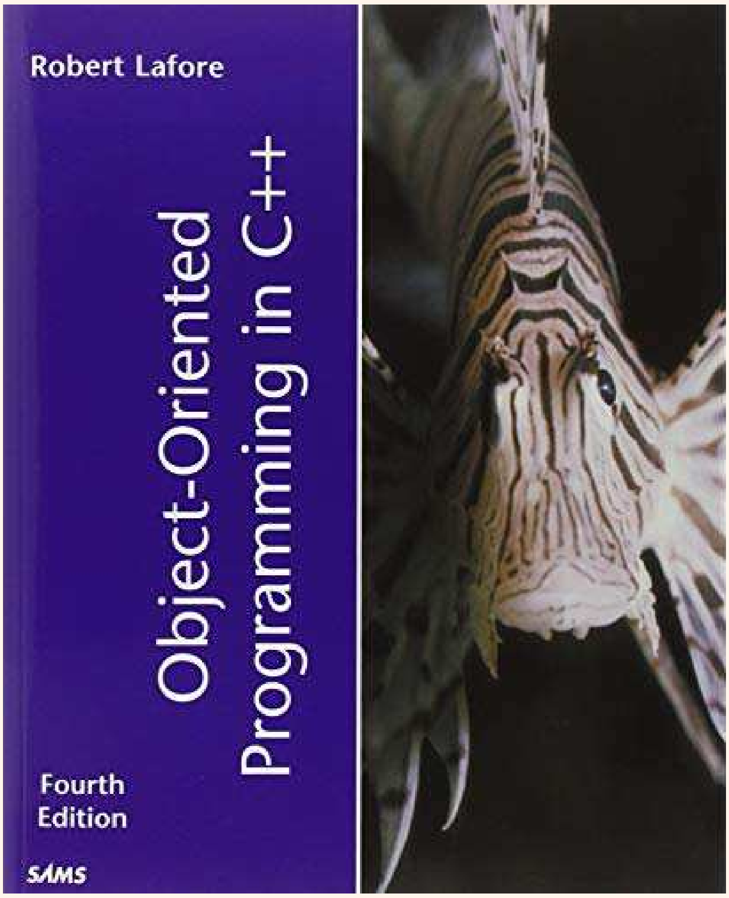


Chapter 10

Pointers

Animated Version

Topics

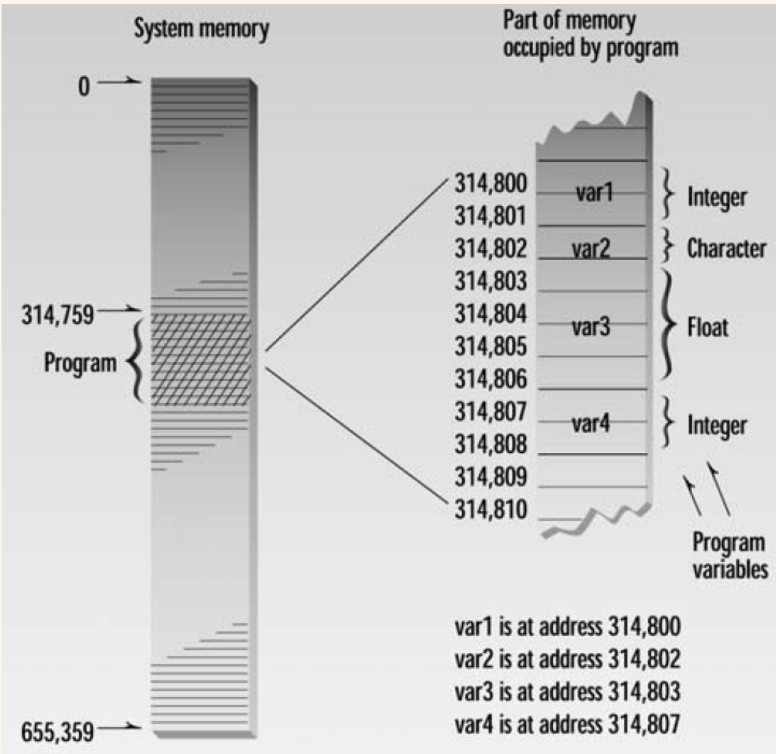
- Addresses and Pointers
- The Address-of Operator &
- Pointers and Arrays
- Pointers and Functions
- Pointers and C-Type Strings
- Memory Management: `new` and `delete`
- Pointers to Objects
- A Linked List Example
- Pointers to Pointers
- A Parsing Example
- Simulation: A Horse Race
- UML State Diagrams
- Debugging Pointers

Introduction

- What are pointers for?
  - Accessing array elements, arrays and strings to functions
  - Passing arguments to a function when the function needs to modify the original argument
  - Obtaining memory from the system
  - Creating data structures such as linked lists
- Java has references, which are sort of watered-down pointers.
- Essential tool for increasing the power of C++: creation of linked lists and binary trees.
- Several key features of C++, such as virtual functions, the new operator, and the this pointer require the use of pointers.

Addresses and Pointers

- key concept: Every byte in the computer's memory has an address.



The Address-of Operator &

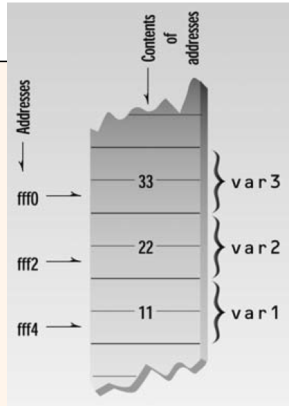
- The << insertion operator interprets the addresses in hexadecimal arithmetic

- Each address differs from the next by exactly 2 bytes. That's because integers occupy 2 bytes of memory (on a 16-bit system).

```
Output:
0x8f4ffff4    <------    address of var1
0x8f4ffff2    <------    address of var2
0x8f4ffff0    <------    address of var3
```

```
// varaddr.cpp
// addresses of va
#include <iostream>
using namespace std;

int main()
   {
   int var1 = 11;  //define and initialize
   int var2 = 22;  //three variables
   int var3 = 33;

   cout << &var1 << endl //print the addresses
        << &var2 << endl //of these variables
        << &var3 << endl;
   return 0;
   }
```

- addresses appear in descending order because local variables are stored on the stack.
- If we had used global variables, they would have ascending addresses, since global variables are stored on the heap, which grows upward.



The Address-of Operator & (2)

- Pointer Variables or pointer:
  - A variable that holds an address value is called a pointer variable, or simply a pointer.
- The asterisk means *pointer to*.
  - pointer to int

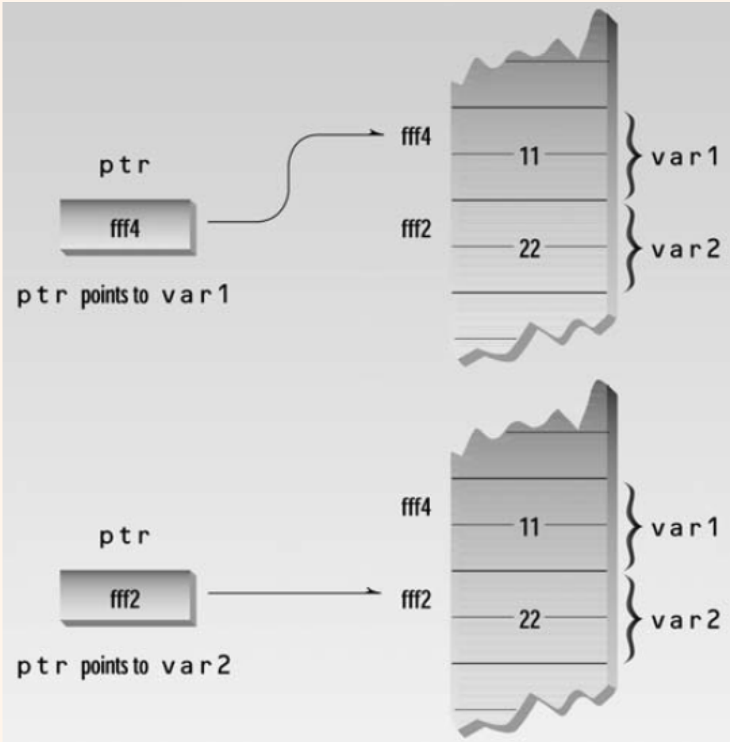
```
Output:
0x8f51fff4    <------    address of var1
0x8f51fff2    <------    address of var2

0x8f51fff4    <------    ptr set to address of var1
0x8f51fff2    <------    ptr set to address of var2
```

```
// ptrvar.cpp
// pointers (addres
#include <iostream>
using namespace std;

int main()
   {
   int var1 = 11; //two integer variables
   int var2 = 22;

   cout << &var1 << endl
        << &var2 << endl << endl;
          //print addresses of variables
   int* ptr;          //pointer to integers

   ptr = &var1;  //pointer points to var1
   cout << ptr << endl;
       //print pointer value

   ptr = &var2;  //pointer points to var2
   cout << ptr << endl; //print pointer value
   return 0;
   }
```

```
char* cptr;            // pointer to char
int* iptr;             // pointer to int
float* fptr;           // pointer to float
Distance* distptr;     // pointer to user-defined Distance class
char* ptr1, * ptr2, * ptr3;  // three variables of type char*
char *ptr1, *ptr2, *ptr3;   // three variables of type char*
```

The Address-of Operator & (3)

- Pointers Must Have a Value
- A pointer can hold the address of any variable of the correct type
- Rogue pointer values can result in system crashes and are difficult to debug, since the compiler gives no warning.
- The moral: Make sure you give every pointer variable a valid address value before using it.

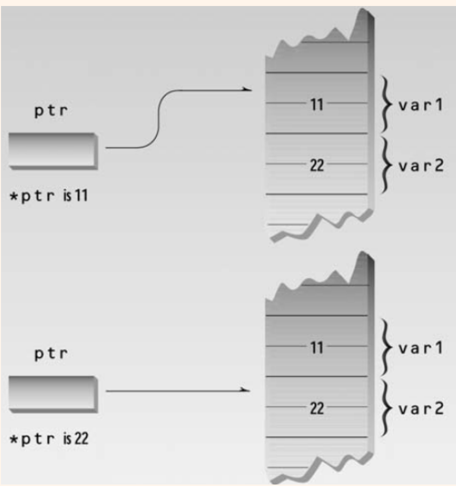


The Address-of Operator & (4)

- Accessing the Variable Pointed To
  - *ptr
    - *dereference / indirection / contents of* operator.
    - the value of the variable pointed to by ptr

```
// ptracc.cpp
// accessing the variable pointed to
#include <iostream>
using namespace std;

int main()
   {
   int var1 = 11;   //two integer variables
   int var2 = 22;

   int* ptr;             //pointer to integers

   ptr = &var1;          //pointer points to var1
   cout << *ptr << endl;//print contents of pointer (11)

   ptr = &var2;          //pointer points to var2
   cout << *ptr << endl;//print contents of pointer (22)
   return 0;
   }
```

Output:
11
22



The Address-of Operator & (5)

- Accessing the Variable Pointed To
  - asterisk used as the dereference operator has a different meaning than the asterisk used to declare pointer variables.
    - The dereference operator precedes the variable and means *value of the variable pointed to by*.
    - The asterisk used in a declaration means *pointer to*.

```
// ptrto.cpp
// other access using pointers
#include <iostream>
using namespace std;

int main()
   {
   int var1, var2;          //two integer variables
   int* ptr;                //pointer to integers

   ptr = &var1;         //set pointer to address of var1
   *ptr = 37;               //same as var1=37
   var2 = *ptr;             //same as var2=var1

   cout << var2 << endl;    //verify var2 is 37
   return 0;
   }
```

Output:
11
22

```
int v;          //defines variable v of type int
int* p;         //defines p as a pointer to int
p = &v;         //assigns address of variable v to pointer p
v = 3;          //assigns 3 to v
*p = 3;         //also assigns 3 to v
```

- Using the dereference operator to access the value stored in an address is called *indirect addressing*, or sometimes *dereferencing*, the pointer.

The Address-of Operator & (6)

- Pointer to `void`
  - general-purpose pointer that can point to any data type.
  - Use: such as passing pointers to functions that operate independently of the data type pointed to.
  - Can cast one pointer of one type to another: Not recommended

```
ptrint = reinterpret_cast<int*>(flovar);
ptrflo = reinterpret_cast<float*>(intvar);
```

```
// ptrvoid.cpp
// pointers to type void
#include <iostream>
using namespace std;

int main()
   {
   int intvar;                 //integer variable
   float flovar;               //float variable

   int* ptrint;                //define pointer to int
   float* ptrflo;              //define pointer to float
   void* ptrvoid;              //define pointer to void

   ptrint = &intvar;           //ok, int* to int*
// ptrint = &flovar;           //error, float* to int*
// ptrflo = &intvar;           //error, int* to float*
   ptrflo = &flovar;           //ok, float* to float*

   ptrvoid = &intvar;          //ok, int* to void*
   ptrvoid = &flovar;          //ok, float* to void*
   return 0;
   }
```

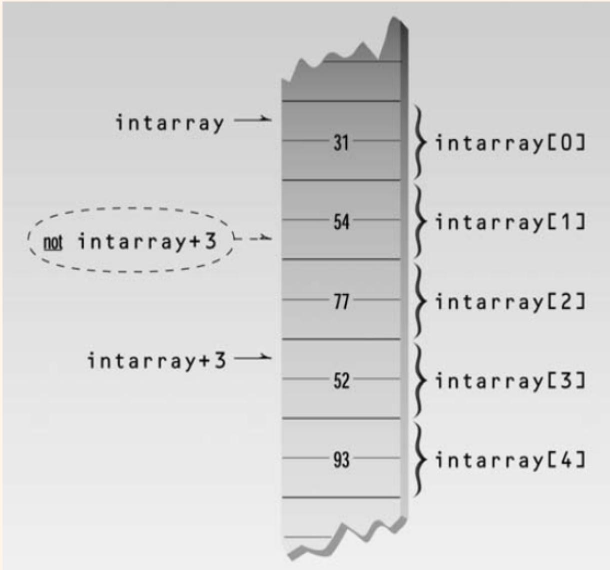
Pointers and Arrays

```
// arrnote.cpp
// array accessed with array notation
#include <iostream>
using namespace std;

int main()
   {                                          //array
   int intarray[5] = { 31, 54, 77, 52, 93 };

   for(int j=0; j<5; j++)          //for each element,
      cout << intarray[j] << endl;     //print value
   return 0;
   }
```

```
31
54
77
52
93
```

```
// ptrnote.cpp
// array accessed with pointer notation
#include <iostream>
using namespace std;

int main()
   {                                          //array
   int intarray[5] = { 31, 54, 77, 52, 93 };

   for(int j=0; j<5; j++)          //for each element,
      cout << *(intarray+j) << endl;   //print value
   return 0;
   }
```

- why a pointer declaration must include the type of the variable pointed to?



Pointers and Arrays (2)

```
// ptrinc.cpp
// array accessed with pointer
#include <iostream>
using namespace std;

int main()
   {
   int intarray[] = { 31, 54, 77, 52, 93 }; //array
   int* ptrint;                         //pointer to int
   ptrint = intarray;               //points to intarray

   for(int j=0; j<5; j++)            //for each element,
      cout << *(ptrint++) << endl;     //print value
   return 0;
   }
```

```
31
54
77
52
93
```

- Pointer Constants and Pointer Variables

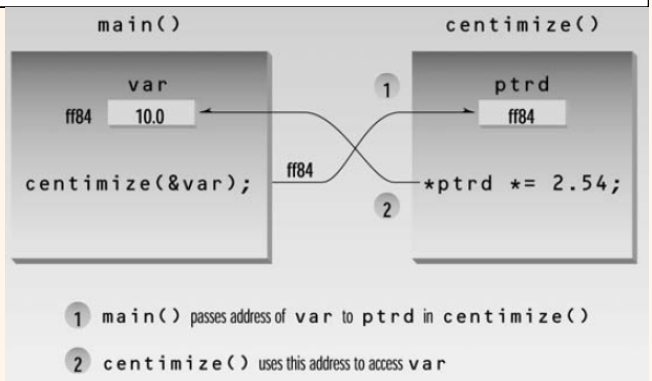
Pointers and Functions

- Passing Simple Variables

```
// passref.cpp
// arguments passed by reference
#include <iostream>
using namespace std;

int main()
   {
   void centimize(double&);   //prototype

   double var = 10.0;        //var has value of 10
    inches
   cout << "var = " << var << " inches" << endl;

   centimize(var);          //change var to centimeters
   cout << "var = " << var << " centimeters" << endl;
   return 0;
   }
//----------------------------------------------------
void centimize(double& v)
   {
   v *= 2.54;                //v is the same as var
   }
```

```
var = 10 inches
var = 25.4 centimeters
```

```
// passptr.cpp
// arguments passed by pointer
#include <iostream>
using namespace std;

int main()
   {
   void centimize(double*);    //prototype

   double var = 10.0;       //var has value of 10 inches
   cout << "var = " << var << " inches" << endl;

   centimize(&var);         //change var to centimeters
   cout << "var = " << var << " centimeters" << endl;
   return 0;
   }
//----------------------------------------------------
void centimize(double* ptrd)
   {
   *ptrd *= 2.54;            //*ptrd is the same as var
   }
```

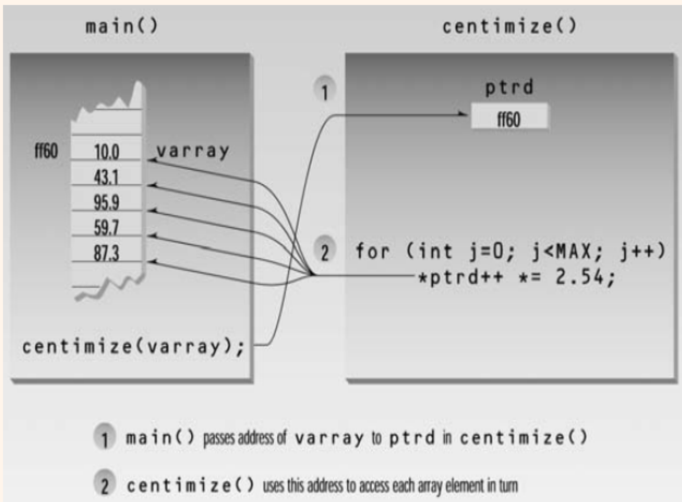


- *A reference is an alias for the original variable, while a pointer is the address of the variable.*

Pointers and Functions (2)

- Passing Arrays

```
// passarr.cpp
// array passed by pointer
#include <iostream>
using namespace std;
const int MAX = 5;        //number of array elements

int main()
   {
   void centimize(double*);  //prototype

   double varray[MAX] = { 10.0, 43.1, 95.9, 59.7, 87.3 };

   centimize(varray);//change elements of varray to cm

   for(int j=0; j<MAX; j++)//display new array values
      cout << "varray[" << j << "]="
           << varray[j] << " centimeters" << endl;
   return 0;
   }
//----------------------------------------------------
void centimize(double* ptrd)
   {
   for(int j=0; j<MAX; j++)
      *ptrd++ *= 2.54;//ptrd points to elements of varray
   }
```




```
varray[0]=25.4 centimeters
varray[1]=109.474 centimeters
varray[2]=243.586 centimeters
varray[3]=151.638 centimeters
varray[4]=221.742 centimeters
```

Pointers and Functions (3)

- Sorting Array Elements

```
// ptrorder.cpp
// orders two arguments using pointers
#include <iostream>
using namespace std;

int main()
   {
   void order(int*, int*);         //prototype

   int n1=99, n2=11;        //one pair ordered, one not
   int n3=22, n4=88;

   order(&n1, &n2);      //order each pair of numbers
   order(&n3, &n4);

   cout << "n1=" << n1 << endl;//print out all numbers
   cout << "n2=" << n2 << endl;
   cout << "n3=" << n3 << endl;
   cout << "n4=" << n4 << endl;
   return 0;
   }
//----------------------------------------------------
void order(int* numb1, int* numb2)//orders two numbers
   {
   if(*numb1 > *numb2)          //if 1st larger than
    2nd,
      {
      int temp = *numb1;      //swap them
      *numb1 = *numb2;
      *numb2 = temp;
      }
   }
```

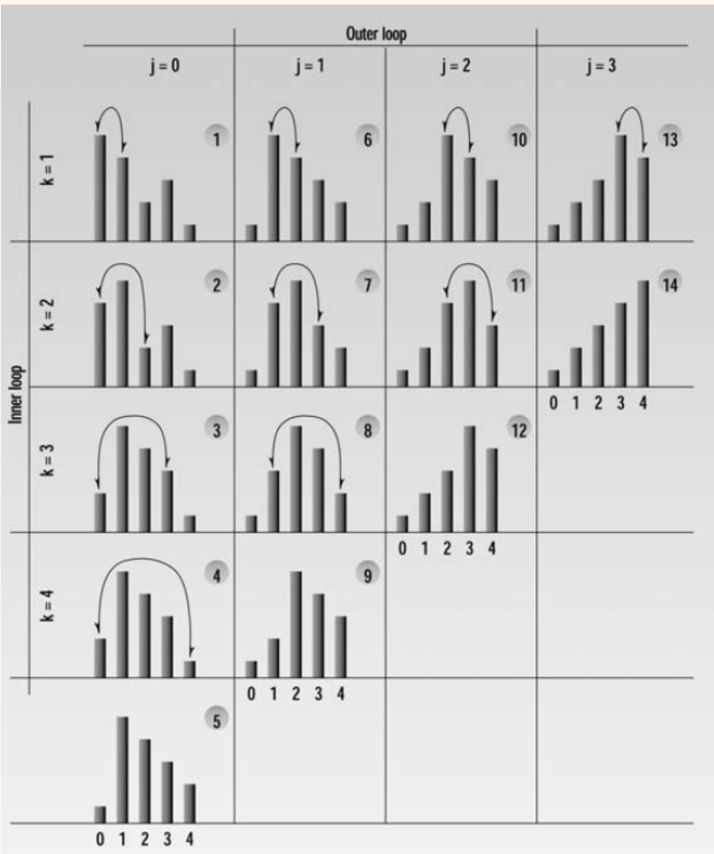
```
n1=11  <------  this and
n2=99  <------  this are swapped, since they weren't in order
n3=22  <------  this and
n4=88  <------  this are not swapped, since they were in order
```

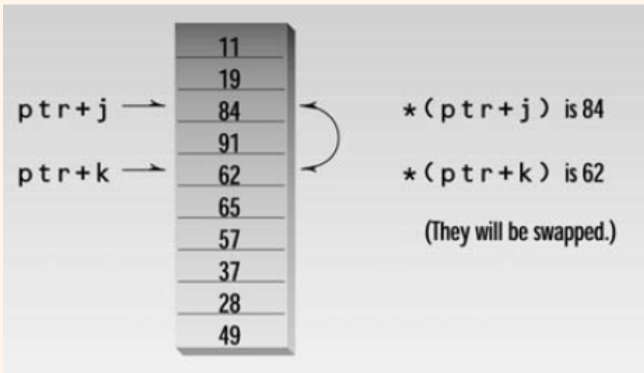
```
// ptrsort.cpp
// sorts an array using pointers
#include <iostream>
using namespace std;
int main()
   {
   void bsort(int*, int);        //prototype
   const int N = 10;             //array size
                                 //test array
   int arr[N] = {37,84,62,91,11,65,57,28,19,49};

   bsort(arr, N);                //sort the array

   for(int j=0; j<N; j++)        //print out sorted array
      cout << arr[j] << " ";
   cout << endl;
   return 0;
   }
//----------------------------------------------------
void bsort(int* ptr, int n)
   {
   void order(int*, int*);       //prototype
   int j, k;                     //indexes to array

   for(j=0; j<n-1; j++)          //outer loop
      for(k=j+1; k<n; k++) //inner loop starts at outer
   order(ptr+j, ptr+k);     //order the pointer contents
   }
//----------------------------------------------------
void order(int* numb1, int* numb2)  //orders two numbers
   {
   if(*numb1 > *numb2)      //if 1st larger than 2nd,
      {
      int temp = *numb1;         //swap them
      *numb1 = *numb2;
      *numb2 = temp;
      }
   }
```

```
11 19 28 37 49 57 62 65 84 91
```

Pointers and Functions (4)

- The Bubble Sort





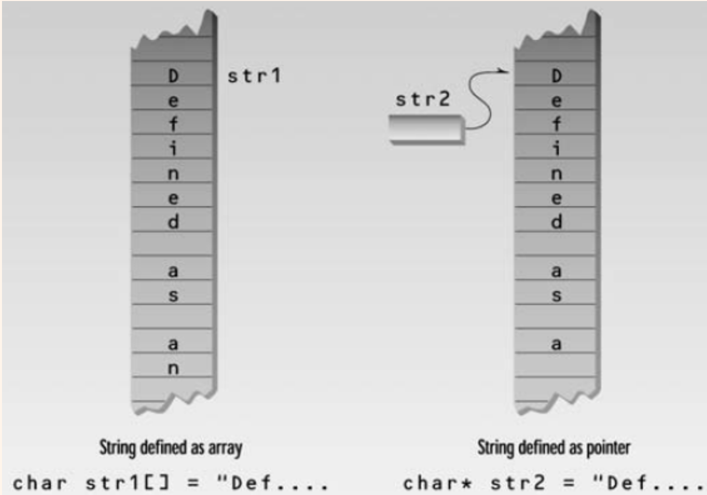
Pointers and C-Type Strings

- Pointers to String Constants

```
// twostr.cpp
// strings defined using array and pointer notation
#include <iostream>
using namespace std;

int main()
   {
   char str1[] = "Defined as an array";
   char* str2 = "Defined as a pointer";

   cout << str1 << endl;     // display both strings
   cout << str2 << endl;

// str1++;        // can't do this; str1 is a constant
   str2++;        // this is OK, str2 is a pointer

   cout << str2 << endl;// now str2 starts "efined..."
   return 0;
   }
```

```
Defined as an array
Defined as a pointer
efined as a pointer
```
← following str2++ ('D' is gone)

str1 is an address—that is, a pointer constant—while str2 is a pointer variable. So str2 can be changed, while str1 cannot.



- *C-type strings are simply arrays of type char. Thus pointer notation can be applied to the characters in strings, just as it can to the elements of any array.*

Pointers and C-Type Strings (2)

- Strings as Function Arguments

```
// ptrstr.cpp
// displays a string with pointer notation
#include <iostream>
using namespace std;

int main()
   {
   void dispstr(char*);      //prototype
   char str[] = "Idle people have the least leisure.";

   dispstr(str);             //display the string
   return 0;
   }
//--------------------------------------------------
void dispstr(char* ps)
   {
   while( *ps )              //until null character,
      cout << *ps++;         //print characters
   cout << endl;
   }
```

- The `const` Modifier and Pointers

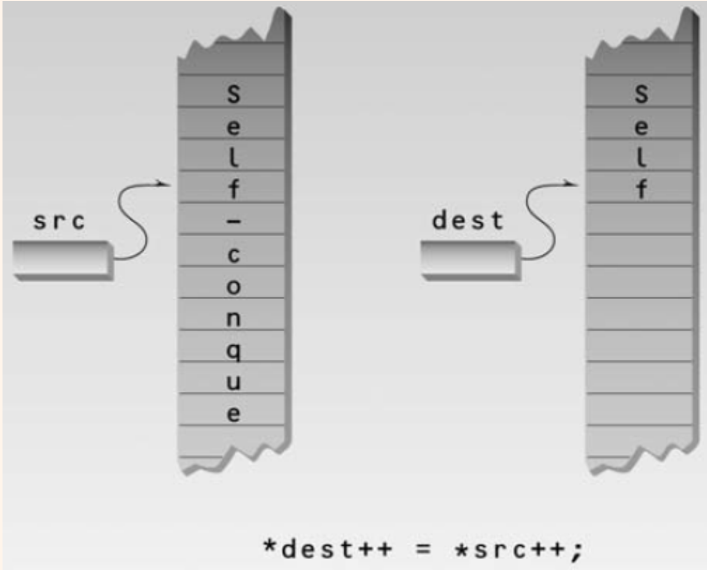
```
const int* cptrInt;  //cptrInt is a pointer to constant int
int* const ptrcInt;  //ptrcInt is a constant pointer to int
```

- *The loop cycles until it finds the null character ('\0') at the end of the string.*

Pointers and C-Type Strings (3)

- Copying a String Using Pointers

```
// copystr.cpp
// copies one string to another with pointers
#include <iostream>
using namespace std;

int main()
   {
   void copystr(char*, const char*);  //prototype
   char* str1 = "Self-conquest is the greatest
    victory.";
   char str2[80];               //empty string

   copystr(str2, str1);         //copy str1 to str2
   cout << str2 << endl;        //display str2
   return 0;
   }
//--------------------------------------------------------------
void copystr(char* dest, const char* src)
   {
   while( *src )             //until null character,
      *dest++ = *src++;      //copy chars from src to dest
   *dest = '\0';                //terminate dest
   }
```



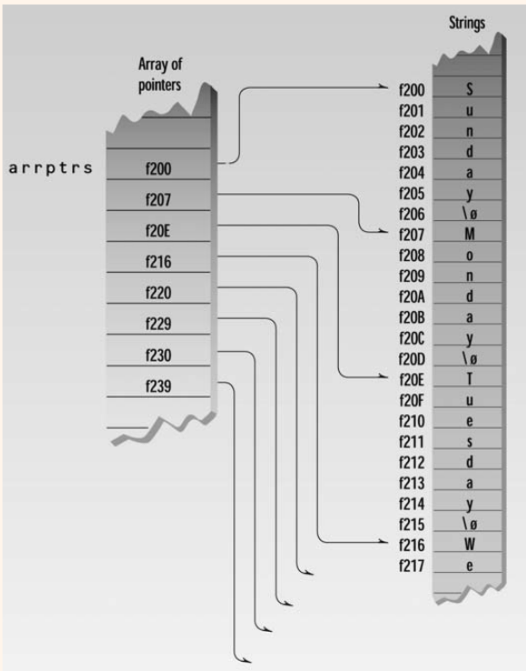
Pointers and C-Type Strings (4)

- Arrays of Pointers to Strings

Overcomes the disadvantage to using an array of strings, in that the subarrays that hold the strings must all be the same length, so space is wasted when strings are shorter than the length of the subarrays.

```
// ptrtostr.cpp
// an array of pointers to strings
#include <iostream>
using namespace std;
const int DAYS = 7;     //number of pointers in array

int main()
   {                    //array of pointers to char
   char* arrptrs[DAYS] = { "Sunday", "Monday",
    "Tuesday",
                 "Wednesday", "Thursday",
                 "Friday", "Saturday"  };

   for(int j=0; j<DAYS; j++)    //display every string
      cout << arrptrs[j] << endl;
   return 0;
   }
```

```
Sunday
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
```



Memory Management: new and delete

- Drawback of using Arrays for data storage
  - We must know at the time we write the program how big the array will be.

```
int arr1[100];
cin >> size;       // get size from user
int arr[size];     // error; array size must be a constant
```

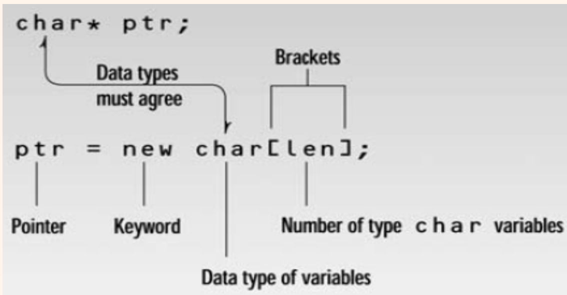
  - We need to define an array sized to hold the largest string we expect, but this wastes memory.
- The new Operator
  - obtains memory from the operating system and returns a pointer to its starting point
- The delete Operator
  - Returns memory to the operating system. Otherwise, reserving many chunks of memory using new will eventually reserve all the available memory and the system will crash.

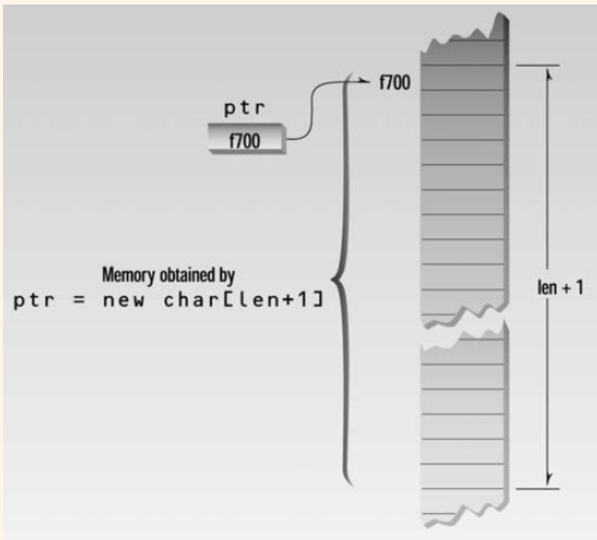
Memory Management: new and delete

- The new and delete Operator

```
// newintro.cpp
// introduces operator new
#include <iostream>
#include <cstring>              //for strlen
using namespace std;

int main()
   {
   char* str = "Idle hands are the devil's workshop.";
   int len = strlen(str);      //get length of str

   char* ptr;                  //make a pointer to char
   ptr = new char[len+1];
                   //set aside memory: string + '\0'

   strcpy(ptr, str); //copy str to new memory area ptr

   cout << "ptr=" << ptr << endl;
                   //show that ptr is now in str

   delete[] ptr;               //release ptr's memory
   return 0;
   }
```

```
ptr=Idle hands are the devil's workshop.
```





new = malloc() + returns a pointer to the appropriate data type

Memory Management: `new` and `delete` (2)

- A String Class Using `new`

```
// newstr.cpp
// using new to get memory for strings
#include <iostream>
#include <cstring>        //for strcpy(), etc
using namespace std;
////////////////////////////////////////////////////////////////
class String              //user-defined string type
   {
   private:
      char* str;          //pointer to string
   public:
      String(char* s)     //constructor, one arg
         {
         int length = strlen(s);//length of string argument
         str = new char[length+1]; //get memory
         strcpy(str, s);            //copy argument to it
         }
      ~String() {                    //destructor
          cout << "Deleting str\n";
         delete[] str;              //release memory
         }
      void display() {               //display the String
          cout << str << endl;
         }
   };
///////////////////////////////////////////////////////////////
int main()
   {                          //uses 1-arg constructor
   String s1 = "Who knows nothing doubts nothing.";
   cout << "s1=";             //display string
   s1.display();
   return 0;
   }
```

- Actually, memory is automatically returned when program terminates.
- But, if a function uses a local variable as a pointer to this memory, the pointer will be destroyed when the function terminates, but the memory will be left as an orphan, taking up space that is inaccessible to the rest of the program.
- Thus it is always good practice to delete memory when done.

Pointers to Objects

```
// englptr.cpp
// accessing member functions by pointer
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class Distance               //English Distance class
   {
   private:
      int feet;
      float inches;
   public:
      void getdist()         //get length from user
         {
         cout << "\nEnter feet: ";  cin >> feet;
         cout << "Enter inches: ";  cin >> inches;
         }
      void showdist()        //display distance
         { cout << feet << "\'-" << inches << '\"'; }
   };
////////////////////////////////////////////////////////////////
int main()
   {
   Distance dist;    //define a named Distance object
   dist.getdist();           //access object members
   dist.showdist();          //   with dot operator

   Distance* distptr;        //pointer to Distance
   distptr = new Distance;
                     //points to new Distance object
   distptr->getdist();       //access object members
   distptr->showdist();      //   with -> operator
   cout << endl;
   return 0;
   }
```

- Referring to Members

```
distptr.getdist();
        // won't work; distptr is not a variable

(*distptr).getdist();  // ok but inelegant

distptr->getdist();    // better approach
```

```
Enter feet: 10 ←—— this object uses the dot operator
Enter inches: 6.25
10'-6.25"

Enter feet: 6  ←—— this object uses the -> operator
Enter inches: 4.75
6'-4.75"
```

Pointers to Objects (2)

```
// englref.cpp
// dereferencing the pointer returned by new
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////
class Distance                          // English Distance
    class
   {
   private:
      int feet;
      float inches;
   public:
      void getdist()                    // get length from
    user
         {
         cout << "\nEnter feet: ";  cin >> feet;
         cout << "Enter inches: ";  cin >> inches;
         }
      void showdist()                   // display distance
         { cout << feet << "\'-" << inches << '\"'; }
   };
////////////////////////////////////////////////////////
int main()
   {
   Distance& dist = *(new Distance);
           // create Distance object
           // alias is "dist"
   dist.getdist();                  // access object members
   dist.showdist();                 //    with dot operator
   cout << endl;
   return 0;
   }
```

- Another Approach to new

  `new Distance`

  –returns a pointer to a memory area large enough for a Distance object

  `*(new Distance)`

  –Rrefer to the original object

- Now can refer to members of `dist` using the *dot* membership operator, rather than `->`

Pointers to Objects (3)

- An Array of Pointers to Objects

```
// ptrobjs.cpp
// array of pointers to objects
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////
class person                          //class of persons
   {
   protected:
      char name[40];                  //person's name
   public:
      void setName()                  //set the name
         {
         cout << "Enter name: ";
         cin >> name;
         }
      void printName()                //get the name
         {
         cout << "\n   Name is: " << name;
         }
   };
////////////////////////////////////////////////////////
```

```
int main()
   {
   person* persPtr[100];//array of pointers to persons
   int n = 0;               //number of persons in array
   char choice;
   do                           //put persons in array
      {
      persPtr[n] = new person;     //make new object
      persPtr[n]->setName();      //set person's name
      n++;                         //count new person
      cout << "Enter another (y/n)? ";//enter another
      cin >> choice;                     //person?
      }
   while( choice=='y' );               //quit on 'n'

   for(int j=0; j<n; j++)               //print names of
      {                                 //all persons
      cout << "\nPerson number " << j+1;
      persPtr[j]->printName();
      }
   cout << endl;
   return 0;
   }  //end main()
```

```
Enter name: Stroustrup        ← user enters names
Enter another (y/n)? y
Enter name: Ritchie
Enter another (y/n)? y
Enter name: Kernighan
Enter another (y/n)? n
Person number 1          ← program displays all names stored
   Name is: Stroustrup
Person number 2
   Name is: Ritchie
Person number 3
   Name is: Kernighan
```

Other Topics

- A Linked List Example
- Self-Containing Classes
- Pointers to Pointers
- A Parsing Example
- Simulation: A Horse Race
- UML State Diagrams
- Debugging Pointers

Summary (1)

- We've learned that everything in the computer's memory has an address, and that addresses are pointer constants. We can find the addresses of variables using the address-of operator &.
- Pointers are variables that hold address values.
  - Pointers are defined using an asterisk (*) to mean pointer to.
  - A data type is always included in pointer definitions (except void*), since the compiler must know what is being pointed to, so that it can perform arithmetic correctly on the pointer.
  - We access the thing pointed to using the asterisk in a different way, as the dereference operator, meaning contents of the variable pointed to by.
- The special type void* means a pointer to any type. It's used in certain difficult situations where the same pointer must hold addresses of different types.
- Array elements can be accessed using array notation with brackets or pointer notation with an asterisk. Like other addresses, the address of an array is a constant, but it can be assigned to a variable, which can be incremented and changed in other ways.

Summary (2)

- When the address of a variable is passed to a function, the function can work with the original variable. (This is not true when arguments are passed by value.) In this respect passing by pointer offers the same benefits as passing by reference, although pointer arguments must be dereferenced or accessed using the dereference operator. However, pointers offer more flexibility in some cases.
- A string constant can be defined as an array or as a pointer.
  - The pointer approach may be more flexible, but there is a danger that the pointer value will be corrupted.
  - Strings, being arrays of type char, are commonly passed to functions and accessed using pointers.
- The *new* operator obtains a specified amount of memory from the system and returns a pointer to the memory. This operator is used to create variables and data structures during program execution. The *delete* operator releases memory obtained with new.

Summary (3)

- When a pointer points to an object, members of the object's class can be accessed using the access operator -> . The same syntax is used to access structure members.
- Classes and structures may contain data members that are pointers to their own type. This permits the creation of complex data structures such as linked lists.
- There can be pointers to pointers. These variables are defined using the double asterisk; for example, int** pptr .
- Multiplicity in UML class diagrams shows the number of objects involved in an association.
- UML state diagrams show how a particular object's situation changes over time. States are represented by rectangles with rounded corners, and transitions between states are represented by directed lines.