Chapter 3 - Functions

- Learning outcomes:

  - *Describe the principles of structured programming*
  - *Apply function prototyping, definition, & calling*
  - *Discuss the scopes & storage classes of variables*
  - *Explain function calls stack, & activation record*

  - *Apply passing arguments to function*
  - *Analyze recursive, inline, & overloaded functions*
  - *Describe & apply default arguments in functions*

Introduction

- "Structured programming" (1960s)
  - use "divide-and-conquer" technique
    - one big problem (main) divided into a set of smaller tasks,
      - each tasks are implemented using a "function"
  - so constructs a program from a set of smaller pieces or modules/components, instead of
    - one long main program (or "spaghetti code")
  - a "top-to-down" (or "boss-to-worker") relationship b/n modules
    - a boss (i.e., the calling function or caller) asks a worker (i.e., the called function or responder) to perform a task,
    - and return the results when the task is done (i.e., report back)

Cont'd…

- Function
  - a group of statements that perform a particular task or group of tasks.
- Modularization
  - structure a program into smaller & more manageable pieces or modules
    - facilitate the design, implementation, operation & maintenance of large programs
- Reusability
  - "write only once, use many time",
    - can be called multiple times in a program, or included any number of times in different programs

C++ Functions

- Standard functions,
  - "built-in functions", came with the C++ language
  - prepackaged under the C++ standard library,
    - above 51 standard libraries `// #included`
  - e.g., math functions declared in:
    - `#include <cmath>` `// #include <math.h>`
- User-defined functions
  - newly developed from scratch by users/programmers
    - e.g., `martixAdd()` *// a function to add matrixes*
  - can be also organized into a library
    - e.g., `"matrix.h"` `// #include "matrix.h"`

The Standard Math Library Functions

- Perform common mathematical calculations,
  - included in **<cmath>** header file *// or <math.h>*
  - most functions take **double**, & return **double**
- Functions called,
  - syntax: **funcName(argument1, argument2, …, N);**
- Example
  - **sqrt(double x);** *// return the square root of x*

```
cout<<sqrt(900.0); // output 30
```

Method	Description	Example
`ceil(x)`	rounds *x* to the smallest integer not less than *x*	`ceil(9.2)` is `10.0` `ceil( -9.8 )` is `-9.0`
`cos(x)`	trigonometric cosine of *x* (*x* in radians)	`cos( 0.0 )` is `1.0`
`exp(x)`	exponential function *ex*	`exp( 1.0 )` is `2.71828` `exp( 2.0 )` is `7.38906`

`fabs(x)`	absolute value of *x*	`fabs( 5.1 )` is `5.1`
		`fabs( 0.0 )` is `0.0`
		`fabs( -8.76 )` is `8.76`

`floor(x)`	rounds *x* to the largest integer not greater than *x*	`floor( 9.2 )` is `9.0` `floor( -9.8 )` is `-10.0`
`fmod(x, y)`	remainder of *x/y* as a floating-point number	`fmod( 13.657, 2.333 )` is `1.992`
`log(x)`	natural logarithm of *x* (base *e*)	`log( 2.718282 )` is `1.0` `log( 7.389056 )` is `2.0`
`log10(x)`	logarithm of *x* (base 10)	`log10( 10.0 )` is `1.0` `log10( 100.0 )` is `2.0`
`pow(x, y)`	*x* raised to power *y* (*xy*)	`pow( 2, 7 )` is `128` `pow( 9, .5 )` is `3`
`sin(x)`	trigonometric sine of *x* (*x* in radians)	`sin( 0.0 )` is `0`
`sqrt(x)`	square root of *x*	`sqrt( 900.0 )` is `30.0` `sqrt( 9.0 )` is `3.0`
`tan(x)`	trigonometric tangent of *x* (*x* in radians)	`tan( 0.0 )` is `0`

Example 1

```cpp
// Example 1- Using standard functions
#include <iostream>

// previous equivalent: #include <math.h>
#include <cmath> // to use pow ()

using namespace std;

int main(){

    double a, b, c;

    cout<<"Enter the value of a and b?\n";
    cin>>a>>b;

    c = pow(a, b);  // function calling

    cout<<"\n"<<a<<" raised to the power of "<<b<<": "<<c;

    cout<<endl;

    return 0;
}
```

Cont'd…

- User-defined functions
  - standard functions may not be enough, to satisfy all users need
    - e.g., *"find the largest of two integers?"* ***// no built-in function***
  - so C++ language provides users to create their own functions
- Function structure,
  - function prototyping **(A)** ***// or "declaration"***
  - function definition **(B)**
- Functions are invoked by,
  - function calling **(C)**
    - it is where execution of the function begins

Function Prototyping

- Function prototype
  - tell the compiler about the existence of the function
    - i.e., its arguments type & return type
- Syntax
  - **`return-type funcName(arg1-type, arg2-type,...);`**
    - use **`void`** as the return type, when returning nothing
- Example
  - **`int square(int);`** *`// takes an int, & returns int`*

    `int square(int a);`
    - optional to specify parameters' name

Function Definition

- Syntax
  - **`return-type funcName(parameter-list)`**

```
{
    … // declarations
    … // statements
}
```

- Parameter list -> **`(type par1, type par2, …)`**
  - comma separated list of parameters
    - data type is needed for each parameters
  - use **`void`** (or leave blank), when no arguments is received
    - e.g., **`int main () {return 0;}`**

Cont'd…

- Return type
  - the data type of value returned by the function
  - use **`void`**, when nothing is returned
    - e.g., **`void main () { }`** ***`// return 0; not needed`***
- Example

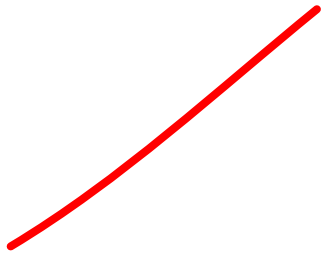
```
int square(int y)
{
    return (y * y);
}
```

- Keyword: **`return`**
  - return data, & the control goes to the function caller

Cont'd…

- Function prototype,
  - must match the function definition
- Example
  - `int max(int, int, int);` *// function prototype*
  - `int max(int x, int y, int z)` *// function definition*

```
{
    … // statements
}
```



- Warning: a function cannot be defined inside another function

Function Calling

- Function calling (or invoking function)
  - syntax: **`funcName(arg1,arg2, …);`**
- Example
  - **`square(x);`** ***// calls function named `square`***
    - pass argument **`x`** to **`square`**
    - function get its own copy of the arguments ***// "pass by value"***
    - after the task finished, return the result
  - **`a = square(x);`** ***// assign `square` to `a`***
    - assign the result returned by **`square`** to **`a`**

Example 2

```
1  // Example 2 - Function to add two numbers, & return the sum
2  #include <iostream>
3
4  using namespace std;
5
6  int addition (int , int ); // function prototype
7
8  int main(){
9
10     int a, b, c;
11
12     cout<<"Enter the value of a and b?\n";
13     cin>>a>>b;
14
15     c = addition(a, b); // function calling
16
17     cout<<"\na+b = "<<c<<endl;
18
19     return 0;
20 }
21
22 // function definition
23 int addition (int x, int y){
24
25     int z;
26
27     z = x + y;
28
29     return z;
30 }
```

Example 3

```
1   // Example 3 - Function to find the square
2   // of integers between 1 and 10
3   #include <iostream>
4
5   using namespace std;
6
7   int square(int); // function prototype
8
9   int main(){
10
11      cout<<"\nThe square of integers between 1-10: ";
12
13      for(int x=1; x <=10; x++)
14          cout<<square(x)<<" "; // function call
15
16      cout<<endl;
17
18      return 0;
19  }
20
21  // function definition
22  int square(int y){
23
24      return y*y;
25  }
```

Cont'd…

- Functions with an empty parameter list,
  - use **`void`** (or leave parameter list empty)
- Example
  - **`void print(void);`** ***// function prototype***
    - optional to specify **`void`**
  - **`void print( );`**
    - **`print`** take no arguments,
    - Also returns no value (or nothing)

Example 4

```
1  // Example 4 - Function with void return type
2  // and empty parameter list
3  #include <iostream>
4
5  using namespace std;
6
7  void printmessage(); // function prototype
8
9  int main(){
10
11     printmessage(); // function calling
12
13     return 0;
14 }
15
16 // function definition
17 void printmessage(){
18
19     cout<<"\nI'm void function!\n";
20
21 }
```

Example 5

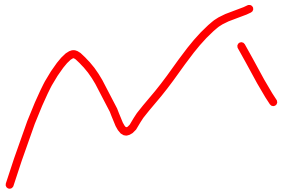
```
1  // Example 5 - Function with parameter list,
2  // but not with return type
3  #include <iostream>
4
5  using namespace std;
6
7  void addition (int , int ); // function prototype
8  int main(){
9
10     int a, b, c;
11
12     cout<<"Enter the value of a and b?"<<endl;
13     cin>>a>>b;
14
15     addition(a, b); // function calling
16
17     return 0;
18 }
19
20 // function definition
21 void addition(int x, int y){
22
23     int z;
24
25     z = x+y;
26
27     cout<<"\na+b = "<<z<<endl;
28 }
```

Scope of Variables

- Scope of variables
  - define where a variable can be accessed in a program
- Local variable
  - declared in the body **{ }** or inside function **()**
  - only accessed by the function declared it
  - e.g., variables declared inside the **`main ()`**
- Global variable
  - declared outside a function, before the **`main ()`**
  - accessed by all the functions, including the **`main ()`**

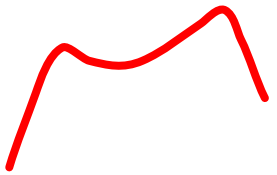
Example 6

```
1  // Example 6 - Using local variables
2  #include <iostream>
3
4  using namespace std;
5
6  void update( ); // function prototype
7
8  int main(){
9
10     int x=5; // declare x, local variable to main()
11
12     cout<<"The value of x local in main (): "<<x<<endl;
13
14     update( ); // function calling
15
16     cout<<"\nThe value of x local in main (): "<<x<<endl;
17
18     update( ); // function calling
19
20     return 0;
21 }
22
23 // update () reinitializes local variable x during each call
24 void update( ){
25
26     int x=25; // declare x, local variable to update()
27
28     cout<<"The value of x on entering update (): "<<x<<endl;
29
30     x++; // increment x by 1
31
32     cout<<"The value of x on exiting update (): "<<x<<endl;
33 }
```



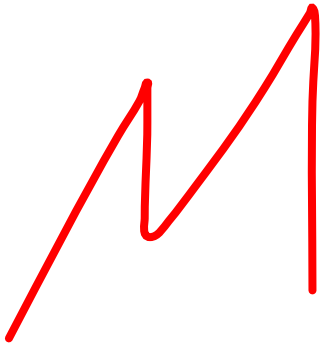
Example 7

```cpp
// Example 7 - Using global variables
#include <iostream>

using namespace std;

void update( ); // function prototype

int x=5; // declare global variable x

int main(){

    // x is not declared in main (),
    // so can be accessed anywhere
    cout<<"The value of x in main(): "<<x<<endl;

    update( ); // function calling

    cout<<"\nThe value of x in main(): "<<x<<endl;

    return 0;
}

// update() modifies global variable
// x during each call
void update(  ){

    // x is not declared in update (),
    // so can be accessed anywhere
    cout<<"\nThe value of x on entering update(): "<<x<<endl;

    x = x*10; // update x

    cout<<"The value of x on exiting update(): "<<x<<endl;
}
```



Example 8

```
1  // Example 8 - Using the same name
2  // for local variable & global variable
3  #include <iostream>
4
5  using namespace std;
6
7  void update( ); // function prototype
8
9  int x=5; // declare x as global variable
10
11 int main(){
12
13     update( ); // function calling
14
15     return 0;
16 }
17
18 // function definition
19 void update( ) {
20
21     int x=25; // declare x local variable to update()
22
23     cout<<"The value of x: "<<x<<endl;
24 }
```

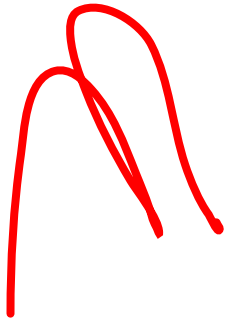


Scope Resolution Operator

- If a global variable has the same name as the local variable,
  - local variable is accessed (by default)
- Scope resolution operator **(::)**
  - used to access a global variable inside a function,
    - when the function has a local variable with the same name
  - syntax: **::varName;**
- Example
  - **cout<<::x;**
  - **y = ::x + 3;**

Example 9

```
1   // Example 9 - Using scope resolution operator
2   #include <iostream>
3
4   using namespace std;
5
6   void update( ); // function prototype
7
8   int x=1; // declare x as global variable
9
10  int main(){
11
12      int x=5; // declare x local variable to main()
13
14      cout<<"The value of local x: "<<x<<endl;
15
16      // access global x in main()
17      cout<<"The value of global x: "<<::x<<endl;
18
19      update( ); // function calling
20
21      return 0;
22  }
23
24  // function definition
25  void update( ){
26
27      int x=25; // declare x local variable to update()
28
29      cout<<"\nThe value of local x: "<<x<<endl;
30
31      // access global x in update()
32      cout<<"The value of global x: "<<::x<<endl;
33  }
```



Storage Classes

- Storage class of variables
  - determine the period a variable exist or kept inside a memory
- Automatic variable
  - created when a program/function enter its block
  - destroyed when a program/function leave its block
- Example
  - all local variables of a function (by default)
    - `int main () {int x; return 0;}` *// implicitly*
  - or declared as `auto int x;` *// explicitly*

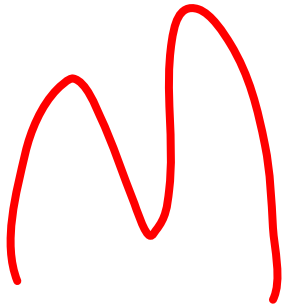
Cont'd…

- Static variable
  - created when program execution begin
  - initialized only once, when it's declared
  - existed for the duration of program execution
- Example
  - all global variables (by default)
    - `int a = 5;` *// same as static `int a = 5` (explicitly)*

      `int main (){return 0;}`
  - local variables declared, `static int x=20;` *// explicitly*
    - `x` retain its value b/n function calls
    - only referenced locally *// function declared*

Example 10

```
1  // Example 10 - auto & static local variables
2  #include <iostream>
3
4  using namespace std;
5
6  void update( ); // function prototype
7
8  int main(){
9
10     int x=5; // declare x local variable to main ()
11
12     cout<<"The value of x local in main(): "<<x<<endl;
13
14     update( ); // function calling
15
16     cout<<"\nThe value of x local in main (): "<<x<<endl;
17
18     update( ); // function calling
19
20     return 0;
21 }
22
23 // update( ) initializes static local variable x
24 // only the first time the function is called,
25 // and the value of x is saved b/n calls to update( )
26 void update( ){
27
28     static int x=50; // declare x static-local variable to update( )
29
30     cout<<"The value of x on entering update(): "<<x<<endl;
31
32     x++; // increment x by 1
33
34     cout<<"The value of x on exiting update(): "<<x<<endl;
35 }
```



Function Calls Stack

- Function call stack
  - also known as "program execution stack"
  - support function call & return mechanism
  - use LIFO (Last-In First-Out) to manage function calls
- Each time, a function (A) calls another function (B),
  - a stack frame (or an "activation record") is pushed to the stack:
    - Maintain the return address,
      - i.e., function called (B) need to return to function calling (A)
    - contain automatic variables,
      - i.e., function called (B) local variables & parameters

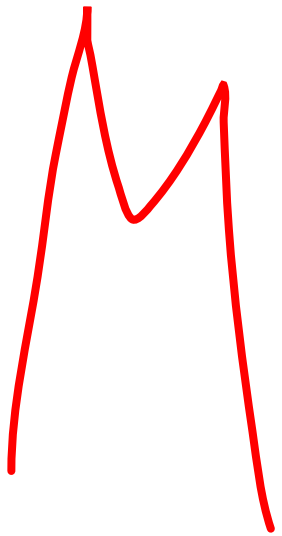
Cont'd…

- When function called (B) return, to the function calling (A),
  - stack frame for the function called (B) is popped from the stack, and
  - the control transfer to the return address, in the popped stack frame
- Function call stack
  - has a maximum size assigned
- "Stack overflow",
  - an error that occur when more function calls occurred than the call stack can store their activation records (due to memory limitations)
  - may result a program crash

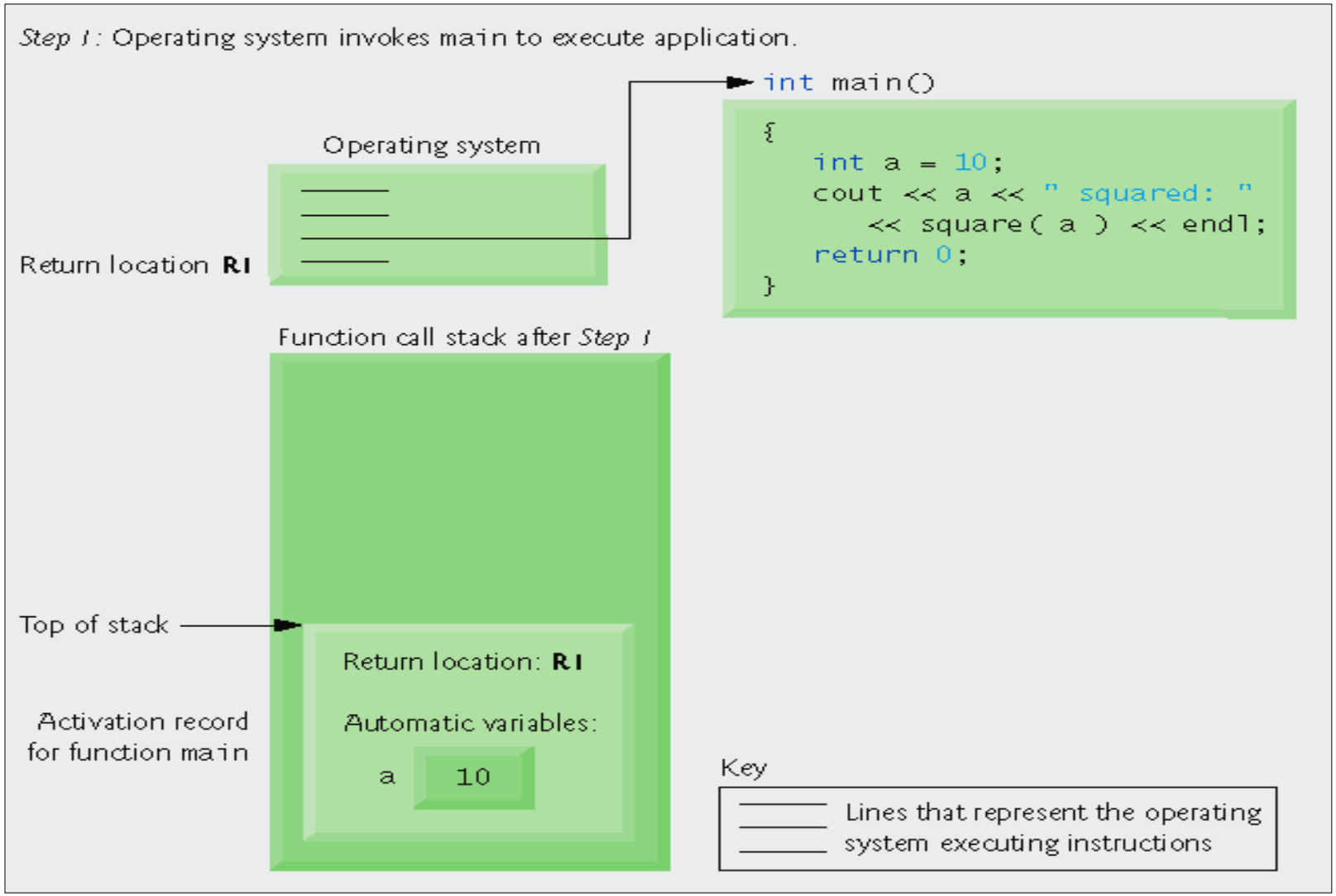
https://stackoverflow.com/

Example 11

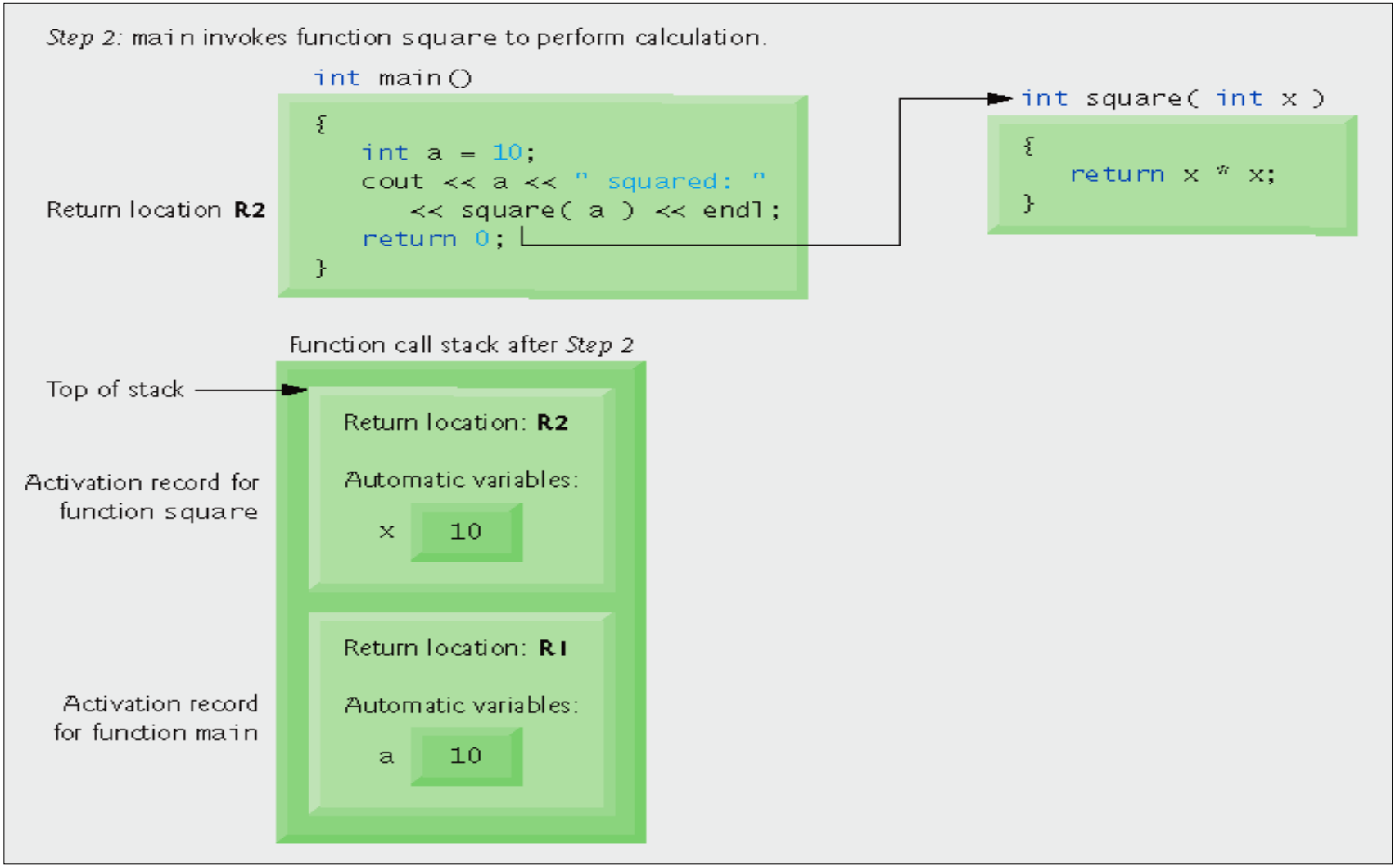
```
1  // Example 11 - Function call stack
2  // and activation records
3  #include <iostream>
4
5  using namespace std;
6
7  int square(int); // function prototype
8
9  int main(){
10
11     // local automatic variable
12     int a=10;
13
14     // calling square function
15     cout<<a<<" squared: "<<square(a);
16
17     cout<<endl;
18
19     return 0;
20 }
21
22 // function definition
23 // x is a local variable
24 int square(int x){
25
26     // calculate and return result
27     return (x*x);
28 }
```



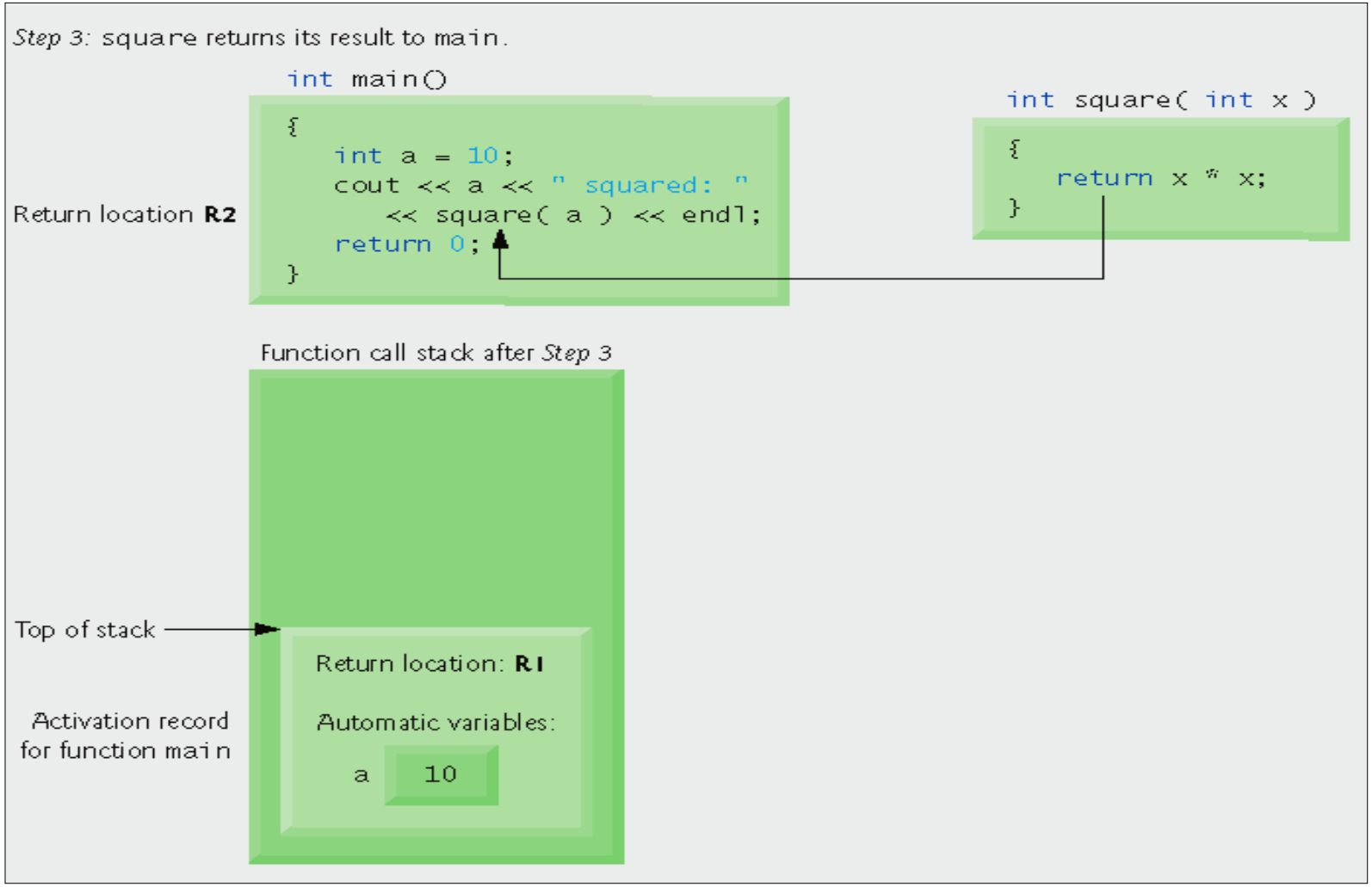
Cont'd…


Step 1: Operating system invokes main to execute application.
int main()
{
int a = 10;
cout << a << " squared: "
<< square(a) << endl;
return 0;
}
Operating system
Return location R1
Function call stack after Step 1
Top of stack
Return location: R1
Automatic variables:
a 10
Activation record for function main
Key
Lines that represent the operating system executing instructions

Cont'd…


Step 2: main invokes function square to perform calculation.
int main()
{
int a = 10;
cout << a << " squared: "
<< square(a) << endl;
return 0;
}
Return location R2
int square(int x)
{
return x * x;
}
Function call stack after Step 2
Top of stack
Return location: R2
Automatic variables:
x 10
Activation record for function square
Return location: R1
Automatic variables:
a 10
Activation record for function main

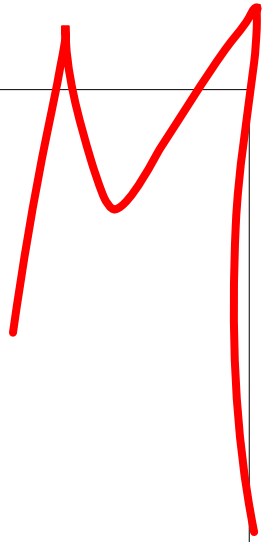
Cont'd…


Step 3: square returns its result to main.
int main()
{
int a = 10;
cout << a << " squared: "
<< square(a) << endl;
return 0;
}
int square(int x)
{
return x * x;
}
Return location R2
Function call stack after Step 3
Top of stack
Return location: R1
Automatic variables:
a 10
Activation record for function main

Passing Arguments to Function

- Pass by value
  - copy of argument value, passed to parameter
  - changes made to function parameters,
    - do not affect values of the arguments
  - thus, prevented from unwanted side effects
- Pass by reference
  - copy of argument address, passed to parameter
  - changes made to function parameters,
    - affect or modify values of the arguments

Example 12



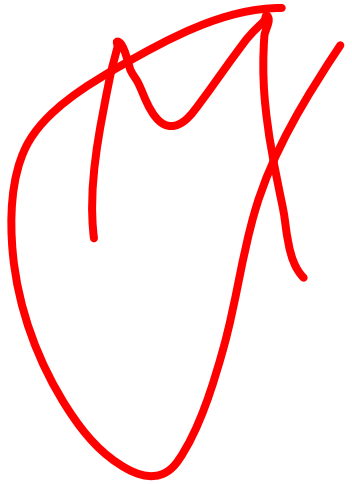
```
1  // Example 12 - Passing arguments by value
2  #include <iostream>
3
4  using namespace std;
5
6  int squareByValue(int); // function prototype
7
8  int main(){
9
10     int x=2;
11
12     cout<<"\nThe value of x before squareByValue(): "<<x<<endl;
13
14     // pass x by value to squareByValue
15     cout<<"\nThe value returned by squareByValue(): "<<squareByValue(x)<<endl;
16
17     cout<<"\nThe value of x after squareByValue(): "<<x<<endl;
18
19     return 0;
20 }
21 // function definition
22 // num receive a copy of x
23 int squareByValue(int num){ //
24
25     num = num*num;
26
27     return num;
28 }
```

Reference Variable

- Reference variable (**&**)
  - serves an "alias" of another (regular) variable
  - holds the "address" of referrer variable *// like pointer variables*
    - but can be dereferenced implicitly, and
    - also must be initialized when declared *// like constant pointers*
- Syntax: `type &referenceName = referrerName;`
- Example
  - `int x=5;`
    `int &y=x;` *// create y an alias for x*
  - `cout<<y;` *// same as cout<<x;*
  - `int a=3;`
    `int &b;` *// error*
  - `b=x;`

Example 13

```
1   // Example - 13 Reference variable
2   #include <iostream>
3
4   using namespace std;
5
6   int main(){
7
8       int x=3;
9       int &y=x;  // y refers to x
10
11      cout<<"x = "<<x<<endl;
12
13      cout<<"y = "<<y<<endl;
14
15      y=7;
16
17      cout<<"\nx = "<<x<<endl;
18
19      cout<<"y = "<<y<<endl;
20
21      return 0;
22  }
```

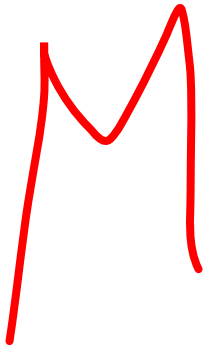


Pass by Reference

- Reference variable
  - used as function parameter
    - e.g., `void myReferenceVar(int &y);` *// fun. prototype*
  - variable name used, in function call
    - e.g., `myReferenceVar(x);`
- Pointer variable
  - used as function parameter
    - e.g., `void myPointerVar(int *b);` *// fun. prototype*
  - `&` used with an argument, in function calling
    - e.g., `myPointerVar(&a);` *// arrays do not need &*

Example 14

```
1  // Example 14 - Pass by reference variable
2  #include <iostream>
3
4  using namespace std;
5
6  void squareByReferenceVar(int &); // function prototype
7
8  int main(){
9
10     int z=4;
11
12     cout<<"\nz before squareByReferenceVar(): "<<z<<endl;
13
14     squareByReferenceVar(z);
15
16     cout<<"\nz after squareByReferencevar(): "<<z<<endl;
17
18     return 0;
19 }
20
21 // function definition
22 // reference variable as function parameter
23 void squareByReferenceVar(int &numberRef){
24
25     numberRef = numberRef*numberRef;
26 }
```



Example 15

```
1  // Example 15 - Pass by pointer variable
2  #include <iostream>
3
4  using namespace std;
5
6  void cubeByPointerVar(int *); // function prototype
7
8  int main(){
9
10     int num=5;
11
12     cout<<"\nThe original value of num: "<<num<<endl;
13
14     // pass the address of num to cubeByPointerVar
15     cubeByPointerVar(&num);
16
17     cout<<"\nThe new value of num: "<<num<<endl;
18
19     return 0;
20 }
21
22 // function definition
23 // pointer variable as function parameter
24 void cubeByPointerVar(int *ptr){
25
26     *ptr = *ptr * *ptr * *ptr;
27 }
```

Passing Array to Function

- Arrays passed by reference
  - array variable used, in function parameter
    - e.g., `void myArray(int y[], int n);` *// fun. prototype*
  - don't need array size *// ignored by compiler*
- Array name used, in function call
  - e.g., `int x[5];`

    `myArray(x, 5);` *// function call*
  - `x` contain the address of 1st element
  - usually, size of array also passed *// but not a must*
    - useful to iterate through elements

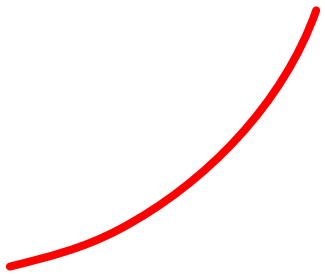
Cont'd…

- Array elements passed by value,
  - e.g., `int y[5]={10,20,30,40,50};`

    `square(y[3]);` *// function call; value of x[3] passed*
  - `int square(int v);` *// function prototype*
- To prevent an array, from being modified,
  - use `const` array, in function parameter
    - e.g., `void doNotModify(const int []);` *// fun. Proto.*
- For 2D arrays, in function parameter
  - e.g., `void printArray(int[][3]);` *// function prototype*
    - must specify sizes of subscripts *// except the first*

Example 16

```
1  // Example 16 - Passing array to function
2  #include <iostream>
3
4  using namespace std;
5
6  void printArray(int [], int); // function prototype
7
8  int main(){
9
10     int array1[]={5, 10, 15};
11
12     int array2[]={2, 4, 6, 8, 10};
13
14     cout<<"\nThe value of array1[]: ";
15
16     printArray(array1, 3); // function calling
17
18     cout<<"\n\nThe value of array2[]: ";
19
20     printArray(array2, 5);
21
22     cout<<endl;
23
24     return 0;
25 }
26
27 // function definition
28 // array variable as function parameter
29 void printArray(int arrayName[], int arraySize){
30
31     for(int i=0; i<arraySize; i++)
32         cout<<arrayName[i]<<" ";
33 }
```



Example 17

```
1  // Example 17 - Effects of passing
2  // an entire array by reference
3  #include <iostream>
4
5  using namespace std;
6
7  // function prototype
8  void modifyArray(int [], int);
9
10 int main(){
11
12     const int arraySize = 5;
13
14     int num[arraySize]={0, 1, 2, 3, 4};
15
16     cout<<"\n***** Effects of passing an entire array by reference *****"<<endl;
17
18     // output original array
19     cout<<"\nThe values of num[] before modifyArray(): ";
20
21     for(int i=0; i<arraySize; i++)
22         cout<<num[i]<<" ";
23
24     // function call
25     // pass array by reference
26     modifyArray(num, arraySize);
27
```

Cont'd…

```
28      // output modified array
29      cout<<"\nThe values of num[] after modifyArray(): ";
30
31      for(int j=0; j< arraySize; j++)
32          cout<<num[j]<<" ";
33
34      cout<<endl;
35
36      return 0;
37  }
38
39  // function definition
40  // b[] points to num[]
41  void modifyArray(int b[], int sizeOfArray){
42
43      for(int k=0; k<sizeOfArray; k++)
44          b[k] = b[k]*2;
45  }
```

Example 18

```
1  // Example 18 - Effects of passing
2  // individual array element by value
3  #include <iostream>
4
5  using namespace std;
6
7  // function prototype
8  void modifyElement(int);
9
10 int main(){
11
12     const int arraySize = 5;
13
14     int num[arraySize]={0, 1, 2, 3, 4};
15
16     cout<<"\n***** Effects of passing individual array element by value *****"<<endl;
17
18     cout<<"\nThe value of num[3] before modifyElement(): "<<num[3]<<endl;
19
20     // pass array element by value
21     modifyElement(num[3]);
22
23     // output the value of num[3]
24     cout<<"\nThe value of num[3] after modifyElement(): "<<num[3]<<endl;
25
26     return 0;
27 }
28
```

Cont'd…

```
29  // function definition
30  // variable e has a copy of the value of num[3]
31  void modifyElement(int e){
32
33      e = e*2;
34
35      cout <<"\nThe value e in modifyElement(): "<<e<<endl;
36  }
```

Recursive Function

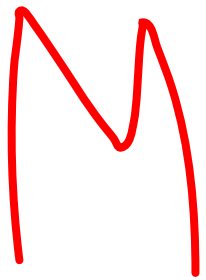
- Recursive function
  - function that call themselves
  - can only solve a base case
- If not base case,
  - break the problem into smaller problem(s)
  - call new copy of function, to work on the smaller problem
    - slowly converges toward the base case
    - function call itself, inside return statement
  - eventually the base case get solved,
    - answer works way back to up, & solve the entire problem

Cont'd…

- Example, to find the factorial of an integer **n**,
  - **n! = n * (n-1) * (n-2) * (…) * 1**
  - the recursive relationship: **n! = n * (n-1)!**
  - **5! = 5 * 4!**
    - **4! = 4 * 3!**
      - **3! = 3 * 2!**
        - ***2! = 2 * 1!***
          - **1! = 1** * **0!** (note: **0! = 1**) **= 1**
  - the base case: **1! = 1**

Example 22

```
1  // Example 22 - Using recursive function
2  // to find the factorial of an integer
3  #include <iostream>
4
5  using namespace std;
6
7  long factorial(long); // function prototype
8
9  int main(){
10
11     long number;
12
13     cout<<"\nEnter an integer number: ";
14     cin>>number;
15
16     // function call
17     cout<<number<<"! = "<<factorial(number)<<endl;
18
19     return 0;
20 }
21
22 // function definition
23 long factorial(long a){
24
25     if (a>1){ // check base case
26
27         return (a * factorial(a-1)); // recursive step
28
29     else{
30
31         return (1); // base case
32     }
33 }
```



Inline Function

- Function call overheads (for a compiler):
  - remember where to return value,
    - when function execution ends
  - provide memory,
    - for function variables
    - for value returned by function
  - pass control from function calling,
    - to the function called
  - pass control back to the calling function
- Solution: inline function

Cont'd…

- Inline function definition,
  - use **`inline`**, before return type
  - appear prior to the **`main()`**
  - copy code into program,
    - instead of making function call
  - substitute arguments
- Reduce function call overhead
- Good for small, & often-used functions

Cont'd…

- Inline function must precede,
  - the function that calls it,
- Eliminates the need for function prototyping
- Example,

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

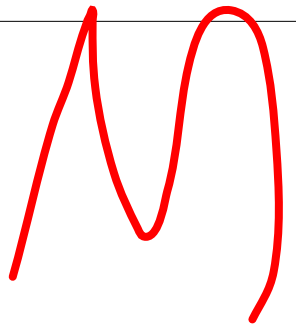
Example 23

```
1  // Example 23 - Using inline function
2  // to calculate the volume of a cube
3  #include <iostream>
4
5  using namespace std;
6
7  // function definition
8  // proceeds function call it
9  inline double cube(double side){
10
11     return (side*side*side);
12 }
13
14 int main(){
15
16     double sideValue;
17
18     cout<<"\nEnter the side length of your cube: ";
19     cin>>sideValue;
20
21     // function call
22     cout<<"\nVolume of cube: "<<cube(sideValue)<<endl;
23
24     return 0;
25 }
```

Default Arguments

- When function called, with an 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
- Example
  - `int myFunction(int x=1, int y=2, int z=3);`
  - `myFunction(3);`
    - **x=3**, **y** & **z** get defaults (rightmost)

Example 24



```
1   // Example 24 - Using default values
2   #include <iostream>
3
4   using namespace std;
5
6   // function parameters
7   // with default values
8   float divide(int a, int b=2);
9
10  int main(){
11
12      cout<<"\nThe result of divide(12): "<<divide(12)<<endl;
13
14      cout<<"\nThe result of divide(20,4): "<<divide(20, 4)<<endl;
15
16      return 0;
17  }
18
19  // function definition
20  float divide(int a, int b){
21
22      return a/b;
23  }
```

Overloaded Function

- Function overloading
  - functions that have the same name,
    - but d/f parameter sets (number, type, & order)
- Example
  -
```
int square(int x)
{
   return (x * x);
}
```
  -
```
float square(float x)
{
   return (x * x);
}
```
  - We can say, "**`square()`** is an overloaded function"

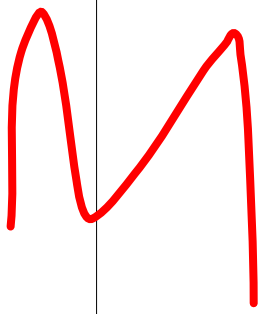
Cont'd…

- The compiler select the proper function,
  - based on the number, type, & order of arguments
  - passed to the function, in the function call.
- Commonly used: to create several functions, of the same name,
  - that perform similar tasks, but work on d/f data types
- Warning:
  - functions can not be overloaded, by return type
  - e.g., `int square(int x){return (x * x);}`

```
float square(int y){return (x * x);} // error
```

Example 25

```
1  // Example 25 - Using overloaded functions
2  #include <iostream>
3
4  using namespace std;
5
6  // function prototype
7  // overloaded functions
8  int square(int x);
9  double square(double y);
10
11 int main(){
12
13     int x;
14     double y;
15
16     x = square(7); // calls int version of square
17
18     y = square(7.5); // calls double version of square
19
20     cout<<"\nThe square of integer 7: "<<x<<endl;
21
22     cout<<"The square of double 7.5: "<<y<<endl;
23
24     return 0;
25 }
```



Cont'd…

```
27  // function definitions
28
29  // square() for int values
30  int square(int x){
31
32      cout<<"\nCalled square with int argument: "<<x;
33
34      return (x*x);
35  }
36
37  // square() for double values
38  double square(double y){
39
40      cout <<"\nCalled square with double argument: "<<y<<endl;
41
42      return (y*y);
43  }
```