

EECS402 Lecture 02

Andrew M. Morgan

Savitch Ch. 3-4
Functions
Value and Reference Parameters

Andrew M. Morgan

1



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...
}
```

Andrew M. Morgan

M



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!

Andrew M. Morgan





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:

Fact. of 5 is: 120

Andrew M. Morgan

M



Complete Function Program

Here is the complete program layout

```
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
}</pre>
```

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.
                                          int addNums(int a, int b, int c)
using namespace std;
                                            int result;
//Function: addNums - Computes sum
//of 3 ints and returns it
                                           result = a + b + c;
int addNums(int a, int b, int c);
                                           return (result); //Return sum
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);
                                                          Result is: 14
 cout << "Result is: " << result;</pre>
 cout << endl;
 return (0);
                                 Andrew M. Morgan
```



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

```
//square an int, and
                                              //square an int, and
//return the value
                                              //return the value
int squareInt(int num);
                                              int square(int num);
                                    Not Overloaded
//square a float, and
                                              //square a float, and
//return the value
                                              //return the value
                                              float square(float num)
float squareFloat(float num)
//Draw a square on
                                              //Draw a square on
//the screen
                                              //the screen
int drawSquare(int x, int y,
                                              int square(int x, int y,
               int len, int wid);
                                                         int len, int wid);
                               Andrew M. Morgan
```

```
Overloading Example
int overloadSum(int a, int b, int c)
                                                int main(void)
 cout << "(i i i) version" << endl;</pre>
                                                  float ans;
 return (a + b + c);
                                                  float f1 = 6.4;
                                                  float f2 = 4.2;
                                                  int i1 = 4;
float overloadSum(float a, float b, float c)
                                                  int i2 = 6;
 cout << "(f f f) version" << endl;
                                                  ans = overloadSum(f1, f2, f2);
 return (a + b + c);
                                                  cout << ans << endl;
                                                  //This (can) produce a compiler warning!
                                                 ans = overloadSum(i1, i2, i2);
                                                  cout << ans << endl;
float overloadSum(int a, float b, float c)
 cout << "(i f f) version" << endl;</pre>
                                                  //This produces a compiler error!
 return (a + b + c);
                                                  //ans = overloadSum(i2, i1, f1);
                                                  //cout << ans << endl;
                                                  ans = overloadSum(i2, (float)i1, f1);
//The following produces a compile error!
                                                  cout << ans << endl;</pre>
                                                                          (f f f) version
//int overloadSum(int a, float b, float c)
                                                  return (0);
                                                                          14.8
// cout << "(i f f) version" << endl;
                                                                          (i i i) version
// return (a + b + c);
                                                                          16
                                                                          (i f f) version
                                                                          16.4
                                      Andrew M. Morgan
```

M

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);
}
```



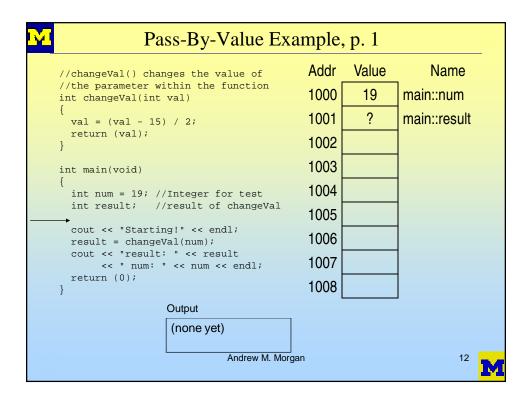
Pass-By-Value

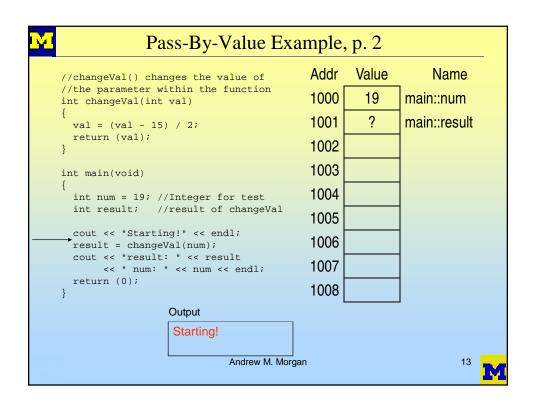
- When a parameter is "passed-by-value" into a function, a new variable is declared to store the parameter
- The initial value of the parameter is copied from the value passed in from the calling function
- · When the parameter is changed, only the copy is changed
- The following slides step through 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

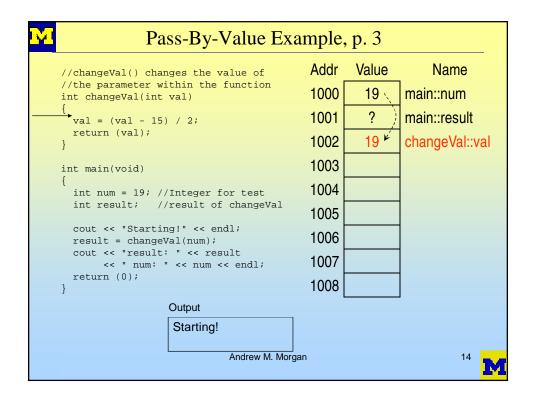
Andrew M. Morgan

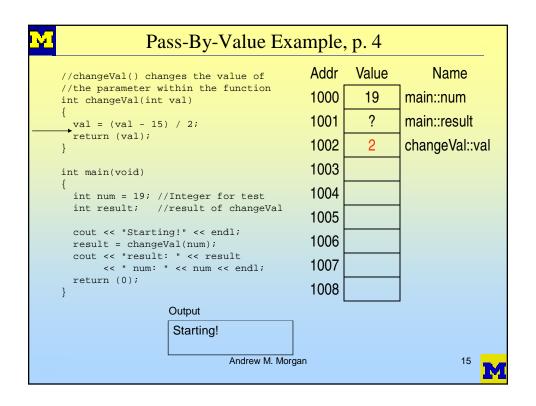
1

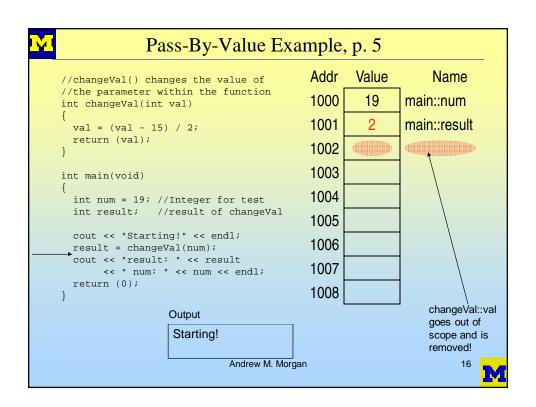


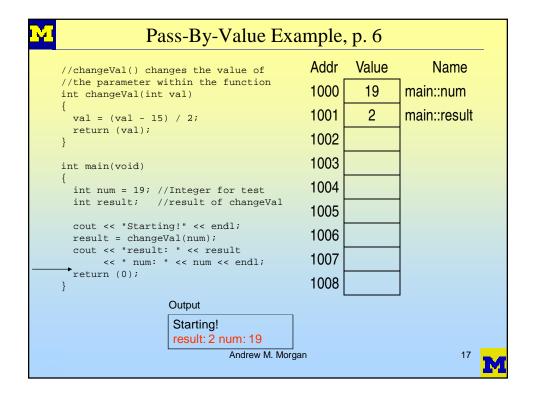














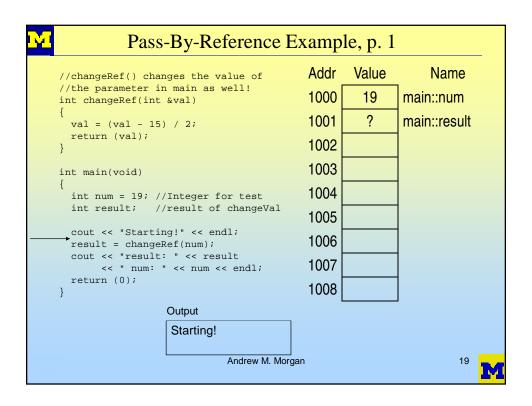
Pass-By-Reference

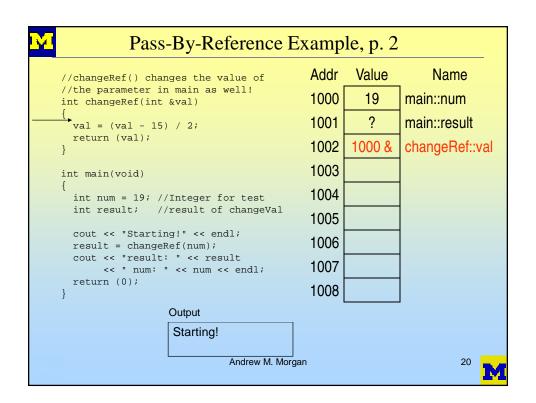
- C++ Only (Not available in C)
- Unlike pass-by-value, parameter "references" the same memory location (no copy is made)
- Accomplished by including an '&' before the parameter name in function prototype and header
- Changing the value of a reference parameter in a function changes the value of the 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

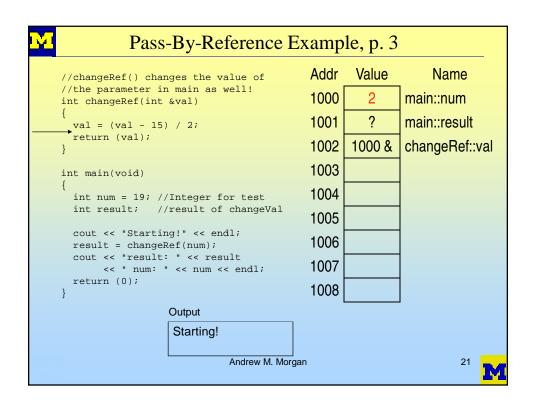
Andrew M. Morgan

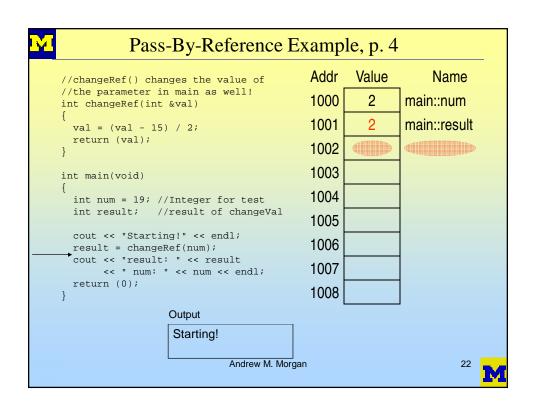
18

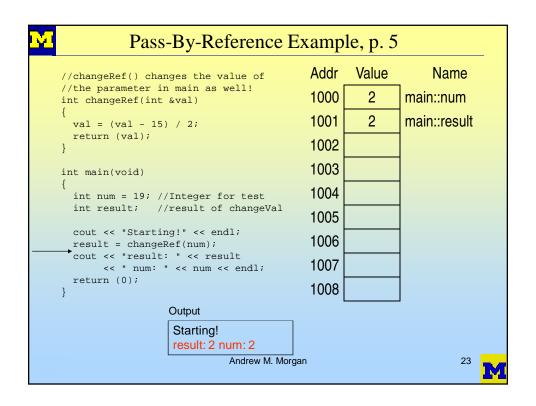












```
Swap Example, Multiple Reference Params
void swap(int &a, int &b) //Pass-by-reference!
 int temp;
 temp = a;
 a = b;
 b = temp;
int main(void)
  int n1 = 5;
 int n2 = 10;
 cout << "Before swap - n1: " << n1 << " n2: " << n2 << endl;</pre>
 swap(n1, n2);
 cout << "After swap - n1: " << n1 << " n2: " << n2 << endl;</pre>
 return (0);
                                      Before swap - n1: 5 n2: 10
                                      After swap - n1: 10 n2: 5
                           Andrew M. Morgan
```



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)

Andrew M. Morgan

25





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

Andrew M. Morgan

26





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.

