第四讲 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。

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。

定义的一般形式如下：

```
数据类型 数组名[常量表达式];
```

➢ 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

一维数组的使用

一维数组可以被显示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};
```

- 数组可以被显示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

  利用 for 对数组进行读入操作：

```
#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;
}
```

一维数组的引用

范式：`数组名[下标表达式 ];`

冒泡的实现：

```
#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;
    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;
}
// 结果:
// input 10 numbers :
// 9 8 7 6 5 4 3 2 1 0
//
// the sorted numbers :
// 0       1       2       3       4       5       6       7       8
9
```

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 + 正数表达式`

- 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。

```
&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
```

辨析：

```
#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; // &array[1] == &array[0] + 1
    cout << &array[0] << endl;
    return 0;
}
// 结果:
// 0x7ffc29f1b2d4
// 0x7ffc29f1b2d4
// 0x7ffc29f1b2d0
```

```
#include <iostream>
using namespace std;
int main()
{
    int array[6] = {10, 12, 13, 14, 15, 16};
    cout << &array[0] << endl;
    cout << &array[0] + 1 << endl;
    cout << &array + 1 << endl;
    return 0;
}
// 结果:
// 0x7fff202dd0a0
// 0x7fff202dd0a4
// 0x7fff202dd0b8
```

二维数组简介

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

```
#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}};
    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;
}
// 结果:
// stu_order =2
// max=  89.00
```

多维数组

使用二维数组可以实现存储一个班的学生的多门成绩，但如果需要存储高中三年一个班的学生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>=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];`

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。

初始化方式

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'};
```

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"};

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

- 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"
  - 这个结尾一定是存在的，通常在定长字符数组里（char[300]）寻找结尾时很常用，循环停止条件就是当前 `char != '\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

为了更加方便地处理字符数组，C++内置了一些库函数来方便这一过程

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	

Tips：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的搭配指向该元素的指针：

```
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
```

指针

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。

- 有时常把指针变量、地址、地址变量统称为指针
  - 但实际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

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`

- “*”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。

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。

`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;`

初始化的几种方式:

- NULL 或空指针常量：`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 *) 0xfffffff0`
- 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`
- 将一个字符串常量赋给一个字符指针，如 `char *cp = “abcdefg”`

指针也可以进行算术运算四种算术运算，分别是：`++,--,+,-`

```
#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;
}
// 结果:
// Address of var[0] = 0x7ffee4ebffec
// Value of var[0] = 10
// Address of var[1] = 0x7ffee4ebfff0
// Value of var[1] = 100
// Address of var[2] = 0x7ffee4ebfff4
// Value of var[2] = 200
```

同样，也可以进行大小的比较，搭配循环语句来使用：

```
#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;
}
// 结果:
// Address of var[0] = 0x7fff66e3540c
// Value of var[0] = 10
// Address of var[1] = 0x7fff66e35410
// Value of var[1] = 100
// Address of var[2] = 0x7fff66e35414
// Value of var[2] = 200
```

指针与字符串

表示形式

```
// 数组形式
char string[] = "hello world;
// 字符指针形式
char *str = "hello world"
```

不同表示形式下的差异

存储方式

- 字符数组由若干元素组成，每个元素存放一个字符
- 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

- 数组是在内存中开辟了一段空间存放字符串
- 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"` ✅

能否被修改

指针变量指向的字符串内容不能被修改（但可以改变其指向的地址），但指针变量的值是可以被修改的

```
// 字符指针
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

// 字符数组
#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++");     // 执行字符串拷贝
    cout << "str addr is\t" << &str << ","
         << "str is\t" << str << endl;
    strcpy(str, "gdlgxy");  // 执行字符串拷贝
    cout << "str addr is\t" << &str << ","
         << "str is\t" << str << endl;
    return 0;
}
// 结果:
// str addr is     0x7ffff00cd69e,str is
// str addr is     0x7ffff00cd69e,str is   c++
// str addr is     0x7ffff00cd69e,str is   gdlgxy
```

指针与数组

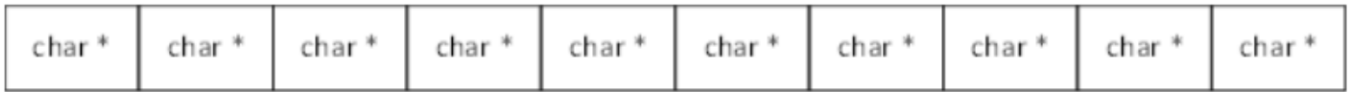
指针数组

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。



指针数组常被用来存储若干个字符串，因为字符串的不定长性，使用二维数组会造成浪费情况。

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

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。

`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

```
#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;
}
// 结果:
// 2
// 3
```

指针常量与常量指针

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`

```
#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;
}
// 结果:
// 10
// 11
```

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

```
#include <iostream>
using namespace std;
int main()
{
    int i = 10;
    int i2 = 11;
    const int *p = &i;
    cout << *p << endl;
    i = 9;              // 正确 仍然可以通过原来的声明修改值，
    //*p = 11;          //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
    p = &i2;            // 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;
    cout << *p << endl; // 11
    return 0;
}
// 结果:
// 10
// 11
```

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;`

```
#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;
}
// 结果：
// 10
// 11
```

指针与函数

指向函数的指针

```
#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
    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;
}
// 结果:
// The MAX is 10
// The MIN is 5
```

引用

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;`

- 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

```
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。

指针与引用的区别

- 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
- 指针可以有多级，但是引用只能是一级 **p 合法，&&a 不合法
- 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
- 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
- “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
- 指针和引用的自增(++)运算意义不一样

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;`

```
#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;
}
```

引用与函数

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。

```
#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;
}
// 结果:
// 0x7ffcee4815d4 1
// 0x7ffcee4815d4 1
```

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 }`

➢ 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。

➢ 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。

```
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。

```
#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;
    }
    // 我超！好骚的用法
    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;
}
```

枚举

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。类似这样的情况的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。

枚举的定义

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

➢ 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（实质上还是整形），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

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;`

➢ 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

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

➢ 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’}; //非法定义 ❌
enum year_set{2000, 1000, 10};  //非法定义 ❌
```

结构体与联合

结构体的定义

```
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;
}
```

➢ 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

结构体变量的初始化

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};`

➢ 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。

```
struct StuStruct
{
    char *name;  //姓名
    int num;     //学号
    int age;     //年龄
    char group;  //所在小组
    float score; //成绩
} stu1, stu2 = {"Tom", 12, 18, 'A', 136.5};
```

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;
}
```

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。

➢ 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

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

```
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;
}
// 结果：  0560
```

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

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;
}
// 结果：  266
```

为什么是266？接下来探索一下：

众所周知，1Byte = 8bit，而 1 char = 1Byte，1 int = 4 Byte。

写入数据的时候，内存是从右往左写的，所以`a.x[0] = 10;` 时，起始内存的前4个字节被写入为：

`0000 0000 | 0000 0000 | 0000 0000 | 0000 1010`

而`a.x[1] = 1;`时，变为：

`0000 0000 | 0000 0000 | 0000 0001 | 0000 1010`

即最终int读取时的二进制编码为：

`00000000000000000000000100001010 = 266`，即得正确输出✔

那么，继续探究，理论分析一下当执行 `a.x[2] = 100` 时会发生什么？

100 的二进制编码为 `1100100`，将其写入第3个Byte中，即得：

`0000 0000 | 0110 0100 | 0000 0001 | 0000 1010`

转换成十进制有：`00000000011001000000000100001010 = 6553866`

程序跑一下：

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

union
{
    int i;
    char x[2];
} a;
int main(void)
{
    a.x[0] = 10;
    a.x[1] = 1;
    a.x[2] = 100;
    printf("%d\n", a.i);
    return 0;
}
// 结果：  6553866
```