The University Of Michigan

EECS402 Lecture 02

Andrew M. Morgan

Savitch Ch. 3-4
Functions
Value and Reference Parameters

Functions

- Allows for modular programming: Write the function once, use it many times
- Parameters allow values from calling function to be used within the function
- May, or may not, return a value to calling function
  - If not, the "return type" of the function is specified as "void"
- General function template:

Function name: A C++ identifier that describes what the function does

Format parameter list: A comma separated list of type-specified values the function will be provided when called

Return type: The data type of the value the function returns

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

  for (i = 1; i <= num; i++)
  {
    factVal *= i;
  }

  return factVal;
}
```

Function body: The statements and logic the function will perform when called

Return statement: Causes the function to end and return a computed value to the caller. Not required for "void" returning functions.

Data type of value being returned must match the data type specified as the functions Return type!

Function Prototype

- A function prototype "declares a function"
  - C++ must see the prototype before the function call can be checked for proper syntax!
- Prototype is what the user will look at when they want to call the function
  - Therefore, function prototypes must include a comment to help user understand the function and what it does
- Here is the function prototype for the factorial function

```
//Computes the factorial of the input parameter
//"num", and returns the result.
int computeFactorial(int num);
```

- The function is called "computeFactorial"
  - Takes in one integer value from the calling function as a parameter
  - Returns a computed integer value to the calling function
  - Prototype is documented
- Style: The function name must be ***descriptive*** of its purpose!
- Style: The function name must be named with a verb!
- Style: The function prototype must be clearly documented with comments!

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 factVal = 1; //Value to return..
  int i;           //Loop variable

  for (i = 1; i <= num; i++)
  {
    factVal *= i;
  }
  return factVal; //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()
{
  int fact;    //Factorial result
  int val = 5;

  fact = computeFactorial(val); //function call

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

  fact = computeFactorial(val + 2); //another call
  cout << "Fact. of " << (val + 2) <<
          " is: " << fact << endl;

  return 0;
}
```

Fact. of 5 is: 120
Fact. of 7 is: 5040

Complete Function Program

- Documented prototype "declares" function
  - Included *prior to* being called in main
- Function definition / body includes implementation details and is included *after* the main function

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

//Programmer: Andrew Morgan, January 2008 (purpose here too)

//Computes the factorial of the input parameter
//"num", and returns the result.
int computeFactorial(int num); //prototype

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

  fact = computeFactorial(val); //function call

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

  fact = computeFactorial(val + 2); //another call
  cout << "Fact. of " << (val + 2) <<
          " 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
}
```

Order is Important!

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

//Programmer: Andrew Morgan, January 2008 (purpose here too)

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

  fact = computeFactorial(val); //function call

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

  fact = computeFactorial(val + 2); //another call
  cout << "Fact. of " << (val + 2) <<
          " 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
}
```

error: 'computeFactorial' was not declared in this scope
fact = computeFactorial(val); //function call

Without a prototype before main, C++ gets to the line with the function call and says "hmmm, looks like a typo since I've never heard of "computeFactorial" before

Order is Important! (version 2)

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

//Programmer: Andrew Morgan, January 2008 (purpose here too)

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
}

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

  fact = computeFactorial(val); //function call

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

  fact = computeFactorial(val + 2); //another call
  cout << "Fact. of " << (val + 2) <<
          " is: " << fact << endl;

  return 0;
}
```

This version will build and run as expected (see the important note below though!).

C++ sees the function definition of computeFactorial first, so it effectively "declares" the function when the definition is seen.

While this technically works, we will NOT use this approach in this class.

Important note: For this class, you must specify a documented prototype BEFORE main, and the function implementation must be included AFTER main.

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;

//Computes sum of all 3 provided values
int addNums(int valA, int valB, int valC);

int main()
{
  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 valA, int valB, int valC)
{
  int sumOfVals;

  sumOfVals = valA + valB + valC;

  return sumOfVals;
}
```

Result is: 14

Void Returning Functions

- Some functions don't need to return a value at all
  - In this case, specify return type as "void" and don't return anything

```
#include <iostream>
using namespace std;

//Prints the main menu to the console.
void printMenu();

int main()
{
  printMenu();

  //do other stuff

  return 0;
}

void printMenu()
{
  cout << "1. Add values" << endl;
  cout << "2. Subtract values" << endl;
  cout << "3. Quit program" << endl;
}
```


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);
}
```

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

  ans = overloadSum(f1, f2, f2);
  cout << ans << endl;

  ans = overloadSum(i1, i2, i2);
  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()
{
  int bar = 7;
  int result;

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


More On Scope: Globals

- A variable declared outside of any function is called a "Global Variable"
- It exists throughout the program, and can be accessed and modified via any function
- ***Global Variables are NOT allowed to be used in this course!***
- Global *constants* are allowed, since their values can't change, and therefore don't lead to any confusion

```
#include <iostream>
using namespace std;

int evilGlobalVar;  //NOT allowed in this course!
const int NUM_DAYS_IN_WEEK = 7; //Global constants are fine!

void rogueFunction(int paramVal)
{
  cout << "FuncParam: " << paramVal << endl;
  cout << "FuncGlobal: " << evilGlobalVar << endl;
  cout << "FuncDays: " << NUM_DAYS_IN_WEEK << endl;
  //cout << "FuncInit: " << initVal << endl;

  evilGlobalVar = 100; //bwahahaha
  //NUM_DAYS_IN_WEEK = 10;
}

int main()
{
  int initVal;

  initVal = 40;
  evilGlobalVar = 50;

  rogueFunction(initVal);

  //cout << "MainParam: " << paramVal << endl;
  cout << "MainGlobal: " << evilGlobalVar << endl;
  cout << "MainDays: " << NUM_DAYS_IN_WEEK << endl;
  cout << "MainInit: " << initVal << endl;

  return 0;
}
```

```
FuncParam: 40
FuncGlobal: 50
FuncDays: 7
MainGlobal: 100
MainDays: 7
MainInit: 40
```


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()
{
  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()
{
  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()
{
  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()
{
  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()
{
  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()
{
  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()
{
  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()
{
  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()
{
  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()
{
  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()
{
  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 &valA, int &valB) //Pass-by-reference!
{
  int temp;

  temp = valA;
  valA = valB;
  valB = temp;
}

int main()
{
  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 named "computeFactorial()" is compact and essentially self-documenting
  - A loop to compute the factorial would not be immediately clear
- Non-duplication
  - Factorial algorithm implemented once, can be used simply by calling the function as needed
- 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)

Remember Short-Circuiting??

```
bool isLessThan20(int inVal)
{
  return inVal < 20;
}

bool changeValTo100(int &valToChange)
{
  bool didChangeIt;

  if (valToChange == 100)
  {
    didChangeIt = false;
  }
  else
  {
    valToChange = 100;
    didChangeIt = true;
  }

  return didChangeIt;
}
```

```
int main()
{
  int aValue = 60;
  int bValue = 30;

  if (isLessThan20(aValue) || changeValTo100(bValue))
  {
    cout << "AB if expression was true!!!" << endl;
  }
  cout << "bValue is: " << bValue << endl;

  int cValue = 10;
  int dValue = 30;

  if (isLessThan20(cValue) || changeValTo100(dValue))
  {
    cout << "CD if expression was true!!!" << endl;
  }
  cout << "dValue is: " << dValue << endl;

  return 0;
}
```

```
AB if expression was true!!!
bValue is: 100
CD if expression was true!!!
dValue is: 30
```

Note that dValue did NOT get updated!!

Since "isLessThan20(cValue)" evaluated to true, there's *no need* to evaluate "changeValTo100(dValue)" at all - the changeValTo100 function doesn't even get called in that case!

Pre-Existing Functions

- C++ standard libraries contain many functions that you usually do not have to write algorithms for yourself
- Example: Many math related functions in a standard math library
  - Must #include <cmath> to access these functions (and include the "using namespace std;" line)
  - Once the library is #include'd, you may utilize functions from the library
    - If you don't #include <cmath>, then you should get an error when trying to use functionality from the math library!
  - Some available functions:
    - double sin(double x)
    - double cos(double x)
    - double pow(double base, double exponent)
    - double sqrt(double x)
    - Etc…

Using Standard Math Library, Example Output

```
#include <iostream>
#include <cmath>
using namespace std;

int main()
{
  double checkVal;
  double resultVal;

  checkVal = 4;
  resultVal = pow(checkVal, 3.0);
  cout << checkVal << "^3.0 = " << resultVal << endl;

  checkVal = 65;
  resultVal = sqrt(checkVal);
  cout << "sqrt of " << checkVal << " = " << resultVal << endl;

  return 0;
}
```

Note: this is *required*, even if you happen to use an IDE that allows you "get away without it"

```
4^3.0 = 64
sqrt of 65 = 8.06226
```


Using Standard Math Library, Example Output

```
#include <iostream>
//#include <cmath>
using namespace std;

int main()
{
  double checkVal;
  double resultVal;

  checkVal = 4;
  resultVal = pow(checkVal, 3.0);
  cout << checkVal << "^3.0 = " << resultVal << endl;

  checkVal = 65;
  resultVal = sqrt(checkVal);
  cout << "sqrt of " << checkVal << " = " << resultVal << endl;

  return 0;
}
```

Note: this is commented for this example!

These functions aren't available unless you #include <cmath>!

```
LinuxPrompt> g++ mathstuff.cpp -o mathstuff
mathstuff.cpp: In function 'int main()':
mathstuff.cpp:11:32: error: 'pow' was not declared in this scope
  resultVal = pow(checkVal, 3.0);
                               ^
mathstuff.cpp:15:28: error: 'sqrt' was not declared in this scope
  resultVal = sqrt(checkVal);
```


Default Parameters To Functions

- If a function's parameter is usually a known value, the parameter can be given a "default value"
  - Default parameters ***must*** be at the end of the parameter list
  - Default values specified in function prototype
  - Argument is not required for default parameters
    - Default value will be used if no argument is provided

```
void print(int valA, int valB = 4);

int main()
{
  print(8, 16);
  print(7);
  //print(); //Will not compile!
  return (0);
}

void print(int valA, int valB)
{
  cout << valA << " " << valB << endl;
}
```

Default parameter (default value is 4)

```
8 16
7 4
```


Additional Reference Material

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.