Chapter 8
Pointers and Pointer-Based Strings

OBJECTIVES

- What pointers are.
- The similarities and differences between pointers and references and when to use each.
- To use pointers to pass arguments to functions by reference.
- To use pointer-based C-style strings.
- The close relationships among pointers, arrays and C-style strings.
- To use pointers to functions.
- To declare and use arrays of C-style strings.

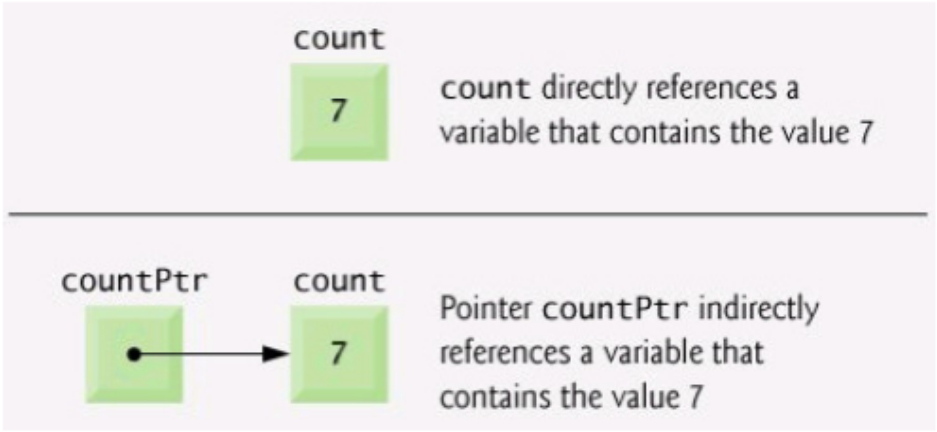
Topics

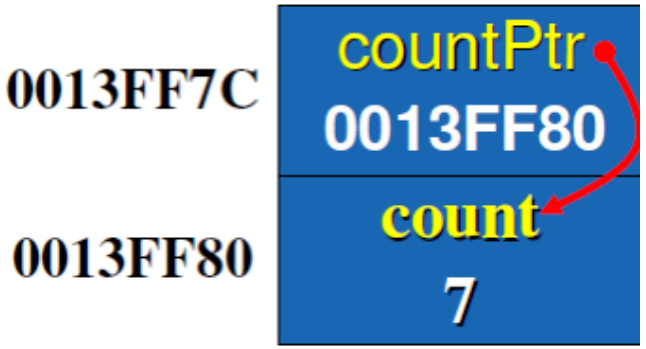
- **8.1 Pointer and Pointer Parameter**
- 8.2 sizeof Operators
- 8.3 Pointer Expressions and Pointer Arithmetic
- 8.4 Using const with Pointers
- 8.5 Relationship Between Pointers and Arrays
- 8.6 Selection Sort Using Pass-by-Reference
- 8.7 Arrays of Pointers
- 8.8 Case Study: Card Shuffling and Dealing Simulation
- 8.9 Function Pointers
- 8.10 Introduction to Pointer-Based String Processing

8.1 Pointer and Pointer Parameter

(1) Pointer Variable(指针变量): Variables contain memory addresses as their values.

```
int *countPtr, count = 7;
countPtr = &count; // 取地址符
```





8.1 Pointer and Pointer Parameter

(2) 声明两个指针变量:

- double *ptrX, ptrY;
  // Error, 仅声明了一个指针变量
- double *ptrX, *ptrY;

指针变量的初始值:

① int *countPtr = 0;

② int *countPtr = NULL;

③ int *countPtr;
  countPtr = 0; // NULL;

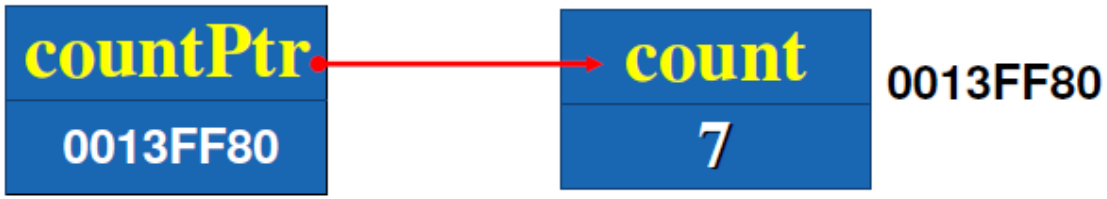
8.1 Pointer and Pointer Parameter

(3) Address operator (&): 单目操作符, 返回操作数的内存地址

```
int *countPtr, count = 7;
countPtr = &count; // 取地址符
```

- `int &countRef = count;` // 引用符

always preceded by a data-type name



8.1 Pointer and Pointer Parameter

- 1. int a = 10, b = 20;
- 2. cout << &a << endl;
- 3. cout << &b << endl;
- 4. cout << &(a + b) << endl;

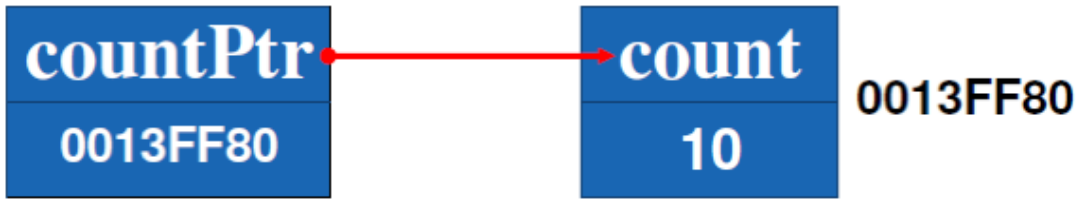
8.1 Pointer and Pointer Parameter

(4) * operator

- indirection/dereferencing operator(间接/解引用), 返回指针指向变量的别名
- int *countPtr, count = 7;

  countPtr = &count;

  *countPtr = 10; // *countPtr是count的别名



8.1 Pointer and Pointer Parameter

```
1.  // Fig. 8.4: fig08_04.cpp, Using the & and * operators, P.311
2.  #include <iostream>
3.  using std::cout;
4.  using std::endl;
5.  int main()
6.  {
7.     int a; // a is an integer, 假设地址0012F580
8.     int *aPtr; // aPtr is an int * -- pointer to an integer
9.
10.    a = 7; // assigned 7 to a
11.    aPtr = &a; // assign the address of a to aPtr
12.
13.    cout << "The address of a is " << &a << "\nThe value of aPtr is " << aPtr;
14.    cout << "\n The value of a is " << a << "\nThe value of *aPtr is " << *aPtr;
15.    cout << "\n &*aPtr = " << &*aPtr << "\n*&aPtr = " << *&aPtr << endl;
16.    return 0;
17. }
```

8.1 Pointer and Pointer Parameter

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  class GradeBook{
5.      int num;
6.  public:
7.      GradeBook( int n ){ num = n; }
8.      void displayMessage(){
9.          cout << "Hello to GradeBook " << num << endl;
10.     }
11. };
12. int main()
13. {
14.         GradeBook book1( 10 ), book2( 20 );
15.         GradeBook *pBook = &book1;
16.
17.         (*pBook).displayMessage();
18.
19.         return 0;
20. }
```

8.1 Pointer and Pointer Parameter

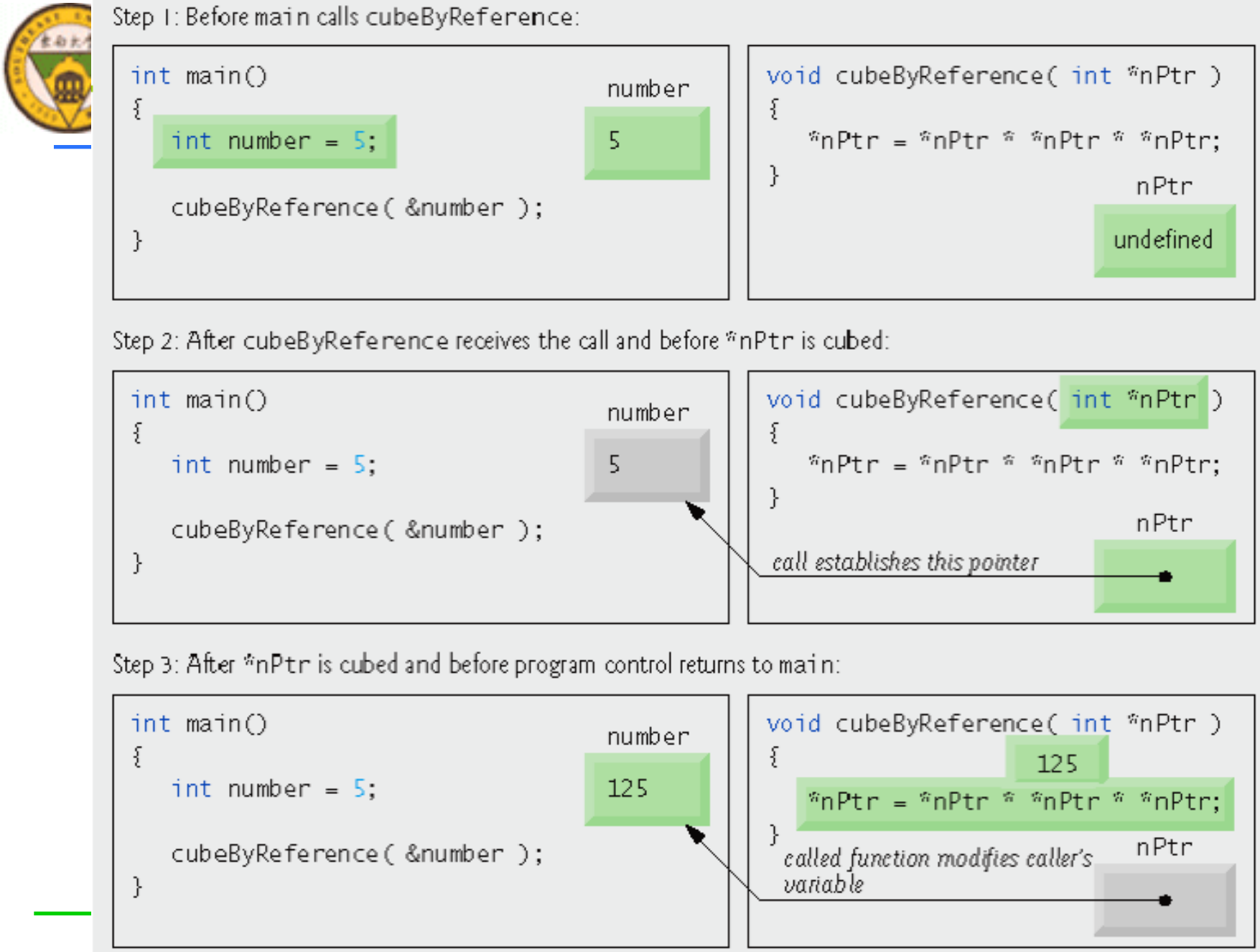
(5) 函数参数传递的两种方式:

- Pass-by-Value, 传值
- Pass-by-Reference, 传引用
  - Reference Parameter, 引用参数
  - Pointer Parameter, 指针参数

8.1 Pointer and Pointer Parameter

```
1.  // Fig. 8.7: fig08_07.cpp, P.300
2.  #include <iostream>
3.  using std::cout;
4.  using std::endl;
5.
6.  void cubeByReference( int *nPtr ) // 指针类型形参
7.  {
8.      *nPtr = *nPtr * *nPtr * *nPtr; // cube *nPtr
9.  }
10. int main()
11. {
12.     int number = 5;
13.     cout << "The original value of number is " << number;
14.     cubeByReference( &number ); // pass number address
15.     cout << "\nThe new value of number is " << number << endl;
16.     return 0;
17. }
```

int *nPtr = &number;


Step 1: Before main calls cubeByReference:
int main()
{
int number = 5;
cubeByReference(&number);
}
number
5
void cubeByReference(int *nPtr)
{
*nPtr = *nPtr * *nPtr * *nPtr;
}
nPtr
undefined
Step 2: After cubeByReference receives the call and before *nPtr is cubed:
int main()
{
int number = 5;
cubeByReference(&number);
}
number
5
void cubeByReference(int *nPtr)
{
*nPtr = *nPtr * *nPtr * *nPtr;
}
nPtr
call establishes this pointer
Step 3: After *nPtr is cubed and before program control returns to main:
int main()
{
int number = 5;
cubeByReference(&number);
}
number
125
void cubeByReference(int *nPtr)
{
125
*nPtr = *nPtr * *nPtr * *nPtr;
}
called function modifies caller's variable
nPtr

Topics

- 8.1 Pointer and Pointer Parameter
- **8.2 Using const with Pointers**
- 8.3 Selection Sort Using Pass-by-Reference
- 8.4 sizeof Operators
- 8.5 Pointer Expressions and Pointer Arithmetic
- 8.6 Relationship Between Pointers and Arrays
- 8.7 Introduction to Pointer-Based String Processing
- 8.8 Arrays of Pointers
- 8.9 Function Pointers

8.2 Using const with Pointers

- 常量变量

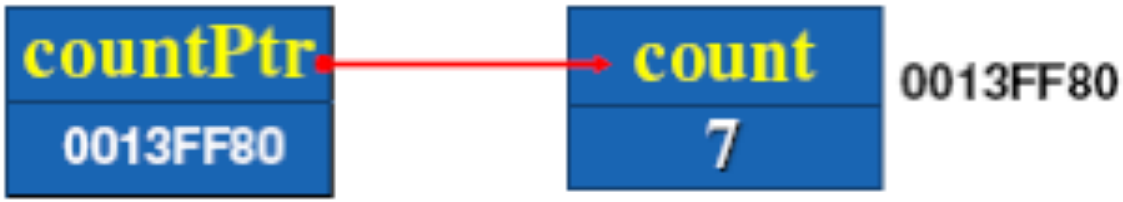
  const int m = 1000; //声明时必须初始化

  表示int型变量m为常量变量

- 问题: 指针类型的变量p如何声明为常量?

  const int * p = ……

- **Constant Pointer**: 指针常量, 指针类型的常量
- **Pointer to Constant**: 常量指针, 指向常量的指针



8.2 Using const with Pointers

分　类	non-constant data	constant data
non-constant pointer	int * p1	const int * p2
constant pointer	int * const p3	const int * const p4

- **p1: Nonconstant pointer to Nonconstant data**
- **数据可以修改: data can be modified through the dereferenced pointer**
- **指针可以修改: and the pointer can be modified to point to other data**

8.2 Using const with Pointers

分　类	non-constant data	constant data
non-constant pointer	int * p1	const int * p2
constant pointer	int * const p3	const int * const p4

- **p2: Nonconstant pointer to Constant data**
- 常量指针,指向常量的指针
- 指针可以修改, 数据不能修改

程序解读（P262）

8.2 Using const with Pointers

分　类	non-constant data	constant data
non-constant pointer	int * p1	const int * p2
constant pointer	int * const p3	const int * const p4

- **p3: Constant pointer to Nonconstant data**
- 指针常量,指针类型的常量
- 指针不能修改, 数据可以修改, 即常量变量
- 注意: 声明时必须进行初始化, 或作为形参通过实参初始化

```
int m1 = 1000, m2 = 2000;
int * const p3 = &m1;   *p3 = 1500;
p3 = &m2;
```

8.2 Using const with Pointers

分　类	non-constant data	constant data
non-constant pointer	int * p1	const int * p2
constant pointer	int * const p3	const int * const p4

- **p4: Constant pointer to Constant data**
- 指向常量的指针常量,常量指针+指针常量
- 指针不能修改, 数据不能修改
- 声明时必须进行初始化, 或作为形参通过实参初始化

程序解读（P264）

Topics

- 8.1 Pointer and Pointer Parameter
- 8.2 Using const with Pointers
- **8.3 Selection Sort Using Pass-by-Reference**
- 8.4 sizeof Operators
- 8.5 Pointer Expressions and Pointer Arithmetic
- 8.6 Relationship Between Pointers and Arrays
- 8.7 Introduction to Pointer-Based String Processing
- 8.8 Arrays of Pointers
- 8.9 Function Pointers

8.3 Selection Sort Using Pass-by-Reference

- Selection Sort(选择排序): 首先找出最小元素, 将其交换至数组0号位; 其次, 在剩余元素中找出最小元素, 将其交换至1号位; 以此类推.

- void swap(int * const element1Ptr, int * const element2Ptr)
- 指针常量, 指向的数据可修改, 指向的内存地址不能改

```
void swap(int *const e1, int *const e2)
{
   int hold=*e1;
   *e1=*e2;
   *e2=hold;
}
void selectionSort(int * const array, int size)
{
   int smallest;
   for(int i=0;i<size;i++)
   {
      smallest=i;
      for(int j=i+1;j<size;j++)
         if(array[j]<array[smallest])
            smallest=j;
      swap(&array[i],&array[smallest]);
   }
}
```

Topics

- 8.1 Pointer and Pointer Parameter
- 8.2 Using const with Pointers
- 8.3 Selection Sort Using Pass-by-Reference
- 8.4 sizeof Operators
- 8.5 Pointer Expressions and Pointer Arithmetic
- 8.6 Relationship Between Pointers and Arrays
- 8.7 Introduction to Pointer-Based String Processing
- 8.8 Arrays of Pointers
- 8.9 Function Pointers

8.4 sizeof Operators

- **Compile-time operator: determine the size of operand (占用的以字节为单位的内存空间大小)**
- **Operand(操作数)**

① 变量和常量名: 是否带括号可选

```
int number;
sizeof( number )
sizeof number
```

② 类型名: 必须带括号

```
sizeof( char )
sizeof( GradeBook )
```

8.4 sizeof Operators

```
16.  int array[20];
33.  cout << sizeof(array); // 80 = 4 * 20
```

- Calculate number of elements

  sizeof(数组名)/sizeof(数组元素类型)

  sizeof(array) / sizeof(int) // 80 / 4 = 20

- 问题:

  void displayMessage(int s[ ], int size)

  为何**size**参数必需?

8.4 sizeof Operators

```
1.  #include <iostream>
2.  using std::cout;
3.  using std::endl;
4.
5.  size_t getSize( double s[ ] ); // size_t 即 unsigned int
6.
7.  int main()
8.  {
9.      double array[ 20 ];
10.
11.     cout << "The number of bytes in the array is " << sizeof( array );
12.     cout << "\nThe number of bytes returned by getSize is " << getSize( array );
13.     return 0;
14. }
15.
16. size_t getSize( double s[ ] )
17. {
18.    return sizeof( s );
19. }
```

sizeof(double)*20 = 160

```
The number of bytes in the array is 160
The number of bytes returned by getSize is 4
```

8.4 sizeof Operators

- 数组名的值即数组首元素的地址
- 当数组名作为实参传递时, 本质上是传递数组地址

- 编译器不区分接受指针参数的函数和接受一维数组名参数的函数

  size_t getSize(double s[ ]);

  size_t getSize(double *s);

- 更进一步, 当编译器遇到**double s[ ]**参数时, 会自动将该参数转换为指向**double**的指针**s**

Topics

- 8.1 Pointer and Pointer Parameter
- 8.2 Using const with Pointers
- 8.3 Selection Sort Using Pass-by-Reference
- 8.4 sizeof Operators
- **8.5 Pointer Expressions and Pointer Arithmetic**
- 8.6 Relationship Between Pointers and Arrays
- 8.7 Introduction to Pointer-Based String Processing
- 8.8 Arrays of Pointers
- 8.9 Function Pointers

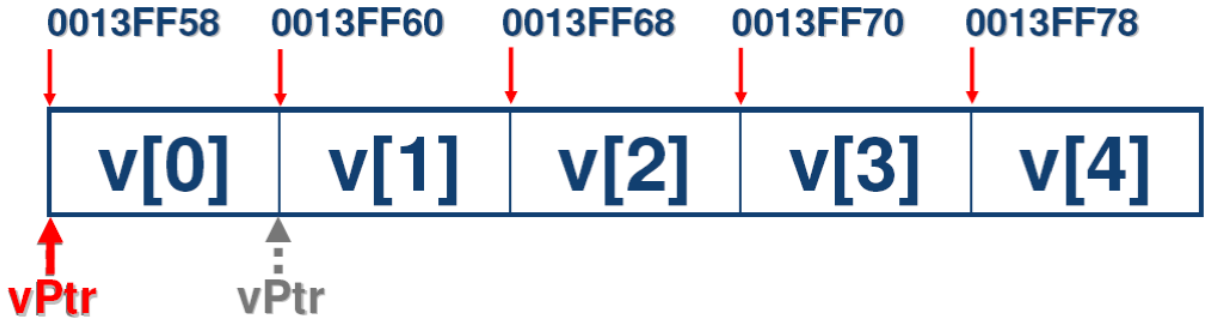
8.5 Pointer Expressions and Pointer Arithmetic

- Pointers are valid operands in arithmetic expressions(算术), assignment expressions(赋值) and comparison expressions(比较).
- However, not all the operators normally used in these expressions are valid with pointer variables.
- 指针运算一般与数组结合应用!

8.5 Pointer Expressions and Pointer Arithmetic

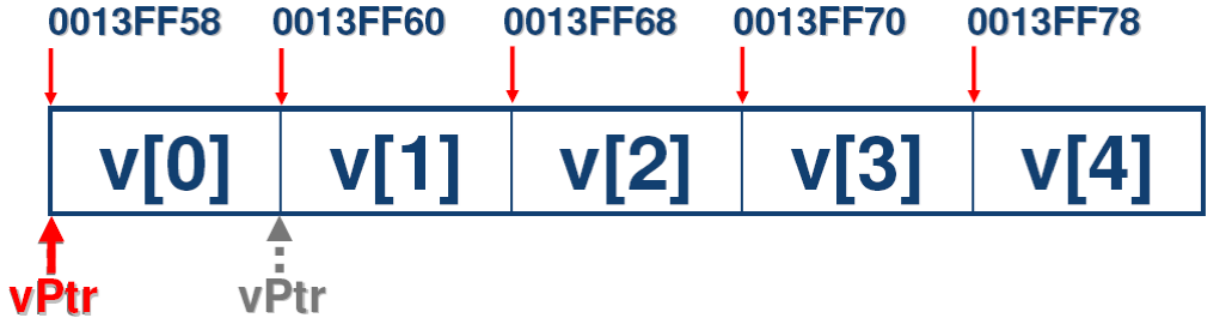
(1)自增、加法赋值运算　　// v[0]

```
double v[5] = {0};
double *vPtr = &v[0]; // 0013FF58
```



- vPtr++;　　// v[1]
- vPtr += 3;　// v[4]

8.5 Pointer Expressions and Pointer Arithmetic



- 注意与数值算术运算的区别！
- 数值运算：

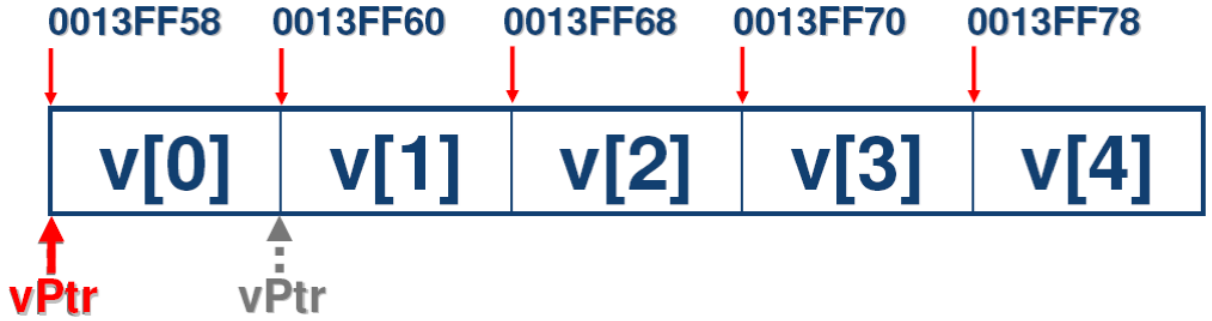
  int num = 0x0013FF58;　　// 1310552

  num++运算后，值为0x0013FF59. // 1310553
- 指针运算：

  vPtr指向元素v[0]，值为0x0013FF58.　　// 1310552

  vPtr++运算后，指向v[1]，值为0x0013FF60. // 1310560

8.5 Pointer Expressions and Pointer Arithmetic

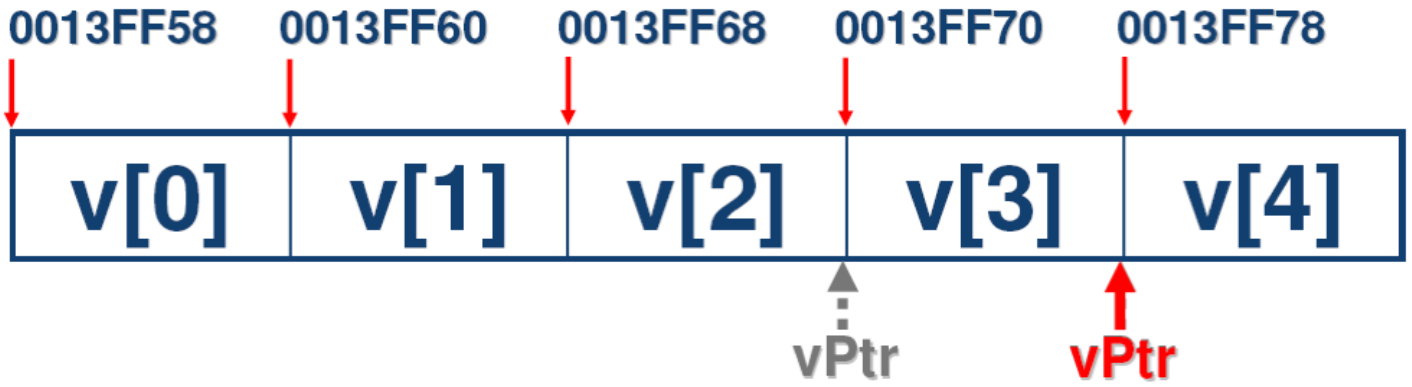


- 指针运算结论:
- 指针运算(假设指向类型type的指针)时, +/-n表示前移/后移n个元素, 其中n称为offset(偏移值)
- 从数值上看, 指针的值是加/减了 n * sizeof(type)

8.5 Pointer Expressions and Pointer Arithmetic

(2)自减、减法赋值运算// v[4]

```
double v[5] = {0};
double *vPtr = &v[4]; // 0013FF78
```

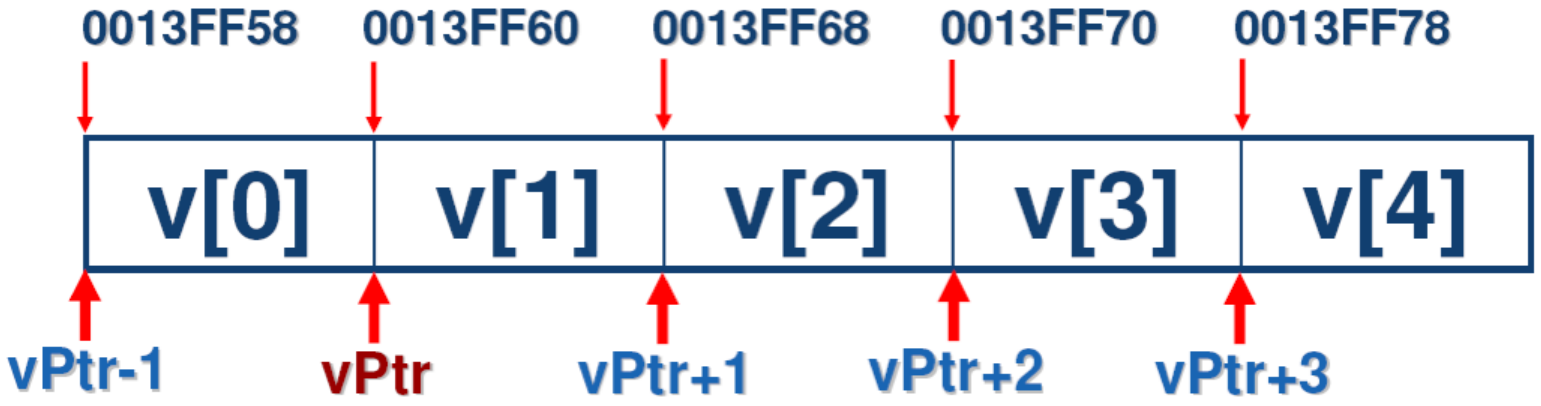


- vPtr--; // v[3]
- vPtr -= 3; // v[0]

8.5 Pointer Expressions and Pointer Arithmetic

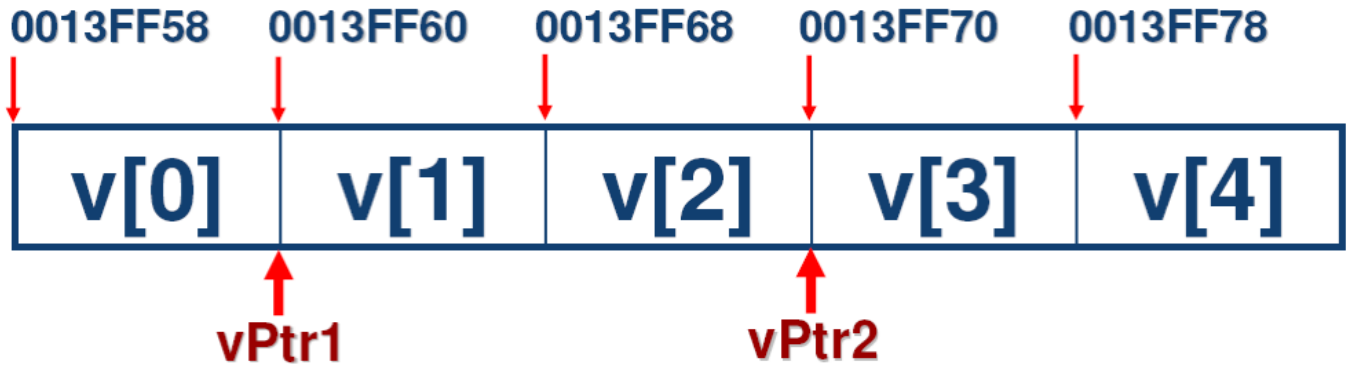
(3)加法、减法表达式// v[1]

```
double v[5] = {0};
double *vPtr = &v[1]; // 0013FF60
```



8.5 Pointer Expressions and Pointer Arithmetic

- double v[5] = {0};
- double *vPtr1 = &v[1]; // 0013FF60
- double *vPtr2 = &v[3]; // 0013FF70



- int x = vPtr2 – vPtr1; cout << x;
- 输出2, 表示两个指针间相差几个元素

8.5 Pointer Expressions and Pointer Arithmetic

- double *vPtr1 = &v[1], *vPtr2 = &v[3];

  vPtr1 + vPtr2

  error C2110: cannot add two pointers
- int m = 0; int *ptrM = &m;

  ptrM++;

  No error, 但此时ptrM指针无意义(Why Array?)
- ptrInt – ptrDouble

  error C2440: cannot convert from 'double *' to 'int *', 必须相同类型指针才能相减

8.5 Pointer Expressions and Pointer Arithmetic

(4)赋值运算和通用指针(Generic pointer)

- 仅有相同类型的指针之间可以进行赋值操作(否则必须进行类型转换), 特例: 通用指针void *

```
int num = 0;
int *ptrNum = &num;
void *p = ptrNum;
```

- 任意类型指针均可以赋值给通用指针, 反之不成立!

8.5 Pointer Expressions and Pointer Arithmetic

- 通用指针仅用于保存地址值, 不能进行解引用和算术运算:

```
cout << *p << endl;
```

error C2100: illegal indirection

```
cout << p+1 << endl;
```

error C2036: 'void *' : unknown size

8.5 Pointer Expressions and Pointer Arithmetic

(5)等价与关系运算

- 等价运算符, 判断某指针是否为空指针

```
if (pGradeBook == 0) // NULL
        cout << “error” << endl;
else
        (*pGradeBook).displayMessage();
```

- 关系运算符, 一般用于数组

Topics

- 8.1 Pointer and Pointer Parameter
- 8.2 Using const with Pointers
- 8.3 Selection Sort Using Pass-by-Reference
- 8.4 sizeof Operators
- 8.5 Pointer Expressions and Pointer Arithmetic
- **8.6 Relationship Between Pointers and Arrays**
- 8.7 Introduction to Pointer-Based String Processing
- 8.8 Arrays of Pointers
- 8.9 Function Pointers

8.6 Relationship Between Pointers and Arrays

- Arrays and pointers are intimately related in C++ and may be used almost interchangeably.

8.6 Relationship Between Pointers and Arrays

```
int b[5];
int *bPtr = &b[0]; // = b;
```

- 数组名b是指向数组首元素的指针常量
- bPtr是指向数组首元素的指针

```
int b[5];
int * const bPtr = &b[0]; // = b
```

- b实际上等价于此处的指针常量bPtr

8.6 Relationship Between Pointers and Arrays

```
int b[5];
int *bPtr = &b[0]; // = b;
```

- 除了b的const限定外, b和bPtr可互换使用:
- 数组subscript下标运算(方括号[ ]运算符)
- 指针offset偏移运算(指针算术运算)

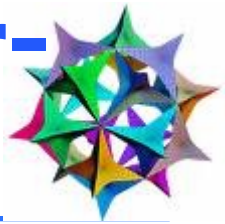
- ✓ b[3] 等价于 *(bPtr+3)
- ✓ &b[3] 等价于 bPtr+3
- ✓ bPtr[3], 指针可以进行下标运算
- ✓ *(b+3), 数组名可进行偏移运算

程序解读（P271）

Topics

- 8.1 Pointer and Pointer Parameter
- 8.2 Using const with Pointers
- 8.3 Selection Sort Using Pass-by-Reference
- 8.4 sizeof Operators
- 8.5 Pointer Expressions and Pointer Arithmetic
- 8.6 Relationship Between Pointers and Arrays
- 8.7 Introduction to Pointer-Based String Processing
- 8.8 Arrays of Pointers
- 8.9 Function Pointers

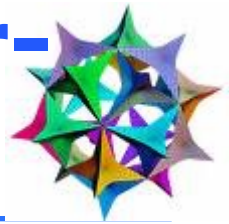
8.7 Introduction to Pointer-Based String Processing



- **Part I: Fundamentals of Characters and Pointer-Based Strings**

- **Part II: String Manipulation Functions of the String-Handling Library**

8.7 Introduction to Pointer-Based String Processing

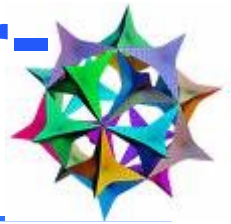


- **Characters字符**
  - ‘A’, ‘+’, ‘?’, ‘\0’, ‘\n’ , ‘\\’ 等
  - 字符常量, 值为其ASCII 码值

- **char类型变量可以直接进行关系运算**

```
1.  char a = 'a', b;
2.  cin >> b;
3.  if ( a > b )
4.      cout << a << endl;
5.  else
6.      cout << b << endl;
```

8.7 Introduction to Pointer-Based String Processing



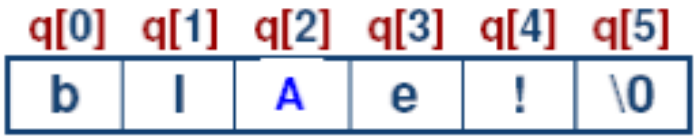
- String字符串
  - include letters(字母A~Z, a-z), digits(数字0~9) and various special characters
  - such as '+', '-', '*', '/' and '$'.
- "Welcome to C++!", "Hello, World!" 等
  - String literals字符串文本, or string constants字符串常量

①值为首字符的地址(char *)

②全局数据区, 静态存储类别

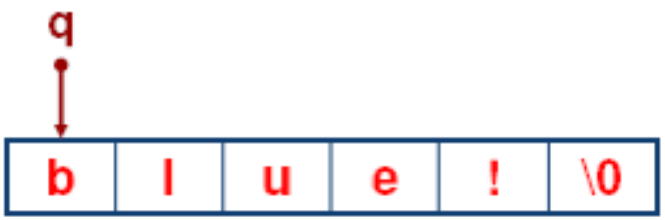
8.7 Introduction to Pointer-Based String Processing

1. char q[ ] = "blue!";
2. q[2] = 'A';
3. cout << q << endl;



- 用字符串常量对数组初始化

```
const
1. char *q = "blue!";
2. *(q+2) = 'A';
3. cout << q << endl;
```



- 将字符串常量的地址赋值给字符指针
- 赋值‘A’时报内存访问错误

8.7 Introduction to Pointer-Based String Processing

(1)字符数组与字符串的关系

- 数组中的元素可以是引用之外的任何类型,其中一种特殊类型即字符数组, 可用于表示字符串!
- C/C++语言中的字符串(*C-Style String*)

  连续内存区域+ 字符+ ‘\0’结尾

  其中‘\0’称为NULL Char, 即空字符, 对应整数值为0

8.7 Introduction to Pointer-Based String Processing

(2)字符数组的初始化

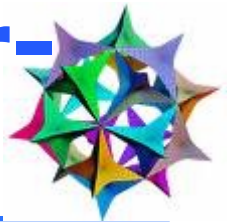
- 初始化列表(与普通数组相同)

```
char stringx[ ] = { 'f', 'i', 'r', 's', 't' };
char string1[ ] = { 'f', 'i', 'r', 's', 't', '\0' };
```

- 通过字符串常量进行初始化

```
char string2[ ] = "first";
```

8.7 Introduction to Pointer-Based String Processing



(3)字符数组的赋值

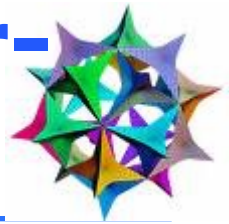
- 循环语句逐个元素赋值(与普通数组相同)
- 流操作运算

```
char string3[ 7 ];
cin >> string3; // Hello
```

- 将用户输入的字符串从string3对应的内存起始地址开始写入, 并在结尾处加空字符!

‘H’	‘e’	‘l’	‘l’	‘o’	‘\0’	3	‘A’

8.7 Introduction to Pointer-Based String Processing



- **int n[20];**

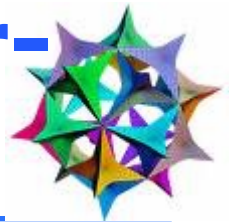
  for(int i = 0; i < 20; i++)

  cin >> n[i];

- **char s[20];**

  cin >> s;
- 若输入多于19个字符, 则溢出
- 遇空白字符认为输入结束

8.7 Introduction to Pointer-Based String Processing



- char s[20];
- cin.getline(s, n); // n = 20

  cin.getline(s, n, '\n'); // 缺省实参
- The function stops reading characters
  - 当读到delimiter character(分隔符, 缺省为'\n')
  - 或已读到n-1 个字符(防止溢出)
- cin >> setw(n) >> s; // n = 20
- 指定最多读入n-1个字符, 并在尾部自动添加null character

7.4 String and Char Array

(4)字符数组的输出

- 逐个元素读取并输出

- 流操作运算

  cout << string3; // char string3[8]

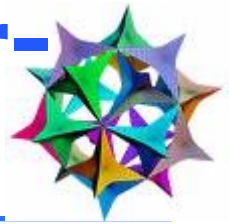
  从**string3**对应的内存起始地址开始读数据,直至遇到空字符

‘H’	‘e’	‘l’	‘l’	‘o’	‘!’	‘!’	‘\0’

Q & A

	非char数组 int num[6]	char数组 char s[6]
1. 初始化	初始化列表 int num[6] = {1, 2, 3};	初始化列表 char s[6] = {'1', '2', '3'};
作为String处理		字符串常量 char s[6] = "Hello";
2. 赋　值	逐个元素 Repetition Structure	逐个元素 Repetition Structure
作为String处理		cin >> s; cin.getline(s, 6);
3. 输　出	逐个元素 Repetition Structure	逐个元素 Repetition Structure
作为String处理		cout << s;

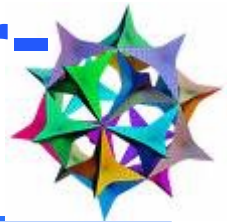
8.7 Introduction to Pointer-Based String Processing



- Part I: Fundamentals of Characters and Pointer-Based Strings

- Part II: String Manipulation Functions of the String-Handling Library
- C语言继承而来的String处理函数库头文件 <cstring>

8.7 Introduction to Pointer-Based String Processing



- char *strcpy(char *s1, const char *s2);
- char *strncpy(char *s1, const char *s2, size_t n);
- char *strcat(char *s1, const char *s2);
- char *strncat(char *s1, const char *s2, size_t n);
- int strcmp(const char *s1, const char *s2);
- int strncmp(const char *s1, const char *s2, size_t n
- char *strtok(char *s1, const char *s2);
- size_t strlen(const char *s);

8.7 Introduction to Pointer-Based String Processing

(1) char *strcpy(char *s1, const char *s2);

- Copies the string s2 into the character array s1. The value of s1 is returned.

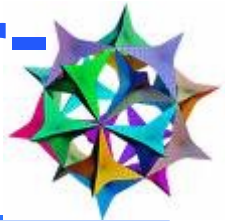
- char s1[8]:

  "Hello":

a	b	c	d	e	f	g	\0
H	e	l	l	o	\0	g	\0

- 调用：cout<<strcpy(s1, "Hello");

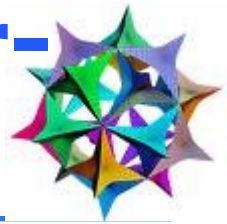
8.7 Introduction to Pointer-Based String Processing



(2)char *strncpy(char *s1, const char *s2, size_t n);

- Copies at most n characters of the string s2 into the character array s1. The value of s1 is returned.
- Note: strncpy并不保证拷贝null character, 仅当n的值大于s2的长度时null character才会拷贝.

8.7 Introduction to Pointer-Based String Processing



(2)char *strncpy(char *s1, const char *s2, size_t n);

- char s1[8]
  char* s2 = “Hel”

  n = 3:

  + s1[3] = ‘\0’:

  n > 3, 如5:

a	b	c	d	e	f	g	\0

H	e	l	d	e	f	g	\0

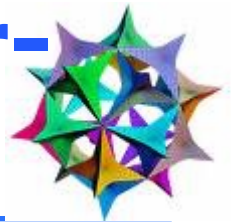
H	e	l	\0	e	f	g	\0

H	e	l	\0	\0	f	g	\0

```
1.  #include <cstring> // Fig_8.31
2.  using std::strcpy;
3.  using std::strncpy;
4.  int main()
5.  {
6.     char x[ ] = "Happy Birthday to You"; // string length 21
7.     char y[ 25 ], z[ 15 ];
8.
9.     strcpy( y, x ); // copy contents of x into y
10.    cout << "The string in array x is: " << x
11.         << "\nThe string in array y is: " << y << '\n';
12.
13.    // copy first 14 characters of x into z
14.    strncpy( z, x, 14 ); // does not copy null character
15.    z[ 14 ] = '\0'; // append '\0' to z's contents
16.    cout << "The string in array z is: " << z << endl;
17.    return 0;
18. }
```

```
x is: Happy Birthday to You
y is: Happy Birthday to You
z is: Happy Birthday
```

8.7 Introduction to Pointer-Based String Processing



(3)char *strcat(char *s1, const char *s2);

- **Appends s2 to s1. The first character of s2 overwrites s1' the terminating null character.**

  s1 + s2 + null character

(4)char *strncat(char *s1, const char *s2, size_t n);

- **Appends at most n characters of string s2 to string s1. The first character of s2 overwrites the terminating null character of s1.**

  s1 + n char of s2 + null character

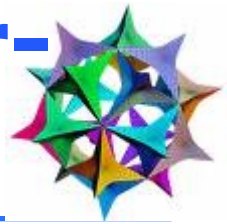
```
1.  char s1[ 20 ] = "Happy "; // length 6
2.  char s2[] = "New Year "; // length 9
3.  char s3[ 40 ] = "";
4.
5.  cout << "s1 = " << s1 << "\ns2 = " << s2;
6.
7.  strcat( s1, s2 ); // concatenate s2 to s1 (lengt
8.  cout << "\n\nAfter strcat(s1, s2):\ns1 = " << s1 << "\ns2 = " << s2
9.
10. // concatenate first 6 characters of s1 to s3
11. strncat( s3, s1, 6 ); // places '\0' after last cha
12. cout << "\n\nAfter strncat(s3, s1, 6):\ns1 = " << s1
13.      << "\ns3 = " << s3;
14.
15. strcat( s3, s1 ); // concatenate s1 to s3
16. cout << "\n\nAfter strcat(s3, s1):\ns1 = " << s1
17.      << "\ns3 = " << s3 << endl;
```

```
s1 = Happy New Year
s2 = New Year
```

```
s1 = Happy New Year
s3 = Happy
```

```
s1 = Happy New Year
s3 = Happy Happy New Year
```

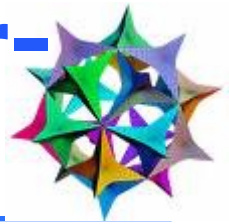
8.7 Introduction to Pointer-Based String Processing



```
1.  char s[100];
2.  cout << strcat( strcpy (s, "Hello "), "world!")
3.           << endl;
```

- Hello world!

8.7 Introduction to Pointer-Based String Processing



(5)int strcmp(const char *s1, const char *s2);

- **Compares the string s1 with the string s2. The function returns a value of:**

① =0: s1 is equal to s2

② <0 (usually -1): s1 is less than s2

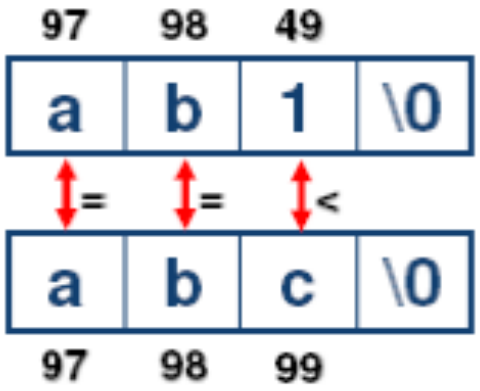
③ >0 (usually 1): s1 is greater than s2.

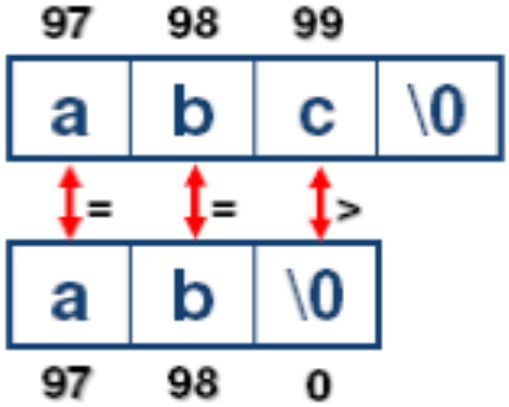
(6)int strncmp(const char *s1, const char *s2, size_t n);

- **Compares up to n characters of the string s1 with the string s2.**

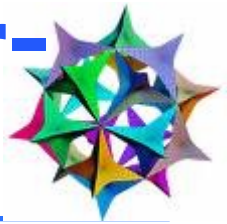
8.7 Introduction to Pointer-Based String Processing

- strcmp(“ab1”, “abc”)　　结果为-1
- strncmp(“ab1”, “abc”, 2) 结果为0
- strcmp(“abc”, “ab”)　　结果为1





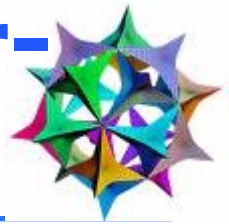
8.7 Introduction to Pointer-Based String Processing



(7)size_t strlen(const char *s);

- Determines the **length** of string s. The number of characters **preceding** the terminating null character is returned.

8.7 Introduction to Pointer-Based String Processing



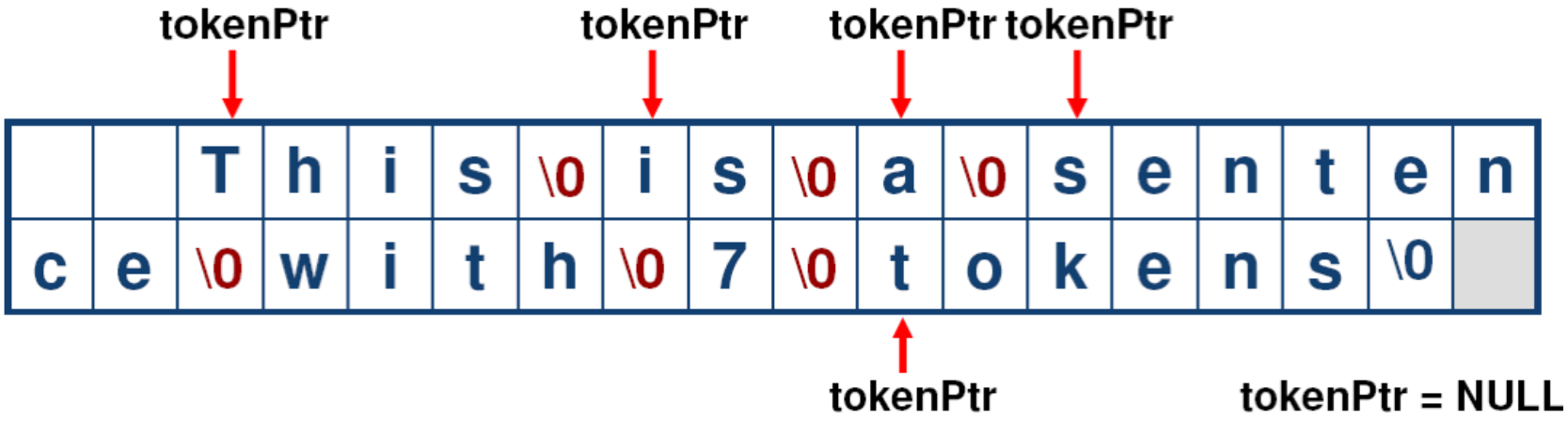
(8)char *strtok(char *s1, const char *s2);

- 将字符串**s2**中的字母作为**delimiter**分隔符, 将字符串**s1**分解为若干个**token.**
- **char sentence[ ]= “ This is a sentence with 7 tokens”;**
- **1. tokenPtr = strtok(sentence, “ ”);**
- **2. tokenPtr = strtok(NULL, “ ”);**

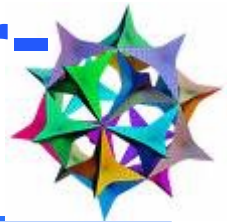
8.7 Introduction to Pointer-Based String Processing

(8)char *strtok(char *s1, const char *s2);

- tokenPtr = strtok(sentence, “ ”);
- tokenPtr = strtok(NULL, “ ”);



8.7 Introduction to Pointer-Based String Processing



- +国际区号-(区号) 本地号码
- char s[ ] = "+86-(025) 52091012";

- 1. strtok(s, "+-() ");
- 2. strtok(NULL, "+-() ");

Topics

- 8.1 Pointer and Pointer Parameter
- 8.2 Using const with Pointers
- 8.3 Selection Sort Using Pass-by-Reference
- 8.4 sizeof Operators
- 8.5 Pointer Expressions and Pointer Arithmetic
- 8.6 Relationship Between Pointers and Arrays
- 8.7 Introduction to Pointer-Based String Processing
- **8.8 Arrays of Pointers**
- 8.9 Function Pointers

8.8 Arrays of Pointers

- 数组元素可以是除引用外的任意类型:

```
int m1 = 11, m2 = 22, m3 = 33;
int *ms[ ] = { &m1, &m2, &m3 };
cout << *ms[0] << ' '
     << *ms[1] << ' '
     << *ms[2]
     << endl;
```

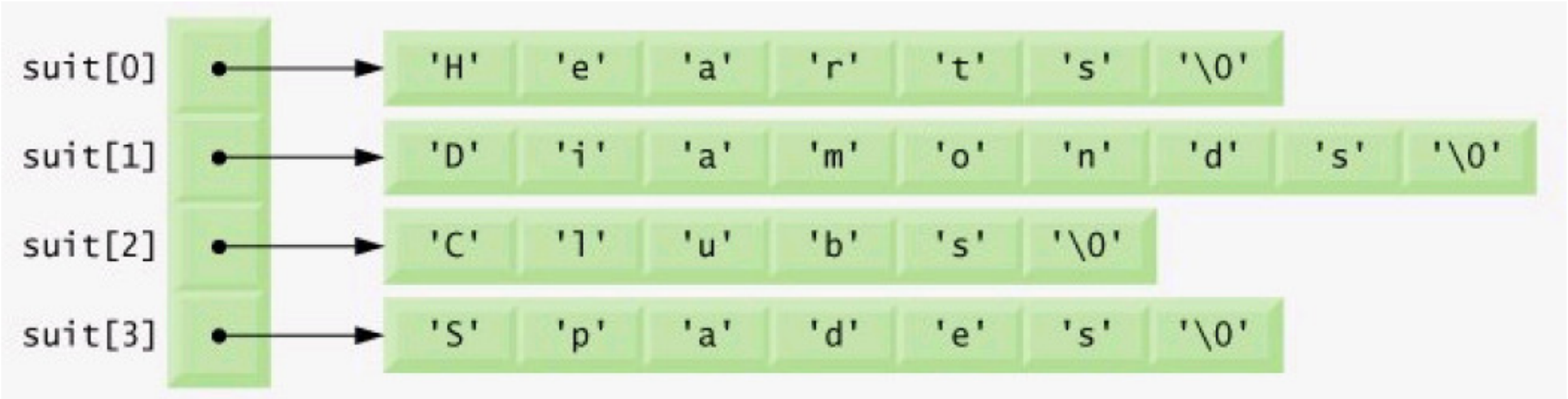
8.8 Arrays of Pointers

- const char *suit[4] = { "Hearts",
  "Diamonds",
  "Clubs",
  "Spades" };

可以省略，但一般保留！

- String array,字符串数组

值为首字符的地址



Topics

- 8.1 Pointer and Pointer Parameter
- 8.2 Using const with Pointers
- 8.3 Selection Sort Using Pass-by-Reference
- 8.4 sizeof Operators
- 8.5 Pointer Expressions and Pointer Arithmetic
- 8.6 Relationship Between Pointers and Arrays
- 8.7 Introduction to Pointer-Based String Processing
- 8.8 Arrays of Pointers
- **8.9 Function Pointers**

8.9 Function Pointers

- 函数代码同样保存在内存中, 因此亦具有内存地址.
- 如何确定函数地址, 如何保存函数地址, 如何根据地址调用函数?

6.11 Function Call Stack and Activation Records(函数调用栈和活动记录)

程序占用内存区域的分布:

- 代码区(code area)
  - 被编译程序的执行代码部分
- 全局数据区(data area)
  - 常量、静态全局量和静态局部量等
- 堆区(heap)
  - 动态分配的内存, new / delete
- 栈区(stack)
  - 函数数据区, 函数形参和局部变量等自动变量所使用的内存区域

8.9 Function Pointers

(1) 函数地址的确定

- **the name of a function is actually the starting address in memory of the code that performs the function's task.**

(2) 函数地址的保存

- **Function Pointer(函数指针): 包含函数地址的指针**

8.9 Function Pointers

- 函数指针的类型
  - ✓ bool ascending(int, int);

    TYPE *p = ascending;

    返回类型　　输入参数

    bool (*p)(int, int) = ascending;
  - ✓ bool descending(int, int);

    bool (*p)(int, int) = descending;

8.9 Function Pointers

- (3) 根据函数地址调用函数

```
bool ascending( int, int );
int main()
{
    bool (*p)(int, int) = ascending;
    int a = 10, b = 20;
    (*p)( a, b ); // (1) 解引用后调用函数
    p( a, b );    // (2) 直接通过地址调用函数
    return 0;
}
```

```
1.  void func( )
2.  {
3.      cout << func << endl;
4.  }
5.  int main( )
6.  {
7.      func( );
8.      void (*p)()  = func;
9.      cout << p << endl;
10.     (*p)( ); // 解引用后调用函数
11.     p( );    // 直接通过地址调用函数
12.     return 0;
13. }
```

```
004010DC
004010DC
004010DC
004010DC
```

8.9 Function Pointers

(4) 应用— 函数指针作为参数

```
1. bool ascending( int a, int b ); // return true if a < b, else false
2. bool descending( int a, int b ); // return true if a > b, else false
```

```
1. void selectionSort( int work[ ], const int size, bool (*compare)( int, int ) )
2. {
3.     if(!(*compare)( work[ smallestOrLargest ], work[ index ] )) ……
4. }
```

```
bool (*compare)( int, int ) = ascending;
```

```
1. int main()
2. {
3.     selectionSort( a, arraySize, ascending );
4.     selectionSort( a, arraySize, descending );
5. }
```

程序解读（P276）

Summary

- 指针Pointer
- sizeof运算符
- 指针表达式和指针运算
- const pointer指针常量和pointer to const
- 常量指针
- 指针和数组的关系
- 指针数组
- 函数指针
- 常用的基于指针字符串的处理函数

Homework

- 实验必选题目:
- 实现PPT 60页中的以下函数:

```
char *strcpy( char *s1, const char *s2 );
int strcmp( const char *s1, const char *s2 );
size_t strlen( const char *s );
```

- 实验任选题目:
- 15(快速排序), 16(迷宫)

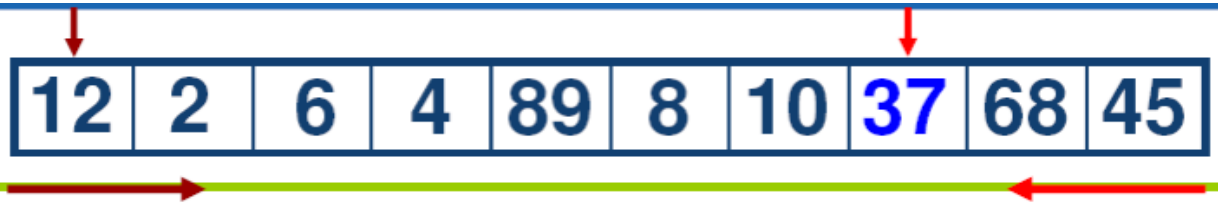
排序算法

- **Quick Sort 快速排序算法**
  - **a) 划分。取未排序数组第一个元素，确定其在数组中的最终位置。确定一个元素的合适位置，有两个未排序子数组。**
  - **b) 递归: 对每个未排序的子数组执行第一步**


```
1.  int partition( int * const array, int left, int right )
2.  {
3.    int position = left;
4.    while ( true )
5.    {
6.      while ( array[ position ] <= array[ right ] && position != right )  right--;
7.      if ( position == right ) return position;
8.      if ( array[ position ] > array[ right ])
9.      {
10.       swap( &array[ position ], &array[ right ] );
11.       position = right;
12.     }
13.
14.     while ( array[ left ] <= array[ position ] && left != position )  left++;
15.     if ( position == left ) return position;
16.     if ( array[ left ] > array[ position ] )
17.     {
18.       swap( &array[ position ], &array[ left ] );
19.       position = left;
20.     }
21.   }
22. }
```

小

大




```
1.  int partition( int * const array, int left, int right )
2.  {
3.     int position = left;
4.     while ( true )
5.     {
6.        while ( array[ position ] <= array[ right ] && position != right )  right--;
7.        if ( position == right ) return position;
8.        if ( array[ position ] > array[ right ])
9.        {
10.          swap( &array[ position ], &array[ right ] );
11.          position = right;
12.       }
13.
14.       while ( array[ left ] <= array[ position ] && left != position )  left++;
15.       if ( position == left ) return position;
16.       if ( array[ left ] > array[ position ] )
17.       {
18.          swap( &array[ position ], &array[ left ] );
19.          position = left;
20.       }
21.    }
22. }
```

小

大


```
1.  int partition( int * const array, int left, int right )
2.  {
3.    int position = left;
4.    while ( true )
5.    {
6.      while ( array[ position ] <= array[ right ] && position != right )  right--;
7.      if ( position == right ) return position;
8.      if ( array[ position ] > array[ right ])
9.      {
10.       swap( &array[ position ], &array[ right ] );
11.       position = right;
12.     }
13.
14.     while ( array[ left ] <= array[ position ] && left != position )  left++;
15.     if ( position == left ) return position;
16.     if ( array[ left ] > array[ position ] )
17.     {
18.       swap( &array[ position ], &array[ left ] );
19.       position = left;
20.     }
21.   }
22. }
```

小

大


```
1.  int partition( int * const array, int left, int right )
2.  {
3.    int position = left;
4.    while ( true )
5.    {
6.      while ( array[ position ] <= array[ right ] && position != right )  right--;
7.      if ( position == right ) return position;
8.      if ( array[ position ] > array[ right ])
9.      {
10.       swap( &array[ position ], &array[ right ] );
11.       position = right;
12.     }
13.
14.     while ( array[ left ] <= array[ position ] && left != position )  left++;
15.     if ( position == left ) return position;
16.     if ( array[ left ] > array[ position ] )
17.     {
18.       swap( &array[ position ], &array[ left ] );
19.       position = left;
20.     }
21.   }
22. }
```

小

大

小

大

```
1.  int partition( int * const array, int left, int right )
2.  {
3.    int position = left;
4.    while ( true )
5.    {
6.      while ( array[ position ] <= array[ right ] && position != right )  right--;
7.      if ( position == right ) return position;
8.      if ( array[ position ] > array[ right ])
9.      {
10.       swap( &array[ position ], &array[ right ] );
11.       position = right;
12.     }
13.
14.     while ( array[ left ] <= array[ position ] && left != position )  left++;
15.     if ( position == left ) return position;
16.     if ( array[ left ] > array[ position ] )
17.     {
18.       swap( &array[ position ], &array[ left ] );
19.       position = left;
20.     }
21.   }
22. }
```

小

大

```
1.  // recursive function to sort array
2.  void quicksort( int * const array, int first, int last )
3.  {
4.     int currentLocation;
5.
6.     if ( first >= last )
7.        return;
8.
9.     // place an element
10.    currentLocation = partition( array, first, last );
11.    // sort left side
12.    quicksort( array, first, currentLocation - 1 );
13.    // sort right side
14.    quicksort( array, currentLocation + 1, last );
15. } // end function quicksort
```