Passing Arguments to a Function

It's very important to understand how arguments are passed to a function, as it affects how you write functions and how they ultimately operate. There are also a number of pitfalls to be avoided, so we'll look at the mechanism for this quite closely.

There are two mechanisms used generally in C++ to pass arguments to functions. The first mechanism applies when you specify the parameters in the function definition as ordinary variables (not references). This is called the **pass-by-value** method of transferring data to a function.

The Pass-by-value Mechanism

With this mechanism, the variables or constants that you specify as arguments are not passed to a function at all. Instead, copies of the arguments are created and these copies are used as the values to be transferred. Figure 3 shows this in a diagram using the example of power() function.

```
int index = 2;
double value = 10.0;
double result = power(value, index);
```

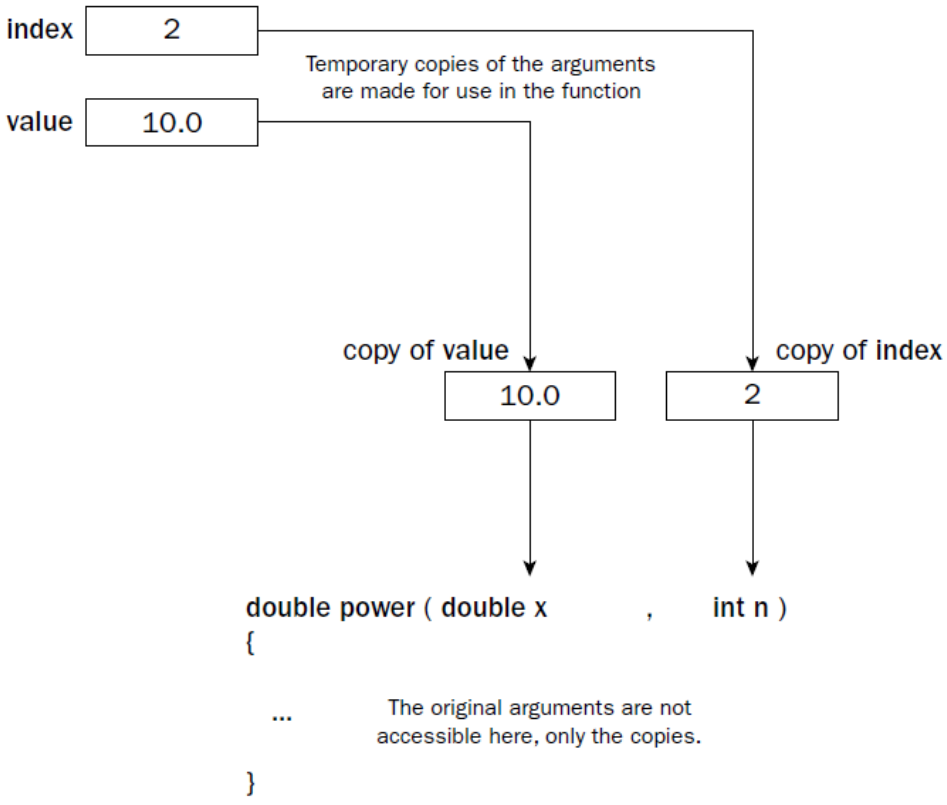

Figure 3

Each time you call the function power(), the compiler arranges for copies of the arguments that you specify to be stored in a temporary location in memory. During execution of the functions, all references to the function parameters are mapped to these temporary copies of the arguments.

Passing-by-value

One consequence of the pass-by-value mechanism is that a function can't directly modify the arguments passed. You can demonstrate this by trying to do so in an example:

```
// A futile attempt to modify caller arguments
#include <iostream>
using namespace std;
int incr10(int num);          // Function prototype
int main(void)
{
    int num = 3;
    cout << endl<< "incr10(num) = " << incr10(num)<< endl
    << "num = " << num;
    cout << endl;
    return 0;
}
// Function to increment a variable by 10
int incr10(int num)          // Using the same name might help...
{
    num += 10;       // Increment the caller argument – hopefully
    return num;      // Return the incremented value
}
```

Of course, this program is doomed to failure. If you run it, you get this output:

```
incr10(num) = 13
num = 3
```

Pointers as Arguments to a Function

When you use a pointer as an argument, the pass-by-value mechanism still operates as before; however, a pointer is an address of another variable, and if you take a copy of this address, the copy still points to the same variable. This is how specifying a pointer as a parameter enables your function to get at a caller argument.

You can change the last example to use the address to demonstrate the effect:

```
// A successful attempt to modify caller arguments
#include <iostream>
using namespace std;
int incr10(int &num);       // Function prototype
int main(void)
{
        int num = 3;
        cout<<incr10(num)<<endl;
        cout<<"num = "<<num<<endl;
        return 0;
}
// Function to increment a variable by 10
int incr10(int &num)   // Function with pointer argument
{
        num += 10;   // Increment the caller argument confidently
        return num;      // Return the incremented value
}
```

Pass-by-pointer

You can change the last example to use a pointer to demonstrate the effect:

```
// A successful attempt to modify caller arguments
#include <iostream>
using namespace std;
int incr10(int* num);                // Function prototype
int main(void)
{
    int num = 3;
    int* pnum = &num;                // Pointer to num
    cout << "Address passed = " << pnum<< endl;
    cout<< "incr10(pnum) = " << incr10(pnum) << endl;
    cout<< "num = " << num<< endl;
    return 0;
}

// Function to increment a variable by 10
int incr10(int* num) // Function with pointer argument
{
    cout << "Address received = " << num<<endl;
    *num += 10;      // Increment the caller argument confidently
    return *num;     // Return the incremented value
}
```

The output from this example is:

```
Address passed = 0x6ffe04
Address received = 0x6ffe04
incr10(pnum) = 13
num = 13
```

The address values produced by your computer may be different from those shown above, but the two values should be identical to each other.

How It Works

In this example, the principal alterations from the previous version relate to passing a pointer, **pnum**, in place of the original variable, num. The prototype for the function now has the parameter type specified as a pointer to int, and the main () function has the pointer **pnum** declared and initialized with the address of num. The function main (), and the function incr10(), output the address sent and the address received respectively, to verify that the same address is indeed being used in both places.

The output shows that this time the variable num has been incremented and has a value that's now identical to that returned by the function.

In the rewritten version of the function incr10(), both the statement incrementing the value passed to the function and the return statement now de-reference the pointer to use the value stored.

Passing Arrays to a Function

You can also pass an array to a function, but in this case the array is not copied, even though a pass-by value method of passing arguments still applies. The array name is converted to a pointer, and a copy of the pointer to the beginning of the array is passed by value to the function. This is quite advantageous because copying large arrays is very time consuming. As you may have worked out, however, elements of the array may be changed within a function and thus an array is the only type that cannot be passed by value.

Example 1:

```
#include <iostream>
using namespace std;
double average (double array[ ], int count);          //Function prototype
void main()
{
        const int max=10;
        double values[max] = { 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0 };
        cout << endl<< "Average = "<< average(values, max);
        cout << endl;
}

// Function to compute an average
double average (double array[ ], int count)
{
        double sum = 0.0;
        for (int i = 0; i < count; i++)
                        sum += array[i];            // Sum array elements
        return sum/count;                           // Return average
}
```

The program produces the following output:

```
Average = 5.5
```

Example 2:

```
#include <iostream>
using namespace std;
double transpose (double array[3][2], double Tarray[2][3]);
void main()
{
	int i=0,j=0;
	double values[3][2]={0}, Tvalues[2][3] = {0};
	cout<<"Input elements of the Array :"<<endl;
	for ( i = 0; i < 3; i++)
		for ( j = 0; j < 2; j++)
			cin>>values[i][j];

	cout<<endl<<"Elements of the Array before Transpose:"<<endl;
	for ( i = 0; i < 3; i++)
	{
		for ( j = 0; j < 2; j++)
			cout<<values[i][j] <<"\t";
		cout<<endl;
	}
	transpose(values,Tvalues);	// Calling the function Transpose
	cout<<endl<<"Elements of the Array after Transpose:"<<endl;

	for ( i = 0; i < 2; i++)
	{
		for ( j = 0; j < 3; j++)
			cout<<Tvalues[i][j] <<"\t";
		cout<<endl;
	}
	cout<<endl<<endl;
}

// Function Transpose
double transpose (double array[3][2], double Tarray[2][3])
{
	for (int p = 0; p < 2; p++)
		for (int q = 0; q < 3; q++)
			Tarray[p][q] = array[q][p];
	return Tarray[2][3];				// Return Array
}
```