MA400: Financial Mathematics

Introductory Course

Lecture 4: Functions

Re-using code via functions

Consider the definition of the binomial coefficient:

$$\binom{n}{r} = \frac{n!}{r!(n-r)!}.$$

This requires the calculation of 3 separate factorials.

So it would make sense to have a `factorial()` function so that we can implement the algorithm once and re-use it:

```
int binom;
binom = factorial(n) / (factorial(r) * factorial(n-r));
```

Actually, you might see the benefit of a `binomial()` function:

```
int coef = binomial(n,r);
```

A simple example from the two-period model

In the two period model of discrete finance, where an initial price S_0, under and interest rate r, either rises to S_0U or falls to S_0D, the implied probability of an increase with the formula:

$$q_{UP} = \frac{S_0(1+r) - S_0D}{S_0U - S_0D} = \frac{(1+r) - D}{U - D}.$$

If we were to use this a lot, we might prefer to wrap this code up into a C++ function of the form:

$$rnm : (U, D, r) \longmapsto \frac{(1+r) - D}{U - D}.$$

The general form of a function

```
type identifier(type identifier, [ ...])
{
  // Function code
}
```

- The type declaration outside the bracket is the **return type** of the function: **every function must have a return type**.
- Of course, every function must have an identifier, with the usual conventions.
- Inside the parentheses are a list of typed arguments that the function may require.
- Note that **the parentheses are required**, even if the function were to take no arguments, e.g. `main()`.
- The first line is sometimes refereed to as the **function header**.
- Inside the code block is the function code; this should return a value consistent with the **return type**.

A skeletal example function definition

```
int factorial(int n)
{
  int result;
  // Function code
  return result;
}
```

- The **return type** (output value) of this function is an integer.
- The name, or identifier, of this function is factorial. The convention is then to refer to this function as `factorial()` in text.
- `factorial()` takes one argument: the **argument type** is `int`...
- ... and the **formal argument** is `n`.
- The `return` statement terminates the function, and should return an output value consistent with the return type.

Note the absence of a trailing semi-colon.

The return type of the function

Note that while the value returned by a function must be consistent with its declared return type, some internal conversions are applied to handle some cases.

```
double factorial(int n)
{
  int result = 0;
  // Function code
  return result;
}
```

Here, the result returned (`result`) is of type `int`, but is automatically converted to `double`.
As well as type conversion, C++ can also do some error trapping, e.g. ensuring that a result is actually returned.

The **void** data type

Every function requires a return type.

What if your function does not need to return a value: e.g. you may have defined a function, `print_formatted()`, purely to print to an output stream?

For this, C++ provides the `void` data type.

```
void print_formatted(...)
{
  // Function code
  return;
}
```

Note that in this case, we can omit the `return` statement.

The **void** data type

We can also use the `void` data type to indicate when a function requires no arguments: e.g. a function that prints a welcome statement.

```
void print_welcome(void)
{
  cout << "Hello, world!";
  return;
}
```

It is a good idea to use `void` in this manner rather than having empty brackets:

```
void print_welcome()
{
  // Function code
}
```

Of course, the most familiar example we have is:

```
int main(void)     // A perfectly valid function header
```

The **void** data type

This is the only way we can use the `void` data type.

In particular:

- `void` cannot be used as an argument type:

  ```
  int factorial(int n, void)
  ```

- variables cannot be declared to be of data type `void`:

  ```
  void empty;
  ```

Invoking a function

Suppose we have defined a function `sum`:

```
int sum(int n)
{
  // Function code
  return result;
}
```

It is invoked (executed) via:

```
 sum(n);
```

Here we distinguish the **actual argument** `n` from the **formal argument** of the definition.
In fact, the following are valid calls too:

```
sum(10);
sum(i);
sum(some_number);
```

Invoking a function

Passing expressions as arguments to functions is valid, but the results are inconsistent.
The order of evaluation of expressions in argument lists is not defined (standardised).

Thus

```
int j = sum(2 * i + 1);
```

is valid, but different compilers can evaluate the expression differently.

In this case, it is safer to use the following:

```
int n = 2 * i + 1;
int j = sum(n);
```

Invoking a function

For functions that require no inputs, **we must still specify the parentheses**.

```
print_welcome();     // Correct.
print_welcome;       // Wrong
```

The latter is not invalid, but will return nothing useful.
Your compiler may give rise to a warning about using a reference, and not a call, to a function.

Declaring a function

Note that before a function can be invoked in a program, it must have previously been declare or defined.
We have already seen a **definition** of a function `factorial()`:

```
int factorial(int n)
{
  // Function code
  return result;
}
```

A **declaration** for this would be:

```
int factorial(int n);
```

Notice the presence of a trailing semi-colon.

A function declaration does not preclude a definition, but it allows the definition to occur in a program after it has been invoked.

Declaring a function

Note that a function can call upon other functions.

- the `main()` function calling upon a function `sum()`.
- a `binomial()` function that calls upon a `factorial()` function.

However, you cannot declare nor define a function from within another function.

```
int main()
{
  int factorial(int n)           // WRONG
  {
     // Function code
  }
  int i = factorial(10);
  return(EXIT_SUCCESS);
}
```

Basic structure of a program

```
#include <iostream>
int factorial(int n);
int main()
{
  // Code
}
int factorial(int n)
{
  // Function code
}
```

1. the include directives calling upon other libraries,
2. declarations of your own functions,
3. the main function,
4. definitions of your own functions.

Providing arguments to a function

All arguments specified in a function header must be given in a call to that function.

However, not all formal arguments may actually be used by the function code.

```
void print_error(int)
{
  cout << "An error has occurred.\n";
}
```

Though it is unused, we must still specify the argument when calling the function:

```
print_error(178);        // Correct
print_error();           // Wrong.
```

Providing arguments to a function

Conversely, we cannot provide more arguments to a function than have been declared in a function header.

Suppose we define a function to calculate the exponential of a given number x using a series expansion up to some pre-determined order.

```
double exp(double exponent)
{
  // Function code using series expansion
}
```

We may write code with the intention of modifying `exp()` in the future so that the user can specify an order of calculation, e.g.

```
exponential(3.1243, 10);          // Wrong
```

The compiler will not allow this to happen.

What we should do is update the function header to read:

```
double exp(double exponent, int);
```

Default arguments

A function **declaration** can also specify default values for any of its arguments:

```
double exp(double x, int order = 10);
```

In this case, the following are all correct:

```
exp(3.142, 10);       // Correct, but 10 not necessary
exp(1.738, 20);
exp(1.421);
```

While a particular default value can be set in either a function declaration or a function definition, **it cannot be set in both** – even if it is to the same value.

Practice consistency.
It is advised to set the default values in the function declaration, rather than the function definition.

Default arguments

With regards to default arguments, the C++ compiler will assume that the **trailing arguments** are the missing ones.
Thus

```
exp(double x, int order = 10);
```

is a valid declaration, but

```
exp(int order = 10, double x);
```

is not.

Providing arguments to a function

All arguments specified in a function header must be given to a function:

- either directly in the function call,
- or indirectly as a default value.

Exercise: Defining an exponential function

a) Using the series expansion of the exponential function, implement and test a function that has two arguments – a double x and an integer n – and returns the series approximation of $\exp(x)$ up to the n^{th} term in the series.
   If the order is not specified when your function is called, it should default to the value 3.
   If no arguments are specified when your function is called, then it should take a default value of $x = 1$.

b) Do the same again, but for $\sin(x)$, where the default order is 3, and the default angle is 0.

In both cases, you can use the `cmath` library to compare your approximation with the values returned by the standard `exp()` and `sin()` functions.

Exercise: Declarations, definitions and default values

Note that formal arguments do not require names (they are processed from right to left).
However, if you are not careful with your coding, this can lead to some strange results.

Given the following declaration and definition for a function `f()`:

```
double f(double x, double y = 20);
double f(double y, double x)
{
  cout << "The first argument has value  " << y << '\n';
  cout << "The second argument has value " << x << '\n';
}
```

what are the results of the following function calls?

```
exponential(3.142,10);
exponential(2.713);
```

Recursive functions

Functions can call other functions; in particular, they can call themselves; this is known as **recursion**.

Recursive functions generally follow the same pattern:

- a base case,
- a call to itself, such that ...
- the argument has changed from before, and in the direction of the base case.

While a very powerful method, you should use recursion with care. Each call to the function implies an overhead, and for very deep recursions (i.e. many calls to itself) your function may be

- inefficient, or
- crippling to the program.

Exercise: The Fibonacci sequence revisited

The Fibonacci sequence was introduced in the computer worksheet.
Implement a function, with the declaration:

```
double u(int n);
```

to recursively generate the Fibonacci sequence as defined in the worksheet.
Check that the results given by the two exercises are consistent.

Scope

Three kinds of scope:

- local,
- file, or global, and
- class.

Variables defined within a function have **local scope**, and are thus hidden from the rest of the program.

Different functions have distinct scopes.

Scope

```
int sum(int n)
{
  int result = 0;
  for (int i = 1 ; i <= n ; ++i)
    result += i * i;
  return result;
}
int main()
{
  int result;
  for (int i = 1 ; i <= 10 ; ++i) {
    result = sum(i);
    cout << "sum of the first " << i << " squares is "
         << result << "\n";
  }
  return(EXIT_SUCCESS);
}
```

File, or global, scope

Variables with global scope are seen throughout the entire file (hence file scope).

In particular, every function defined in a file has access to such global variables.

As such, it is recommended that you keep any global variables as constant, using the `const` specifier.

The scope resolution operator: ::

If a local variable and a global variable have the same name, then the local variable hides the global variable.

However, the global variable can still be accessed using the **scope resolution operator**, ::.

```
const double x = 3.14159;
int main()
{
  double x = 3.1;
  cout << "x = " << x << ", ::x = " << ::x << '\n';
  return(EXIT_SUCCESS);
}
```

This will return:

```
x = 3.1, ::x = 3.14159
```

The scope resolution operator: `::`

We have seen the scope resolution operator before in the context of the standard namespace:

```
std::cout
```

Suppose we have different implementations of particular functions: for example, valuation functions that arise from a discrete model, and those that arise from continuous model.

We may choose to collect these in

- `discrete.cpp`
- `continuous.cpp`

If these files contain functions that share the same name, they can then be distinguished using `::`. E.g.:

- `discrete::value()`
- `continuous::value()`

Storage class `static`

Each time control enters a function body, all local variables are initialized.
That is, there is no memory of their values once control leaves the function body.

Sometimes you may want a variable to retain its value after control leaves its defining block.

This can be done using the `static` class within a function body, e.g.

```
static int count;
static int grand_total = 0;
```

The variables `count` and `grand_total` are only initialized once.
By default, all static variables are initialized to zero when no value is given.

Storage class static

```
void tally()
{
  static int count;
  cout << "The value of count is " << count++ << '\n';
}
int main()
{
  for (int i = 0 ; i <= 3 ; ++i )
    tally();
  return(EXIT_SUCCESS);
}
```

produces

```
The value of count is 0
The value of count is 1
The value of count is 2
The value of count is 3
```

Overloading function names

It is possible to use the same function name for functions with different function bodies.

For example, we may want to define a `pow()` function that takes two arguments, and raises one to the power of the other.
Now if the exponent is an integer, we could use a very simple algorithm using, say, a `for` loop.
However, our method might be slightly more complicated if the exponent was, say, π.

We would then have two functions:

```
double pow(double x, int y);
double pow(double x, double y);
```

Writing both of these in a C++ program would be valid.

Overloading function names

In general, multiple function definitions must be differentiated by either:

- different numbers of arguments, or
- different argument types.

For example, the following are valid:

```
double norm(double x, double y, double z);
double norm(double x, double y);

float norm(float x, float y);
```

However, this is not:

```
float norm(double x, double y, double z);
```

Overloading function names

Consider

```
double norm(double x, double y);
float norm(float x, float y);
```

By default

```
double call1 = norm(2.0, 3.0);
```

will call the double version of our `norm()` function.
To ensure that the float version is chosen over the double version, we would specify float arguments, e.g.

```
double call2 = norm(2.0f, 3.0f);
```

Exercise: Overloading functions

Implement overloaded functions to calculate an approximation to:

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}$$

One version should have an argument and return type of `double` and a second version should use the `float` type.
The number of terms used in each function should be appropriate for the type (see lecture 2 for accuracy levels).
Test the functions either against the library function, `exp()`, which is declared in `<cmath>`, or by using your function to calculate

$$e^x e^{-x} = 1.$$

Check that the test program really does invoke the two different functions.

The `main()` function

Recall, this function is required of all C++ programs (and may be the only such function).

As with other functions, `main()` can take arguments.

However, it faces restrictions that general functions do not.

- It cannot call itself.
- It cannot be overloaded.
- There can only be one `main()` function in any program.

Example: One way of defining a factorial function

```
int factorial( int n )
{
  int result = 1;
  if ( n > 0 ) {
    do {
      result *= n;
      --n;
    } while ( n > 1 );
  }
  else if ( n < 0 ) {
    cout << "Error in factorial function: "
         << "argument = " << n << '\n';
    exit(EXIT_FAILURE);          // From <cstdlib>
  }
  return result;
}
```

Exercise

Using a suitable function from the Standard Library supplied with your system, write a program that measures how long it takes to evaluate the factorial function.
Use various function arguments and compare the times for both the **recursive** and **iterative** versions of the function.
It is worth using at least four-byte integers for this example, since the factorial function grows rapidly with the magnitude of its argument.
(On a very fast computer, you may find it difficult to get a non-zero elapsed time with a sensible value for the factorial function argument. In this case you should time the same calculation carried out many times.)

Exercise

Use the power series expansion of $\sin(x)$ for $|x| < \infty$ to implement a sine function with the declaration:

```
float sin(float x);
```

(Truncate the series and only consider $|x| \leq \pi/2$.)
Compare the accuracy and time taken to evaluate `sin(x)` with the Standard Library function in `<cmath>`. Make sure you really do invoke the two overloaded `sin()` functions.
Aim for accuracy and efficiency:

1. Do not evaluate unnecessary terms in the power series.
2. Instead of evaluating the factorials for every function call, use pre-calculated values for the inverses. In fact, for a truncated power series, more accurate values of the coefficients are given in standard mathematical tables.
3. Reduce the number of multiplications by using nested parentheses (Horner's method) as in:

$$\sin x \approx \left(\left(\left(\left(a_8 x^2 + a_6\right) x^2 + a_4\right) x^2 + a_2\right) x^2 + 1\right) x.$$

Exercise

Write a function `test_prime()` that tests whether a positive integer is prime.
The function should have just one argument – an unsigned integer `n` – and return a boolean value: **true** if `n` is prime, and **false** otherwise.

Write a second function with declaration:

```
  void list_primes(unsigned m)
```

that uses `test_prime()` to list all prime numbers up to `m`.

Write a program using the `test_prime()` and `list_primes()` functions, that prompts for a positive integer and then lists all primes less than or equal to this integer.

Additional exercises

§5.11, Chapter 5, Capper.

- Exercise 8.
- Exercise 6.
- Exercise 10.