IS 0020
Program Design and Software Tools
Introduction to C++ Programming

Lecture 2
Functions and Arrays

Jan 13, 2004

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);

  or
  - functionName(argument1, argument2, …);
- Example

```
cout << sqrt( 900.0 );
```

  - sqrt (square root) function The preceding statement would print 30
  - All functions in math library return a **`double`**

Math Library Functions

- 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.`**

5

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

6

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 )
{
 …
}
```


7

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

8

Function Prototypes

- Function signature
  - Part of prototype with name and parameters
    - **`double maximum( double, double, double );`**

      (Function signature: `maximum( double, double, double )`)
- Argument Coercion
  - Force arguments to be of proper type
    - Converting **`int`** (4) to **`double`** (4.0)

    `cout << sqrt(4)`
  - Conversion rules
    - Arguments usually converted automatically
    - Changing from **`double`** to **`int`** can truncate data
      - 3.4 to 3
  - Mixed type goes to highest type (promotion)
    - **`int * double`**

Function Prototypes

Data types	
`long double`	
`double`	
`float`	
`unsigned long int`	(synonymous with `unsigned long`)
`long int`	(synonymous with `long`)
`unsigned int`	(synonymous with `unsigned`)
`int`	
`unsigned short int`	(synonymous with `unsigned short`)
`short int`	(synonymous with `short`)
`unsigned char`	
`char`	
`bool`	(`false` becomes 0, `true` becomes 1)

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
    - For functions, name exists 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
    - Globals: 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:`**)

Scope Rules

- Block scope
  - Begins at declaration, ends at right brace **`}`**
    - Can only be referenced in this range
  - Local variables, function parameters
  - **`static`** variables still have block scope
    - Storage class separate from scope
- Function-prototype scope
  - Parameter list of prototype
  - Names in prototype optional
    - Compiler ignores
  - In a single prototype, name can be used once

17

Outline

fig03_12.cpp
(1 of 5)

```
// Fig. 3.12: fig03_12.cpp
// A scoping example.
#include <iostream>

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

void useLocal( void );
void useStaticLocal( void )
void useGlobal( void );

int x = 1;      // global vari

int main()
{
   int x = 5;   // local variable to main

   cout << "local x in main's out

   { // start new scope

      int x = 7;

      cout << "local x in main's inner scope is " << x << endl;

   } // end new scope
```

Declared outside of function; global variable with file scope.

Local variable with function scope.

Create a new block, giving **x** block scope. When the block ends, this **x** is destroyed.

18

Outline

fig03_12.cpp
(2 of 5)

```

   cout << "local x in main's outer scope is " << x << endl;

   useLocal();       // useLocal has local x
   useStaticLocal(); // useStaticLocal has static local x
   useGlobal();      // useGlobal uses global x
   useLocal();       // useLocal reinitializes its local x
   useStaticLocal(); // static local x retains its prior value
   useGlobal();      // global x also retains its value

   cout << "\nlocal x in main is " << x << endl;

   return 0;   // indicates successful termination

} // end main

```


19

Outline

fig03_12.cpp
(3 of 5)

```
// useLocal reinitializes local variable x during each call
void useLocal( void )
{
   int x = 25;  // initialized each time useLocal is called

   cout << endl << "local x is
        << " on entering useLo
   ++x;
   cout << "local x is " << x
        << " on exiting useLoc

} // end function useLocal

```

Automatic variable (local variable of function). This is destroyed when the function exits, and reinitialized when the function begins.

20

Outline

fig03_12.cpp
(4 of 5)

```
// useStaticLocal initializes static local variable x only the
// first time the function is called; value of x is saved
// between calls to this function
void useStaticLocal( void )
{
   // initialized only first time useStaticLocal is called
   static int x = 50;

   cout << endl << "local static x is " << x
        << " on entering useStaticLocal" << endl;
   ++x;
   cout << "local static x is " <<
        << " on exiting useStaticLo

} // end function useStaticLocal

```

Static local variable of function; it is initialized only once, and retains its value between function calls.

21

Outline

```
// useGlobal modifies global variable x during each call
void useGlobal( void )
{
   cout << endl << "global x is " << x
        << " on entering useGlobal" << endl;
   x *= 10;
   cout << "global x is " << x
        << " on exiting useGlobal" << endl;

} // end function useGlobal
```

This function does not declare any variables. It uses the global **x** declared in the beginning of the program.

03_12.cpp
of 5)

03_12.cpp
output (1 of 2)

```
local x in main's outer scope is 5
local x in main's inner scope is 7
local x in main's outer scope is 5

local x is 25 on entering useLocal
local x is 26 on exiting useLocal

local static x is 50 on entering useStaticLocal
local static x is 51 on exiting useStaticLocal

global x is 1 on entering useGlobal
global x is 10 on exiting useGlobal
```


22

Recursion

- Recursive functions
  - Functions that call themselves
  - Can only solve a base case
- If not base case
  - Break problem into smaller problem(s)
  - Launch new copy of function to work on the smaller problem (recursive call/recursive step)
    - Slowly converges towards base case
    - Function makes call to itself inside the return statement
  - Eventually base case gets solved
    - Answer works way back up, solves entire problem

23

Recursion

- Example: factorial

  $$n! = n * (n - 1) * (n - 2) * \ldots * 1$$

  - Recursive relationship (n! = n * (n – 1)!)

    $5! = 5 * 4!$

    $4! = 4 * 3!\ldots$

  - Base case (1! = 0! = 1)

24

Outline

fig03_14.cpp
(1 of 2)

```
// Fig. 3.14: fig03_14.cpp
// Recursive factorial function.
#include <iostream>

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

#include <iomanip>

using std::setw;

unsigned long factorial( unsigned long ); // function prototype

int main()
{
   // Loop 10 times. During each iteration, calculate
   // factorial( i ) and display result.
   for ( int i = 0; i <= 10; i++ )
      cout << setw( 2 ) << i << "! = "
           << factorial( i ) << endl;

   return 0;  // indicates successful termination

} // end main
```

Data type **unsigned long** can hold an integer from 0 to 4 billion.

```
// recursive definition of function factorial
unsigned long factorial( unsigned long number )
{
   // base case
   if ( number <= 1 )
      return 1;

   // recursive step
   else
      return number * factorial( number - 1 );

} // end function factorial
```

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

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

Outline

fig03_14.cpp
(2 of 2)

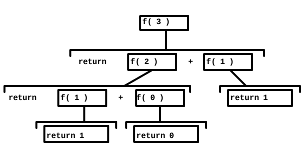
fig03_14.cpp
output (1 of 1)

Example Using Recursion: Fibonacci Series

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

```
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

```
inline double cube( const double s )
   { return s * s * s; }
```

  - **const** tells compiler that function does not modify **s**

Outline

fig03_19.cpp
(1 of 2)

```
// Fig. 3.19: fig03_19.cpp
// Using an inline function to calculate.
// the volume of a cube.
#include <iostream>

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

// Definition of inline function cube. Definition of function
// appears before function is called, so a function prototype
// is not required. First line of function definition acts as
// the prototype.
inline double cube( const double side )
{
   return side * side * side;  // calculate cube

} // end function cube

```


Outline

fig03_19.cpp
(2 of 2)

fig03_19.cpp
output (1 of 1)

```
int main()
{
   cout << "Enter the side length of your cube: ";

   double sideValue;

   cin >> sideValue;

   // calculate cube of sideValue and display result
   cout << "Volume of cube with side "
        << sideValue << " is " << cube( sideValue ) << endl;

   return 0;  // indicates successful termination

} // end main
```

```
Enter the side length of your cube: 3.5
Volume of cube with side 3.5 is 42.875
```


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)**
    - Read "**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\n";
   cout << "Value returned by squareByValue: "
        << squareByValue( x ) << endl;
   cout << "x = " << x << " after squareByValue\n" << endl;

```

Notice the **&** operator, indicating pass-by-reference.

Outline

fig03_20.cpp
(1 of 2)


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

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

// squareByValue multiplies number by itse
// result in number and returns the new va
int squareByValue( int number )
{
   return number *= number;  // caller's argument not modified

} // end function squareByValue

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

} // end function squareByReference
```

Changes **number**, but original parameter (**x**) is not modified.

Changes **numberRef**, an alias for the original parameter. Thus, **z** is changed.

Outline

fig03_20.cpp
(2 of 2)


```
x = 2 before squareByValue
Value returned by squareByValue: 4
x = 2 after squareByValue

z = 4 before squareByReference
z = 16 after squareByReference
```

Outline

fig03_20.cpp
output (1 of 1)

References and Reference Parameters

- Pointers
  - Another way to pass-by-refernce
- References as aliases to other variables
  - Refer to same variable
  - Can be used within a function

```
int count = 1;     // declare integer variable count
Int &cRef = count; // create cRef as an alias for count
++cRef; // increment count (using its alias)
```

- References must be initialized when declared
  - Otherwise, compiler error
  - Dangling reference
    - Reference to undefined variable

Outline

fig03_21.cpp
(1 of 1)

fig03_21.cpp
output (1 of 1)

```
// 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
```

```
x = 3
y = 3
x = 7
y = 7
```

y declared as a reference to **x**.

Outline

fig03_22.cpp
(1 of 1)

fig03_22.cpp
output (1 of 1)

```
// 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
```

Uninitialized reference – compiler error.

```
Borland C++ command-line compiler error message:
 Error E2304 Fig03_22.cpp 11: Reference variable 'y' must be
   initialized- in function main()

Microsoft Visual C++ compiler error message:
 D:\cpphtp4_examples\ch03\Fig03_22.cpp(11) : error C2530: 'y' :
   references must be initialized
```


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

- Unary 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 andglobals

Function Overloading

- Function overloading
  - Functions with same name and different parameters
  - Should perform similar tasks
    - I.e., function to square `int`s and function to square `float`s

```
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 keyword **`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
  - Function definition like normal, except formal types used

Function Templates

- Example

```
template < class T > // or template< typename T >
T square( T value1 )
{
   return value1 * value1;
}
```

  - **`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 **`int`**s

```
int x;
int y = square(x);
```


fig03_27.cpp
(1 of 3)

```
// Fig. 3.27: fig03_27.cpp
// Using a function template.
#include <iostream>

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

// definition of function template maximum
template < class T >  // or template < typename T >
T maximum( T value1, T value2, T value3 )
{
   T max = value1;

   if ( value2 > max )
      max = value2;

   if ( value3 > max )
      max = value3;

   return max;

} // end function template maximum

```

Formal type parameter **`T`** placeholder for type of data to be tested by **`maximum`**

`maximum` expects all parameters to be of the same type.

Outline

fig03_27.cpp
(2 of 3)

```
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 << "\n\nInput three double values: ";
   cin >> double1 >> double2 >> double3;

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

```

`maximum` called with various data types.

Outline

fig03_27.cpp
(3 of 3)

fig03_27.cpp
output (1 of 1)

```
   // demonstrate maximum with char values
   char char1, char2, char3;

   cout << "\n\nInput 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

} // end main
```

```
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
```


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 ]`** … **`c[ n - 1 ]`**
  - Nth element as position N-1

Arrays

- Array elements like other variables
  - Assignment, printing for an integer array **`c`**
    ```
    c[ 0 ] =  3;
    cout << c[ 0 ];
    ```
- Can perform operations inside subscript
  - **`c[ 5 – 2 ]`** same as **`c[3]`**

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 same value

```
int n[ 5 ] = { 0 };
```

  - If array size omitted, initializers determine size

```
int n[] = { 1, 2, 3, 4, 5 };
```

- 5 initializers, therefore 5 element array

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

53

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

54

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

55

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] );`**

56

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 [] );`**

57

Outline

fig04_14.cpp
(1 of 3)

```
// Fig. 4.14: fig04_14.cpp
// Passing arrays and individual array elements to functions.
#include <iostream>

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

#include <iomanip>

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:"
        << "\n\nThe values of the original array are:\n";

   // output original array
   for ( int i = 0; i < arraySize; i++ )
      cout << setw( 3 ) << a[ i ];
```

Syntax for accepting an array in parameter list.

58

Outline

fig04_14.cpp
(2 of 3)

```

   cout << endl;

   // pass array a to modifyArray b
   modifyArray( a, arraySize );

   cout << "The values of the modified array are:\n";

   // output modified array
   for ( int j = 0; j < arraySize; j++ )
      cout << setw( 3 ) << a[ j ];

   // output value of a[ 3 ]
   cout << "\n\n\n"
        << "Effects of passing array e
        << "\n\nThe value of a[3] is "

   // pass array element a[ 3 ] by val
   modifyElement( a[ 3 ] );

   // output value of a[ 3 ]
   cout << "The value of a[3] is " << a[ 3 ] << endl;

   return 0;  // indicates successful termination

} // end main
```

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.

59

Outline

fig04_14.cpp
(3 of 3)

```

// in function modifyArray, "b" points to
// the original array "a" in memory
void modifyArray( int b[], int sizeOfArray )
{
   // multiply each array element by 2
   for ( int k = 0; k < sizeOfArray; k++ )
      b[ k ] *= 2;

} // end function modifyArray

// in function modifyElement, "e" is a lo
// array element a[ 3 ] passed from main
void modifyElement( int e )
{
   // multiply parameter by 2
   cout << "Value in modifyElement is "
        << ( e *= 2 ) << endl;

} // end function modifyElement
```

Although named **b**, the array points to the original array **a**. It can modify **a**'s data.

Individual array elements are passed by value, and the originals cannot be changed.

60

Outline

fig04_14.cpp
output (1 of 1)

```
Effects of passing entire array by reference:

The values of the original array are:
  0  1  2  3  4
The values of the modified array are:
  0  2  4  6  8


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
```

Outline

fig04_15.cpp
(1 of 2)

```
// Fig. 4.15: fig04_15.cpp
// Demonstrating the const type qualifier.
#include <iostream>

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

void tryToModifyArray( const int [] );  //

int main()
{
   int a[] = { 10, 20, 30 };

   tryToModifyArray( a );

   cout << a[ 0 ] << ' ' << a[ 1 ] << ' ' << a[ 2 ] << '\n';

   return 0;  // indicates successful termination

} // end main

```

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

Outline

fig04_15.cpp
(2 of 2)

fig04_15.cpp
output (1 of 1)

```
// 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

} // end function tryToModifyArray
```

```
d:\cpphtp4_examples\ch04\Fig04_15.cpp(26) : error C2166:
   l-value specifies const object
d:\cpphtp4_examples\ch04\Fig04_15.cpp(27) : error C2166:
   l-value specifies const object
d:\cpphtp4_examples\ch04\Fig04_15.cpp(28) : error C2166:
   l-value specifies const object
```


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)
- 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

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Column subscript
Array name
Row subscript

Multiple-Subscripted Arrays

65

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

```
int b[ 2 ][ 2 ] = { { 1, 2 }, { 3, 4 } };
                      Row 0     Row 1
```

1	2
3	4

```
int b[ 2 ][ 2 ] = { { 1 }, { 3, 4 } };
```

1	0
3	4

Pointers

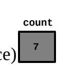
66

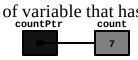
- Pointers
  - Powerful, but difficult to master
  - Simulate pass-by-reference
  - Close relationship with arrays and strings
- Can declare pointers to any data type
- Pointer initialization
  - Initialized to **0**, **NULL**, or address
    - **0** or **NULL** points to nothing

Pointer Variable Declarations and Initialization

67

- 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

    ```
    int *myPtr;
    ```

    declares pointer to **int**, pointer of type **int ***
  - Multiple pointers require multiple asterisks

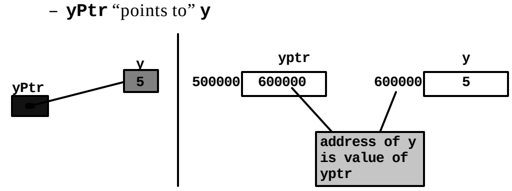
    ```
    int *myPtr1, *myPtr2;
    ```


Pointer Operators

68

- **&** (address operator)
  - Returns memory address of its operand
  - Example

    ```
    int y = 5;
    int *yPtr;
    yPtr = &y;    // yPtr gets address of y
    ```
  - **yPtr** "points to" **y**



Pointer Operators

- **`*`** (indirection/dereferencing operator)
  - Returns synonym for object its pointer operand points to
  - **`*yPtr`** returns **`y`** (because **`yPtr`** points to **`y`**).
  - dereferenced pointer is lvalue

    ```
    *yptr = 9;      // 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"

Calling Functions by Reference

- 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

fig05_07.cpp
(1 of 2)

```
// Fig. 5.7: fig05_07.cpp
// Cube a variable using pass-by-reference
// with a pointer argument.
#include <iostream>

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

void cubeByReference( int * );   // prototype

int main()
{
   int number = 5;

   cout << "The original value of numbe

   // pass address of number to cubeByR
   cubeByReference( &number );

   cout << "\nThe new value of number is " << number << endl;

   return 0;  // indicates successful termination

} // end main

```

Prototype indicates parameter is pointer to **`int`**

Apply address operator **`&`** to pass address of number to **`cubeByReference`**

`cubeByReference` modified variable **`number`**

```
// calculate cube of *nPtr; modifies variable number in main
void cubeByReference( int *nPtr )
{
   *nPtr = *nPtr * *nPtr * *nPtr;  // cube
   
} // end function cubeByReference
```

```
The original value of number is 5
The new value of number is 125
```

cubeByReference receives address of **int** variable,
i.e., pointer to an **int**

Modify and access **int** variable using indirection operator *

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

Using const with Pointers

- **const** pointers
  - Always point to same memory location
  - Default for array name
  - Must be initialized when declared


```
1  // Fig. 5.13: fig05_13.cpp
2  // Attempting to modify a constant pointer to
3  // non-constant data.
4
5  int main()
6  {
7     int x, y;
8
9     // ptr is a constant pointer to an int
10    // be modified through pt
11    // same memory location.
12    int * const ptr = &x;
13
14    *ptr = 7;  // allowed: *p
15    ptr = &y;  // error: ptr is const; can
16
17    return 0;  // indicates successful ter
18
19 } // end main
```

```
d:\cpphtp4_examples\ch05\Fig05_13.cpp(15) : error C2166:
   l-value specifies const object
```

ptr is constant pointer to

Can modify **x** (pointed to by

Cannot modify **ptr** to point to new address since **ptr** is constant.

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

Outline

fig05_14.cpp
(1 of 1)

```
1  // Fig. 5.14: fig05_14.cpp
2  // Attempting to modify a constant pointer to constant data.
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  int main()
9  {
10    int x = 5, y;
11
12    // ptr is a constant pointer to a co
13    // ptr always points to the same loc
14    // at that location cannot be modified.
15    const int *const ptr = &x
16
17    cout << *ptr << endl;
18
19    *ptr = 7;  // error: *ptr                                value
20    ptr = &y;  // error: ptr is const; cannot assign new address
21
22    return 0;  // indicates successful termination
23
24 } // end main
```

ptr is constant pointer to integer constant.

Cannot modify **x** (pointed to

Cannot modify **ptr** to point to new address since **ptr** is constant.

Outline

_14.cpp
ut (1 of 1)

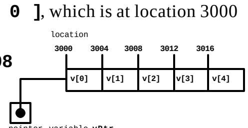
```
d:\cpphtp4_examples\ch05\Fig05_14.cpp(19) : error C2166:
   l-value specifies const object
d:\cpphtp4_examples\ch05\Fig05_14.cpp(20) : error C2166:
   l-value specifies const object
```

Line 19 generates compiler

Line 20 generates compiler error by attempting to assign new address to constant pointer.

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 `int`s
  - `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 ]`



Pointer Expressions and Pointer Arithmetic

- 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

81

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)

82

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 ]`**

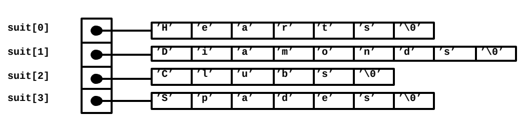
83

Arrays of Pointers

- Arrays can contain pointers
  - Commonly used to store array of strings

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

  - Each element of **suit** points to **`char *`** (a string)
  - Array does not store strings, only pointers to strings



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

84

Function Pointers

- Calling functions using pointers
  - Assume parameter:
    - **`bool ( *compare ) ( int, int )`**
  - Execute function with either
    - **`( *compare ) ( int1, int2 )`**
      - Dereference pointer to function to execute

    OR
    - **`compare( int1, int2 )`**
      - Could be confusing
        - User may think **`compare`** name of actual function in program

Outline

fig05_25.cpp
(1 of 5)

```
// Fig. 5.25: fig05_25.cpp
// Multipurpose sorting program using function pointers.
#include <iostream>

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

#include <iomanip>

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 };

```

Parameter is pointer to function that receives two integer parameters and returns **bool** result.

Outline

fig05_25.cpp
(2 of 5)

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

   // 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 << "\nData items in ascending order\n";
   }

   // sort array in descending order; pass function descending
   // as an agrument to specify descending sorting order
   else {
      bubble( a, arraySize, descending );
      cout << "\nData items in descending order\n";
   }

```


Outline

fig05_25.cpp
(3 of 5)

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

   cout << endl;

   return 0;  // indicates successful termination

} // end main

// multipurpose bubble sort; parameter comp
// the comparison function that determines
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 c
      for ( int count = 0; count < s

         // if adjacent elements are
         if ( (*compare)( work[ count ], work[ count + 1 ] ) )
            swap( &work[ count ], &work[ count + 1 ] );
```

compare is pointer to function that receives two integer parameters and returns **bool** result.

Parentheses necessary to indicate pointer to function

Call passed function **compare**; dereference pointer to execute function.

Outline

fig05_25.cpp
(4 of 5)

```

} // end function bubble

// 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;

} // end function swap

// 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 ascending

```

```
// determine whether elements are out of order
// for a descending order sort
bool descending( int a, int b )
{
   return b > a;   // swap if b is greater than a

} // 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
```

Outline

fig05_25.cpp
(5 of 5)

fig05_25.cpp
output (1 of 1)

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 fu
   // 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;

```

Array initialized with names of three functions; function names are pointers.

Outline

fig05_26.cpp
(1 of 3)


```
   // 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 compl

   return 0;  // indicates successful termination

} // end main

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

} // end function1

```

Call chosen function by dereferencing corresponding element in array.

Outline

fig05_26.cpp
(2 of 3)

93

```
void function2( int b )
{
   cout << "You entered " << b
        << " so function2 was called\n\n";

} // end function2

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

} // end function3
```

```
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.
```

Outline

fig05_26.cpp
(3 of 3)

fig05_26.cpp
output (1 of 1)

94

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

95

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' };**

96

Fundamentals of Characters and Strings

- 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

- 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

`char *strcpy( char *s1, const char *s2 );`	Copies the string **`s2`** into the character array **`s1`**. The value of **`s1`** is returned.
`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.
`char *strcat( char *s1, const char *s2 );`	Appends the string **`s2`** to the string **`s1`**. 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.
`int strcmp( const char *s1, const char *s2 );`	Compares the string **`s1`** with the string **`s2`** The function returns a value of zero, less than zero or greater than zero if **`s1`** is equal to, less than or greater than **`s2`**, respectively.

String Manipulation Functions of the String-handling Library

`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.
`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 to-ken is returned by each call. If there are no more tokens when the function is called, **`NULL`** is returned.
`size_t strlen( const char *s );`	Determines the length of string **`s`** The number of characters preceding the terminating null character is returned.