

Creating and calling your own functions from main()



Outline chap 6 Read study sections 6.1-6.5,6.9-6.11

Objectives

- 1.Modularity
- 2.Programmer-Defined Functions
- 3.Parameter passing
- 4.Numerical Techniques with C++ (Roots of Polynomials and Numerical integration)

Be sure to run you tube video(s) in the online syllabus

Outline

Objectives topics for you to master

1. Modularity
2. Programmer-Defined Functions
3. Parameter Passing
4. Problem Solving Applied: Calculating a Center of Gravity
5. Random Numbers
6. Numerical Technique: Roots of Polynomials*
7. Problem Solving Applied: System Stability*
8. Numerical Technique: Integration*

More on Objectives

Develop problem solutions in C++ containing:

Functions from the standard C++ library.

Programmer-defined functions.

Functions that generate random numbers

Simulation Techniques.

Techniques for finding real roots to polynomials.

Numerical integration techniques.

Modularity

A problem solution contains numerous functions. Not just `main()` which must be included

- `main()` is a programmer defined function.
- `main()` often references functions defined in one of the standard libraries.
- `main()` can also call other programmer defined functions.

These programmer defined Functions, or **modules**, are independent statement blocks outside of `main()` that are written to perform a specialized task and are called to do their task from `main()`.

Modules

A C++ source program can be thought of as an ordered collection of executable tasks:

- input data
- analyze data
- output results

In C++ these tasks can be performed using programmer defined functions and types.

We can test the modules separately and can be used many times.

Program lengths can be shortened by modules.

Details can be hidden in modules so you don't always have to know exactly how the module works. E.G. you need the squareroot and call `sqrt()` from `cmath` but you are not concerned how the function (module) did the calculation.

The latter is called abstraction (ie hide details) and modules become “black boxes”.

Modular Problem Solutions

Complex problems can be broken down into sub tasks performed on and/or by objects, making it easier to implement in C++.

Modular, object-oriented programs have significant advantages over writing the entire solution in `main()`:

- Multiple programmers
- Testing/Debugging/Maintaining
- Reduce duplication of code

- We have been using pre-defined functions from libraries (`cmath`)
- Eg. `sin()` `log()` etc
- Now we can create our own library of functions we have built with modules or user defined functions. The user here is us the programmer.

Functions or modules

Can be defined to:

- 1. return a single value to the calling function.
 - Eg. $\sin(x)$ returns a value but does not change the value of x in `main()`!
- 2. perform a data independent task.
 - A function to print a report!
- 3. Modify data in *calling context* via the parameter list (*pass by reference.*) ie can change values of variables in `main`
- More Examples to follow but the basic idea is the functioned called will change the value of variables in `main()`.

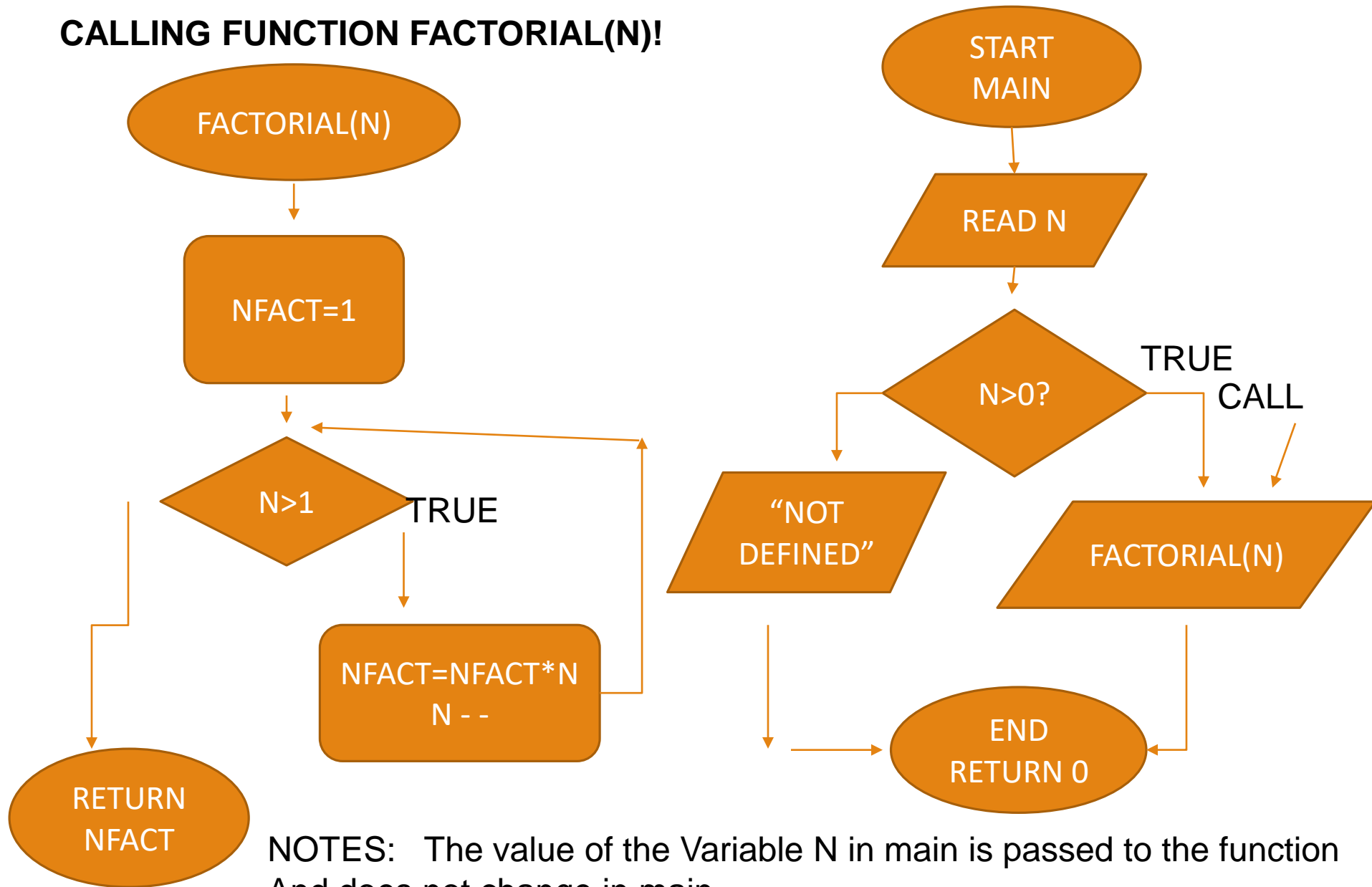
Value Returning Functions

A value returning function *returns* a single value to the calling program.

The function header declares the type of value to be returned.

A return statement is *required* in the statement block.

CALLING FUNCTION FACTORIAL(N)!



NOTES: The value of the Variable N in main is passed to the function And does not change in main.

VARIABLE NFACT IN THE FUNCTION IS NOT KNOWN TO MAIN ONLY THE RETURNED VALUE GOES BACK TO MAIN!

Example – Calling the Factorial Function from main()-value returning function

```
//return name argument type
int factorial(int n); //function prototype(defines its type)
int main() {
    int num;
    cin >> num;
    if (num >= 0) // The call argument(matches parameter)
        cout << num << "! is " << factorial(num) << endl;
        //n is the argument of the
        //factorial function call
    else
        cout << "not defined for negative numbers"
            << endl;
    return 0;
} //end main
```

HERE IS WHERE THE FUNCTION IS PLACED IE AFTER MAIN() (sometimes before)

Example “value returning” - Factorial

In math:

- $n! = n*(n-1)*(n-2)*...*1$
- n is a positive integer
- $0!$ is 1 by definition

```
//function definition: n! = n*(n-1)*(n-2)*...*1
//                                0! is 1 by definition
//Function factorial returns n!
//Function factorial assumes n is non-negative int
```

```
int factorial(int n) // <-function header note: num->n
    //function header has the function name and shows
    what is returned here an int! and “n” is named a
    “parameter” which matches the “argument” of the
    The argument in the call
    {
        int nfact = 1;
        while (n>1) {
            nfact = nfact * n;
            n--;
        } //end while block
        return nfact;
    } //end factorial WE CONSIDER n=3 & play computer!!!
```

Function Prototype in the last example

In C++, identifiers must be defined ***before*** they may be referenced. I.e. function prototype!

Since the main function is the entry point for the application, we like to have it appear near the beginning of the program.

i.e. “function prototypes” put just before `int main(){..`

A “function prototype” provides sufficient information for the compiler to process references to the function.

- The definition may then be provided later in the code.

Function Prototype examples for value returning functions

Syntax:

```
return-type function_name ( [parameter_list] );
```

Examples

Valid References

Invalid References

```
double sinc(double);  
double sinc(double x);
```

```
Assume double  
y(-5);  
sinc(y)  
sinc(1.5)  
sinc(10)  
sinc(0.0)
```

```
sinc("hello")  
sinc('0')  
sinc(cout)
```

```
Double celsiusToFahr(double celsius)    100    '100'(string!)
```

This is used in the next program note variable name in function Prototype is optional just type is ok!

Hand in HW #13 25 pts.

What will this **program segment** do?

show table of all variables and detailed output as usual!

```
a=7; b=10;
```

```
c= pyth(a,b);
```

```
cout << "The mystery result = "<< c<<".";
```

```
return (0);
```

```
}
```

```
float pyth (float x, float y)
```

```
float c1,c2;
```

```
{
```

```
c1 = pow(x,2) +pow (y,2);
```

```
c2 = sqrt(c1);
```

```
return (c2);
```

```
}
```

```

/* Chapter6_1.cpp An example using a call to a function that returns a value! One argument to match one parameter */
#include<iostream> //Required for cin, cerr
#include<fstream> //Required for ifstream, ofstream
#include<string> //Required for string, c_str()
using namespace std;
double celsiusToFahr(double celsius); //Programmer defined function. Function prototype
/* Program chapter6_1 This program reads temperatures in degrees Celsius */
/* from an input file, calls a conversion function and writes converted temperatures to an output file. */
int main()
{
ifstream fin; //Declare variables
ofstream fout;
string filename;
double cels, fahr;;
//Open files.
cout << "Enter name of input file\n ";
cin >> filename;
fin.open(filename.c_str());
if (fin.fail())
{
cerr << "Could not open the file " << filename << endl;
exit(1);
}
fout.open("CelsiusToFahr.dat");
fin >> cels;
while (!fin.eof()) //while not end of file
{
fahr = celsiusToFahr(cels); //Function call. //Convert temperature and write to file.
cout << cels << " " << fahr << endl; // echo to screen to check values
fout << "Celsius = "<<cels<< " Equivalent Farenheit=" <<fahr << endl;
fin >> cels;
}
fin.close();
fout.close();
return 0;
} // end of main()
/*-----*/
/* This function performs a conversion from degrees Celsius to degrees Fahrenheit. */
/* Precondition: celsius holds a temperature in degrees Celsius */
/* Postcondition: returns degrees Fahrenheit */
double celsiusToFahr(double celsius) //Function header.
{
double temp; //Declare local variables
temp = (9.0 / 5.0)*celsius + 32.0; //Convert from degrees celsius to degrees Fahrenheit.
return temp;
} // end of the function

```



```
/* An example using a call to a function that returns a value! One argument to match one parameter */
```

```
#include<iostream> //Required for cin, cerr
```

```
#include<fstream> //Required for ifstream, ofstream
```

```
#include<string> //Required for string, c_str()
```

```
using namespace std;
```

```
double celsiusToFahr(double celsius); //Programmer defined function. Function prototype
```

```
/* Program chapter6_1 This program reads temperatures in degrees Celsius */
```

```
/* from an input file, calls a conversion function and writes converted temperatures to an output file. */
```

```
int main()
```

```
{
```

```
    ifstream fin; //Declare objects
```

```
    ofstream fout;
```

```
    string filename;
```

```
    double cels, fahr;;
```

```
    //Open input file.
```

```
    cout << "Enter name of input file\n ";
```

```
    cin >> filename;
```

```
    fin.open(filename.c_str());
```

```
    if (fin.fail())
```

```
    {
```

```
        cerr << "Could not open the file " << filename << endl;
```

```
        exit(1);
```

```
    }
```

```

// open output file
fout.open("CelsiusToFahr.dat");
// get first input data
fin >> cels;
while (!fin.eof()) //while not end of file
{
    fahr = celsiusToFahr(cels); //Function call. //Convert temperature to write to file.
    cout << cels << " " << fahr << endl; // first echo to screen to check values
    fout << "Celsius = "<<cels<< " Equivalent Fahrenheit=" <<fahr << endl;
    fin >> cels; // get next value continue in while to eof()
}
fin.close();
fout.close();
return 0;
} // end of main()
/*-----*/
/* This function performs a conversion from degrees Celsius to degrees Fahrenheit. */
/* Precondition: celsius holds a temperature in degrees Celsius */
/* Postcondition: returns degrees Fahrenheit */
double celsiusToFahr(double celsius) //Function header.
{
    double temp; //Declare local variables (not known to main)
    temp = (9.0 / 5.0)*celsius + 32.0; //Convert from celsius to Fahrenheit.
    return temp; // sent the Fahrenheit value for the received Celsius to main()
} // end of the function

```

HAND IN LABORATORY TASK: LAB #21

1. PRACTICE FIRST create data file below and run the previous program (**do not hand in..**just practice). run the last program to follow its logic. Consider the these Celsius temperatures for your input file

-100
-32
-17.777
0
20
30
100

2. NOW Create the opposite function ie. **Read Fahrenheit temperatures from the file and output Celsius.** create an appropriate data file from -10F to +300F (ABOUT 8 OR MORE VALUES) and hand in the program, input and output and screen output info in the lab (that is Hand in All!). **Be sure to use 212F and 32 F to check** your expected output is being calculated properly. ie 100C and 0C is expected if not your formula is bad!

Function Definitions(for return a value and no value returned)

Syntax:

```
return-type function_name ( [parameter_list] ) {  
    // declarations and statements  
    return expression!  
}
```

RETURNS SOME VALUE

```
Void function_name ( [parameter_list] ) {  
    // declarations and statements  
}
```

No value returned does
Something useful-
Void function

```
int factorial(int n)  
{  
    int nfactorial = 1;  
    while (n>1) { nfactorial = nfactorial * n;  
                n--;  
    } //end while  
    return nfactorial;  
}
```

Return example

```
void drawBlock(ostream& out, int size) {  
    for (int height = 0; height < size; height++) {  
        for (int width = 0; width < size; width++)  
            out << "*";  
        out << endl;  
    }
```

NO VALUE RETURNED
BUT DOES SOMETHING

```
} // NOTE: void with one parameter('size')  
// i.e. gets from main value of size
```

NOTE NO RETURN
STATEMENT! Void!

Void Functions

A void function declares `void` as a return type.

A return statement is optional.

If a return statement is used, it has the following form

- **return;**
- A **void function** performs a task, and then control returns back to the caller--but, it does not return a value.
- Call has to **stand alone** and cannot be in an expression. For example we use library functions in expressions (`sin()`, `sqrt()`) etc. but not void ones!
- NOTE Only two kinds of functions for C++: value-returning functions and void functions. Both value-returning functions and void functions receive values through their **parameter lists**. A **value-returning function can only return one value to the call.**

Examples void functions see you tube tutorial for excellent presentation:

<https://www.youtube.com/watch?v=vXxoAzIkU7k> and others follow!

```
void PrintIntersecting() { // example of void function no parameters filled from main!
```

```
    cout << "    $ " << endl;
```

```
    cout << "    $    $ " << endl;
```

```
    cout << "    $        $ " << endl; }
```

```
void PrintGrade(string name, float score); // more than one parameter passed to it!
```

```
Example PrintGrade("Irving", (90+87.5)/2);    Or use as
```

```
name = "Irving"; overallScore = (90+87.5)/2;
```

```
PrintGrade(name, overallScore);
```

We use & to connect an address so the variable which will change in main.

And we can write void functions to do that also like

```
void UpdateBalance(float amount, float & balance) Or void UpdateBalance(float, float&);
```

```
// Purpose: Update the balance of a bank account by incorporating the current transaction
```

```
// Precondition: the value of the amount is assigned and sent to the void function
```

```
// Postcondition: the value of the balance is updated in main!
```

```
    newvalue= oldvalue + bonus;
```

```
    updateBalance(newvalue, balance) // call in main
```

```
UpdateBalance(float amount, float & balance) /header
```

```
{ balance = balance + amount; // value in "newvalue is passed to "amount"
```

```
    return; // address of balance is passed and will change after action.
```

```
} // return is optional since we are not returning anything!
```

Parameter Passing (2 types)

1. Pass by Value
2. Pass by reference (address)

Basic idea

When a function is called from main and we use parameters there are two ways main() passes and receives information.'

1. Value Parameter: Main() uses variables we call **arguments (I like this)** or **reference parameters** and calls the function with these variables and the function has corresponding variables known **as parameters** (independent of main() ie. Main does not know them) and the function uses the value passed to the parameters to calculate a return value to main(). These receiving parameters are called **Value Parameters**.

2. Reference Parameter: The parameters used by main() in the call to the function are known to the function (they know the **address in memory** and thus the identity of them) After the function does its thing the **variables in main() will change!**

:

We Focus on Parameter Passing (2 types) continued

1. Pass by Value

Value Parameters: expanded info.

Value Parameter: receives a copy of the content of corresponding **arguments** in the `main()` call to the function. They have in the memory assigned to the function (local memory for the function) their own copy of the values sent (independent of the memory used by `main()`).

Thus, when the function executes these parameter it does the calculations in their own memory space and returns a single value!

They can accept expressions, constants, or variables from the function call parameters.

Here Even Void Functions can use Value or reference Parameters

Example Prototype `void Name(dataType variable, dataType variable)`

no value is returned but the void function will use the values to manipulate or calculate something. As we saw in the examples.

Parameter Passing (2 types) continued

2. Pass by reference (address)

More on arguments or reference Parameters:

Parameters(s) in the function receive the memory address for the corresponding arguments in the main() call to the function and thus the function Via the parameters know the identity of the arguments when the function executes, since it receives the addresss where info is stored it can change those values at the address known to main(), thus this function call is

sharing the same memory space with Main()

This way **more than one value** can change and be passed back to main()

Best used when you want the function call to change a value of a variable(s) in main() and also useful when you want more than one value changed since So you can get more than One if you pass multiple addresses :

Possibly a time saver in some cases

Only variables can be arguments in the call, not constants or expressions.

NOTE: Sometimes the arguments in the function call are named **reference parameters** and the name “parameters” is used for the function itself and may be also called the **local variables** of the function.

REVIEW Function Terminology

Function Prototype

- describes how a function is called : in the beginning of program

E.G. `double celsiusToFahr(double celsius);` //Programmer defined function. Function prototype
`double bigbang(x,y,z)`

Function Call

- references and invokes (i.e. causes execution of) a function
- `cout <<bigbang(sqrt(f),q,u/9);` `sqrt(f),q,u/9` are the arguments
- `fahr = celsiusToFahr(cels);` `cels` is an argument

Function Arguments

- Expressions provided as parameters to function call

Function Definition The actual functions at the end or sometimes in the Beginning of a program.

- function header
- statement block

E.g.,
“n”

```
int factorial(int n) // <-function header with formal parameter
{
    int nfactorial = 1; // nfactorial is internal parameter
    while (n>1) {
        nfactorial = nfactorial * n; BLOCK
        n--;
    } //end while block
    return nfactorial;
}
```

Formal Parameters

- Variables defined in the function header and used in definition to access the function's parameters

Formal parameters in the function must agree with arguments in order, number and data type. Arguments are the variables etc in the call.

Example: pass by value call we have seen.

```
int fact(int);    \*prototype to let the main()
program know the name of the function and its return
and needed argument/parameter types*/
```

```
int main()
{
    int n, factorial; //1 <- sequence of execution
    cin >> n; //2
    if(n>=0)
    {
        factorial = fact(n); //3
        cout << n <<"! is " << factorial << endl; //7
    } //end if
    return 0;
} //end main
```

```
int fact(int num) //4 & Function Definition
int nfact = 1; //5
{
    while(num>1)
    {
        nfact = nfact*num;
        num--;
    } // end while
    return(nfact); //6
} //end fact
```

```

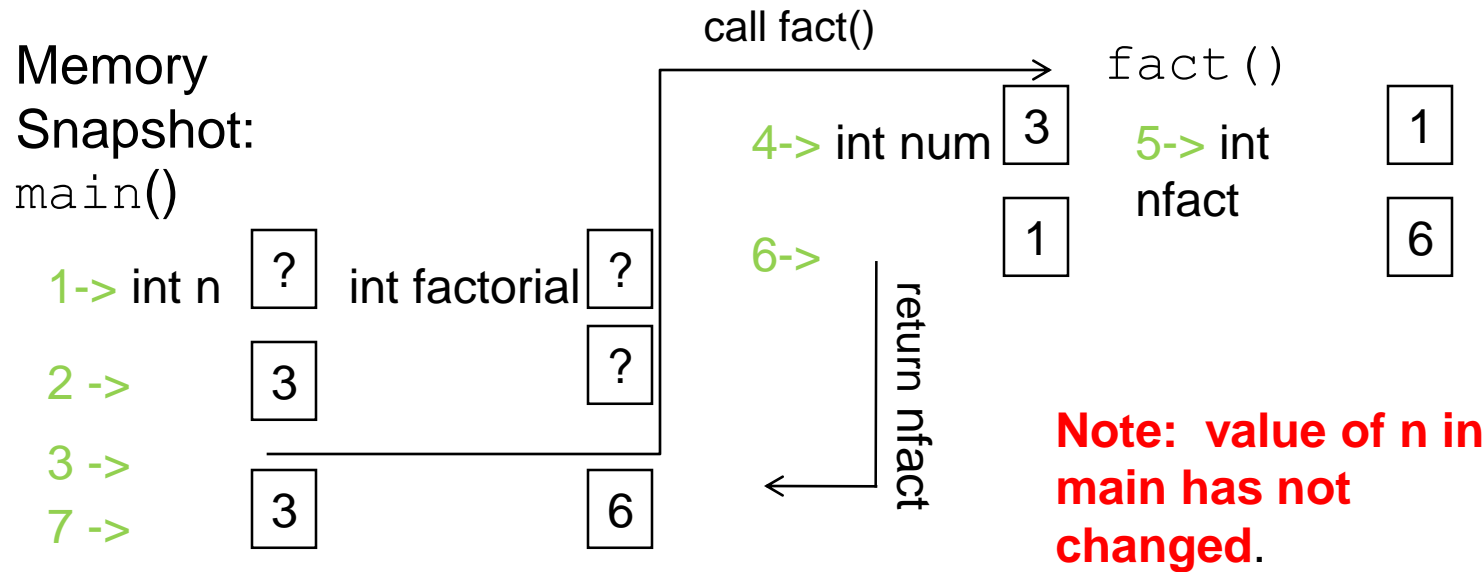
int fact(int);
int main()
{
    int n, factorial; //1
    cin >> n; //2
    if(n>=0)
    {
        factorial = fact(n); //3
        cout << n <<"! is "
        << factorial << endl; //7
    } //end if
return 0;
} //end main

```

```

//Function Definition
int fact(int num) //4
{
    int nfact = 1; //5
    while(num>1)
    {
        nfact = nfact*num;
        num--;
    }
return(nfact); //6
} //end fact

```



```

/* Program chapter6_3    pass by value another example */
/* This program prints 21 values of sinc(x) in the interval [a,b] */
#include<iostream> //Required for cin, cout
#include<cmath> //Required for sin().
using namespace std;
double sinc(double x); //Function Prototype
int main()
{ double a, b, x_incr, new_x; // Declare objects in main()
  cout << "Enter endpoints a and b (a<b): \n";
  cin >> a >> b;
  x_incr = (b- a)/20; // increment between the points
  cout.setf(ios::fixed); cout.precision(6); // Set Formats
  cout << "x and sinc(x) \n"; // table of sinc(x) values call to sinc()
  for (int k=0; k<=20; k++)
  { new_x = a + k*x_incr;
    cout << new_x << " " << sinc(new_x) << endl; // call to sinc() here!
  }
  return 0; // Exit program.
} // end main

double sinc(double x) /*This function evaluates the sinc function*/
{  if (fabs(x) < 0.0001) // x here picks up the values of x in main()
    { return 1.0; // but this x has its own memory and does not
    } // interfere with the values of x in main()
  else
  { return sin(x)/x;
  }
} // end of function

```

```

/* Program chapter6_3.cpp    pass by value another example */
/* This program prints 21 values of sinc(x) in the interval [a,b]
*/
#include<iostream> //Required for cin, cout
#include<cmath> //Required for sin().
using namespace std;
double sinc(double x); //Function Prototype
int main()
{ double a, b, x_incr, new_x; // Declare objects in main()
  cout << "Enter endpoints a and b (a<b): \n";
  cin >> a >> b;
  x_incr = (b- a)/20; // increment between the points
  cout.setf(ios::fixed);  cout.precision(6); // Set Formats
  cout << "x and sinc(x) \n"; // table of sinc(x) values call to
sinc()
  for (int k=0; k<=20; k++)
  { new_x = a + k*x_incr;
    cout << new_x << " " << sinc(new_x) << endl; // call to
sinc() here!
  }
  return 0; // Exit program.
} // end main

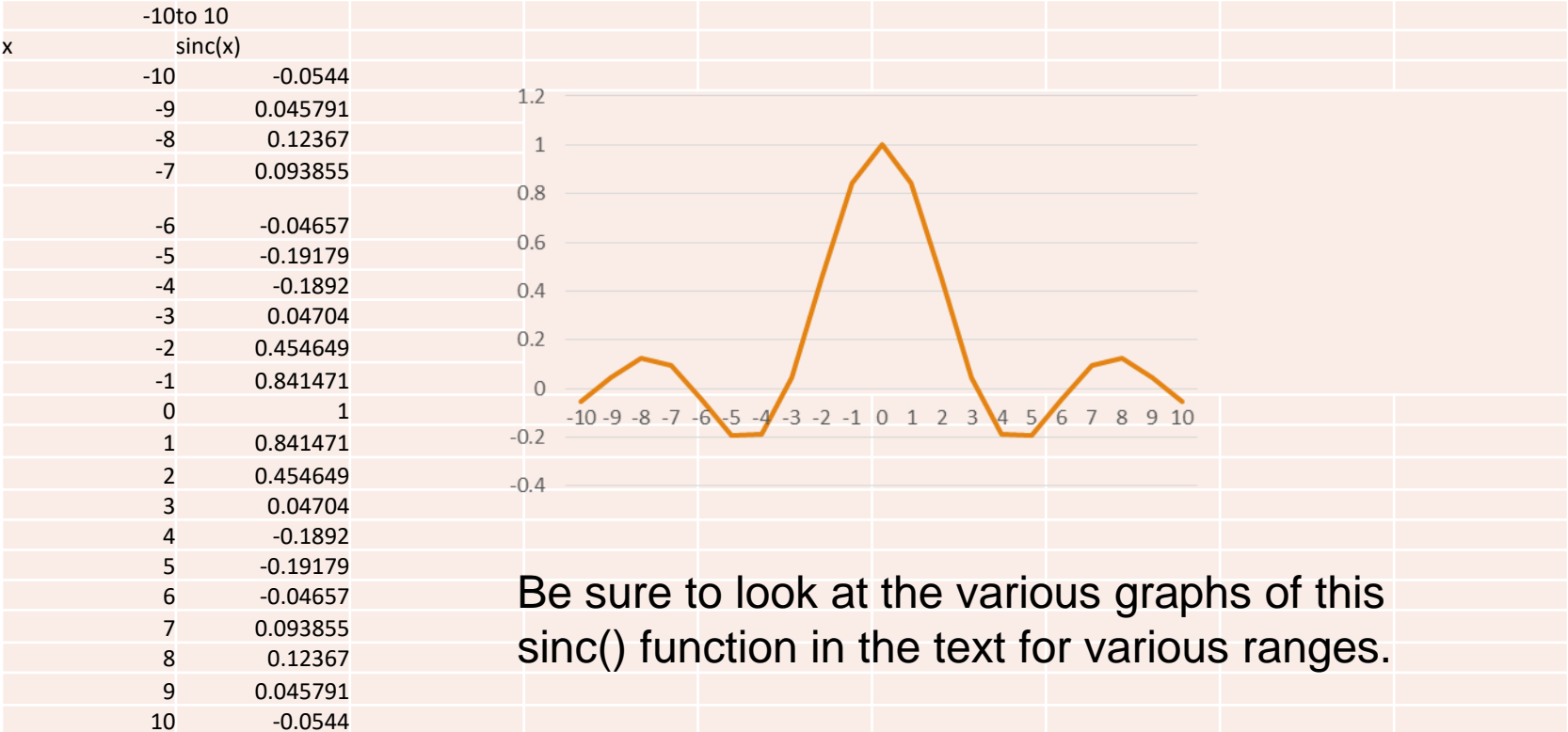
```

```
/*This function evaluates the sinc function= sin(x)/x */  
  
double sinc(double x)  
{  
    // function block  
    // x here(parameter) picks up the  
    // values of new_x (argument) in main()  
    // but this "x" has its own memory and does not  
    // interfere with the values of new_x in main()  
    // this x is local to the function  
    if (fabs(x) < 0.0001  
        {  
            return 1.0;  
        }  
    else  
    {  
        return sin(x)/x;  
    }  
} // end of function
```

So you would like a graph of the output?. Since you have two columns of numbers, just making a copy of the output in the usually way is almost impossible to transfer to an excel spreadsheet where making graphs is easy so long as you, As in this case, copy two columns of numbers.

SIMPLE solution is to output one of the variables at a time to create single Columns which are easy to transfer to the excel spreadsheet and create A graph.. The professor will demonstrate such with this last program

Once a graph and the columns of data are produced a paper copy can be made And attached to the hand-in when asked for. Looking like the following.



HAND IN LABORATORY TASK: LAB #22 create ROBBINS(x)

1. RUN THE PREVIOUS PROGRAM FROM -20 TO +20 AND PRACTICE DOING A GRAPH OF THE OUTPUT
DO NOT HAND IN!

2. **Part A.** MODIFY THE PREVIOUS PROGRAM TO CALCULATE A NEW FUNCTION with more values in the same range as above.
 $ROBBINS(X) = (10 * \sin(x) / x) + \cos(x)$.

OUTPUT X (SAME RANGE AS ABOVE **but 100 values or more**) AND THE Corresponding ROBBINS() VALUES.
COPY AS USUAL TO THE final PROGRAM TO HAND IN.

Part B.

Use the graph technique discussed and output only the value of the function ROBBINS(X) (ie the y axis values) and transfer The data to an Excel spreadsheet and plot the data and attach to the data sheet.


Print up the this sheet of data and graph and **hand in** with a copy of the modified Program and its output data (PART 2A).

Extra credit have the correct values on the x axis!

***Class exercise:


Consider the following function

```
int positive(double a, double b, double c)
{
    int count;
    count =0;
    if (a>0)
    {
        ++count;
    }
    if (b>0)
    {
        ++count;
    }
    if (c>0)
    {
        ++count;
    }
    return count;
}
```



main() has the call

```
x= 25; //arguments
total = positive(x, sqrt(x), x-30)
total =?
```



Show memory of **parameters**
And arguments?

Pass by Reference: Details

Pass by reference allows modification of a function argument.

Must append an **&** to the parameter data type in both the function **prototype** and function **header**

```
void getDate(int& day, int& mo, int& year)
```

Formal parameter(in the function) becomes an *alias* for the argument.

- *The argument must be a variable of a compatible type.*

So Any changes to the formal parameter(variable) in the function directly change the value of the argument(variable) in the main program.

In other words, the address of the variables in main are passed so the function is working directly with the same variables in main and memory will change for those.

Example: call by reference

```
#include <iostream>
using namespace std;
void swap(double&, double&); //function prototype
int main()
{
    double x=5, y=10;
    swap(x,y); //function call; x y are arguments
    cout >> "x = " << x << ", " << " y= " << y << endl;
    return 0;
} //end main

//Function swap interchanges the values of two variables
void swap(double& x, double& y)
{
    double temp; //local variable temp
    temp = x; //x & y effectively are the same as main
    x=y;
    y=temp;
    return; //optional return statement
} //end swap
```

WHAT IS x and y after the call?

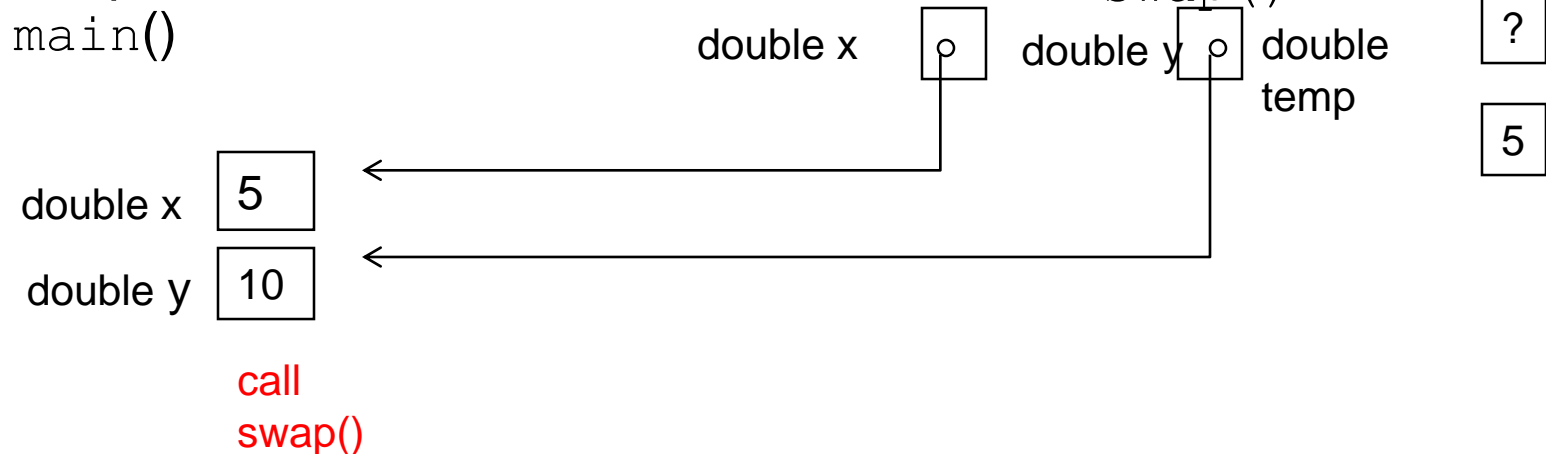
```

Program Trace: pass by reference //Function Definition
#include <iostream>
using namespace std;
void swap(double&, double&); //prototype
int main()
{
    double x=5, y=10;
    swap(x,y); //x y are arguments
    cout << "x = " << x << ', '
        << " y= " << y << endl;
    return 0;
} //end main

void swap(double& x, double& y)
{
    double temp;//
    temp = x;
    x=y;
    y=temp;
    return; //optional
} //end swap

```

Memory Snapshot:



```

Program Trace: pass by reference //Function Definition
#include <iostream>
using namespace std;
void swap(double&, double&); //prototype
int main()
{
    double x=5, y=10;
    swap(x,y); //x y are arguments
    cout << "x = " << x << ', '
        << " y= " << y << endl;
    return 0;
} //end main

void swap(double& x, double& y)
{
    double temp;//
    temp = x;
    x=y;
    y=temp;
    return; //optional
} //end swap

```

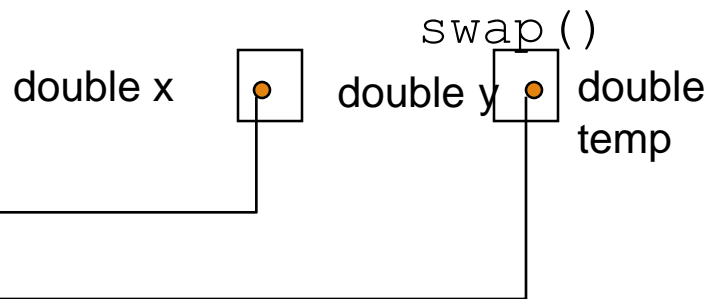
Memory

Snapshot:

main()

double x 10

double y 10



?

5

```

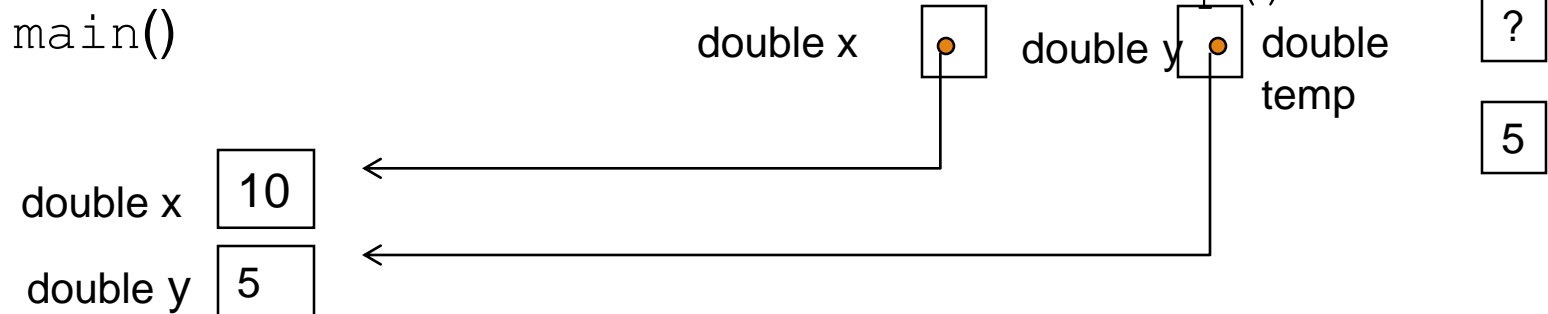
Program Trace: pass by reference //Function Definition
#include <iostream>
using namespace std;
void swap(double&, double&); //prototype
int main()
{
    double x=5, y=10;
    swap(x,y); //x y are arguments
    cout << "x = " << x << ', '
        << " y= " << y << endl;
    return 0;
} //end main

void swap(double& x, double& y)
{
    double temp;//
    temp = x;
    x=y;
    y=temp;
    return; //optional
} //end swap

```

Memory

Snapshot:



return;

Note: variable temp is only know to the function, we say its local!
 While the memory place of x and y is known in main() and swap()

Program Trace: pass by reference

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

void swap(double&, double&); //prototype
int main()
{
    double x=5, y=10;
    swap(x,y); //x y are arguments
    cout << "x = " << x << ', '
         << " y= " << y << endl;
    return 0;
} //end main
```

```
//Function Definition
void swap(double& x, double& y)
{
    double temp;//
    temp = x;
    x=y;
    y=temp;
    return; //optional
} //end swap
```

Memory

Snapshot:

```
main()
  double x 

|    |
|----|
| 10 |
|----|


  double y 

|   |
|---|
| 5 |
|---|


```

Arguments have been
modified
SEE SIMILAR PROGRAM
AND ANALYSIS IN TEXT!

CLASS EXERCISE. CONSIDER swapint() . → IF THE CALL WAS from main() that had each of the following explain what do you think happens?

1. int x=1, y =4;....swapint(x,y)
2. swapint(10,4)
3. swapint(x,y+5)
4. double x =1.5, y=3.2;.....swapint (x,y)
5. What is the output of the following program

```
void swapint(int& x, int& y)
{
    double temp;//
    temp = x;
    x=y;
    y=temp;
    return; //optional
} //end swap
```

```
#include <iostream>
Using namespace std;
Void funincpp(int first, int& second);
```

```
int main()
{
    int n1(0), n2(0);
    funincpp(n1,n2);
    cout<<n1<<endl<<n2<<endl;
    return 0;
}
```

```
Void funincpp(int first, int& second)
{
    first++;
    second += 2;
    return;
}
```

Scope and Storage Class

Scope refers to the portion of the program in which it is valid to reference a function or a variable. It is the part of the program that an object is visible or accessible. We can thus for example put a function before `main()` which effects when objects are accessible.

In particular, we define:

Local scope - a local variable is defined within a function or a block and can be accessed only within the function or block that defines it. This “local” nature has been mentioned in our previous examples.

Global scope (also called file scope) - a global variable is defined outside the **main** function and can be accessed by any function within the program file.

Example are the “constants” you have place **before** `main()`, they will be known to all functions!

Storage Class – 4 Types

scope is related to the **storage class** which is basically the lifetime of a variable

automatic - key word **auto** - default for local variables

- Memory set aside for local variables is not reserved when the block in which the local variable was defined is exited. I.e. **Local variables**
- **They are gone to the rest of the program after the function finishes.**

external - key word **extern** - default for global variables

- Memory is reserved for a global variable throughout the execution life of the program. i.e. **global variables** (all functions can access)

static - key word **static**

- Requests that memory for a local variable be reserved throughout the execution life of the program. The value stays after functions finish.

register - key word **register**

- Requests that a variable should be placed in a high speed memory register rather than regular memory. Not always possible even when asked.

Illustrating scope and storage class->

Class?

Global object?

Local objects?

Note use of "x"? and "count"

Any Static?

```
#include <iostream>
using namespace std;
...
int count;    // note outside main()
int main()
{   int x, y ;
    ..... All functions are called
}
int calc( int a, int b)
{   int x;
    count +=x; ...
}
void checkit (int sum)
{   count += sum;
}

void mystatic()
{   int x(0);
    static int csi(0);
    x++;
    csi++;
    cout << x << ',' << csi;
    return;
}
```

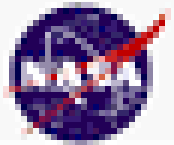
H.W #14

Exam Practice chap 6 #1-4,8-14 39 PTS 3 EACH TOTAL PTS= 63 TO 75
EXTRA CREDIT 15-17 (SEE "CLASS" DEFINITION IN CHAP 2 AND 3 AND SEC 6.7)
12PTS 4 EACH be sure to show memory snapshots for 15 to 17

What is the output?: MEMORY: Use tables for all variable values & Specify variable (scopes) types!!

#include <iostream> for 30 pts (BE SURE TO USE TABLE AND OUTPUT COLUMNS)

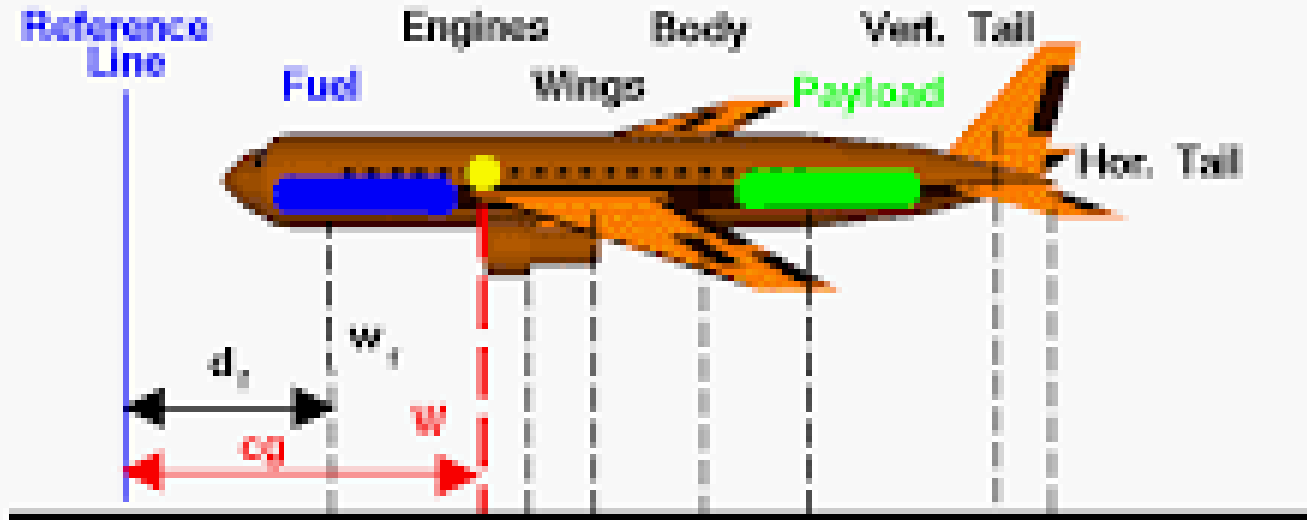
```
using namespace std;
int X(6), C(3);
int TestMe(int &Y, int Z);
int main (void)
{
    int A, B, W;
    A = 5;      B = 2;
    X = 1;
    W = TestMe(A, B);    // Last two output lines
    cout << "A = " << A << " B = " << B << endl;
    cout << "X = " << X << " C = " << C << endl;
    return 0;
}
int TestMe (int &Y, int Z)
{
    int C;
    cout << "Y = " << Y << " Z = " << Z << endl;
    cout << "C = " << C << endl;
    cout << "X = " << X << endl;
    C = 4;      Z = 7;
    X = C + Z;
    Y = X + Z;
    cout << "X = " << X << " Z = " << Z << endl;
    cout << "Y = " << Y << endl;
    return Z;
}
```



Center of Gravity - cg

Aircraft Application

Glenn
Research
Center



Each component has some weight w_i located some distance d_i from reference line.
Distance cg times the weight W equals the sum of the component distance times weight.

$$cg W = d_1 w_1 + d_2 w_2 + d_3 w_3 + d_4 w_4 + \dots$$

$$cg W = \sum_i (w_i d_i)$$

$$cg = \frac{\sum w_i d_i}{W}$$

Component distance X weight = moment

Sum of moments
Text Units inch-lbs

Text problem the reference line is in the nose!

Problem Solving Applied: Calculating the Center of Gravity

1. PROBLEM STATEMENT

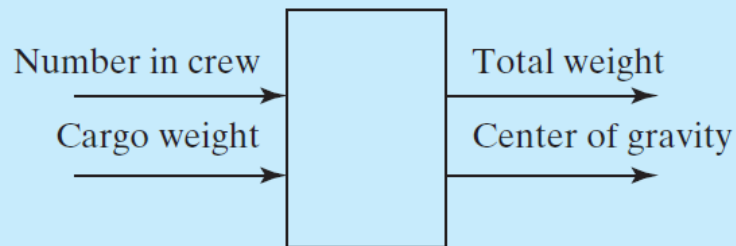
Determine the total weight and center of gravity of an aircraft, based on the number of crew members (a maximum of two is allowed) and the weight of the cargo (a maximum of 5,000 pounds is allowed). To compute the center of gravity, the program will take each weight and multiply it by its distance from the nose of the airplane. These products, called moments, are added together, and the sum is divided by the total weight to give us the center of gravity. The empty weight of the aircraft is known to be 9,021 pounds, and its empty center of gravity is 305 inches from the nose of the aircraft. Thus, the empty moment is 2,751,405 inch-pounds. The aircraft can hold a maximum of 540 gallons of fuel. For simplicity, we will assume that the tank is full at the time of takeoff and, with a fuel weight of 6.7 pounds per gallon, the fuel moment is known to be 1169167.3 inch-pounds.

Here we know the moments of the empty weight and fuel tank
We need the total weight, W , and moments of cargo and crew.
To solve $cg = \sum wd_i / W$ and note cg will be in inches.

Problem Solving Applied: Calculating the Center of Gravity

2. INPUT/OUTPUT DESCRIPTION

The I/O diagram illustrates that the inputs to this program are the number of crew members (one to two) and the weight of the cargo. The output is a report of the total weight of the aircraft and the center of gravity. Built in data types `int` and `double` will be used for all data objects.



Problem Solving Applied: Calculating the Center of Gravity

3. HAND EXAMPLE

Suppose two crew members board the aircraft with a total cargo weight of 100 pounds. Assuming an average weight of 160 pounds per person, we calculate the crew moment as follows:

Number of crew members * average weight per person *
distance from crew to nose of aircraft.

The cargo moment is calculated as follows:

cargo weight * distance from cargo bay to nose of aircraft.

Problem Solving Applied: Calculating the Center of Gravity

The aircraft manual gives the distance in inches from nose of aircraft to the crew seats as 120 inches and the distance from the nose of the aircraft to the cargo bay as 345 inches. Thus, the crew moment is

$$2 * 160 * 120 = 38400 \text{ inch-pounds.}$$

The cargo moment is

$$100 * 345 = 34500 \text{ inch-pounds.}$$

The total weight of the aircraft is

crew weight + cargo weight + fuel weight + weight of the empty aircraft, or

$$320 + 100 + 3618 + 9021 = 13059 \text{ pounds.}$$

The center of gravity is the sum of the moments divided by the total weight:

$$\begin{aligned} &= (38400 + 34500 + 2751405 + 1169167.3) \text{ inch-pounds} / 13059 \text{ pounds} \\ &= 1438172.3 \text{ inch-pounds} / 13059 \text{ pounds} = 305.802 \text{ inches.} \end{aligned}$$

Problem Solving Applied: Calculating the Center of Gravity

4. ALGORITHM DEVELOPMENT

We first develop the decomposition outline because it divides the solution into a series of sequential steps:

Decomposition Outline

1. Read the number of crew members and the cargo weight.
2. Calculate total weight.
3. Calculate the center of gravity.

NOTE: Distances (moment arms) are all given values and
Can be constants in our program for this example

Several approaches to this problem but

**THE TEXT VERSION OF THE PROGRAM IS PRESENTED NEXT WHICH YOU
CAN USE FOR LAB#23 6.4. CPP**

```

/chapter 6_4.cpp *center of gravity and total weight*/
#include<iostream> //Required for cin, cout
using namespace std;
const double PERSON_WT(160.0); //Average weight/person
const double FUEL_MOMENT(1169167.3); //Fuel tank moment
const double EMPTY_WT(9021.0); //Standard empty weight
const double EMPTY_MOMENT(2751405.0); //empty moment
const double FUEL_WT(3618.0); //Full fuel weight
const double CARGO_DIST(345.0); // weight
const double CREW_DIST(120.0); //weight
double CargoMoment(double); //function prototypes here
double CrewMoment(int);
void GetData(int&, double&);
int main()
{ int crew; //number of crew on board (1 or 2)
  double cargo; //weight of baggage, pounds
  double total_weight, center_of_gravity;
  cout.setf(ios::fixed); //Set format flags.
  cout.setf(ios::showpoint);
  cout.precision(1);
  GetData(crew, cargo); // call function
  total_weight = EMPTY_WT + crew*PERSON_WT + cargo+ FUEL_WT;
  center_of_gravity = (CargoMoment(cargo) + CrewMoment(crew)+ FUEL_MOMENT + EMPTY_MOMENT)/total_weight;
  cout << endl << "The total weight is " << total_weight<< " pounds. \n"
  << "The center of gravity is " << center_of_gravity<< " inches from the nose of the plane.\n";
  return(0);
} //end main
double CargoMoment(double weight)
{return(CARGO_DIST*weight);
} //end CargoMoment-----*/
double CrewMoment(int crew)
{return(CREW_DIST*crew*PERSON_WT);
} //end CrewMoment
void GetData(int& crew, double& cargo)
{cout << "enter number of crew members (Maximum of 2) ";
  cin >> crew;
  while(crew <= 0 || crew > 2)
  {
  cout << endl << crew << " is an invalid entry\n"<< " re-enter number of crew, 0 < crew <= 2 ";
  cin >> crew;
  } //end while
  cout << crew << " crew members, thank you.\n\n";
  cout << "enter weight of cargo (Maximum of 5000 lbs) ";
  cin >> cargo;
  while(cargo < 0 || cargo > 5000)
{cout << endl << cargo<< " is an invalid entry" << " re-enter cargo weight, 0 < cargo <= 5000\n ";
  cin >> cargo;
} //end while
  cout << cargo << " pounds of cargo loaded. Thank you.\n\n";
  return;
} //end getdata

```

```

/*center of gravity and total weight*/
#include<iostream> //Required for cin, cout
using namespace std;
const double PERSON_WT(160.0); //Average weight/person
const double FUEL_MOMENT(1169167.3); //Fuel tank moment
const double EMPTY_WT(9021.0); //Standard empty weight
const double EMPTY_MOMENT(2751405.0); //empty moment
const double FUEL_WT(3618.0); //Full fuel weight
const double CARGO_DIST(345.0); // weight
const double CREW_DIST(120.0); //weight
double CargoMoment(double); //function prototypes here
double CrewMoment(int);
void GetData(int&, double&);
int main()
{ int crew; //number of crew on board (1 or 2)
  double cargo; //weight of baggage, pounds
  double total_weight, center_of_gravity;
  cout.setf(ios::fixed); //Set format flags.
  cout.setf(ios::showpoint);
  cout.precision(1);
  GetData(crew, cargo); // call functions start here
  total_weight = EMPTY_WT + crew*PERSON_WT + cargo+ FUEL_WT;
  center_of_gravity = (CargoMoment(cargo) + CrewMoment(crew)+
    FUEL_MOMENT + EMPTY_MOMENT)/total_weight;
  cout << endl << "The total weight is " << total_weight<< " pounds. \n"
  << "The center of gravity is " << center_of_gravity<< " inches from the
    nose of the plane.\n";
  return(0); } //end main

```

```

double CargoMoment(double weight)
{return(CARGO_DIST*weight);
} //end CargoMoment
/*-----*/
double CrewMoment(int crew)
{   return(CREW_DIST*crew*PERSON_WT);
} //end CrewMoment
/*-----*/
void GetData(int& crew, double& cargo)
{   cout << "enter number of crew members (Maximum of 2) ";
    cin >> crew;
    while(crew <= 0 || crew > 2)
    { cout << endl << crew << " is an invalid entry\n"
        << " re-enter number of crew, 0 < crew <= 2 ";
        cin >> crew;
    } //end while
    cout << crew << " crew members, thank you.\n\n";
    cout << "enter weight of cargo (Maximum of 5000 lbs) ";
    cin >> cargo;
    while(cargo < 0 || cargo > 5000)
    {   cout << endl << cargo << " is an invalid entry"
        << " re-enter cargo weight, 0 < cargo <= 5000\n ";
        cin >> cargo;
    } //end while
    cout << cargo << " pounds of cargo loaded. Thank you.\n\n";
    return;
} //end getdata

```

HAND IN LABORATORY TASK: LAB #23 EXTRA CREDIT 50 POINTS

Run the calculating the center of gravity program: Study it carefully!

1. Check the text for their hand example. try it yourself and then run your program to be sure it reproduces the hand example. I.e. All outputs in the “TESTING” part of the problem. DO NOT HAND IN THE LATTER!

2. Modify the program by adding a second cargo bay that is 522 inches from the nose of the aircraft and has a maximum cargo weight of 1000 pounds.

Run it and attach the output of the total weight and center of gravity once you do As well as copy the screen for input values similar to the “TESTING” section.

3. Do **the hand calculation** for part 2 and hand that in with the Lab. Did you get about the same answer as the program gave you in 2? Yes! then no problem,

No! go back and redesign the code.. Run till you get a reasonable match.

HINT: YOU HAVE TO TAKE INTO ACCOUNT TWO CARGO BAYS NOT ONE AND ADD THE APPROPRIATE CODE (IE FUNCTION OR FUNCTIONS)

Numerical Technique:

Roots of functions.

1. incremental search find where the function (say $y=f(x)$) changes sign for values of x by considering an interval suspected of the sign change. If it does then increment x values to get closer to the root and converge on the root (done detail in text but covered litely here)

Increment search is done on the Roots of Polynomials in text

A polynomial is generally expressed
in the form:

$$f(x) = a_0x^N + a_1x^{N-1} + a_2x^{N-2} + \dots + a_{N-2}x^2 + a_{N-1}x + a_N$$

Find values of x such that $f(x) = 0$

Degree of the polynomial is N

N roots (may be real or complex)

Easy to find roots if polynomial is factored into linear terms.

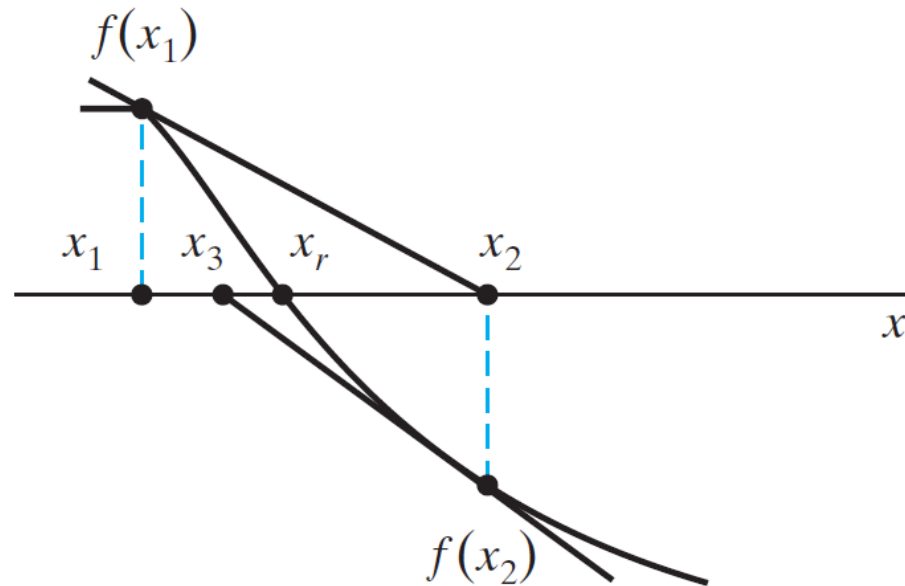
E.g. $0 = x^2 - 5x - 14 = (x-7)(x+2)$ so $x-7=0$ or $x=7$ one root $x+2=0$ or $x=-2$ the other root

Search for roots in polynomial over an interval by breaking the interval into smaller subintervals such that function is negative on one end and positive on the other end.

- Thus there is at least 1 real root
- This is always an odd number of real roots
- Keep decreasing the interval size to 'narrow in' on one root.

Newton-Raphson Method

Very Popular root-finding method using a function and its first derivative. Iteratively estimates a root from the function and its derivative. Usually requires fewer iterations than an incremental search. Note: We make an initial (intelligent) guess x_1 -> In figure below $f(x_1)$ is >0 and a slope (derivative) is projected from $f(x_1)$ to get x_2 and find $f(x_2)$ is <0 which tells us the $f(x)$ has crossed x or has a 0 value between x_1 and x_2 at x_r here. From $f(x_2)$ we projected another slope and see we get x_3 which is closer to the root x_r . By repeating this process we converge on the real root on the x -axis.

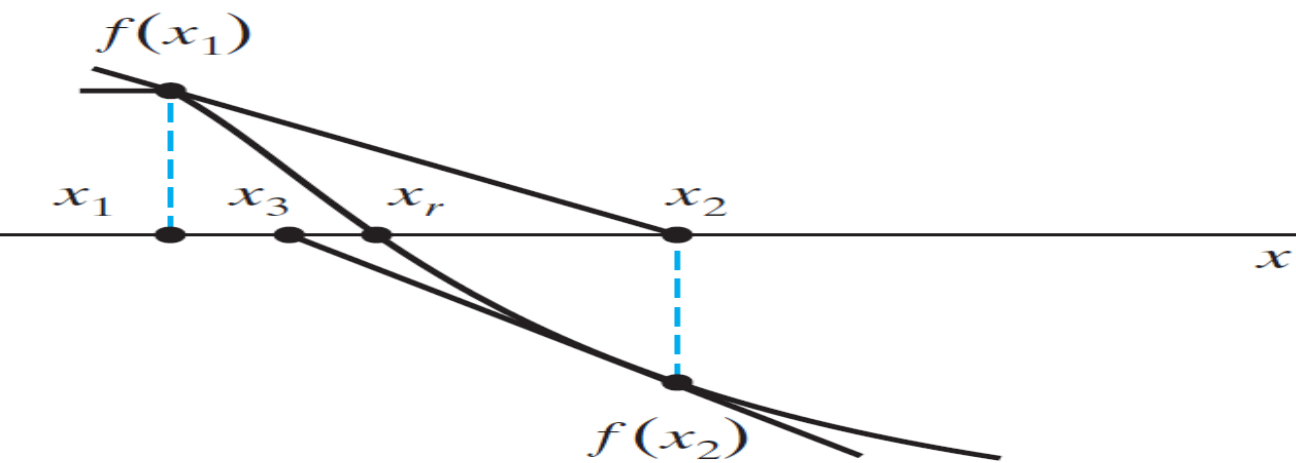


Using Newton-Raphson Method: For polynomials up to 3rd order

Given $y=p(x)=a_0x^3 + a_1x^2 + a_2x + a_3$ We set up a program to handle the general 3rd order case but will solve $y=p(x)=x^2 + 4x + 3$ and the derivative $y' = 2x + 4$ (ie slope! for next value)

We will keep getting closer to the root solution by repeating the process of repeated steps known as **iteration**. Namely, keep using the derivative to get a new value of x and evaluate the function at that point check if it is 0 for us, if not use the derivative from that new point to get a another point (This is iteration)

And set a point our program to exit with our solution when the function evaluated is close enough to 0 within a value we set, known as the **tolerance**. Similar to what we did before with conditions like $(x < 0.001)$.



The key to get the next point x_2 is to note the straight line from $x_1, f(x_1)$ to $x_2, 0$ and thus we can use the straight line formula between these two points as follows. Starting from scratch you know the formula with $y=mx+b$ so we have for the straight line generated by the slope, m at point x_2 $y=0$

So at x_2 the line is $0 = mx_2 + b$ and at point x_1 the same line is $f(x_1) = mx_1 + b$

Subtract the first from the second we get $f(x_1) - 0 = mx_1 - mx_2$

or $f(x_1)/m = x_1 - x_2$ but we need x_2 Solving for x_2 gives $x_2 = x_1 - f(x_1)/m$

and we know the slope $m=f'(x_1)$ so

$$x_2 = x_1 - f(x_1)/f'(x_1) \quad \text{is our iteration formula}$$

so we need the first point then the function and its derivative evaluated at that point to predict the next point which we will see done in the program that follows which uses for $f(x)$ a polynomial indicated by $p(x)$. Keep in mind the general $f(x)$ form just shown for problem solving.

```

/* Program chapter6_11 */
/* This program finds the real roots of a cubic polynomial */
/* p(x) using the Newton-Raphson method. */
/* this slide and next have commentary by the Prof! */
#include<iostream> //Required for cin, cout
#include<cmath> //Required for pow()
using namespace std;
int main()
{
    // Declare objects.
    int iterations(0);
/*integrations is the count for how many times we repeat the
process of getting close to the root*/
    double a0, a1, a2, a3, x, p, dp, tol; // dp is the derivative
    // Get user input.
    cout << "Enter coefficients a0, a1, a2, a3\n";
    cin >> a0 >> a1 >> a2 >> a3;
    cout << "Enter initial guess for root\n";
//hopefully intelligent especially if we graph the function
    cin >> x;
    // Evaluate p at initial guess. ie  $p=p(x_1)=f(x_1)$  from before
     $p = a0*pow(x,3) + a1*x*x + a2*x + a3$ ; value of the function
    // Determine tolerance by the value of the function at x.
    tol = fabs(p); //ie test tol aganst 0! Is it near 0 enough?

```

```

/* we set up our loop to repeat our iterations with 2 conditions*/
/*if we are not near zero or less than 100 loops have been done*/
/*keep going till we reach our "0" or stop after 100 loops*/
while(tol > 0.001 && iterations < 100)
{
// Calculate the derivative. Here 3rd order poly is a 2nd order one
dp = 3*a0*x*x + 2*a1*x + a2; // this is our m=f'(x)=dp
// Calculate next estimated root, x2, from x2=x1+f(x1)/f'(x1)
// here x on the left is x2 and x on the right is x1
// and p=f(x1) and dp =f'(x1)
x = x - p/dp;
// Evaluate p at estimated root.
p = a0*x*x*x + a1*x*x + a2*x + a3;
tol = fabs(p);
iterations++;
}
if(tol < 0.001) // wow we are out of the root and got one
{
cout << "Root is " << x << endl
<< iterations << " iterations\n";
}
else // no luck!
cout << "Did not converge after 100 iterations\n";
return 0;
}

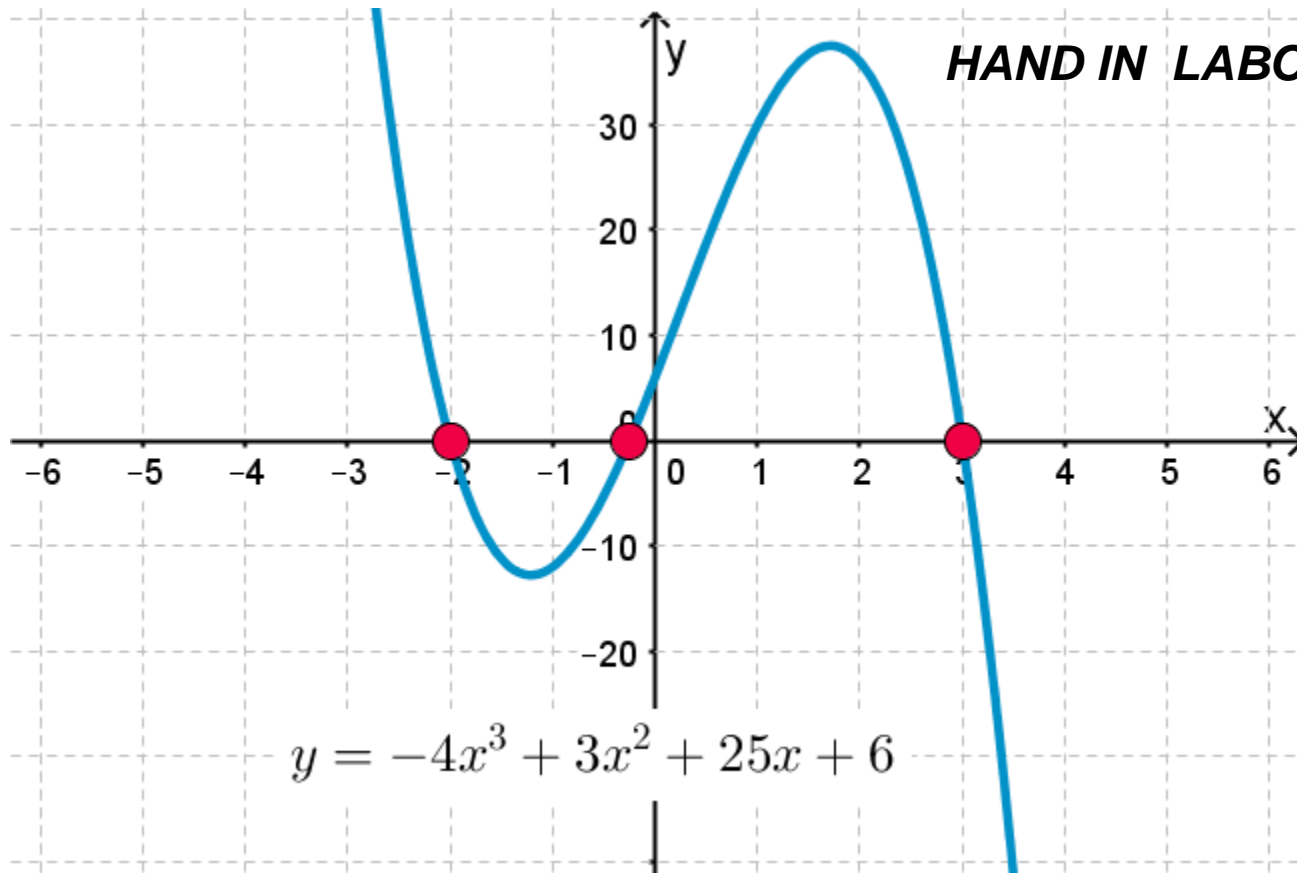
```

```

/*-----*/
/* Program chapter6_11 */
/*
/* This program finds the real roots of a cubic polynomial */
/* using the Newton-Raphson method. */
#include<iostream> //Required for cin, cout
#include<cmath> //Required for pow()
using namespace std;
int main()
{
    // Declare objects.
    int iterations(0);
    double a0, a1, a2, a3, x, p, dp, tol;
    // Get user input.
    cout << "Enter coefficients a0, a1, a2, a3\n";
    cin >> a0 >> a1 >> a2 >> a3;
    cout << "Enter initial guess for root\n";
    cin >> x;
    // Evaluate p at initial guess.
    p = a0*pow(x,3) + a1*x*x + a2*x + a3;
    // Determine tolerance.
    tol = fabs(p);
    while(tol > 0.001 && iterations < 100)
    {
        // Calculate the derivative.
        dp = 3*a0*x*x + 2*a1*x + a2;
        // Calculate next estimated root.
        x = x - p/dp;
        // Evaluate p at estimated root.
        p = a0*x*x*x + a1*x*x + a2*x + a3;
        tol = fabs(p);
        iterations++;
    }
    if(tol < 0.001)
    {
        cout << "Root is " << x << endl
            << iterations << " iterations\n";
    }
    else
        cout << "Did not converge after 100 iterations\n";
    return 0;
}

```

HAND IN LABORATORY TASK: LAB #24



$$y = -4x^3 + 3x^2 + 25x + 6$$

LAB 24: 1. Do not hand in this part. Run the previous program for the text examples of the coefficients a_1, a_2, a_3, a_4 as 0 1 4 3 and guess for root at $x = 5$ and a second time guess at $x = -4$ did you get the root solutions -0.999799 and -3.0003 then proceed.

2. Using the curve above where some of the roots are obvious run the program for this cubic equation and use close guesses (**not exact values** and think of a slope projection to the x axis) to the roots to see if you can get all three real roots. Show the run and output for all three. Have fun.

HAND CHECK YOUR ANSWERS! SHOW $y = 0$ for all three roots (on lab hand in).

Numerical Integration (area of a Trapezoid)

Integration of a function over an interval computes the area under the graph of the function.

- Important relationships in engineering (e.g. distance, velocity, acceleration, work, electric fields etc).

Computable through several techniques. We use here only the Trapezoidal Rule.

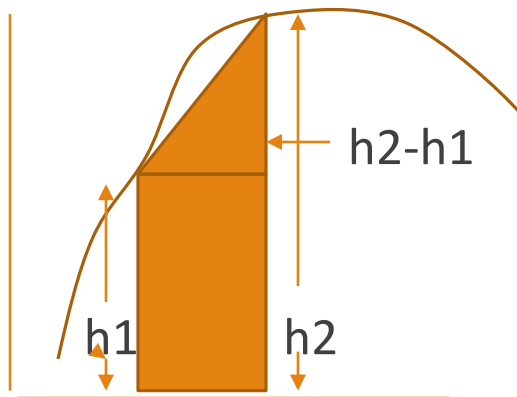
An approximate techniques for area calculations under a curve. See figure on left

Essentially over an interval eg. x_i to x_{i+1} we calculate the area of a rectangle plus triangle

Which forms the trapezoid (dark area on left).

$$\text{Area} = \text{base} * h1 + 1/2 * \text{base} * (h2 - h1)$$

$$\text{Area} = 1/2 * \text{base} * (h1 + h2) = \text{trapezoid area}$$

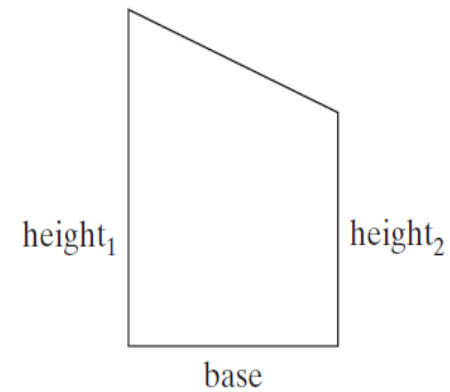


- base

Trapezoidal Rule continued

The Approximate integral by breaking integration interval into subintervals, x_1 to x_2 and x_3 to x_4 etc, illustrated in the lower figure and estimating area under the graph in each by a trapezoid. The trapezoid areas $A_1 + A_2 + A_3 + A_4$ ETC is an approximation of the interval between $a = x_1$ and $b = x_5$ see lower figure

$$\text{Area} = 1/2 * \text{base} * (\text{height}_1 + \text{height}_2).$$



applying the area formula on the left to

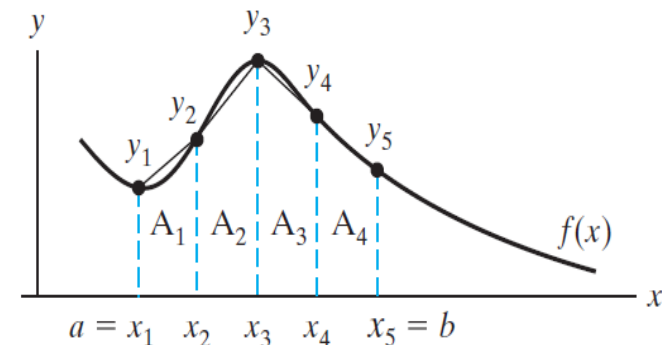
to each subsection with the base between the x values the same ($x_2 - x_1 = \text{base} = x_3 - x_2$ etc)

$$A_1 = 1/2 * \text{base} * (y_1 + y_2) \text{ with } y_1 = f(x_1) \text{ etc}$$

$$A_2 = 1/2 * \text{base} * (y_2 + y_3)$$

$$A_3 = 1/2 * \text{base} * (y_3 + y_4)$$

$$A_4 = 1/2 * \text{base} * (y_4 + y_5)$$



On developing the program.

we will divide the interval into n parts and that gives us the base = $(b-a)/n$

$$\int_a^b f(x) dx \sim \frac{\text{base}}{2} (y_1 + 2 \sum_{k=2}^N y_k + y_{N+1})$$

We use a for loop to sum up the Σ terms with the first and last values added.

And recall $y_k = f(x_k)$ in the formulas.

We setup a function call for whatever $f(x)$ we are considering and the program that follows is very general, since all you have to do is redefine the function that is called to the function you need an integral of.

In fact, the actual integration is also a function so our program will have two function calls one to the integration process via the Trapezoidal rule and that

```

/* Program chapter6_12 */
/* This program estimates the area under a given curve */
/* using trapezoids with equal bases. */
#include<iostream> //Required for cin, cout
#include<cmath> //Required for exp() function we will integrate
using namespace std;
// The two Function prototypes we will use.
double integrate(double a, double b, int n);
double f(double x); //the function f(x) is computed for values of x
int main()
{
    int num_trapezoids; // Declare objects (variables of main)
    double a, b, area;
    cout << "Enter the interval endpoints, a and b\n"; //user input
    cin >> a >> b;
    cout << "Enter the number of trapezoids\n";
    cin >> num_trapezoids;
    // Estimate area under the curve of  $4e^{-x}$ 
    area = integrate(a, b, num_trapezoids); //call integrate
    // Print result since integrate called the exp().
    cout << "Using " << num_trapezoids
        << " trapezoids, the estimated area is "
        << area << endl;
    return 0;
}

```

```
/*-----*/
double integrate(double a, double b, int n)
{
    // Declare objects local variables for this function
    double sum(0), x, base, area;
    base = (b-a)/n;
    for(int k=2; k<=n; k++) // start building trapezoid formula
    {
        x = a + base*(k-1);
        sum = sum + f(x); // function called here for values of x
    }
    area = 0.5*base*(f(a) + 2*sum + f(b)); // f(a) and f(b) called
    return area; // final area is returned
}
double f(double x) // this function is called in the above
{
    return(4*exp(-x)); // here you can change the function!
}
/*-----
--*/
```

```

/*-----*/
/* Program chapter6_12 */
/*          */
/* This program estimates the area under a given curve */
/* using trapezoids with equal bases. */
#include<iostream> //Required for cin, cout
#include<cmath> //Required for exp()
using namespace std;
// Function prototypes.
double integrate(double a, double b, int n);
double f(double x);
int main()
{
    // Declare objects
    int num_trapezoids;
    double a, b, area;
    // Get input from user.
    cout << "Enter the interval endpoints, a and b\n";
    cin >> a >> b;
    cout << "Enter the number of trapezoids\n";
    cin >> num_trapezoids;
    // Estimate area under the curve of 4e^-x
    area = integrate(a, b, num_trapezoids);
    // Print result.
    cout << "Using " << num_trapezoids
         << " trapezoids, the estimated area is "
         << area << endl;
    return 0;
}
/*-----*/
/*-----*/
double integrate(double a, double b, int n)
{
    // Declare objects.
    double sum(0), x, base, area;
    base = (b-a)/n;
    for(int k=2; k<=n; k++)
    {
        x = a + base*(k-1);
        sum = sum + f(x);
    }
    area = 0.5*base*(f(a) + 2*sum + f(b));
    return area;
}
double f(double x)
{
    return(4*exp(-x));
}
/*-----*/

```

HAND IN LABORATORY TASK: LAB #25

1. Run the previous integration program for $f(x) = 4e^{-x}$ from 0 to 1 for trapezoidal numbers $n=5, 50, 100$ and 10,000 **comment on accuracy?**. Since the actual value of the integral is known to be 2.528482 (note 6 decimal digits) which you should check by hand calculation. Compare your four answers with the known value above (show %differences) **USE 6 SIGNIFICANT DIGITS (TRY SCIENTIFIC IF YOU HAVE PROBLEMS GETTING 6 DECIMAL DIGITS). EXTRA CREDIT: RUN PART 1 FOR CONSTANTLY INCREASING** to extremely high numbers **and comment on accuracy.** Be sure to get the best value of the definite integral for comparison.

2. Find the area under the curve of the $f(x) = 3 \sin(2x)$ from 0 to 1.5 radians. Run the program for various trapezoidal numbers as in part 1. **Do a hand calculation of the integral** (show this in the report) and compare your results **which was the best RESULT for trapezoidal numbers?**(state this and show % differences!)

NOTE: two hand calculations in this assignment!