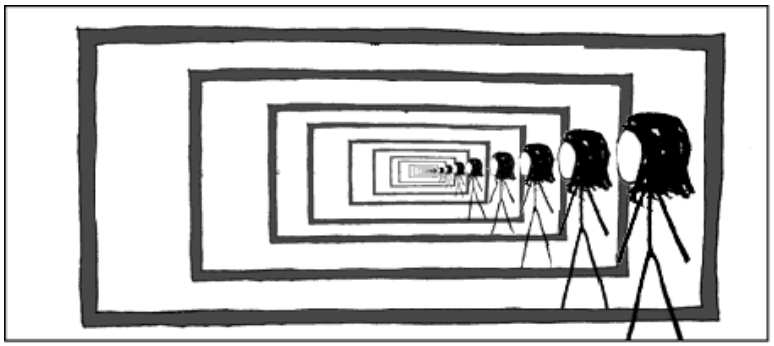
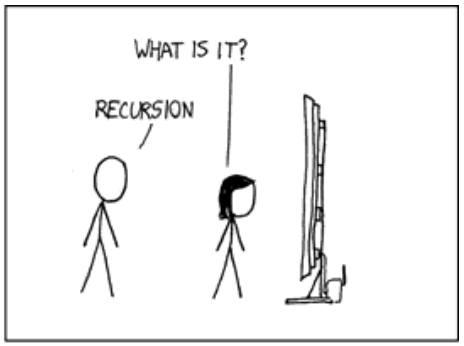
CS319: Scientific Computing

Functions and Quadrature

Dr Niall Madden

Week 4: 5th and 7th, February, 2025





Slides and examples: https://www.niallmadden.ie/2425-CS319

Outline

Slides and examples:
https://www.niallmadden.ie/2425-CS319

Overview of this week's classes

This week, we will study the use of **functions** in C++, which we started (very briefly) at the end of Week 3.

In Scientific Computing, we use the term "**function**" in two different, but related ways:

- A mathematical function, such as $f(x) = e^{-x}$ or $u(x, y) = \sin(\pi x) \cos y$.
- A function we code to preform a task, such a determining if an integer is positive. Or, often, working with mathematical functions: to calculate derivatives at a point (Lab 2) or integrals.

And often we'll combine both ideas!

Overview of this week's classes

However, we'll motivate some of this study with a key topic in Scientific Computing: **Quadrature**, which is also known as **Numerical Integration**.

Later, we'll use this as an opportunity to study the idea of **experimental analysis** of algorithms.

- A **Quadrature** method, in one dimension, is a method for estimating definite integrals. The applications are far too numerous to list, but feature in just about every area of Applied Mathematics, Probability Theory, and Engineering, and even some areas of pure mathematics.
- They are methods for estimating integrals of functions. So this gives us two reasons to code functions:
  (i) As the functions we want to integrate;
  (ii) As the algorithms for doing the integration.

But before we get on to actual methods, we'll learn the basics of writing functions in C++.

Functions

A good understanding of **functions**, and their uses, is of prime importance.

Some functions return/compute a single value. However, many important functions return more than one value, or modify one of its own arguments.

For that reason, we need to understand the difference between **call-by-value** and **call-by-reference** (⟵ later).

Functions

Every C++ program has at least one function: `main()`

Example

```
#include <iostream>
int main(void )
{
   /* Stuff goes here */
   return(0);
}
```

Each function consists of two main parts: Header/Prototype and Body/Definition.

1. Header

The Function "header" or **prototype** gives the function's

- return value data type, or `void` if there is none, and
- parameter list data types or `void` if there are none.
- The header line ends with a semicolon.

The prototype is often given near the start of the file, before the **main()** section.

Syntax for function header:

```
ReturnType FnName (type1, type2, ...);
```

Examples:

2. Function definition

- The **function definition** can be anywhere in the code (after the header).
- First line is the same as the prototype, except variables names need to be included, and that line does not end with a semi-colon.
- That is followed by the body of the function contained within curly brackets.

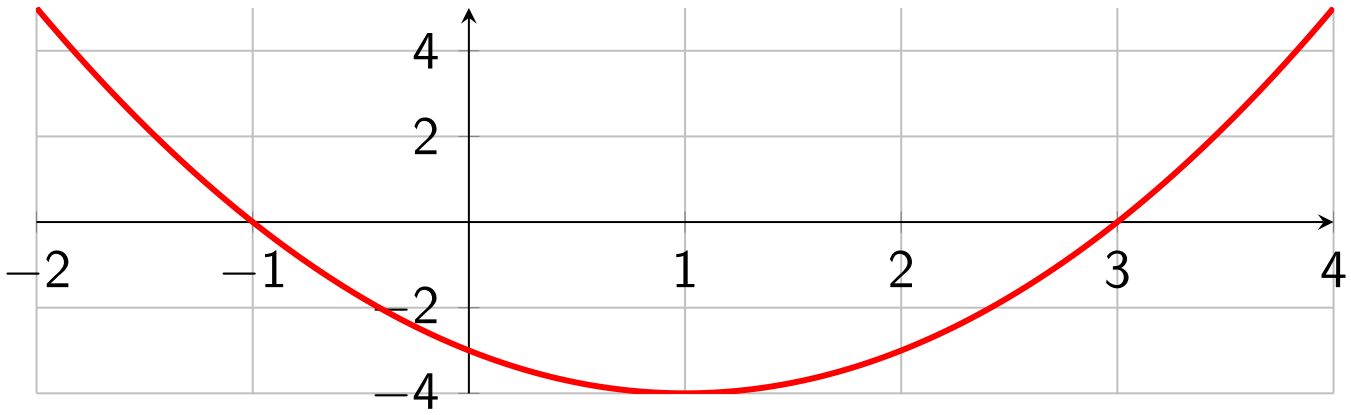
Syntax:

```
ReturnType FnName (type1 param1, type2 param2, ...)
{
      statements
}
```

- `ReturnType` is the data type of the data returned by the function.
- `FnName` the identifier by which the function is called.
- `type1 param1, ...` consists of
  - the data type of the parameter
  - the name of the parameter will have in the function. It acts within the function as a local variable.
- the statements that form the function's body, contained with braces `{...}`.

Since this is a course on scientific computing, we'll often need to define mathematical functions from $\mathbb{R} \to \mathbb{R}$ (more or less), such as $f(x) = e^{-x}$. Typically, such functions map one or more `double`s onto another `double`.

The example we'll look at is $f(x) = x^2 - 2x - 3$.



00MathFunction.cpp

```
#include <iostream>
#include <iomanip>

double f(double x) // x^2 - 2x - 3
{
  return (x*x - 2*x -3);
}

int main(void){
  double x;
  std::cout << std::fixed << std::showpoint;
  std::cout << std::setprecision(2);
  for (int i=0; i<=10; i++)
  {
    x = -1.0 + i*.5;
    std::cout << "f("<< x << ")="<< f(x) << std::endl;
  }
  return(0);
}
```

In this example, we write a function that takes an non-negative integer input and checks it its a composite (`true`) or prime (`false`).

01IsComposite.cpp (header)

```
// 01IsComposite.cpp
// An example of a simple function.

#include <iostream>

bool IsComposite(int i);
```

Calling the `IsComposite` function

`01IsComposite.cpp` (main)

```
 8 int main(void )
   {
10   int i;

12   std::cout << "Enter a natural number: ";
     std::cin >> i;

     std::cout << i << " is a " <<
16     (IsComposite(i) ? "composite":"prime") << " number."
              << std::endl;

     return(0);
20 }
```

Defining the `IsComposite` function

`01IsComposite.cpp` (function definition)

```
   bool IsComposite(int i)
26 {
     int k;
28   for (k=2; k<i; k++)
       if ( (i%k) == 0)
30       return(true);

32   return(false); // If we get to here, i has no divisors between 2 and i-1
   }
```

Most functions will return some value. In rare situations, they don't, and so have a `void` return value.

`02Kth.cpp` (header)

```
// 02Kth.cpp:
// Another example of a simple function.
// Author: Niall Madden
// Date: 05 Feb 2025
// Week 04: CS319 - Scientific Computing
#include <iostream>
void Kth(int i);
```

02Kth.cpp (main)

```
int main(void )
{
  int i;

  std::cout << "Enter a natural number: ";
  std::cin >> i;

  std::cout << "That is the ";
  Kth(i);
  std::cout << " number." << std::endl;

  return(0);
}
```

02Kth.cpp (function definition)

```
// FUNCTION KTH
// ARGUMENT: single integer
// RETURN VALUE: void (does not return a value)
// WHAT: if input is 1, displays 1st, if input is 2, displays 2nd,
// etc.
void  Kth(int i)
{
  std::cout << i;
  i = i%100;
  if ( ((i%10) == 1) && (i != 11))
    std::cout << "st";
  else if ( ((i%10) == 2) && (i != 12))
    std::cout << "nd";
  else if ( ((i%10) == 3) && (i != 13))
    std::cout << "rd";
  else
    std::cout << "th";
}
```

Numerical Integration

Numerical integration is an important topic in scientific computing. Although the history is ancient, it continues to be a hot topic of research, particularly when computing with high-dimensional data.

In this section, we want to estimate definite integrates of one-dimensional functions:

$$\int_a^b f(x)dx.$$

We'll use one of the simplest methods: the Trapezium Rule.

03QuadratureV01.cpp (headers)

```
// 03QuadrateureV01.cpp:
// Trapezium Rule (TR) quadrature for a 1D function
// Author: Niall Madden
// Date: 06 Feb 2025
// Week 04: CS319 - Scientific Computing
#include <iostream>
#include <cmath>   // For exp()

double f(double); // prototype
double f(double x) {  return(exp(x)); } // definition
```

03QuadratureV01.cpp (main)

```
12 int main(void )
   {
14   std::cout << "Using the TR to integrate f(x)=exp(x)\n";
     std::cout << "Integrate f(x) between x=0 and x=1.\n";
16   double a=0.0, b=1.0;
     double Int_f_true = exp(1)-1;
18   std::cout << "Enter value of N for the Trap Rule: ";
     int N;
20   std::cin >> N; // Lazy! Should do input checking.
```

03QuadratureV01.cpp (main continued)

```
22    double h=(b-a)/double(N);
      double Int_f_TR = (h/2.0)*f(a);
24    for (int i=1; i<N; i++)
        Int_f_TR += h*f(a+i*h);
26    Int_f_TR += (h/2.0)*f(b);

28    double error = fabs(Int_f_true - Int_f_TR);

30    std::cout << "N=" << N << ", Trap Rule=" << Int_f_TR
                << ", error=" << error << std::endl;
32    return(0);
  }
```

Typical output:

Next it makes sense to write a function that implements the Trapezium Rule, so that it can be used in different settings.

The idea is pretty simple:

- As before, `f` will be a globally defined function.
- We write a function that takes as arguments `a`, `b` and `N`.
- The function implements the Trapezium Rule for these values, and the globally defined `f`.

04QuadratureV02.cpp (header)

```
// 04QuadrateureV02.cpp: Trapezium Rule as a function
// Trapezium Rule (TR) quadrature for a 1D function
// Author: Niall Madden
// Date: Feb 2025
// Week 04: CS319 - Scientific Computing
#include <iostream>
#include <cmath>   // For exp()
#include <iomanip>

double f(double x) {  return(exp(x)); } // definition
double TrapRule(double a, double b, int N);
```

04QuadratureV02.cpp (main)

```
int main(void )
{
  std::cout << "Using the TR to integrate in 1D\n";
  std::cout << "Integrate between x=0 and x=1.\n";
  double a=0.0, b=1.0;
  double Int_true_f = exp(1)-1; // for f(x)=exp(x)

  std::cout << "Enter value of N for the Trap Rule: ";
  int N;
  std::cin >> N; // Lazy! Should do input checking.

  double Int_TR_f = TrapRule(a,b,N);
  double error_f = fabs(Int_true_f - Int_TR_f);

  std::cout << "N=" << std::setw(6) << N <<
    ", Trap Rule=" << std::setprecision(6) <<
    Int_TR_f << ", error=" <<  std::scientific <<
    error_f << std::endl;
  return(0);
```

04QuadratureV02.cpp (function)

```
34 double TrapRule(double a, double b, int N)
   {
36   double h=(b-a)/double(N);
     double QFn = (h/2.0)*f(a);
38   for (int i=1; i<N; i++)
       QFn += h*f(a+i*h);
40   QFn += (h/2.0)*f(b);
     return(QFn);
42 }
```

Functions as arguments to functions

We now have a function that implements the Trapezium Rule. However, it is rather limited, in several respects. This includes that the function, `f`, is hard-coded in the `TrapRule` function. If we want to change it, we'd edit the code, and recompile it.

Fortunately, it is relatively easy to give the name of one function as an argument to another.

The following example shows how it can be done.

Functions as arguments to functions

05QuadratureV03.cpp(header)

```
// 05QuadrateureV03.cpp: Trapezium Rule as a function
// that takes a function as argument
// Week 04: CS319 - Scientific Computing
#include <iostream>
#include <cmath>   // For exp()
#include <iomanip>

double f(double x) {  return(exp(x)); } // definition
double g(double x) {  return(6*x*x); } // definition

double TrapRule(double Fn(double), double a, double b,
                int N);
```

Functions as arguments to functions

05QuadratureV03.cpp (part of main())

```
20    std::cout << "Which shall we integrate: \n"
                << "\t 1. f(x)=exp(x) \n\t 2. g(x)=6*x^2?\n";
22    int choice;
      std::cin >> choice;
24    while (!(choice == 1 || choice  == 2) )
      {
26      std::cout << "You entered " << choice
                  <<". Please enter 1 or 2: ";
28      std::cin >> choice;
      }
30    double Int_TR=-1; // good place-holder
      if (choice == 1)
32      Int_TR = TrapRule(f,a,b,10);
      else
34      Int_TR = TrapRule(g,a,b,10);

36    std::cout << "N=10" << ", Trap Rule="
                << std::setprecision(6) << Int_TR  << std::endl;
38    return(0);
}
```

Functions as arguments to functions

05QuadratureV03.cpp (TrapRule())

```
42 double TrapRule(double Fn(double), double a,
                  double b, int N)
44 {
     double h=(b-a)/double(N);
46   double QFn = (h/2.0)*Fn(a);
     for (int i=1; i<N; i++)
48     QFn += h*Fn(a+i*h);
     QFn += (h/2.0)*Fn(b);

     return(QFn);
52 }
```

Functions with default arguments

In our previous example, we wrote a function with the header
`double TrapRule(double Fn(double), double a, double b, int N);`

And then we called it as
`Int_TR = TrapRule(f,a,b,10);`

That is, when we were not particularly interested in the value of `N`, we took it to be 10.

It is easy to adjust the function so that, for example, if we called the function as
`Int_TR = TrapRule(f,a,b);`
it would just be assumed that $N = 10$. All we have to do is adjust the function header.

Functions with default arguments

To do, this we specify the value of N in the **function prototype**. You can see this in 05QuadratureV04.cpp. In particular, note Line 10:

06QuadratureV04.cpp (line 10)

```
10 double TrapRule(double Fn(double), double a,
                   double b, int N=10); // default N=10
```

This means that, if the user does not specify a value of N, then it is taken that $N = 10$.

Functions with default arguments

Important:

- You can specify default values for as many arguments as you like. For example:

```
1 double TrapRule(double Fn(double), double a=0.0,
          double b=1.0, int N=10);
```

- If you specify a default value for an argument, you must specify it for any following arguments. For example, the following would cause an error.

```
2 double TrapRule(double Fn(double), double a=0.0,
          double b=1.0, int N);
```

Pass-by-value

In C++ we need to distinguish between

- the value stored in the variable.
- a variable's identifier (might not be unique)
- a variable's (unique) memory address

In C++, if (say) `v` is a variable, then `&v` is the memory address of that variable.

We'll return to this at a later point, but for now we'll check the output of some lines of code that output a memory address.

Pass-by-value

07MemoryAddresses.cpp

```
    int i=12;
10  std::cout << "main: Value stored in i: " << i << '\n';
    std::cout << "main: address of i: " << &i << '\n';
12  Address(i);
    std::cout << "main: Value stored in i: " << i << '\n';
```

Typical output might be something like:

```
main: The value stored in i  is 12
main: The address of i is 0x7ffcd1338314
```

Pass-by-value

A while back we learned that, when we pass a variable as an argument to a function, a new **copy** of the variable is made.

This is called **pass-by-value**.

Even if the variable has the same name in both `main()` and the function called, and the same value, they are different: the variables are **local** to the function (or block) in which they are defined.

We'll test this by writing a function that

- Takes a `int` as input;
- Displays its value and its memory address;
- Changes the value;
- Displays the new value and its memory address.

Pass-by-value

07MemoryAddresses.cpp

```
18 void Address(int i)
   {
20   std::cout << "Address: Value stored in i: " << i << '\n';
     std::cout << "Address: address of i: " << &i << '\n';
22   i+=10; // Change value of i
     std::cout << "Address: New val stored in i: " << i << '\n';
24   std::cout << "Address: address of i: " << &i << '\n';
   }
```

Pass-by-value

Finally, let's call this function:

07MemoryAddresses.cpp

```
    int i=12;
10  std::cout << "main: Value stored in i: " << i << '\n';
    std::cout << "main: address of i: " << &i << '\n';
12  Address(i);
    std::cout << "main: Value stored in i: " << i << '\n';
14  std::cout << "main: address of i: " << &i << '\n';
```

Pass-by-value

In many case, "pass-by-value" is a good idea: a function can change the value of a variable passed to it, without changing the data of the calling function.

But sometimes we **want** a function to be able to change the value of a variable in the calling function.

The classic example is function that

- takes two `int`eger inputs, `a` and `b`;
- after calling the function, the values of `a` and `b` are swapped.

Pass-by-value

08SwapByValue.cpp

```
#include <iostream>
void Swap(int a, int b);

int main(void )
{
  int a, b;

  std::cout << "Enter two integers: ";
  std::cin >> a >> b;

  std::cout << "Before Swap: a=" << a << ", b=" << b
            << std::endl;
  Swap(a,b);
  std::cout << "After Swap: a=" << a << ", b=" << b
            << std::endl;

  return(0);
}
```

Pass-by-value

```
void Swap(int x, int y)
{
  int tmp;

  tmp=x;
  x=y;
  y=tmp;
}
```

This won't work.

We have passed only the *values stored in the variables a and b*. In the `swap` function these values are copied to local variables x and y. Although the local variables are swapped, they remained unchanged in the calling function.

What we really wanted to do here was to use **Pass-By-Reference** where we modify the contents of the memory space referred to by `a` and `b`. This is easily done...

Pass-by-value

...we just change the declaration and prototype from

```
void Swap(int x, int y)  // Pass by value
```

to

```
void Swap(int &x, int &y)  // Pass by Reference
```

the pass-by-reference is used.

Exercises

Exercise (Simpson's Rule)

- *Find the formula for Simpson's Rule for estimating* $\int_a^b f(x)dx$.
- *Write a function that implements it.*
- *Compare the Trapezium Rule and Simpson's Rule. Which appears more accurate for a given* N*?*

Exercise

Change the `Address()` *function in* `07MemoryAddresses.cpp` *so that the variable* `i` *is passed by reference.*
How does the output change?