Outline

- Function Definitions
- Function Prototypes
- Data Types
- Storage Classes and Scope Rules
- C++ Function Call Stack and Activation Records
- References
- Default Arguments
- Function Overloading and Templates
- Recursion
- Q&A
Function Definitions

Three ways to return control to the calling statement:

- If the function does not return a result:
  - Program flow reaches the function-ending right brace or
  - Program executes the statement `return`;

- If the function does return a result:
  - Program executes the statement `return expression;`
    - `expression` is evaluated and its value is returned to the caller.
Function Prototypes and Argument Coercion

Function prototype

- Also called a function declaration
- Indicates to the compiler:
  - Name of the function
  - Type of data returned by the function
  - Parameters the function expects to receive
    - Number of parameters
    - Types of those parameters
    - Order of those parameters
Function prototypes are required in C++. Use `#include` preprocessor directives to obtain function prototypes for the C++ Standard Library functions from the header files for the appropriate libraries (e.g., the prototype for math function `sqrt` is in header file `<cmath>`).
Math Library Functions

Global functions

- Do not belong to a particular class
- Have function prototypes placed in header files
  - Can be reused in any program that includes the header file and that can link to the function’s object code

Example: `sqrt` in `<cmath>` header file

- `sqrt( 900.0 )`
- All functions in `<cmath>` are global functions

### Math library functions Cont’d

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ceil(x)</code></td>
<td>rounds <code>x</code> to the smallest integer not less than <code>x</code></td>
<td><code>ceil(9.2)</code> is 10.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>ceil(-9.8)</code> is -9.0</td>
</tr>
<tr>
<td><code>cos(x)</code></td>
<td>trigonometric cosine of <code>x</code> (in radians)</td>
<td><code>cos(0.0)</code> is 1.0</td>
</tr>
<tr>
<td><code>exp(x)</code></td>
<td>exponential function $e^x$</td>
<td><code>exp(1.0)</code> is 2.71828</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>exp(2.0)</code> is 7.38906</td>
</tr>
<tr>
<td><code>fabs(x)</code></td>
<td>absolute value of <code>x</code></td>
<td><code>fabs(5.1)</code> is 5.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>fabs(0.0)</code> is 0.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>fabs(-8.76)</code> is 8.76</td>
</tr>
<tr>
<td><code>floor(x)</code></td>
<td>rounds <code>x</code> to the largest integer not greater than <code>x</code></td>
<td><code>floor(9.2)</code> is 9.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>floor(-9.8)</code> is -10.0</td>
</tr>
<tr>
<td><code>fmod(x, y)</code></td>
<td>remainder of <code>x/y</code> as a floating-point number</td>
<td><code>fmod(2.6, 1.2)</code> is 0.2</td>
</tr>
<tr>
<td><code>log(x)</code></td>
<td>natural logarithm of <code>x</code> (base $e$)</td>
<td><code>log(2.71828)</code> is 1.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>log(7.389056)</code> is 2.0</td>
</tr>
<tr>
<td><code>log10(x)</code></td>
<td>logarithm of <code>x</code> (base 10)</td>
<td><code>log10(10.0)</code> is 1.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>log10(100.0)</code> is 2.0</td>
</tr>
<tr>
<td><code>pow(x, y)</code></td>
<td><code>x</code> raised to power <code>y</code> ($x^y$)</td>
<td><code>pow(2, 7)</code> is 128</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>pow(9, 0.5)</code> is 3</td>
</tr>
<tr>
<td><code>sin(x)</code></td>
<td>trigonometric sine of <code>x</code> (in radians)</td>
<td><code>sin(0.0)</code> is 0</td>
</tr>
<tr>
<td><code>sqrt(x)</code></td>
<td>square root of <code>x</code> (where <code>x</code> is a nonnegative value)</td>
<td><code>sqrt(9.0)</code> is 3.0</td>
</tr>
<tr>
<td><code>tan(x)</code></td>
<td>trigonometric tangent of <code>x</code> (in radians)</td>
<td><code>tan(0.0)</code> is 0</td>
</tr>
</tbody>
</table>
Function Prototypes and Argument Coercion Cont’d

- **Function signature (or simply signature)**
  - The portion of a function prototype that includes the name of the function and the types of its arguments
  - Does not specify the function’s return type

- **Functions in the same scope must have unique signatures**
  - The scope of a function is the region of a program in which the function is known and accessible
Common Programming Error 1

It is a compilation error if two functions in the same scope have the same signature but different return types.
Function Prototypes and Argument Coercion Cont’d

- Argument Coercion
  - Forcing arguments to the appropriate types specified by the corresponding parameters
    - For example, calling a function with an integer argument, even though the function prototype specifies a double argument
      - The function will still work correctly
Function Prototypes and Argument Coercion Cont’d

C++ Promotion Rules

- Indicate how to convert between types without losing data
- Apply to expressions containing values of two or more data types
  - Such expressions are also referred to as mixed-type expressions
  - Each value in the expression is promoted to the “highest” type in the expression
    - Temporary version of each value is created and used for the expression
      - Original values remain unchanged

Function Prototypes and Argument Coercion Cont’d

- C++ Promotion Rules Cont’d
  - Converting a value to a lower fundamental type
    - Will likely result in the loss of data or incorrect values
    - Can only be performed explicitly
      - By assigning the value to a variable of lower type (some compilers will issue a warning in this case) or
      - By using a cast operator
# Promotion hierarchy for fundamental data types

<table>
<thead>
<tr>
<th>Data types</th>
<th>Size (B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>long double</td>
<td>12</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
</tr>
<tr>
<td>unsigned long int</td>
<td></td>
</tr>
<tr>
<td>long int</td>
<td></td>
</tr>
<tr>
<td>unsigned int</td>
<td></td>
</tr>
<tr>
<td>int</td>
<td></td>
</tr>
<tr>
<td>unsigned short int</td>
<td></td>
</tr>
<tr>
<td>short int</td>
<td></td>
</tr>
<tr>
<td>unsigned char</td>
<td></td>
</tr>
<tr>
<td>char</td>
<td></td>
</tr>
<tr>
<td>bool</td>
<td></td>
</tr>
</tbody>
</table>

- long double: 12 B
- double: 8 B
- float: 4 B
- unsigned long int: synonymous with unsigned long
- long int: synonymous with long
- unsigned int: synonymous with unsigned
- int: synonymous with unsigned
- unsigned short int: synonymous with unsigned short
- short int: synonymous with short
- unsigned char: synonymous with char
- char: 1 B
C++ Standard Library Header Files

- C++ Standard Library header files
  - Each contains a portion of the Standard Library
    - Function prototypes for the related functions
    - Definitions of various class types and functions
    - Constants needed by those functions
  - Header file names ending in `.h`
    - Are "old-style" header files
    - Superseded by the C++ Standard Library header files
Enumeration

- A set of integer constants represented by identifiers
  - The values of enumeration constants start at 0, unless specified otherwise, and increment by 1

- Defining an enumeration
  - Keyword `enum`
  - Comma-separated list of identifier names enclosed in braces
  - `enum Months { JAN = 1, FEB, MAR, APR };`
Common Programming Error 2

Assigning the integer equivalent of an enumeration constant to a variable of the enumeration type is a compilation error.
Storage Classes

- Each identifier has several attributes:
  - Name, type, size and value
  - Also storage class, scope and linkage

- C++ provides five storage-class specifiers:
  - auto, register, extern, mutable and static

- Identifier’s storage class:
  - Determines the period during which that identifier exists in memory
Storage Classes Cont’d

- Identifier’s scope
  - Determines where the identifier can be referenced in a program

- Identifier’s linkage
  - Determines whether an identifier is known only in the source file where it is declared or across multiple files that are compiled, then linked together

- An identifier’s storage-class specifier helps determine its storage class and linkage
Storage Classes Cont’d

- **Automatic storage class**
  - Declared with keywords `auto` and `register`

- **Automatic variables**
  - Created when program execution enters block in which they are defined
  - Exist while the block is active
  - Destroyed when the program exits the block

- Only local variables and parameters can be of automatic storage class
  - Such variables normally are of automatic storage class
Performance Tip 1

Automatic storage is a means of conserving memory, because automatic storage class variables exist in memory only when the block in which they are defined is executing.
Performance Tip 2

The storage-class specifier register can be placed before an automatic variable declaration to suggest that the compiler maintain the variable in one of the computer’s high-speed hardware registers. If intensely used variables such as counters or totals are maintained in hardware registers, the overhead of repeatedly loading the variables from memory into the registers and storing the results back into memory is eliminated.
Storage Classes Cont’d

- Storage-class specifier auto
  - Explicitly declares variables of automatic storage class
  - Local variables are of automatic storage class by default
    - So keyword auto rarely is used
Storage Classes Cont’d

Storage-class specifier register

- Data in the machine-language version of a program is normally loaded into registers for calculations and other processing
  - Compiler tries to store register storage class variables in a register
  - The compiler might ignore register declarations
    - May not be sufficient registers for the compiler to use
Common Programming Error 3

Using multiple storage-class specifiers for an identifier is a syntax error. Only one storage class specifier can be applied to an identifier. For example, if you include `register`, do not also include `auto`.
Performance Tip 3

Often, register is unnecessary. Today’s optimizing compilers are capable of recognizing frequently used variables and can decide to place them in registers without needing a register declaration from the programmer.
Storage Classes Cont’d

Two types of identifiers with static storage class

- External identifiers
  - Such as global variables and global function names

- Local variables declared with the storage class specifier `static`

Global variables

- Created by placing variable declarations outside any class or function definition
- Can be referenced by any function
Storage Classes Cont’d

- Local variables declared with static
  - Known only in the function in which they are declared
  - Retain their values when the function returns to its caller
    - Next time the function is called, the static local variables contain the values they had when the function last completed
  - If numeric variables of the static storage class are not explicitly initialized by the programmer
    - They are initialized to zero
Scope Rules

- Scope
  - Portion of the program where an identifier can be used
  - Four scopes for an identifier
    - Function scope
    - File scope
    - Block scope
    - Function-prototype scope
Scope Rules Cont’d

File scope
- For an identifier declared outside any function or class
- Global variables, function definitions and function prototypes placed outside a function all have file scope

Function scope
- Labels (identifiers followed by a colon such as start:) are the only identifiers with function scope
  - Cannot be referenced outside the function body
  - Labels are implementation details that functions hide from one another
Scope Rules Cont’d

**Block scope**

- Identifiers declared inside a block have block scope
  - Block scope begins at the identifier’s declaration
  - Block scope ends at the terminating right brace (}) of the block in which the identifier is declared

- Local variables and function parameters have block scope
  - The function body is their block
Scope Rules Cont’d

Block scope Cont’d

- Any block can contain variable declarations
- Identifiers in an outer block can be “hidden” when a nested block has a local identifier with the same name
- Local variables declared static still have block scope, even though they exist from the time the program begins execution
  - Storage duration does not affect the scope of an identifier
Scope Rules Cont’d

Function-prototype scope

- Only identifiers used in the parameter list of a function prototype have function-prototype scope
- Parameter names appearing in a function prototype are ignored by the compiler
  - Identifiers used in a function prototype can be reused elsewhere in the program without ambiguity
  - However, in a single prototype, a particular identifier can be used only once
Scoping Example

1 // Fig. 6.12: fig06_12.cpp
2 // A scoping example.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 void useLocal(void); // function prototype
8 void useStaticLocal(void); // function prototype
9 void useGlobal(void); // function prototype
10
11 int x = 1; // global variable
12
13 int main()
14 {
15    int x = 5; // local variable to main
16    cout << "local x in main's outer scope is " << x << endl;
17
18       // start new scope
19       int x = 7; // hides x in outer scope
20       cout << "local x in main's inner scope is " << x << endl;
21       // end new scope
22    cout << "local x in main's outer scope is " << x << endl;
23}
24
25 cout << "local x in main's outer scope is " << x << endl;
Scoping Example Cont'd

// useLocal has local x
useLocal();

// useStaticLocal has static local x
useStaticLocal();

// useGlobal uses global x
useGlobal();

// useLocal reinitializes its local x
useLocal();

// static local x retains its prior value
useStaticLocal();

// global x also retains its value
useGlobal();

// useLocal reinitializes local variable x during each call
void useLocal(void)
{
    int x = 25; // initialized each time useLocal is called
    cout << "\nlocal x is " << x << " on entering useLocal" << endl;
    x++;
    cout << "local x is " << x << " on exiting useLocal" << endl;
}

// end function useLocal

// Local variable that gets recreated and reinitialized each time useLocal is called

void useStaticLocal( void )
{
    static int x = 50; // initialized first time useStaticLocal is called

    cout << "\nlocal static x is " << x << " on entering useStaticLocal" << endl;
    x++;
    cout << "local static x is " << x << " on exiting useStaticLocal" << endl;
} // end function useStaticLocal

void useGlobal( void )
{
    cout << "\nglobal x is " << x << " on entering useGlobal" << endl;
    x *= 10;
    cout << "global x is " << x << " on exiting useGlobal" << endl;
} // end function useGlobal

// useGlobal modifies global variable x during each call

// useStaticLocal initializes static local variable x only the
// first time the function is called; value of x is saved
// between calls to this function

// static local variable that gets initialized only once

// Statement refers to global variable x because no local variable named x exists
Scoping Example Cont’d

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

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

local static x is 51 on entering useStaticLocal
local static x is 52 on exiting useStaticLocal

global x is 10 on entering useGlobal
global x is 100 on exiting useGlobal

local x in main is 5
Function Call Stack and Activation Records

- Data structure: collection of related data items
- Stack data structure
  - Analogous to a pile of dishes
  - When a dish is placed on the pile, it is normally placed at the top
    - Referred to as pushing the dish onto the stack
Function Call Stack and Activation Records Cont’d

- Stack data structure Cont’d

  - Similarly, when a dish is removed from the pile, it is normally removed from the top
    - Referred to as popping the dish off the stack

  - A last-in, first-out (LIFO) data structure
    - The last item pushed (inserted) on the stack is the first item popped (removed) from the stack

Function Call Stack and Activation Records Cont’d

Function Call Stack

- Sometimes called the program execution stack
- Supports the function call/return mechanism

Each time a function calls another function, a stack frame (also known as an activation record) is pushed onto the stack

- Maintains the return address that the called function needs to return to the calling function
- Contains automatic variables—parameters and any local variables the function declares
Function Call Stack and Activation Records Cont’d

- When the called function returns
  - Stack frame for the function call is popped
  - Control transfers to the return address in the popped stack frame

- If a function makes a call to another function
  - Stack frame for the new function call is simply pushed onto the call stack
  - Return address required by the newly called function to return to its caller is now located at the top of the stack.
Function Call Stack and Activation Records Cont’d

- Stack overflow
  - Error that occurs when more function calls occur than can have their activation records stored on the function call stack (due to memory limitations)
// Fig. 6.13: fig06_13.cpp
// square function used to demonstrate the function
// call stack and activation records.
#include <iostream>
using namespace std;

int square(int); // prototype for function square

int main() {
    int a = 10; // value to square (local automatic variable in main)
    cout << a << " squared: " << square(a) << endl; // display a squared
    return 0; // indicate successful termination
}

// returns the square of an integer
int square(int x) { // x is a local variable
    return x * x; // calculate square and return result
}

10 squared: 100
Function call stack after the operating system invokes main

Step 1: Operating system invokes main to execute application.

```cpp
int main()
{
    int a = 10;
    cout << a << " squared: "
        << square(a) << endl;
    return 0;
}
```

Operating system calls main, pushing an activation record onto the stack.
Function call stack after main invokes function square

Step 2: main invokes function square to perform calculation.

```cpp
int main()
{
    int a = 10;
    cout << a << " squared: " << square(a) << endl;
    return 0;
}
```

```
int square( int x )
{
    return x * x;
}
```

**main** calls function **square**, pushing another stack frame onto the function call stack.
Function call stack after function square returns to main.

Program control returns to main and square's stack frame is popped off.
Inline Functions

- Inline functions
  - Reduce function call overhead—especially for small functions
  - Qualifier `inline` before a function’s return type in the function definition
    - “Advises” the compiler to generate a copy of the function’s code in place (when appropriate) to avoid a function call
Inline Functions Cont’d

- Inline functions Cont’d

  - Trade-off of inline functions
    - Multiple copies of the function code are inserted in the program (often making the program larger)

  - The compiler can ignore the inline qualifier and typically does so for all but the smallest functions
Any change to an inline function could require all clients of the function to be recompiled. This can be significant in some program development and maintenance situations.
Good Programming Practice 1

The **inline** qualifier should be used only with small, frequently used functions.
Performance Tip 4

Using inline functions can reduce execution time but may increase program size.
The const qualifier should be used to enforce the principle of least privilege. Using the principle of least privilege to properly design software can greatly reduce debugging time and improper side effects and can make a program easier to modify and maintain.
// Fig. 6.18: fig06_18.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
}

int main()
{
    double sideValue; // stores value entered by user
    cout << "Enter the side length of your cube: ";
    cin >> sideValue; // read value from user

    // calculate cube of sideValue and display result
    cout << "Volume of cube with side " << sideValue << " is " << cube(sideValue) << endl;
    return 0; // indicates successful termination
}

Enter the side length of your cube: 3.5
Volume of cube with side 3.5 is 42.875
References and Reference Parameters

Two ways to pass arguments to functions

- **Pass-by-value**
  - A copy of the argument’s value is passed to the called function
  - Changes to the copy do not affect the original variable’s value in the caller
    - Prevents accidental side effects of functions

- **Pass-by-reference**
  - Gives called function the ability to access and modify the caller’s argument data directly
References and Reference Parameters Cont’d

Reference Parameter

- An alias for its corresponding argument in a function call
- & placed after the parameter type in the function prototype and function header
- Example

```cpp
int &count in a function header
```
- Pronounced as “count is a reference to an int”

- Parameter name in the body of the called function actually refers to the original variable in the calling function
```cpp
// Fig. 6.19: fig06_19.cpp
// Comparing pass-by-value and pass-by-reference with references.
#include <iostream>
using std::cout;
using std::endl;

int squareByValue(int); // function prototype (value pass)
void squareByReference(int &); // function prototype (reference pass)

int main()
{
    int x = 2; // value to square using squareByValue
    int z = 4; // value to square using squareByReference

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

    // demonstrate squareByReference
    cout << "z = " << z << " before squareByReference\n";
    squareByReference(z);
    cout << "z = " << z << " after squareByReference\n" << endl;
    return 0; // indicates successful termination
} // end main
```

Parameter Passing Cont’d

```c++
// 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
} // end function squareByValue

// 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
} // end function squareByReference

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

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

Receives copy of argument in `main`

Receives reference to argument in `main`

Modifies variable in `main`
Many programmers do not bother to declare parameters passed by value as const, even though the called function should not be modifying the passed argument. Keyword const in this context would protect only a copy of the original argument, not the original argument itself, which when passed by value is safe from modification by the called function.
For the combined reasons of clarity and performance, many C++ programmers prefer that modifiable arguments be passed to functions by using pointers, small nonmodifiable arguments be passed by value and large nonmodifiable arguments be passed to functions by using references to constants.
References and Reference Parameters Cont’d

References

- Can also be used as aliases for other variables within a function
  - All operations supposedly performed on the alias (i.e., the reference) are actually performed on the original variable
  - Must be initialized in their declarations
    - Cannot be reassigned afterward

- Example

```cpp
int count = 1;
int &cRef = count;
cRef ++;
```

References and Reference Parameters Cont’d

- Returning a reference from a function
  - Functions can return references to variables
    - Should only be used when the variable is static
  - Dangling reference
    - Returning a reference to an automatic variable
      - That variable no longer exists after the function ends
Common Programming Error 4

Attempting to reassign a previously declared reference to be an alias to another variable is a logic error. The value of the other variable is simply assigned to the variable for which the reference is already an alias.
Default Arguments

Default argument

- A default value to be passed to a parameter
  - Used when the function call does not specify an argument for that parameter
- Must be the rightmost argument(s) in a function’s parameter list
- Should be specified with the first occurrence of the function name
  - Typically the function prototype
// Fig. 6.22: fig06_22.cpp
// Using default arguments.
#include <iostream>

using namespace std;

int boxVolume(int length = 1, int width = 1, int height = 1);

int main()
{
    // no arguments--use default values for all dimensions
    cout << "The default box volume is: " << boxVolume();

    // specify length; default width and height
    cout << "The volume of a box with length 10, width 1 and height 1 is: " << boxVolume(10);

    // specify length and width; default height
    cout << "The volume of a box with length 10, width 5 and height 1 is: " << boxVolume(10, 5);

    // specify all arguments
    cout << "The volume of a box with length 10, width 5 and height 2 is: " << boxVolume(10, 5, 2)
    << endl;
    return 0;  // indicates successful termination
} // end main

Default arguments

Calling function with no arguments; uses three defaults

Calling function with one argument; uses two defaults

Calling function with two arguments; uses one default

Calling function with three arguments; uses no defaults
// function boxVolume calculates the volume of a box
int boxVolume( int length, int width, int height )
{
    return length * width * height;
} // end function boxVolume

The default box volume is: 1

The volume of a box with length 10, width 1 and height 1 is: 10

The volume of a box with length 10, width 5 and height 1 is: 50

The volume of a box with length 10, width 5 and height 2 is: 100

Note that default arguments were specified in the function prototype, so they are not specified in the function header.

Unary Scope Resolution Operator

- Unary scope resolution operator (::)

  - Used to access a global variable when a local variable of the same name is in scope
  - Cannot be used to access a local variable of the same name in an outer block
Unary Scope Resolution Operator Example

```cpp
// Fig. 6.23: fig06_23.cpp
// Using the unary scope resolution operator.
#include <iostream>
using std::cout;
using std::endl;

int number = 7; // global variable named number

int main()
{
    double number = 10.5; // local variable named number

    // display values of local and global variables
    cout << "Local double value of number = " << number
         << "\nGlobal int value of number = " << ::number << endl;
    return 0; // indicates successful termination
} // end main
```

Local double value of number = 10.5
Global int value of number = 7

Unary scope resolution operator used to access global variable `number`
Function Overloading

- Overloaded functions
  - Overloaded functions have
    - Same name
    - Different sets of parameters
  - Compiler selects proper function to execute based on number, types and order of arguments in the function call
  - Commonly used to create several functions of the same name that perform similar tasks, but on different data types
Good Programming Practice 2

Overloading functions that perform closely related tasks can make programs more readable and understandable.

Function Overloading Cont’d

How the compiler differentiates overloaded functions

- Overloaded functions are distinguished by their signatures
- Name mangling or name decoration
  - Compiler encodes each function identifier with the number and types of its parameters to enable type-safe linkage
- Type-safe linkage ensures that
  - Proper overloaded function is called
  - Types of the arguments conform to types of the parameters

Common Programming Error 5

Creating overloaded functions with identical parameter lists and different return types is a compilation error.
Common Programming Error 6

A function with default arguments omitted might be called identically to another overloaded function; this is a compilation error. For example, having in a program both a function that explicitly takes no arguments and a function of the same name that contains all default arguments results in an error when an attempt is made to use that function name in a call passing no arguments. The compiler does not know which function to choose.

Function Templates

**Function templates**

- **More compact and convenient form of overloading**
  - Identical program logic and operations for each data type

- **Function template definition**
  - Defines a whole family of overloaded functions
  - Begins with the `template` keyword
  - Contains template parameter list of formal type parameters for the function template enclosed in angle brackets (`<>`)

- **Formal type parameters**
  - Preceded by keyword `typename` or `class`
Function Templates Cont'd

Function-template specializations

- Generated automatically by the compiler to handle each type of call to the function template

- Example for function template \texttt{max} with type parameter \texttt{T} called with \texttt{int} arguments
  - Compiler detects a \texttt{max} invocation in the program code
  - \texttt{int} is substituted for \texttt{T} throughout the template definition
  - This produces function-template specialization \texttt{max<int>
// Fig. 6.26: maximum.h
// Definition of function template maximum.

// or template< typename T >
template <class T> // or template< typename T >
T maximum( T value1, T value2, T value3 )
{
    T maxValue = value1; // assume value1 is maximum
    // determine whether value2 is greater than maxValue
    if ( value2 > maxValue )
        maxValue = value2;

    // determine whether value3 is greater than maxValue
    if ( value3 > maxValue )
        maxValue = value3;

    return maxValue;
} // end function template maximum

Using formal type parameter T in place of data type
// Fig. 6.27: fig06_27.cpp
// Function template maximum test program.
#include <iostream>
using std::cout;
using std::cin;
using std::endl;

#include "maximum.h" // include definition of function template maximum

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);
}

Invoking maximum with int arguments
Invoking maximum with double arguments
// 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
Recursion

- **Factorial**
  - The factorial of a nonnegative integer \( n \), written \( n! \) (and pronounced “\( n \) factorial”), is the product
    - \( n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1 \)
  - **Recursive definition of the factorial function**
    - \( n! = n \cdot (n-1)! \)

- **Example**
  - \( 5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 \)
    - \( 5! = 5 \cdot (4 \cdot 3 \cdot 2 \cdot 1) \)
    - \( 5! = 5 \cdot (4!) \)

Recursive Factorial Function

```
1 // Fig. 6.29: fig06_29.cpp
2 // Testing the recursive factorial function.
3 #include <iostream>
4 using std::cout;
5 using std::endl;

7 #include <iomanip>
8 using std::setw;
9
10 unsigned long factorial(unsigned long); // function prototype
11
12 int main()
13 {
14    // calculate the factorials of 0 through 10
15    for (int counter = 0; counter <= 10; counter++)
16       cout << setw(2) << counter << "! = " << factorial(counter) << endl;
17
18    return 0; // indicates successful termination
19 } // end main
```

First call to `factorial` function
Recursive Factorial Function
Cont’d

```c
// recursive definition of function factorial
unsigned long factorial(unsigned long number) {
    if (number <= 1) // test for base case
        return 1;    // base cases: 0! = 1 and 1! = 1
    else // recursion step
        return number * factorial(number - 1);
} // end function factorial
```

Base cases simply return 1

Recursive call to `factorial` function with a slightly smaller problem

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

The Fibonacci series

- 0, 1, 1, 2, 3, 5, 8, 13, 21, ...
- Begins with 0 and 1
- Each subsequent Fibonacci number is the sum of the previous two Fibonacci numbers

Can be defined recursively as follows:

- \( \text{fibonacci}(0) = 0 \)
- \( \text{fibonacci}(1) = 1 \)
- \( \text{fibonacci}(n) = \text{fibonacci}(n-1) + \text{fibonacci}(n-2) \)
Set of recursive calls to function fibonacci
// Fig. 6.32: fig06_32.cpp
// Testing the iterative factorial function.
#include <iostream>
using std: : cout;
using std: : endl;

#include <iomanip>
using std: : setw;

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

int main()
{
   // calculate the factorials of 0 through 10
   for ( int counter = 0; counter <= 10; counter++ )
      cout << setw( 2 ) << counter << "! = " << factorial( counter ) << endl;

   return 0;
} // end main

// iterative function factorial
unsigned long factorial( unsigned long number )
{
   unsigned long result = 1;

Iterative Factorial Function Cont’d

```
// iterative declaration of function factorial
for (unsigned long i = number; i >= 1; i--)
    result *= i;
return result;
} // end function factorial
```

Iterative approach to finding a factorial

<table>
<thead>
<tr>
<th>Factorial</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0!</td>
<td>1</td>
</tr>
<tr>
<td>1!</td>
<td>1</td>
</tr>
<tr>
<td>2!</td>
<td>2</td>
</tr>
<tr>
<td>3!</td>
<td>6</td>
</tr>
<tr>
<td>4!</td>
<td>24</td>
</tr>
<tr>
<td>5!</td>
<td>120</td>
</tr>
<tr>
<td>6!</td>
<td>720</td>
</tr>
<tr>
<td>7!</td>
<td>5040</td>
</tr>
<tr>
<td>8!</td>
<td>40320</td>
</tr>
<tr>
<td>9!</td>
<td>362880</td>
</tr>
<tr>
<td>10!</td>
<td>3628800</td>
</tr>
</tbody>
</table>

Recursion vs Iteration

Negatives of recursion

- Overhead of repeated function calls
  - Can be expensive in both processor time and memory space
- Each recursive call causes another copy of the function (actually only the function’s variables) to be created
  - Can consume considerable memory

Iteration

- Normally occurs within a function
- Overhead of repeated function calls and extra memory assignment is omitted
Any problem that can be solved recursively can also be solved iteratively (nonrecursively). A recursive approach is normally chosen in preference to an iterative approach when the recursive approach more naturally mirrors the problem and results in a program that is easier to understand and debug. Another reason to choose a recursive solution is that an iterative solution is not apparent.