Program Components in C++

- Modules: functions and classes
  - Programs use new and “prepackaged” modules
    - New: programmer-defined functions, classes
    - Prepackaged: from the standard library
  - Functions invoked by function call
    - Function name and information (arguments) it needs
- Function definitions
  - Only written once
  - Hidden from other functions

Math Library Functions

- Perform common mathematical calculations
  - Include the header file `<cmath>`
- Functions called by writing
  - functionName (argument);
  - functionName(argument1, argument2, ...);
- Example
  - `cout << sqrt( 900.0 );`
- All functions in math library return a `double`

- Function arguments can be
  - Constants
    - `sqrt( 4 );`
  - Variables
    - `sqrt( x );`
  - Expressions
    - `sqrt( sqrt( x ) );`
    - `sqrt( 3 - 6x );`
- Other functions
  - `ceil(x)`, `floor(x)`, `log10(x)`, etc.
Functions

• Functions
  – Modularize a program
  – Software reusability
    • Call function multiple times

• Local variables
  – Known only in the function in which they are defined
  – All variables declared in function definitions are local variables

• Parameters
  – Local variables passed to function when called
  – Provide outside information

Function Definitions

• Function prototype
  – Tells compiler argument type and return type of function
  – int square( int );
  – Function takes an int and returns an int

• Calling/invoking a function
  – square(x);

• Format for function definition
  
  return-value-type  function-name( parameter-list )
  {
  declarations and statements
  }

• Prototype must match function definition
  – Function prototype
  
  double maximum( double, double, double );
  – Definition
    
    double maximum( double x, double y, double z )
    {
    …
    }

Function Definitions

• Example function
  
  int square( int y )
  {
  return y * y;
  }

• return keyword
  – Returns data, and control goes to function’s caller
    • If no data to return, use return;
    – Function ends when reaches right brace
    • Control goes to caller
  – Functions cannot be defined inside other functions
Function Prototypes

Data types
- long double
- double
- float
- aligned long int (synonymous with unsigned long)
- unsigned long int (synonymous with unsigned long)
- long int (synonymous with int)
- aligned int (synonymous with int)
- int
- unsigned short int (synonymous with unsigned short)
- short int (synonymous with short)
- aligned char
- char

Fig. 3.5 Promotion hierarchy for built-in data types.

Header Files

- Header files contain
  - Function prototypes
  - Definitions of data types and constants
- Header files ending with .h
  - Programmer-defined header files
    
  #include "myheader.h"
- Library header files
  
  #include <cmath>

Enumeration: enum

- Enumeration
  - Set of integers with identifiers
    
    enum typeName {constant1, constant2 ... };
  - Constants start at 0 (default), incremented by 1
  - Constants need unique names
  - Cannot assign integer to enumeration variable
    - Must use a previously defined enumeration type
- Example
    
    enum Status {CONTINUE, WON, LOST};
    Status enumVar;
    enumVar = WON; // cannot do enumVar = 1

Storage Classes

- Variables have attributes
  - Have seen name, type, size, value
  - Storage class
    - How long variable exists in memory
    - Scope
      - Where variable can be referenced in program
    - Linkage
      - For multiple-file program (see Ch. 6), which files can use it
Storage Classes

• Automatic storage class
  – Variable created when program enters its block
  – Variable destroyed when program leaves block
  – Only local variables of functions can be automatic
    • Automatic by default
    • Keyword `auto` explicitly declares automatic
  – `register` keyword
    • Hint to place variable in high-speed register
    • Good for often-used items (loop counters)
    • Often unnecessary, compiler optimizes
  – Specify either `register` or `auto`, not both

`register int counter = 1;`

Storage Classes

• Static storage class
  – Variables exist for entire program
  – May not be accessible, scope rules still apply (more later)
  • `auto` and `register` keyword
    – Created and active in a block
    – Local variables in function
    – `register` variables are kept in CPU registers
  • `static` keyword
    – Local variables in function
    – Keeps value between function calls
    – Only known in own function
  • `extern` keyword
    – Default for global variables/functions
    – Global, defined outside of a function block
    – Known in any function that comes after it

Scope Rules

• Scope
  – Portion of program where identifier can be used
• File scope
  – Defined outside a function, known in all functions
  – Global variables, function definitions and prototypes
• Function scope
  – Can only be referenced inside defining function
  – Only labels, e.g., identifiers with a colon (`case:`)
1 // Fig. 3.12: fig03_12.cpp
2 // A scoping example.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 void useLocal( void );        // function prototype
9 void useStaticLocal( void );  // function prototype
10
11 usingGlobal();      // function prototype
12
13 int x = 1;     // global variable
14
15 int main()
16 {
17    int x = 5;   // local variable to main
18
19    cout << "local x in main's outer scope is " << x << endl;
20
21    { // start new scope
22       int x = 7;  // local variable to main
23       // start new scope
24       cout << "local x in main's inner scope is " << x << endl;
25       // end new scope
26    } // end new scope
27
28    cout << "local x in main is " << x << endl;
29
30    useLocal();       // useLocal has local x
31    useStaticLocal(); // useStaticLocal has static local x
32    useGlobal();      // useGlobal uses global x
33    useLocal();      // useLocal reinitializes its local x
34    useStaticLocal(); // static local x retains its prior value
35    useGlobal();      // global x also retains its value
36
37    return 0;   // indicates successful termination
38
39 } // end main
40
41 // useLocal
42 void useLocal( void )
43 {
44    int x = 25;  // initialized each time useLocal is called
45
46    cout << endl << "local x is " << x
47      << " on entering useLocal" << endl;
48    ++x;
49    cout << "local x is " << x
50      << " on exiting useLocal" << endl;
51
52 } // end function useLocal
53
54 // useStaticLocal initializes local variable x during each call
55 void useStaticLocal( void )
56 {
57    int x = 50;
58
59    cout << endl << "local static x is " << x
60      << " on entering useStaticLocal" << endl;
61    ++x;
62    cout << "local static x is " << x
63      << " on exiting useStaticLocal" << endl;
64
65 } // end function useStaticLocal
66
67 // useGlobal initializes local variable x during each call
68 void useGlobal()
69 {
70    int x = 50;
71
72    cout << "local x in main's outer scope is " << x << endl;
73    useLocal();      // useLocal has local x
74
75    useStaticLocal(); // useStaticLocal has static local x
76    useGlobal();      // useGlobal uses global x
77
78    useLocal();      // useLocal reinitializes its local x
79    useStaticLocal(); // static local x retains its prior value
80    useGlobal();      // global x also retains its value
81
82    cout << "local x in main is " << x << endl;
83    return 0;   // indicates successful termination
84
85 } // end function useGlobal
86
87 // function useLocal initializes local variable x during each call
88 void useLocal()
89 {
90    int x = 50;
91
92    cout << "local x in main's outer scope is " << x << endl;
93    useLocal();      // useLocal has local x
94
95    useStaticLocal(); // useStaticLocal has static local x
96    useGlobal();      // useGlobal uses global x
97
98    useLocal();      // useLocal reinitializes its local x
99    useStaticLocal(); // static local x retains its prior value
100   useGlobal();      // global x also retains its value
101
102   cout << "local x in main is " << x << endl;
103   return 0;   // indicates successful termination
104
105 } // end function useLocal
106
107 // function useStaticLocal initializes local variable x during each call
108 void useStaticLocal()
109 {
110    int x = 50;
111
112    cout << "local x in main's outer scope is " << x << endl;
113    useLocal();      // useLocal has local x
114
115    useStaticLocal(); // useStaticLocal has static local x
116    useGlobal();      // useGlobal uses global x
117
118    useLocal();      // useLocal reinitializes its local x
119    useStaticLocal(); // static local x retains its prior value
120    useGlobal();      // global x also retains its value
121
122    cout << "local x in main is " << x << endl;
123    return 0;   // indicates successful termination
124
125 } // end function useStaticLocal
126
127 // function useGlobal initializes local variable x during each call
128 void useGlobal()
129 {
130    int x = 50;
131
132    cout << "local x in main's outer scope is " << x << endl;
133    useLocal();      // useLocal has local x
134
135    useStaticLocal(); // useStaticLocal has static local x
136    useGlobal();      // useGlobal uses global x
137
138    useLocal();      // useLocal reinitializes its local x
139    useStaticLocal(); // static local x retains its prior value
140    useGlobal();      // global x also retains its value
141
142    cout << "local x in main is " << x << endl;
143    return 0;   // indicates successful termination
144
145 } // end function useGlobal
146
147 // function useLocal initializes local variable x during each call
148 void useLocal()
149 {
150    int x = 50;
151
152    cout << "local x in main's outer scope is " << x << endl;
153    useLocal();      // useLocal has local x
154
155    useStaticLocal(); // useStaticLocal has static local x
156    useGlobal();      // useGlobal uses global x
157
158    useLocal();      // useLocal reinitializes its local x
159    useStaticLocal(); // static local x retains its prior value
160    useGlobal();      // global x also retains its value
161
162    cout << "local x in main is " << x << endl;
163    return 0;   // indicates successful termination
164
165 } // end function useLocal
166
167 // function useStaticLocal initializes local variable x during each call
168 void useStaticLocal()
169 {
170    int x = 50;
171
172    cout << "local x in main's outer scope is " << x << endl;
173    useLocal();      // useLocal has local x
174
175    useStaticLocal(); // useStaticLocal has static local x
176    useGlobal();      // useGlobal uses global x
177
178    useLocal();      // useLocal reinitializes its local x
179    useStaticLocal(); // static local x retains its prior value
180    useGlobal();      // global x also retains its value
181
182    cout << "local x in main is " << x << endl;
183    return 0;   // indicates successful termination
184
185 } // end function useStaticLocal
186
187 // function useGlobal initializes local variable x during each call
188 void useGlobal()
189 {
190    int x = 50;
191
192    cout << "local x in main's outer scope is " << x << endl;
193    useLocal();      // useLocal has local x
194
195    useStaticLocal(); // useStaticLocal has static local x
196    useGlobal();      // useGlobal uses global x
197
198    useLocal();      // useLocal reinitializes its local x
199    useStaticLocal(); // static local x retains its prior value
200    useGlobal();      // global x also retains its value
Recursion

- Example: factorial
  \[ n! = n \times (n - 1) \times (n - 2) \times \ldots \times 1 \]
  - Recursive relationship \( (n! = n \times (n - 1)!) \)
  \[ 5! = 5 \times 4! \]
  \[ 4! = 4 \times 3! \ldots \]
  - Base case \((1! = 0! = 1)\)
Recursive definition of function factorial:

```cpp
unsigned long factorial( unsigned long number )
{
    // base case
    if ( number <= 1 )
        return 1;
    // recursive step
    else
        return number * factorial( number - 1 );
}
```

The base case occurs when we have 0! or 1! All other cases must be split up (recursive step).

The factorial sequence:

- 0! = 1
- 1! = 1
- 2! = 2
- 3! = 6
- 4! = 24
- 5! = 120
- 6! = 720
- 7! = 5040
- 8! = 40320
- 9! = 362880
- 10! = 3628800

Example Using Recursion: Fibonacci Series

- Fibonacci series: 0, 1, 1, 2, 3, 5, 8...
- Each number is the sum of the two previous ones
- Example of a recursive formula:
  \[ f(n) = f(n-1) + f(n-2) \]
- C++ code for Fibonacci function

```cpp
long fibonacci( long n )
{
    if ( n == 0 || n == 1 )  // base case
        return n;
    else
        return fibonacci( n - 1 ) + fibonacci( n - 2 );
}
```

Example Using Recursion: Fibonacci Series

- Order of operations:
  - return fibonacci( n - 1 ) + fibonacci( n - 2 );
- Recursive function calls:
  - Each level of recursion doubles the number of function calls
  - 30th number ≈ 2^30 = 4 billion function calls
- Exponential complexity

Recursion vs. Iteration

- Repetition:
  - Iteration: explicit loop
  - Recursion: repeated function calls
- Termination:
  - Iteration: loop condition fails
  - Recursion: base case recognized
- Both can have infinite loops
- Balance between performance (iteration) and good software engineering (recursion)
Inline Functions

- **Inline functions**
  - Keyword *inline* before function
  - Asks the compiler to copy code into program instead of making function call
    - Reduce function-call overhead
    - Compiler can ignore *inline*
  - Good for small, often-used functions
- **Example**
  ```c++
  inline double cube( const double s )
  {
      return s * s * s;
  }
  ```

References and Reference Parameters

- **Call by value**
  - Copy of data passed to function
  - Changes to copy do not change original
  - Prevent unwanted side effects
- **Call by reference**
  - Function can directly access data
  - Changes affect original
- **Reference parameter**
  - Alias for argument in function call
    - Passes parameter by reference
    - Use & after data type in prototype
    - *void myFunction( int &data )*
    - Real "data is a reference to an int"
    - Function call format the same
      - However, original can now be changed
// Fig. 3.20: fig03_20.cpp
// Comparing pass-by-value and pass-by-reference
// with references.

#include <iostream>

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

int squareByValue( int );         // function prototype
void squareByReference( int & );  // function prototype

int main()
{
    int x = 2;
    int z = 4;

    // demonstrate squareByValue
    cout << "x = " << x << " before squareByValue
";
    cout << "Value returned by squareByValue: "
         << squareByValue( x ) << endl;
    cout << "x = " << x << " after squareByValue
" << endl;

    // demonstrate squareByReference
    cout << "z = " << z << " before squareByReference
";
    squareByReference( z );
    cout << "z = " << z << " after squareByReference
" << endl;

    return 0;  // indicates successful termination
}

// squareByValue multiplies number by itself, stores the
// result in number and returns the new value of number
int squareByValue( int number )
{
    return number *= number;  // caller's argument not modified
}

// squareByReference multiplies numberRef by itself and
// stores the result in the variable to which numberRef
// refers in function main
void squareByReference( int &numberRef )
{
    numberRef *= numberRef;   // caller's argument modified
}

References and Reference Parameters

- Pointers
  - Another way to pass-by-reference
- References as aliases to other variables
  - Refer to same variable
  - Can be used within a function
    - int count = 1; // declare integer variable count
    - int &chef = count; // create chef as an alias for count
    - ++chef; // increment count (using its alias)
- References must be initialized when declared
  - Otherwise, compiler error
  - Dangling reference
    - Reference to undefined variable
// Fig. 3.21: fig03_21.cpp
// References must be initialized.
#include <iostream>

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

int main()
{
  int x = 3;
  // y refers to (is an alias for) x
  int &y = x;
  cout << "x = " << x << endl << "y = " << y << endl;
  y = 7;
  cout << "x = " << x << endl << "y = " << y << endl;
  return 0;  // indicates successful termination
} // end main

// Fig. 3.22: fig03_22.cpp
// References must be initialized.
#include <iostream>

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

int main()
{
  int x = 3;
  int &y; // Error: y must be initialized
  cout << "x = " << x << endl << "y = " << y << endl;
  y = 7;
  cout << "x = " << x << endl << "y = " << y << endl;
  return 0;  // indicates successful termination
} // end main

Default Arguments

• Function call with omitted parameters
  – If not enough parameters, rightmost go to their defaults
  – Default values
    • Can be constants, global variables, or function calls
• Set defaults in function prototype
  int myFunction(int x = 1, int y = 2, int z = 3);
  – myFunction(3)
    • x = 3, y and z get defaults (rightmost)
  – myFunction(3, 5)
    • x = 3, y = 5 and z gets default

Unitary Scope Resolution Operator

• Unitary scope resolution operator (::)
  – Access global variable if local variable has same name
  – Not needed if names are different
  – Use ::variable
    • y = ::x + 3;
  – Good to avoid using same names for locals and globals
Function Overloading

- Function overloading
  - Functions with same name and different parameters
    - Should perform similar tasks
      - i.e., function to square integers and function to square floats
        ```cpp
        int square(int x) { return x * x; }
        float square(float x) { return x * x; }
        ```
  - Overloaded functions distinguished by signature
    - Based on name and parameter types (order matters)
    - Name mangling
    - Encodes function identifier with parameters
    - Type-safe linkage
      - Ensures proper overloaded function called

Function Templates

- Compact way to make overloaded functions
  - Generate separate function for different data types
- Format
  - Begin with `template`
    - Formal type parameters in brackets `<>`
      - Every type parameter preceded by `typename` or `class` (synonyms)
      - Placeholders for built-in types (i.e., `int`) or user-defined types
  - Specify arguments types, return types, declare variables
- Example
  ```cpp
  template < class T >  // or template < typename T >
  T square( T value1 )
  {
    T max = value1;
    if ( value2 > max )
      max = value2;
    if ( value3 > max )
      max = value3;
    return max;
  }
  ```
- T is a formal type, used as parameter type
- Above function returns variable of same type as parameter
- In function call, T replaced by real type
  - If int, all T's become ints
  - int x;
  - int y = square(x);
```cpp
int main()
{
    // demonstrate maximum with int values
    int int1, int2, int3;
    cout << "Input three integer values: ";
    cin >> int1 >> int2 >> int3;

    // invoke int version of maximum
    cout << "The maximum integer value is: " << maximum( int1, int2, int3 );

    // demonstrate maximum with double values
    double double1, double2, double3;
    cout << "Input three double values: ";
    cin >> double1 >> double2 >> double3;

    // invoke double version of maximum
    cout << "The maximum double value is: " << maximum( double1, double2, double3 );

    // demonstrate maximum with char values
    char char1, char2, char3;
    cout << "Input three characters: ";
    cin >> char1 >> char2 >> char3;

    // invoke char version of maximum
    cout << "The maximum character value is: " << maximum( char1, char2, char3 ) << endl;

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

Input three integer values: 1 2 3
The maximum integer value is: 3
Input three double values: 3.3 2.2 1.1
The maximum double value is: 3.3
Input three characters: A C B
The maximum character value is: C
Declaring Arrays

• When declaring arrays, specify
  – Name
  – Type of array
    • Any data type
  – Number of elements
  – Type arrayName[ arraySize ];
    int a[ 10 ]; // array of 10 integers
    float b[ 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, slightest elements 0
  • If too many syntax error
    int n[ 5 ] = { 0 };
  – To set every element to same value
    int n[ 5 ] = { 1, 2, 3, 4, 5 };
  – If array size omitted, initializers determine size
    int n[] = { 1, 2, 3, 4, 5 };
  • 5 initializers, therefore 5 element array

• Strings
  – Arrays of characters
  – All strings end with null (\0)
  – Examples
    • char string1[] = "hello";
    • 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
Examples Using Arrays

• Recall static storage
  – If static, local variables save values between function calls
  – Visible only in function body
  – Can declare local arrays to be static
    • Initialized to zero
      static int array[3];
  – If not static
    • Created (and destroyed) in every function call

Passing Arrays to Functions

• 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
  – Individual array elements passed-by-value
    • Like regular variables
      square( myArray[3] );

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

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 [] );
// Fig. 4.14: fig04_14.cpp
// Passing arrays and individual array elements to functions.
#include <iostream>
#include <iomanip>

using std::cout;
using std::endl;
using std::setw;

void modifyArray( int[], int );  // appears strange
void modifyElement( int );

int main()
{
const int arraySize = 5; // size of array a
int a[ arraySize ] = { 0, 1, 2, 3, 4 };  // initialize a

cout << "Effects of passing entire array by reference:

The values of the original array are:
";
for ( int i = 0; i < arraySize; i++ )
cout << setw( 3 ) << a[ i ];
cout << endl;
// pass array a to modifyArray by reference
modifyArray( a, arraySize );

cout << "The values of the modified array are:
";
for ( int j = 0; j < arraySize; j++ )
cout << setw( 3 ) << a[ j ];
// output value of a[ 3 ]
cout << "

Effects of passing array element by value:

The value of a[3] is 6
Value in modifyElement is 12
The value of a[3] is 6";

return 0;  // indicates successful termination
}

void modifyArray( int b[], int sizeOfArray )
{
// multiply each array element by 2
for ( int k = 0; k < sizeOfArray; k++ )
b[ k ] *= 2;
}

void modifyElement( int e )
{
// multiply parameter by 2
cout << "Value in modifyElement is 
" << ( e *= 2 ) << endl;
}

Although named b, the array points to the original array a in memory.
Individual array elements are passed by value, and the originals cannot be changed.
Pass array name (a) and size to function. Arrays are passed-by-reference.
Pass a single array element by value; the original cannot be modified.
// Fig. 4.15: fig04_15.cpp
// Demonstrating the const type qualifier.

#include <iostream>

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

void tryToModifyArray( const int [] );  // function prototype

int main()
{
    int a[] = { 10, 20, 30 };
    tryToModifyArray( a );
    cout << a[ 0 ] << ' ' << a[ 1 ] << ' ' << a[ 2 ] << '
';
    return 0;  // indicates successful termination
}

Array parameter declared as const. Array cannot be modified, even though it is passed by reference.

In function tryToModifyArray, "b" cannot be used to modify the original array "a" in main.

void tryToModifyArray( const int b[] )
{
    b[ 0 ] /= 2;   // error
    b[ 1 ] /= 2;   // error
    b[ 2 ] /= 2;    // error
}

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 7 6 (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)
• Swap function?

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

<table>
<thead>
<tr>
<th>Row 1</th>
<th>Row 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2</td>
<td>1 2</td>
</tr>
<tr>
<td>2 1</td>
<td>2 1</td>
</tr>
<tr>
<td>1 2</td>
<td>1 2</td>
</tr>
<tr>
<td>1 2</td>
<td>1 2</td>
</tr>
</tbody>
</table>

Multiple-Subscripted Arrays (1 of 2)
Multiple-Subscripted Arrays

• To initialize
  – Default of 0
  – Initializers grouped by row in braces

  \[
  \begin{array}{c}
  \text{int } b[2][2] = \{ \{ 1, 2 \}, \{ 3, 4 \} \}; \\
  \text{Row 0} & \text{Row 1} \\
  1 & 2 \\
  3 & 4
  \end{array}
  \]

  \[
  \begin{array}{c}
  \text{int } b[2][2] = \{ \{ 1 \}, \{ 3, 4 \} \}; \\
  1 & 0 \\
  3 & 4
  \end{array}
  \]

Pointers

• Pointers
  – Powerful, but difficult to master
  – Simulate pass-by-reference
  – Close relationship with arrays and strings

• Can declare pointers to any data type
• Pointer initialization
  – Initialized to \texttt{0}, \texttt{NULL}, or address
  – \texttt{0} or \texttt{NULL} points to nothing

Pointers

• Pointers
  – Powerful, but difficult to master
  – Simulate pass-by-reference
  – Close relationship with arrays and strings

• Can declare pointers to any data type
• Pointer initialization
  – Initialized to \texttt{0}, \texttt{NULL}, or address
  – \texttt{0} or \texttt{NULL} points to nothing

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
    \[
    \text{int } *\text{myPtr};
    \]
  – declares pointer to \texttt{int}, pointer of type \texttt{int *}
  – Multiple pointers require multiple asterisks
    \[
    \text{int } *\text{myPtr1}, *\text{myPtr2};
    \]

Pointer Operators

• \& (address operator)
  – Returns memory address of its operand
  – Example
    \[
    \text{int } y = 5;
    \text{int } *yPtr;
    yPtr = &y; \quad \text{// } yPtr \text{ gets address of } y
    \]

  \[
  \begin{array}{c}
  \text{Address of } y \text{ is value of } yPtr
  \end{array}
  \]
**Pointer Operators**

* (indirection/dereferencing operator)
  - Returns synonym for object its pointer operand points to
  - \( *y\text{Ptr} \) returns \( y \) (because \( y\text{Ptr} \) points to \( y \)).
  - Dereferenced pointer is lvalue
    
    \[ *y\text{ptr} = 9; \quad // \text{assigns 9 to } y \]
  - \( * \) and \& are inverses of each other

**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

* return can return one value from function

Arguments passed to function using reference arguments
  - Modify original values of arguments
  - More than one value "returned"

Pass-by-reference with pointer arguments
  - Simulate pass-by-reference
    - Use pointers and indirection operator
    - Pass address of argument using \& operator
    - Arrays not passed with \& because array name already pointer
    - * operator used as alias/nickname for variable inside of function
# Using `const` with Pointers

## `const` qualifier
- Value of variable should not be modified
- `const` used when function does not need to change a variable

## Principle of least privilege
- Award function enough access to accomplish task, but no more

## 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
// Fig. 5.13: fig05_13.cpp
// Attempting to modify a constant pointer to non-constant data.

int main()
{
    int x, y;

    int * const ptr = &x; // ptr is a constant pointer to an integer that can be modified through ptr,
    // but ptr always points to the same memory location.

    *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.
// Fig. 5.14: fig05_14.cpp
// Attempting to modify a constant pointer to constant data.
#include <iostream>

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

int main()
{
    int x = 5, y;

    // ptr is a constant pointer to a constant integer.
    // ptr always points to the same location; the integer
    // at that location cannot be modified.
    const int *const ptr = &x;

    cout << *ptr << endl;

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

    return 0;  // indicates successful termination
} // end main

// Fig05_14.cpp
(1 of 1)

Outline
77
fig05_14.cpp
(1 of 1)

Outline
78
fig05_14.cpp
(1 of 1)

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 [0]. which is at location 3000
    - vpTr = 3000
    - vpTr += 2; sets vpTr to 3008
    - vpTr points to v[2]

Pointer Expressions and Pointer Arithmetic

Subtracting pointers

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

- 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

Outline
79
fig05_14.cpp
(1 of 1)

Outline
80
fig05_14.cpp
(1 of 1)
### 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
  - Each element of suit points to char *(a string)
  - Array does not store strings, only pointers to strings

### Function Pointers

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

using std::setw;

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

    // sort array in ascending order; pass function ascending
    // as an argument to specify ascending sorting order
    if ( order == 1 ) {
        bubble( a, arraySize, ascending );
        cout << "Data items in ascending order
    
        // sort array in descending order; pass function descending
        // as an argument to specify descending sorting order
    } else {
        bubble( a, arraySize, descending );
        cout << "Data items in descending order
    
        // 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 a pointer to
// the comparison 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
}

// compare is pointer to 
// function that receives two 
// integer parameters and 
// returns bool result.
// Parameter is pointer to 
// function that receives two 
// integer parameters and 
// returns bool result.
# 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

```cpp
// 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
    // each takes an int argument and returns 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;
    }

    cout << "Program execution completed." << endl;
    return 0;  // indicates successful termination
}

void function1( int a )
{
    cout << "You entered " << a
        << " so function1 was called

    // Array initialized with names of three functions, function names are pointers.

    1 // Fig. 5.26. fig05_26.cpp
    2 // Demonstrating an array of pointers to functions.
    3 #include <iostream>
    4 using std::cout;
    5 using std::cin;
    6 using std::endl;
    7
    8 // function prototypes
    9 void function1( int );
    10 void function2( int );
    11 void function3( int );
    12
    13 int main()
    14 {
    15     // initialize array of 3 pointers to functions
    16     void (*f[3])( int ) = { function1, function2, function3 };
    17     int choice;
    18
    19     cout << "Enter a number between 0 and 2, 3 to end: ";
    20     cin >> choice;
    21
    22     // process user's choice
    23     while ( choice >= 0 && choice < 3 ) {
    24         // invoke function at location choice in array f and pass choice as an argument
    25         (*f[ choice ])( choice );
    26
    27         cout << "Enter a number between 0 and 2, 3 to end: ";
    28         cin >> choice;
    29     }
    30
    31     cout << "Program execution completed." << endl;
    32     return 0;  // indicates successful termination
    33 }
    34
    35 void function1( int a )
    36 {
    37         cout << "You entered " << a
    38             << " so function1 was called
    39     // Call chosen function by dereferencing corresponding element in array.

    © 2003 Prentice Hall, Inc. All rights reserved.
    © 2003 Prentice Hall, Inc. All rights reserved.
```
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' };

Reading strings
- Assign input to character array word[ 20 ]
  - cin >> word
  - Reads characters until whitespace or EOF
- String could exceed array size
  - cin >> setw( 20 ) >> word;
  - Reads 19 characters (space reserved for '\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

- `<cstring>` provides functions to
  - Manipulate string data
  - Compare strings
  - Search strings for characters and other strings
  - Tokenize strings (separate strings into logical pieces)

- `int strcmp( const char *s1, const char *s2 );` - Compares the string `s1` with the string `s2`. The function returns zero, less than zero or greater than zero if `s1` is equal to, less than or greater than `s2`, respectively.

- `char *strncpy( char *s1, const char *s2, size_t n );` - Copies the string `s2` into the character array `s1`. The value of `s1` is returned.

- `char *strcat( char *s1, const char *s2 );` - Appends the string `s2` to the string `s1`. The value of the first character of `s2` overwrites the terminating null character of `s1`. The value of `s1` is returned.

- `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`. The value of `s1` is returned.

- `size_t strlen( const char *s );` - Determines the length of string `s`. The function returns a value of zero, less than zero or greater than zero if `s` is equal to, less than or greater than `s2`, respectively.

- `char *strtok( char *s1, const char *s2 );` - A sequence of calls to `strtok` breaks string `s1` into “tokens”—logical pieces such as words in a line of text—delimited by characters contained in string `s2`. The first call contains `s1` as the first argument, and subsequent calls to continue tokenizing the same string contain NULL 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, NULL is returned.

- `int strncmp( const char *s1, const char *s2, size_t n );` - Compares up to `n` characters of the string `s1` with the string `s2`. The function returns zero, less than zero or greater than zero if `s1` is equal to, less than or greater than `s2`, respectively.