CSCI 240 Lecture Notes - Part 9

We have seen that when passing simple variables to a function, the function *cannot* change those variables.

We have also seen that when passing arrays to a function, the function *can* change the values in the array - in part because the thing passed is the *address* of the array.

It turns out that in C++ a function *can* change the values of simple variables passed to it. We have not told you the whole truth - yet. The whole truth is that in C++ there are *two* slightly different ways to arrange this.

When you (the programmer) design a new function, you are in control of when and how you will do this and when you won't. When you *use* an existing function, someone else has already decided. So you must understand both ways to properly supply the arguments if you want to use other people's functions.

Why would you want to have a function change the value of an argument passed to it?

Suppose the purpose of the function was to calculate two or more answers (results). With the pass-by-value mechanism we have used up to now, you can't change *even one* simple variable, and you can *return* at most *one* value.

What if you want two or three answers?

As a simple example, suppose you want to write a function that calculates both the quotient and the remainder of an integer division. We would like to be able to call it something like this:

```
divide( dividend, divisor, quotient, remainder );
```

and have the answers stored into quotient and remainder by the function.

We can do that! One way is by using **call-by-reference**, the other is **call-by-address**.

We will cover only *call-by-reference* in this course. It is new in C++ and is a bit easier. *Call-by-address* is used in C and is also supported in C++. You should know both (sorry!) because both are used in C++ programming.

Call-by-Reference

To pass an argument to a function by reference so that the function can change it, you must

- declare a variable of the proper type in the calling program (to hold the value value that will be passed and changed)
- use an & in the function prototype and header after the data type of the argument
- alter the argument (using its local name in the function) directly

Here is an example:

Caller:

```
//prototype

void fn( int&);


//variable declaration

int num;


//function call

fn(num);
```

Function:

```
void fn( int &i )
  {
  //actually stores 10 into num back in the caller

  i = 10;
  }
```

Notice that if you look at the *function call* itself, you can't tell if it is pass-by-value or pass-by-reference. Only by looking at the function prototype or header can you tell that the argument is passed by reference so it can be altered. Likewise, if you look in the *function body* you can't tell. The only way to tell is to look at the function prototype or header. That's what the C++ compiler looks at. When it sees that & it will (quietly) arrange to pass the address of the variable to the function, so the function can access the original. So pass-by-reference is really passing an address, but it's done automatically when you use the &.

Here is the *divide()* function and calling code mentioned above:

Caller:

```
//prototype

void divide( int, int, int&, int& );


//declarations

int num1 = 20, num2 = 6, quo, rem;


//function call

divide( num1, num2, quo, rem);

//now quo has a value of 3 and rem has a value of 2
```

Function:

```
void divide( int dividend, int divisor, int& q, int& r )
  {
  q = dividend / divisor;
  r = dividend % divisor;
  }
```

Notice that we passed the divisor and dividend by *value* since we just wanted to use their values and not alter them. We passed the quotient and remainder by *reference* since the purpose of the function is to *alter* both of these values.

Notice, too, that quotient and remainder were not initialized by the caller since their values are never actually *used* in the function, they are just *assigned* values. But the divisor and dividend were initialized since their values were *used* to compute the answers.

For example, remember *swap()* as explained in lecture? It could be written as a function as follows:

```
void swap( double a[], int top, int small )
  {
  double temp;

  temp = a[top];
  a[top] = a[small];
  a[small] = temp;
  }
```

In this case we had to pass the array and two integers to serve as the subscripts of the elements we wanted to swap, because the only way to use a function to change a part of the array was to pass the whole thing.

We would call it as follows:

```
swap( ar, top, s );
```

Now, however, we could write it using reference arguments:

```
void rSwap( double & val1, double & val2)
  {
  double temp;

  temp = val1;
  val1 = val2;
  val2 = temp;
  }
```

And it would be called by code like this:

```
rSwap( ar[top], ar[s] );
```

In fact, any two double in memory could be swapped using this function now, not just doubles in a particular array.

```
double first = 4.7;
double second = 6.8;


rSwap(first, second);
```

Now first has 6.8 and second has 4.7

More Examples of Call By Reference

Example 1: suppose we have a program that prints (on paper) a multi-page report. We want a function to print a title or header at the top of each new page, complete with page number.

We will declare a page number variable in *main()*, but would like the function to increment it automatically each time it is called.

```
void pgHeader(int&);


int main()
{
int pageNum = 1;

while (...)
  {
  // more code ...

  if (time to print a pg hdr)
    pgHeader(pageNum);
  }

return 0;
}


void pgHeader( int& pnum )
{
cout << "The Title " << pnum;

//increment pageNum in main()

pnum++;
}
```

Note: we could alternately have designed *pgHeader()* to *return* an incremented page number (the old pass-by-value way). For comparison, here it is:

```
int pgHeader(int);


int main()
{
int pageNum = 1;

while (…)
  {
  // code to do something useful…

  if (it's time to print a page header)
    pageNum = pgHeader(pageNum);
  }

return 0;
}


int pgHeader(int pnum)
{
cout << "The Title " <<  pnum;

pnum++;  // increment copy

return (pnum); // new value of page number
}
```

Example 2: suppose we have a *sum* and a *count*, and want to write a function to calculate an average. However, sometimes the *count* will be zero, so we also want the function to tell us if it was able to successfully accomplish its task.

We could design the function to return a value indicating success or failure, and to pass back the calculated average (when count != 0).

We will return 0 to indicate failure and 1 to indicate success.

```
int calcAvg( int sum, int count, double &avg )
{
if (count == 0)
  return (0);
else
  {
  avg = (double)sum / count;
  return (1);
  }
}
```

And the calling function:

```
int total, count, retCode;
double avg;


//get values into total and count

retCode = calcAvg(total, count, avg);

if ( retCode == 0 )
  cout << "no average calc'd";
else
  cout << "Average is " << avg;
```

Here, the function is returning one value (the return code) and passing back one value - the calculated average, when the calculation is performed, i.e. when count is not 0. If the count is 0, the *avg* argument is not altered.

Study these examples until they all seem clear to you. Notice the patterns that are common to each function call. This may take some time. You will probably find it helpful to come back to them several times. Each time, you will be able to understand them more quickly. After you have done this, try writing your own similar functions.

Sample Exercises

Here are a few sample exercises to try.

1. Write a function that takes an array of integers and passes back the highest and lowest value in the array.
2. Write a function that takes an array of doubles and passes back the sum and the average of the numbers in the array.
3. Write a function that takes an integer number and passes back the sum of the integers up to that number. For example, passing in 5 would pass back 1 + 2 + 3 + 4 + 5 = 15

Local Reference Variables

We will not have much use for this topic in this course, but you should be aware of it: it is possible to declare a special variable type that can hold a reference to another simple variable:

```
int a = 4;
int& b = a;


b = 5;
```

What does this mean?

Line 1: declare a simple int variable *a*, and give it a value of 4.

Line 2: declare a reference variable, *b*, and make it "refer to" an int variable, *a*. The *int &* denotes a new data type, one that can hold a reference to an int. In a sense, *b* is an alias for *a*.

Line 3: when we assign 5 to *b*, we really are assigning 5 to what *b* refers to, which is *a*. *a* now has the value 5.

We can make reference variables of any simple data type. You may revisit this topic in future courses.