

W 3.1, 3.2

C++ Functions Modular Program Design

Introduction to C and C++, CA212 © Brian Stone 2001

1

Functions

- Functions allow a program to be broken down into smaller units of code.
- Modular programming is facilitated thus in C and C++
- C is a small language, extendible by using libraries of functions.
- Check the standard function libraries.

Introduction to C and C++, CA212 © Brian Stone 2001

2

Functions

- **main()** the main function of any program.
- Libraries available, e.g. maths library.
 - sqrt(x)
 - exp(x)
 - log(x)
- User defined functions, build your own.

Introduction to C and C++, CA212 © Brian Stone 2001

3

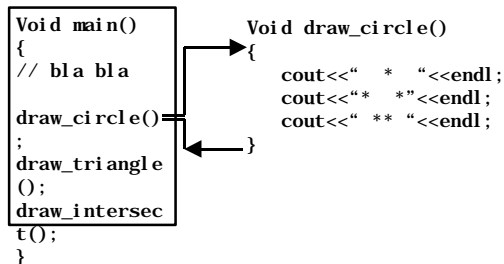
Simplest Functions

- Some functions do not communicate any data
- Following function simply draws a figure
- Illustrates some of the important ideas
- Functions break up the problem facing programmers

Introduction to C and C++, CA212 © Brian Stone 2001

4

Flow of Control



Introduction to C and C++, CA212 © Brian Stone 2001

5

Minimal Function Definition Syntax



Introduction to C and C++, CA212 © Brian Stone 2001

6

A Simple Program

```
// FILE: stikfig.cpp
#include <iostream.h>
void main ()
{
    // Draw the figure.
    draw_circle (); // Draw a circle.
    draw_triangle (); // Draw a triangle.
    draw_intersect (); // Draw intersecting lines.
}

// DRAWS A CIRCLE
void draw_circle ()
{
    cout << " * " << endl;
    cout << " * * " << endl;
    cout << " * * * " << endl;
} // end draw_circle

// DRAWS A TRIANGLE
void draw_triangle ()
{
    // Draw a triangle.
    draw_intersect ();
    draw_base ();
} // end draw_triangle
```

Introduction to C and C++, CA212 © Brian Stone 2001

7

```
// DRAWS INTERSECTING LINES
void draw_intersect ()
{
    cout << " / \ " << endl;
    cout << " / \ \ " << endl;
    cout << " / \ \ \ " << endl;
} // end draw_intersect

// DRAWS A HORIZONTAL LINE
void draw_base ()
{
    cout << " ----- " << endl;
} // end draw_base
```

Introduction to C and C++, CA212 © Brian Stone 2001

8

Common Math Functions

- Commonly used math library functions include
 - $\cos(x)$ x is in radians, returns a value which must be assigned as follows
 - `myvar = cos(rads);`
 - $\log(x)$, $\text{pow}(x, y)$, $\sin(x)$, $\text{sqrt}(x)$, $\text{fmod}(x, y)$
- These functions accept data through the parameter list
- See Deitel & Deitel 3rd Ed. Page 161

Introduction to C and C++, CA212 © Brian Stone 2001

9

Using Math 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

Introduction to C and C++, CA212 © Brian Stone 2001

10

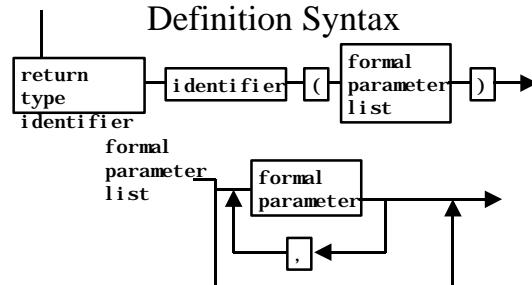
Arguments (also known as parameters)

- Function arguments (or parameters) can be
 - Constants
 - `sqrt(4);`
 - Variables
 - `sqrt(x);`
 - Expressions
 - `sqrt(sqrt(x));`
 - `sqrt(3 - 6x);`

Introduction to C and C++, CA212 © Brian Stone 2001

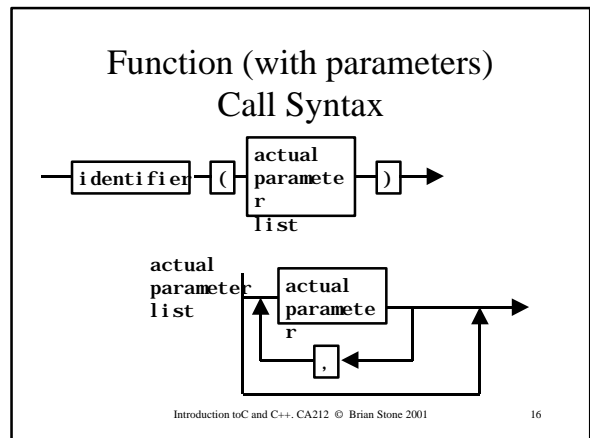
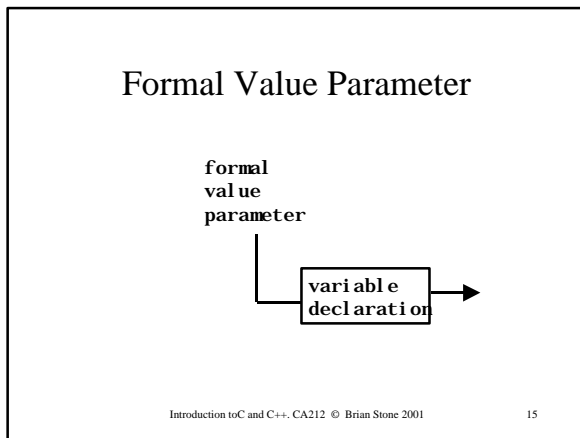
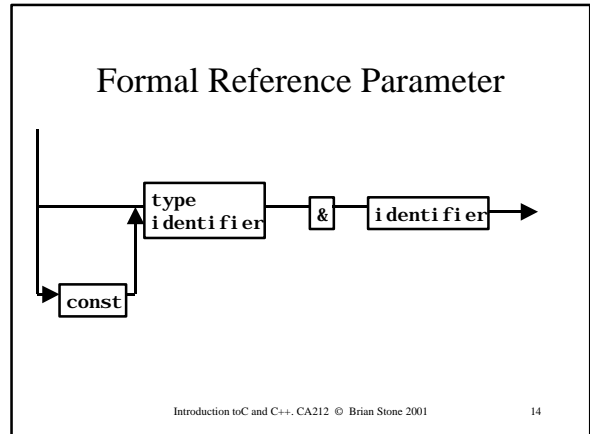
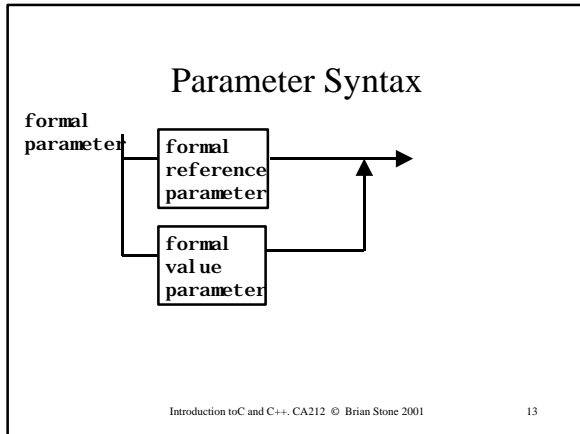
11

Function (with parameters or arguments) Definition Syntax



Introduction to C and C++, CA212 © Brian Stone 2001

12



Function Definitions

- **Prototype** tells compiler what to expect to see in the code and defers implementation until later

```

// Creating and using a programmer-defined function
#include <iostream.h>
int square(int); // function prototype
int main()
{
    for (int x = 1; x <= 10; x++)
        cout << square(x) << " "; //function call
    cout << endl;
    return 0;
}
// Function definition
int square(int y)
{
    return y * y;
}
    
```

Introduction to C and C++, CA212 © Brian Stone 2001 17

Terminology

- Make sure that you *understand* and *use* the following terminology
 - Prototype
 - user defined function
 - invoke
 - parameter
 - return, return type

Introduction to C and C++, CA212 © Brian Stone 2001 18

Function Definitions with Return

- Formal definition (syntax) below
- Also need prototypes
- Calling convention required

Formal Syntax

```
Return-type function-name(parameter-list)
{
    declarations
    statements
}
```

Introduction to C and C++, CA212 © Brian Stone 2001

19

Example Function with Return()

```
/* A programmer-defined square function */
#include <iostream.h>
int square(int); /* function prototype */
void main(){
    int x;
    for (x = 1; x <= 10; x++)
        cout << square(x); //function call
    cout << endl;
    return 0;
}
/* Function definition */
int square(int y)
{
    return y * y;
}
```

Introduction to C and C++, CA212 © Brian Stone 2001

20

Function Interactions Through Parameters

- Functions may accept data from where they are called.
- Functions may return data to where they were called.
- Functions may have local data not accessible to the rest of the program.

Introduction to C and C++, CA212 © Brian Stone 2001

21

Function with Parameters by Value

```
// Fig. 3.4: fig03_04.cpp
// Finding the maximum of three integers
#include <iostream.h>
int maximum( int, int, int ); // function prototype
int main()
{
    int a, b, c, mymax;

    cout << "Enter three integers: ";
    cin >> a >> b >> c; // a, b and c below are arguments to the maximum function
    cout << "Maximum is: " << maximum( a, b, c ) << endl;

    mymax = maximum(a, b, c); // my code to demonstrate a value returning
    return 0;
}
```

Introduction to C and C++, CA212 © Brian Stone 2001

22

```
/* 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;
}
```

Introduction to C and C++, CA212 © Brian Stone 2001

23

Parameters by Value

- Local copy of parameter made
- Actual parameter unaffected by assignments to local copy
- **Automatic Variable** created off the stack
- Question
 - what is the effect of replacing
 - mymax = maximum(a, b, c); //with
 - maximum(a, b, c); // ?

Introduction to C and C++, CA212 © Brian Stone 2001

24

Another Function with Parameters by Value

```
/* Cube a variable using call by value */
#include <iostream.h>
int cubeByValue(int );
main(){
    int number = 5;
    int return_num = 0;
    cout<<"The original value of number is \n"<< number;
    return_num = cubeByValue(number);
    cout<<"The new value of number is \n"<< return_num;
    cout<<"The old value of number is \n"<< number;
    return 0;
}
int cubeByValue(int n)
{ return n * n * n; /* cube local variable n */}
```

Introduction to C and C++, CA212 © Brian Stone 2001

25

For Practice Only

- Do questions (they are mostly very small)
 - (3.12), (3.13), (3.14), (3.18), (3.20), (3.48 graded)
- Harder ones include the following
 - (3.41), (3.42)
- Ask me for the code for all of these, will put it on the network later !

Introduction to C and C++, CA212 © Brian Stone 2001

26

Graded Exercises I

- Ex. 3.48 (use return) Deitel & Deitel 3rd Ed.
- Write two separate functions using value parameters and a *return* statement to calculate each of the following.....
 - area of a circle
 - circumference of a circle
 - a rectangle
 - a triangle
 - volume of a sphere ($4/3 * \text{Pi} * r^3$)

Introduction to C and C++, CA212 © Brian Stone 2001

27

Graded Exercises II

- Write a program which a builder will use to calculate the area of garden left on a site after building a house. The house is a rectangle, as is the site. You must subtract the area of the house from the area of the site. Make sure that the house fits into the site, check the lengths and sides of each. Use the function for area of a rectangle from the previous question.

Introduction to C and C++, CA212 © Brian Stone 2001

28

Parameters by Reference I

- If more than one value has to be returned from a function, then we cannot simply use the **return ()** statement.
- We can send in the memory location of the variables we want to change.
- We use '&' in the formal parameter list to tell the compiler that we want to change these values.

Introduction to C and C++, CA212 © Brian Stone 2001

29

```
/* A rewritten square function with reference parameter */
#include <iostream.h>
void square(int &); /* function prototype with reference parameter*/
void main(){
    int x;
    for (x = 1; x <= 10; x++){
        square(x); //function call
        cout<< x; }
    cout<<endl;
}
void square(int &y) // no return statement here, note the '&'
{
    y *= y;
}
```

Introduction to C and C++, CA212 © Brian Stone 2001

30

Notes on `void square(int &);`

- The return type is **void**, no return statement will appear in the body of the function
- The prototype includes a **'&'** in the parameter list, this denotes a *reference parameter*.
- The function call passes **'x'** as a parameter, this appears as **'y'** within the body of the function, **y** is **x**, not a local copy, we have passed the *memory location* of **'x'** for use as **'y'** !

Function with Parameters by Reference II

```
/* Cube a variable using call by reference */
#include <stdio.h>
void cubeByReference(int &); //note the void return type here and the '&'

void main(){
    int number = 5;
    cout<<"The original value of number is \n"<< number;
    cubeByReference(number); // no need to assign value from function
    cout<<"The new value of number is \n"<< number;
    return 0;
}
void cubeByReference(int &nRef) // used the '&' again
{nRef = nRef * nRef * nRef; /* cube number in main */} // no need for return()
```

Reference Parameters in Code

- We do not use the return statement in the function
- For the prototype we use
 - `void cubeByReference(int &);`
- To invoke the function we use
 - `cubeByReference(number);`
- It looks then same here as before, just examine the formal parameters to distinguish value or reference

Function with Parameters by Reference III

```
#include <iostream.h>
int squareByValue(int);
void squareByReference(int &);

main(){
    int x = 2, z = 4;
    cout << "x = " << x << " before squareByValue" << endl;
    << "Value returned by squareByValue: "
    << squareByValue(x) << endl;
    << "x = " << x << " after squareByValue" << endl << endl;
    cout << "z = " << z << " before squareByReference" << endl;
    squareByReference(z);
    cout << "z = " << z << " after squareByReference" << endl;
    return 0;
}

int squareByValue(int a){
    return a * a;
    // caller's argument not modified
}

void squareByReference(int &cRef)
{
    cRef *= cRef;
    // caller's argument modified
}
```

Storage

- Several issues exist
 - Scope
 - Persistence
- We will be revisiting this later with classes
- For now lets look at variables...

Storage Classes

- Storage class specifiers
 - Storage class
 - Where object exists in memory
 - Scope
 - Where object is referenced in program
 - Linkage
 - Where an identifier is known
- Automatic storage
 - Object created and destroyed within its block
 - **auto**
 - Default for local variables.
 - Example:

```
auto float x, y;
```
 - **register**
 - Tries to put variables into high-speed registers
 - Can only be used with local variables and parameters

3.10 Storage Classes

- Static storage
 - Variables exist for entire program execution
 - **static**
 - Local variables defined in functions
 - Keep value after function ends
 - Only known in their own function
 - **Extern**
 - Default for global variables and functions.
 - Known in any function

Identifier Scope Rules

- File scope
 - Defined outside a function, known in all functions
 - Examples include, global variables, function definitions and functions prototypes
- Function scope
 - Can only be referenced inside a function body
 - Only labels (**start:**, **case:**, etc.)
- Block scope
 - Declared inside a block. Begins at declaration, ends at }
 - Variables, function parameters (local variables of function)
 - Outer blocks "hidden" from inner blocks if same variable name
- Function prototype scope
 - Identifiers in parameter list
 - Names in function prototype optional, and can be used anywhere

```

1 // Fig. 3.12: fig03_12.cpp
2 // A scoping example
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 void af( void ); // function prototype
9 void bf( void ); // function prototype
10 void cf( void ); // function prototype
11
12 int x = 1; // global variable
13
14 int main()
15 {
16     int x = 5; // local variable to main
17
18     cout << "local x in outer scope of main is " << x << endl;
19
20     { // start new scope
21         int x = 7;
22
23         cout << "local x in inner scope of main is " << x <<
24             << endl; // end new scope
25
26     }
27     cout << "local x in outer scope of main is " << x << endl;
28
29     a(); // a has automatic local x
30     b(); // b has static local x
31     a(); // a reinitializes local x
32     b(); // static local x is 5
33     c(); // global x also re
34

```

1. Function prototypes

- 1.1 Initialize global variable
- 1.2 Initialize local variable
- 1.3 Initialize local variable in block

2. Call functions

3. Output results

local x is different inside and outside the block.

local x in outer scope of main is 5
local x in inner scope of main is 7
local x in outer scope of main is 5

```

35     cout << "local x in main is " << x << endl;
36
37     return 0;
38 }
39
40 void a( void )
41 {
42     int x = 25; // initialized each time a is called
43
44     cout << endl << "local x in a is " << x
45         << " after entering a" << endl;
46     ++x;
47     cout << "local x in a is " << x
48         << " before exiting a" << endl;
49 }
50
51 void b( void )
52 {
53     static int x = 50; // static initialization on
54                       // first time b is called.
55     cout << endl << "local static x is " << x
56         << " on entering b" << endl;
57     ++x;
58     cout << "local static x is " << x
59         << " on exiting b" << endl;
60 }
61
62 void c( void )
63 {
64     cout << endl << "global x is " << x
65         << " on entering c" << endl;
66     x *= 10;
67     cout << "global x is " << x << " on exiting c" << endl;
68 }

```

Local automatic variables are created and destroyed each time a is called.

local x in a is 25 after entering a
local x in a is 26 before exiting a

Local static variables are not destroyed when the function ends.

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

Global variables are always accessible. Function c references the global x.

global x is 1 on entering c
global x is 10 on exiting c

```

local x in outer scope of main is 5
local x in inner scope of main is 7
local x in outer scope of main is 5

local x in a is 25 after entering a
local x in a is 26 before exiting a

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

global x is 1 on entering c
global x is 10 on exiting c

local x in a is 25 after entering a
local x in a is 26 before exiting a

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

global x is 10 on entering c
global x is 100 on exiting c
local x in main is 5

```

Program Output

Introduction to C and C++, CA212 © Brian Stone 2001 41

Introducing Pointers

Direct access to memory is forbidden in Java, but is common and even necessary in C and C++

Introduction to C and C++, CA212 © Brian Stone 2001 42

Pointer Variable Declarations and Initialization

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

Introduction to C and C++, CA212 © Brian Stone 2001

43

Using Pointers

- Can declare pointers to any data type
- Pointers initialization
 - Initialized to `0`, `NULL`, or an address
 - `0` or `NULL` points to nothing

Introduction to C and C++, CA212 © Brian Stone 2001

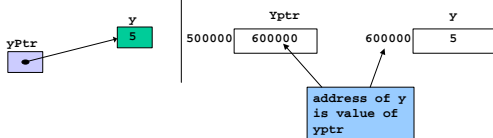
44

Pointer Operators

- & (address operator)
 - Returns the address of its operand
 - Example

```
int y = 5;
int *yPtr;
yPtr = &y; // yPtr gets address of y
```

- `yPtr` "points to" `y`



Introduction to C and C++, CA212 © Brian Stone 2001

45

Pointer Operators...

- * (indirection/dereferencing operator)
 - Returns the value of what its operand points to
 - `*yPtr` returns `y` (because `yPtr` points to `y`).
 - * can be used to assign a value to a location in memory
 - `*yPtr = 7; // changes y to 7`
 - Dereferenced pointer (operand of *) must be an lvalue (no constants)
- * and & are inverses
 - Cancel each other out
 - `*&myVar == myVar`
and
`&*yPtr == yPtr`

Introduction to C and C++, CA212 © Brian Stone 2001

46

```
1 // Fig. 5.4: E1005_04.cpp
2 // Using the & and * operators
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 int main()
9 {
10     int a; // a is an integer
11     int *aPtr; // aPtr is a pointer to an integer
12
13     a = 7;
14     aPtr = &a; // aPtr set to address of a
15
16     cout << "The address of a is " << &a
17         << "\nThe value of aPtr is " << aPtr;
18
19     cout << "\nThe value of a is " << a
20         << "\nThe value of *aPtr is " << *aPtr;
21
22     cout << "\nShowing that * and & are inverses of
23         << "each other.\n*&aPtr = " << *&aPtr;
24     return 0;
25 }
```

The address of `a` is the value of `aPtr`.

The * operator returns an alias to what its operand points to. `aPtr` points to `a`, so `*aPtr` returns `a`. This is called dereferencing. Notice how * and & are inverses.

1. Declare variables

```
The address of a is 006AFDF4
The value of aPtr is 006AFDF4
The value of a is 7
The value of *aPtr is 7
Showing that * and & are inverses of each other.
*&aPtr = 006AFDF4
*&aPtr = 006AFDF4
```

3. Print Program Output

2001 47

Calling Functions by Reference

- Call by reference with pointer arguments
 - Pass address of argument using & operator
 - Allows you to change actual location in memory
 - * operator used as alias/nickname for variable inside of function

```
void doubleNum( int *number )
{
    *number = 2 * ( *number );
}
```

- `*number` used as nickname for the variable passed in

- When the function is called, must be passed an address

```
doubleNum( &myNum );
```

Introduction to C and C++, CA212 © Brian Stone 2001

48


```

1 // Fig. 5.7: Fig05_07.cpp
2 // Cube a variable using call-by-reference
3 // with a pointer argument
4 #include <iostream>
5
6 using std::cout;
7 using std::endl;
8
9 void cubeByReference( int * ); // p
10
11 int main()
12 {
13     int number = 5;
14
15     cout << "The original value of number is " << number;
16     cubeByReference( &number );
17     cout << "\nthe new value of number is " << number << endl;
18     return 0;
19 }
20
21 void cubeByReference( int *nPtr ) // Define function
22 {
23     *nPtr = *nPtr * *nPtr * *nPtr; // cube
24 }

```

Notice how the address of number is given - **cubeByReference** expects a pointer (an address of a variable).

Inside **cubeByReference**, ***nPtr** is used (***nPtr** is number). This is **dereferencing**.

1. Function prototype - takes a pointer to an int.
 1.1 Initialize variables
 2. Call function

Define function

Program Output

```

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

```

Introduction to C and C++, CA212 © Brian Stone 2001 49

We will Revisit Pointers

- Arrays make extensive use of pointers
- Pointers to user defined data structures
 - Arrays and Strings
 - Implementing stacks and queues, linked lists and trees
 - Classes and Structs
 - Pointers to Functions even!
- Pointers are used in File I/O