

在参数传递过程中，对形参和实参的要求是（）

- ☐ A 函数定义时，形参一直占用存储空间
- ☒ B 实参可以是常量、变量或表达式
- ☐ C 形参可以是常量、变量或表达式
- ☐ D 形参和实参类型和个数都可以不同

提交

决定C++语言中函数的返回值类型的是（ ）

- ☐ A return语句中的表达式类型
- ☐ B 调用该函数时系统随机产生的类型
- ☐ C 调用该函数时的主调用函数类型
- ☒ D 在定义该函数时所指定的数据类型

设已定义i和k为int类型变量，则以下for循环语句()

```
for ( i =0; i!=0; i++ )
```

```
cout<< " * * * * n ";
```

- ☐ A 判断循环结束的条件不合法
- ☐ B 是无限循环
- ☒ C 循环一次也不执行
- ☐ D 循环只执行一次

提交

执行以下程序段后, 输出的结果为 ()

```
int a=4,b=5,t=0;
```

```
if(a>b) t=a;a=b;b=t;
```

```
cout<<"a="<<a<<"",b="<<b<<endl;
```

- ☐ A a=5,b=4
- ☐ B a=4,b=5
- ☒ C a=5,b=0
- ☐ D 语法错误

提交



桂林电子科技大学
GUILIN UNIVERSITY OF ELECTRONIC TECHNOLOGY



人工智能学院
SCHOOL OF ARTIFICIAL INTELLIGENCE

第四讲 用户自定义数据类型

赵彬

zhaobin@guet.edu.cn

4.1 前言

单一的基本数据类型适用于简单的数据对象处理，面对复杂的数据对象则需要通过将一些简单的基本数据类型组合成较为复杂的数据类型才能予以描述和解决。C++中提供了**用户自定义数据类型**。

用户自定义数据类型也称**构造数据类型**，包括**数组、指针、引用类型、枚举、结构和联合**。

4.2 数组

数组是一组有序数据的集合。数组中各数据的排列是有一定规律的，下标代表数据在数组中的序号，用一个数组名和下标唯一确定数组中的元素，数组中的每一个元素都属于同一个数据类型。数组可以是一维的，也可以是多维的。

4.2.1 一维数组的定义

- 一维数组的本质就是在内存中申请一段连续区域，用于记录多个类型相同的数据。定义的一般形式如下：

```
数据类型 数组名[常量表达式];
```

其中：

- 数据类型可以是除void类型以外的任何一种数据类型，包括基本数据类型和已经定义的用户自定义数据类型
- 数组名即数组的名称，用于记录连续内存区域内在内存中的首地址。数组名的命名规则和变量名相同
- 常量表达式必须是一个unsigned int类型的正整数，表示数组的大小或者长度，也就是数组所包含数据元素的个数
- 必须为数组指定大于或等于1的维数

4.2 数组

4.2.1 一维数组的定义

维数值必须是常量表达式，即不能用非const变量来指定数组维数。

```
const int buf_size=32, max_size=200;  
int staff_size = 27;  
char a[buf_size];           ✓  
float b[max_size-5];        ✓  
int c[staff_size];          ✗
```


4.2 数组

4.2.1 一维数组的定义

数据类型相同的多个数组可以在同一条语句中予以定义。

```
float a1[5], a2[10];
```

数据类型相同的基本数据类型变量和数组同一条语句中予以定义。

```
char s, a [5];
```

4.2 数组

4.2.2 一维数组的使用

一维数组可以被显式的用一组数来初始化，其一般语法格式如下：

```
数据类型 数组名[常量表达式]={初值1, 初值2, ……初值n};
```

其中：

- 数组可以被显式的用一组数据来初始化，这组数据用逗号分开，放在大括号中

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

- 初值与数组元素存在一一对应的关系，初值的总数量不能够超过指定的大小
- 被显式初始化的数组是不需要指定维数值的，此时编译器会根据列出来的元素个数确定数组的维数

```
int a[]={0,1,2,3};
```

- 若初值的个数小于数组的大小，则未指定值得数组元素被赋值为0。在函数外部定义数组，如果没有对数组进行初始化，其数组元素的值为0
- 一个数组不能被另外一个数组初始化，也不能被赋值给另一个数组，需要把一个数组拷贝到另一个数组中，必须按照下标顺序依次拷贝每个元素

4.2 数组

4.2.2 一维数组的使用

【例 1】利用for循环对数组元素依次赋值

```
#include <iostream>
using namespace std;
int main()
{
    int a[10], i;
    for (i = 0; i < sizeof(a) / sizeof(int); i++)
    {
        cin >> a[i]; // 给数组a每个元素赋值。
    }
    return 0;
}
```

4.2 数组

4.2.3 一维数组的引用

对一维数组的引用可以通过**控制数组下标**来实现。数组元素是通过数组名和下标来区分的，带下标的数组元素也可以称为下标变量，下标变量可以像简单的变量一样参与各种运算。引用一维数组元素的格式如下：

数组名[下标表达式];

其中：

- 下标表达式必须是**整数**
- 不同于数组定义时用来确定数组长度的常量表达式，下标表达式的数据类型可以是整型常量或整型表达式，而且大多数情况下必须是整型常/变量及其表达式。

```
int n, a[50], m=5
```

```
a[2+6]=56;
```

✓

```
cin>>n; int a[n];
```

✗

```
a[0]=a[5]+a[2+1]-a[2*3];
```

✓

```
a[m]=20;
```

✓

- 当定义了一个长度为m的一维数组a时，C++规定数组下标从0开始，依次为0、1、2、3、.....n-1，对应的数组元素分别表示为a[0]、a[1]、.....a[n-1]。

4.2 数组

4.2.3 一维数组的引用

【例 2】假如有 n 个人，各人年龄不同，希望按年龄将他们从小到大排列

```
#include<iostream>
using namespace std;
int main()
{const int maxN=10;
int a[maxN];
int i,j,t;
cout<<"input 10 numbers :\n";
for (i=0;i<10;i++)
cin>>a[i];      //输入数组元素
cout<<endl;
```

4.2 数组

4.2.3 一维数组的引用

【例 2】假如有 n 个人，各人年龄不同，希望按年龄将他们从小到大排列(续)

```
for(j=0;j<9;j++)
    for(i=0;i<9-j;i++)    //从待排序序列中选择一个最大的数组元素
        if (a[i]>a[i+1])
            { t=a[i];    //交换数组元素
              a[i]=a[i+1];
              a[i+1]=t;}
cout<<"the sorted numbers :"<<endl;
for(i=0;i<10;i++)
    cout<<a[i]<<"\t";    //显示排序结果
cout<<endl;
return 0;
}
```

4.2 数组

4.2.4 一维数组的地址

数组名本身就是一个地址值，代表数组的首地址，所以数组元素的地址通过数组名来读取，其格式如下：

数组名+整数表达式

其中：

- 该表达式称为符号地址表达式，不代表实际的地址值

例如：一个浮点型的一维数组a, a[3]的符号地址表达式为a+3,它的实际地址为a+3*sizeof(float)

- 数组名是一个地址常量，不能作为左值（指在赋值表达式(assignment expression)中作为将要赋予值的地址的表达式，它必须是一个变量）
- 在使用数组的过程中最常犯的错误就是数组下标越界，通过数组的下标来得到数组内指定索引的元素，称作对数组的访问。数组访问越界不会造成编译错误，访问的是其他变量的内存。

4.2 数组

4.2.4 一维数组的地址

针对数组array, array和array[0]都是指第一个元素的地址, 使用时注意:

&array[0]+1 //加上一个元素的长度即第二个元素的地址

&array+1 //加上整个数组的长度

4.2 数组

4.2.4 一维数组的地址

&array[0]+1 //加上一个元素的长度即第二个元素的地址
&array+1 //加上整个数组的长度

```
#include<iostream>
using namespace std;
int main()
{
    int array[6] = { 10,12,13,14,15,16};
    cout << &array[1] << endl;
    cout << &array[0]+1 << endl;
    cout << &array[0]<< endl;
    return 0;
}
```

```
C:\Windows\system32\cmd.e
006FFDF0
006FFDF0
006FFDEC
请按任意键继续. . .
```

```
#include<iostream>
using namespace std;
int main()
{
    int array[6] = { 10,12,13,14,15,16};
    cout << &array[0] << endl;
    cout << &array+1<< endl;
    return 0;
}
```

```
C:\Windows\system32\cmd.
00AFF7AC
00AFF7C4
请按任意键继续. . .
```

4.2 数组

4.2.5 二维数组的定义

一维数组可以解决例如多个学生同一门成绩处理的问题，但如果一个班的学生有多门课程的成绩需要处理，此时一维数组就难以实现存取的过程，C++提供了**二维数组**来解决这样的问题。

二维数组和一维数组相比多了一个维度，二维数组定义的格式如下：

```
数据类型 数组名[常量表达式1][常量表达式2];
```

其中：

- 对数据类型、数组名以及常量表达式的要求和一维数组定义时的要求相同
- 常量表达式1代表第一维元素的个数，常量表达式2代表第二维元素的个数。如果将二维数组看成数学上的一个矩阵，则第一维元素的个数代表矩阵的行数，第二维元素的个数代表矩阵的列数，二维数组的定义格式可以改写为：

```
数据类型 数组名[行数][列数];
```

4.2 数组

4.2.5 二维数组的定义

数据类型 数组名[常量表达式1][常量表达式2];

- 如果把一维数组看成是由一个个相同类型的数据构成的排列，那么二维数组也可以看出是由一维数组构成的。例如二维数组 $a[m][n]$ ，则可以看成由 m 个一维数组构成，其中每个一维数组中有 n 个元素

4.2 数组

4.2.6 二维数组的使用

二维数组的初始化形式与一维数组类似：

数组类型 数组名[常量表达式1][常量表达式2]=初值表；

其中：初值表的定义形式有两种，即常数定义长度下的初始化（分行和不分行）。

（1）常数定义长度下的初始化----分行

以二维数组A[m][n]为例，初值表的格式为：

M的初值表={M[0]初值表,M[1]初值表,...,M[m-1]初值表}
int A[3][4]={ {1,2,3,4},{5,6,7,8},{9,10,11,12} }

（2）常数定义长度下的初始化----不分行

这种形式下的初值表与一维数组的初值表相同，初值表的项数不超过各维元素个数的乘积(总元素个数)。数组元素按内存排列顺序依次从初值表中取值，下列各数组不分行实现了初始化，结果与（1）中相同。

int A[3][4]={1,2,3,4,5,6,7,8,9,10,11,12}

4.2 数组

4.2.6 二维数组的使用

当使用常数定义长度下的初始化且不分行的情况下省略高维元素个数时，高维元素个数为：

向上取整数（初值表项数/低维元素个数）

例： `int b[][3]={1,2,3,4,0,0,0}` 高维元素个数为3

注意：不能先声明再赋全值。

`int [2][3]`

`int [2][3] = {{1,2,3},{4,5,6}}`

✗

4.2 数组

4.2.6 二维数组的引用

二维数组元素的引用格式如下：

数组名 [行下标表达式] [列下标表达式]

其中：

- 对数组名以及行、列下标表达式的要求和一维数组一致。
- 行、列下标的值同样是从0开始， $a[m][n]$ 表示数组的第 $m+1$ 行、第 $n+1$ 列的元素。
- 数组元素如果定义数组 $a[m][n]$ ，即数组第1维大小为 n ，第2维大小为 m 。 $a[i][j]$ 的排列位置与在内存中的地址计算公式如下：

$a[i][j]$ 的排列位置=第1维大小 $\times i + j + 1$;
 $a[i][j]$ 的地址= a 的起始地址+(第1维大小 $\times i + j) \times \text{sizeof}(\text{数据类型})$

4.2 数组

4.2.6 二维数组的引用

$a[i][j]$ 的排列位置=第1维大小 $n*i+j+1$;
 $a[i][j]$ 的地址= a 的起始地址+(第1维大小 $n*i+j$)*sizeof(数据类型)

例如：数组 $a[4][4]$

a_{00}	a_{01}	a_{02}	a_{03}
a_{10}	a_{11}	a_{12}	a_{13}
a_{20}	a_{21}	a_{22}	a_{23}
a_{30}	a_{31}	a_{32}	a_{33}

a_{21} 元素在数组中的位置是 $2*4+1+1=10$

4.2 数组

4.2.6 二维数组的引用

【例 2】有一个班5个学生，已知每个学生有5门课的成绩，要求输出平均成绩最高的学生的成绩以及该学生的序号

思路：

- 用二维数组，行代表学生，列代表一门课的成绩
- 要存放5个学生5门课的成绩和平均成绩，数组的大小应该是5×6

```
#include<iostream>
#include<stdio.h>
using namespace std;
int main()
{
    int i,j,max_i;
    float sum,max=0;
    float s[5][6]={ {78,82,93,74,65},{91,82,72,76,67},
                    {100,90,85,72,98},{67,89,90,65,78},
                    {77,88,99,45,89} };
}
```


4.2 数组

4.2.6 二维数组的引用

【例 2】 有一个班5个学生，已知每个学生有5门课的成绩，要求输出平均成绩最高的学生的成绩以及该学生的序号（续）

```
for (i=0;i<5;i++) //求出每个人的平均成绩，放在数组每一行的最后一列
{
    sum=0;
    for (j=0;j<5;j++)
        sum=sum+s[i][j];
    s[i][5]=sum/5;
}
for (i=0;i<5;i++) //找出最高的平均分和该学生的序号
    if (s[i][5]>max)
        {max=s[i][5];max_i=i;}
cout<<"stu_order ="<<max_i<<endl;
printf("max=%7.2f\n",max);
return 0;
}
```

4.2 数组

4.2.6 多维数组

使用二维数组可以实现存储一个班的学生们的多门成绩，但如果需要存储高中三年一个班的学生们的多门成绩则需要更高维的数组才能解决问题，三维以及高于三维的数组称为**多维数组**。

同样，三维数组**可以看成是由二维数组构成的**，三维数组是以二维数组为元素的数组，如果将一个二维数组看成是一张由行、列组成的表，三维数组则是由一张张表排成的一“本”表，第三维的下标为表的“页”码。同理，一个 $n(n \geq 3)$ 维数组是以一个 $n-1$ 维数组为元素的数组。只要内存够大，定义多少维的数组没有什么限制，但是一般情况下，使用二维数组就已经能够解决大部分的问题。

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];
```

4.2 数组

4.2.6 字符数组

数组的数据类型可以是除void型之外的任意数据类型，用于存放字符型数据的数组称为字符数组。字符数组和普通数组一样可以分为一维数组和多维数组。

用字符数组初始化：

```
char c1[10];  
c[0]='I'; c[1]=' '; c[2]='a'; c[3]='m'; c[4]=' '; c[5]='h'; c[6]='a'; c[7]='p'; c[8]='p'; c[9]='y';  
char c2[ ]={'g','d','l','g','x','y'};
```

4.2 数组

4.2.6 字符数组

用字符串进行初始化

在C++中对于字符串的处理可以通过字符数组实现，采用字符串对字符数组初始化的语法格式如下：

```
数据类型 数组名[常量表达式]={“字符串常量”};
```

其中：利用字符串初始化一维数组时，可以省略花括号{}。

```
char c3[ ]="china";  
char c4[ ][3]={ "I", "Love", "china" };
```

4.2 数组

4.2.6 字符数组

注意：

- 如果定义一个字符数组a[10],表示定义了10个字节的连续内存空间，**包括字符串的结尾“\0”**
- 如果字符串的长度小于数组的长度，则只将字符串中的字符给数组中前面的元素，剩下的内存空间系统会自动用“\0”填充

4.2 数组

4.2.6 字符数组

字符数组和一般数组一样，可以通过数组名以及下标的方式实现对字符数组元素的引用。为了方便对字符数组的处理，C++提供了许多专门处理字符和字符串的函数，如表1所示。

表 1 常用字符与字符串处理函数

函数形式	功能	头文件
gets(字符数组)	从终端输入一个字符串到字符数组	Cstring
puts(字符数组)	将一个字符串(以'\0'结束的字符序列)输出到终端	
strcat(字符数组1, 字符数组2)	连接两个字符数组中的字符串把字符串2接到字符串1的后面	
strcpy(字符数组1, 字符串2)	将字符串2复制到字符数组1中去	
strcmp(字符串1, 字符串2)	比较串1和串2。若串1=串2，则函数值为0;若串1>串2，则函数值为一个正整数;若串1<串2，则函数值为一个负整数	
strlen(字符数组)	测试字符串长度	
strlwr(字符串)	将字符串中大写字母换成小写字母	
strupr(字符串)	将字符串中小写字母换成大写字母	
toupper(字符串)	将小写字符转换成大写字符	Ctype
tolower(字符串)	将大写字符转换成小写字符	

4.2 数组

4.2.6 数组与函数

在将数组运用到函数的时候，先要了解数组的两个特殊性质。

(1)不允许拷贝和赋值

不能将数组的内容拷贝给其他数组作为其初始值，也不能用数组为其他数组赋值。

```
int a[] = {0,1,2}; // 含有三个整数的数组  
int s2[] = a;      // 错误：不允许使用一个数组初始化另一个数组  
a2 = a;           // 错误：不能把一个数组直接赋值给另一个数组
```


4.2 数组

4.2.6 数组与函数

(2)使用数组是通常将其转化成指针

在C++语言中，指针和数组有非常紧密的联系。使用数组的时候编译器一般会把它转换成指针。

通常情况下，使用取地址符来获取指向某个对象的指针，取地址符也可以用于任何对象。数组的元素也是对象，对数组使用下标运算符得到该数组指定位置的元素。因此像其他对象一样，对数组的元素使用取地址符就能得到指向该元素的指针：

```
string nums[] = {"one", "two", "three"}; // 数组元素是string对象  
string *p = &nums[0];                // p指向nums的第一个元素
```


4.3 指针

指针是C++的主要难点之一，也是C++语言最重要的特性之一。在计算机科学中，指针（pointer）是编程语言中的一个对象，利用地址，它的值直接指向（points to）存在电脑存储器中另一个地方的值。由于通过地址能找到所需的变量单元，可以说，地址指向该变量单元。正确的使用指针可以：

- 为函数提供修改调用变元的灵活手段
- 支持C++动态分配子程序
- 可以改善某些子程序的效率，在数据传递时，如果数据块较大（比如说数据缓冲区或比较大的结构），这时就可以使用指针传递地址而不是实际数据，即提高传输速度，又节省大量内存
- 为动态数据结构（如二叉树、链表）提供支持

4.3 指针

4.3.1 指针的定义与使用

定义变量之后会给每个变量分配内存空间，内存只不过是一个存放数据的空间，可以理解为装鸡蛋的篮子或者是装水果的箱子，现在我们把它想象成电影院的座位，电影院中的每个座位都要编号，而我们的内存要存放各种各样的数据，当然我们要知道我们的这些数据存放在什么位置，所以内存也要和座位一样进行编号了，这就是我们所说的内存编址（为内存进行地址编码）。座位可以是遵循“一个座位对应一个号码”的原则，从“第 1 号”开始编号。而内存则是按一个字节接着一个字节的次序进行编址。每个字节都有个编号，我们称之为内存地址。

由于人们并不关心实际地址，而是关心每个地址里面存放的值，所以一般利用变量名来存取该变量的内容。如果需要知道实际地址，我们可以使用取地址运算符来实现，例如，`&i` 代表是取 `i` 变量所在的地址编号。

4.3 指针

4.3.1 指针的定义与使用

地址又如何存放呢？其实指针就像是其它变量一样，所不同的是**一般的变量包含的是实际的真实的数据，而指针包含的是一个指向内存中某个位置的地址。**

其实生活中处处都有指针，我们也处处在使用它。有了它我们的生活才更加方便了。比如有天你说你要学习C++，要借用同学的一本C++参考资料，同学把书给你送过去发现你已经跑出去跑步了，于是同学把书放在了你桌子上书架的第二层三号的位置。并写了一张纸条：你要的书在第二层三号的书架上。当你回来时，看到这张纸条，你就知道了书放在哪了。纸条本身不是书，它上面也没有放着书。那么你又如何知道书的位置呢？因为纸条上写着书的位置，其实这张纸条就是一个指针了。它上面的内容不是书本身，而是书的地址，你通过纸条这个指针找到了我借给你的这本书。

注意啦！



指针变量和**指针**是两个概念，指针变量是存放地址的变量，而指针是一个地址

4.3 指针

4.3.1 指针的定义与使用

指针变量定义的一般形式为：

数据类型 *指针变量名；

例如：
`int * i;`
`int * j;`

其中：

- “*” 表示该变量的类型为指针类型。指针变量名为 i 和 j，而不是 *i 和 *j
- 数据类型是可以是基本数据类型，也可以是构造数据类型以及void类型
- 在定义指针变量时必须指定其数据类型。指针变量的数据类型是用来指定该指针变量可以指向的变量的类型。比如“int*i; ”表示 i 只可以指向 int 型变量。换句话说，数据类型就表示指针变量里面所存放的“变量的地址”所指向的变量可以是什么类型的。说得简单点就是：

以“int * i; ”为例，“*”表示这个变量是一个指针变量，而“int”表示这个变量只能存放 int 型变量的地址。

4.3 指针

4.3.1 指针的定义与使用

定义了指针之后，则得到了一个可以存放地址的指针变量。若在定义之后没有对指针赋值，那么此时系统会**随机给指针变量分配地址值**，随机的地址值则会指向不确定的内存空间，盲目的访问可能会对系统造成很大的危害。给指针变量赋值的形式有两种，如下：

1. 数据类型 *指针变量名=初始地址表达式;
2. 指针变量名 = 地址表达式;

注意啦！



第一种在定义变量的同时进行变量的赋值，初始地址值可以是地址常量、地址表达式、指针变量表达式。

第二种是在定义变量后，使用赋值语句给指针变量赋值。

4.3 指针

4.3.1 指针的定义与使用

对指针进行初始化时有以下几种常用方式：

- ✓ 采用NULL或空指针常量，如 `int *p=NULL;` 或 `char *P=2-2;` 或 `float *p=0`
- ✓ 取一个对象的地址然后赋给一个指针，如 `int i = 3; int *ip = &i`
- ✓ 取一个指针常量赋给一个指针，如 `long *p=(long *) 0xffffffff0`
- ✓ 将一个指针的地址赋给一个指针，如果 `int i=3; int *ip=&i; int **pp = &ip`
- ✓ 将一个字符串常量首地址赋给一个字符指针，如 `char *cp = "abcdefg"`

对指针进行初始化或赋值的实质是将一个地址或者同类型的指针赋给它，而不管这个地址是怎么取得的。

4.3 指针

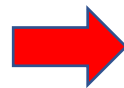
4.3.1 指针的定义与使用

在初始化指针时，“=”的右操作数必须为内存中数据的地址，不可以是变量，也不可以直接用整型地址值（`int * p=0`除外）。此时，`*p`只是表示定义的是个指针变量，并没有间接取值的意思。

例：

```
int *p;  
*p = 7;
```

X



```
int k;  
int *p;  
p = &k;  
*p = 7;
```

✓

`p`指向7所在的地址，`*p=7`给`p`所指向的内存赋值，`p`没有赋值，因而`p`所指的内存位置是随机的，没有初始化。

在初始化指针时，“=”的左操作数可以是`*p`，也可以是`p`。当为`*p`时，改变的是所指的地址存放的数据；当为`p`时，改变的是`p`所指向的地址。

4.3 指针

4.3.1 指针的定义与使用

指针是一个用数值表示的地址。因此，可以对指针进行四种算术运算：

++、--、+、-。

(1) 递增一个指针

由于数组是一个常量指针，不能递增。而变量指针可以递增，在程序中可以使用指针代替数组。

4.3 指针

4.3.1 指针的定义与使用

【例 1】演示指针加法运算

```
#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++; //或者写成prt=prt+1;
    }

    return 0; }
```

C:\Windows\system32\cmd.exe

```
Address of var[0] = 001EF8C8
Value of var[0] = 10
Address of var[1] = 001EF8CC
Value of var[1] = 100
Address of var[2] = 001EF8D0
Value of var[2] = 200
请按任意键继续. . .
```

4.3 指针

4.3.1 指针的定义与使用

(2) 递减一个指针

同样地，对指针进行递减运算，即把值减去其数据类型的字节数。

【例 2】演示指针减法运算

```
#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--; // 或者写成 ptr = ptr - 1;
    }
    return 0; }
```

C:\Windows\system32\cmd.exe

```
Address of var[3] = 0113F7A8
Value of var[3] = 200
Address of var[2] = 0113F7A4
Value of var[2] = 100
Address of var[1] = 0113F7A0
Value of var[1] = 10
请按任意键继续. . .
```

4.3 指针

4.3.1 指针的定义与使用

(3) 指针的比较

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

【例 3】演示指针比较

```
#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] ) //指针比较
    {
        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;}
```

C:\Windows\system32\cmd.exe

```
Address of var[0] = 00BAFE28
Value of var[0] = 10
Address of var[1] = 00BAFE2C
Value of var[1] = 100
Address of var[2] = 00BAFE30
Value of var[2] = 200
请按任意键继续. . .
```

4.3 指针

4.3.2 指针与字符串

指针变量的数据类型可以是包括void型的任意数据类型，其中字符型指针用于存放字符型变量的地址，而字符串可以以字符数组的形式在内存中存储。

(1) 字符串的表示形式

数组形式：

```
char string[] = "hello world";
```

字符指针形式：

```
char *str = "hello world";
```

4.3 指针

4.3.2 指针与字符串

数组形式与字符指针形式都是字符串的表示形式，但是这两种表示形式大不相同。下面以数组形式字符串`char string[] = "hello world";`与指针形式字符串`char *str = "hello world";`为例：

- 储存方式：（1）字符数组由若干元素组成，每个元素存放一个字符，（2）而字符指针变量只存放字符串的首地址，不是整个字符串。
- 存储位置：（1）数组是在内存中开辟了一段空间存放字符串；（2）而字符指针是在文字常量区开辟了一段空间存放字符串，将字符串的首地址付给指针变量`str`。
- 赋值方式

```
char str[10]; str = "hello";
```

✗

```
char *a; a = "hello";
```

✓

4.3 指针

4.3.2 指针与字符串

➤ 可否被修改

(1) 指针变量指向的字符串内容不能被修改，但指针变量的值是可以被修改的

```
char *p="hello"; //字符指针指向字符串常量
```

```
*p='a'; //错误，常量不能被修改，即指针变量指向的字符串内容不能被修改
```



```
char *p="hello"; //字符指针指向字符串常量
```

```
char ch='a'; p=&ch; //指针变量指向可以改变
```


4.3 指针

4.3.2 指针与字符串

➤ 可否被修改

(2) 字符串数组内容可以被修改，但字符串数组名所代表的字符串首地址不能被修改。

【例 4】 字符串数组内容可以被修改，但字符串数组名所代表的字符串首地址不能被修改

```
#include <iostream>
#include <Cstring>
using namespace std;
const int MAX = 3;
int main ()
{
    char str[10]=" ";
    cout<<"str addr is\t"<<&str<<","<<"str is\t"<<str<<endl;
    strcpy(str,"c++");
    cout<<"str addr is\t"<<&str<<","<<"str is\t"<<str<<endl;
    strcpy(str,"gdlgxy");
    cout<<"str addr is\t"<<&str<<","<<"str is\t"<<str<<endl;
    return 0;
}
```

C:\Windows\system32\cmd.exe

```
str addr is      012FF824, str is
str addr is      012FF824, str is c++
str addr is      012FF824, str is gdlgxy
请按任意键继续. . .
```

4.3 指针

4.3.3 指针与数组

➤ 使用指针操作符*存取数组

在C++中，由于数组是按照下标顺序连续在内存中存放的，而指针的加减运算的特点使指针操作特别符合处理存储在一段连续内存空间中的同类型数据。这样使用指针操作对数组及其元素来进行操作就十分方便。数组的数组名其实可以看作一个指针，就一维数组而言：

```
int array[10]={0,1,2,3,4,5,6,7,8,9},value;  
value=array[0]; //也可写成： value=*array;  
value=array[3]; //也可写成： value=*(array+3);  
value=array[4]; //也可写成： value=*(array+4);
```

其中，数组名array 代表数组本身，类型是int[10]，但如果把array 看做指针的话，它指向数组的第0个单元，类型是int* 所指向的类型是数组单元的类型即int。因此*array 等于0 。同理，array+3 是一个指向数组第3 个单元的指针，所以*(array+3)等于3。其它依此类推。也就是存取数组array[i]元素的等效方式为*(array+i) 。

4.3 指针

4.3.3 指针与数组

➤ 指针数组

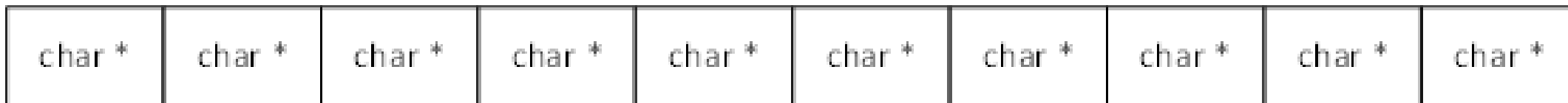
指针数组的本质是数组，数组中每一个成员都是同一类型的指针变量。定义一维指针数组的语法形式如下：

```
数据类型 *数组名[下标表达式];
```

例如：

```
char * pArray[10];
```

由于运算符[]的优先级高于*，pArray先与“[]”结合，构成一个数组的定义，char *修饰的是数组的内容，即数组的每个元素。该指针数组在内存中的如下图所示。



4.3 指针

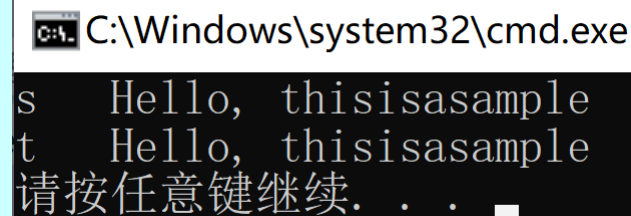
4.3.3 指针与数组

➤ 指针数组

指针数组常被用来存储若干个字符串，由于每个字符串的长度不一样，如果用二维字符数组存储将会浪费空间。

【例 5】指针数组演示

```
#include <iostream>
#include <Cstring>
using namespace std;
const int MAX = 3;
int main()
{
    char *str[3] = { "Hello, thisisasample", "Hi,goodmorning.", "helloworld" };
    char s[80], t[90];
    strcpy(s, str[0]);
    strcpy(t, *str);
    cout << "s  " << s << endl;
    cout << "t  " << t << endl;
    return 0;
}
```



4.3 指针

4.3.3 指针与数组

➤ 数组指针

数组指针也称行指针，它本质上是一个指针，用来指向数组。定义一维数组指针的语法形式如下：

```
数据类型 (*指针名) [下标表达式];
```

例如：

```
int (*p)[n];
```

虽然运算符[]的优先级高于*，但是()优先级高，首先说明p是一个指针，指向一个整型的一维数组，这个一维数组的长度是n，也可以说是p的步长。也就是说执行p+1时，p要跨过n个整型数据的长度。

4.3 指针

4.3.3 指针与数组

➤ 数组指针

```
int array[10];  
int (*ptr)[10];  
ptr=&array;  
  
sizeof(int(*)[10])==4  
sizeof(int[10])==40  
sizeof(ptr)==4
```

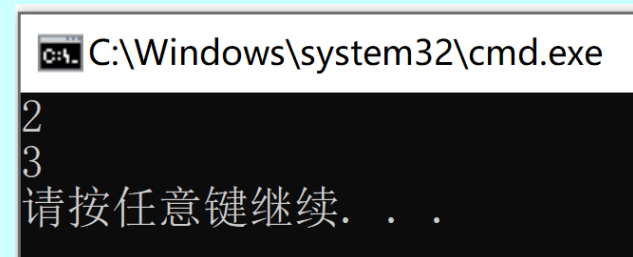
其中ptr是一个指针，它的类型是int(*)[10],指向的类型为int[10]，用整个数组的首地址进行进行初始化。sizeof(指针名称)的结果是指针自身类型的大小。

4.3 指针

4.3.3 指针与数组

【例 6】指针数组和数组指针的区别

```
#include <iostream>
using namespace std;
int main()
{   int c[4]={1,2,3,4};
    int *a[4];           //指针数组
    int (*b)[4];         //数组指针
    b=&c;
    for(int i=0;i<4;i++) //将数组c中元素赋给数组a
    {
        a[i]=&c[i];
    }
    cout<<*a[1]<<endl;
    cout<<(*b)[2]<<endl;
    return 0;}
```



```
C:\Windows\system32\cmd.exe
2
3
请按任意键继续. . .
```

4.3 指针

4.3.3 指针常量和常量指针

指针可以分为**指针常量**和**常量指针**，前者的操作对象是指针值（即地址值），是对指针本身的修饰，后者的操作对象是通过指针间接访问的变量的值，是对被指向对象的修饰。

指针常量本质是一个常量，而用指针修饰它。指针常量的值是指针，这个值因为是常量，所以不能被赋值。在C/C++中，指针常量声明格式如下：

```
数据类型 *const 指针名=变量名;
```

只要const位于指针声明操作符右侧，就表明声明的对象是一个常量，且它的内容是一个指针，即一个地址。指针常量在声明时就要赋初始值。一旦赋值，以后这个常量再也不能指向别的地址。

例：

```
int a;  
int *const b=&a;
```

4.3 指针

4.3.3 指针常量和常量指针

虽然指针常量的值不能变，但是它所指向的对象是可以变化的，因为并没有限制它指向的对象是常量。

例：

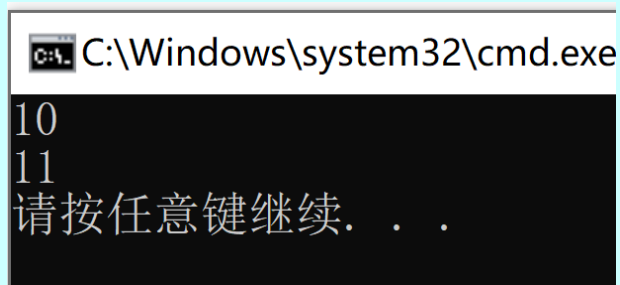
```
char *a="abcd1234";  
char *b="bcde";  
char *const c=&a  
  
a[0]='x'; 等价于  *c[0]='x';
```

4.3 指针

4.3.3 指针常量和常量指针

【例 6】指针常量的演示

```
#include <iostream>
using namespace std;
int main() {
    int i = 10;
    int *const p = &i;
    cout<< *p<<endl;
    //p++;           //Error,因为p是const 指针, 因此不能改变p指向的内容
    (*p)++;          //OK,指针是常量, 指向的地址不可以变化,但是指向的地址所对应的内容可以变化
    cout<< *p<<endl;
    return 0;
}
```



```
C:\Windows\system32\cmd.exe
10
11
请按任意键继续. . .
```


4.3 指针

4.3.3 指针常量和常量指针

常量指针本质是指针，表示这个指针是一个指向常量的指针（变量）。指针指向的对象是常量，那么这个对象不能被更改。其语法格式如下：

```
const 数据类型 * 指针名=变量名;    const int *p;  
  
或  
  
数据类型 const * 指针名=变量名;    int const *p;
```

常量指针指向的对象不能通过这个指针来修改，但是可以通过原来的声明修改，即常量指针可以被赋值为变量的地址。限制通过这个指针修改变量的值，即是常量指针之意。

4.3 指针

4.3.3 指针常量和常量指针

【例 7】常量指针的演示

```
#include <iostream>
using namespace std;
int main() {
    int i = 10;
    int i2 = 11;
    const int *p = &i;
    cout<< *p<<endl;
    i = 9;
    // *p = 11;
    p = &i2;
    cout<< *p<<endl; // 11
    return 0; }
```

//正确仍然可以通过原来的声明修改值,
//错误,*p是const int的,不可修改,即常量指针不可修改其指向内容
//OK,指针还可以指向别处,因为指针只是个变量,可以随意指向;

4.3 指针

4.3.4 指向常量的指针常量

指向常量的**指针常量**就是一个**常量**，且它**指向的对象也是一个常量**。因为是一个指针常量，那么它指向的对象当然是一个指针对象，而它又指向常量，说明它指向的对象不能变化，故指向常量的指针常量简称为**常指针常量**，其定义格式如下：

```
const 数据类型 * const 指针名=变量名;
```

或

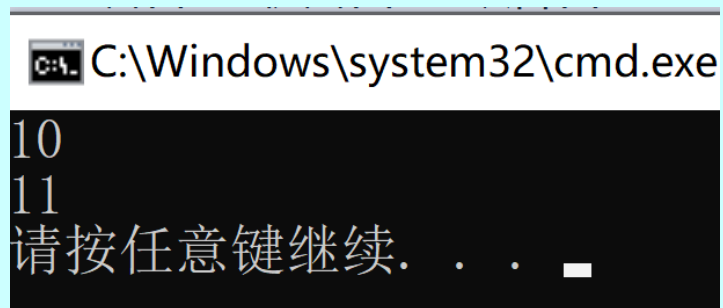
```
数据类型 const * const 指针名=变量名;
```

4.3 指针

4.3.4 指向常量的指针常量

【例 8】常量指针的演示

```
#include <iostream>
using namespace std;
int main() {
    int i = 10;
    const int *const p = &i; //p为常指针常量
    printf("%d\n", *p);
    //p++;    //错误! 编译器报错提示: increment of read-only variable 'p'
    //(*p)++; //错误! 编译器报错提示: increment of read-only location '*p'
    i++;      //正确,仍然可以通过原来的声明修改值
    cout<<*p<<endl;
    return 0;
}
```



```
C:\Windows\system32\cmd.exe
10
11
请按任意键继续. . .
```

4.3 指针

4.3.5 指针与函数

1、指针作为函数参数

C++中有两种主要的**数据传递方式**，分别是**值传递方式**和**地址传递方式**。如果希望在另外一个函数中修改本函数中变量的值，那么在调用函数时只能传递该变量的地址。在C++中，函数的参数不仅可以是基本类型的变量、对象名和数组名等，也可以是指针。

在实际编程中都是传递指针，只在以下两个条件下才会直接传递数据，且两个条件缺一不可：

- 数据很小，比如一个int型变量
- 不需要修改它的值，只是使用它的值

此外需要注意的是，数组名本身就是地址，所以如果传递数组的话直接传递数组名即可。接收的形参可以定义成数组也可以定义为同类型的指针。另外，指针能使被调函数返回一个以上的结果。

4.3 指针

4.3.5 指针与函数

1、指针作为函数参数

【例 10】用值传递的形式对两个数据进行交换

```
#include <iostream>
using namespace std;
int Swap(int a, int b); //函数声明
int main(void)
{   int i = 3, j = 5;
    cout<<"交换前i="<<i<<"j="<<j<<endl;
    Swap(i, j);
    cout<<"交换后i="<<i<<"j="<<j<<endl;
    return 0;}
int Swap(int a, int b)
{   int buf;
    buf = a;
    a = b;
    b = buf;
    return 0; }
```

?

4.3 指针

4.3.5 指针与函数

1、指针作为函数参数

【例 11】用地址传递的形式对两个数据进行交换

```
#include <iostream>
using namespace std;
int Swap(int *a, int *b); //函数声明
int main(void)
{   int i = 3, j = 5;
    cout<<"交换前i="<<i<<"j="<<j<<endl;
    Swap(&i, &j);
    cout<<"交换后i="<<i<<"j="<<j<<endl;
    return 0;}
int Swap(int *a, int *b)
{   int buf;
    buf = *a;
    *a = *b;
    *b = buf;
    return 0; }
```

?

4.3 指针

4.3.5 指针与函数

2、指针作为函数返回值

和别的数据类型一样，**指针也能够作为函数的一种返回值类型**。我们把返回指针的函数称为**指针函数**。使用指针函数和普通数据类型作为返回值相比最大的优点就是**前者可以实现在函数调用结束时把大量的数据返回给主调函数**，后者只能在调用结束后返回一个值。

4.3 指针

4.3.5 指针与函数

2、指针作为函数返回值

【例 12】用指针函数对数组元素进行输出

```
#include<iostream>
using namespace std;
int* f2(int a[],int i);
int main()
{
    int a[] = { 1,2,3,4,5};
    //f2函数返回的是一个指针，需要解引用取内容
    cout<<*f2(a,2)<<endl;    //3
        int *n = f2(a,2);
    cout<<*n<<endl;    //3
    *f2(a,3) = 14;    //相当于a[3];
    for(int i=0;i<5;i++)
        cout<<a[i]<<" ";    //1 2 3 14 5
    cout<<endl;
    return 0;}
int* f2(int a[],int i)
{
    return &a[i]; }
```

4.3 指针

4.3.5 指针与函数

3、指向函数的指针

指向函数的指针又可称为函数指针。函数具有可赋值给指针的物理内存地址，一个函数的函数名就是一个指针，它指向函数的代码。**一个函数的地址是该函数的进入点**，也是调用函数的地址。函数的调用可以通过函数名，也可以通过指向函数的指针来调用。函数指针还允许将函数作为变元传递给其他函数。不带括号和变量列表的函数名，可以表示函数的地址，正如不带下标的数组名可以表示数组的首地址。定义形式如下：

```
类型 (*指针变量名) (参数列表) ;
```

其中：

类型为函数指针所指函数的返回值类型；参数列表则列出了该指针所指函数的形参类型和个数。

4.3 指针

4.3.5 指针与函数

3、指向函数的指针

例：

```
int (*p) (int i, int j);
```

P是一个指针，指向一个函数，该函数有2个整型参数，返回类型为int

函数指针和其他变量一样，在使用之前需要进行赋值，函数指针所指向的函数必须已经被定义，且指向的是函数代码的起始地址，其语法形式如下：

```
函数指针名 = 函数名;
```

在赋值之后，就可以通过函数指针直接引用该指针所指向的函数，即该函数指针可以和函数名一样出现在函数名能出现的任何地方。

调用函数指针所指向的函数有以下两种形式：

- (1) 函数名 (实参表)
- (2) (*函数指针名) (实参表) ;

4.3 指针

4.3.5 指针与函数

3、指向函数的指针

【例 13】用函数指针实现两个数的比较

```
#include <iostream>
using namespace std;
#define GET_MAX      0
#define GET_MIN      1
int get_max(int i,int j)
{
    return i>j?i:j; }
int get_min(int i,int j)
{
    return i>j?j:i; }
int compare(int i,int j,int flag)
{
    int ret;
    int (*p)(int,int);
    //这里定义了一个函数指针，就可以根据传入的flag，灵活地决定其是指向求大数或求小数的函数
    //便于方便灵活地调用各类函数
```

4.3 指针

4.3.5 指针与函数

3、指向函数的指针

【例 13】用函数指针实现两个数的比较(续)

```
    if(flag == GET_MAX)
        p = get_max;
    else
        p = get_min;
    ret = p(i,j);
    return ret;}

int main()
{
    int i = 5,j = 10,ret;
    ret = compare(i,j,GET_MAX);
    cout<<"The MAX is"<<ret<<endl;
    ret = compare(i,j,GET_MIN);
    cout<<"The MIN is"<<ret<<endl;
    return 0 ;}
```

4.4 引用

4.4.1 引用的定义

引用可以理解成为对象起了另外一个名字，引用了另外一种类型。引用和指针一样可以间接的对变量进行访问，但是引用在使用上比指针更安全。定义引用时，程序把引用和它的初始值绑定在一起，而不是将初始值拷贝给引用。因为无法令引用重新绑定到另外一个对象，因此引用必须初始化。定义一个引用型变量的语法格式如下：

```
数据类型 &引用变量名 = 变量名;
```

其中：

- 引用只能在初始化的时候引用一次，不能改变再引用其他的变量
- 变量名为已经定义的变量
- 数据类型必须和引用变量的类型相同

4.4 引用

4.4.1 引用的定义

例：

```
int a=1;  
int &b = a; //b指向a  
int &b; // 报错，引用必须被初始化
```

b是一个引用型变量，它被初始化为对整型变量a的引用，此时系统没有给b分配内存空间，b与被引用变量a具有相同的地址，即两个变量使用的是同一个内存空间，修改两者其中的任意一个的值，另一个值也会随之发生变化。

```
a=5;  
cout<<b; // 输出5  
b=10;  
cout<<a; // 输出10
```

4.4 引用

4.4.1 引用的定义

指针与引用的区别：

1、指针是一个变量，存储的是一个地址，指向内存中一个单元；引用与原来的变量实质上相同，只是原变量的一个别名

例：

```
int a=1, *p=&a;  
int a=1, &b=a;
```

2、指针可以有多级，但是引用只能是一级

例：

```
**p 合法， &&a不合法
```

4.4 引用

4.4.1 引用的定义

指针与引用的区别：

- 3、指针的值可以为NULL,但是引用的值不能为NULL,并且引用在定义时必须初始化
- 4、指针的值在初始化后可以改变, 指向其他的存储单元, 但是引用在进行初始化后不能再改变
- 5、“sizeof引用”得到的是所指向的变量（对象）的大小, 而“sizeof指针”得到的是指针本身的大小
- 6、指针和引用的自增(++)运算意义不一样

4.4 引用

4.4.2 常引用

用`const`声明的引用就是常引用。常引用所引用的对象不能被更改。常引用经常作为函数的形参，防止对实参的误修改。常引用的声明形式为：

```
const 类型说明符 &引用名;
```

4.4 引用

4.4.2 常引用

【例 14】常引用作为函数形参

```
#include <iostream>
using namespace std;
void fun(const double &d); //常引用作为函数参数
int main(){
    double d = 3.14;
    fun(d);
    return 0;
}
void fun(const double &d){
    // 常引用作形参，在函数中不能更新d所引用的对象
    double i = 6.66;
    // d = i; 此处将报错!!!
    cout << "d = " << d << endl;
}
```

4.4 引用

4.4.3 引用与函数

1、引用作为函数参数

当引用作为函数参数进行传递时，实质上传递的是实参本身，即传递进来的不是实参的一个拷贝，因此对形参的修改其实是对实参的修改，所以在用引用进行参数传递时，不仅节约时间，而且可以节约空间。

比如：

```
#include<iostream>
using namespace std;
void test(int &b)
{
    cout<<&b<<" "<<b<<endl;
}
int main(void)
{
    int a=1;
    cout<<&a<<" "<<a<<endl;
    test(a);
    return 0;
}
```

4.4 引用

4.4.3 引用与函数

2、引用作为函数的返回值

函数返回值为引用型的语法形式为：

类型 &函数名（形参列表）{ 函数体 }

其中：

- 以引用返回函数值，定义函数时需要在函数名前加 &。
- 用引用返回一个函数值的最大好处是，在内存中不产生被返回值的副本。

4.4 引用

4.4.3 引用与函数

2、引用作为函数的返回值

- 不能返回局部变量的引用。主要原因是局部变量会在函数返回后被销毁，因此被返回的引用就成为了"无所指"的引用，程序会进入未知状态。

注意啦！



```
int &func()
{
    int q;
    return q; //编译时错误
}
```

- 不能返回函数内部new分配的内存的引用。虽然不存在局部变量的被动销毁问题，可对于这种情况（返回函数内部new分配内存的引用），又面临其它问题。例如，被函数返回的引用只是作为一个临时变量出现，而没有被赋予一个实际的变量，那么这个引用所指向的空间（由new分配）就无法释放。
- 可以返回类成员的引用，但最好是const。

4.4 引用

4.4.3 引用与函数

2、引用作为函数的返回值

【例 15】 引用作为函数的返回值实现数组元素的重新赋值

```
#include <iostream>
using namespace std;
double a[] = { 10.1, 12.6, 33.1, 24.1, 50.0 };
double& set( int i )
{ return a[i]; }           // 返回第 i 个元素的引用 }
int main ()
{   cout << "改变前的值" << endl;
    for ( int i = 0; i < 5; i++ )
    {   cout << "a[" << i << "] = ";
        cout << a[i] << endl; }
    set(1) = 20.23;           // 改变第 2 个元素
    set(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 << "a[" << i << "] = ";
        cout << a[i] << endl; }
    return 0; }
```

4.5 枚举

4.5.1 枚举的定义

在现实生活中，有些情况只能取有限几个可能的值，比如一个星期有7天，分别是星期一、星期二……星期日；一年有12个月，一月、二月……十二月。类似这样的情况例子我们可以在计算机中表示成int、char等类型的数据，但是这样又很容易和不表示星期或者月份的整数混淆。C++中的枚举类型就是专门来解决这类问题的数据类型。

4.5 枚举

4.5.1 枚举的定义

如果一个变量只有几种可能的值，可以定义为枚举(enumeration)类型。枚举类型是 C++ 中的一种派生数据类型，它是由用户定义的若干枚举常量的集合。所谓“枚举”是指将变量的值一一列举出来，变量的值只能在列举出来的值的范围内。声明枚举类型用enum开头，其定义的一般形式如下：

```
enum 枚举类型名 {枚举常量1, 枚举常量2, .....枚举常量n};
```

4.5 枚举

4.5.1 枚举的定义

```
enum Weekday{SUN, MON, TUE, MED, THU, FRI, SAI};
```

其中：

- 关键字enum—指明其后的标识符是一个枚举类型的名字
- 枚举常量表—由枚举常量构成。"枚举常量"或称"枚举成员"，是以标识符形式表示的整型量，表示枚举类型的取值。枚举常量表列出枚举类型的所有取值，各枚举常量之间以"，"间隔，且必须各不相同。取值类型与条件表达式相同
- 枚举元素是常量，不能对它们赋值
- 枚举常量代表该枚举类型的变量可能取的值，编译系统为每个枚举常量指定一个整数值，默认状态下，这个整数就是所列举元素的序号，序号从0开始

4.5 枚举

4.5.2 枚举变量的使用

定义枚举类型的主要目的是：增加程序的可读性。枚举类型最常见也最有意义的用处之一就是用来描述状态量。可以类型与变量同时定义（甚至可以省去类型名），格式如下：

```
enum {Sun,Mon,Tue,Wed,Thu,Fri,Sat} weekday1, weekday2;
```

其中：

- 枚举变量的值只能取枚举常量表中所列的值，就是整型数的一个子集
- 枚举变量占用内存的大小与整型数相同
- 枚举变量只能参与赋值和关系运算以及输出操作，参与运算时用其本身的整数值

4.5 枚举

4.5.2 枚举变量的使用

```
enum color{RED, BLUE, WHITE,BLACK, GREEN} color 1, color 2, color3, color4;
```

允许以下操作：

```
color3=RED;    //枚举常量值赋给枚举变量  
color4=color3; //相同类型的枚举变量赋值  
int i=color3;   //枚举变量值赋给整型变量, i=0  
int j=GREEN;    //枚举常量值赋给整型变量, j=4
```

允许的运算关系有==, <, >, <=, >=, != 等

4.5 枚举

4.5.2 枚举变量的使用

注意啦！



- 枚举变量可以直接输出，但不能直接输入。如： `cin >> color3;` //非法
- 不能直接将常量赋给枚举变量。如： `color1=1;` //非法
- 不同类型的枚举变量之间不能相互赋值。如： `color1=color3;` //非法
- 枚举变量的输入输出一般都采用switch语句将其转换为字符或字符串；枚举类型数据的其他处理也往往应用switch语句，以保证程序的合法性和可读性
- 枚举常量只能以标识符形式表示，而不能是整型，字符型等文字常量，**以下定义是非法的：**

```
enum letter_set{'a', 'd', 'f'};  
enum year_set{2000, 1000, 10};
```

4.6 结构体与联合

4.6.1 结构体的定义

聚合数据类型能够同时存储超过一个的单独数据。C++提供了两种类型的聚合数据类型，分别是数组和结构体。数组是相同元素的集合，它的每个元素是通过下标引用或指针间接访问的。结构体也是一些值的集合，这些值称为它的成员，但一个结构的成员可能具有不同的类型。数组元素可以通过下标访问，这是因为数组元素长度相同，但在结构体中并非如此，由于每个成员的类型可能不同，那么长度也就可能不同，所以就不能通过下标来访问。但是结构体成员都有自己的名字，他们是通过名字访问的。另外，结构体在表达式中使用时，不能被替换为指针。结构体变量也无法使用下标来选择特定的成员。C++的结构体可以包含函数，这样，C++的结构体也具有类的功能，与class不同的是，结构体包含的函数默认为public，而不是private。

4.6 结构体与联合

4.6.1 结构体的定义

结构体类型的定义方式如下：

```
struct 结构体类型名  
{ 数据类型1 成员名1;  
  数据类型2 成员名2;  
  ....  
  数据类型n 成员名n;  
}
```

其中：

- struct是关键字，表示定义的是一个结构体类型
- 结构体类型名和成员名必须是一个合法的标识符
- 结构体类型成员不限数量，数据类型可以是基本数据类型，也可以是用户自定义数据类型

4.6 结构体与联合

4.6.1 结构体的定义

例：

```
struct A  
{ int a;  
  char ch;  
}
```

定义一个结构体A，该结构体包含两个成员：整型a、字符ch
结构体名A可以省略，作为匿名结构体类型，但一般不建议。

4.6 结构体与联合

4.6.2 结构体变量的定义和使用

结构体变量定义与其他类型的定义格式相同，既可以和结构体类型一起定义，也可以先定义结构体类型，在需要使用的时候再定义结构体变量。

结构体变量的初始化方法与数组的初始化方法相似，格式如下：

```
结构体类型名 结构体变量名={成员名1的值, 成员名2的值, ..., 成员名n的值};
```

其中：

- 初值表位于一对花括号内部，每个成员值用逗号进行分隔。
- 初始化结构变量时，成员值表中的顺序要和定义时的顺序相同。
- 只有在定义结构变量时才能对结构变量进行整体初始化，在定义了结构体变量后每个成员只能单独初始化。
- 在定义结构类型与变量时不能对成员进行初始化。

4.6 结构体与联合

4.6.2 结构体变量的定义和使用

访问结构体成员的两种方式：

1、通过“.”操作符访问成员，格式如下：

结构体变量名.成员名;



```
struct A
{
    int x;
    char a[10];
} Stu;
int main()
{
    strcpy(Stu.a, "jiegouti");
    Stu.x = 10;
    cout << Stu.x << " " << Stu.a << endl;
    return 0;
}
```

4.6 结构体与联合

4.6.2 结构体变量的定义和使用

访问结构体成员的两种方式：

2、通过“->”操作符访问成员，格式如下：

指向结构体变量的指针名->结构体变量名的成员名；



```
struct A
{
    int x;
    char a[10];
} Stu;
// struct A Stu;
int main()
{
    struct A *p;    //定义一个结构体A的指针变量p
    p = &Stu;
    strcpy(p->a, "jiegouti");
    p->x = 20;
    cout << p->x << " " << p->a << endl;
    return 0;
}
```


4.6 结构体与联合

4.6.3 联合的定义

联合（union）是一种特殊的类，也是一种构造类型的数据结构，也称为**共用体类型**。在一个联合类型内可以定义多种不同的数据类型，一个被说明为该“联合”类型的变量中，允许装入该联合类型所定义的任何一种数据，**这些数据共享同一段内存**，达到节省空间的目的，定义联合体的语法形式如下：

```
union 联合类型名
{ 数据类型1 成员名1;
  数据类型2 成员名2;
  .....
  数据类型n 成员名n;
}
```

4.6 结构体与联合

4.6.3 联合的定义

联合类型和结构体有一些相似，但是二者有本质上的不同。在结构体中各成员有各自的内存空间，**一个结构变量的总长度是各成员长度之和**（空结构除外，同时不考虑边界调整）。在联合类型中，各成员共享一段内存空间，**一个联合变量的长度等于各成员中最长的长度**。

注意啦！



- 联合类型的共享并不是把多个成员装入一个联合变量中，而是指该联合变量可以赋予任一成员值，但每次只能赋一种值，赋入新值则覆盖旧值
- 结构变量可以作为函数参数，函数也可以返回指向结构的指针变量；而联合变量不能作为函数参数，函数参数也不能返回指向联合的指针变量，但函数可以使用指向联合变量的指针，也可以使用联合数组

4.6 结构体和联合

例1:

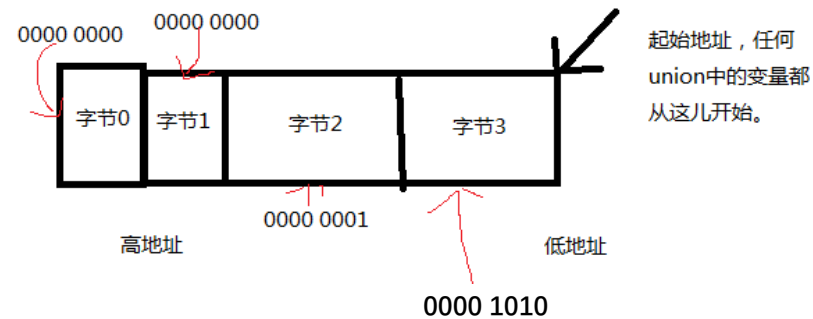
```
union my
{
    struct
    {
        int x;
        int y;
        int z;
    } u;
    int k;
} a;

int main()
{
    union my *p;
    p=&a;
    a.u.x = 4;
    a.u.y = 5;
    a.u.z = 6;
    p->k = 0;
    cout << a.u.x << a.u.y << a.u.z << p->k << endl;
}
```

例2:

```
union {
    int i;
    char x[2];
} a;

int main(void)
{
    a.x[0] = 10;
    a.x[1] = 1;
    printf("%d\n",a.i);
    return 0;
}
```



4.7 补充内容

4.7.1 考虑边界调整的结构体内存

边界调整(字节对齐)原则:

- 1、结构体中的元素是按照顺序一个个放入内存，而且每个元素放入的首地址（相对地址）一定是自己本身长度的整数倍

```
struct X{  
    char a;  
    int b;  
    double c;  
}SRT;  
int main()  
{  
    cout << sizeof(SRT) << endl;  
    cout << sizeof(SRT.a) << endl;  
    cout << sizeof(SRT.b) << endl;  
    cout << sizeof(SRT.c) << endl;  
    return 1;  
}
```

输出为 sizeof(SRT)=16

sizeof(SRT.a)=1 sizeof(SRT.b)=4

sizeof(SRT.c)=8

a占了0 ~ 3 b占4 ~ 7 c占8 ~ 15

0 4 8 分别是 char int double 的整数倍位置

4.7 补充内容

4.7.1 考虑边界调整的结构体内存

边界调整(字节对齐)原则:

2、结构体所占内存为最大数据所占内存的整数倍

```
struct X{
    char a;
    double b;
    int c;
}SRT;
int main()
{
    cout << sizeof(SRT) << endl;
    cout << sizeof(SRT.a) << endl;
    cout << sizeof(SRT.b) << endl;
    cout << sizeof(SRT.c) << endl;
    return 1;
}
```

sizeof(SRT)=24 char 放在了0 ~ 7 double
在8 ~ 15 int在16 ~ 19

20 ~ 23为对齐补全

4.8 作业

1、编写程序定义一个结构体数据类型并说明一个结构体数据类型的数组，通过指针变量输出该数组中各元素的值，要求输出结果如下所示：

学号 姓名 数学 C++

1 张三 87 98

2 李四 67 82

3 王刚 54 60

4 刘丽 100 90

5 陈军 88 95

2、编写一个程序，寻找二维数组中每一行的最大值。

3、从输入的一行字符串中求出最长英文单词长度及最长单词个数，并输出长度和个数，单词之间只能用一个或多个空格隔开。如输入字符串” I am a student” 时，最长单词的长度为7，个数为1。而输入字符串” word body book try” 时，最长单词的长度为4，个数为3，即有三个单词均为最长单词。

4、输入3个整数a、b、c，按从大到小的顺序输出。要求通过指针实现