MA400: Financial Mathematics

Introductory Course

Lecture 6: Pointers and functions

Pointers as function arguments

So far, we have passed arguments to functions by **value**: that is, the function makes a copy of the value of each argument and then manipulates it.

However, it is possible to pass pointers as arguments to functions, so that they can manipulate the actual quantities.

```
void swap(int *pt_x , int *pt_y)
{
  int temp;
  temp = *pt_x;
  *pt_x = *pt_y;
  *pt_y = temp;
}
```

This could then be called using:

```
swap( &i, &j);
```

Note that you should think very carefully about whether you wish a function to have this ability.

Arrays as function arguments

One valid need is to pass arrays as arguments to functions.

```
double sum(double pt[], int n = 100)
{
  double temp = 0.0;
  for ( int i = 0 ; i < n ; ++i )
    temp += pt[i];
  return temp;
}
```

To use such a function, all we must pass for the array argument is to pass an **address**.

```
double price[100];
sum(price);
```

Note that, due to the function header, we cannot pass an array of integers to this function.

Arrays as function arguments

If all we need to pass is an **address**, then we can also perform the following:

```
double price[100];      // Set price[i] = i + 1
sum(price,4);           // Returns 10
sum(price[0],4);        // Returns 10
sum(price[1],3);        // Returns  9
```

Of course, we again must be wary that we do not go beyond the limits of our array.
Both the following will try to call `price[100]`, and thus return arbitrary results:

```
sum(price[100]);
sum(price[99],2);
```

Arrays as function arguments

Recall the function header for `sum()`:

```
double sum(double pt[], int n)
```

Here `pt` is simply a pointer (not the whole array). As such, the following is an equivalent header:

```
double sum(double *pt, int n)
```

It can even be used as a pointer, e.g. being assigned addresses:

```
double sum(double pt[], int n)
{
  double temp = 0.0, *pt_end;
  pt_end = pt + n;
  while ( pt < pt_end )
    temp += *pt++;
  return(temp);
}
```

Multi-dimensional arrays as function arguments

Multi-dimensional arrays can also be passed as arguments to functions.

Here, it is necessary to specify to the function all the arrays dimensions except the first.

So, for a function to act on 2×3 arrays of doubles, we might have the following function declarations:

```
double fn_1(double x[2][3]);
double fn_2(double x[][3]);
```

Multi-dimensional arrays as function arguments

To pass an appropriate array to these functions, we must pass to them the **base address of the array**, e.g.

```
double out1 = fn_1(x);
```

Note that the following is **not valid**:

```
double out2 = fn_1( &x[0][0] )    // Wrong
```

In this context, `&x[0][0]` is interpreted as the address of a `double`, not the base address of the array.

Working with the storage map

Compare the following two code fragments for summing two 100×100 arrays.

```
void sum(double a[][100],double b[][100],double c[][100])
{
  for ( int i = 0 ; i < 100 ; ++i )
    for ( int j = 0 ; j < 100 ; ++j )
      c[i][j] = a[i][j] + b[i][j];
}
sum(a, b, c);

void sum(double *pt_a, double *pt_b, double *pt_c)
{
  double *pt_end = pt_c + 100 * 100;
  while (pt_c < pt_end)
    *pt_c++ = *pt_a++ + *pt_b++;
}
 sum(&a[0][0], &b[0][0], &c[0][0]);
```

Passing arguments to `main()`

The `main()` function is able to take two arguments; however, these must be of a very particular type:

- an `int` – conventionally called `argc` – **and**
- an array of `char` pointers – conventionally called `argv`.

That is, the `main()` function definition can have the form:

```
int main(int argc, char *argv[])
{
  // Main code
}
```

- What do these represent?
- How are they passed to the `main()` function?

Passing arguments to main()

In the worksheet entitled **Your programming environment**, you should have seen how to run your compiled program from the command line.
Briefly:

1. Given a source file entitled my_program.cpp, ...
2. ...your compiler will produce an executable called my_program.exe.
3. This can be run from a DOS prompt simply by typing the program name and hitting Enter.

   ```
   H:\>my_program.exe 10 1.0
   ```

   or just

   ```
   H:\>my_program 10 1.0
   ```

To pass the arguments "10" and "1.0" to my_program we could simply enter them after the program name.

Passing arguments to main()

What happens when we execute:

```
my_program.exe 10 1.0
```

The program will receive **three** command line arguments.

- the program name: `my_program.exe` or `my_program`,
- the first parameter: `10`,
- the second parameter: `1.0`.

Thus, `argc` is equal to 3.

These are then associated with elements in the `argv` array:

- `argv[0]` points to the program name,
- `argv[1]` points to the argument "10",
- `argv[2]` points to the argument "1.0".

Passing arguments to main()

If my_program.cpp consists of the following code fragment:

```
int main(int argc, char *argv[])
{
  cout << "The program name is: " << argv[0] << '\n';
  cout << "There are " << argc << " arguments\n";
  for ( int i = 0 ; i < argc ; ++i )
    cout << "\tArgument " << i
         << " is " << argv[i] << '\n';
  return(EXIT_SUCCESS);
}
```

Then

```
H:\>my_program.exe 10 1.0
The program name is: my_program.exe
There are 3 arguments
        Argument 0 is my_program.exe
        Argument 1 is 10
        Argument 2 is 1.0
```

Passing arguments to main()

Note that when executing:

```
my_program.exe 10 1.0
```

all the arguments received are interpreted as **strings**.

To use arguments as numbers, you must first convert them. <cstdlib> provides several functions to do this:

- atoi() converts a string to an integer:

```
int i = atoi("1234");       // i has value 1234
int j = atoi("123x4");      // j has value 123
int k = atoi("x1234");      // k has value 0
```

- atof() converts a string to a double:

```
double x = atof("123");     // y has value 123
double y = atof("12.3");    // y has value 12.3
double z = atof("12.3e45"); // z has value 1.23e+45
```

Exercise

(Follows on from the last exercise in the handout on Functions.) Write a program `prime` using `test_prime()` and `list_primes()` that takes an integer on the command line and then lists all primes less than or equal to this integer. It should be executed using:

```
prime 100
```

a) Run your `prime` program with various values for the command line parameter. Does `test_prime()` deal appropriately with all possible arguments? If not, make suitable modifications to your program.

b) Modify your program so that it only lists the primes between two numbers specified by two command line arguments.
   For example, by typing:

   ```
   prime 100 300
   ```

   You should include code that appropriately handles alternative inputs such as:

   ```
   prime 300 100
   ```

Functions as function arguments

There are many reasons we might wish to pass a function as an argument to another function.

Suppose we wish to rewrite our function `sum()` so that it sums the first n values of a given function.
We would modify earlier versions of this program with the line:

```
temp += f(i);
```

But if we wish to do this for other functions `g()`, `h()`, ... as well, we would have to write new versions of `sum()` for each of them!

It would be easier to have just one `sum()` function, which could take both an integer and the function to sum over.

Functions as function arguments

A pointer to a function looks very much like a function header; just with the addition of a *.
Thus, given the function declarations:

```
int count();
double f(double x, double y);
```

we would declare pointers to them using:

```
int (*count)();
double (*f)(double x, double y);
```

Here, the function operator () binds tighter than the dereferencing operator *, hence the need for brackets.

Functions as function arguments

Thus, a suitable function definition for our revised `sum()` function would be:

```
double sum(double (*op)(int), int n);
{
  double temp = 0.0;
  for ( int i = 0 ; i < n ; ++i )
    temp += (*op)(i);
  return temp;
}
```

Note that op is a pointer so we must dereference it when we use it. However, we can also call upon op directly.

Functions as function arguments

Thus, for a program that could explore the sum of the first n integers, squares, and cubes we could have something like this:

```
double p1(int i);  // Declare a function p1(i) = i
double p2(int i);  // Declare a function p2(i) = i^2
double p3(int i);  // Declare a function p3(i) = i^3
double sum(double (*p)(int), int n);

int main()
{
  .....
  double s1 = sum(p1,10); // Sum of first 10 integers
  double s2 = sum(p2,20); // Sum of first 20 square
  double s3 = sum(p3,30); // Sum of first 30 cubes
  .....
}
```

Functions as function arguments

You can also define arrays of pointers to functions, though the notation is a little trickier.
Since both the index operator [] and the function operator () bind tighter than the dereference operator *, we must have:

```
double (*p[3])(int);
```

We can then initialize them (**very important**):

```
p[0] = &p1;
p[1] = &p2;
p[2] = &p3;
```

We could then rewrite our code fragment as follows:

```
double s[3];
for ( int j = 0 ; j < 3 ; ++j )
  s[j] = sum(p[j],n);
```

Pass by Reference

Recall, for functions that take pointers as arguments, their definitions take the form:

```
void swap(int *pt_x , int *pt_y)
{
  int temp;
  temp = *pt_x;
  *pt_x = *pt_y;
  *pt_y = temp;
}
```

which can be called by:

```
swap( &i, &j );
```

Pass by Reference

An alternative syntax is available for us do this, using the **reference declarator** & to declare reference arguments

```
void swap( int &x, int &y )
{
  int temp;
  temp = x;
  x = y;
  y = temp;
}
```

which can be called by:

```
swap( a, b);
```

Reference variables

References essentially play the role of aliases.

```
int x;
int &y = x;
x = 10;        // Calling on y will return the value 10
y = 200;       // Calling on x will return the value 200
```

In this situation, `y` is known as a **reference variable**.

Pass by Reference

Once again, the use of references (as with pointers) in a function erodes the modularity of your code:
Auditing what is happening to variables in your program is made all the more difficult if they can be changed within the bodies of functions.

You should therefore use sparingly, and only when necessary. E.g.

- when accessing a large quantity of memory,
- when changing variables in the calling environment.

Exercise: Pass by reference

Use pass by reference to implement and test a function that interchanges the values of three `double` variables – `x`, `y` and `z` – so that $x \leq y \leq z$.

Reference return values

Functions may also return references (i.e. aliases); here the reference declarator is attached to the function identifier.
For example:

```
double &component(double *vector, int i)
{
  return vector[i-1];
}
```

This allows us to access the elements of an array, but with the indices starting from 1.

```
double y[5] = {1, 2, 3, 4, 5};
cout << y[3] << '\n';
cout << component(y,4) << '\n';
```

Reference return values

Alternatively, one could use an alias to return just the diagonal elements of an $n \times n$ array.
For example, suppose we were working with 5×5 arrays.

```
double &Diag(double x[][5], int element)
{
  return x[element-1][element-1];
}
```

For an appropriate array, e.g. `A[5][5]`, the following are equivalent ways of obtaining the i^{th} diagonal entry:

```
Diag(A, i)
```

Reference return values

Finally, note that we can do this without the subscript notation, using pointers directly.

```
double &diag(double *pt_matrix, int row)
{
  return *(pt_matrix + (row - 1) * 5 + (row - 1));
}
```

The i^{th} diagonal element is then called via:

```
diag(&A[0][0], i)
```

Exercise

Implement a function to transpose a matrix.
The transposition should overwrite the original matrix.