# 1 Review of Function Basics

- Useful analysis/computational units.
- Object and nonobject functions.
- Functions improve modularity of programs.
- Functions normally have two parts:
  - Prototype (sometimes called interface)
  - Definition (sometimes called implementation)

## 1.1 General Function Layout

*return_type* **name** ( <*args*> )
{
    *function body*

    *[return statement;]*
}

*return_type*
`void`, basic data types (`int`, `double`, etc.), or
user/system defined types

**name**
Any valid name.

<*args*>
The names and types of all arguments (zero or more),
separated by commas

*function body*
Zero or more valid statements.

*[return statement;]*
Non-void functions should have a `return` statement.

### 1.2 IsDigit function

Test if a character is a digit or not. There is a built-in function, `isdigit()`, for performing this operation.

```
int  IsDigit( char ch )
{
     int iRetVal = 0;

     if( ch >= '0' && ch <= '9' ) iRetVal = 1;

     return iRetVal;
}
```

## 2 Functions—A Mathematical Approach

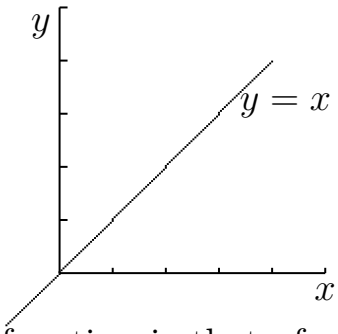
Write C++ functions from a more mathematical perspective.

- Some level of familiarity.
- Build on current knowledge.

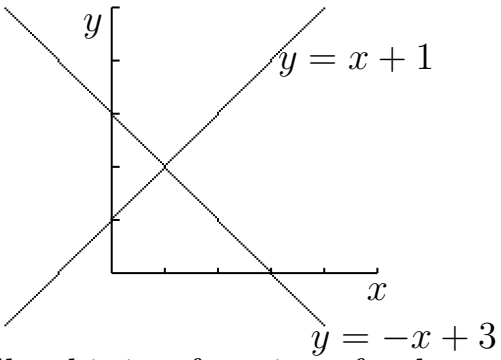
### 2.1 Mathematical Functions

Most people are familiar with the mathematical notation $y = f(x)$. This says that $y$ is a *function* of $x$.

About the simplest function that takes this form is $y = x$.




Another common function is that of a line that crosses the $y$ axis (the point that the line crosses the $y$ axis is called the $y$-intercept). A function that describes this can be written as $y = mx + b$. [Recall that $m$ is the slope of the line and $b$ is the $y$-intercept.] The figure below shows lines for $y = x + 1$ and $y = -x + 3$.




Mathematically, this is a function of $x$, but $m$ and $b$ also have an effect on the line as shown above.

Programatically, we can describe this function as

$$y = f(m, x, b)$$

Writing a C++ function for this is fairly straight forward. Consider how we would write this function for integer values of our parameters $m$, $x$, and $b$.

[**Note:** we can do this for real values also.]

```
int GenLinePt( int m, int x, int b )
{
    int y;

    y = m * x + b;

    return  y;
}
```

**Note** that the variables are named the same as our original mathematical description. Why?

```
double GenLinePt( double m, double x, double b )
{
    double y = m * x + b;

    return  y;
}
```

Does the order of the arguments matter? Not really. The *signature* of `GenLinePt()` could have been:

```
double GenLinePt( double x, double m, double b );
```

instead of

```
double GenLinePt( double m, double x, double b );
```

Again, does the order of the arguments matter?

The most important thing is that the right parameters are used when the function is invoked! Arguments in the wrong order are the source of many programming errors. This is one reason that putting *magic numbers* into programs is such a bad practice.

## 2.2 Example 1: Points on a line

```
/*  LPCalc.cpp
 *
 *  Line Point Calculator.
 *
 * ------------------------------------------------ */

#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>

int GenLinePt( int m, int x, int b );

int main()
{
    fstream fOut( "points.out", ios::out );
    if( !fOut )     //  check that file opened properly
    {
        cout << "Unable to open:  points.out" << endl;
        exit( -1 );
    }

    fOut << "Line Point Calculator\n" << endl;

    int  slope = 2;      //  line characteristics
    int  intercept = 3;
    int  x1 = 0;         //  Starting point
    int  x2 = 5;
    int  y1, y2;
```

```
    y1 = GenLinePt( slope, x1, intercept );
    y2 = GenLinePt( slope, x2, intercept );

    fOut << "slope:        " << slope << endl;
    fOut << "y-intercept:  " << intercept << endl;
    fOut << "Point 1:  (" << x1 << "," << y1 << ")" << endl;
    fOut << "Point 2:  (" << x2 << "," << y2 << ")" << endl;

    fOut.close();        //  close file
}


/*  GenLinePt
 *
 *  Generate a point on a line.
 *
 *  Parameters
 *  m   ---  slope of the line
 *  x   ---  the x-value of the point to generate
 *  b   ---  y-axis intercept
 */
int GenLinePt( int m, int x, int b )
{
    int y;

    y = m * x + b;

    return  y;
}
```

Output (stored in points.out):

```
Line Point Calculator

slope:        2
y-intercept:  3
Point 1:  (0,3)
Point 2:  (5,13)
```

## 2.3 Example 2: Polynomial Evaluation

Consider the function that describes a general 2nd order polynomial:

$$y = ax^2 + bx + c$$

Let's write a reasonably general program to calculate points on the curve over some specified range.

```
/*  PolyEval.cpp
 *
 *  Polynomial evaluator.
 *
 * ---------------------------------------------- */

#include <iostream.h>
#include <iomanip.h>
#include <fstream.h>
#include <stdlib.h>

    //  Prototypes
double  EvalPoly2( double x, double a, double b, double c );
void ShowCoefficients( ofstream& fOut,
                       double a, double b, double c );

int main()
{
    char *outFileName = "poly.out";
    ofstream fOut( outFileName, ios::out );
    if( !fOut )     //  check that file opened properly
    {
        cout << "Unable to open:  " << outFileName << endl;
        exit( -1 );
    }

        //  Polynomial coefficients
    double  a =  1.0;
    double  b = -1.0;
    double  c =  1.0;
```

```
    ShowCoefficients( fOut, a, b, c );

        //  Evaluation range variables
    int     nVals  = 10;
    double  xStart = -5.0;
    double  xEnd   =  5.0;
    double  dx = (xEnd-xStart)/(double)nVals;

    fOut << "Evaluation variables:" << endl;
    fOut << "xStart:  " << xStart << endl;
    fOut << "xEnd:    " << xEnd   << endl;
    fOut << "dx:      " << dx     << endl;

    double  x, y;

    fOut << "x          y" << endl;
    for( x = xStart ; x <= xEnd ; x += dx )
    {
        y = EvalPoly2( x, a, b, c );
        fOut << setw(10) << x << "  " << setw(10) << y << endl;
    }

    fOut.close();
}
```

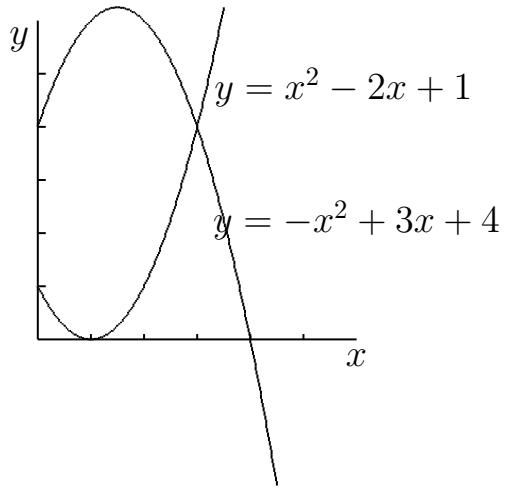
```
/*  Poly2
 *                                                    2
 *  Evaluate a general 2nd order polynomial--- a x  + b x + c
 */
double  EvalPoly2( double x, double a, double b, double c )
{
    double rVal;

    rVal = a*x*x + b*x + c;

    return rVal;
}

void ShowCoefficients( ofstream& fOut,
                       double a, double b, double c )
{
    fOut << "Polynomial coefficients for:" << endl;
    fOut << "\t    2" << endl;
    fOut << "\t a x  + b x + c" << endl;
    fOut << "a:  " << a << endl;
    fOut << "b:  " << b << endl;
    fOut << "c:  " << c << endl << endl;
}
```

Output (stored in `poly.out`):

```
Polynomial coefficients for:
    2
 a x  + b x + c
a:  1
b:  -2
c:  1

Evaluation variables:
xStart:  0
xEnd:    3.5
dx:      0.35

        x         y
        0         1
     0.35    0.4225
      0.7      0.09
     1.05    0.0025
      1.4      0.16
     1.75    0.5625
      2.1      1.21
     2.45    2.1025
      2.8      3.24
     3.15    4.6225
      3.5      6.25
```

```
Polynomial coefficients for:
    2
 a x  + b x + c
a:  -1
b:  3
c:  4

Evaluation variables:
xStart:  0
xEnd:    4.5
dx:      0.45

        x         y
        0         4
     0.45    5.1475
      0.9      5.89
     1.35    6.2275
      1.8      6.16
     2.25    5.6875
      2.7      4.81
     3.15    3.5275
      3.6      1.84
     4.05   -0.2525
      4.5     -2.75
```

Plotting these points, we see the following:



These curves were drawn with the following *code* using TEX:

```
%  Draw axes
\Draw
{\Line(120,0)   \Move(0,-7)  \Text(--$x$--)  }
{\Line(0,120)   \Move(-7,-7) \Text(--$y$--)  }

%  x-axis tick marks

%\MoveTo(20,0)  \LineTo(20,2)
% ...
%\MoveTo(100,0) \LineTo(100,2)

\Do(1,5) { \T=\DoReg; \T*20;  \MoveTo(\Val\T,0)  \LineTo(\Val\T,2) }

%  y-axis tick marks

\Do(1,5) { \T=\DoReg; \T*20;  \MoveTo(0,\Val\T)  \LineTo(3,\Val\T) }

%  y = a x^2 + b x + c  where:  x = #1, a = #2,  b = #3, c = #4

\DecVar\tOne
\DecVar\tTwo

\Define\polyTwo(4){
    \tOne=#1; \Q=#1; \tOne*\Q;  \Q=#2; \tOne*\Q;
    \tTwo=#1; \Q=#3; \tTwo*\Q;
    \R=\tOne; \R+\tTwo;  \R+#4; }

%  label curve
{\MoveTo(115,95)  \Text(--$y = x^2 - 2x + 1$--)  }

\polyTwo(0,1,-2,1)    %  Starting position
\MoveTo(0,\Val\R)
\Do(0,35){
    \T=\DoReg;  \T/10;
    \polyTwo(\T,1,-2,1)
    \LineTo(\Val\T,\Val\R) }
\EndDraw
```

## 2.4 Other function-related topics

- Parameter passing mechanisms
- Recursion
- Overloading