Arrays

- Array
  - Consecutive group of memory locations
  - Same name and type (int, char, etc.)

- To refer to an element
  - Specify array name and position number (index)
  - Format: arrayname[ position number ]
  - First element at position 0

- N-element array c
  \[ c[0], c[1], \ldots, c[n-1] \]
  - Nth element as position N-1
Declaring Arrays

• When declaring arrays, specify
  – Name
  – Type of array
  • Any data type
  – Number of elements
  – `type arrayName [ arraySize ];`
    ```
    int c[ 10 ];  // array of 10 integers
    float d[ 3284 ];  // array of 3284 floats
    ```

• Declaring multiple arrays of same type
  – Use comma separated list, like regular variables
    ```
    int b[ 100 ], x[ 27 ];
    ```

Examples Using Arrays

• Initializing arrays
  – For loop
    • Set each element
  – Initializer list
    • Specify each element when array declared
      ```
      int n[ 5 ] = { 1, 2, 3, 4, 5 };
      ```
    • If not enough initializers, rightmost elements 0
    • If too many, syntax error
  – To set every element to 0
    ```
    int n[ 5 ] = { 0 };
    ```
  – If array size omitted, initializers determine size
    ```
    int n[] = { 1, 2, 3, 4, 5 };
    ```
    • 5 initializers, therefore 5 element array

• `static int array[3];` ??
Examples Using Arrays

• Strings
  – Arrays of characters
  – All strings end with null (\'\0\')
  – Examples
    * char string1[] = "hello";
      - Null character implicitly added
      - string1 has 6 elements
    * char string1[] = { 'h', 'e', 'l', 'l', 'o', '\0' };
      – Subscripting is the same
        String1[ 0 ] is 'h'
        string1[ 2 ] is 'l'

Examples Using Arrays

• Input from keyboard
  char string2[ 10 ];
  cin >> string2;
  – Puts user input in string
    • Stops at first whitespace character
    • Adds null character
  – If too much text entered, data written beyond array
    • We want to avoid this

• Printing strings
  cout << string2 << endl;
  • Does not work for other array types
  • Characters printed until null found
Passing Arrays to Functions

• Specify name without brackets
  – To pass array `myArray` to `myFunction`
    ```
    int myArray[ 24 ];
    myFunction( myArray, 24 );
    ```
  – Array size usually passed, but not required
    • Useful to iterate over all elements

• Arrays passed-by-reference
  – Functions can modify original array data
  – Value of name of array is address of first element
    • Function knows where the array is stored
    • Can change original memory locations

Passing Arrays to Functions

• Functions taking arrays
  – Function prototype
    • `void modifyArray( int b[], int arraySize );`
    • `void modifyArray( int [], int );`
      – Names optional in prototype
        • Both take an integer array and a single integer
  – No need for array size between brackets
    • Ignored by compiler
  – If declare array parameter as `const`
    • Cannot be modified (compiler error)
    • `void doNotModify( const int [] );`
Sorting Arrays

- **Example:**
  - Go left to right, and exchange elements as necessary
    - One pass for each element
  - Original: 3 4 2 7 6
  - Pass 1: 3 2 4 6 7 (elements exchanged)
  - Pass 2: 2 3 4 6 7
  - Pass 3: 2 3 4 6 7 (no changes needed)
  - Pass 4: 2 3 4 6 7
  - Pass 5: 2 3 4 6 7
  - Small elements "bubble" to the top (like 2 in this example)

Multiple-Subscripted Arrays

- **Multiple subscripts**
  - \( a[i][j] \)
  - Tables with rows and columns
  - Specify row, then column
  - "Array of arrays"
    - \( a[0] \) is an array of 4 elements
    - \( a[0][0] \) is the first element of that array

- **To initialize**
  - Default of 0
  - Initializers grouped by row in braces

```c
int b[2][2] = {{1, 2}, {3, 4}};
int b[2][2] = {{1}, {3, 4}};
```
Pointer Variable Declarations and Initialization

- Pointer variables
  - Contain memory addresses as values
  - Normally, variable contains specific value (direct reference)
  - Pointers contain address of variable that has specific value (indirect reference)
- Indirection
  - Referencing value through pointer
- Pointer declarations
  - * indicates variable is pointer
    ```c
    int *myPtr;
    ```
    declares pointer to `int`, pointer of type `int *`
  - Multiple pointers require multiple asterisks
    ```c
    int *myPtr1, *myPtr2;
    ```

Pointer Operators

- & (address operator)
  - Returns memory address of its operand
  - Example
    ```c
    int y = 5;
    int *yPtr;
    yPtr = &y;
    ```
- `yPtr` "points to" `y`
- * - indirection/dereferencing operator
- `*yPtr` returns `y`
  - Dereferenced pointer is lvalue
    ```c
    *yPtr = 9
    ```
Calling Functions by Reference

• 3 ways to pass arguments to function
  – Pass-by-value
  – Pass-by-reference with reference arguments
  – Pass-by-reference with pointer arguments

• Arguments passed to function using reference arguments
  – Modify original values of arguments
  – More than one value “returned”

  \[
  \text{int } \text{Cube}(\text{int } \ast x) \{ \quad \ldots \}
  \]

  Function call:

  \[
  \text{Cube}(\&a)
  \]

Using \texttt{const} with Pointers

• \texttt{const} qualifier
  – Value of variable should not be modified
  – \texttt{const} used when function does not need to change a variable
  – Principle of least privilege

• \texttt{const} pointers
  – Always point to same memory location
  – Default for array name
  – Must be initialized when declared

• Four ways to pass pointer to function
  – Nonconstant pointer to nonconstant data
    • Highest amount of access
  – Nonconstant pointer to constant data
  – Constant pointer to nonconstant data
  – Constant pointer to constant data
    • Least amount of access
```cpp
int main()
{
    int x, y;
    int * const ptr = &x;

    *ptr = 7; // allowed: *ptr is not const
    ptr = &y;  // error: ptr is const; cannot assign new address

    return 0;  // indicates successful termination
}
```

Line 15 generates compiler error by attempting to assign new address to constant pointer.

```
int main()
{
    int x = 5, y;
    const int *const ptr = &x;

    cout << *ptr << endl; // cannot modify x (pointed to by ptr)
    *ptr = 7; // error: *ptr is const
    ptr = &y;  // error: ptr is const; cannot assign new address

    return 0; // indicates successful termination
}
```
Pointer Expressions and Pointer Arithmetic

Pointer arithmetic
- Increment/decrement pointer (++ or --)
- Add/subtract an integer to/from a pointer (+ or +=, - or -=)
- Pointers may be subtracted from each other
- Pointer arithmetic meaningless unless performed on pointer to array

- 5 element int array on a machine using 4 byte ints
  - vPtr points to first element v[0], which is at location 3000
    vPtr = 3000
  - vPtr += 2; sets vPtr to 3008
    vPtr points to v[2]

- Subtracting pointers
  - Returns number of elements between two addresses
    vPtr2 = v[2];
    vPtr = v[0];
    vPtr2 - vPtr == 2

- Pointer assignment
  - Pointer can be assigned to another pointer if both of same type
  - If not same type, cast operator must be used
  - Exception: pointer to void (type void *)
    - Generic pointer, represents any type
    - No casting needed to convert pointer to void pointer
    - void pointers cannot be dereferenced
Pointer Expressions and Pointer Arithmetic

- **Pointer comparison**
  - Use equality and relational operators
  - Comparisons meaningless unless pointers point to members of same array
  - Compare addresses stored in pointers
    - Example: could show that one pointer points to higher numbered element of array than other pointer
  - Common use to determine whether pointer is 0 (does not point to anything)

Relationship Between Pointers and Arrays

- **Arrays and pointers closely related**
  - Array name like constant pointer
  - Pointers can do array subscripting operations

- **Accessing array elements with pointers**
  - Element $b[\ n\ ]$ can be accessed by $*(bPtr + n)$
    - Called pointer/offset notation
  - Addresses
    - $&b[\ 3\ ]$ same as $bPtr + 3$
  - Array name can be treated as pointer
    - $b[\ 3\ ]$ same as $*(b + 3)$
  - Pointers can be subscripted (pointer/subscript notation)
    - $bPtr[\ 3\ ]$ same as $b[\ 3\ ]$
Arrays of Pointers

- Arrays can contain pointers
  - Commonly used to store array of strings
    ```c
    ```
  - Each element of `suit` points to `char *` (a string)
  - Array does not store strings, only pointers to strings

```c
suit[0] = "Hearts";
suit[1] = "Diamonds";
suit[2] = "Clubs";
suit[3] = "Spades";
```

- `suit` array has fixed size, but strings can be of any size

Function Pointers

- Calling functions using pointers
  - Assume parameter:
    ```c
    bool (*compare)(int, int)
    ```
  - Execute function with either
    ```c
    (*compare)(int1, int2)
    ```
    - Dereference pointer to function to execute
  OR
    ```c
    compare(int1, int2)
    ```
    - Could be confusing
      ```c
      User may think compare name of actual function in program
      ```

© 2003 Prentice Hall, Inc. All rights reserved.
// Fig. 5.25: fig05_25.cpp
// Multipurpose sorting program using function pointers.
#include <iostream>
#include <iomanip>

using std::cout;
using std::cin;
using std::endl;

#include <iomanip>

// prototypes
void bubble( int [], const int, bool (*)( int, int ) );
void swap( int * const, int * const );
bool ascending( int, int );
bool descending( int, int );

int main()
{
  const int arraySize = 10;
  int order;
  int counter;
  int a[ arraySize ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };

  cout << "Enter 1 to sort in ascending order,
" << "Enter 2 to sort in descending order: ";
  cin >> order;
  cout << "Data items in original order\n";

  for ( counter = 0; counter < arraySize; counter++ )
    cout << setw( 4 ) << a[ counter ];

  cout << " Data items in ascending order\n";
  if ( order == 1 ) { bubble( a, arraySize, ascending );
    cout << " Data items in ascending order\n";
  }

  cout << " Data items in descending order\n";
  else { bubble( a, arraySize, descending );
    cout << " Data items in descending order\n";
  }

  return 0;
}
// output sorted array
for ( counter = 0; counter < arraySize; counter++ )
cout << setw( 4 ) << a[ counter ];
cout << endl;
return 0; // indicates successful termination

// multipurpose bubble sort; parameter compare is pointer
// to function that determines sorting order
void bubble( int work[], const int size,
bool (*compare)( int, int ) )
{
    // loop to control passes
    for ( int pass = 1; pass < size; pass++ )
    {
        // loop to control number of comparisons per pass
        for ( int count = 0; count < size - 1; count++ )
        {
            // if adjacent elements are out of order, swap them
            if ( (*compare)( work[ count ], work[ count + 1 ] ) )
                swap( &work[ count ], &work[ count + 1 ] );
        }
    }
}

// swap values at memory locations to which
// element1Ptr and element2Ptr point
void swap( int * const element1Ptr, int * const element2Ptr )
{
    int hold = *element1Ptr;
    *element1Ptr = *element2Ptr;
    *element2Ptr = hold;
}

// determine whether elements are out of order
// for an ascending order sort
bool ascending( int a, int b )
{
    return b < a; // swap if b is less than a
}

// end function bubble

// end function swap
// end function ascending

// multipurpose bubble sort; parameter compare is pointer
// to function that determines sorting order
void bubble( int work[], const int size,
            bool (*compare)( int, int ) )
{
    // loop to control passes
    for ( int pass = 1; pass < size; pass++ )
    {
        // loop to control number of comparisons per pass
        for ( int count = 0; count < size - 1; count++ )
        {
            // if adjacent elements are out of order, swap them
            if ( (*compare)( work[ count ], work[ count + 1 ] ) )
                swap( &work[ count ], &work[ count + 1 ] );
        }
    }
}

// swap values at memory locations to which
// element1Ptr and element2Ptr point
void swap( int * const element1Ptr, int * const element2Ptr )
{
    int hold = *element1Ptr;
    *element1Ptr = *element2Ptr;
    *element2Ptr = hold;
}

// determine whether elements are out of order
// for an ascending order sort
bool ascending( int a, int b )
{
    return b < a; // swap if b is less than a
}

// end function bubble

// end function swap
// end function ascending
94// determine whether elements are out of order
95// for a descending order sort
96bool descending( int a, int b )
97{
98    return b > a;   // swap if b is greater than a
99}
100 } // end function descending

Enter 1 to sort in ascending order,
Enter 2 to sort in descending order: 1
Data items in original order
2   6   4   8  10  12  89  68  45  37
Data items in ascending order
2   4   6   8  10  12  37  45  68  89

Enter 1 to sort in ascending order,
Enter 2 to sort in descending order: 2
Data items in original order
2   6   4   8  10  12  89  68  45  37
Data items in descending order
89  68  45  37  12  10   8   6   4   2

Function Pointers

• Arrays of pointers to functions
  – Menu-driven systems
  – Pointers to each function stored in array of pointers to functions
    • All functions must have same return type and same parameter types
  – Menu choice → subscript into array of function pointers
// Fig. 5.26: fig05_26.cpp  
// Demonstrating an array of pointers to functions.  
#include <iostream>  
using std::cout;  
using std::cin;  
using std::endl;  

// function prototypes  
void function1( int );  
void function2( int );  
void function3( int );  

int main()  
{  
    // initialize array of 3 pointers to functions that each  
    // take an int argument and return void  
    void (*f[ 3 ])( int ) = { function1, function2, function3 };  

    int choice;  
    cout << "Enter a number between 0 and 2, 3 to end: ";  
    cin >> choice;  

    // process user's choice  
    while ( choice >= 0 && choice < 3 ) {  
        // invoke function at location choice in array f  
        // and pass choice as an argument  
        (*f[ choice ])( choice );  
        cout << "Enter a number between 0 and 2, 3 to end: ";  
        cin >> choice;  
    }  

    return 0;  
} // end main  

void function1( int a )  
{  
    cout << "You entered " << a  
    << " so function1 was called\n\n";  
} // end function1
void function2( int b )
{
    cout << "You entered " << b
        << " so function2 was called\n\n";
}

void function3( int c )
{
    cout << "You entered " << c
        << " so function3 was called\n\n";
}

Enter a number between 0 and 2, 3 to end: 0
You entered 0 so function1 was called

Enter a number between 0 and 2, 3 to end: 1
You entered 1 so function2 was called

Enter a number between 0 and 2, 3 to end: 2
You entered 2 so function3 was called

Enter a number between 0 and 2, 3 to end: 3
Program execution completed.

Fundamentals of Characters and Strings

- Character constant
  - Integer value represented as character in single quotes
  - 'z' is integer value of z
    - 122 in ASCII
- String
  - Series of characters treated as single unit
  - Can include letters, digits, special characters +, -, *, ...
  - String literal (string constants)
    - Enclosed in double quotes, for example:
      "I like C++"
  - Array of characters, ends with null character '\0'
  - String is constant pointer
    - Pointer to string's first character
      - Like arrays
Fundamentals of Characters and Strings

• String assignment
  – Character array
    • char color[] = "blue";
      – Creates 5 element char array color
        • last element is '\0'
  – Variable of type char *
    • char *colorPtr = "blue";
      – Creates pointer colorPtr to letter b in string "blue"
        • "blue" somewhere in memory
  – Alternative for character array
    • char color[] = { 'b', 'l', 'u', 'e', '\0' };
Fundamentals of Characters and Strings

- **cin.getline**
  - Read line of text
  - `cin.getline( array, size, delimiter );`
  - Copies input into specified `array` until either
    - One less than `size` is reached
    - `delimiter` character is input
  - Example
    ```
    char sentence[ 80 ];
    cin.getline( sentence, 80, '\n' );
    ```

String Manipulation Functions of the String-handling Library

String handling library `<cstring>` provides functions to
- Manipulate string data
- Compare strings
- Search strings for characters and other strings
- Tokenize strings (separate strings into logical pieces)
### String Manipulation Functions of the String-handling Library

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>char *strcpy( char *s1, const char *s2 );</code></td>
<td>Copies the string <code>s2</code> into the character array <code>s1</code>. The value of <code>s1</code> is returned.</td>
</tr>
<tr>
<td><code>char *strncpy( char *s1, const char *s2, size_t n );</code></td>
<td>Copies at most <code>n</code> characters of the string <code>s2</code> into the character array <code>s1</code>. The value of <code>s1</code> is returned.</td>
</tr>
<tr>
<td><code>char *strcat( char *s1, const char *s2 );</code></td>
<td>Appends the string <code>s2</code> to the string <code>s1</code>. The first character of <code>s2</code> overwrites the terminating null character of <code>s1</code>. The value of <code>s1</code> is returned.</td>
</tr>
<tr>
<td><code>char *strncat( char *s1, const char *s2, size_t n );</code></td>
<td>Appends at most <code>n</code> characters of the string <code>s2</code> to string <code>s1</code>. The first character of <code>s2</code> overwrites the terminating null character of <code>s1</code>. The value of <code>s1</code> is returned.</td>
</tr>
<tr>
<td><code>int strcmp( const char *s1, const char *s2 );</code></td>
<td>Compares the string <code>s1</code> with the string <code>s2</code>. The function returns a value of zero, less than zero or greater than zero if <code>s1</code> is equal to, less than or greater than <code>s2</code>, respectively.</td>
</tr>
<tr>
<td><code>int strncmp( const char *s1, const char *s2, size_t n );</code></td>
<td>Compares up to <code>n</code> characters of the string <code>s1</code> with the string <code>s2</code>. The function returns zero, less than zero or greater than zero if <code>s1</code> is equal to, less than or greater than <code>s2</code>, respectively.</td>
</tr>
<tr>
<td><code>int strlen( const char *s );</code></td>
<td>Determines the length of string <code>s</code>. The number of characters preceding the terminating null character is returned.</td>
</tr>
<tr>
<td><code>char *strtok( char *s1, const char *s2 );</code></td>
<td>A sequence of calls to <code>strtok</code> breaks string <code>s1</code> into &quot;tokens&quot;—logical pieces such as words in a line of text—delimited by characters contained in string <code>s2</code>. The first call contains <code>s1</code> as the first argument, and subsequent calls to continue tokenizing the same string contain <code>NULL</code> as the first argument. A pointer to the current token is returned by each call. If there are no more tokens when the function is called, <code>NULL</code> is returned.</td>
</tr>
</tbody>
</table>