CHAPTER 4
FUNCTIONS

Dr. Shady Yehia Elmashad

Outline

1. Introduction
2. Program Components in C++
3. Math Library Functions
4. Functions
5. Function Definitions
6. Function Prototypes
7. Header Files
8. Recursion
9. Example Using Recursion: The Fibonacci Series
10. Recursion vs. Iteration
11. Functions with Empty Parameter Lists
12. Inline Functions
13. Default Arguments
14. Function Overloading

1. Introduction

- Divide and conquer
  - ➢ Construct a program from smaller pieces or components.
  - ➢ Each piece more manageable than the original program.

2. Program Components in C++

- Programs written by
  - combining new functions with “prepackaged” functions in the C++ standard library.
  - The standard library provides a rich collection of functions.

- Functions are invoked by a function call
  - A function call specifies the function name and provides information (as arguments) that the called function needs
  - Boss to worker analogy:

    A boss (the calling function or caller) asks a worker (the called function) to perform a task and return (i.e., report back) the results when the task is done.

2. Program Components in C++

- Function definitions
  - Only written once
  - These statements are hidden from other functions.
  - Boss to worker analogy:

    The boss does not know how the worker gets the job done; he just wants it done

3. Math Library Functions

- Math library functions
  - Allow the programmer to perform common mathematical calculations
  - Are used by including the header file **`<cmath>`**
- Functions called by writing

  functionName (*argument*)
- Example

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

  - Calls the **`sqrt`** (square root) function. The preceding statement would print **`30`**
  - The **`sqrt`** function takes an argument of type **`double`** and returns a result of type **`double`**, as do all functions in the math library

3. Math Library Functions

- Function arguments can be
  - Constants

```
sqrt( 4 );
```

  - Variables

```
sqrt( x );
```

  - Expressions

```
sqrt( sqrt( x ) );
sqrt( 3 - 6x );
```

4. Functions

- Functions
  - ➢ Allow the programmer to modularize a program
- 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 when the function is called that provide the function with outside information

5. Function Definitions

- Create customized functions to
  - ➢ Take in data
  - ➢ Perform operations
  - ➢ Return the result
- Format for function definition:

```
return-value-type  function-name( parameter-list )
  {
      declarations and statements
  }
```

- Example:

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

Outline

```cpp
// Fig. 3.3: fig03_03.cpp
// Creating and using a programmer-defined function
#include <iostream>

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

int square( int );   // function prototype

int main()
{
   for ( int x = 1; x <= 10; x++ )
      cout << square( x ) << "  ";

   cout << endl;
   return 0;
}

// Function definition
int square( int y )
{
   return y * y;
}
```

Notice how parameters and return value are declared.

1. Function prototype

2. Loop

3. Function definition

```
1   4   9   16   25   36   49   64   81   100
```

Program Output

```
// Fig. 3.4: fig03_04.cpp
// Finding the maximum of three integers
#include <iostream>

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

int maximum( int, int, int );   // function prototype

int main()
{
   int a, b, c;

   cout << "Enter three integers: ";
   cin >> a >> b >> c;

   // a, b and c below are arguments to
   // the maximum function call
   cout << "Maximum is: " << maximum( a, b, c ) << endl;
```

Outline

1. Function prototype (3 parameters)

2. Input values

2.1 Call function

```

   return 0;
}

// Function maximum definition
// x, y and z below are parameters to
// the maximum function definition
int maximum( int x, int y, int z )
{
   int max = x;

   if ( y > max )
      max = y;

   if ( z > max )
      max = z;

   return max;
}
```

```
Enter three integers: 22 85 17
Maximum is: 85

Enter three integers: 92 35 14
Maximum is: 92

Enter three integers: 45 19 98
Maximum is: 98
```

Outline

3. Function definition

Program Output

6. Function Prototypes

- Function prototype
  - ➢ Function name
  - ➢ Parameters
    - Information the function takes in
  - ➢ Return type
    - Type of information the function passes back to caller (default **`int`**)
    - **`void`** signifies the function returns nothing
  - ➢ Only needed if function definition comes after the function call in the program
- Example:

```
int maximum( int, int, int );
```

  - ➢ Takes in 3 **`int`**s
  - ➢ Returns an **`int`**

7. Header Files

- Header files
  - ➢ Contain function prototypes for library functions
  - ➢ **`<cstdlib>`** , **`<cmath>`**, etc.
  - ➢ Load with **`#include <filename>`**

    - Example:

```
#include <cmath>
```

- Custom header files
  - ➢ Defined by the programmer
  - ➢ Save as **`filename.h`**
  - ➢ Loaded into program using

```
#include "filename.h"
```

Room Area (Rectangle)

```
# include < iostream.h >
float findArea ( float ,float ) ;                    // Function declaration (prototype)
void main ( )  {
float room_L, room_W, room_Area;
cout << “ Please enter the room width “ ;
cin >> room_W ;
cout << “ Please enter the room length “ ;
cin >> room_L ;

room_Area  =  findArea (room_W, room_L );       // Function call

cout << “ The area of your room is: :”  << room_Area  << “ square unit ” ;
}

float findArea ( L , W )   {                   // Function definition
float area;
Area = L * W ;
return Area ;
}
```

Celsius to Fahrenheit Temperature Converter

```
# include < iostream.h >
float convert ( float ) ;                    // Function declaration (prototype)
void main ( )  {
float Temp_Fah,  Temp_Cen;
cout << “ Please enter the temperature in fahrenheit “ ;
cin >> Temp_Fah ;

Temp_Cen  = convert ( Temp_Fah ) ;     // Function call

cout << “ The temperature in centigrade is: :”  << Temp_Cen ;
}

float covert ( float Fah )   {                      // Function definition
float Cen;
Cen = (Fah – 32 ) * (5 / 9 );
return Cen ;
}
```

Odd or Even

```
# include < iostream.h >

void odd_even ( int ) ;                    // Function declaration (prototype)

void main ( )  {
int number ;
cout << " Please enter a number: " ;
cin >> number ;
odd_even ( number ) ;                      // Function call
}

void odd_even ( int number )   {      // Function definition
if ( number % 2 = = 0 )
cout << " you number is even ";
else
cout << " you number is odd ";
}
```

Positive or Negative

```
# include < iostream.h >

void poitive_negative ( int ) ;          // Function declaration (prototype)

void main ( )  {
int number ;
cout << “ Please enter a number: “ ;
cin >> number ;
poitive_negative ( number ) ;          // Function call
}

void poitive_negative ( int number )   {          // Function definition
if ( number > 0 )
cout << “ you number is positive “;
else
cout << “ you number is negative“;
}
```

Swap

```
# include < iostream.h >
void Swap( int , int ) ;                    // Function declaration (prototype)
void main ( )  {
int n1, n2 ;
cout << " Please enter the value of number 1" ;
cin >> n1 ;
cout << " Please enter the value of number 2" ;
cin >> n2 ;

Swap ( n1, n2 ) ;      // Function call
}

void Swap  ( int n1 , int n2 )   {                    // Function definition
int temp ;
temp = n1;
n1 = n2;
n2 = temp;
cout << " The value stored in number 1 is now: " << n1 << endl ;
cout << " The value stored in number 2 is now: " << n2 << endl ;
}
```

Product and Quotient of two numbers

```
# include < iostream.h >
float Product ( float  , float ) ;                    // Function declaration (prototype)
float Quotient( float  , float ) ;                    // Function declaration (prototype)
void main ( )  {
int n1, n2 ;
cout << “ Please enter two numbers “ ;
cin >> a >>b ;

R1 = Product (a, b);                    // Function call
R2 = Quotient (a, b);                   // Function call
cout << “the product of them is” << R1 <<“ and the division is” << R2;
}

float Product  ( float a , float b )   {                  // Function definition
return a*b;
}
float Quotient ( float a , float b )   {                  // Function definition
If (b ! = 0)  {
float Q = a / b;
return Q;  }
else   cout << “ You couldn’t divide by zero. “ ;
}
```

Draw the following Pattern

```
# include < iostream.h >
void draw_line ( void ) ;                // Function declaration (prototype)
void main ( )  {

draw_line ( ) ;                          // Function call
draw_line ( ) ;                          // Function call
cout << “ Welcome “ << endl ;
draw_line ( ) ;                          // Function call
cout << “ First Year “ ;
draw_line ( ) ;                          // Function call
draw_line ( ) ;                          // Function call
}

void draw_line ( void )   {            // Function definition
for ( int i = 0 ; i < 5 ; i ++ )   {
cout << “ * “ ; }
cout << endl ;
}
```

```
* * * * *
* * * * *
Welcome
* * * * *
First Year
* * * * *
* * * * *
```

8. Recursion

- Recursive functions
  - ➢ Are functions that calls themselves
  - ➢ Can only solve a base case
  - ➢ If not base case, the function breaks the problem into a slightly smaller, slightly simpler, problem that resembles the original problem and
    - - Launches a new copy of itself to work on the smaller problem, slowly converging towards the base case
    - - Makes a call to itself inside the **`return`** statement
  - ➢ Eventually the base case gets solved and then that value works its way back up to solve the whole problem

8. Recursion

- Example: factorial

  *n! = n * (n – 1) * (n – 2) * … * 1*

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

    *5! = 5 * 4!*

    *4! = 4 * 3!…*

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

9. Example Using Recursion: The Fibonacci Series

- Fibonacci series: 0, 1, 1, 2, 3, 5, 8...
  - ➢ Each number sum of two previous ones
  - ➢ Example of a recursive formula:

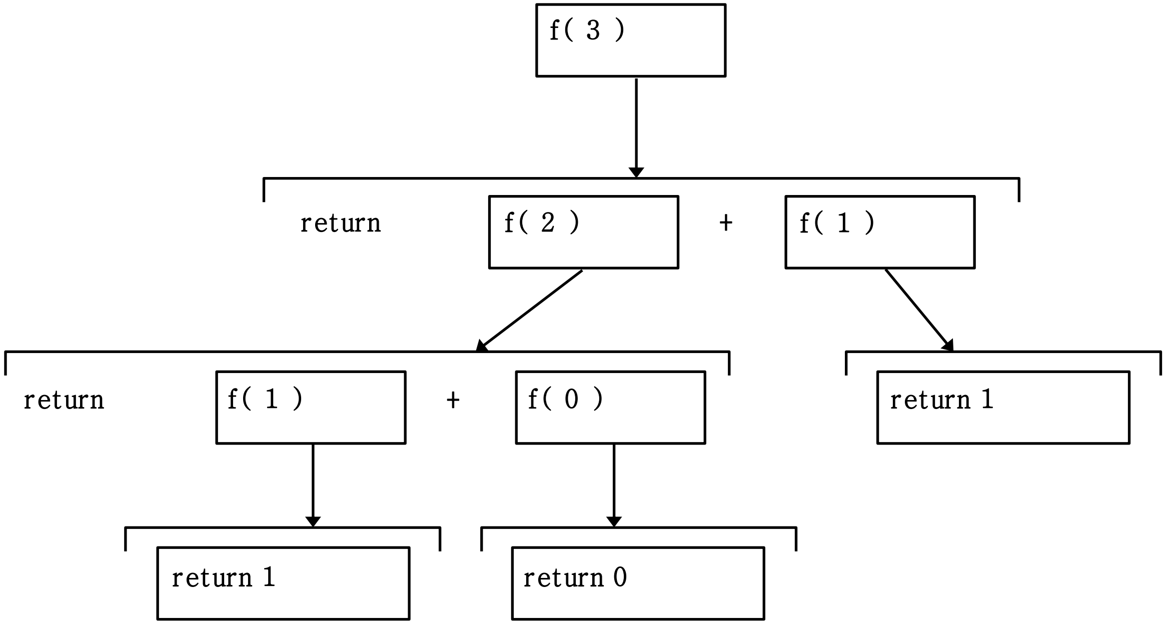
```
fib(n) = fib(n-1) + fib(n-2)
```

- 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 );
}
```

9. Example Using Recursion: The Fibonacci Series

- Diagram of Fibonnaci function



```cpp
// Fig. 3.15: fig03_15.cpp
// Recursive fibonacci function
#include <iostream>

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

unsigned long fibonacci( unsigned long );

int main()
{
    unsigned long result, number;

    cout << "Enter an integer: ";
    cin >> number;
    result = fibonacci( number );
    cout << "Fibonacci(" << number << ") = " << result << endl;
    return 0;
}

// Recursive definition of function fibonacci
unsigned long fibonacci( unsigned long n )
{
   if ( n == 0 || n == 1 )  // base case
      return n;
   else                     // recursive case
      return fibonacci( n - 1 ) + fibonacci( n - 2 );
}
```

Only the base cases return values. All other cases call the **`fibonacci`** function again.

Outline

1. Function prototype

1.1 Initialize variables

2. Input an integer

2.1 Call function `fibonacci`

2.2 Output results.

3. Define `fibonacci` recursively

Program Output

```
Enter an integer: 0
Fibonacci(0) = 0

Enter an integer: 1
Fibonacci(1) = 1

Enter an integer: 2
Fibonacci(2) = 1

Enter an integer: 3
Fibonacci(3) = 2

Enter an integer: 4
Fibonacci(4) = 3

Enter an integer: 5
Fibonacci(5) = 5

Enter an integer: 10
Fibonacci(10) = 55

Enter an integer: 6
Fibonacci(6) = 8

Enter an integer: 20
Fibonacci(20) = 6765

Enter an integer: 30
Fibonacci(30) = 832040

Enter an integer: 35
Fibonacci(35) = 9227465
```

10. 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)

11. Functions with Empty Parameter Lists

- Empty parameter lists
  - Either writing **`void`** or leaving a parameter list empty indicates that the function takes no arguments

```
void print();
```

or

```
void print( void );
```

  - Function **`print`** takes no arguments and returns no value

Outline

```
// Fig. 3.18: fig03_18.cpp
// Functions that take no arguments
#include <iostream>

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

void function1();
void function2( void );

int main()
{
   function1();
   function2();

   return 0;
}

void function1()
{
   cout << "function1 takes no arguments" << endl;
}

void function2( void )
{
   cout << "function2 also takes no arguments" << endl;
}
```

Notice the two ways of declaring no arguments.

1. Function prototypes (take no arguments)

2. Call the functions

3. Function definitions

```
function1 takes no arguments
function2 also takes no arguments
```

Program Output

12. Inline Functions

- **`inline`** functions
  - ➢ Reduce function-call overhead
  - ➢ Asks the compiler to copy code into program instead of using a function call
  - ➢ Compiler can ignore **`inline`**
  - ➢ Should be used with small, often-used functions
- Example:

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

13. Default Arguments

- If function parameter omitted, gets default value
  - ➢ Can be constants, global variables, or function calls
  - ➢ If not enough parameters specified, rightmost go to their defaults
- Set defaults in function prototype

```
int defaultFunction( int x = 1,
   int y = 2, int z = 3 );
```

```cpp
// Fig. 3.23: fig03_23.cpp
// Using default arguments
#include <iostream>

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

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

int main()
{
   cout << "The default box volume is: " << boxVolume()
        << "\n\nThe volume of a box with length 10,\n"
        << "width 1 and height 1 is: " << boxVolume( 10 )
        << "\n\nThe volume of a box with length 10,\n"
        << "width 5 and height 1 is: " << boxVolume( 10, 5 )
        << "\n\nThe volume of a box with length 10,\n"
        << "width 5 and height 2 is: " << boxVolume( 10, 5, 2 )
        << endl;

   return 0;
}

// Calculate the volume of a box
int boxVolume( int length, int width, int height )
{
   return length * width * height;
}
```

Outline

1. Function prototype

2. Print default volume

2.1 Print volume with one parameter

2.2 Print with 2 parameters

2.3 Print with all parameters.

3. Function definition

Program Output

```
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
```

Notice how the rightmost values are defaulted.

14. Function Overloading

- Function overloading
  - ➢ Having functions with same name and different parameters
  - ➢ Should perform similar tasks (i.e., a 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; }
```

  - ➢ Program chooses function by signature
    - signature determined by function name and parameter types
  - ➢ Can have the same return types

Outline

```cpp
// Fig. 3.25: fig03_25.cpp
// Using overloaded functions
#include <iostream>

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

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

double square( double y ) { return y * y; }

int main()
{
   cout << "The square of integer 7 is " << square( 7 )
        << "\nThe square of double 7.5 is " << square( 7.5 )
        << endl;

   return 0;
}
```

Functions have same name but different parameters

1. Define overloaded function

2. Call function

```
The square of integer 7 is 49
The square of double 7.5 is 56.25
```

Program Output

Example: What is the O/P?

```
# include < iostream.h >
int number = 10 ;
void display ( void ) ;
void main ( )   {
int number = 20 ;
cout << “ The value of the number is “ << number << endl ;
display ( ) ;
}
void display ( void ) {
cout << “ The value of the number now is “ << number ;
}
```

```
Output:          The value of the number is 20
                 The value of the number now is 10
```