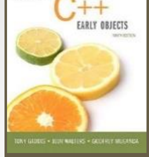
COP2334

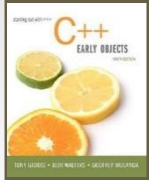
Introduction to Object Oriented Programming with C++

D. Singletary

<u>Module 5</u>

Ch. 6 Functions

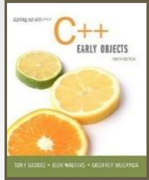




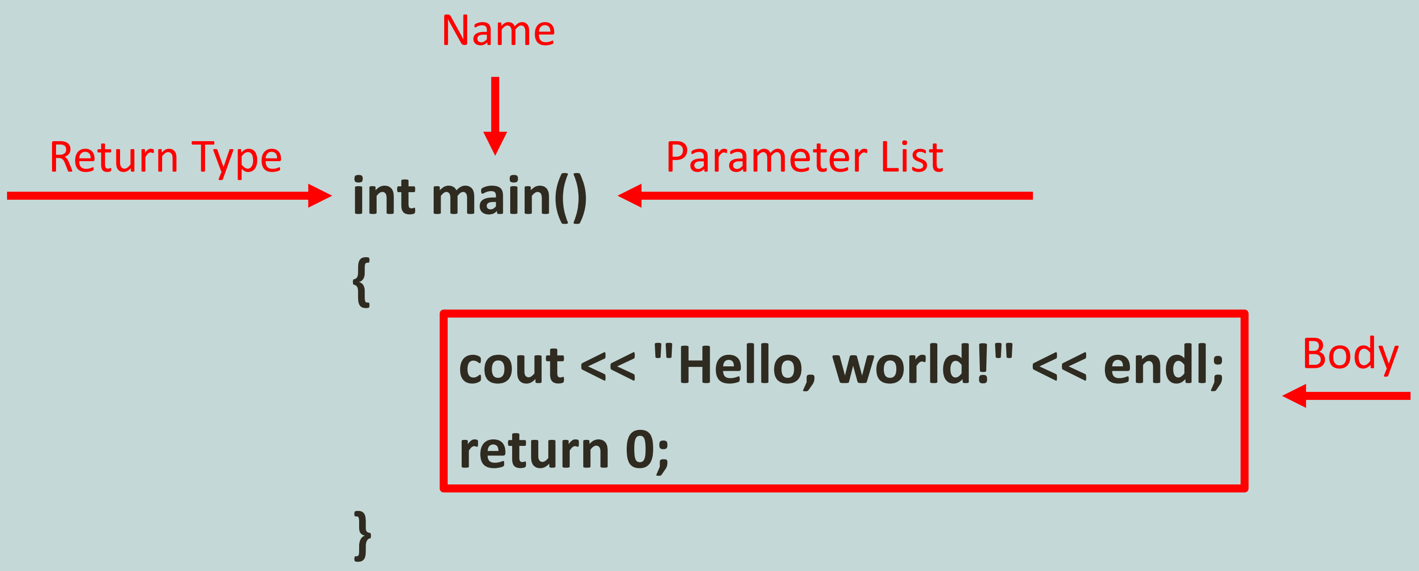
Modular Programming

- Modular programming means to use functions to break a problem into small, manageable pieces ("divide and conquer", "stepwise refinement")
  - Breaking a program into modules by using functions simplifies them
  - Functions allow a complex operation to be performed as needed ("code reuse")

```
int main()
{
        cout << "Hello, world!" << endl;
        return 0;
}
```

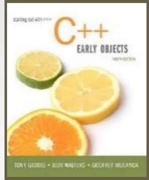


The Structure of a Function

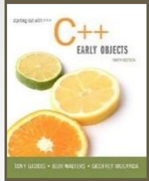


```
int main()
{
    cout << "Hello, world!" << endl;
    return 0;
}
```

- Name: function names are similar to variables
  - camel-hump notation
  - meaningful
- Parameter list: stores arguments passed into the function
  - if no parameters, the list is empty, denoted by empty parentheses ()
- Body: the statements which perform the task that the function is performing
  - enclosed in curly braces {}
- Return type: the type of the value returned to the caller of the function (e.g. int, float, double, char, ...)
  - a function returning nothing returns a void type
  - no return statement is necessary for functions returning void

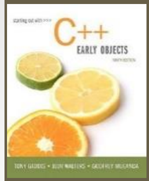


Function Definition Examples



```
int calcIntSquare(int i)
{
	return i * i;
}

double calcTotalPrice(double p, double  tax)
{
	return p * (1.0 + tax);
}

void displayMessage()
{
	cout << "Hello, World!" << endl;
}
```

Function Call Examples

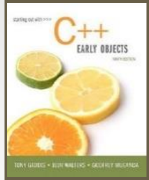


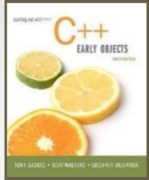
```
int sqVal = calcIntSquare(4);
cout << "square of 4 is " << sqVal << endl;
cout << "square of 8 is " << calcIntSquare(8) << endl;

cout << "total price is " <<
        calcTotalPrice(10.00, 0.06) << endl;

displayMessage();

cout << displayMessage() << endl;  // WRONG!
// do not use a function returning void as an argument

string msg = displayMessage(); // WRONG!
// do not assign a void result
```

Function Prototypes

- Functions must be "known" by the caller in order to be used correctly.
- One way we accomplish this is to define the function before it is called:

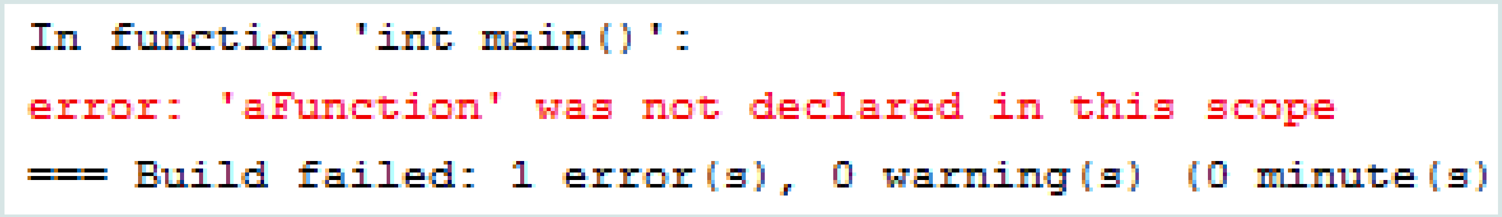
```
int aFunction(int i)    // function is defined before calling
{
     return i + i;
}

int main()
{
     int x = aFunction(10);    // main function "knows" aFunction
     return 0;
}
```





- We usually want to position the main function at the top of our program. But if we move the function below main(), the caller does not know about it and the program does not build:

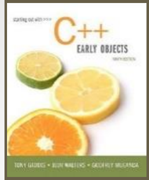
```
int main()
{
    int x = aFunction(10);   // main function does not know aFunction
    return 0;
}

int aFunction(int i)   // function is defined after calling
{
    return i + i;
}
```

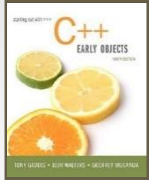


```
In function 'int main()':
error: 'aFunction' was not declared in this scope
=== Build failed: 1 error(s), 0 warning(s) (0 minute(s)
```

- Declaring a function prototype (a function header terminated with a semi-colon (no function body) "informs" the caller about the function so the program can build successfully

```
int aFunction(int i);    // this is a function prototype

int main()
{
      int x = aFunction(10);    // main function is aware of aFunction
      return 0;
}

int aFunction(int i)    // function is defined before being called
{
      return i + i;
}
```



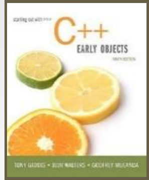


- A function prototype must be declared exactly like the header of the defined function, but with a terminating semi-colon and no body:

```
int aFunction(int i);    // this is a function prototype

int aFunction(int i)      // this is the function header
{
        return i + i;
}
```

- More prototypes:

```
double anotherFunction(double d1, double d2);

void aVoidFunction(float x, float y, float z);

// following is legal in C++, but not in this course!
char aCharFunction(int);  // no parameter name
```



Passing Data Into a Function

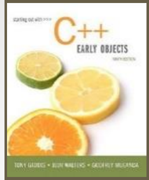
- Data is passed into a function by using arguments in the function call, which are then saved as parameters in the function:

```
int total = add2Numbers(2, 4);
```

2 and 4 are function arguments

```
int add2Numbers(int num1, int num2)
{
        return num1 + num2;
}
```

2 is saved in the parameter num1 and 4 is saved in the parameter num2



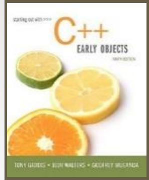
- Note: parameters cannot be combined:

```
int add2Number(int num1, num2);
```

- This results in a build error:
  - error: 'num2' has not been declared

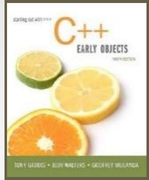
- Argument/parameter types do not have to match exactly (upcasts are not a problem), but be careful when mixing types:

```
double d1 = 2.5, d2 = 4.7;
int total = add2Numbers(d1, d2);

int add2Numbers(int num1, int num2) { ...
```



- Identify the problem with this function call:

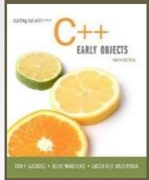
```
int add2TinyNumbers(char num1, char num2);

int main()
{
    int i1 = 2000, i2 = 4000;
    cout << "result = " << add2TinyNumbers(i1, i2) << endl;

    return 0;
}

int add2TinyNumbers(char num1, char num2)
{
   return num1 + num2;
}
```



Passing Data by Value

- Passing arguments into function parameters results in a copy of the arguments being made
  - The original arguments are not modified by the function
- This is referred to as "pass by value"

```
int main() {
        ...
        int number = 12;
        changeMe(number);  // pass number
        ...  // original value of number is unchanged

void changeMe(int myVal)  // myVal = number
{
        myVal = 0;
}
```

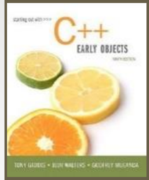


```
void changeMe(int aVal);

int main()
{
    int number = 12;

    // display number
    cout << "in main, number is " << number << endl;

    // call changeMe function, pass number's value as argument
    changeMe(number);

    cout << "back in main, number is " << number << endl;
    return 0;
}

// change the passed value
void changeMe(int myVal)
{
    myVal = 0;

    cout << "in changeMe, myVal is " << myVal << endl;
}
```

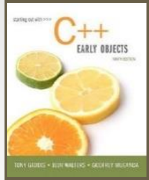
Notice that prototype parameter names do not have to match actual parameter names (but they should)

```
in main, number is 12
in changeMe, myVal is 0
back in main, number is 12
```

The return Statement

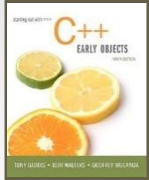


- The return statement causes a function to return immediately
  - It may or may not send a value back to the caller
- Functions that return a value: **return <value>;**
  - <value> is a value conforming to the declared return type and must be included
  - Only one value can be returned
- Functions that return void: **return;**
  - No value is included
  - The final return statement in a function returning void can be implied
  - Explicit return statements cause earlier return

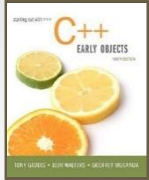


- Good programming practices use a single return statement in a function
  - A single "exit point" at the end of the function makes programs easier to maintain

```
int calculateWithManyReturns(int num)
{
        if (num < 10)
                return num + 5;
        else if (num < 20)
                return num + 10;
         else if (num < 30)
                return num + 20;
         return 50;
}
```



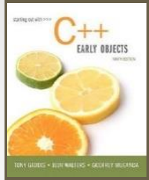
```
int calculateWithOneReturn(int num)
{
        int result = 50;

        if (num < 10)
                result = num + 5;
        else if (num < 20)
                result = num + 10;
         else if (num < 30)
                result = num + 20;

          return result;
}
```



- // Identify the logic error in this function

```
int add2Numbers(int num1, int num2)
{
        if (num1 == 0)
                return 0;
        else if (num2 == 0)
                return 0;
        else if (num1 + num2 > 0)
                return num1 + num2;
}
```

Returning a Boolean Value



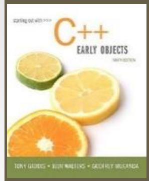
- Functions which return boolean (bool) values of false or true (0 and 1) have names which start with the prefix "is" by convention:

```
bool isEven(int number);
bool isValid(string s);
```

- Usage:

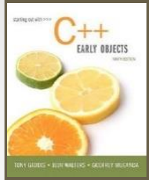
```
if (isEven(val))   // same as if (isEven(val) == true)
      cout "val is even" << endl;
else
      cout "val is odd" << endl;
```

```
// returns true if number is even
bool isEven(int number)
{
        bool result = false;
        if (number % 2 == 0)
                result = true;

        return result;
}
```



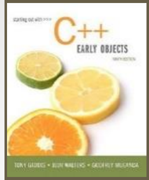
Using Functions in a Menu

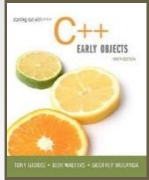
- Functions are well-suited for use in menu-driven programs.
  - Instead of performing complex calculations inside an if-else or switch block, we can call functions with specific arguments
  - The main function then is used as a way to organize the program flow instead of representing a single large and difficult-to-maintain block of code.



```
do
{
    displayMenu();
    choice = getChoice();
    if (choice != MENU_EXIT)
    {
        switch (choice)
        {
            case MENU_ADULT:
                showFees(ADULT_RATE);
                break;
            case MENU_CHILD:
                showFees(CHILD_RATE);
                break;
            case MENU_SENIOR:
                showFees(SENIOR_RATE);
                break;
        }
    }
}  while (choice != MENU_EXIT);
```

menuwithfunction.cpp





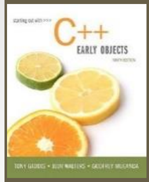
The system Function

- system(command) can be used to execute commands in the operating system environment
  - e.g. to clear the screen in Windows

```
#include <cstdlib>
…
system("cls");
```

- Should be avoided as it can be a security risk
  - https://www.securecoding.cert.org/confluence/pages/viewpage.action?pageId=2130132

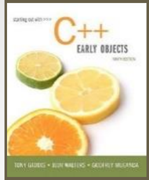
Local vs. Global Variables

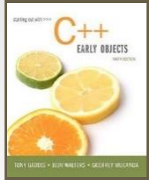


- Local variables are declared in functions; their scope extends within the block in which they are declared (e.g. top of function through the end of the function, inside a for loop, etc)
- Global variables are declared outside of any function; their scope extends from the point of declaration through the end of the file

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

void incrGlobalCount();

int gCount = 0;  // global counter variable

int main()
{
```

- Global variables should only be used in rare instances
  - They make debugging difficult
  - They make code less portable
  - They make you lose points on assignments unless they are explicitly called for by the instructor
- Use local variables and pass via arguments and function parameters instead
- Instructor's convention: prefix global variables with "g" to indicate their "global-ness"

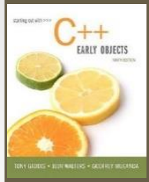




```
#include <iostream>
using namespace std;

void incrGlobalCount();

int gCount = 0;

int main()
{
   cout << "global count = " << gCount << endl;
   gCount++;
   incrGlobalCount();
   cout << "global count = " << gCount << endl;
   return 0;
}

// increment the global variable
void incrGlobalCount()
{
   gCount++;
   return;
}
```

```
global count = 0
global count = 2
```

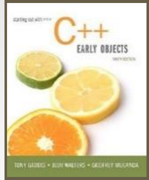
global.cpp



Global Named Constants

- While global variables are inherently evil and should be avoided, global constants are chocolate-coated goodness and should be enthusiastically embraced.

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

const int MAXCOUNT = 100;  // global, unchangeable

int main()
{
```

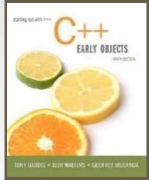


Shadowing Variables

- Avoid giving global variables (or constants) the same name as local variables or parameters
- This results in the local variable shadowing (hiding) the global and preventing its use
  - hard-to-maintain
  - unintended consequences

```
int gCount = 0;

int main()
{
   cout << "global count = " << gCount << endl;
   gCount++;
   incrGlobalCount();
   incrLocalCount();
   cout << "global count = " << gCount << endl;
   return 0;
}

// increment the global counter
void incrGlobalCount()
{
   gCount++;
   return;
}

// increment a local counter
void incrLocalCount()
{
   int gCount = 100;
   gCount++;
   cout << "local count = "  << gCount << endl;
}
```

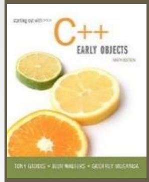
```
global count = 0
local count = 101
global count = 2
```

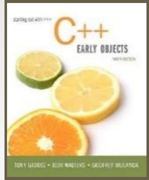


Static Local Variables

- Local variables are "destroyed" (cease to exist) when a function exits
- We may want a function to "remember" a local variable's value from one call to the next
- Declaring a variable as static allows it to remain in existence throughout the life of the program

```
void showStatic()
{
    static int numCalls = 0;

    cout << "This function has been called " <<
        ++numCalls << " times. " << endl;
}
```



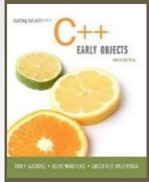


```
// static.cpp
// demonstrates global variables

#include <iostream>
using namespace std;

void showStatic();

int main()
{
   for (int i = 0; i < 5; i++)
      showStatic();
   return 0;
}

// increment and display a static value
void showStatic()
{
   static int numCalls = 0;  // initialization only happens once

   cout << "This function has been called " << ++numCalls << " times. " << endl;
}
```

```
This function has been called 1 times.
This function has been called 2 times.
This function has been called 3 times.
This function has been called 4 times.
This function has been called 5 times.
```

static.cpp

Default Arguments



- Default arguments are passed to a function's parameters automatically if no argument is provided in the function call
  - Usually listed in the prototype

```
void showArea(double length = 20.0, double width = 10.0);
```

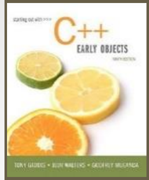
- The values must be literals or constants (cannot be variables)
- The function can be called with or without arguments

```
showArea();    // use default arguments
showArea(12.0);   // length provided, use default width
showArea(12.0, 5.5);    // length and width
```

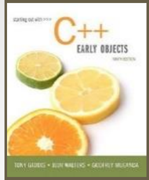
```
void displayStars(int starsPerRow = 10, int numRows = 1);

int main()
{
	cout << "1. default:" << endl;
	displayStars();
	cout << "2. 5 per row, default rows:" << endl;
	displayStars(5);
	cout << "3. 7 per row, 3 rows:" << endl;
	displayStars(7,3);
	return 0;
}

// display specified number of stars per row
void displayStars(int starsPerRow, int numRows)
{
	for (int row = 1; row <= numRows; row++)
	{
		for (int star = 1; star <= starsPerRow; star++)
			cout << '*';
		cout << endl;
	}
}
```

```
1. default:
**********
2. 5 per row, default rows:
*****
3. 7 per row, 3 rows:
*******
*******
*******
```

defaultargs.cpp

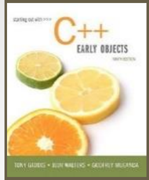


Using Reference Variables as Parameters



- A reference variable is a variable that references the memory location of another variable (vs. just making a copy of its value)
  - Any change made to the reference variable is actually made to the variable it references
  - Reference variables can be used as function parameters by using an ampersand ('&')

```
void doubleNum(int &refVar) // refVar is a "reference to an int"
{
	refVar *= 2;
}
```



- Function prototypes for functions which use reference variables must also include the ampersand
  - The ampersand can be adjacent to the type or the variable (be consistent - I prefer it next to the type)

```
void doubleNum(int& refVar);
…
void doubleNum(int& refVar)
{
```

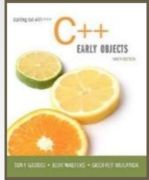
- The ampersand is not used in the function call

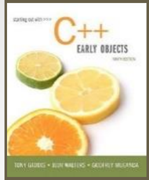
```
int a = 5;
doubleNum(a);  // after function returns, a == 10
```

```
void doubleNum(int& refVar);

int main()
{
   int value = 4;
   const int MAXVAL = 10;

   cout << "in main, value is " << value << endl;
   cout << "calling doubleNum..." << endl;
   doubleNum(value);
   cout << "back in main, value is " << value << endl;
   // doubleNum(10); // Error: cannot call with a literal!
   // doubleNum(MAXVAL); // Error: cannot call with a const!
   // doubleNum(value * value); // Error: cannot call with an expression
   return 0;
}

// double the number passed as refVar
void doubleNum(int& refVar)
{
   refVar *= 2;
}
```

```
in main, value is 4
calling doubleNum...
back in main, value is 8
```

refvars.cpp



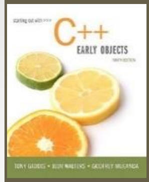


```
void addThree(int& num1, int& num2, int& num3, int& sum);

int main()
{
    int in1 = 0, in2 = 0, in3 = 0, sum = 0;

    addThree(in1, in2, in3, sum);

    cout << "added " << in1 << " + " << in2 << " + " << in3 <<
            ", result = " << sum << endl;

    return 0;
}

// input and add three values
void addThree(int& num1, int& num2, int& num3, int& sum)
{
    cout << "Enter three integer values: ";
    cin >> num1 >> num2 >> num3;
    sum = num1 + num2 + num3;
}
```

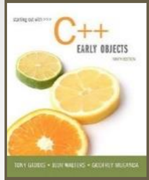
```
Enter three integer values:
3 5 7
added 3 + 5 + 7, result = 15
```

addthree.cpp

Pass by Value vs. Pass by Reference



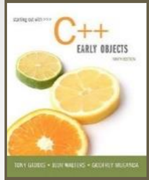
- Pass by **value** when:
  - argument is a const, literal, or expression
  - variable should not have its value changed
  - when exactly one value needs to be returned (use a return statement)
- Pass by **reference** when:
  - two or more variables passed to the function need to be modified
  - when pass by reference is required (e.g file stream objects)



Overloading Functions

- Two or more functions may have the same name if their parameter lists are different

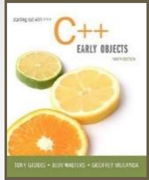
```
int square(int number);
double square(double number);
...

// square the number
int square(int number)
{
    return number * number;
}

// square the number (overloaded)
double square(double number)
{
    return number * number;
}
```



```
int square(int number);
double square(double number);

int main()
{
   int iValToSquare = 10;
   double dValToSquare = 10.0;

   cout << "squared integer = " << square(iValToSquare) << endl;
   cout << fixed << setprecision(2) << "squared double = " << square(dValToSquare) << endl;

   return 0;
}

// square the number
int square(int number)
{
   cout << "    in integer square function" << endl;
   return number * number;
}

// square the number (overloaded)
double square(double number)
{
   cout << "    in double square function" << endl;
   return number * number;
}
```

```
    in integer square function
squared integer = 100
    in double square function
squared double = 100.00
```

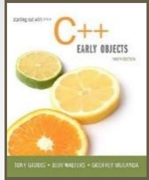
overload.cpp



```
int multiply(int n1, int n2);
int multiply(int n1, int n2, int n3);
int multiply(int n1, int n2, int n3, int n4);

int main()
{
	cout << "multiply 2 = " << multiply(2, 3) << endl;
	cout << "multiply 3 = " << multiply(2, 3, 4) << endl;
	cout << "multiply 4 = " << multiply(2, 3, 4, 5) << endl;
	return 0;
}

// multiply the numbers
int multiply(int n1, int n2)
{
	return n1 * n2;
}

int multiply(int n1, int n2, int n3)
{
	return n1 * n2 * n3;
}

int multiply(int n1, int n2, int n3, int n4)
{
	return n1 * n2 * n3 * n4;
}
```

```
multiply 2 = 6
multiply 3 = 24
multiply 4 = 120
```

overload2.cpp



The exit() Function

- exit(arg) causes a program to terminate, regardless of which function or control mechanism is executing
  - "arg" value is passed to the operating system
- not recommended, can leave resources in an inconsistent state; usually used when serious errors have occurred

```
void doCalculation();

int main()
{
      doCalculation();
      return 0;
}

void doCalculation()
{
      cout << "2 + 2 is 4" << endl;
      exit(0); // unconditional exit
}
```