What we will learn about this week:

- Procedural Abstraction (Information Hiding)
- Void functions
- Call-by-reference parameters

Review

- Top-Down Design
- Pre-defined functions
- Programmer definer functions
  - function prototype
  - function call
  - function header
  - function definition
- Type casting

Buying Pizza

Problem Definition: Buying the large "economy" size does not always save money. The program will compare the sizes of two pizzas to determine which is the better buy.

Input: diameter and price of two pizzas

Output: cost per square inch for each pizza , which is the better buy.

Analysis:

1. Get input for both pizzas
2. Compute price per square inch for large pizza
3. Compute price per square inch for small pizza
4. Determine which is the better buy
5. Display the results

Display 3.9 Buying Pizza *(part 1 of 2)*

```
//Determines which of two pizza sizes is the best buy.
#include <iostream>
using namespace std;

double unitprice(int diameter, double price);
//Returns the price per square inch of a pizza. The formal
//parameter named diameter is the diameter of the pizza in inches.
//The formal parameter named price is the price of the pizza.

int main( )
{
    int diameter_small, diameter_large;
    double price_small, unitprice_small,
           price_large, unitprice_large;

    cout << "Welcome to the Pizza Consumers Union.\n";
    cout << "Enter diameter of a small pizza (in inches): ";
    cin >> diameter_small;
    cout << "Enter the price of a small pizza: $";
    cin >> price_small;
```

```
cout << "Enter diameter of a large pizza (in inches): ";
cin >> diameter_large;
cout << "Enter the price of a large pizza: $";
cin >> price_large;

unitprice_small = unitprice(diameter_small, price_small);
unitprice_large = unitprice(diameter_large, price_large);

cout.setf(ios::fixed);
cout.setf(ios::showpoint);
cout.precision(2);
cout << "Small pizza:\n"
     << "Diameter = " << diameter_small << " inches\n"
     << "Price = $" << price_small
     << " Per square inch = $" << unitprice_small << endl
     << "Large pizza:\n"
     << "Diameter = " << diameter_large << " inches\n"
     << "Price = $" << price_large
     << " Per square inch = $" << unitprice_large << endl;
```

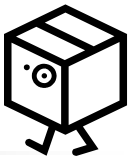
Display 3.9 Buying Pizza *(part 2 of 2)*

```
    if (unitprice_large < unitprice_small)
        cout << "The large one is the better buy.\n";
    else
        cout << "The small one is the better buy.\n";
    cout << "Buon Appetito!\n";

    return 0;
}

double unitprice(int diameter, double price)
{
    const double PI = 3.14159;
    double radius, area;

    radius = diameter/double(2);
    area = PI * radius * radius;
    return (price/area);
}
```

Sample Dialogue

```
Welcome to the Pizza Consumers Union.
Enter diameter of a small pizza (in inches): 10
Enter the price of a small pizza: $7.50
Enter diameter of a large pizza (in inches): 13
Enter the price of a large pizza: $14.75
Small pizza:
Diameter = 10 inches
Price = $7.50 Per square inch = $0.10
Large pizza:
Diameter = 13 inches
Price = $14.75 Per square inch = $0.11
The small one is the better buy.
Buon Appetito!
```

Information Hiding
(Black Box Analogy)



- **A function's author (programmer) should know everything about how the function does its job, but nothing but specifications about how the function will be used.**

- **The client programmer -- the programmer who will call the function in her code -- should know only the function specifications, but nothing about how the function is implemented.**

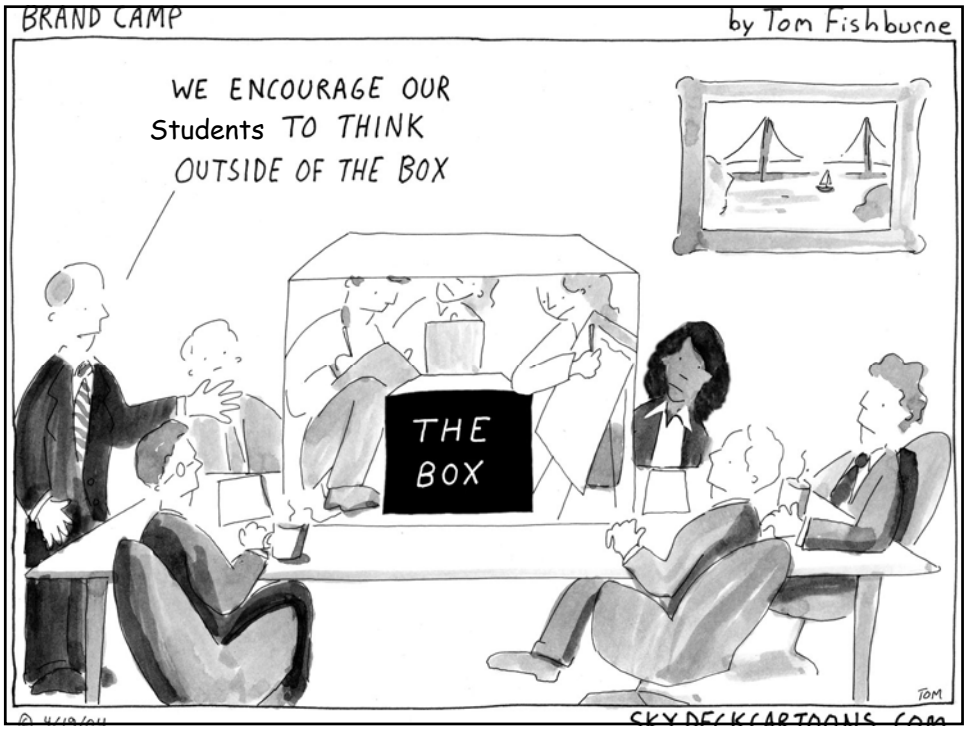
Display 3.7 Page 118

Definition 1

```
double new_balance(double balance_par, double rate_par)
{
    double interest_fraction, interest;

    interest_fraction = rate_par/100;
    interest = interest_fraction*balance_par;
    return (balance_par + interest);
}
```

Definition 2

```
double new_balance(double balance_par, double rate_par)
{
    double interest_fraction, updated_balance;

    interest_fraction = rate_par/100;
    updated_balance = balance_par*(1 + interest_fraction);
    return updated_balance;
}
```



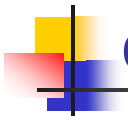
Local Variables

- Variables that are defined within the function are local variables.
- Local variables belong only to the function and cannot be used outside of the function.
- Even if a variable within the main program has the same name, they are different variables.

Scope

Variables local to a function are said to have that function as their *scope.*

Variables that are defined within the main program are local to the main program and have the main program as their *scope*.



Global Constants and Parameters

- If a constant will be used by more than one function, its declaration can be placed outside the body of the main program at the beginning of your program making the constant global.
- A global constant can be used in any function that follows (including main).

Example of Global Constants

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

//global constants
const double PI = 3.14;

//function prototypes

int main()
{
        //body of main
}
```

Global variables

- It is possible to declare global variables in the same way as global constants (without the const, of course)
- But unlike constants that can't be changed, there can be confusion caused by using global variables. For example, a function uses the same variable name as one of the global variables.

Call-by-Value parameters

- The formal parameters for the function, which are local to the function, are initialized to the value of the arguments passed in the function call.
- Even if they have the same name, the variables local to main are **not changed** by the function, even if it has the same name as the parameter.
- Remember that you do not have to declare a passed parameter in the function.
- Order of arguments is the order they're plugged into parameters, regardless of what they're called.

Take a break!!!

Void Functions

- does not return a value
  - There is no "*return* expression; " statement
    (Though a return; with no expression is allowed.)
- keyword void where type returned would be
  - *void* type replaces the more familiar return type such as *int* or *double*.
- A call to a void function is an executable statement, rather than being part of an expression.

Syntax for a Void function

```
// function prototype
void function_name (parameter list);

// function definition
void function_name (parameter list) // header
{
//function body
}
```

Examples

- Example without a return statement :
  Display 4.2 page 160
- Example with a return statement:
  Display 4.3 page 162

Display 4.2 void-Function - (1 of 4)

```
//Program to convert a Fahrenheit temperature to a Celsius temperature.
#include <iostream>


void initialize_screen( );
//Separates current output from the output
//of the previously run program.

double celsius(double fahrenheit);
//Converts a Fahrenheit temperature to a Celsius temperature.

void show_results(double f_degrees, double c_degrees);
//Displays output. Assumes that c_degrees
//Celsius is equivalent to f_degrees Fahrenheit.
```

Display 4.2 void-Function - (2 of 4)

```
int main( )
{

   using namespace std;
   double f_temperature, c_temperature;

   initialize_screen( );
   cout << "I will convert a Fahrenheit temperature"
        << " to Celsius.\n"
        << "Enter a temperature in Fahrenheit: ";
   cin >> f_temperature;

   c_temperature = celsius(f_temperature);

   show_results(f_temperature, c_temperature);
   return 0;
}
```

Display 4.2 void-Function - (3 of 4)

```
//Definition uses iostream:
void initialize_screen( )
{
   using namespace std;
   cout << endl;
   return;                    //The return is optional.
}



double celsius(double fahrenheit)
{
   return ((5.0/9.0)*(fahrenheit - 32));
}
```

Display 4.2 void-Function - (4 of 4)

```
//Definition uses iostream:
void show_results(double f_degrees, double c_degrees)
{
   using namespace std;
   cout.setf(ios::fixed);
   cout.setf(ios::showpoint);
   cout.precision(1);
   cout << f_degrees
        << " degrees Fahrenheit is equivalent to\n"
        << c_degrees << " degrees Celsius.\n";
   return;                                  // The return is optional.
}
```

Return Statements in void functions

- In C++ both void functions and value-returning functions can have return statements.
- In value-returning functions the return must have an argument. In *void*-functions, the return must *NOT* have an argument.
- There is an implicit (compiler generated) return statement at the final closing brace of a *void* function. This does not mean you never need a return; in a *void*-function.

Display 4.3 page 162 – Use of return in a void function

```
// function prototype
void ice_cream_division (int number, double total_weight);

// function definition
void ice_cream_division (int number, double total_weight)
{
    using namespace std;
    double portion;

    if (number == 0)
        return;                    // return if none
    portion = total_weight / error;
    cout << Each one receives " << portion
        << " ounces of icecream." << endl;
}
```

Why do we need **call-by-reference** parameters?

- Input subtasks should be carried out with a function call. This is not adequate for more than one return value. We need another mechanism.
- With a Call-by-Value parameter, the corresponding argument is only read for its value. The argument can be variable, but this is not necessary. The parameter is initialized with the **value** of the value-parameter and the value of the variable itself does **not** change.
- With Call-by-Reference, the corresponding argument **must** be variable, and the behavior of the function is ***as if the variable were substituted for the parameter.***

Call-by-reference parameters

- It is the **address** (memory location) of the variable that is passed to the function, not the value, so it acts as if the actual variable is substituted for the parameter.
- The **value** of the variable **is changed** by the function.
- The **ampersand &** is added to the type name to indicate call-by-reference.
- Call by reference can be used to return multiple values

Display 4.4 Call by Reference parameters (1 of 4)

```
//Program to demonstrate call-by-reference parameters.
#include <iostream>

void get_numbers(int& input1, int& input2);
//Reads two integers from the keyboard.

void swap_values(int& variable1, int& variable2);
//Interchanges the values of variable1 and variable2.

void show_results(int output1, int output2);
//Shows the values of variable1 and variable2, in that order.
```

Display 4.4 Call by Reference parameters (2 of 4)

```
int main( )
{
   int first_num, second_num;

   get_numbers(first_num, second_num);
   swap_values(first_num, second_num);
   show_results(first_num, second_num);
   return 0;
}
```

Display 4.4 Call by Reference parameters (3 of 4)

```
//Uses iostream:
void get_numbers(int& input1, int& input2)
{
   using namespace std;
   cout << "Enter two integers: ";
   cin >> input1
       >> input2;
}

void swap_values(int& variable1, int& variable2)
{
   int temp;

   temp = variable1;
   variable1 = variable2;
   variable2 = temp;
}
```

Display 4.4 Call by Reference parameters (4 of 4)

```
//Uses iostream:
void show_results(int output1, int output2)
{
   using namespace std;
   cout << "In reverse order the numbers are: "
        << output1 << " " << output2 << endl;
}
```

Mixed parameter lists

It is entirely feasible to have call-by-value parameters mixed in with call-by- reference parameters.

Example:

```
void good_stuff (int& par1, int par2, double& par3);

good_stuff ( arg1, 17, arg3);       //function call
```

The constant 17 is permissible because par2 is a value parameter.

The code in the body of the function has the ability to change arg1 and arg3.

☹ Pitfall ☹

Omitting an ampersand (&) when you intend a reference parameter is a mistake that bites twice:

- First it makes your code run incorrectly, the compiler probably won't catch it.
- And it is very difficult to find because *it looks right*.

Example: Display 4.7, page 175

Display 4.7 Inadvertent local variables (1 of 3)

```
// Inadvertent local variables. Shows what happens when you omit & when
you want call-by-reference parameters.


#include <iostream>

void get_numbers(int& input1, int& input2);
//Reads two integers from the keyboard.

void swap_values(int variable1, int variable2);        ← Forgot the &
//Interchanges the values of variable1 and variable2.

void show_results(int output1, int output2);
//Shows the values of variable1 and variable2, in that order.
```

Display 4.7 Inadvertent local variables (2 of 3)

```
int main( )
{
    using namespace std;
    int first_num, second_num;

    get_numbers(first_num, second_num);
    swap_values(first_num, second_num);
    show_results(first_num, second_num);
    return 0;
}
```

Display 4.7 Inadvertent local variables (1 of 3)

```
void swap_values(int variable1, int variable2)
{
   int temp;

   temp = variable1;              Forgot the & here, which
   variable1 = variable2;       Makes these inadvertent local variables
   variable2 = temp;
}

//Uses iostream:
void get_numbers(int& input1, int& input2)
{
   using namespace std;
   cout << "Enter two integers: ";
   cin >> input1
       >> input2;
}

//Uses iostream:
void show_results(int output1, int output2)
{
   using namespace std;
   cout << "In reverse order the numbers are: "
        << output1 << " " << output2 << endl;
}
```

Procedural Abstraction

- A function may call another function.
- Function definition cannot be placed within a function definition, must be placed after main or in another file.
- The situation is exactly the same as if the first call had been in the main function.
- Example: **Display 4.8** page 178

Display 4.8 Function Calling Another Function (1 of 4)

```
//Program to demonstrate a function calling another function.
#include <iostream>

void get_input(int& input1, int& input2);
//Reads two integers from the keyboard.

void swap_values(int& variable1, int& variable2);
//Interchanges the values of variable1 and variable2.

void order(int& n1, int& n2);
//Orders the numbers in the variables n1 and n2
//so that after the function call n1 <= n2.

void give_results(int output1, int output2);
//Outputs the values in output1 and output2.
//Assumes that output1 <= output2
```

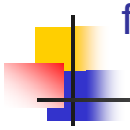
Display 4.8 Function Calling Another Function (1 of 4)

```
int main( )
{
   int first_num, second_num;

   get_input(first_num, second_num);
   order(first_num, second_num);
   give_results(first_num, second_num);
   return 0;
}
```

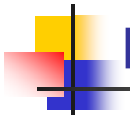
```
//Uses iostream:
void get_input(int& input1, int& input2)
{
   using namespace std;
   cout << "Enter two integers: ";
   cin >> input1 >> input2;
}

void swap_values(int& variable1, int& variable2)
{
   int temp;

   temp = variable1;
   variable1 = variable2;
   variable2 = temp;
}
```

```
void order(int& n1, int& n2)
{
   if (n1 > n2)
      swap_values(n1, n2);
}

//Uses iostream:
void give_results(int output1, int output2)
{
   using namespace std;
   cout << "In increasing order the numbers are: "
        << output1 << " " << output2 << endl;
}
```

Designing, testing and debugging functions

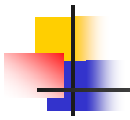


- Programmers typically write functions, not programs
- Every function should be designed, coded and tested as a separate unit from the rest of the program.
- To test a function you need a driver program.
- Every function should be tested in a program in which every other function in that program has already been completely tested and debugged.
- If your function uses another function you may need to write a stub.
- This is a catch 22. You need a framework to develop and test, but the framework must be debugged as well.



Examples of drivers and stubs

- **Display 4.10 Driver Program - page 188**
- **Display 4.11 Program that uses a Stub – page 189**



Summary

- All subtasks in a program can be implemented as functions, either as **void-functions** or **value-returning functions**.
- A formal parameter is a kind of place holder that is filled with a function argument when the function is called.
  - In **call-by-value,** the value of the argument is copied into the parameter. An argument corresponding to a call-by-value parameter will not be changed by the call.
  - In **call-by-reference** the argument must be a variable, and the effect is as if the variable had been substituted for the parameter. An argument corresponding to a call-by-reference parameter may be changed by the call, but this is not necessary. The syntax to make a parameter call-by-reference is to insert an ampersand (&) between the type and the parameter in the function prototype and function header.

- Function prototype comments should be divided into Pre-conditions and Post-conditions. The pre-condition describes the requirements that must be in effect when the function is called, and the post-conditions describe the effect of the call, including any returned value and/or changed arguments.
- Every function should be tested in a completely tested and debugged program.
  - A driver program is a short program that does nothing but test a function.
  - A stub is a simplified function used in place of a function definition that has not yet been tested (perhaps not written) so that the rest of the program can be tested.

Take Another Break

You probably need it by now!

triangle.cpp

```
////////////////////////////////////////////////////////////////////////////
//                          ECC EDP-121 Spring 2001                       //
//                                                                        //
//  Type of Assignment:     Take Home                                     //
//  Problem Number:         5                                             //
//  Author:                 Glenn Mayer                                   //
//  Section Number:         99                                            //
//  Date Assigned:          01/10/01                                      //
//  Program Name:           Triangle Program                              //
//  Textbook Reference:     Handout #5                                    //
//  File Name:              triangle.cpp                                  //
//                                                                        //
//  Purpose of Program:                                                   //
//      This program will read in three floating point numbers and        //
//      will determine if they could form the three sides of a triangle.  //
////////////////////////////////////////////////////////////////////////////

// Include Section
#include <iostream.h>

// Main Program
int main( )
{
    // Variable Declations
    double side1;           // the first side entered by the user
    double side2;           // the second side entered by the user
    double side3;           // the third side entered by the user
    int triangle;           // this is the results of the triangle calculation

    const int good = 1;     // this is the value that goes into the variable
                            // triangle if the three sides make a triangle
    const int bad = 2;      // this is the value that goes into the variable
                            // triangle if the three sides do not make a triangle
    char answer;            // the users answer to the question
                            // "do you want to do it again?"
                            // a 'y' or a 'Y' is YES, anything else is
                            // interpreted as NO
```

```
// Output Identification
cout << "Take Home #5 by Glenn Mayer - "
     << "Triangle Program\n\n";

// Main Program
// describe program's purpose to the user
cout << "This program will read in three floating point numbers and\n";
cout << "will determine if they could form the three sides of a triangle.\n\n";

do      // keep repeating the code as long as the user wants
{
    // get the three sides from the user
    cout << "\nEnter side 1: ";
    cin >> side1;                          // get side 1
    cout << "Enter side 2: ";
    cin >> side2;                          // get side 2
    cout << "Enter side 3: ";
    cin >> side3;                          // get side 3
```

triangle.cpp

```
       // determine if a triangle can be made
       //      Note:   For each side on a triangle, the side must be
       //              shorter that the sum of the opposite two sides
       if(side1 < (side2 + side3) &&   // make sure that side 1 is not too long
          side2 < (side1 + side3) &&   // make sure that side 2 is not too long
          side3 < (side1 + side2))     // make sure that side 3 is not too long
       {
           triangle = good;            // mark this triangle as good
       }
       else
       {
           triangle = bad;             // mark this triangle as bad
       }

       // print the results
       cout << "It is ";               // print the first part of the sentence
       if (triangle == bad)            // if this is a bad triangle
       {
           cout << "NOT ";             // print the word NOT
       }
       cout << "possible to make a triangle with sides of ";   // print rest of sentence
       cout << side1 << ", " << side2 << ", and " << side3 << ".\n";

       // check to see if they want to do it again
       cout << "Do you wish to repeat the calculation? (Y or N) : ";
       cin >> answer;

   } while (answer == 'y' || answer == 'Y');

   // let user know that its all over
   cout << "\n\nEnd Program.\n";

   return 0;
}
```

```
Take Home #5 by Glenn Mayer - Triangle Program

This program will read in three floating point numbers and
will determine if they could form the three sides of a triangle.


Enter side 1: 3
Enter side 2: 5
Enter side 3: 5
It is possible to make a triangle with sides of 3, 5, and 5.
Do you wish to repeat the calculation? (Y or N) : y

Enter side 1: 3
Enter side 2: 4
Enter side 3: 10
It is NOT possible to make a triangle with sides of 3, 4, and 10.
Do you wish to repeat the calculation? (Y or N) : Y

Enter side 1: 5
Enter side 2: 10
Enter side 3: 5
It is NOT possible to make a triangle with sides of 5, 10, and 5.
Do you wish to repeat the calculation? (Y or N) : n


End Program.
Press any key to continue
```