EECS402 Lecture 02

Andrew M. Morgan

Savitch Ch. 3-4
Functions
Value and Reference Parameters

Functions

- Allows for modular programming
  - Write the function once, call it many times
  - Group similar things together
- Parameters allow values from calling function to be used within the function
- May, or may not, return a value to calling function
- General function template:

```
return_type function_name(formal parameter list)
{
  function code...
}
```


Function Prototype

- A function prototype "declares a function"
- Provides user with information about the function.
  - Name of function, what values (and types) need to be passed in, what type to expect the function to return
- Here is a function prototype for a factorial function

```
int computeFactorial(int num);
```

- The function is called "computeFactorial"
  - Takes in one integer value from the calling function as a parameter
  - Returns an integer value to the calling function
- Note: The function name should be ***descriptive*** of its purpose!

Function Definition

- A function definition provides the implementation (in C++) of an algorithm
- Here is a function definition for computing factorial of a number passed in by the user:

```
int computeFactorial(int num) //Function header
{
  int result = 1; //Value to return..
  int i;          //Loop variable

  for (i = 1; i <= num; i++)
  {
    result *= i;
  }
  return (result); //Returns an integer, as expected
}
```

- Function header matches function prototype (no ; though)

Function Call

- A function is "called" when you want to use the algorithm that was implemented in the function
- Here is how the main function would call computeFactorial:

```
int main(void)
{
  int fact;    //Factorial result
  int val = 5;

  fact = computeFactorial(val);  //function call!

  cout << "Fact. of " << val << " is: " << fact << endl;

  return (0);
}
```

Fact. of 5 is: 120

Complete Function Program

- Here is the complete program layout

```
#include <iostream> //Need for cout.
using namespace std;

//Function: computeFacorial - Computes
//factorial of num and returns it.
int computeFactorial(int num); //proto

int main(void)
{
  int fact;    //Factorial result
  int val = 5;

  fact = computeFactorial(val); //call

  cout << "Fact. of " << val
       << " is: " << fact << endl;

  return (0);
}
```

```
int computeFactorial(int num) //header
{
  int result = 1; //Value to return..
  int i;          //Loop variable

  for (i = 1; i <= num; i++)
  {
    result *= i;
  }
  return (result); //Return factorial
}
```

Fact. of 5 is: 120

- Note: The order shown is important

Multiple Parameters

- Often, multiple values from the calling function are needed
- Any number of parameters can be passed in to a function

```
#include <iostream> //Need for cout.
using namespace std;

//Function: addNums - Computes sum
//of 3 ints and returns it
int addNums(int a, int b, int c);

int main(void)
{
  int num1 = 5; //Integer for test
  int num2 = 3; //Integer for test
  int result;   //Result of call

  result = addNums(num1, 6, num2);

  cout << "Result is: " << result;
  cout << endl;

  return (0);
}
```

```
int addNums(int a, int b, int c)
{
  int result;

  result = a + b + c;

  return (result); //Return sum
}
```

Result is: 14

Overloading Functions

- Multiple functions can have same name
  - Must have unique parameter list, though
- Function signature
  - Function name and types and order of parameters in parameter list
  - Functions ***must*** have a unique signature
- Overloading: Multiple functions with same name

Not Overloaded

```
//square an int, and
//return the value
int squareInt(int num);

//square a float, and
//return the value
float squareFloat(float num)

//Draw a square on
//the screen
int drawSquare(int x, int y,
               int len, int wid);
```

Overloaded

```
//square an int, and
//return the value
int square(int num);

//square a float, and
//return the value
float square(float num)

//Draw a square on
//the screen
int square(int x, int y,
           int len, int wid);
```


Overloading Example

```
int overloadSum(int a, int b, int c)
{
  cout << "(i i i) version" << endl;
  return (a + b + c);
}

float overloadSum(float a, float b, float c)
{
  cout << "(f f f) version" << endl;
  return (a + b + c);
}

float overloadSum(int a, float b, float c)
{
  cout << "(i f f) version" << endl;
  return (a + b + c);
}

//The following produces a compile error!
//int overloadSum(int a, float b, float c)
//{
//  cout << "(i f f) version" << endl;
//  return (a + b + c);
//}
```

```
int main(void)
{
  float ans;
  float f1 = 6.4;
  float f2 = 4.2;
  int   i1 = 4;
  int   i2 = 6;

  ans = overloadSum(f1, f2, f2);
  cout << ans << endl;
  //This (can) produce a compiler warning!
  ans = overloadSum(i1, i2, i2);
  cout << ans << endl;

  //This produces a compiler error!
  //ans = overloadSum(i2, i1, f1);
  //cout << ans << endl;
  ans = overloadSum(i2, (float)i1, f1);
  cout << ans << endl;

  return (0);
}
```

```
(f f f) version
14.8
(i i i) version
16
(i f f) version
16.4
```


Some Words On Scope

- Any variable declared in a function is "local" to that function
- It only exists from the time it's declared until the end of the function
  - The function add4() can NOT access the variables bar, or result, from main().
  - The function main() can NOT access the variables foo, or result, from add4().
  - Even though both functions have a variable called result - they are unique variables, in unique addresses, with unique scopes.

```
int add4(int foo)
{
  int result;

  result = foo + 4;
  return (result);
}

int main(void)
{
  int bar = 7;
  int result;

  result = add4(bar);
  return (0);
}
```


Pass-By-Value

- When a parameter is "passed-by-value" into a function, a new variable is declared to store the parameter
- The initial value of the parameter is copied from the value passed in from the calling function
- When the parameter is changed, only the copy is changed
- The following slides step through an example
  - Left side: Program with arrow indicating current statement
  - Right side: Memory contents at each step through the program
  - Bottom: Current output of program

Pass-By-Value Example, p. 1

```
//changeVal() changes the value of
//the parameter within the function
int changeVal(int val)
{
  val = (val - 15) / 2;
  return (val);
}

int main(void)
{
  int num = 19; //Integer for test
  int result;   //result of changeVal

  cout << "Starting!" << endl;
  result = changeVal(num);
  cout << "result: " << result
       << " num: " << num << endl;
  return (0);
}
```

Addr	Value	Name
1000	19	main::num
1001	?	main::result
1002		
1003		
1004		
1005		
1006		
1007		
1008		

Output

(none yet)

Pass-By-Value Example, p. 2

```
//changeVal() changes the value of
//the parameter within the function
int changeVal(int val)
{
  val = (val - 15) / 2;
  return (val);
}

int main(void)
{
  int num = 19; //Integer for test
  int result;   //result of changeVal

  cout << "Starting!" << endl;
  result = changeVal(num);
  cout << "result: " << result
       << " num: " << num << endl;
  return (0);
}
```

Addr	Value	Name
1000	19	main::num
1001	?	main::result
1002		
1003		
1004		
1005		
1006		
1007		
1008		

Output

Starting!

Pass-By-Value Example, p. 3

```
//changeVal() changes the value of
//the parameter within the function
int changeVal(int val)
{
  val = (val - 15) / 2;
  return (val);
}

int main(void)
{
  int num = 19; //Integer for test
  int result;   //result of changeVal

  cout << "Starting!" << endl;
  result = changeVal(num);
  cout << "result: " << result
       << " num: " << num << endl;
  return (0);
}
```

Addr	Value	Name
1000	19	main::num
1001	?	main::result
1002	19	changeVal::val
1003		
1004		
1005		
1006		
1007		
1008		

Output

Starting!

Pass-By-Value Example, p. 4

```
//changeVal() changes the value of
//the parameter within the function
int changeVal(int val)
{
  val = (val - 15) / 2;
  return (val);
}

int main(void)
{
  int num = 19; //Integer for test
  int result;   //result of changeVal

  cout << "Starting!" << endl;
  result = changeVal(num);
  cout << "result: " << result
       << " num: " << num << endl;
  return (0);
}
```

Addr	Value	Name
1000	19	main::num
1001	?	main::result
1002	2	changeVal::val
1003		
1004		
1005		
1006		
1007		
1008		

Output

Starting!

Pass-By-Value Example, p. 5

```
//changeVal() changes the value of
//the parameter within the function
int changeVal(int val)
{
  val = (val - 15) / 2;
  return (val);
}

int main(void)
{
  int num = 19; //Integer for test
  int result;   //result of changeVal

  cout << "Starting!" << endl;
  result = changeVal(num);
  cout << "result: " << result
       << " num: " << num << endl;
  return (0);
}
```

Addr	Value	Name
1000	19	main::num
1001	2	main::result
1002		
1003		
1004		
1005		
1006		
1007		
1008		

changeVal::val goes out of scope and is removed!

Output

Starting!

Pass-By-Value Example, p. 6

```
//changeVal() changes the value of
//the parameter within the function
int changeVal(int val)
{
  val = (val - 15) / 2;
  return (val);
}

int main(void)
{
  int num = 19; //Integer for test
  int result;   //result of changeVal

  cout << "Starting!" << endl;
  result = changeVal(num);
  cout << "result: " << result
       << " num: " << num << endl;
  return (0);
}
```

Addr	Value	Name
1000	19	main::num
1001	2	main::result
1002		
1003		
1004		
1005		
1006		
1007		
1008		

Output

Starting!
result: 2 num: 19

Pass-By-Reference

- C++ Only (Not available in C)
- Unlike pass-by-value, parameter "references" the same memory location (no copy is made)
- Accomplished by including an '&' before the parameter name in function prototype and header
- Changing the value of a reference parameter in a function changes the value of the variable in the calling function (since the same memory is referenced)
- Argument in function call MUST be a variable
  - Can not be a literal or a constant (since it could be changed)
- Allows for multiple values to be "returned" from a function
- An example is traced on the following slides

Pass-By-Reference Example, p. 1

```
//changeRef() changes the value of
//the parameter in main as well!
int changeRef(int &val)
{
  val = (val - 15) / 2;
  return (val);
}

int main(void)
{
  int num = 19; //Integer for test
  int result;   //result of changeVal

  cout << "Starting!" << endl;
  result = changeRef(num);
  cout << "result: " << result
       << " num: " << num << endl;
  return (0);
}
```

Addr	Value	Name
1000	19	main::num
1001	?	main::result
1002		
1003		
1004		
1005		
1006		
1007		
1008		

Output

Starting!

Pass-By-Reference Example, p. 2

```
//changeRef() changes the value of
//the parameter in main as well!
int changeRef(int &val)
{
  val = (val - 15) / 2;
  return (val);
}

int main(void)
{
  int num = 19; //Integer for test
  int result;   //result of changeVal

  cout << "Starting!" << endl;
  result = changeRef(num);
  cout << "result: " << result
       << " num: " << num << endl;
  return (0);
}
```

Addr	Value	Name
1000	19	main::num
1001	?	main::result
1002	1000 &	changeRef::val
1003		
1004		
1005		
1006		
1007		
1008		

Output

Starting!

Pass-By-Reference Example, p. 3

```
//changeRef() changes the value of
//the parameter in main as well!
int changeRef(int &val)
{
  val = (val - 15) / 2;
  return (val);
}

int main(void)
{
  int num = 19; //Integer for test
  int result;   //result of changeVal

  cout << "Starting!" << endl;
  result = changeRef(num);
  cout << "result: " << result
       << " num: " << num << endl;
  return (0);
}
```

Addr	Value	Name
1000	2	main::num
1001	?	main::result
1002	1000 &	changeRef::val
1003		
1004		
1005		
1006		
1007		
1008		

Output

Starting!

Pass-By-Reference Example, p. 4

```
//changeRef() changes the value of
//the parameter in main as well!
int changeRef(int &val)
{
  val = (val - 15) / 2;
  return (val);
}

int main(void)
{
  int num = 19; //Integer for test
  int result;   //result of changeVal

  cout << "Starting!" << endl;
  result = changeRef(num);
  cout << "result: " << result
       << " num: " << num << endl;
  return (0);
}
```

Addr	Value	Name
1000	2	main::num
1001	2	main::result
1002		
1003		
1004		
1005		
1006		
1007		
1008		

Output

Starting!

Pass-By-Reference Example, p. 5

```
//changeRef() changes the value of
//the parameter in main as well!
int changeRef(int &val)
{
  val = (val - 15) / 2;
  return (val);
}

int main(void)
{
  int num = 19; //Integer for test
  int result;   //result of changeVal

  cout << "Starting!" << endl;
  result = changeRef(num);
  cout << "result: " << result
       << " num: " << num << endl;
  return (0);
}
```

Addr	Value	Name
1000	2	main::num
1001	2	main::result
1002		
1003		
1004		
1005		
1006		
1007		
1008		

Output

Starting!
result: 2 num: 2

Swap Example, Multiple Reference Params

```
void swap(int &a, int &b) //Pass-by-reference!
{
  int temp;

  temp = a;
  a = b;
  b = temp;
}

int main(void)
{
  int n1 = 5;
  int n2 = 10;

  cout << "Before swap - n1: " << n1 << " n2: " << n2 << endl;
  swap(n1, n2);
  cout << "After swap  - n1: " << n1 << " n2: " << n2 << endl;
  return (0);
}
```

Before swap - n1: 5 n2: 10
After swap - n1: 10 n2: 5

Advantages of Modularity

- Cleaner code
  - A call to a function called "computeFactorial()" is compact and essentially self-documenting
  - A loop to compute the factorial would not be immediately clear
- Breaks the program into smaller pieces
  - Real world: Write specifications and prototypes for needed functions, then distribute different functions to different people - parallel coding is faster
- Easier testing
  - How to test one, huge, monolithic, 30,000 line program?
  - Modular program can be tested module by module (function by function, in this case)

Modular Testing - Driver Programs

- Driver programs allow you to test a newly written function
- The purpose of a driver program is simply to call your function and output some results to check correctness
- Most main programs in lectures so far have been driver programs - to demonstrate the use of other functions
- Especially helpful when the function you are writing is buried deep in some million line project
  - If adding functionality to a simulation that takes 12 hours to run, you don't want to have to run 50 test cases (25 days) using the entire simulation just to test one function

Modular Testing - Stubs

- Stubs allow you to test a program that is unfinished
- If waiting for someone else to finish an important function, you would still want to do some testing.
- Provide the function prototype and a "dummy" body
  - Stub does not return actual value that the function will, but allows you to call the function as if it were complete, for testing.
- This simply allows you to have the function defined in some way so when the function is ready, the stub is simply replaced with the actual function.