# 标签

# C++ 基本语法描述

- 每条语句占一行，每条语句以终止符（分号）结束
- 一行代码中不可分割的元素叫做标记（token），必须要用空格、制表符和回车将两个标记分开。
  空格、制表符和回车成为空白（white space)
- 每个函数都有一个开始花括号和一个结束花括号，这两个花括号各占一行
- 函数中的语句都相对于花括号进行缩进
- 与函数名称相关的圆括号周围没有空白
- C++ 程序是一组函数，而每个函数又是一组语句。
  C++程序由一个或多个被称为函数的模块组成。

预处理器编译指令`#include`

C++语句类型：

- 声明语句
- 赋值语句
- 消息语句
- 函数调用
- 函数原型
- 返回语句

## 声明语句

程序中的声明语句叫做定义声明（defining declaration），简称定义（declaration）
使用声明语句来指出储存类型并提供位置标签。
声明语句提供了两种信息：需要的内存以及该内存单元的名称。而编译器负责分配和标记内存细节。所有的变量都需要声明。
声明通产指出了要存储的数据类型和程序对储存在这里的数据使用的名称，这意味着它将导致编译器为变量分配内存空间。
除了定义声明外，还有引用声明（命令计算机使用在其他地方定义的变量）
通常，声明不一定是定义。
对于声明变量，尽量在首次使用前声明。

## 赋值语句

赋值语句将值赋给储存单元
符号= 叫做赋值运算符。赋值运算符可以连续使用，在此时，赋值将从右向左进行。（P20）

### 函数

有无返回值都叫函数

## using 编译指令

- 将 `using namespace std;` 放在函数定义之前，让文件中所有的函数都能够使用名称空间std中所有的元素。
- 将 `using namespace std;` 放在特定的函数定义中，让该函数能够使用名称空间 `std` 种的所有元素。
- 在特定的函数种使用类似 `using std::cout;` 这样的编译指令，而不是 `using namespace std;` ，让该函数能够使用特定的元素，如 `cout`。
- 完全不使用编译指令 `using` 而在需要使用std中的元素时，使用前缀 `std::`

编译指令 using namespace

# 输入输出

`<<` 插入运算符
`>>` 抽取运算符
用于处理输入输出的预定义对象（cin和cout）是istream和ostream的实例。

标记间的换行符和空格看作是可相互替换的。针对语句过长时（P22）

`cout` 会删除结尾的零。

## cin

cin使用空白来确定字符串的结束位置，并在字符串结尾自动添加空字符。

### cin类成员函数

| 函数 | 功能 |
| --- | --- |
| getline(s1,ArSize) | 读取一行输入，直到换行符。随后丢弃换行符. |
| get(s1,ArSize) | 读取一行输入，直到换行符。并将换行符留在输入队列中 |
| clear() | 清除之前输入错误产生的错误状态。 |

## cout

### cout控制符

控制符位于名称空间 `std` 中不能用作变量名。

- `endl` 控制换行
- `dec` 控制以十进制格式显示整数
- `hex` 以十六进制格式显示整数
- `oct` 以八进制格式显示整数

### cout成员函数。

1. `cout.put()`
   该函数显示一个字符
   2.`cout.setf()`
   迫使输出使用定点表示法，会覆盖掉 `cout` 会删除结尾的零的行为。

参数：`ios_base::fixed`()、`ios_base::floatfield`()、`ios_base::booaplpha`(设置一个标记，该标记命令cout显示true和false而不是0和1)

# 文本输入、输出

## 文本输出

输入一开始都是字符数据（文本数据），然后cin对象负责将文本转换为其他类型。

源代码文件就属于文本文件。

必须包含头文件`fstream`，然后声明自己的`ofstream`对象（最少一个）

使用文件输出的主要步骤：

1. 包含头文件fstream
2. 创建一个ofstream对象
3. 将该ofstream对象同一个个文件关联起来。
4. 就像cout那样使用该ofstream对象。
5. 使用后关闭文件。

声明格式：

```
ofstream outFile;
ofstream fout;
```

然后将这种对象与特定的文件关联起来：

```
outFile.open("fish.txt");
char filename[50];
cin>>filename;
fout.open(filename);
```

方法open接受一个C风格字符串作为参数，这也可以是一个字符串，也可以是储存在数组中的字符串。

声明一个ofstream对象并将其同文件管来拿起来之后，便可以像使用cout那样使用它。

所有的可用于cout的操作和方法（如<<、endl和setf()）都可以用于ofstream对象。

程序使用完该文件后，应该将其关闭

```
outFile.close();
```

方法不需要使用close()不需要使用文件名作为参数，因为outFile已将同特定的文件关联起来。

如果没关闭文件。在程序结束时，程序正常终止时将自动关闭它。

注意：在默认情况下，如果文件名已存在，open()方法将首先截断该文件，既将其长度截断到零——丢弃原有的内容，然后将新的输出加到该文件中。

# 文本输入

必须包含头文件`fstream`

必须声明自己的ifstream对象

```
ifstream infile;
ifstream fin;
```

将对象与特定的文件关联起来

```
infile.open("fish.txt");
char filename[50];
cin>>filename;
fin.open(filename);
```

方法open()接受一个C风格字符串作为参数，可以是一个字面字符串，也可以是储存在数组中的字符串。

如果试图打开一个不存在的文件用于输入，这种错误将导致ifstream对象进行输入时失败。检查文件是否被成功打开的首先方法是使用is_open()。

```
infile.open("bowling.txt");
if (!infile.is_open())
{
    exit(EXIT_FAILURE);
}
```

如果文件成功的被打开，方法is_open()将返回true

函数`exit()`的原型实在头文件`cstdlib`中定义的，在该头文件中，还定义了一个用于同操作系统通信的参数值`EXIT_FAILURE`。

函数exit()终止程序。

windows文本文件都以回车字符和换行字符结尾。通常情况下，C++在读取文件时将这两个字符转换为换行符，并在写入文件是执行相反的操作，

```
#include<iostream>
#include<fstream>
#include<cstdlib>
#include<ctime>
const int SIZE = 60;
int main()
{
    using namespace std;
    char filename[SIZE];
    ifstream inFile;
    cout << "Enter name of data file: ";
    cin.getline(filename,SIZE);
    inFile.open(filename);
    if (!inFile.is_open())
    {
        cout << "Could  not open the file " << filename << endl;
```

```
        cout << "Program terminating.\n";
        exit(EXIT_FAILURE);
    }
    double value;
    double sum = 0.0;
    int count = 0;

    inFile >> value;
    while (inFile.good())
    {
        ++count;
        sum += value;
        inFile >> value;
    }
    if (inFile.good())
    {
        cout << "End of file reached.\n";
    }
    else if (inFile.fail())
    {
        cout << "Input terminated by data mismatch.\n";
    }
    else
    {
        cout << "Input terminated for unknown reason.\n";
    }
    if (count == 0)
        cout << "No data processed.\n";
    else
    {
        cout << "Items read: " << count << endl;
        cout << "Sum: " << sum << endl;
        cout << "Average:" << sum / count << endl;
    }
    inFile.close();
    return 0;
}
```

**注意**检查文件是否被打开至关重要。

注意：应特别注意文件读取循环的正确设计。

- 程序读取文件不应超过EOF。如果最后一次读取数据时遇到EOF，方法eof()将返回true
- 在读取时，要注意类型需匹配。如果最后一次读取操作中发生了类型不匹配，方法fail()将返回true（EOF也返回true）。
- 可能会出现以外的问题。如文件受损或硬件故障。如果最后一次读取文件发生了这样的问题，方法bad()将返回true。
- 可以采用good()方法一并检测上述问题。没有错误时返回true。方法good()指出最后一次读取输入的操作是否成功。

```
while (inFile.good())
{
    body
}
```

分步检测出错误原因：

```
if (inFile.eof())//判断是否到达了eof
    cout<<"End of file reached.\n";
else if (inFile.faail())/判断是否是类型不匹配
    cout<<"Input terminated by data misamatch.\n";
else
    cout<<"Input terminated for unknown reason.\n";
```

分步检测代码跟在循环的后面，用于判断循环为何终止。

可以将两条输入语句用一条用作循环测试的输入语句代替。，用来代替good()方法检测。

```
while (inFile >> value)
{
    //循环体
    //跳出循环条件
}
```

# 数据类型

面向对象编程（OOP）的本质是设计并拓展自己的数据类型。设计自己的数据类型是让类型与数据匹配
内置的C++类型分为两组：基本类型和复合类型。
其中，基本类型分为整形和浮点型。基本类型又称算术类型。
复合类型包括数组、字符串、指针和结构。（复合类型是使用其他类型创建的数据类型）

程序储存信息必须记录3个基本属性：

- 信息将储存在哪里
- 要储存什么值
- 储存何种信息

> 机员及内存的基本单元是位（bit）。
> 字节通常指的是8位的内存单元，但是取决于具体实现有可能也是16位等。
> 八位组（octet）表示8位字节。

sizeof运算符返回类型或变量的长度，单位为字节。

头文件`climits`（老式实现位`limits.h`）中包含了关于整形限制的信息（定义了各种限制的符号名称）

初始化：初始化将赋值与声明合并在一起。
如果不对函数内部定义的变量进行初始化，该变量的值将是不确定的。这意味着该百年来个的值将是他被创建之前，相应内存单元保存的值。
初始化方式：

- 声明语句后面跟赋值语句
- 数据类型 变量名 = 表达式;
- C++使用不同的方式来初始化不同的类型。
- C++11初始化方式：用{}，大括号初始化方式可以用于任何类型，是通用的初始化语法。
- 而且在C++11中的初始化方式{}可以使用=，也可以不使用。
- 大括号内不含东西的话，变量将被初始化为零。

# 整形

整形关键字：`short`、`int`、`long`、`long long`、`char`、`bool`、`wchat_t`

| 整型类型 | 整形关键字 | 长度 |
| --- | --- | --- |
| 短整型 | short | 至少16位 |
| 整形 | int | 至少16位 |
| 长整形 | long | 至少32位 |
| 长长整形 | long long | 至少64位 |
| 字符型 | char | 8位 |
| 宽字符型 | wchar_t | 16位 |
| 布尔类型 | bool | |

整形可以分为符号类型和无符号类型。
仅当数值不会为负时才应使用无符号类型。
优点：可以增大变量能够储存的最大值。
声明，用关键词`unsiged`来修饰声明。unsigned本事就是unsigned int的缩写。

`short`、`int`、`long`和`long long`这四种类型都是符号类型。

`sizeof`运算符返回类型或变量长度，单位为字节。
头文件climit中包含了关于整形限制的信息。，定义了个中限制的符号名称。

## 头文件limits

头文件limits中定义了符号常量来表示类型的限制

# 浮点数

浮点数能够表示带小数部分的数。

| 类型 | 关键字 | 有效位 |
| --- | --- | --- |
| 单精度 | float | 32位 |
| 双精度 | double | 48位 |
| 长双精度 | long double | 80、96或128位 |

有效位是指数字中有意义的位。有效位数不依赖于小数点的位置。
关于有效位数的限制可以从头文件`cfloat`或`float.h`

需要注意：float只能表示到小数点后6位。

### 浮点数的书写

1. 标准小数点表示法：
   `2.34`
2. E表示法:适合表示非常大和非常小的数。e大小写均可。
   格式：`dddE+n`
   `2.3e+8`
   `2.6E-2`
   `-5.3e5`

**浮点数优点**：

1. 可以表示整数之间的值
2. 表示范围大

**浮点数缺点**
浮点数运算的速度通常比整数运算慢，且精度将降低。

# 数组（array）

作用：能够储存多个同类型的值。

## 数组声明

声明格式：

`typeName  arrayName[arraySize]`

- typeName指出了储存在数组中的每个元素的类型
- arrayName 数组名
- 表达式arraySize指定元素数目。必须是**整型常数**、**const值**或**常量表达式**。

arraySize在编译时都是已知的，即不能是变量。

例：

```
double arrayDou[5];
```

## 数组使用

**数组的使用**：可以使用**下标**单独访问数组中的元素。下标从0开始编号，最后一个元素的索引比数组元素少1。

**注意：**编译器不会检查使用的下标是否有效。也就是说下标所表示的元素有可能不在数组中。

使用带索引的方括号表示法来指定数组元素。比如：array[0]、array[1]、、、

对数组使用sizeof运算符：

- 对数组名使用，将得到整个数组的长度。
- 对数组元素使用，将得到元素的长度。
- 返回的长度都是以字节为单位。

## 数组初始化

只有定义数组时才能初始化，不能将一个数组数组赋给另一个数组。

1. 使用列表`{}`初始化。
    - 初始化数组时，可省略等号（=）。
    - 可不在打括号内包含任何东西，这将把所有元素都设置为零。
    - 列表初始化禁止窄缩转换。
    - 如果初始化时方括号（[]）内为空，C++将计算元素个数作为数组长度。
2. 使用下标分别给数组中的元素赋值。

初始化时，提供的值可以少于数组的元素数目，即对数组一部分进行初始化，此时编译器将把其他元素设置为0。

不初始化的危害：其元素值将是被分配的内存单元中的值，这将是不确定的。

# 字符串

C++内有两种字符串：C风格字符串和string类

要将字符串储存在数组中，有两种方式：

- 将数组初始化为字符串常量
- 将键盘或文件输入读入到数组中。

## C风格字符串

C风格字符串起源于C语言，并在C++中得到沿用。

C风格字符串本质上是以空字符（`\0`）结尾的一维char类型数组。因此，C风格字符串的长度总是比它所储存的字符多上做少一位空间。

声明C风格字符串：

```
char s1[8] = {'b','e','a', ',','\0'};
char s2[8] = {"HE"};
char s3[] = "HE";//会自动在结尾添加'\0'
char s4[8] = "HE";//会自动在结尾添加'\0'，后续的元素皆设置为'\0'
```

# C语言库函数

头文件`cstring` 老式：`string.h`

| 序号 | 函数 & 目的 |
| --- | --- |
| 1 | **strcpy(s1, s2);** 复制字符串 s2 到字符串 s1。 |
| 2 | **strcat(s1, s2);** 连接字符串 s2 到字符串 s1 的末尾。连接字符串也可以用 + 号，例如: `string str1 = "runoob"; string str2 = "google"; string str = str1 + str2;` |
| 3 | **strlen(s1);** 返回字符串 s1 的长度。 |
| 4 | **strcmp(s1, s2);** 如果 s1 和 s2 是相同的，则返回 0；如果 s1<s2 则返回值小于 0；如果 s1>s2 则返回值大于 0。 |
| 5 | **strchr(s1, ch);** 返回一个指针，指向字符串 s1 中字符 ch 的第一次出现的位置。 |
| 6 | **strstr(s1, s2);** 返回一个指针，指向字符串 s1 中字符串 s2 的第一次出现的位置。 |

需要注意的是，函数名代表着数组的地址。

注意：不能用关系运算符比较C风格字符串。

# string类

要包含头文件`string`，`string`类位于名称空间`std`中

string完全兼容C风格字符串的所有操作，并有所扩充。并且，程序能自动处理string的大小。

string类的设计能够将string对象作为一个实体，也可以作为一个聚合对象。

string对象不适用空字符来标记字符串末尾。

## string操作

1. 赋值：使用等号（=）可以将一个string对象赋给另一个string对象。

```
string str1;
string str2 = "HE";
str1 = str2;
```

2. 拼接：使用加号（+）可以将string对象拼接起来。

```
string str3;
str3 = str1 +str2;
```

3. 附加：使用（+=）将字符串附加到string对象的末尾。

```
str1+=str2;
```

4. 比较：重载运算符`!=`，至少有一个操作数为string对象，另一个操作数可以是string 对象，也可以是C风格字符串。

```
str1!=str2
```

**注意：**操作符对象可以是C风格字符串和string对象或两个string对象\

# 结构

一个结构可以储存多种类型的数据。

结构声明定义了结构的数据属性，并定义了一种新类型。

创建结构两步：

1. 定义结构描述。描述并标记了能够储存在结构中的各种数据类型
2. 按描述创建结构变量（结构数据对象）

## 定义结构

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

其中，object_name可以省略。object_name代表着可以在定义结构时便可以创建结构变量。用这种方式创建的还能量也可以同时被初始化。

声明结构变量：

```
struct type_name object_names;
type_name object_names;
```

在声明变量时关键字struct可以省略。

**注意：**与变量不同，C++倡导使用外部结构声明。

**注意：**还可以声明没有名称的结构类型（省略名称），同时定义一个结构类型和一个这种类型的变量。

## 使用结构

我们使用结构类型，最常使用的便是结构中的成员。

1. 使用**成员运算符（.）**来访问结构成员。类的访问成员函数便源于此。
2. 结构初始化使用列表初始化`{}`。列表中的数据的类型顺序要能与定义结构时成员类型的顺序一致。其余则和普通使用列表初始化无疑。
3. 结构可以作为函数的参数，也可以作为它的返回值。
4. 成员赋值：可以使用赋值运算符（=）将一个结构赋给另一个结构。

## 结构数组

可以创建元素为结构的数组。方法与创建普通数组无异。

初始化数组：两重`{}`。初始化的用逗号分隔的值列表中的每个值本身又是一个被括在花括号中、用逗号的值列表。

# 共用体

共用体能够储存不同的数据结构，但只能同时储存其中一种结构。

声明共用体和创建共用体变量与结构无疑，使用也并无差别。

共用体的长度为其最大成员的长度。

共用体用途：当数据项使用两种或更多种数据格式（但不会同时使用时），可节省空间。因此常用于操作系统数据结构或硬件数据结构。

# 枚举

枚举（enum）可以创建符号常量，也可以定义新类型。使用枚举的句法与结构类似。

## 枚举的使用

可以将整数值赋给枚举量，第一个枚举量对应0，以此类推。需要使用强制类型转换，前提是int值是有效的。

在不进行强制类型转换的情况下，只能将定义枚举时使用的枚举量赋给这种枚举的变量。

不能将非enum值赋给enum变量。

枚举只能使用赋值运算符。（因为只定义了赋值运算符）

枚举是整形，可被提升为int类型，但int类型不能自动转换为枚举类型。因此，可在算术表达式中同时使用枚举和常规整数。

## 设置枚举值

1. 枚举值默认与整数一一对应，但可以通过赋值运算符来显式地设置枚举量地值。
2. 指定的值必须是整数，而却也可以值定义其中的一部分枚举量的值。这种情况下，后面没被初始化枚举量的值比前面的枚举量大1.
3. 可以创建多个值相同的枚举量。
4. 可以将long long 类型的整数值赋给枚举量。

# 指针

指针是一个变量储存的是值地地址，而不是值本身。对一个不同变量运用地址运算符（&）便可得到它地地址。

对指针使用间接值（解除引用）运算符（`*`）便可以得到改地址处储存的值。

cout显示地址通常使用十六进制。

**指针与常规变量区别：**

- 对于常规变量。值是指定的量，而地址是派生量。

- 对于指针。地址是指定的量，值是派生量。

## 声明和初始化指针。

指针声明必须指定指针指向的数据地类型。

```
typeName * pName;
```

`*`左右两侧空格：

- 是可选的
- 风格根据语言变。
- `typeName *pName`强调*pName是typeName类型地值
- `typeName* pName`强调typeName*是一个新类型，指向typeName类型的指针。
- 两侧空格对于编译器没有差别。

在声明多个指针时，每一个指针都需使用一个*。

在C++中，`typeName*`是一种复合类型，是指向`typeName*`的指针。

因此需要注意：指针变量不仅是指针，而且是指向特定类型的指针。说明指针是基于其他类型的数据类型。

可以在声明指针时初始化指针。

注意，被初始化的是指针，而不是它指向的值。

注意：一定要在对指针应用解除引用运算符之前，将指针初始化为一个确定的、适当的地址

## 使用new关键字来分配内存。

指针可以在运行阶段分配为命名的内存以存储值。分配之后，只能通过指针来访问内存。

C++也可以使用C语言的malloc()来分配内存。在C++中，更普遍的是使用new关键字。

为一个数据对象获得并指定分配内存的通用格式：

```
typeName * pointer_name = new typeName;
```

需要在两个地方指定数据类型：用来指定需要什么样的内存和用来生命合适的指针。

使用new关键字分配的内存块是从堆或自由储存区的内存去于分配内存。

## 使用delete关键字释放内存

delete运算符，它使得使用完内存后，能够将其归还给内存池。归还或释放的内存可供程序的其他部分使用。

通用格式：

```
delete pointer_name;
```

注意：delete 后跟的指针名，只是由new关键字分配内存的指针。不能使用delete来释放声明变量（即不是new分配的）所获得的内存。

注意：delete释放内存，但不会删除指针。释放内存后的指针仍可以被重新赋予内存。

注意：new 和delete一定要成对使用。即new为指针分配内存后，一定要有delete为其释放内存，否则会放生内存泄漏。

注意：对空指针使用delete是安全的。

注意：不能连续两次释放同一个指针。

所以，由于delete的特性，我们最好不要让多个指针同时指向同一个内存块。

## 使用new来创建动态数组

**静态联编：**在编译时给数组分配内存，意味着数组实在编译时被加入到程序中的。

**动态联编：**可以在程序运行时选择数组的长度，意味着数组是在程序运行时创建的。（动态数组）

创建动态数组格式：

```
typeName * pointer_name = new typeName [num_elements];
```

使用new关键字创建数组之后，若要释放内存。我们要用另一种格式的delete来释放。

```
delete [] pointer_name;
```

方括号含义：告诉程序，应释放整个数组，而不仅仅是指针指向的元素。

new []和delete[]时成对的。

## 指针和数组

指针和数组基本等价的原因在于指针算数和C++内部处理数组的方式

**指针算数**：

1. 将指针变量加1后，增加的量等于它指向的类型的字节数。
2. 可以将一个指针减去另一个指针。获得两个指针的差。
   执行减法运算时，将得到一个**整数**，仅当两个指针指向同一个数组时才有意义（也可以指向超出结尾的一个位置），得到两个元素之间的间隔。用在数组时，是求数组元素数目的简便方法。

C++将数组名解释为地址。

**数组表示式的本质**

使用数组表示法时，C++将执行下面的转换：

```
arrayname[i] becomes *(arrayname +i)
```

使用的是指针时，也将执行同样的转换：

```
pointername[i] becomes *(pointername + i)
```

**指针和数组的区别**

1. 可以修改指针的值，而数组名是常量。
2. 对数组应用sizeof运算符得到的是数组的长度，而对指针应用sizeof得到的是指针的长度，即使指针指向的使一个数组。（在运用sizeof运算符的时候，C++不会将数组名解释为地址。）

**数组名的两大作用：**

1. 数组名被解释为数组第一个元素的地址。同样，数组名也是整个数组的地址。
2. 数组名代表了整个数组。（例：对数组运用sizeof运算符）将地址运算符&用于数组名时，将返回整个数组的地址。

**数组表示法和指针表示法**

使用方括号数组表示法等同于对指针解除引用。

所以，在很多情况下，可以以同样的方式使用指针名和数组名。

# 指针和字符串

在C++中，用引号括起的字符串也是第一个元素的地址。

在cout和多数C++表达式中，char数组名、char指针以及用引号括起的字符串常量都被解释为字符串第一个字符串的地址。

C++将C风格字符串视为地址。

实例：

▶ 代码实例

**注意**，字符串字面值是常量，在用指针储存字符串字面值时要在声明中使用const关键字。如此，编译器将禁止指针指向地位置中的内容，同时也使得指针本身免受影响。

**不同编译器对于字符串字面值的处理：**

- 有些编译器将字符串字面值是为制只读常量，如果试图修改他们，将导致运行阶段错误。
- 有些编译器只使用字符串字面值的一个副本来表示程序中所有的该字面值。但存在下述问题。

**C++不能保证字符串字面值被唯一地储存**，如果在程序中多次使用了一个字符串字面值，则编译器可能会储存字符串的多个副本，也可能只储存一个副本。

所以，不能使用字符串常量或为被初始化的指针来接受输入，可以使用string对象。

在将字符出读入程序时，应使用已分配的内存地址。该地址可能是数组名，也可以是使用new初始化的指针。

**cout显示指针的策略：**

- 如果给cout提供一个指针，他将打印地址。但如果指针的类型为char*，则cout将显示指向的字符串。
- 如果要显示的是字符串的地址，则必须将这种指针强制转换为另一种指针类型。

**获得字符串的副本的方法**

1. 需要分配内存来储存该字符串
   方法：声明一个数组，或使用new（更加自由）
2. 将原数组中的字符串复制到新分配的内存当中。
   使用库函数strcpy：`strcpy(ps,animal);`。
   ps:目标地址，animal：要复制的字符串

**注意**：需要注意要复制的字符串的长度。如果长度比目标数组大，函数将会把剩余的部分复制到数组后面的内存字节中，这可能会覆盖程序正在使用的内存。

应对方法：库函数`strncpy()`，该函数接受第三个参数：要复制的字符串的最大值。

**注意：**这个最大数值要少于目标数组的长度，比如说目标数组长度是20，则这个参数最大是19。因为用数组储存字符串时，要保留一位给`\0`，以标记字符串的结尾。 这个`\0`是函数自动添加的。

**注意：**应使用`strcpy()`或`strncpy()`将字符串赋给数组，而不是赋值运算符。

# 使用new创建动态结构

动态联编优于静态联编，结构亦然。

创建和使用结构步骤：

1. 创建结构
2. 访问成员。

**访问动态结构成员：**

创建动态数组时，不能将成员运算符句点用于结构名，因为这种结构没有名称，只有地址。

这种情况下因该使用箭头成员运算符（`->`）。

另一种方法是对只想结构的指针使用接触引用运算符再使用成员运算符。

例：

```
(*struct_pointername).member
(*ps).price
```

在这种情况下，根据运算符优先规则应该使用括号。

**不同情况下在结构上使用的运算符：**

1. 结构标识符是**结构名**，就使用句点运算符（`.`）。
2. 结构标识符是**指向结构的指针**，则使用箭头运算符（`->`）。

# 指针和const

**将const用于指针有两种情况：**

1. 将指针指向一个常量对象。作用是防止通过该指针来修改指针所指向的值。
2. 将指针本身声明为常量。作用是防止改变指针指向的位置。

**将const用于指针后的指针赋值：**

1. 可以将const变量的地址赋给指向const的指针。
2. 不可以将const的地址赋给常规指针（可以通过强制类型转换来强制赋值。）。
3. 仅当只有一层间接关系（指针指向的都是常规变量、常量）时，才能将非const指针赋给const指针
4. 如果数据类型本身并不是指针，则可以将const数据或非const数据的地址赋给指向cosnt的指针，但只能将非const数据的地址赋给非const指针。
5. 不能将常量数组的地址赋给非常量指针。即，不能将数组名作为参数传递给使用非常亮形参的函数。

**将指针参数声明为指向常理数据的指针有两条理由：**

- 这样可以避免无意间修改数据而导致的编程错误
- 使用const是的函数能够处理const和非const实参，否则只能接受非const数据。

# 变量

变量声明格式：
`typeName valueName;`
声明中所使用的类型描述了信息的类型和变量名（使用符号来表示其值）

## 变量名命名方案

- 在名称中只能使用

用以区分数据类型的变量名的前缀和后缀

但在命名的风格中，一致性和精度是最重要的。

## 变量初始化

使用`{}`的初始化方式被称为列表初始化。常用于给复杂的数据类型提供值列表。

C++11将使用大括号的列表初始化作为一种通常的初始化方式。

## auto声明方式

作用：让编译器能能够根据初始值的类型判断变量的类型。

处理复杂类型时，auto将显出优势。

```
std::vector<double> scores;
auto pv = scores.begin();
```

**注意**：auto以前和现在的含义有所不同。

# 常量

创建常量的方法：

1. `const type name = value;`
2. `#defiane`

关键字`const`叫限定符。

需要注意，在使用const创建变量时，要注意对const初始化。
在创建变量时，推荐使用`const`，而不是`#define`。一是`const`能够指定数据类型，二是可以使用C++的作用域规则将定义限制在特定的函数或文件中，三是可以将`const`用于更复杂的类型。

## 常量命名方法

1. 通常会将常量名称的首字母大写。
2. 整个名称大写，在使用`#define`创建常量时常常使用。
3. 在名称前添加 k 字母。

# 整形字面值

整形字面值（常量）是显性书写的常量。
C++以3种方式书写常量，是根据进制的不同而划分的：基数为10、8、16.
C++表示法：
C++使用前一两位来标识数字常量的基数。

- 第一位为1~9，则为十进制
- 第一位为0，第二位为1~7则为8进制
- 前两位为0x或0X则为十六进制。
  默认情况下cout以十进制格式显示整数，除非特别设定。
  而数据是以二进制的方式储存在计算中。

# 字符串常量

## 字符串常量（字符串字面值）与字符常量区别：（以`'H'`和`"H"`为例）

两者不能互换。字符常量（如`'H'`）是字符串编码的简写形式。而`"H"`是由字符H和\0组合在一起的字符串。

而且，`"H"`实际上表示的是字符串所在的内存地址。

## 拼接字符串常量

拼接字符串字面值：任何两个由**空白（空格、制表符、换行符）**分隔的字符串常量都将自动拼接为一个。

在拼接时，不会在被连接的字符串之间添加空格。这时，第一个字符串中的`\0`字符将被第二个字符串中的第一个字符取代。

拼接字符串可用于字符串变量初始化时。

```
    char s1[] = "siud" "sdf\n";
    cout << s1;
    cout << "sgd" "sdf\n";
    cout << "sgd"
        <<"sdf\n";
```

# 确定常量的类型

整形常量通常情况下会储存为int类型，除非数值太大，或有后缀。
后缀是放在数字常量后面的字母，用于表示类型。

- l、L表示long常量
- u、U表示unsigned int常量
- ul、uL表示unsigned long常量。（顺序不限）
- ll、LL表示long long 常量
- ull 、Ull、uLL、ULL表示unsigned long long

浮点常量通常情况下会储存为double。

- f、F表示float

- l、L表示long double

# 类型转换

C++会自动进行类型转换

- 将一种算术类型的值赋给另一种算数类型的变量时，C++对值进行转换。
- 表达式中包含不同的类型时，C++对值进行转换。
- 将参数传递给函数时，C++将对值进行转换。

## 1.初始化和赋值时进行的转换

在将一种类型的值赋给另一种类型的变量时，值将被转换成被接受变量的类型。

需要注意，将一个取值范围小的值赋给另一个取值范围大的类型不会有什么问题，比如`int`类型赋给`long`。

但将取值范围大的值赋给另一个取值范围大的类型会出现某些问题：

**潜在的数值转换问题：**

| 转换 | 潜在的问题 |
| --- | --- |
| 将较大的浮点类型转换为较小的浮点类型，如double 转换成float | 精度（有效数位）降低，值可能超出目标类型的取值范围，结果将不确定。 |
| 将浮点类型转换为整形 | 小数部分丢失，值可能超出目标类型的取值范围，结果将不确定。 |
| 将较大的整形转换为较小的整形，如将long转换为short | 值可能超出目标类型的取值范围，通常只复制右边的字节。 |

任何数字值或指针值都可以被隐式转换为bool值。

需要注意：对于bool类型，0为false，非零值为true。

将布尔值赋给int类型时，false被转换为0，true被转换为1。

## 2. 以{}方式初始化时进行的转换（C++11）

在使用`{}`初始化时时，类型转换比较严格。

不允许**窄缩**，不允许将浮点型转换为整形。

当编译器知道目标变量能够正确地储存赋给它的值，就可以将不同的整形之间转换或将整形转换为浮点型。
例：

```
int x = 66;
const x1 = 66;

char c1 = {66}; //能正确赋值
char c2 = {x};//不能,因为x是一个变量，其值可能会很大
char c2 = {x1}; //能
```

## 3.表达式中的转换

表达式中包含不同的算术类型时，C++将执行自动转换：

1. 一些类型出现时便会自动转换
2. 有些类型在与其他类型同时出现在表达式中时将被转换

自动转换：**整型提升**：

bool、char、unsigned char、signed char和short将被转换为 int。（ture 被转换为1、false被转换为0）.

`short`的长度不同，unsigned short在整形提高时转换成的类型不同：

- short比int短，则`unsigned short`会被转换成int
- short与int长度相同，`unsigned short`则会被转换成`unsigned int`，确保不会丢失数据。

同理，wchar_t会被提升为下列类型中第一个宽度足够储存wchar_t取值范围的类型：int、unsigned int、long或unsigned long。

算数运算时：当运算涉及两种类型时，较小的类型将被转换为较大的类型。编译器通过校验表来确定在算数表达式中执行的转换。

▶ 校验表

总的来说，就是操作数表示宽度宽的优先。

▶ 整形级别

## 4. 传递参数时的的转换

传递参数时通常由C++函数原型控制。这个**控制**也可取消，但不推荐。

传递参数时，char和short类型（有符号和无符号）会被整形提升。

再将参数传递给取消**控制**的函数时，将float提升为double。

## 5.强制类型转换

强制类型转换可以显性地进行类型转换。

格式：

```
(typename) value  //来自C语言
value (typename)  //来自C++，格式类似于函数调用
```

**注意**：强制类型转换不会修改被转换的变量本身，而是创建一个新的、被指定类型的值，可以在表达式中使用这个值。

强制类型转换运算符：

```
static_cast<typeName> (value)
```

转换运算符比传统的强制类型转换更为严格。

▶ 类型转换实例

# 类型别名

建立类型别名：

1. 使用预处理器

```
#define BYTE char //用char替换所有的BYTE从而使BYTE成为char的别名
```

2. 使用C++（和C）关键字`typedef`来创建别名

```
//通用格式：
typedef typeName aliasName;
//例：
typedef char byte
```

可以使用类型别名创建指针的别名

typedef不会创建新类型，而只是为已有的类型创建一个新名称。

# 循环

在设计循环时要注意以下指导原则：

- 指定循环终止的条件
- 在首次测试之前初始化条件
- 在条件被再次测试之前更新条件。

# for循环

for循环的组成部分要完成下面的步骤

1. 设置初始值
2. 执行测试，检测循环是否应当继续进行。
3. 执行循环操作
4. 更新用于测试的值。

for循环格式：

```
for(initialization;test-expression;update-expression)
    body
```

注意：C++语法将整个for看成一条语句。循环只执行一次初始化。

注意：for是入口条件循环，每次循环之前都将计算测试表达式的值。

注意：for语句是C++关键字，因此编译器不会将for是为一个函数。
C++常用的方式是再for和括号之间加上一个空格，而省略函数名与括号之间的空格。

# while循环

while也是入口条件循环。

检测数组中特定的字符是不是空字符：

```
while (name[i] != '\0')
//或：
while (name[i])
```

**不会将某些数值排除在外的、终止循环的方式：**将`cin>>`用作测试条件消除了这种限制，因为他接受任何有效的数字输入（需要使用循环来输入数字时，可以考虑这种方式）。非数字输入将设置了一个错误条件，禁止进一步读取输入。如果程序在输入循环后还需要进行输入，则必须使用`cin.clear()`重置输入，然后还可能需要通过读取不合法的输入来丢弃他们。

```
while (!cin)
{
    statements
}
```

## 延时循环

头文件`ctime`提供了解决方案：

定义了符号常量`CLOCKS_PER_SEC`:该常量等于每秒钟包含的系统时间单位数。

系统时间除以这个值，可以得到秒数，获或将描述乘以`CLOCKS_PER_SEC`，可以得到以系统时间单位为单位的时间。

函数clock()返回程序开始后所用的系统时间。但是返回时间不一定是秒，返回类型在不同的系统上可能不一样。

ctime将clock_t作为clock返回类型的别名。

```
#include<iostream>
#include<ctime> //describes clock() function, clock_t type
int main()
{
    using namespace std;
    cout << "Enter the delay time, in seconds: ";
    float secs;
    cin >> secs;
    clock_t delay = secs * CLOCKS_PER_SEC;
    cout << "starting\a\n";
    clock_t start = clock();
    while (clock() - start < delay)
        ;
    cout << "done \a\n";
    return 0;
}
```

## do-while循环

是出口循环。

```
do
    body
while (test-expression)
```

## 基于范围的for循环

# 运算符

## 算术运算符

5中基本的算术运算：加减乘除和求模。
运算符及操作数构成了表达式。
常量和变量都可用用于操作数
求模运算符要求两个操作数必须是整数。

## 赋值运算符

`=`

### 结合赋值运算符

| | | |
|---|---|---|
| **+=** | **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 |

# 自增自减运算符

自增自减运算符分为前缀和后缀。

对于内置类型，采用那种格式不会有差别；但对于用户定义的类型，如果用户定义的递增和递减运算符，则**前缀格式的效率更高。**

因为后缀运算符首先要复制一个副本将其加1，然后将复制的副本返回。而前缀是将值直接加1，然后返回结果。

## 在指针中运用递增、递减运算符

指针递增和递减遵循算数规则。

运用到指针上，会增加，或减少其指向的数据类型占用的字节数。

可以将接触引用运算符和递增、递减运算符结合起来使用。

首先，将什么接触引用，讲什么递增，这取决于运算符的位置和优先级。

前缀递增、前缀递减和接触引用运算符的优先级相同，以从**右向左**的方式进行结合。

```
*++pt//先将++应用与pt，然后将*应用于被递增后的pt
++*pt//先取得pt指向的值，然后将这个值加1
```

后缀递增、后缀递减优先级相同，但比前缀运算符的优先级高，这来给你个运算符从**左向右**的方式进行结合。

```
*pt++//先将++运用于pt，随后是*。但++是后缀，意味着，在对pt接触引用之后才执行递增操作。
```

# 其他运算符

| 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。 |
| 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。 |

# 表达式

任何值或任何有效的值和运算符的组合都是表达式。再C++中每个表达式都有值。

关系表达式将被判定为bool值true或false

<<运算符的有限级比表达式中使用的运算符高，所以要使用括号。

```
cout<<(x>2);
```

**表达式的副作用**：副作用指的是计算表达式时对某些东西（如储存在变量中的值）进行了修改

**顺序点**是程序执行过程中的一个点，对于C++而言每个分号都是一个顺序点。在C++11文档中，不再使用了，旨在更清晰的描述多线程编程。

**完整表达式**不是另一个更大表达式的子表达式。

# 函数

要使用函数，要有

- 提供函数定义
- 提供函数原型
- 调用函数

# 定义函数

两类函数：有返回值和没返回值（void函数）。

void函数通用格式：

```
void functionNamae(parameterList)
{
    statement(s)
    return;
}
```

其中，parameterList制定了传递给函数的参数类型和数量。可选的返回语句标记了函数的结尾。否则，函数将在右花括号出结束。

通常，可以用void函数来执行某种操作。

有返回值的函数将生成一个值，并将它返回给调用函数。这种函数的类型被声明为返回值的类型。

```
typeName functionName(parameterList)
{
    statements
    return values;
}
```

有返回值的函数必须要使用返回值语句，以便将值返回给调用函数。值可以是任意形式，但值的类型必须是typeName，或可以被转换为typeName类型。

C++对于返回值的类型有一定的限制：不能是数组，但可以是除此之外的任何类型。（但可以把数组作为结构或对象的组成成分来返回）。

函数在执行返回语句后结束。如果函数包含第一条语句，则函数在执行遇到的第一条返回语句后结束。

# 函数原型和函数调用

## 函数原型的重要性

函数原型描述了函数到编译器的借口，他将函数返回值以及参数的类型和数量告诉编译器。

## 原型的语法

函数原型是一条语句，因此必须以分号结束。

函数原型与函数定义中的函数头类似，其中的变量名是可选的。原型中的函数名相当于占位符。

## 原型的功能

- 编译器正确处理函数返回值
- 编译器检查使用的参数数目是否正确
- 编译器检查使用的参数类型是否正确；如果不正确，则转换为正确的类型（如果可能的话）

仅当有意义时，圆形话才会导致类型转换。

在编译阶段进行的原型被称为静态类型检查。

静态类型检查课捕获许多在运行阶段非常难以捕获的错误。

# 函数参数和按值传递

用于接受传递值的变量被称为形参。又称参量

传递给函数的值被称为实参。又称参数。

因此参数传递将参数赋给参量。

在函数声明中声明的变量（包括参数）时该函数私有的。被称为局部变量。（自动变量）自动被分配和释放的。

在定义函数时，也在函数头中使用由逗号分隔的参数声明列表。在待用函数时，也是用逗号将参数分隔开。

圆形中的变量名不必与定义中的变量名相同，而且可以省略。

# 函数具体应用

## 函数与数组

函数是处理更复杂的类型的关键。

要使函数能处理数组，需要在形参列表中声明一个数组：

```
int sum_arr(int arr[],int n)
```

arr实际上是一个指针，但在函数体内被看作数组。

在C++中，当且仅当用于函数头或函数原型中，int *arr和int arr[]的含义才是相同的。他们都意味着arr是一个int指针。

对于数组遍历来说，使用指针加法和使用数组下标是一样的。

**将数组作为参数的意义。**：将数组的位置（地址）、包含的元素种类（类型）以及元素数目（n变量）提交给函数。

将数组地址作为参数可以节省赋值整个数组所需的时间和内存。但是增加了破坏数据的风险，可以用const限定符来解决这个问题。

**使用const保护数组**：在声明形参时使用关键字const

```
void show_array(const double ar[] ,int n);
```

该声明表明，指针指向的时常量数据。但这并不意味着原始数组必须是常量，而只是意味着不能在show_array()函数中使用ar来修改这些数据。因此函数将数组视为只读数据。条件是：只有**一层间接关系**，即：只能用于指向基本类型的指针，二维数组就不能使用该技术。

**使用数组区间的函数**：

1. 将指向数组起始处的指针作为一个参数，将数组长度作为第二个参数。（指针指出数组的位置和数组类型）
2. 指定元素区间（range），传递两个指针：一个标记数组的开头，另一个指针表示数组的尾部。STL方法使用“超尾”概念来指定区间。对于数组而言，表示数组结尾的参数将是指向最后一个元素后面的指针。

# 函数与二维数组

与一维数组一样，二维数组的数组名依旧被视为其地址。

因此在函数中声明中，二维数组的形参应是一个指针。

因为二维数组本质上是指向指针的指针。

假如由如下代码：

```
int data[3][4] = {{1,2,3,4},{9,8,7,6},{2,4,6,8}};
int total = sum(data,3);
```

要怎么声明sum()函数的原型?

```
//1
int sum(int (*ar)[4],int size);
//2
int sum(int ar[][4],int size);
//两种格式含义相同
```

我们要声明一个指向由4个int著称的数组的指针。函数参数不能是数组。

虽然该函数执行了4列的数组，但是长度变量制定了行数，因此此函数坠数组的行数没有限制。

**注意：**数组的方括号与指针的解除引用运算符含义一样。

对于二维数组：

```
ar2[r][c] == *(*(ar2 + c) + c)//一样的
```

**注意：**对于二维数组，不能使用const来保护数据，因为const只能用于只想基本类型的指针。这与二维数组的本质有关。

# 函数和C风格字符串

## 将C风格字符串作为参数的函数

如果要将C风格字符串作为参数传递给函数，则表达字符串的方式有3种：

1. char数组
2. 用引号括起的字符串常量
3. 被设置为字符串的地址的char指针。

因为上述3者的本质都是char指针（`char*`），所以函数的形参类型应为`char*`类型，当然，也可以为`char[]`类型

C风格字符串与char数组之间的一个重要区别是，字符串内有内置的结束字符（没有结束字符的char数组只是一个数组）。

这意味着可以省去作为字符串长度的形参，而由函数检查字符串（结束条件是遇到空值字符）。

其中，用来储存字符串长度的变量可以声明为`unsigned int`类型，因为字符串的长度不能为负数。

处理字符串中字符的标准形式：

```
while (*str)
{
    statements
    str++;
}
```

当指针指向字符串结尾时，`*str`的值为`\0`，从而结束循环。

### 返回C风格字符串的函数

函数无法返回一个字符串，但可以返回字符串的地址。（效率也更高）

可以使用new关键字来动态的为字符串分配内存。

可以在函数内使用new关键字分配内存，然后在主函数内使用delete来释放内存。

但缺点是，一定要使用delete来释放内存。

## 函数和结构

结构的特殊性在于，整个结构可以被视为一个整体。

可以按值传递结构，这时，函数使用的是结构的副本。

与数组不同，结构名只是结构的名称，想要获得结构的地址还要使用&运算符。

推荐将结构的地址作为参数传递。因为如果按值传递结构，当结构比较大时，会降低系统运行速度。

但当结构比较小时，按值传递结构最合理。

# 递归

递归函数调用自己

```
void recurs(arguments)
{
    statements1
    if (test)
        recurs(arguments)
    statements2
}
```

以上是void类型的递归函数，包含一个递归调用的递归

递归方法有时被称为分而治之策略。

也可以有包含多个递归调用的递归

# 函数指针

函数也有地址。函数的地址是储存其机器语言代码的内存的开始地址

要将函数指针传递给函数：

1. 获取函数的地址
   使用函数名即可,要将函数作为参数传递必须使用函数名。一定要区分传递的是函数还是函数的返回值
2. 声明函数指针
   声明指针时，必须指定指针指向的函数类型。声明应指定函数的返回类型以及函数的特征标（参数列表）

   ```
   //函数原型
   double pam(int);
   //函数指针
   double (*pf)(int);
   ```

   因此，只需将函数原型中的函数名替换为`(*pf)`即可，必须要有括号
3. 使用指针来调用函数
   `(*pf)`与函数名意义相同。
   调用形式：
   - `(*pf)(参数列表)`
   - `pf(参数列表)`

   这两种调用背后的思想是不同的，但是C++都认为正确。

### 函数指针探幽

```
const double * f1(const double ar[], int n);
const double * f2(const double [] , int)
const double * f3(const double * , int)
```

以上三个函数原型的特征标相互等价。

# 库函数

## Cctype