Passing Data to Functions

We've learned so far that putting parenthesis behind a name lets the compiler know that it is in reference to a function.

The parenthesis can also serve the purpose of **passing** data to a function and in some cases return data from the function.

Consider built in library functions that we have used in the past such as those found when using:

```
#include <cmath>
```

When that compiler directive is used we have access to lots of mathematical functions that we pass information in order for them to complete their task.

For example, if we want to raise a term to a power we would call the function:

```
pow(number,2);
```

to take the value stored in number and raise it to the second power.

The items number and 2 are said to be **arguments** and they are **passed** to the receiving function.

There are two ways to pass data to functions: **passing by value** and **passing by reference**.

Passing by Value

When you pass a variable to a function by value, a copy of the value in the variable is given to the function for it to use.

If the variable is changed within the function, the original copy of the variable in the calling function remains the same.

Passing by Reference

Functions that pass variables by reference will pass any changes you make to the variables back to the calling function.

For example:

```
void get_values(float &income, float &expense)
{
     cout << "Enter this month's income amount: $";
     cin >> income;
     cout << "Enter this month's expense amount: $";
     cin >> expense;
}
```

the code above will get two values from the user and pass them back through the parenthesis to the calling function and variables.

To pass a variable by reference, simply precede the variable name with an ampersand (&) in the function definition.

This technique should be used as sparingly as possible because it is not as safe as passing by value.

When you pass by value, you know the variable can't be changed by the function.

When you pass by reference, a programming error in the function could cause a problem throughout the program.

Returning Values Using return

When you do need to pass values back to the calling function a better way of doing this is to make use of the **return** command.

We know that all functions that aren't **void** type should return a value. Just like the main function always returns 0 to the operating system.

When other functions return a value it is returned to the calling function.

In the example below the variable stored in mm is returned to the calling function.

```
double inches_to_millimeters(double inches)
{
     double mm;     //local variable for calculation
     mm = inches / 0.03937;

return (mm);
}
```

The code below shows how you could use this function.

The statement `millimeters = inches_to_millimeters(inches);` calls the `inches_to_millimeters` function and passes the value in the variable `inches` to the function.

The function returns the length in millimeters, and the calling statement assigns the millimeters length to the variable `millimeters`.

// inches_to_millimeters.cpp

```
#include <iostream>

using namespace std;

double inches_to_millimeters(double inches);

int main()
{
     double millimeters;
     double inches = 17.5;

     millimeters = inches_to_millimeters(inches);

     cout << inches << "inches = " << millimeters << "
     millimeters" << endl;

return 0;
}
```

When using the **return** statement, keep the following points in mind:

- A function can return only one value using return. Use passing by reference to return multiple values from a function. (Or split your function into multiple parts.)

- When a return statement is encountered, the function will exit and return the value specified, even if other program lines exist below the return.

- A function can have more than one return statement to help simplify an algorithm. For example, a return statement could be in an if structure allowing a function to return early if a certain condition is met.

More About Function Prototypes

A function prototype consists of the function's return type, name, and argument list.

So far, the function prototypes specified the parameter names in the argument list.

However, this is not necessary as long as the type is specified.

For example in `inches_to_millimeters` the function could be written as:

```
double inches_to_millimeters(double);
```

In `get_values` the function could be written as:

```
void get_values(float &, float &);
```