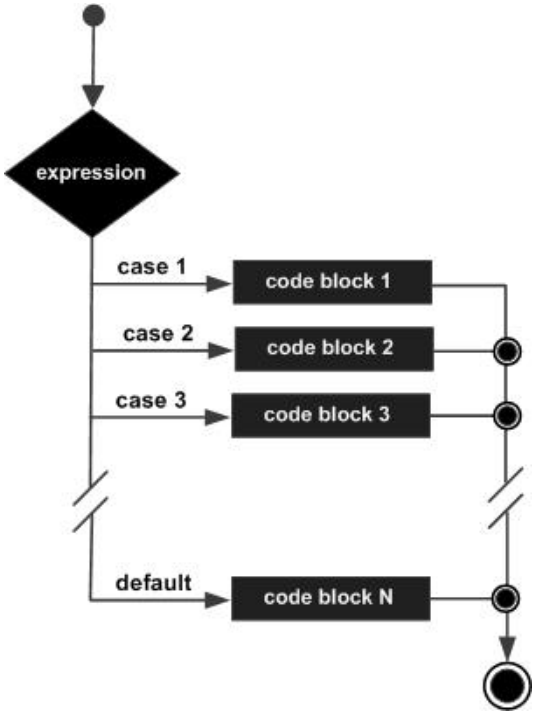
```
switch(expression){
    case constant-expression  :
       statement(s);
       break; // 可选的
    case constant-expression  :
       statement(s);
       break; // 可选的

    // 您可以有任意数量的 case 语句
    default : // 可选的
       statement(s);
}
```

switch 语句必须遵循下面的规则：

- **switch** 语句中的 **expression** 必须是一个**整型或枚举类型**，或者是一个 class 类型，其中 class 有一个单一的**转换函数**将其转换为**整型或枚举类型**。

- 在一个 switch 中可以有**任意数量的 case 语句。**每个 case 后跟一个要比较的值和一个冒号。
- case 的 **constant-expression** 必须与 switch 中的变量具有相同的数据类型，且必须是一个常量或字面量。
- 当被测试的变量等于 case 中的常量时，case 后跟的语句将被执行，直到遇到 **break** 语句为止。
- 当遇到 **break** 语句时，switch 终止，控制流将跳转到 switch 语句后的下一行。
- 不是每一个 case 都需要包含 **break。**如果 case 语句不包含 **break**，控制流将会 *继续* 后续的 case，**直到遇到 break 为止。**
- 一个 **switch** 语句可以有一个可选的 **default** case，出现在 switch 的结尾。default case 可用于在上面所有 case 都不为真时执行一个任务。default case 中的 **break** 语句不是必需的。

流程图




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

int main ()
{
   // 局部变量声明
   char grade = 'D';
```

```
   switch(grade)
   {
   case 'A' :
      cout << "很棒！" << endl;
      break;
   case 'B' :
   case 'C' :
      cout << "做得好" << endl;
      break;
   case 'D' :
      cout << "您通过了" << endl;
      break;
   case 'F' :
      cout << "最好再试一下" << endl;
      break;
   default :
      cout << "无效的成绩" << endl;
   }
   cout << "您的成绩是 " << grade << endl;

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
您通过了
您的成绩是 D
```

switch语句的嵌套

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

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

   switch(a) {
      case 100:
         cout << "这是外部 switch 的一部分" << endl;
```

```
         switch(b) {
            case 200:
               cout << "这是内部 switch 的一部分" << endl;
         }
   }
   cout << "a 的准确值是 " << a << endl;
   cout << "b 的准确值是 " << b << endl;

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
这是外部 switch 的一部分
这是内部 switch 的一部分
a 的准确值是 100
b 的准确值是 200
```

函数

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

函数定义与声明

函数定义

C++ 中的函数定义的一般形式如下：

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

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

- **返回类型**：一个函数可以返回一个值。**return_type** 是函数返回的值的数据类型。

有些函数执行所需的操作而不返回值，在这种情况下，return_type 是关键字 **void**。

- **函数名称**：这是函数的实际名称。函数名和参数列表一起构成了函数签名。
- **参数**：参数就像是占位符。当函数被调用时，主程序向参数传递一个值，这个值被称为**实参**。
- **参数列表**包括**函数参数的类型、顺序、数量**。参数是可选的，也就是说，函数可能不包含参数。
- **函数主体**：函数主体包含一组定义函数执行任务的语句。
- 标准库函数可以被重新定义，原有定义会被覆盖

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

```
// 函数返回两个数中较大的那个数

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

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

   return result;
}
```

函数声明

函数**声明**会告诉编译器**函数名称**及**如何调用函数**。函数的实际主体可以单独定义。

也就是说，可以先给出函数的声明，再进行定义

函数声明包括以下几个部分：

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

针对上面定义的函数 max()，以下是函数声明：

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

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

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

在一个源文件中定义函数且在另一个文件中**调用函数**时，函数声明是必需的且要在**调用函数的文件顶部声明函数。**

调用函数

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

当程序调用函数时，由**被调函数**执行其所定义的任务，结束后再继续执行**主调函数**

调用函数时，传递所需参数，如果函数返回一个值，则可以存储返回值。例如：

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

// 函数声明
int max(int num1, int num2);

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

   // 调用函数来获取最大值
   ret = max(a, b);

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

   return 0;
}

// 函数返回两个数中较大的那个数
int max(int num1, int num2)
{
   // 局部变量声明
```

```
   int result;

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

   return result;
}
```

把 max() 函数和 main() 函数放一块，编译源代码。当运行最后的可执行文件时，会产生下列结果：

```
Max value is : 200
```

函数的嵌套与递归

函数嵌套

函数嵌套指的是**函数嵌套调用**，即被调函数可以成为另一个函数的**主调函数**

函数**不允许嵌套定义。**

函数递归

递归调用是一种**特殊的嵌套调用。**在定义一个函数的过程中**直接调用**或**间接调用函数本身**称为函数的递归调用，这样的函数称为**递归函数。**例如：

```
#include <stdio.h>

int fibonaci(int i)
{
   if(i == 0)
   {
      return 0;
   }
   if(i == 1)
   {
      return 1;
   }
   return fibonaci(i-1) + fibonaci(i-2);
}

int  main()
```

```
{
    int i;
    for (i = 0; i < 10; i++)
    {
       printf("%d\t\n", fibonaci(i));
    }
    return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
0
1
1
2
3
5
8
13
21
34
```

递归的特点

- 每级函数调用都有**各自的变量**，这些变量同名但实际上并不是一个变量。
- 递归调用通常用**堆栈**来实现。每次递归后的函数都会放在**前一次递归后的函数上面**，

达到递归终止条件后，最上面的函数开始执行操作然后**丢出堆栈**，一直到最下面的函数。

- 递归函数中**递归调用前的语句**，按**被调函数的顺序**执行；递归函数中**递归调用后的语句**，按**被调函数相反的顺序**执行
- **每级**递归函数的调用都会**从头开始**
- 递归函数必须**包含能让递归调用停止的语句。**

函数参数

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

形式参数就像函数内的**其他局部变量**，在进入函数时被创建，退出函数时被销毁。

传递函数形参的方式如下

传值调用

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

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

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

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

   return;
}
```

现在，通过传递实际参数来调用函数 **swap()**：

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

// 函数声明
void swap(int x, int y);

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

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

   // 调用函数来交换值
   swap(a, b);

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

```
   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

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

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

指针调用

向函数传递参数的**指针调用**方法，把参数的地址复制给形式参数。

在函数内，该地址用于**访问调用中要用到的实际参数。**这意味着，**修改形式参数会影响实际参数。**

按指针传递值，参数指针被传递给函数，就像传递其他值给函数一样。

因此相应地，在下面的函数 **swap()** 中，要声明函数参数为指针类型，该函数用于交换参数所指向的两个整数变量的值。

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

   return;
}
```

现在，通过指针传值来调用函数 **swap()**：

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

```
// 函数声明
void swap(int *x, int *y);

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

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

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

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

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

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

引用调用

向函数传递参数的**引用调用**方法，把**引用的地址**复制给形式参数。

在函数内，该引用用于访问调用中要用到的实际参数。这意味着，**修改形式参数会影响实际参数。**

按引用传递值，参数引用被传递给函数，就像传递其他值给函数一样。

因此相应地，在下面的函数 **swap()** 中，要声明函数参数为引用类型，该函数用于交换参数所指向的两个整数变量的值。

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

   return;
}
```

现在，让我们通过引用传值来调用函数 **swap()**：

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

// 函数声明
void swap(int &x, int &y);

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

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

   /* 调用函数来交换值 */
   swap(a, b);

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

   return 0;
}

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

```
   return;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

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

参数的默认值

定义函数时可以为参数列表中后的每一个参数**指定默认值。**

调用函数时，如果**未传递参数的值**，则会**使用默认值**；如果指定了值，则会忽略默认值，使用传递的值。

这是通过在函数定义中使用赋值运算符来为参数赋值的。例如：

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

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

  result = a + b;

  return (result);
}

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

   // 调用函数来添加值
   result = sum(a, b);
   cout << "Total value is :" << result << endl;

   // 再次调用函数
```

```
   result = sum(a);
   cout << "Total value is :" << result << endl;

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

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

当一个函数需要声明时，默认值应设置在**函数的声明语句**中，而不是函数的定义中。没有声明时，默认值在**函数的定义**中

指定默认值后，调用该函数时**被指定默认值的参数可以不用填写，**但有默认值的形参均应写在**形参列表右侧。**

换言之，**有默认值**的形参**右边**的形参均为**有默认值**的形参

不同的源文件中，对**同一个函数的声明**可以对它的**同一个参数指定不同的默认值**；

Lambda 函数与表达式

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

- 首先，编译器**隐式地为λ表达式定义一个类**；
- 然后，为该类**重载函数调用操作符 ( )**，其功能**即λ表达式**；
- 最后，**隐式地创建该类的对象，并调用匿名函数。**

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

这在作为**STL算法**的谓词时非常有用

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

```
[capture](parameters)->return-type{body}//有返回值
[capture](parameters){body}//没有返回值
```

例如：

```
[](int x, int y){ return x < y ; }//有返回值
[]{ ++global_x; } //没有返回值
```

->返回值类型 可以省略，此时编译器根据值**自动确定返回值类型。**

在一个更为复杂的例子中，返回类型可以被明确的指定如下：

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

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

如**定义后不立即调用**，可以定义一个**auto变量并用λ表达式赋值**，这个变量起到函数的作用，例如：

```
int main ()
{
    auto add = [ ](int x,int y) -> int { return x+y;};
        cout << add(1,2) << endl;//add储存了这个函数的行为
}
```

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

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

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

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

重载函数

在同一个作用域内，可以声明几个功能类似的**同名函数**，但是这些同名函数的**形式参数**（指参数的个数、类型或者顺序）**必须不同。**

不能仅通过返回类型的不同来重载函数。

重载函数匹配的要求

精确匹配

指实参与某个重载函数的**形参类型完全相同或性质上相同。**性质上相同包括**数组名代表对应类型指针，函数名代表函数指针**

提升匹配

对实参进行**整型提升**，或者**精度提高再匹配**

标准转换匹配

对实参进行标准转换匹配，下面的五条规则优先级相等。如果同时满足几个重载函数，那么这样的**重载函数调用无效。**具体如下：

- 任何**算术型间**可以互相转换
- **枚举类型**可以转换成**算术型**
- **零**可以转化成**算术型或者指针类型**
- **任何类型的指针**都可以转化成**void指针**
- **派生类指针**可以转化成**基指针**

重载函数不能同时为带默认值函数

下面的实例中，同名函数 **print()** 被用于输出不同的数据类型：

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

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

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

```
      }

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

int main(void)
{
   printData pd;

   // 输出整数
   pd.print(5);
   // 输出浮点数
   pd.print(500.263);
   // 输出字符串
   char c[] = "Hello C++";
   pd.print(c);

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
整数为: 5
浮点数为: 500.263
字符串为: Hello C++
```

内联函数

形式上类似于函数，效率上类似于类函数宏。将要多次使用的小函数定义为内联可提高效率。

```
inline void fun()
```

1. 内联函数最好为**小函数**。inline只是对编译器的建议，函数过大如包含循环体或switch语句编译器将不予理会。一般**递归函数**不能定义为内联函数。
2. 内联函数有**文件作用域**，因此在**每个文件中调用都必须有其定义**，不止声明。可以调用包含其定义的头文件。

3. 内联函数会在**编译时展开**

数组

数组简介

C++ 支持**数组**数据结构，它可以存储**一个固定大小的相同类型元素**的顺序集合。

数组的声明是声明一个**数组变量**，比如 numbers，然后使用 numbers[0]、numbers[1]、...、numbers[99] 来代表一个个单独的变量。

数组中的特定元素可以通过索引访问。

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

声明数组

以一维数组为例，在 C++ 中要声明一个数组，需要指定元素的类型和元素的数量，如下所示：

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

arraySize 必须是一个大于零的整数常量，**type** 可以是任意有效的 C++ 数据类型。

例如，要声明一个类型为 double 的包含 10 个元素的数组 **balance**，声明语句如下：

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

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

也可以通过typedef，为数组类型专门定义一个别名

```
typedef int A[10];//有一个名为A的数组类型
A a;//a是一个有十个元素的数组
```

初始化数组

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

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

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

如果省略了数组的大小，数组的大小则为**初始化时元素的个数。**因此，如果：

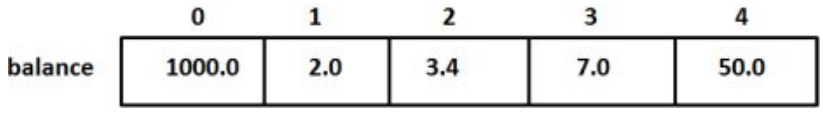
```
double balance[] = {1000.0, 2.0, 3.4, 7.0, 50.0};
```

下面是一个为数组中某个元素赋值的实例：

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

上述的语句把数组中第五个元素的值赋为 50.0。所有的数组都是以 0 作为它们第一个元素的索引，也被称为基索引，

数组的最后一个索引是数组的总大小减去 1。以下是上面所讨论的数组的的图形表示：



访问数组元素

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

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

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

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

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	您可以通过指定不带索引的数组名称来生成一个指向数组中第一个元素的指针。
传递数组给函数	您可以通过指定不带索引的数组名称来给函数传递一个指向数组的指针。
从函数返回数组	C++ 允许从函数返回数组。

多维数组

C++ 支持多维数组。多维数组声明的一般形式如下：

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

例如，下面的声明创建了一个三维 5 . 10 . 4 整型数组：

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

通常较常用一维数组，二维数组

二维数组

二维数组的定义

多维数组最简单的形式是**二维数组**。声明一个 x 行 y 列的二维整型数组，形式如下：

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

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

下面是一个二维数组，包含 3 行和 4 列：

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

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

存储数组**按行优先。**先存储第0行，再存储第1行，直至最后一行。

可以认为，二维数组是由**一维数组构造而成。**二维数组中的一维数组**不能单独引用**，如a[0][1]中不能调出a[0]

初始化二维数组

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

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

内部嵌套的括号是可选的，下面的初始化与上面是等同的：

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

若为**全部元素提供初值**，内层的花括号可以省去，将所有初始数据**按行为序**写在一个花括号内，且定义数组时**可以省略第一维长度。**

访问二维数组元素

二维数组中的元素是通过使用下标（即数组的行索引和列索引）来访问的。例如：

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

上面的语句将获取数组中第 3 行第 4 个元素。

指向数组的指针

在下面的声明中：

```
double runoobAarray[50];
```

runoobAarray 是第一个元素的地址。

因此，下面的程序片段把 **p** 赋值为 **runoobAarray** 的第一个元素的地址：

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

p = runoobAarray;
```

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

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

int main ()
{
   // 带有 5 个元素的双精度浮点型数组
   double runoobAarray[5] = {1000.0, 2.0, 3.4, 17.0, 50.0};
   double *p;

   p = runoobAarray;

   // 输出数组中每个元素的值
   cout << "使用指针的数组值 " << endl;
   for ( int i = 0; i < 5; i++ )
   {
       cout << "*(p + " << i << ") : ";
       cout << *(p + i) << endl;
   }

   cout << "使用 runoobAarray 作为地址的数组值 " << endl;
   for ( int i = 0; i < 5; i++ )
   {
       cout << "*(runoobAarray + " << i << ") : ";
       cout << *(runoobAarray + i) << endl;
   }

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
使用指针的数组值
*(p + 0) : 1000
*(p + 1) : 2
*(p + 2) : 3.4
*(p + 3) : 17
*(p + 4) : 50
使用 runoobAarray 作为地址的数组值
*(runoobAarray + 0) : 1000
*(runoobAarray + 1) : 2
*(runoobAarray + 2) : 3.4
*(runoobAarray + 3) : 17
*(runoobAarray + 4) : 50
```

传递数组给函数

C++ 中可以通过指定**不带索引的数组名**来传递**一个指向数组的指针。**

C++ 传数组给一个函数，数组类型自动转换为指针类型，因而**传的实际是地址。**

如果要在函数中传递一个一维数组作为参数，必须以下面三种方式来声明函数形式参数，

这三种声明方式的结果是一样的，因为每种方式都会告诉编译器将要接收一个整型指针。同样地，也可以传递一个多维数组作为形式参数。

传入二维数组时，**形参的列数**必须要写，否则，**无法计算x[i][j]的内存地址**

方式 1

形式参数是一个指针：

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

方式 2

形式参数是一个已定义大小的数组：

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

方式 3

形式参数是一个未定义大小的数组：

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

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

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

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

  avg = double(sum) / size;

  return avg;
}
```

现在，让我们调用上面的函数，如下所示：

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

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

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

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

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

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
平均值是:  214.4
```

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

使用数组区间的函数

可以通过向函数传递两个指针，一个指针指向数组开头，另一个标识数组的尾部。

要注意的是，为了辨识方便，标识尾部的指针指向数组最后一个元素的下一个指针，这样在说明数组尾部的指针时，只需要数组开头的指针加上数组长度即可，如下

```
int sum_arr(const int* begin,const int* end)
{
        const int*pt;
        int total=0;
        for(pt=begin;pt!=end;pt++)
        total+=*pt;
        return total;
}
int cookies[8]={1,2,4,8,16,32,64,128};
int sum=sum_arr(cookies,cookies+8)//”超尾“的数组末尾指针
```

函数返回数组

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

如果要从函数返回一个一维数组，必须声明一个返回指针的函数，如下：

```
int* myFunction()
{
int myArray[3] = {1, 2, 3};
return myArray;
}
```

如果简单地返回指向局部数组的指针，当函数结束时，局部数组将被销毁，指向它的指针将变得无效。

为了避免以上情况，要使用静态数组或者动态分配数组。

使用**静态数组**需要在函数内部创建一个**静态数组**，并将其地址返回，例如：

```
int* myFunction()
{
static int myArray[3] = {1, 2, 3};
return myArray;
}
```

下面的函数，它会生成 10 个随机数，并使用数组来返回它们，具体如下：

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

using namespace std;

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

  // 设置种子
  srand( (unsigned)time( NULL ) );
  for (int i = 0; i < 10; ++i)
```

```
  {
    r[i] = rand();
    cout << r[i] << endl;
  }

  return r;
}

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

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

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

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

```
*(p + 8) : 1820354158
*(p + 9) : 667126415
```

使用动态分配数组需要在函数内部使用 new 运算符来分配一个数组，并在函数结束时使用 delete 运算符释放该数组，例如：

实例

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

int* createArray(int size) {
    int* arr = new int[size];
    for (int i = 0; i < size; i++) {
        arr[i] = i + 1;
    }
    return arr;
}

int main() {
    int* myArray = createArray(5);
    for (int i = 0; i < 5; i++) {
        cout << myArray[i] << " ";
    }
    cout << endl;
    delete[] myArray; // 释放内存
    return 0;
}
```

在上面的例子中，我们声明了一个名为 createArray 的函数，它接受一个整数参数 size，并返回一个由整数填充的整数数组。我们使用 new 运算符在堆上动态分配了一个数组，并在函数内部填充了数组。最后，函数返回了指向数组的指针。

在 main 函数中，我们调用了 createArray 函数，并将返回的数组指针存储在 myArray 中。然后我们遍历了数组并打印了每个元素的值。最后，我们使用 delete[] 运算符释放了 myArray 所占用的内存，以避免内存泄漏。

当上面的代码被编译和执行时，它会产生下列结果：

```
1 2 3 4 5
```

当使用动态分配数组时，调用函数的代码负责释放返回的数组。这是因为在函数内部分配的数组在函数结束时不会自动释放。

字符串

C风格字符串

C 风格的字符串起源于 C 语言，并在 C++ 中继续得到支持。

字符串实际上是使用 **null** 字符 **\0** 终止的一维字符数组。

因此，一个以 null 结尾的字符串，包含了组成字符串的字符。

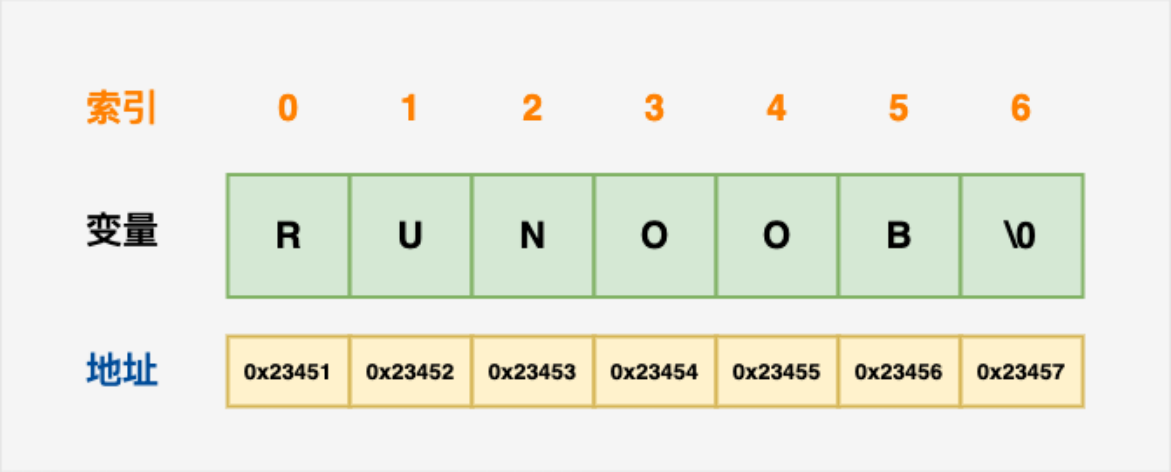
下面的声明和初始化创建了一个 **RUNOOB** 字符串。由于在数组的末尾存储了空字符，所以字符数组的大小比单词 **RUNOOB** 的字符数**多一个**。

```
char site[7] = {'R', 'U', 'N', 'O', 'O', 'B', '\0'};
```

依据数组初始化规则，可以把上面的语句写成以下语句：

```
char site[] = "RUNOOB";
```

以下是 C/C++ 中定义的字符串的内存表示：



其实，不需要把 **null** 字符放在字符串常量的末尾。C++ 编译器会在初始化数组时，自动把 **\0** 放在字符串的末尾。

<string.h>

C++包含了C的标准库<string.h>，内容如下：

库变量

下面是头文件 string.h 中定义的变量类型：

序号	变量 & 描述
1	**size_t** 这是无符号整数类型，它是 **sizeof** 关键字的结果。

库宏

下面是头文件 string.h 中定义的宏：

序号	宏 & 描述
1	**NULL** 这个宏是一个空指针常量的值。

常用string.h函数

下面是头文件 string.h 中定义的函数：

序号	函数 & 描述
*3	void *memcpy(void *dest, const void *src, size_t n) 从 src 复制 n 个字符到 *dest*。
*4	void *memmove(void *dest, const void *src, size_t n) 另一个用于从 *src* 复制 n 个字符到 *dest* 的函数。
*5	void *memset(void *str, int c, size_t n) 复制字符 c（一个无符号字符）到参数 *str* 所指向的字符串的前 n 个字符。
*6	char *strcat(char *dest, const char *src) 把 *src* 所指向的字符串追加到 *dest* 所指向的字符串的结尾。
*9	int strcmp(const char *str1, const char *str2) 把 *str1* 所指向的字符串和 *str2* 所指向的字符串进行比较。
10	int strncmp(const char *str1, const char *str2, size_t n) 把 *str1* 和 *str2* 进行比较，最多比较前 n 个字节。
*12	char *strcpy(char *dest, const char *src) 把 *src* 所指向的字符串复制到 *dest*。
*13	char *strncpy(char *dest, const char *src, size_t n) 把 *src* 所指向的字符串复制到 *dest*，最多复制 n 个字符。
*16	size_t strlen(const char *str) 计算字符串 str 的长度，直到空结束字符，但不包括空结束字符。
*17	char *strpbrk(const char *str1, const char *str2) 检索字符串 *str1* 中第一个匹配字符串 *str2* 中字符的字符，不包含空结束字符。也就是说，依次检验字符串 str1 中的字符，当被检验字符在字符串 str2 中也包含时，则停止检验，并返回该字符位置。
*20	char *strstr(const char *haystack, const char *needle) 在字符串 *haystack* 中查找第一次出现字符串 *needle*（不包含空结束字符）的位置。

此外，还有输入函数gets(),输出函数puts()

C++的string类

string类是C++标准库中的一个类，用于表示字符串。

通常而言，一个常量字符串不宜用string类，而应该用常量字符数组代替，

string类也可以使用STL接口，它包含begin(),end()等成员

下面是string类的成员函数及示范代码(构造和析构省略)

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

int main() {
    string sample[] = {"abcdef", "abcdef", "abcdef", "abcdef"};

    // 1. 使用 "=" 成员函数
    sample[0] = "xyz"; // 将第一个字符串替换为 "xyz"
    cout << "sample[0] = " << sample[0] << endl; // 输出 "sample[0] = xyz"
    cout << "sample[1] = " << sample[1] << endl; // 输出 "sample[1] = abcdef"

    // 2. 使用 "Swap" 成员函数
    sample[0].swap(sample[1]); // 交换第一个和第二个字符串
    cout << "sample[0] = " << sample[0] << endl; // 输出 "sample[0] = abcdef"
    cout << "sample[1] = " << sample[1] << endl; // 输出 "sample[1] = xyz"

    // 3. 使用 "+=" 成员函数
    sample[2] += "123"; // 在第三个字符串的末尾添加 "123"
    cout << "sample[2] = " << sample[2] << endl; // 输出 "sample[2] = abcdef123"

    // 4. 使用 "insert" 成员函数
    sample[3].insert(3, "123"); // 在第四个字符串的第三个字符位置插入 "123"
    cout << "sample[3] = " << sample[3] << endl; // 输出 "sample[3] = abc123def"

    // 5. 使用 "erase" 成员函数
    sample[0].erase(1, 2); // 删除第一个字符串的第1到第2个字符
    cout << "sample[0] = " << sample[0] << endl; // 输出 "sample[0] = adef"

    // 6. 使用 "clear" 成员函数
    sample[1].clear(); // 清空第二个字符串
    cout << "sample[1] = " << sample[1] << endl; // 输出 "sample[1] = "

    // 7. 使用 "resize" 成员函数
    sample[2].resize(5); // 将第三个字符串的长度缩短为5
    cout << "sample[2] = " << sample[2] << endl; // 输出 "sample[2] = abcde"

    // 8. 使用 "replace" 成员函数
    sample[3].replace(3, 3, "XYZ"); // 将第四个字符串的第3到第5个字符替换为 "XYZ"
    cout << "sample[3] = " << sample[3] << endl; // 输出 "sample[3] = abcXYZef"
```

```
    // 9. 使用 "+" 运算符
    string concat = sample[2] + sample[3]; // 将第三个字符串和第四个字符串拼接起来
    cout << "concat = " << concat << endl; // 输出 "concat = abcdeabcXYZef"

    // 10. 使用比较运算符 "==", "!=", "<", "<=", ">", ">="
    if (sample[0] == sample[1]) {
        cout << "sample[0] == sample[1]"
        } else {
    cout << "sample[0] != sample[1]" << endl; // 输出 "sample[0] != sample[1]"
        }

        if (sample[2] < sample[3]) {
    cout << "sample[2] < sample[3]" << endl; // 输出 "sample[2] < sample[3]"
} else {
    cout << "sample[2] >= sample[3]" << endl; // 输出 "sample[2] >= sample[3]"
}

// 11. 使用 "size" 和 "length" 成员函数
cout << "sample[0].size() = " << sample[0].size() << endl; // 输出 "sample[0].size()
= 4"
cout << "sample[1].length() = " << sample[1].length() << endl; // 输出 "sample[1].le
ngth() = 0"

// 12. 使用 "max_size" 成员函数
cout << "sample[0].max_size() = " << sample[0].max_size() << endl; // 输出 "sample
[0].max_size() = 18446744073709551615"

// 13. 使用 "empty" 成员函数
if (sample[1].empty()) {
    cout << "sample[1] is empty" << endl; // 输出 "sample[1] is empty"
} else {
    cout << "sample[1] is not empty" << endl;
}

// 14. 使用 "capacity" 成员函数
cout << "sample[0].capacity() = " << sample[0].capacity() << endl; // 输出 "sample
[0].capacity() = 4"

// 15. 使用 "reserve" 成员函数
sample[0].reserve(10); // 请求第一个字符串的容量至少为10
cout << "sample[0].capacity() = " << sample[0].capacity() << endl; // 输出 "sample
[0].capacity() = 10"

// 16. 使用 "[]" 运算符
sample[2][0] = 'x'; // 修改第三个字符串的第一个字符
cout << "sample[2] = " << sample[2] << endl; // 输出 "sample[2] = xbcde"

// 17. 使用 ">>" 运算符
```

```
cin >> sample[1]; // 从标准输入读入一个字符串并赋值给第二个字符串
cout << "sample[1] = " << sample[1] << endl; // 输出 "sample[1] = {输入的字符串}"

// 18. 使用 "<<" 运算符
cout << "sample[3] = " << sample[3] << endl; // 输出 "sample[3] = abcXYZef"

// 19. 使用 "copy" 成员函数
char buffer[6]; // 声明一个长度为6的字符数组
sample[2].copy(buffer, 5); // 将第三个字符串的前5个字符复制到 buffer 中
buffer[5] = '\0'; // 在 buffer 的末尾添加一个 null 字符，以使其成为一个 C 风格字符串
cout << "buffer = " << buffer << endl; // 输出 "buffer = xbcd"

// 20. 使用 "c_str" 成员函数
const char* cstr = sample[3].c_str(); //
cout << "cstr = " << cstr << endl; // 输出 "cstr = abcXYZef"

// 21. 使用 "data" 成员函数
char* data = &sample[2][0]; // 获取第三个字符串的字符数组地址
cout << "data = " << data << endl; // 输出 "data = xbcde"

// 22. 使用 "substr" 成员函数
string substr = sample[3].substr(3, 3); // 获取第四个字符串从第4个字符开始长度为3的子串
cout << "substr = " << substr << endl; // 输出 "substr = XYZ"

// 23. 使用 "find" 成员函数
size_t pos = sample[3].find("XYZ"); // 在第四个字符串中查找子串 "XYZ"，返回其起始位置
if (pos != string::npos) { // 若找到则输出位置，否则输出未找到
    cout << "pos = " << pos << endl; // 输出 "pos = 3"
} else {
    cout << "not found" << endl;
}

// 24. 使用 "begin" 和 "end" 成员函数
for (auto it = sample[2].begin(); it != sample[2].end(); ++it) {
    cout << *it << " "; // 逐个输出第三个字符串的字符
}
cout << endl; // 输出一个换行符

// 25. 使用 "rbegin" 和 "rend" 成员函数
for (auto it = sample[0].rbegin(); it != sample[0].rend(); ++it) {
    cout << *it << " "; // 逆序输出第一个字符串的字符
}
cout << endl; // 输出一个换行符

// 26. 使用 "get_allocator" 成员函数
allocator<char> alloc = sample[0].get_allocator(); // 获取第一个字符串的内存分配器
char* p = alloc.allocate(5); // 从该分配器中分配5个字符的内存
alloc.deallocate(p, 5); // 将该内存归还给分配器
```

```
return 0;
}
```

raw字符串

raw字符串是不使用转义字符输入的字符串，如输入\n时，就相当于字符'\'和'n'，使用如下

```
cout<<R("abcde\ns")<<endl;
```

如果要输入括号,可以用+*来标注字符串的起始位置和终止位置，使用如下

```
cout<<R"+*("(who would'nt?)",she whispered.)+*"
```

输出为

```
"(who would'nt?)",she whispered.
```

指针

每一个变量都有一个内存位置，每一个内存位置都定义了可使用连字号（&）运算符访问的地址，它表示了在内存中的一个地址。

指针是一个变量，其值为另一个变量的地址，即，**内存位置的直接地址。**

声明与初始化

像其他变量或常量一样，在使用指针存储其他变量地址之前，对其进行声明。指针变量声明的一般形式为：

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

在这里，**type** 是指针的基类型，它必须是一个有效的 C++ 数据类型，

var-name 是指针变量的名称。用来声明指针的星号 * 与乘法中使用的星号是相同的。

但是，在这个语句中，星号是用来指定一个变量是指针。以下是有效的指针声明：

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

所有指针的值的实际数据类型，不管是整型、浮点型、字符型，还是其他的数据类型，都是一样的，都是一个代表内存地址的长的十六进制数。

不同数据类型的指针之间唯一的不同是，指针所指向的变量或常量的数据类型不同。

在声明指针时可以声明多级指针，即一个存放另一个指针变量地址的指针。这样的指针可用于指向一个多维数组：

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

在声明的基础上，可以同时用一个地址**初始化**指针。

NULL初始化指针

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

NULL 指针是一个定义在标准库中的值为零的常量。请看下面的程序：

```
#include <iostream>

using namespace std;

int main ()
{
int *ptr = NULL;

cout << "ptr 的值是 " << ptr ;

return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
ptr 的值是 0
```

在大多数的操作系统上，程序**不允许访问地址为 0** 的内存，因为该内存是操作系统保留的。

然而，内存地址 0 有特别重要的意义，它表明该指针不指向一个可访问的内存位置。但按照惯例，如果指针包含空值（零值），则假定它不指向任何东西。

检查一个空指针，可以使用 if 语句，如下所示：

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

如果所有未使用的指针都被赋予空值，同时避免使用空指针，就可以**防止误用一个未初始化**的指针。

因为很多时候，**未初始化的变量存有一些垃圾值**，导致程序难以调试。

相关运算

算术运算

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

```
ptr++
```

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

递增和与整数加法

每当指针进行一次递增运算，它都会指向下一个内存位置，移动字节的多少视变量类型而定

进行加法可以视为进行相应次数的递增

下面的程序递增变量指针，以便顺序访问数组中的每一个元素：

```
#include <iostream>

using namespace std;
```

```
const int MAX = 3;

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

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

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

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

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

递减和与整数减法

同样地，对指针进行递减和减法运算，即把值减去其数据类型的字节数，如下所示：

```
#include <iostream>

using namespace std;
const int MAX = 3;

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

```
   int  *ptr;

   // 指针中最后一个元素的地址
   ptr = &var[MAX-1];
   for (int i = MAX; i > 0; i--)
   {
      cout << "Address of var[" << i << "] = ";
      cout << ptr << endl;

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

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

```
Address of var[3] = 0xbfdb70f8
Value of var[3] = 200
Address of var[2] = 0xbfdb70f4
Value of var[2] = 100
Address of var[1] = 0xbfdb70f0
Value of var[1] = 10
```

与基类型相同的指针相减

两个指向相同数组的指针间可以相减，得到的结果为两个指针所指数组元素的下标之差

关系运算

指针可以用关系运算符进行比较，如 ==、< 和 >。

当且仅当p1 和 p2 指向**两个相关的变量**，比如**同一个数组中的不同元素**，则可对 p1 和 p2 进行大小比较。

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

```
#include <iostream>

using namespace std;
const int MAX = 3;
```

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

   // 指针中第一个元素的地址
   ptr = var;
   int i = 0;
   while ( ptr <= &var[MAX - 1] )
   {
      cout << "Address of var[" << i << "] = ";
      cout << ptr << endl;

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

      // 指向上一个位置
      ptr++;
      i++;
   }
   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Address of var[0] = 0xbfce42d0
Value of var[0] = 10
Address of var[1] = 0xbfce42d4
Value of var[1] = 100
Address of var[2] = 0xbfce42d8
Value of var[2] = 200
```

关系运算

取地址和解引用运算符

取地址运算符&

&的运算对象只能是**变量名、数组元素、结构体成员，不能是表达式。**运算结果即为对象的地址。

解引用运算符*

也称为**间接访问运算符。***的运算对象只能是指针变量，其运算结果得到**运算对象所指变量。**如p1=&a，p1=a。

指针在声明时使用空格，在解引用变量时省略空格

指针与函数

与其他变量一样，指针可以作为函数的形参或返回值。

作为**返回值**时，函数会**返回一个地址**

不能把**函数体**内**局部变量的地址**作为返回给**调用者的指针。**因为在函数执行完后，**局部变量在栈中的地址空间就被收回了。**

函数指针与回调函数

函数指针

函数指针是**指向函数的指针变量。**

通常我们说的指针变量是指向一个整型、字符型或数组等变量，而函数指针是指向函数。事实上，函数名也是一个地址

函数指针可以像一般函数一样，用于**调用函数、传递参数。**

函数指针变量的声明：

```
typedef int (*fun_ptr)(int,int); // 声明一个指向同样参数、返回值的函数指针类型，可以看作将函数声明的函数名换成指针
```

以下实例声明了函数指针变量 p，指向函数 max：

```
#include <stdio.h>

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

int main(void)
{
    /* p 是函数指针 */
    int (* p)(int, int) = & max; // &可以省略
    int a, b, c, d;

    printf("请输入三个数字:");
```

```
    scanf("%d %d %d", & a, & b, & c);

    /* 与直接调用函数等价，d = max(max(a, b), c) */
    d = p(p(a, b), c);

    printf("最大的数字是: %d\n", d);

    return 0;
}
```

编译执行，输出结果如下：

```
请输入三个数字:1 2 3
最大的数字是: 3
```

回调函数

函数指针变量可以作为某个函数的参数来使用的，回调函数就是一个**通过函数指针调用的函数。**

简单讲：回调函数是由别人的函数执行时调用你实现的函数。

下面的实例中 **populate_array()** 函数定义了三个参数，其中第三个参数是函数的指针，通过该函数来设置数组的值。

实例中定义了回调函数 **getNextRandomValue()**，它返回一个随机值，它作为一个函数指针传递给 **populate_array()** 函数。

populate_array() 将调用 **10** 次回调函数，并将回调函数的返回值赋值给数组。

```
#include <stdlib.h>
#include <stdio.h>

void populate_array(int *array, size_t arraySize, int (*getNextValue)(void))
{
    for (size_t i=0; i<arraySize; i++)
        array[i] = getNextValue();
}

// 获取随机值
int getNextRandomValue(void)
{
```

```
    return rand();
}

int main(void)
{
    int myarray[10];
    /* getNextRandomValue 不能加括号，否则无法编译，因为加上括号之后相当于传入此参数时传入了
int , 而不是函数指针*/
    populate_array(myarray, 10, getNextRandomValue);
    for(int i = 0; i < 10; i++) {
        printf("%d ", myarray[i]);
    }
    printf("\n");
    return 0;
}
```

编译执行，输出结果如下：

```
16807 282475249 1622650073 984943658 1144108930 470211272 101027544 1457850878 1458
777923 2007237709
```

引用

引用变量是一个别名，也就是说，它是某个**已存在变量**的**另一个名字**。

一旦把引用初始化为某个变量，就可以使用该**引用名称**或**变量名称**来指向变量。

引用的声明与初始化

变量名称是变量附属在内存位置中的标签，而引用当成是变量附属在内存位置中的第二个标签。

因此，可以通过原始变量名称或引用来访问变量的内容。例如：

```
int i = 17;
```

我们可以为 i 声明引用变量，如下所示：

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

在这些声明中，& 读作**引用**。

因此，第一个声明可以读作 "r 是一个初始化为 i 的整型引用"，

第二个声明可以读作 "s 是一个初始化为 d 的 double 型引用"。

下面的实例使用了 int 和 double 引用：

```
#include <iostream>

using namespace std;

int main ()
{
   // 声明简单的变量
   int    i;
   double d;

   // 声明引用变量
   int&    r = i;
   double& s = d;

   i = 5;
   cout << "Value of i : " << i << endl;
   cout << "Value of i reference : " << r  << endl;

   d = 11.7;
   cout << "Value of d : " << d << endl;
   cout << "Value of d reference : " << s  << endl;

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

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

引用的注意事项

- **数组**不能定义引用类型
- **不存在空引用。**引用必须连接到一块合法的内存。
- 一旦引用被初始化为一个对象，就**不能被指向到另一个对象。**
- 引用**必须在创建时被初始化。**
- 引用初始化时**只能绑定左值**，不能绑定常量值。

引用作为形参

下面的实例使用了引用来实现引用调用函数。

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

// 函数声明
void swap(int& x, int& y);

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

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

   /* 调用函数来交换值 */
   swap(a, b);

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

   return 0;
}

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

```
   return;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

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

返回引用

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

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

```
#include <iostream>

using namespace std;

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

double& setValues(int i) {
   double& ref = vals[i];
   return ref;   // 返回第 i 个元素的引用，ref 是一个引用变量，ref 引用 vals[i]


}

// 要调用上面定义函数的主函数
int main ()
{

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

   setValues(1) = 20.23; // 改变第 2 个元素
```

```
   setValues(3) = 70.8;  // 改变第 4 个元素

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

当上面的代码被编译和执行时，它会产生下列结果：

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

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

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

如果希望使用**引用返回值**而不改变它，可以将函数的返回类型声明为const引用;

但如果函数的参数均为const，返回值也应该被const修饰

按值、址、引用返回的原则

对于使用传递的值但不作修改的函数

按值返回

当数据对象很小，如**内置数据类型**或**小型结构**

按址返回

当数据对象为**数组**或**较大的结构**时，返回类型应该声明为**const指针**

按引用返回

当数据对象为**较大的结构**或**类对象**，返回类型应该声明为**const引用**

对于修改调用函数中数据的函数

按址返回

当数据对象为**内置数据类型**、**数组**、**结构**时

按引用返回

当数据对象为**结构**或**类对象**时

右值引用

前面所提到的引用均为**左值引用**，右值引用即将**一个引用变量与一个计算表达式**相关联

例如我们定义一个值大小为1，1所在的地址是未知的，但如果将1赋给一个右值引用变量，就可以通过这个变量得知1的地址

判断是左值还是右值，关键在于这个值**是不是匿名**的？是不是**临时的**，使用完后马上就销毁的？

右值引用的声明

```
typename&& x;
```

移动语义

移动语义之所以要使用**右值引用**这个概念，是因为**右值**本身就是一个**将要消亡**的值，程序知道它是右值就不会为它再分配一块内存

如果用的是左值引用，程序就会再申请一块内存先复制左值再进行赋值，这就成了**复制语义**了

移动构造函数与移动赋值运算符

例如

```
class MyString {
public:
    MyString() : data(nullptr), length(0) {}

    // 移动构造函数
    MyString(MyString&& other) noexcept : data(nullptr), length(0) {
        // 从源对象中接管资源所有权
        data = other.data;
        length = other.length;

        // 将源对象置于有效但未定义的状态
        other.data = nullptr;
        other.length = 0;
    }

    // 移动赋值运算符
    MyString& operator=(MyString&& other) noexcept {
        if (this != &other) {
            // 释放当前对象的资源
            delete[] data;

            // 从源对象中接管资源所有权
            data = other.data;
            length = other.length;

            // 将源对象置于有效但未定义的状态
            other.data = nullptr;
            other.length = 0;
        }
        return *this;
    }
private:
    char* data;
    size_t length;
};
```

移动构造函数和**移动赋值运算符**应当是**noexcept**的

移动先将**other对象的数据**挪到**本对象**上，再将other对象**本身的数据置空**

如果是**赋值运算符**，还要先**判断两者是否是同一个对象**

强制移动

如果需要将左值引用于移动语义，需要标准库中的**std::move**将左值转变为**右值引用类型**，例如

```
MyString str1;
    str1.setData("Hello");

    MyString str2(std::move(str1));  //将str1转变为右值引用类型，使用移动构造函数
```

日期&时间的获取

使用日期和时间相关的函数和结构，需要在 C++ 程序中引用 **<ctime>** 头文件。

相关类型、函数

相关类型

有四个与时间相关的类型：**clock_t、time_t、size_t** 和 **tm**。类型 clock_t、size_t 和 time_t 能够把系统时间和日期表示为某种整数。

结构类型 **tm** 把日期和时间以 C 结构的形式保存，

tm 结构的定义如下：

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

库函数

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

当前日期和时间

下面的实例获取当前系统的日期和时间，包括本地时间和协调世界时（UTC）。

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

using namespace std;

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

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

   cout << "本地日期和时间：" << dt << endl;

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

当上面的代码被编译和执行时，它会产生下列结果：

```
本地日期和时间：Sat Jan  8 20:07:41 2011

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

使用结构 tm 格式化时间

tm 结构以 C 结构的形式保存日期和时间。大多数与时间相关的函数都使用了 tm 结构。

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

using namespace std;

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

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

   tm *ltm = localtime(&now);

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

当上面的代码被编译和执行时，它会产生下列结果：

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

基本输入和输出

C++的基本输入和输出

C++ 的 I/O 发生在**流中**，流是**字节序列**。

字节流从设备（如键盘、磁盘驱动器、网络连接等）流向内存称为**输入操作**。

字节流从内存流向设备（如显示屏、打印机、磁盘驱动器、网络连接等）称为**输出操作**。

I/O 库头文件

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

标准输出流（cout）

预定义的对象 **cout** 是 **iostream** 类的一个实例。

cout 对象"连接"到标准输出设备，通常是显示屏。

在cout输出流中，两个字符串间在没有<<运算符的情况下，如果中间相隔一个空格或换行，会被拼接成一个字符串

cout 是与流插入运算符 << 结合使用的，如下所示：

```
#include <iostream>

using namespace std;

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

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

当上面的代码被编译和执行时，它会产生下列结果：

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

C++ 编译器根据要输出变量的数据类型，选择合适的流插入运算符来显示值。

<< 运算符被重载来输出内置类型（整型、浮点型、double 型、字符串和指针）的数据项。

流插入运算符 << 在一个语句中可以多次使用，**endl** 用于在行末添加一个换行符并刷新缓冲区，在不需要刷新缓冲区时应该用**换行符**代替

cout.put()是类似putchar()的函数

标准输入流（cin）

预定义的对象 **cin** 是 **iostream** 类的一个实例。

cin 对象附属到标**准输入设备**，通常是键盘。

cin在读到**空字符**时便会**结束读取**，并且会将读取的内容存放在**缓冲区**直至**按下回车**，如下：

```
#include<iostream>
int main()
{
        using namespace std;
        char ch;
        int count=0;// 记录输入的字符个数
        cin>>ch;
        while(ch!='#')//输入'#'时停止输出
        {
                cout<<ch;
                ++count;
                cin>>ch;
        }
        cout<<endl<<cout;
}
```

执行该程序：

```
see ken run#really fast//输入
seekenrun
9
```

当输入空格时，**cin会停止输入**；

当输入#时，虽然停止输出，但还能继续输入，说明cin的输入会存放在**缓冲区**中。

cin 是与流提取运算符 >> 结合使用的，如下所示：

```
#include <iostream>

using namespace std;

int main( )
{
   char name[50];

   cout << "请输入您的名称：  ";
   cin >> name;
   cout << "您的名称是：  " << name << endl;

}
```

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

```
请输入您的名称：  cplusplus
您的名称是：  cplusplus
```

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

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

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

这相当于下面两个语句：

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

cin.get(char)

cin.get(ch)是一个能读取输入中的下一个字符并将其赋值给ch的函数，可以**读取空格**

cin.get()是另一种重载版本，相当于getchar()

字符串的读取

cin.getline()

cin.getline()函数读取整行，接受两个参数，

第一个参数为待输入字符串，第二个参数是读取字符个数，当输入换行时或读取字符个数已达上限时停止输入。

```
cin.getline(string name,int size)
```

cin.get()

cin.get()函数与cin.getline()函数类似，但是cin.get()会**保留回车。**

getline()

这个getline()函数与cin对象的方法cin.getline()不同，getline()接受两个参数，第一个为输入流，第二个为一个string对象,如下

```
string str;
getline(cin,str);
```

标准错误流（cerr）

预定义的对象 **cerr** 是 **iostream** 类的一个实例。

cerr 对象附属到标准输出设备，通常也是显示屏，但是 **cerr** 对象是非缓冲的，且每个流插入到 cerr 都会立即输出。

cerr 也是与流插入运算符 << 结合使用的，如下所示：

```
#include <iostream>

using namespace std;

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

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

当上面的代码被编译和执行时，它会产生下列结果：

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

标准日志流（clog）

预定义的对象 **clog** 是 **iostream** 类的一个实例。

clog 对象附属到标准输出设备，通常也是显示屏

clog 对象是缓冲的。这意味着每个流插入到 clog 都会先存储在缓冲区，直到**缓冲填满**或者**缓冲区刷新**时才会输出。

clog 也是与流插入运算符 << 结合使用的，如下所示：

```
#include <iostream>

using namespace std;

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

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

当上面的代码被编译和执行时，它会产生下列结果：

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

良好的编程实践告诉我们，使用 cerr 流来显示错误消息，而其他的日志消息则使用 clog 流来输出。

头文件<iomanip>

设置字段宽度setw(int n)

```
cout << setw(5) << 248 << endl;
运行结果
```

```
  248
```

流输出进制

dec 置基数为10 相当于"%d"
hex 置基数为16 相当于"%X"
oct 置基数为8 相当于"%o"

流输出二进制需要包含<bitset.h>头文件，调用函数bitset<int size>(int n)将十进制输出为二进制,其中的size是二进制数长度。

```
cout << hex << 255 << endl;
cout << bitset<11>(255) << endl;
运行结果
ff
00011111111
```

填充字符setfill(char c)

在预设宽度中，如果存在没有用完的宽度大小，则用字符c填充

```
cout << setfill('0') << setw(6) << 248 << endl;
运行结果：
000248
```

进制转换setbase(int n)

将某一个十进制数**转换为一个8或16进制**的数，如果是2 ~ 36任意进制还是使用**<stdlib.h>**头文件中**itoa()**函数。

```
cout<< setbase(8) << setw(5) << 255 << endl;
运行结果：
  377
```

设置小数精度setprecision(int n)

控制输出流显示浮点数的数字个数（包括整数部分），默认的流输出数值有效位是6。

```
cout << setprecision(3) << 22.123 << endl;
cout << setprecision(8) << 22.123 << endl; //比原字符长不会补零!
运行结果:
22.1
22.123
```

设置格式标志setiosflags(ios_base::fmtflags mask)

用`setiosflags(ios::fixed)`或者直接用`fixed`都行

方法	格式
setiosflags(ios::scientific)	是用指数方式表示实数
setiosflags(ios::fixed)	固定的浮点显示
setiosflags(ios::left)	左对齐
setiosflags(ios::right)	右对齐
setiosflags(ios::skipws)	忽略前导空白
setiosflags(ios::uppercase)	16进制数大写输出
setiosflags(ios::lowercase)	16进制小写输出
setiosflags(ios::showpoint)	强制显示小数点
setiosflags(ios::showpos)	强制显示符号

```
cout<< fixed << setw(10) << 3.1415926 << endl;
cout<< setiosflags(ios::fixed) << setw(10) << 3.1415926 << endl;
运行结果:
  3.141593
  3.141593
```

重置格式标志resetiosflags(ios_base::fmtflags mask)

终止已经设置的输出格式状态，在括号中应指定内容

```
cout << fixed << setw(10) << 3.1415926 << endl;
cout << resetiosflags(ios::fixed) << setw(10) << 3.1415926 << endl;
cout << hex << showbase << 100 << endl;
cout << resetiosflags(ios::showbase) << 100 << endl;
运行结果:
  3.141593
```

```
   3.14159
0x64
64
```

C的基本输入和输出

标准文件

C 语言把所有的设备都当作文件。设备（比如显示器）被处理的方式与文件相同。

以下三个文件会在程序执行时自动打开，以便访问键盘和屏幕。

标准文件	文件指针	设备
标准输入	stdin	键盘
标准输出	stdout	屏幕
标准错误	stderr	您的屏幕

getchar() & putchar() 函数

int getchar(void) 函数从屏幕读取下一个可用的字符，并把它返回为一个整数。

这个函数在同一个时间内只会读取一个单一的字符。可以在循环内使用这个方法，以便从屏幕上读取多个字符。接受的字符包括**空格**、

当希望程序在显示结果后按任意键再退出时，可在return语句前加一个**getchar**。

int putchar(int c) 函数把字符输出到屏幕上，并返回相同的字符。

这个函数在同一个时间内只会输出一个单一的字符。可以在循环内使用这个方法，以便在屏幕上输出多个字符。

如输入一个0-127的数字，会显示对应的**ascii字符**

在包含stdio头文件时，编译器将这两个函数看作**类函数宏**。

如下：

```c
#include <stdio.h>

int main( )
{
   int c;

   printf( "Enter a value :");
```

```
   c = getchar( );

   printf( "\nYou entered: ");
   putchar( c );
   printf( "\n");
   return 0;
}
```

当上面的代码被编译和执行时，它会等待输入一些文本，

当输入一个文本并按下回车键时，程序会继续并只会读取一个单一的字符，显示如下：

```
$./a.out
Enter a value :runoob

You entered: r
```

gets() & puts() 函数

char *gets(char *s) 函数从 **stdin** 读取一行到 **s** 所指向的缓冲区，直到一个终止符或 EOF。

int puts(const char *s) 函数把字符串 s 和一个尾随的换行符写入到 **stdout**。

```
#include <stdio.h>

int main( )
{
   char str[100];

   printf( "Enter a value :");
   gets( str );

   printf( "\nYou entered: ");
   puts( str );
   return 0;
}
```

当上面的代码被编译和执行时，它会等待您输入一些文本，当您输入一个文本并按下回车键时，程序会继续并读取一整行直到该行结束，显示如下：

```
$./a.out
Enter a value :runoob

You entered: runoob
```

scanf() 和 printf() 函数

C 语言中的 I/O (输入/输出) 通常使用 printf() 和 scanf() 两个函数。

scanf() 函数用于**从标准输入（键盘）读取**并格式化， printf() 函数**发送格式化输出**到标准输出（屏幕）。

int scanf(const char *format, ...) 函数从标准输入流 **stdin** 读取输入，并根据提供的 **format** 来浏览输入。

printf函数

int printf(const char *format, ...) 函数把输出写入到标准输出流 **stdout** ，并根据提供的格式产生输出。

format 可以是一个简单的常量字符串，可以分别指定 **%s、%d、%c、%f** 等来输出或读取**字符串**、**整数**、**字符**或**浮点数**。

整型格式控制符

不同进制输出

十进制格式

基本整型：%d/%md

长整型：%ld/%mld

无符号基本整型：%u/%mu

无符号长整型：%lu/%mlu

八进制形式

基本整型：%o/%mo

长整型：%lo/%mlo

十六进制形式

基本整型：%x/%mx

长整型：%lx/%mlx

输出确定字段宽度

m为整数值，**表示字段宽度**

若数据位数小于m，则左端补以空格；大于m，则按实际位数输出。

不打m，就输出数据的所有数位。

足够大的固定字段宽度可以让输出变得整齐美观。

显示各进制数的前缀

%0/%0x

限定数据字节数

只取内存中二进制格式下数据的前几个字节，可能会改变数据。

修饰符为**h。如%hd。**

指定输出short型/long型

指定**输出short型，后缀为h**；指定**输出long型，后缀为l**

补零输入

使不足字段宽度的空位全部补上0；当指定左对齐(-)或有指定精度时，忽略补0

带正号输出

当输出数据为正数时，在%后打+可以使数前面输出一个正号；不是正数没有正号

浮点型格式控制符

以小数形式输出浮点型数据

%f/%m.nf/%-m.nf

%f用于输出单精度和双精度浮点数，没给出输出数据的位数，则按系统默认6位数输出。

m表示输出数据所占的总宽度（**包括小数点所占的1列**），n表示**小数部分所占的位数。**

补空格规则同整形格式控制符。

%-m.nf跟不加负号的基本相同，但它**会使数据向左靠，右边补足空格。**

double型双精度数据输出时，还可以使用%lf，但与%f无区别。

以指数形式输出浮点型数据

%e/%m.ne 如3.5→3.50000e+000 默认保留小数位中算上e。

适用于非常大或非常小的数，输出结果比%f简洁。**只能输出浮点型数据**，输出整型就会错误。

以普通形式输出浮点型数据

%g

由系统根据数值大小**自动选用%f或%e**。若实现不能确定输出浮点数的宽度，可以用%g格式输出。不输出无意义的0.

字符型格式控制符

%c/%mc

m是输出宽度，同理。

字符串格式控制符

%s/%ms/%m.ns/%-m.ns

m表示输出宽度，n表示只输出**字符串的左边n个字符。**

scanf函数

scanf函数按指定格式从键盘读取数据并赋给指定变量。调用形式为：

scanf（“格式控制字符串”，输入项地址列表）

- 输入项地址列表由若干个地址组成，以逗号分隔。每个地址对应输入数据所要存储的内存地址，可以是变量的地址或数组名。
- **输入long int必须用%ld，输入double必须用%lf/%le。**
- **scanf中也可以使用%md、%*md的格式控制符。%md是从左向右截取m列输入。%*md是跳过m列输入。%*d是直接跳过这次输入。**
- 以**%d、%f**格式控制符输入多个数据时，数据间以一或多个空格间隔，也可用回车键、tab间隔。**用%c输入时，空格、回车键、跳格键被当作字符输入。**
- 如果格式控制字符串中还有其他字符，则在输入数据时，**对应**的位置上应输入**同样**的字符。
- 输入数值数据时，输入**空格、回车、tab或非法字符**，数据输入视为结束
- scanf检测到输入**非法字符**时，scanf返回值为**0**；检测到**文件结尾**时，会返回**EOF。**

现在让我们通过下面这个简单的实例来加深理解：

```
#include <stdio.h>
int main( ) {

   char str[100];
   int i;

   printf( "Enter a value :");
   scanf("%s %d", str, &i);

   printf( "\nYou entered: %s %d ", str, i);
   printf("\n");
   return 0;
}
```

当上面的代码被编译和执行时，它会等待您输入一些文本，当您输入一个文本并按下回车键时，程序会继续并读取输入，显示如下：

```
$./a.out
Enter a value :runoob 123

You entered: runoob 123
```

字符判断函数

这类函数被放在**ctype.h文件**中，可以用来判断字符**是否是某一类特殊的字符。**

若属于，则返回值为非零，否则为0

单字节	宽字节	描述
isalnum	iswalnum	是否为字母数字
isalpha	iswalpha	是否为字母
islower	iswlower	是否为小写字母
isupper	iswupper	是否为大写字母
isdigit	iswdigit	是否为数字
isxdigit	iswxdigit	是否为16进制数字
iscntrl	iswcntrl	是否为控制字符
isgraph	iswgraph	是否为图形字符（例如，空格、控制字符都不是）
isspace	iswspace	是否为空格字符（包括制表符、回车符、换行符等）
isblank	iswblank	是否为空白字符(C99/C++11新增)（包括水平制表符）
isprint	iswprint	是否为可打印字符
ispunct	iswpunct	是否为标点
tolower	towlower	转换为小写
toupper	towupper	转换为大写

文件输入和输出

C++文件读写

从文件读取流和向文件写入流需要用到 C++ 中的标准库 **fstream**，它定义了三个新的数据类型：

数据类型	描述
ofstream	该数据类型表示**输出文件流**，用于创建文件并向文件写入信息。
ifstream	该数据类型表示**输入文件流**，用于从**文件读取信息。**
fstream	该数据类型通常表示**文件流**，且同时具有 **ofstream** 和 **ifstream** 两种功能，这意味着它可以**创建文件**，向文件**写入信息**，从文件**读取信息。**

要在 C++ 中进行**文件处理**，必须在 C++ 源代码文件中包含头文件 **<iostream>** 和 **<fstream>**。

打开文件

在从文件读取信息或者向文件写入信息之前，必须先打开文件。

ofstream 和 **fstream** 对象都可以用来打开文件进行**写操作**，

如果只需要打开文件进行**读操作**，则使用 **ifstream** 对象。

下面是 **open() 函数的标准语法**，open() 函数是 fstream、ifstream 和 ofstream 对象的一个成员。

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

在这里，**open()** 成员函数的第一参数**指定要打开的文件的名称和位置**，第二个参数定义文件**被打开的模式。**

模式标志	描述
ios::app	**追加模式。**所有写入都**追加到文件末尾。**
ios::ate	文件打开后**定位到文件末尾。**
ios::in	打开文件用于**读取。**
ios::out	打开文件用于**写入。**
ios::trunc	如果**该文件已经存在**，其内容将在打开文件之前被截断，即把**文件长度设为 0。**

可以把以上两种或两种以上的模式结合使用。

例如，如果想要以写入模式打开文件，并希望截断文件，以防文件已存在，那么可以使用下面的语法：

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

类似地，如果想要打开一个文件用于读写，可以使用下面的语法：

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

关闭文件

当 C++ 程序终止时，它会自动关闭刷新所有流，释放所有分配的内存，并关闭所有打开的文件。

但程序员应该在程序终止前关闭所有打开的文件。

下面是 **close() 函数**的标准语法，**close() 函数**是 fstream、ifstream 和 ofstream 对象的**一个成员。**

```
void close();
```

写入和读取文件

在 C++ 编程中，我们使用流插入运算符（ << ）和流提取运算符（ >> ）向文件写入和读取信息，

与键盘输入或屏幕输出不同的是，在这里使用的是 **ofstream/ifstream** 或 **fstream** 对象，而不是 **cout** 对象。

读取 & 写入实例

下面的 C++ 程序以读写模式打开一个文件。在向文件 afile.dat 写入用户输入的信息之后，程序从文件读取信息，并将其输出到屏幕上：

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

int main ()
{

   char data[100];

   // 以写模式打开文件
   ofstream outfile;
   outfile.open("afile.dat");

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

   // 向文件写入用户输入的数据
   outfile << data << endl;

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

   // 再次向文件写入用户输入的数据
   outfile << data << endl;

   // 关闭打开的文件
   outfile.close();

   // 以读模式打开文件
   ifstream infile;
   infile.open("afile.dat");

   cout << "Reading from the file" << endl;
```

```
   infile >> data;

   // 在屏幕上写入数据
   cout << data << endl;

   // 再次从文件读取数据，并显示它
   infile >> data;
   cout << data << endl;

   // 关闭打开的文件
   infile.close();

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列输入和输出：

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

上面的实例中使用了 cin 对象的附加函数，比如 getline()函数从外部读取一行，**ignore()函数**会忽略掉之前读语句留下的**多余字符**。

文件位置指针

istream 和 **ostream** 都提供了用于重新定位文件位置指针的成员函数。

这些成员函数包括关于 istream 的 **seekg**（"seek get"）和关于 ostream 的 **seekp**（"seek put"）。

seekg 和 seekp 的参数通常是一个**长整型**。第二个参数可以用于**指定查找方向**。

查找方向可以是 **ios::beg**（**默认的**，从流的开头开始定位），也可以是 **ios::cur**（从流的当前位置开始定位），也可以是 **ios::end**（从流的末尾开始定位）。

文件位置指针是一个**整数值**，指定了从**文件的起始位置**到**指针所在位置**的**字节数**。下面是关于定位 "get" 文件位置指针的实例：

```
// 定位到 fileObject 的第 n 个字节（假设是 ios::beg）
fileObject.seekg( n );

// 把文件的读指针从 fileObject 当前位置向后移 n 个字节
fileObject.seekg( n, ios::cur );

// 把文件的读指针从 fileObject 末尾往回移 n 个字节
fileObject.seekg( n, ios::end );

// 定位到 fileObject 的末尾
fileObject.seekg( 0, ios::end );
```

C文件读写

一个文件，无论它是文本文件还是二进制文件，都是代表了一系列的字节。

C 语言不仅提供了访问顶层的函数，也提供了底层（OS）调用来处理存储设备上的文件。本章将讲解文件管理的重要调用。

打开文件

使用 **fopen( )** 函数来创建一个新的文件或者打开一个已有的文件，这个调用会初始化类型 **FILE** 的一个对象，类型 **FILE** 包含了所有用来控制流的必要的信息。

下面是这个函数调用的原型：

```
FILE *fopen( const char *filename, const char *mode );
```

在这里，**filename** 是字符串，用来命名文件，访问模式 **mode** 的值可以是下列值中的一个：

模式	描述
r	打开一个已有的文本文件，允许读取文件。
w	打开一个文本文件，允许写入文件。如果文件不存在，则会创建一个新文件。在这里，您的程序会从文件的开头写入内容。如果文件存在，则该会被截断为零长度，重新写入。
a	打开一个文本文件，以追加模式写入文件。如果文件不存在，则会创建一个新文件。在这里，您的程序会在已有的文件内容中追加内容。
r+	打开一个文本文件，允许读写文件。
w+	打开一个文本文件，允许读写文件。如果文件已存在，则文件会被截断为零长度，如果文件不存在，则会创建一个新文件。
a+	打开一个文本文件，允许读写文件。如果文件不存在，则会创建一个新文件。读取会从文件的开头开始，写入则只能是追加模式。

如果处理的是二进制文件，则需使用下面的访问模式来取代上面的访问模式：

```
"rb", "wb", "ab", "rb+", "r+b", "wb+", "w+b", "ab+", "a+b"
```

关闭文件

为了关闭文件，请使用 fclose( ) 函数。函数的原型如下：

```
int fclose( FILE *fp );
```

如果成功关闭文件，**fclose( )** 函数返回零，如果关闭文件时发生错误，函数返回 **EOF**。

这个函数实际上会清空缓冲区中的数据，关闭文件，并释放用于该文件的所有内存。EOF 是一个定义在头文件 **stdio.h** 中的常量。

C 标准库提供了各种函数来按字符或者以固定长度字符串的形式读写文件。

写入文件

fputc()

fputc()是把**字符写入到流中**的最简单的函数：

```
int fputc( int c, FILE *fp );
```

函数 **fputc()** 把参数 c 的字符值写入到 fp 所指向的输出流中。如果写入成功，它会返回写入的字符，如果发生错误，则会返回 **EOF**。

fputs()

fputs()把一个**以 null 结尾的字符串**写入到流中：

```
int fputs( const char *s, FILE *fp );
```

函数 **fputs()** 把字符串 **s** 写入到 fp 所指向的输出流中。如果写入成功，它会返回一个非负值，如果发生错误，则会返回 **EOF**。

fprintf

使用 **int fprintf(FILE *fp,const char *format, ...)** 函数把一个字符串写入到文件中。尝试下面的实例：

```
#include <stdio.h>
```

```c
int main()
{
   FILE *fp = NULL;

   fp = fopen("/tmp/test.txt", "w+");
   fprintf(fp, "This is testing for fprintf...\n");
   fputs("This is testing for fputs...\n", fp);
   fclose(fp);
}
```

当上面的代码被编译和执行时，它会在 /tmp 目录中创建一个新的文件 **test.txt**，并使用两个不同的函数写入两行。接下来让我们来读取这个文件。

读取文件

fgetc()

fgetc()是从文件读取单个字符的最简单的函数：

```c
int fgetc( FILE * fp );
```

fgetc() 函数从 fp 所指向的输入文件中读取一个字符。返回值是读取的字符，如果发生错误则返回 **EOF**。

fgets()

fgets()函数从流中读取一个字符串：

```c
char *fgets( char *buf, int n, FILE *fp );
```

函数 **fgets()** 从 fp 所指向的输入流中读取 n - 1 个字符。它会把读取的字符串复制到缓冲区 **buf**，并在最后追加一个 **null** 字符来终止字符串。

如果这个函数在读取最后一个字符之前就遇到一个换行符 '\n' 或文件的末尾 EOF，则只会返回读取到的字符，包括换行符。

fscanf()

使用 **int fscanf(FILE *fp, const char *format, ...)** 函数来从文件中读取字符串，但是在遇到**第一个空格和换行符**时，它会停止读取。

```c
#include <stdio.h>

int main()
{
   FILE *fp = NULL;
   char buff[255];

   fp = fopen("/tmp/test.txt", "r");
   fscanf(fp, "%s", buff);
   printf("1: %s\n", buff );

   fgets(buff, 255, (FILE*)fp);
   printf("2: %s\n", buff );

   fgets(buff, 255, (FILE*)fp);
   printf("3: %s\n", buff );
   fclose(fp);

}
```

```
1: This
2: is testing for fprintf...

3: This is testing for fputs...
```

首先，**fscanf()** 方法只读取了 **This**，因为它在后边遇到了一个空格。

其次，调用 **fgets()** 读取剩余的部分，直到行尾。

最后，调用 **fgets()** 完整地读取第二行。

二进制 I/O 函数

下面两个函数fread()和fwrite()用于**二进制输入和输出**

```c
size_t fread(void *ptr, size_t size_of_elements,
             size_t number_of_elements, FILE *a_file);

size_t fwrite(const void *ptr, size_t size_of_elements,
             size_t number_of_elements, FILE *a_file);
```

这两个函数都是用于存储块的读写 - 通常是数组或结构体。

文件的定位

复位函数rewind

使文件位置指示器**重新指向文件的第一个字符**

格式为：

rewind(指针变量名)

定位函数fseek

格式为：

fseek(文件指针变量名，偏移量，起点)

其中起点有0.1.2三个值。

- 取0时，以**开头**为基准偏移。
- 取1时，以**当前位置**为基准偏移。
- 取2时，以**末尾**为基准偏移。

- stdio.h宏定义SEEK_SET为0
- SEEK_CUR为1
- SEEK_END为2

偏移量为**正时向后偏，为负时向前。**

新位置**落在开头之前，自动跳到开头**;落到**末尾之后**，可能导致难以预料的错误。

一般**fseek用于二进制文件。**文本文件数据长度难以确定，易混乱。

显示位置指针函数ftell

返回位置指针**相对于文件开头的偏移量。**

格式为：

ftell(文件指针变量名)

文件结束检测函数feof

当文件处于文件结束位置时，返回1，否则0

格式为：

feof(fp)

判断出错函数ferror

调用输入输出函数出现错误时，函数返回**非零值**，无误**返回0**.

在执行fopen时，ferror初值为0.

格式为：

ferror(文件指针变量名)

文件重定向

默认情况下，C使用**标准I/O**包查找标准输入作为**输入源**，这称为stdin流。可以通过**重定向**，使文件作为程序的输入源或输出源。

重定向输入

用<运算符。格式为：

程序名<文件名

重定向输出

用>运算符。格式为：

程序名>文件名

没有指定文件时会**创建**一个文件。

组合重定向

/程序名<文件名1>文件名2

文件1输入程序1和文件2中

/程序名>文件名1<文件名2

程序输出到文件1，文件2输入文件1中。

- 在一条命令中，输入文件名和输出文件名**不能相同。**
- 重定向运算符必须连接**一个程序和一个数据文件**，不能是两个程序或者是两个文件。
- 重定向运算符不能读取**多个文件的输入**，也不能把输出定向到多个文件。
- windows环境还有>>，可以**将数据添加到文件的末尾。**

- 而|运算符可以将**一个文件的输出连接到另一个文件的输入**

下面是一个实例(以windows程序为例):

1. 有一个在桌面名为program.exe的程序，该程序从标准输入读取两个整数，并将它们的和输出到标准输出。这是该程序的源码。

```
#include <stdio.h>

int main() {
    int num1, num2;

    scanf("%d", &num1);  // 从标准输入读取第一个整数
    scanf("%d", &num2);  // 从标准输入读取第二个整数

    int sum = num1 + num2;

    printf("Sum: %d\n", sum);  // 输出计算结果到标准输出

    return 0;
}
```

2. 创建一个名为`input.txt`的文件，并在其中写入两个整数，如:

```
10
20
```

3. 在Windows系统中，按下Win键 + R，然后输入"cmd"，打开命令提示符（Command Prompt）。然后按Enter键来打开相应的命令行界面。

使用 `cd` 命令（Change Directory）来改变当前目录。在命令行中，输入以下命令:

```
cd /d C:\Users\yuy788\Desktop
```

使用 `/d` 选项可以在需要时更改驱动器。按下 Enter 键执行命令。命令行会将当前目录更改为指定的目录。

4. 运行程序并将输入重定向到`input.txt`文件，将输出重定向到`output.txt`文件，使用以下命令

```
program.exe < input.txt > output.txt
```

5. 检查 `output.txt` 文件的内容，包含程序的输出结果：

```
Sum: 30
```

通过以上步骤，我们成功将程序的输入从文件 `input.txt` 中读取，将输出写入到 `output.txt` 文件中，实现了输入输出重定向。

自定义数据类型

结构

结构是 C一种用户自定义的可用的数据类型，可以存储不同类型的数据项。

定义结构

结构体定义由关键字 **struct** 和**结构体名**组成，结构体名可以根据需要自行定义。

struct 语句定义了一个包含多个成员的新的数据类型，struct 语句的格式如下：

```
struct tag {
    member-list
    member-list
    member-list
    ...
} variable-list ;
```

tag 是结构体类型名。

member-list 是**标准的变量定义**，比如 **int i;** 或者 **float f;**，或者其他有效的变量定义。

variable-list 结构变量。在定义结构体类的时候可以同时定义一个结构体变量在结构的末尾，最后一个分号之前。

下面是声明 Book 结构的方式：

```
struct Books
{
   char  title[50];
```

```
   char  author[50];
   char  subject[100];
   int   book_id;
} book;
```

在一般情况下，**tag、member-list、variable-list** 这 3 部分至少要出现 2 个。以下为实例：

```
//此声明声明了拥有3个成员的结构体，分别为整型的a，字符型的b和双精度的c
//同时又声明了结构体变量s1
//这个结构体并没有标明其标签,这样的结构体不能声明第二个
struct
{
    int a;
    char b;
    double c;
} s1;

//此声明声明了拥有3个成员的结构体，分别为整型的a，字符型的b和双精度的c
//结构体的标签被命名为SIMPLE,没有声明变量
struct SIMPLE
{
    int a;
    char b;
    double c;
};
//用SIMPLE标签的结构体，另外声明了变量t1、t2、t3
struct SIMPLE t1, t2[20], *t3;

//现在可以用Simple2作为类型声明新的结构体变量
Simple2 u1, u2[20], *u3;
```

结构体的成员可以包含其他结构体，也可以包含指向自己结构体类型的指针，而通常这种指针的应用是为了实现一些更高级的数据结构如链表和树等。

如果两个结构体互相包含，则需要对其中一个结构体进行不完整声明，如下所示：

```
struct B;    //对结构体B进行不完整声明

//结构体A中包含指向结构体B的指针
struct A
{
    struct B *partner;
```

```
    //other members;
};

//结构体B中包含指向结构体A的指针，在A声明完后，B也随之进行声明
struct B
{
    struct A *partner;
    //other members;
};
```

结构体变量的初始化

```
#include <stdio.h>

struct Books
{
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
} book = {"C 语言", "RUNOOB", "编程语言", 123456};

int main()
{
    printf("title : %s\nauthor: %s\nsubject: %s\nbook_id: %d\n", book.title, book.a
uthor, book.subject, book.book_id);
}
```

执行输出结果为：

```
title : C 语言
author: RUNOOB
subject: 编程语言
book_id: 123456
```

访问结构成员

访问结构的成员要使用**成员访问运算符（.）**。成员访问运算符是结构变量名称和我们要访问的结构成员之间的一个句号。下面的实例演示了结构的访问：

```c
#include <stdio.h>
#include <string.h>

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

int main( )
{
   struct Books Book1;        /* 声明 Book1，类型为 Books */
   struct Books Book2;        /* 声明 Book2，类型为 Books */

   /* Book1 详述 */
   strcpy( Book1.title, "C Programming");
   strcpy( Book1.author, "Nuha Ali");
   strcpy( Book1.subject, "C Programming Tutorial");
   Book1.book_id = 6495407;

   /* Book2 详述 */
   strcpy( Book2.title, "Telecom Billing");
   strcpy( Book2.author, "Zara Ali");
   strcpy( Book2.subject, "Telecom Billing Tutorial");
   Book2.book_id = 6495700;

   /* 输出 Book1 信息 */
   printf( "Book 1 title : %s\n", Book1.title);
   printf( "Book 1 author : %s\n", Book1.author);
   printf( "Book 1 subject : %s\n", Book1.subject);
   printf( "Book 1 book_id : %d\n", Book1.book_id);

   /* 输出 Book2 信息 */
   printf( "Book 2 title : %s\n", Book2.title);
   printf( "Book 2 author : %s\n", Book2.author);
   printf( "Book 2 subject : %s\n", Book2.subject);
   printf( "Book 2 book_id : %d\n", Book2.book_id);

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Book 1 title : C Programming
Book 1 author : Nuha Ali
Book 1 subject : C Programming Tutorial
Book 1 book_id : 6495407
Book 2 title : Telecom Billing
Book 2 author : Zara Ali
Book 2 subject : Telecom Billing Tutorial
Book 2 book_id : 6495700
```

结构作为函数参数

```c
#include <stdio.h>
#include <string.h>

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

/* 函数声明 */
void printBook( struct Books book );
int main( )
{
   struct Books Book1;        /* 声明 Book1，类型为 Books */
   struct Books Book2;        /* 声明 Book2，类型为 Books */

   /* Book1 详述 */
   strcpy( Book1.title, "C Programming");
   strcpy( Book1.author, "Nuha Ali");
   strcpy( Book1.subject, "C Programming Tutorial");
   Book1.book_id = 6495407;

   /* Book2 详述 */
   strcpy( Book2.title, "Telecom Billing");
   strcpy( Book2.author, "Zara Ali");
   strcpy( Book2.subject, "Telecom Billing Tutorial");
   Book2.book_id = 6495700;

   /* 输出 Book1 信息 */
   printBook( Book1 );
```

```
   /* 输出 Book2 信息 */
   printBook( Book2 );

   return 0;
}
void printBook( struct Books book )
{
   printf( "Book title : %s\n", book.title);
   printf( "Book author : %s\n", book.author);
   printf( "Book subject : %s\n", book.subject);
   printf( "Book book_id : %d\n", book.book_id);
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Book title : C Programming
Book author : Nuha Ali
Book subject : C Programming Tutorial
Book book_id : 6495407
Book title : Telecom Billing
Book author : Zara Ali
Book subject : Telecom Billing Tutorial
Book book_id : 6495700
```

向函数传递一个结构是低效的，**传递其指针**更为高效。

指向结构的指针

定义指向结构的指针如下所示：

```
struct Books *struct_pointer;
```

使用指向该结构的指针访问结构的成员要使用 -> 运算符，如下所示：

```
struct_pointer->title;
```

用结构指针来重写上面的实例：

```c
#include <stdio.h>
#include <string.h>

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

/* 函数声明 */
void printBook( struct Books *book );
int main( )
{
   struct Books Book1;        /* 声明 Book1，类型为 Books */
   struct Books Book2;        /* 声明 Book2，类型为 Books */

   /* Book1 详述 */
   strcpy( Book1.title, "C Programming");
   strcpy( Book1.author, "Nuha Ali");
   strcpy( Book1.subject, "C Programming Tutorial");
   Book1.book_id = 6495407;

   /* Book2 详述 */
   strcpy( Book2.title, "Telecom Billing");
   strcpy( Book2.author, "Zara Ali");
   strcpy( Book2.subject, "Telecom Billing Tutorial");
   Book2.book_id = 6495700;

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

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

   return 0;
}
void printBook( struct Books *book )
{
   printf( "Book title : %s\n", book->title);
   printf( "Book author : %s\n", book->author);
   printf( "Book subject : %s\n", book->subject);
   printf( "Book book_id : %d\n", book->book_id);
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Book title : C Programming
Book author : Nuha Ali
Book subject : C Programming Tutorial
Book book_id : 6495407
Book title : Telecom Billing
Book author : Zara Ali
Book subject : Telecom Billing Tutorial
Book book_id : 6495700
```

共用

共用体在相同的内存位置存储不同的数据类型，它提供了一种**使用相同的内存位置**的有效方式。

一个带有多成员的**共用体**任何时候**只能有一个成员带有值。**

定义共用体

union 语句的**格式如下**：

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

union tag 是可选的，每个 member definition 是标准的变量定义，比如 int i; 或者 float f; 或者其他有效的变量定义。

在共用体定义的末尾，最后一个分号之前，可以指定一个或多个共用体变量，这是可选的。

下面定义一个名为 Data 的共用体类型，有三个成员 i、f 和 str：

```
union Data
{
   int i;
   float f;
```

```
   char  str[20];
} data;
```

现在，**Data** 类型的变量可以存储一个整数、一个浮点数，或者一个字符串。但只能同时存储一个。

共用体占用的内存应足够存储**共用体中最大的成员。**

访问共用体成员

访问共用体的成员要使用**成员访问运算符（.）**下面的实例演示了共用体的用法：

```
#include <stdio.h>
#include <string.h>

union Data
{
   int i;
   float f;
   char  str[20];
};

int main( )
{
   union Data data;

   printf( "Memory size occupied by data : %d\n", sizeof(data));

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
data.i : 1917853763
data.f : 4122360580327794860452759994368.000000
data.str : C Programming
```

在这里，我们可以看到共用体的 **i** 和 **f** 成员的值有损坏，因为最后赋给变量的值占用了内存位置，这也是 **str** 成员能够完好输出的原因。

现在让我们再来看一个相同的实例，这次我们在同一时间只使用一个变量，这也演示了使用共用体的主要目的：

```c
#include <stdio.h>
#include <string.h>

union Data
{
   int i;
   float f;
   char  str[20];
};

int main( )
{
   union Data data;

   data.i = 10;
   printf( "data.i : %d\n", data.i);

   data.f = 220.5;
   printf( "data.f : %f\n", data.f);

   strcpy( data.str, "C Programming");
   printf( "data.str : %s\n", data.str);

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
data.i : 10
data.f : 220.500000
data.str : C Programming
```

在这里，所有的成员都能完好输出，因为同一时间只用到一个成员。

枚举

枚举是一种用于定义一组**具有离散值的常量**的基本数据类型，可以让数据更简洁，更易读。

枚举类型通常用于为程序中的**一组相关的常量**取名字，以便于程序的可读性和维护性。

枚举类型的定义

每个枚举常量可以用一个标识符来表示，也可以为它们指定一个整数值，如果没有指定，那么默认从 **0** 开始递增。

枚举语法定义格式为：

```
enum  枚举名  {枚举元素1,枚举元素2,……};
```

用枚举的方式表示一周：

```
enum DAY
{
      MON=1, TUE, WED, THU, FRI, SAT, SUN
};
```

第一个枚举成员的默认值为整型的 0，后续枚举成员的值在前一个成员上加 1。

在这个实例中把第一个枚举成员的值定义为 1，第二个就为 2，以此类推。

可以在定义枚举类型时改变枚举元素的值：

```
enum season {spring, summer=3, autumn, winter};
```

没有指定值的枚举元素，其值为前一元素加 1。也就说 spring 的值为 0，summer 的值为 3，autumn 的值为 4，winter 的值为 5

枚举变量的定义

前面我们只是声明了枚举类型，接下来我们看看如何定义枚举变量。

我们可以通过以下三种方式来定义枚举变量

- **先定义枚举类型，再定义枚举变量**

```
enum DAY
{
      MON=1, TUE, WED, THU, FRI, SAT, SUN
};
enum DAY day;
```

- **定义枚举类型的同时定义枚举变量**

```
enum DAY
{
      MON=1, TUE, WED, THU, FRI, SAT, SUN
} day;
```

- **省略枚举名称，直接定义枚举变量**

```
enum
{
      MON=1, TUE, WED, THU, FRI, SAT, SUN
} day;//这样的枚举类型只能定义一个
```

```
#include <stdio.h>

enum DAY
{
      MON=1, TUE, WED, THU, FRI, SAT, SUN
};

int main()
{
    enum DAY day;
    day = WED;
    printf("%d",day);
    return 0;
}
```

以上实例输出结果为：

```
3
```

可以用枚举元素或整数对枚举变量进行初始化。但如果使用整数时必须进行**类型转换**。如：

```
Day enDau=（Day）0;
```

枚举类型的遍历

在C ++中，枚举类型是被当做 int 或者 unsigned int 类型来处理的，按照 C ++规范是没有办法遍历枚举类型的。

不过在一些特殊的情况下，枚举类型必须连续是可以实现有条件的遍历。

以下实例使用 for 来遍历枚举的元素：

```
#include <stdio.h>

enum DAY
{
      MON=1, TUE, WED, THU, FRI, SAT, SUN
} day;
int main()
{
    // 遍历枚举元素
    for (day = MON; day <= SUN; day++) {
        printf("枚举元素：%d \n", day);
    }
}
```

以上实例输出结果为：

```
枚举元素：1
枚举元素：2
枚举元素：3
枚举元素：4
枚举元素：5
枚举元素：6
枚举元素：7
```

以下枚举类型不连续，这种枚举无法遍历。

```
enum
{
    ENUM_0,
    ENUM_10 = 10,
```

```
    ENUM_11
};
```

枚举在 switch 中的使用

```
#include <stdio.h>
#include <stdlib.h>
int main()
{

    enum color { red=1, green, blue };

    enum  color favorite_color;

    /* 用户输入数字来选择颜色 */
    printf("请输入你喜欢的颜色: (1. red, 2. green, 3. blue): ");
    scanf("%u", &favorite_color);

    /* 输出结果 */
    switch (favorite_color)
    {
    case red:
        printf("你喜欢的颜色是红色");
        break;
    case green:
        printf("你喜欢的颜色是绿色");
        break;
    case blue:
        printf("你喜欢的颜色是蓝色");
        break;
    default:
        printf("你没有选择你喜欢的颜色");
    }

    return 0;
}
```

以上实例输出结果为：

```
请输入你喜欢的颜色: (1. red, 2. green, 3. blue): 1
你喜欢的颜色是红色
```

枚举与整数

- 枚举值可以被**赋给整型变量。**如int i;i=Mon；但**整型值**不能直接赋给**枚举变量**，要**转换，不同枚举类型**之间的变量不能互相赋值。
- 枚举变量可以进行**算术运算**，但是得到的值是**整型值**，不能赋给枚举变量
- 枚举值可以进行**关系运算**，系统以**枚举元素序号大小**作为比较依据
- 枚举变量**不能直接输入和输出，**直接输出枚举变量只能得到序号

以下实例将整数转换为枚举：

```
#include <stdio.h>
#include <stdlib.h>

int main()
{

    enum day
    {
        saturday,
        sunday,
        monday,
        tuesday,
        wednesday,
        thursday,
        friday
    } workday;

    int a = 1;
    enum day weekend;
    weekend = ( enum day ) a;  //类型转换
    //weekend = a; //错误
    printf("weekend:%d",weekend);
    return 0;
}
```

以上实例输出结果为：

```
weekend:1
```

重命名变量

typedef

typedef 可以来为类型取一个新的名字。下面的实例为单字节数字定义了一个术语 **BYTE**：

```
typedef unsigned char BYTE;
```

在这个类型定义之后，标识符 BYTE 可作为类型 **unsigned char** 的缩写，例如：

```
BYTE  b1, b2;
```

按照惯例，定义时会大写字母，以便提醒用户类型名称是一个象征性的缩写，但也可以使用小写字母

可以使用 **typedef** 来为用户自定义的数据类型取一个新的名字，如下：

```
#include <stdio.h>
#include <string.h>

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

int main( )
{
   Book book;

   strcpy( book.title, "C 教程");
   strcpy( book.author, "Runoob");
   strcpy( book.subject, "编程语言");
   book.book_id = 12345;

   printf( "书标题 : %s\n", book.title);
   printf( "书作者 : %s\n", book.author);
   printf( "书类目 : %s\n", book.subject);
   printf( "书 ID : %d\n", book.book_id);
```

```
   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
书标题 : C 教程
书作者 : Runoob
书类目 : 编程语言
书 ID : 12345
```

using

通过**using编译指令**起到重命名的作用，更为直观，例如

```
typedef const char* pc1
using pc1=const char*//使用using
```

两种效果是**等价**的

位操作

二进制数、位、字节

有符号整数的表示

二进制补码

概念

用1字节的后7位表示0-127，第8位设置为0时表示正数，1为负数。为0时，补码就是真码。

相反数

求补码数所代表的相反数，要对每一位进行反转，末尾加1，就可以得到相反数的原码，从而求得其真值。

二进制反码

正数的反码即其补码。

对于每个数，求它的相反数只需把每一位都反转。反码可以表示-127~+127之间的数。

二进制浮点数的表示

二进制小数

二进制小数点后的每一位都是用2的次幂作分母，如.101就是0.5+0+0.125=0.625.因此，二进制小数不能精确的表示某些十进制小数。

浮点数表示法

一个浮点数的表示是存储二进制小数和指数同时进行的。

按位运算符

按位取反符(~)

对每一位都取反码。单目运算符。

按位与(&)

二元运算符。对两个数每一位逐次比较生成一个新值，如果这两个数的同一位都为1，新值相应的位上记1，否则记0.如(10010011)&(00111101)=(00010001)

- **掩码**

  通过按位与，可以利用一个掩码将另一个数的**相应位置盖住。**例如：

  (00000010)(掩码)&(11001111)=(00000010)

  被遮盖的码，对应的位会变成0，因此可以说掩码中的0是不透明，1是透明

  通过掩码，可以检查某一位的值。

- **清空位**

  与掩码类似。将需要清空的位，清空码改成0.

按位或(|)

类似按位与

- **打开位**

  通过按位或，可以利用一个控制码将另一个数**相应的位置变成1.**

按位异或(^)

对两个数逐次比较，若两个数的**该位不一致，则记为1.**

移位运算符

左移(<<)/右移(>>)

将每一位向左/右移动指定的位数，空出来的部分**用0补齐。**位数最好不要为负。

如(10001010)<<2=(00101000)

算术移位与逻辑移位

两者的区别仅在**右移**时有所体现。

算术移位：原值的**符号位如为1**，则右移时移入的位**均为1**；否则**为0**

逻辑移位：移入的位**均为0**

有符号值的右移操作究竟使用**逻辑移位**还是**算术移位取决于编译器。**因此，使用有符号数的右移操作的程序是**不可移植**的。

移位运算符相当于十进制通过**移动小数点的办法进行移位。**

位域

有些信息在存储时，并不需要占用一个完整的字节，而只需占几个或一个二进制位。例如在存放一个开关量时，只有 0 和 1 两种状态，用 1 位二进位即可。

为了**节省存储空间**，并使**处理简便**，C 语言又提供了一种数据结构，称为"**位域**"或"位段"。

"位域"把一个字节中的二进位划分为几个不同的区域，并说明每个区域的位数。

每个域有一个域名，允许在程序中**按域名**进行操作。这样就可以把**几个不同的对象**用一个字节的**二进制位域**来表示。

位域的定义和位域变量的说明

位域定义与结构定义相仿，其形式为：

```
struct 位域结构名
{

 位域列表

};
```

其中位域列表的形式为：

```
type [member_name] : width ;
```

下面是有关位域中变量元素的描述：

元素	描述
type	只能为 int(整型)，unsigned int(无符号整型)，signed int(有符号整型) 三种类型，决定了如何解释位域的值。
member_name	位域的名称。
width	位域中位的数量。宽度必须**小于或等于指定类型的位宽度。**

带有预定义宽度的变量被称为**位域。**位域可以存储多于 1 位的数，

例如，需要一个变量来存储从 0 到 7 的值，可以定义一个宽度为 3 位的位域，如下：

```
struct
{
  unsigned int age : 3;
} Age;
```

上面的结构定义指示 C 编译器，age 变量将只使用 3 位来存储这个值，使用超过 3 位则无法编译。

```
struct bs{
    int a:8;
    int b:2;
    int c:6;
}data;
```

data 为 bs 变量，共占两个字节。其中位域 a 占 8 位，位域 b 占 2 位，位域 c 占 6 位。

让我们再来看一个实例：

```
struct packed_struct {
  unsigned int f1:1;
  unsigned int f2:1;
  unsigned int f3:1;
  unsigned int f4:1;
  unsigned int type:4;
```

```
  unsigned int my_int:9;
} pack;
```

在这里，packed_struct 包含了 6 个成员：四个 1 位的标识符 f1..f4、一个 4 位的 type 和一个 9 位的 my_int。

让我们来看下面的实例：

```
#include <stdio.h>
#include <string.h>

struct
{
  unsigned int age : 3;
} Age;

int main( )
{
   Age.age = 4;
   printf( "Sizeof( Age ) : %d\n", sizeof(Age) );
   printf( "Age.age : %d\n", Age.age );

   Age.age = 7;
   printf( "Age.age : %d\n", Age.age );

   Age.age = 8; // 二进制表示为 1000 有四位，超出
   printf( "Age.age : %d\n", Age.age );

   return 0;
}
```

当上面的代码被编译时，它会带有警告，当上面的代码被执行时，它会产生下列结果：

```
Sizeof( Age ) : 4
Age.age : 4
Age.age : 7
Age.age : 0
```

对于位域的定义尚有以下几点说明：

- 一个位域存储在同一个字节中，如一个字节所剩空间不够存放另一位域时，则会从下一单元起存放该位域。也可以有意使某位域从下一单元开始。例如：

```
struct bs{
    unsigned a:4;
    unsigned  :4;    /* 空域 */
    unsigned b:4;    /* 从下一单元开始存放 */
    unsigned c:4
}
```

在这个位域定义中，a 占第一字节的 4 位，后 4 位填 0 表示不使用，b 从第二字节开始，占用 4 位，c 占用 4 位。

- 位域的宽度**不能超过它所依附的数据类型的长度**，

  成员变量都是有类型的，这个类型限制了成员变量的最大长度，：后面的数字不能超过这个长度。

- 位域可以是**无名位域**，这时它只用来作填充或调整位置。**无名的位域是不能使用的。**例如：

```
struct k{
    int a:1;
    int  :2;    /* 该 2 位不能使用 */
    int b:3;
    int c:2;
};
```

从以上分析可以看出，位域在本质上就是一种**结构类型**，不过其成员是按二进位分配的。

位域的使用

位域的使用和结构成员的使用相同，其一般形式为：

```
位域变量名.位域名
位域变量名->位域名
```

位域允许用各种格式输出。

请看下面的实例：

```
#include <stdio.h>

int main(){
    struct bs{
        unsigned a:1;
        unsigned b:3;
```

```
        unsigned c:4;
    } bit,*pbit;
    bit.a=1;    /* 给位域赋值（应注意赋值不能超过该位域的允许范围） */
    bit.b=7;    /* 给位域赋值（应注意赋值不能超过该位域的允许范围） */
    bit.c=15;    /* 给位域赋值（应注意赋值不能超过该位域的允许范围） */
    printf("%d,%d,%d\n",bit.a,bit.b,bit.c);    /* 以整型量格式输出三个域的内容 */
    pbit=&bit;    /* 把位域变量 bit 的地址送给指针变量 pbit */
    pbit->a=0;    /* 用指针方式给位域 a 重新赋值，赋为 0 */
    pbit->b&=3;    /* 使用了复合的位运算符 "&="，相当于：pbit->b=pbit->b&3，位域 b 中原有值为 7，与 3 作按位与运算的结果为 3（111&011=011，十进制值为 3） */
    pbit->c|=1;    /* 使用了复合位运算符"|="，相当于：pbit->c=pbit->c|1，其结果为 15 */
    printf("%d,%d,%d\n",pbit->a,pbit->b,pbit->c);    /* 用指针方式输出了这三个域的值 */
}
```

上例程序中定义了位域结构 bs，三个位域为 a、b、c。说明了 bs 类型的变量 bit 和指向 bs 类型的指针变量 pbit。

这表示位域也可以使用指针。

预处理器

C 预处理器不是编译器的组成部分，但是它是编译过程中一个单独的步骤。

简言之，C 预处理器是一个文本替换工具，它们会指示编译器在实际编译之前完成所需的预处理。

所有的预处理器命令都是以井号（#）开头。它必须是第一个非空字符，为了增强可读性，预处理器指令应从第一行开始。

下面列出了所有重要的预处理器指令：

指令	描述
#define	定义宏
#include	包含一个源代码文件
#undef	取消已定义的宏
#ifdef	如果宏已经定义，则返回真
#ifndef	如果宏没有定义，则返回真
#if	如果给定条件为真，则编译下面代码
#else	#if 的替代方案
#elif	如果前面的 #if 给定条件不为真，当前条件为真，则编译下面代码
#endif	结束一个 if……else 条件编译块
#error	当遇到标准错误时，输出错误消息
#pragma	使用标准化方法，向编译器发布特殊的命令到编译器中

预处理器实例

#define

```
#define MAX_ARRAY_LENGTH 20
```

这个指令把所有的 MAX_ARRAY_LENGTH 定义为 20。

使用 *#define* 定义常量来增强可读性。

标识符与字符序列应**相隔一个以上的空格**或一个**制表符**。

宏允许**被嵌套使用**。一个宏名可以出现在**另一个宏的替换文本中**。

宏名习惯一般用**大写**。

宏定义中可以没有替换文本。这样的宏定义通常作为条件编译检测的一个标志。

#include

```
#include <stdio.h>
#include "myheader.h"
```

第一行从**系统库**中获取 stdio.h，并添加文本到当前的源文件中。

下一行 从**本地目录**中获取 **myheader.h**，并添加内容到当前的源文件中。

#undef

```
#undef  FILE_SIZE
#define FILE_SIZE 42
```

取消已定义的 FILE_SIZE，并定义它为 42。

#ifndef

```
#ifndef MESSAGE
   #define MESSAGE "You wish!"
#endif
```

只有当 MESSAGE 未定义时，才定义 MESSAGE。

同一个文件中只能将同一个头文件包含一次，为了避免出错，可以使用下面的预处理器编译指令，如下

```
#ifndef COORDIN_H
#define COORDIN_H
#include<head.h>
#endif
```

如果该头文件已经出现过，那么COORDIN_H就是已经定义的，在同个程序的另个文件中就不会再包含这个头文件

#ifdef

```
#ifdef DEBUG
   /* Your debugging statements here */
#endif
```

如果**定义了 DEBUG**，则执行处理语句。

#elif

```
#if SYSTEM_1//#if后的宏定义为0时不执行程序，#ifdef后的宏若没被定义不执行程序
   # include "system_1.h"
#elif SYSTEM_2
   # include "system_2.h"
#elif SYSTEM_3
   ...
#endif
```

#error

error Not C11可以让屏幕上出现error:error Not C11.

预定义宏

宏	描述
DATE	当前日期，一个以 "MMM DD YYYY" 格式表示的字符常量。
TIME	当前时间，一个以 "HH:MM:SS" 格式表示的字符常量。
FILE	这会包含当前文件名，一个字符串常量。
LINE	这会包含当前行号，一个十进制常量。
STDC	当编译器以 ANSI 标准编译时，则定义为 1。

让我们来尝试下面的实例：

```
#include <stdio.h>

main()
{
   printf("File :%s\n", __FILE__ );
   printf("Date :%s\n", __DATE__ );
   printf("Time :%s\n", __TIME__ );
   printf("Line :%d\n", __LINE__ );
   printf("ANSI :%d\n", __STDC__ );

}
```

当上面的代码（在文件 **test.c** 中）被编译和执行时，它会产生下列结果：

```
File :test.c
Date :Jun 2 2012
```

```
Time :03:36:24
Line :8
ANSI :1
```

预处理器运算符

宏延续运算符（\）

一个宏通常写在一个单行上。但是如果宏太长，一个单行容纳不下，则使用宏延续运算符（\）。例如：

```
#define  message_for(a, b)  \
    printf(#a " and " #b ": We love you!\n")
```

字符串常量化运算符（#）

在宏定义中，当需要把一个**宏的参数**转换为**字符串常量**时，则使用字符串常量化运算符（#）。

在宏中使用的该运算符有一个特定的参数或参数列表。例如：

```
#include <stdio.h>

#define  message_for(a, b)  \
    printf(#a " and " #b ": We love you!\n")

int main(void)
{
   message_for(Carole, Debra);
   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Carole and Debra: We love you!
```

标记粘贴运算符（##）

宏定义内的标记粘贴运算符（##）会合并两个参数。它允许在宏定义中两个独立的标记被合并为一个标记。例如：

```
#include <stdio.h>

#define tokenpaster(n) printf ("token" #n " = %d", token##n)

int main(void)
{
   int token34 = 40;

   tokenpaster(34);
   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
token34 = 40
```

这是怎么发生的，因为这个实例会从编译器产生下列的实际输出：

```
printf ("token34 = %d", token34);
```

defined() 运算符

预处理器 **defined** 运算符是用在常量表达式中的，用来确定一个标识符是否已经使用 #define 定义过。

如果指定的标识符**已定义**，则值为**真**（非零）。

如果指定的标识符**未定义**，则值为**假**（零）。下面的实例演示了 defined() 运算符的用法：

```
#include <stdio.h>

#if !defined (MESSAGE)
   #define MESSAGE "You wish!"
#endif

int main(void)
```

```
{
   printf("Here is the message: %s\n", MESSAGE);
   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Here is the message: You wish!
```

参数化的宏

可以使用**参数化的宏**来模拟函数。例如，下面的代码是计算一个数的平方：

```
int square(int x) {
   return x * x;
}
```

我们可以使用宏重写上面的代码，如下：

```
#define square(x) ((x) * (x))
```

在使用带有参数的宏之前，必须使用 **#define** 指令定义。

参数列表是括在圆括号内，且必须紧跟在宏名称的后边。宏名称和左圆括号之间不允许有空格。例如：

```
#include <stdio.h>

#define MAX(x,y) ((x) > (y) ? (x) : (y))

int main(void)
{
   printf("Max between 20 and 10 is %d\n", MAX(10, 20));
   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Max between 20 and 10 is 20
```

命令行参数

执行程序时，可以从**命令行**传值给 C 程序。这些值被称为**命令行参数**，

当要从外部控制程序，而不是在代码内对这些值进行硬编码时，命令行参数显得尤为重要了。

命令行参数构成

命令行参数是使用 **main() 函数参数**来处理的，

其中，**argc** 是指**传入参数的个数**，**argv[]** 是一个**指针数组**，指向传递给程序的每个参数。

下面是一个简单的实例，检查命令行是否有提供参数，并根据参数执行相应的动作：

```
#include <stdio.h>

int main( int argc, char *argv[] )
{
   if( argc == 2 )
   {
      printf("The argument supplied is %s\n", argv[1]);
   }
   else if( argc > 2 )
   {
      printf("Too many arguments supplied.\n");
   }
   else
   {
      printf("One argument expected.\n");
   }
}
```

使用一个参数，编译并执行上面的代码，它会产生下列结果：

```
$./a.out testing
The argument supplied is testing
```

使用两个参数，编译并执行上面的代码，它会产生下列结果：

```
$./a.out testing1 testing2
Too many arguments supplied.
```

不传任何参数，编译并执行上面的代码，它会产生下列结果：

```
$./a.out
One argument expected
```

argv[0] 存储程序的名称，**argv[1]** 是一个指向第一个命令行参数的指针，*argv[n] 是最后一个参数。

如果没有提供任何参数，argc 将为 1，否则，如果传递了一个参数，**argc** 将被设置为 2。

多个命令行参数之间用**空格分隔**，但是如果参数本身带有空格，那么传递参数的时候应把参数放置在双引号 "" 或单引号 '' 内部。

重新编写上面的实例，向程序传递一个放置在双引号内部的命令行参数：

```
#include <stdio.h>

int main( int argc, char *argv[] )
{
   printf("Program name %s\n", argv[0]);

   if( argc == 2 )
   {
      printf("The argument supplied is %s\n", argv[1]);
   }
   else if( argc > 2 )
   {
      printf("Too many arguments supplied.\n");
   }
   else
   {
      printf("One argument expected.\n");
   }
}
```

使用一个用空格分隔的简单参数，参数括在双引号中，编译并执行上面的代码，它会产生下列结果：

```
$./a.out "testing1 testing2"

Programm name ./a.out
The argument supplied is testing1 testing2
```

可变参数列

可变参数列允许定义一个能**根据具体的需求**接受**可变数量的参数。**

C中可变参数列

声明方式为：

```
int func_name(int arg1, ...);
```

其中，省略号 **...** 表示**可变参数列表。**

下面的实例演示了这种函数的使用：

```
int func(int, ... )  {
   .
   .
   .
}

int main() {
   func(2, 2, 3);
   func(3, 2, 3, 4);
}
```

函数 **func()** 最后一个参数写成省略号，即三个点号（**...**），

省略号之前的那个参数是 **int**，代表了要传递的**可变参数的总数。**

使用这个功能需要使用 **stdarg.h** 头文件，该文件提供了实现可变参数功能的函数和宏。具体步骤如下：

- 定义一个函数，最后一个参数为省略号，省略号前面可以设置自定义参数。
- 在函数定义中创建一个 **va_list** 类型变量，该类型是在 stdarg.h 头文件中定义的。

- 使用 **int** 参数和 **va_start()** 宏来初始化 **va_list** 变量为一个参数列表。宏 **va_start()** 是在 stdarg.h 头文件中定义的。
- 使用 **va_arg()** 宏和 **va_list** 变量来访问参数列表中的每个项。
- 使用宏 **va_end()** 来清理赋予 **va_list** 变量的内存。

常用的宏有：

- `<strong>va_start(ap, last_arg)</strong>`：初始化可变参数列表。

  `ap` 是一个 `va_list` 类型的变量，`last_arg` 是最后一个固定参数的名称（也就是**可变参数列表之前的参数**）。

  该宏将 `ap` 指向可变参数列表中的第一个参数。
- `<strong>va_arg(ap, type)</strong>`：获取可变参数列表中的下一个参数。`ap` 是一个 `va_list` 类型的变量，`type` 是**下一个参数**的类型。

  该宏返回类型为 `type` 的值，并将 `ap` 指向下一个参数。
- `<strong>va_end(ap)</strong>`：结束可变参数列表的访问。`ap` 是一个 `va_list` 类型的变量。该宏将 `ap` 置为 `NULL`。

现在让我们按照上面的步骤，来编写一个带有可变数量参数的函数，并返回它们的平均值：

```
#include <stdio.h>
#include <stdarg.h>

double average(int num,...)
{
    va_list valist;
    double sum = 0.0;
    int i;

    /* 为 num 个参数初始化 valist */
    va_start(valist, num);

    /* 访问所有赋给 valist 的参数 */
    for (i = 0; i < num; i++)
    {
       sum += va_arg(valist, int);
    }
    /* 清理为 valist 保留的内存 */
    va_end(valist);

    return sum/num;
}
```

```
int main()
{
   printf("Average of 2, 3, 4, 5 = %f\n", average(4, 2,3,4,5));
   printf("Average of 5, 10, 15 = %f\n", average(3, 5,10,15));
}
```

在上面的例子中，average() 函数接受一个整数 num 和任意数量的整数参数。函数内部使用 va_list 类型的变量 va_list 来访问可变参数列表。在循环中，每次使用 va_arg() 宏获取下一个整数参数，并输出。最后，在函数结束时使用 va_end() 宏结束可变参数列表的访问。

当上面的代码被编译和执行时，它会产生下列结果。

应该指出的是，函数 **average()** 被调用两次，每次第一个参数都是表示**被传的可变参数的总数。**

省略号被用来**传递可变数量的参数。**

```
Average of 2, 3, 4, 5 = 3.500000
Average of 5, 10, 15 = 10.000000
```

C++中可变参数模板

模板参数包和函数参数包

模板参数包Args是包含参数列中所有**参数类型**的一个模板，而**函数参数包args**包含参数列中**所有的参数**，声明为

```
template<typename...Args>//Args是一个参数包，如果输入1,2.0,包中就会有int,double两个类型
returntype functioname(Args...args)//args是函数参数包，有1和2.0
```

展开参数包

```
template<typename T,typename...Args>
void show_list3(T value,Args...args)
{
  show_list3(Args...args);
}
```

在传入一个参数包后，参数包的第一项匹配到T，其他项匹配到下一个参数包，不断重复，当最终参数包只剩下一个参数时，递归就会停止。

通过**递归的方式**，可以**将参数包展开**

动态内存管理

C中的内存管理函数

C中内存管理函数只有在能够程序执行时快速重新分配内存空间的**realloc函数**这一点上优于C++的运算符new、delete。

在面向对象编程时，还是new、delete较为合适

C 语言提供了一些函数和运算符，使得程序员可以对内存进行操作，包括分配、释放、移动和复制等。

序号	函数和描述
1	**void *calloc(int num, int size);** 在内存中动态地分配 num 个长度为 size 的连续空间，并将每一个字节都初始化为 0。所以它的结果是分配了 num*size 个字节长度的内存空间，并且每个字节的值都是 0。
2	**void free(void *address);** 该函数释放 address 所指向的内存块,释放的是动态分配的内存空间。
3	**void *malloc(int num);** 在堆区分配一块指定大小的内存空间，用来存放数据。这块内存空间在函数执行完成后不会被初始化，它们的值是未知的。
4	**void *realloc(void *address, int newsize);** 该函数重新分配内存，把内存扩展到 **newsize**。

注意：void * 类型表示未确定类型的指针。C、C++ 规定 void * 类型可以通过类型转换强制转换为任何其它类型的指针。

动态分配内存

使用malloc函数和calloc函数分配动态内存

重新调整内存的大小和释放内存

当程序退出时，操作系统会自动释放所有分配给程序的内存，但是，建议在不需要内存时，都应该调用函数 **free()** 来释放内存。

或者通过调用函数 **realloc()** 来增加或减少已分配的内存块的大小。让我们使用 realloc() 和 free() 函数，再次查看上面的实例：

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main()
{
```

```
   char name[100];
   char *description;

   strcpy(name, "Zara Ali");

   /* 动态分配内存 */
   description = (char *)malloc( 30 * sizeof(char) );
   if( description == NULL )
   {
      fprintf(stderr, "Error - unable to allocate required memory\n");
   }
   else
   {
      strcpy( description, "Zara ali a DPS student.");
   }
   /* 假设您想要存储更大的描述信息 */
   description = (char *) realloc( description, 100 * sizeof(char) );
   if( description == NULL )
   {
      fprintf(stderr, "Error - unable to allocate required memory\n");
   }
   else
   {
      strcat( description, "She is in class 10th");
   }

   printf("Name = %s\n", name );
   printf("Description: %s\n", description );

   /* 使用 free() 函数释放内存 */
   free(description);
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Name = Zara Ali
Description: Zara ali a DPS student.She is in class 10th
```

不重新分配额外的内存，strcat() 函数会生成一个错误，因为存储 description 时可用的内存不足。

C 语言中常用的内存管理函数和运算符

- **memcpy()** 函数：用于从源内存区域复制数据到目标内存区域。

它接受三个参数，即**目标内存区域的指针**、**源内存区域的指针**和要**复制的数据大小**（以字节为单位）。例如：

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

int main() {
    char src[] = "Hello, World!";
    char dest[20];

    std::memcpy(dest, src, sizeof(src));

    std::cout << "Copied string: " << dest << std::endl;

    return 0;
}
```

编译后运行，输出结果如下：

```
Copied string: Hello, World!
```

该示例代码演示了如何使用memcpy函数将源字符串"Hello, World!"复制到目标字符数组dest中。

通过使用**std::memcpy函**数，源字符串的内容被拷贝到目标字符数组中，并输出结果。

- **memmove()** 函数：类似于 memcpy() 函数，但它可以处理重叠的内存区域。

它接受三个参数，即**目标内存区域的指针**、**源内存区域的指针**和要**复制的数据大小**（以字节为单位）。例如：

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

int main() {
    char str[] = "Hello, World!";
    char buffer[20];

    std::memmove(buffer, str, sizeof(str));
```

```
    std::cout << "Moved string: " << buffer << std::endl;

    return 0;
}
```

编译后运行，输出结果如下：

```
Moved string: Hello, World!
```

该示例代码演示了如何使用**memmove函数**将源字符串"Hello, World!"**移动到目标字符数组buffer**中。

与**memcpy函数不同**的是，**memmove函数**可以正确处理**源字符串**与**目标字符数组的重叠**情况，确保数据的正确移动，并输出结果。

- **memset()函数**：为一段连续内存空间分配同一个初始值

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

int main() {
    char str[20];

    std::memset(str, 'A', sizeof(str));

    std::cout << "Filled string: " << str << std::endl;

    return 0;
}
```

编译并运行后，输出结果如下

```
Filled string: AAAAAAAAAAAAAAAAAAAA
```

C++中的内存管理函数

C++ 程序中的内存分为两个部分：

- **栈：**在**函数内部声明**的所有变量都将占用**栈内存。**
- **堆：**这是程序中**未使用的内存**，在程序运行时可用于**动态分配内存。**

通过下面的内存管理运算符，可以动态分配内存

new 和 delete 运算符

下面是使用 new 运算符来为任意的数据类型动态分配内存的通用语法：

```
new data-type;
```

在这里，**data-type** 可以是包括数组在内的任意内置的数据类型，也可以是包括类或结构在内的用户自定义的任何数据类型。

new 与 malloc() 函数相比，其主要的优点是，new 不只是分配了内存，它还创建了对象。

new运算符在申请内存空间的时候还可以同时初始化，如下

```
int *p=new int(6)//*pi所指的空间存储6
int *ar=new int[4]{2,4,6,7}//指向一个数组
```

在任何时候，当某个已经动态分配内存的变量不再需要使用时，可以使用 delete 操作符释放它所占用的内存，如下所示：

```
delete pvalue;        // 释放 pvalue 所指向的内存
```

下面的实例中使用了上面的概念，演示了如何使用 new 和 delete 运算符：

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

int main ()
{
   double* pvalue  = NULL; // 初始化为 null 的指针
   pvalue  = new double;   // 为变量请求内存

   *pvalue = 29494.99;     // 在分配的地址存储值
   cout << "Value of pvalue : " << *pvalue << endl;
```

```
   delete pvalue;         // 释放内存

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Value of pvalue : 29495
```

但用delete或free撤消动态变量后，C++编译程序**不会把它的指向赋为NULL**，这时就会出现一个**指向无效空间的“悬浮指针“**

定位new运算符

定位new是new的另一种重载，使用前要包含头文件<new>.定位new可以将指针**指向已知的一段空间**

示范代码如下：

```
#include<new>
struct chaff
{
        char dross[20];
        int slag;
};
char buffer1[50];
char buffer2[500];
int main()
{
        chaff *p1,*p2;
        int *p3,*p4;
        p1=new chaff;
        p3=new int[20]//这两个new运算符都申请了堆上的一段内存
        p2=new(buffer1)chaff;
        p4=new(buffer2)int[20];//这两个定位new运算符将p2，p4指向已知的buffer1和buffer2
}
```

注意：

- 程序中两个定位运算符申请的空间不能相同，可以利用如下代码再不提前开辟内存空间的情况下使用定位new

```
char buffer[512]
int main()
{
        string* pc1,pc2;
        pc1=new(buffer) string;
        pc2=new(buffer+sizeof(*pc1)) string//将pc2放到相邻的位置
}
```

- 定位new创建的动态对象不能调用delete运算符，要先依次对这些动态对象显式调用析构，再对它们所申请使用的内存空间进行delete

数组的动态内存分配

假设我们要为一个字符数组（一个有 20 个字符的字符串）分配内存，我们可以使用上面实例中的语法来为数组动态地分配内存，如下所示：

```
char* pvalue  = NULL;   // 初始化为 null 的指针
pvalue  = new char[20]; // 为变量请求内存
```

要删除我们刚才创建的数组，语句如下：

```
delete [] pvalue;        // 删除 pvalue 所指向的数组
```

下面是 new 操作符的通用语法，可以为多维数组分配内存，如下所示：

一维数组

```
// 动态分配,数组长度为 m
int *array=new int [m];

//释放内存
delete [] array;
```

二维数组

```
int **array;
// 假定数组第一维长度为 m， 第二维长度为 n
```

```
// 动态分配空间
array = new int *[m];
for( int i=0; i<m; i++ )
{
    array[i] = new int [n];
}
//释放
for( int i=0; i<m; i++ )
{
    delete [] array[i];
}
delete [] array;
```

二维数组实例测试：

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

int main()
{
    int **p;
    int i,j;   //p[4][8]
    //开始分配4行8列的二维数据
    p = new int *[4];
    for(i=0;i<4;i++){
        p[i]=new int [8];
    }

    for(i=0; i<4; i++){
        for(j=0; j<8; j++){
            p[i][j] = j*i;
        }
    }
    //打印数据
    for(i=0; i<4; i++){
        for(j=0; j<8; j++)
        {
            if(j==0) cout<<endl;
            cout<<p[i][j]<<"\t";
        }
    }
    //开始释放申请的堆
    for(i=0; i<4; i++){
        delete [] p[i];
    }
    delete [] p;
```

```
    return 0;
}
```

对象的动态内存分配

对象与简单的数据类型没有什么不同。例如，请看下面的代码，我们将使用一个对象数组来理清这一概念：

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

class Box
{
   public:
      Box() {
         cout << "调用构造函数！" <<endl;
      }
      ~Box() {
         cout << "调用析构函数！" <<endl;
      }
};

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

   delete [] myBoxArray; // 删除数组
   return 0;
}
```

如果要为一个包含四个 Box 对象的数组分配内存，构造函数将被调用 4 次，同样地，当删除这些对象时，析构函数也将被调用相同的次数（4次）。

当上面的代码被编译和执行时，它会产生下列结果：

```
调用构造函数！
调用构造函数！
调用构造函数！
调用构造函数！
调用析构函数！
调用析构函数！
```

```
调用析构函数！
调用析构函数！
```

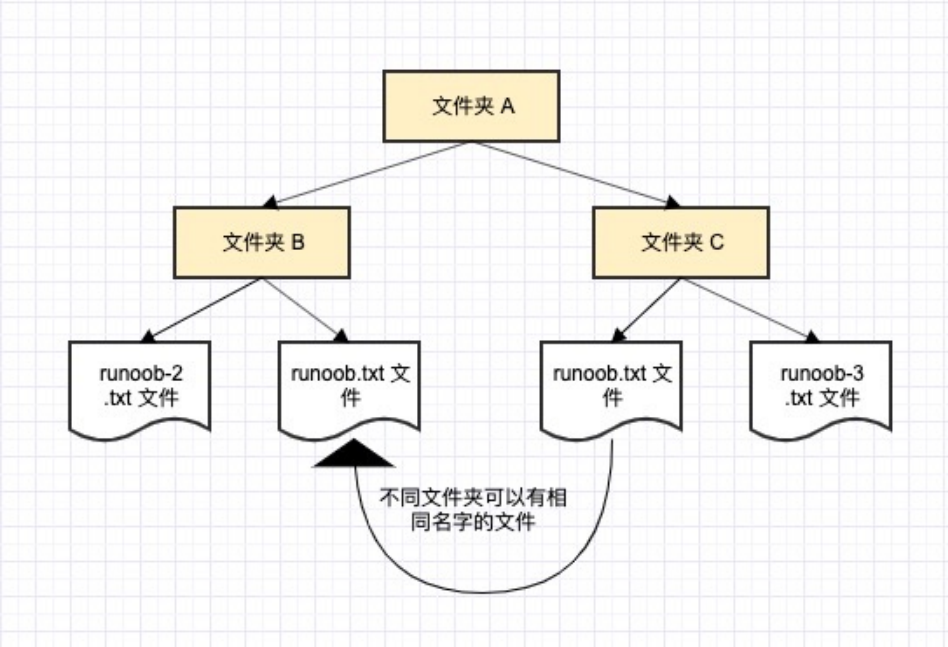
在面向对象编程时，最好使用new和delete，C中的内存管理函数创建对象时不能调用构造函数

命名空间

在C++ 应用程序中，当我们写一个名为 xyz() 的函数，在另一个可用的库中也存在一个相同的函数 xyz()时，编译器无法判断使用的是哪一个 xyz() 函数。

因此，引入了**命名空间**这个概念，专门用于解决上面的问题，它可作为附加信息来区分不同库中相同名称的函数、类、变量等。本质上，命名空间就是定义了一个范围。

举一个计算机系统中的例子，**一个文件夹(目录)**中可以包含多个文件夹，**每个文件夹**中**不能有相同的文件名**，但**不同文件夹中的文件可以重名。**



定义命名空间

命名空间的定义使用关键字 **namespace**，后跟命名空间的名称，如下所示：

```
namespace namespace_name {
   // 代码声明
}
```

这个声明同样也可以用于将**一个变量或函数**加入这个命名空间。

要注意的是，命名空间不能在**代码块中**定义，只能在**代码块外**定义，它的链接属性默认为外部的

为了调用带有命名空间的函数或变量，需要在前面加上命名空间的名称并使用**域解析符**，如下所示：

```
name::code;  // code 可以是变量或函数
```

让我们来看看命名空间如何为变量或函数等实体定义范围：

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

// 第一个命名空间
namespace first_space{
   void func(){
      cout << "Inside first_space" << endl;
   }
}
// 第二个命名空间
namespace second_space{
   void func(){
      cout << "Inside second_space" << endl;
   }
}
int main ()
{

   // 调用第一个命名空间中的函数
   first_space::func();

   // 调用第二个命名空间中的函数
   second_space::func();

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Inside first_space
Inside second_space
```

using 指令

使用 **using namespace** 指令时，在使用命名空间时就可以不用在前面加上命名空间的名称。

这个指令会告诉编译器，后续的代码将**使用指定的命名空间中**的名称。

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

// 第一个命名空间
namespace first_space{
   void func(){
      cout << "Inside first_space" << endl;
   }
}
// 第二个命名空间
namespace second_space{
   void func(){
      cout << "Inside second_space" << endl;
   }
}
using namespace first_space;
int main ()
{

   // 调用第一个命名空间中的函数
   func();

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Inside first_space
```

using 指令也可以用来指定命名空间中的**特定项目**。例如，如果**只打算使用 std 命名空间中的 cout 部分**，可以使用如下的语句：

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

随后的代码中，在使用 cout 时就可以不用加上命名空间名称作为前缀，但是 **std** 命名空间中的**其他项目仍然需要加上命名空间名称**作为前缀，如下所示：

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

int main ()
{

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

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

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

using 指令引入的名称遵循正常的范围规则。名称从使用 **using** 指令开始是可见的，直到**该范围结束**。也就是说，using在代码块中时，其范围是局部的

using指令不能在程序内同时指向两个命名空间

using指令要避免出现在头文件中，并在**#include之后**

不连续的命名空间

命名空间可以定义在几个不同的部分中，因此命名空间是由**几个单独定义的部分**组成的。一个命名空间的**各个组成部分**可以分散在多个文件中。

如果命名空间中的某个组成部分需要**请求定义在另一个文件中的名称**，则仍然需要**声明该名称。**

下面的命名空间定义可以是**定义一个新的命名空间**，也可以是为**已有的命名空间增加新的元素**：

```
namespace namespace_name {
   // 代码声明
}
```

嵌套的命名空间

命名空间**可以嵌套**，在一个命名空间中定义另一个命名空间，如下所示：

```
namespace namespace_name1 {
   // 代码声明
   namespace namespace_name2 {
      // 代码声明
   }
}
```

可以通过使用 **::** 运算符来访问嵌套的命名空间中的成员：

```
// 访问 namespace_name2 中的成员
using namespace namespace_name1::namespace_name2;

// 访问 namespace_name1 中的成员
using namespace namespace_name1;
```

在上面的语句中，如果使用的是 namespace_name1，那么在该范围内 namespace_name2 中的元素**也是可用的**，如下所示：

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

// 第一个命名空间
namespace first_space{
   void func(){
      cout << "Inside first_space" << endl;
   }
   // 第二个命名空间
   namespace second_space{
      void func(){
         cout << "Inside second_space" << endl;
      }
   }
}
using namespace first_space::second_space;
int main ()
{

   // 调用第二个命名空间中的函数
```

```
    func();

    return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Inside second_space
```

匿名命名空间

只能在**当前的源文件**中使用。可代替**static全局变量**。

```
namespace
{int x,y;
}
```

异常

C的异常机制

在发生错误时，大多数的 C 函数调用返回 1 或 NULL，同时会设置一个错误代码 **errno**，

该错误代码是全局变量，表示在函数调用期间发生了错误。在 errno.h 头文件中找到各种各样的**错误代码**。

在程序初始化时，可以把 errno 设置为 0，0值表示程序中没有错误。

errno、perror() 和 strerror()

C 语言提供了 **perror()** 和 **strerror()** 函数来显示与 **errno** 相关的文本消息。

- **perror()** 函数显示传给它的字符串，后跟一个冒号、一个空格和当前 errno 值的文本表示形式。
- **strerror()** 函数，返回一个指针，指针指向当前 errno 值的文本表示形式。

现在尝试打开一个不存在的文件。这里使用函数来演示用法。

另外有一点需要注意，应该使用 **stderr** 文件流来输出所有的错误。

```c
#include <stdio.h>
#include <errno.h>
#include <string.h>

extern int errno ;

int main ()
{
   FILE * pf;
   int errnum;
   pf = fopen ("unexist.txt", "rb");
   if (pf == NULL)
   {
      errnum = errno;
      fprintf(stderr, "错误号: %d\n", errno);
      perror("通过 perror 输出错误");
      fprintf(stderr, "打开文件错误: %s\n", strerror( errnum ));
   }
   else
   {
      fclose (pf);
   }
   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
错误号: 2
通过 perror 输出错误: No such file or directory
打开文件错误: No such file or directory
```

被零除的错误

在进行除法运算时，如果不检查除数是否为零，则**会导致一个运行时错误。**

为了避免这种情况发生，下面的代码在进行除法运算前会先检查除数是否为零：

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
```

```
   int dividend = 20;
   int divisor = 0;
   int quotient;

   if( divisor == 0){
      fprintf(stderr, "除数为 0 退出运行...\n");
      exit(-1);
   }
   quotient = dividend / divisor;
   fprintf(stderr, "quotient 变量的值为 : %d\n", quotient );

   exit(0);
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
除数为 0 退出运行...
```

程序退出状态

通常情况下，程序成功执行完一个操作正常退出的时候会带有值 **EXIT_SUCCESS。**

在这里，**EXIT_SUCCESS** 是**宏**，它被定义为 0。

如果程序中存在一种错误情况，当您退出程序时，会带有状态值 **EXIT_FAILURE**，被定义为 **-1。**所以，上面的程序可以写成：

```
#include <stdio.h>
#include <stdlib.h>

main()
{
   int dividend = 20;
   int divisor = 5;
   int quotient;

   if( divisor == 0){
      fprintf(stderr, "除数为 0 退出运行...\n");
      exit(EXIT_FAILURE);
   }
   quotient = dividend / divisor;
   fprintf(stderr, "quotient 变量的值为: %d\n", quotient );
```

```
   exit(EXIT_SUCCESS);
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
quotient 变量的值为 : 4
```

C++的异常机制

报告运行错误abort()

向**标准错误流**发送消息**abnormal program termination**(程序异常终止)而后**终止程序**，并返回一个随实现而异的值

abort()所实现的程序终止将**直接终止程序**，**不返回到main()**

```
...
double a,b=0;
double divide(double a,double b)
{
        if(b==0)
        {
                std::abort();
        }
        else
        return a/b
}
double c=divide(a,b);
...
```

除数为0，输出错误

```
abnormal program termination
```

异常处理关键词try、catch、throw

这三个关键词共同管理异常处理机制。

- **try**后跟花括号括起一段**有风险的代码**，意在说明这里可能会**出现风险。**

- 当try**中的代码块出现了异常**时，**throw**关键词将一个变量传到**catch所在的代码块**
- **catch**所括起的代码块对**throw所给的参数处理**，并**输出异常信息。**

也就是说，在try出现异常时，先跳到throw，再跳到catch。如下代码

```
//输入两数求其调和平均数，如果两个数是相反数，则将会引发除0异常
double hmean(double a,double b)
{
  if(a==-b)
        throw "bad arguments:a=-b not allowed";
  return 2.0*a*b/(a+b);
}

int main()
{
        double x,y,z;
        std::cin>>x>>y//输入两数
        {
          try{
                z=hmean(x,y);
                }
          catch(const char *s)
        {
          std::cout<<s<<std::endl
        }
        std::cout<<"answer is"<<z<<endl;
}
```

程序步骤是这样的：

1. 输入两个相反数，主函数在执行hmean时，发现a与b为相反数。
2. 由于hmean的调用在try之中，这**引发了一个异常。**
3. 在引发异常后，throw语句**传出一个代表异常信息的字符串**，并**逐句返回**到try语句块。
4. 程序发现异常与该catch块匹配，**跳过try块直接执行catch块。**
5. **结束程序**

要注意的是，异常机制在执行的时候，栈中函数从**throw逐步返回到try**的过程中，就**不再执行两个语句块之间的函数**了，

但将会**依次为这之间函数的变量**进行**栈解退**，对象进行**析构**

异常机制的嵌套

try语句可以**被嵌套**，出现异常时将首先在**内层try语句**寻找**throw**，而后在**外层try语句**寻找异常

用对象传递异常

可以专门构造一个**包含异常信息**的对象，这个对象包含了可能出现的异常的信息

这样使用throw语句块时，就可以**传出一个异常对象**，catch就可以根据这个异常对象返回所可能出现的各种异常信息

如果有一个**异常类继承层次结构**，应该将catch位于**层次结构最下面的异常类的catch语句**放在**最前面**，将**捕获基类异常**的catch语句放在**最后面**

noexcept

在函数声明的分号前加**关键词noexcept**，告诉编译器这个函数**不会报错**，无需对其进行检查从而优化代码

```
double harm(double a)noexcept;//这个函数肯定不会有异常
```

exception类

头文件**<exception>**定义了**exception类**，可以用作其他**异常类的基类**

stdexception头文件

包含了一系列异常类

what()虚函数

what虚函数**返回一个字符串**，在**exception派生类**中可以重新定义它

```
#include<exception>
//一个异常类
class bad_hmean :public std:exception
{
public:
  const char* what(){return "bad arguments to hmean()"}// what函数
};
...
```

bad_alloc异常和new

头文件new中包含**bad_alloc类**的声明，这是从exception类公有派生而来的

如果**new申请不到内存**，bad_alloc类中的what函数就会返回一个字符串"std::bad_alloc"

terminate()

当程序中出现**未定义过的异常**时，如果包含了**头文件<exception>**，就首先调用函数**terminate()**

默认情况下terminate()调用**abort函数**，通过**set_terminate()函数**可以修改**terminate**将调用的函数

```
set_terminate(func)
```

断言机制

宏**assert**在**<cassert>**中定义，可以描述程序执行到断言处应该满足的条件，如果程序**不满足**这个条件，程序就执行**异常终止**

```
assert(x==1)//如果程序执行到这里x不等于1，就异常终止
```

当程序中处于**开发测试阶段**时，宏**NDEBUG未定义**，断言**有效**

程序**开发结束**时，宏**NDEBUG有定义**，断言就不再有效了