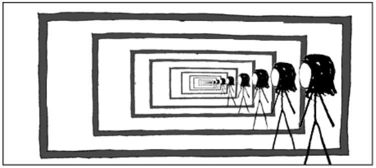
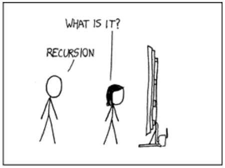
CS319: Scientific Computing

Quadrature, and Functions in C++

Dr Niall Madden

Week 4: **9am and 4pm**, 31 January, 2024

Slides and examples: https://www.niallmadden.ie/2324-CS319

Class times

	Mon	Tue	Wed	Thu	Fri
9 – 10			✓	LAB	
10 – 11					
11 – 12					LAB
12 – 1					LAB
1 – 2					
2 – 3					
3 – 4					
4 – 5			✓		

Reminder: **labs again this week** (and every week).

- Thursday 9-10
- Friday 11-12
- Friday 12-1.

For more, see https://www.niallmadden.ie/2324-CS319/#labs

Outline

Slides and examples:
https://www.niallmadden.ie/2324-CS319

Overview of this week's classes

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

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 review some notes from last week, and examples I didn't get to.

Functions (again)

In Week 3 we studied how to write functions in C++.

- Each function consists of two main parts: a **header** (or "prototype") and the function definition.
- The "header" is a single line of code that appears (usually) before the `main()` function. It 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 parameter list can include variable names, but they are treated as comments.
- The header line ends with a semicolon.

Syntax for function header:

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

Functions (again)

Examples:

Functions (again)

- 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.

Functions (again)

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 `{...}`.

Functions (again)

00IsComposite.cpp (header)

```
// 00IsComposite.cpp
// An example of a simple function.
// Author: Niall Madden
// Date: 31 Jan 2024
// Week 4: CS319 - Scientific Computing

#include <iostream>

bool IsComposite(int i);
```

Functions (again)

00IsComposite.cpp (main)

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

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

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

  return(0);
}
```

Functions (again)

00IsComposite.cpp (function definition)

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

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

Functions (again)

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

`01Kth.cpp` (header)

```cpp
// 01Kth.cpp:
// Another example of a simple function.
// Author: Niall Madden
// Date: 31 Jan Feb 2024
// Week 04: CS319 - Scientific Computing
#include <iostream>
void Kth(int i);
```

Functions (again)

01Kth.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);
}
```

Functions (again)

01Kth.cpp (function definition)

```cpp
// 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.

Numerical Integration

02QuadratureV01.cpp (headers)

```
// 02QuadrateureV01.cpp:
// Trapezium Rule (TR) quadrature for a 1D function
// Author: Niall Madden
// Date: 31 Jan 2024
// Week 04: CS319 - Scientific Computing
#include <iostream>
#include <cmath>   // For exp()

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

Numerical Integration

02QuadratureV01.cpp (main)

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

Numerical Integration

02QuadratureV01.cpp (main continued)

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

  double error = fabs(Int_f_true - Int_f_TR);

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

Numerical Integration

Typical output:

Trapezium Rule as a function

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`.

03QuadratureV02.cpp(header)

```
// 03QuadrateureV03.cpp: Trapezium Rule as a function
// Trapezium Rule (TR) quadrature for a 1D function
// Author: Niall Madden
// Date: 31 Jan Feb 2024
// 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);
```

Numerical Integration — Trapezium Rule as a function

03QuadratureV02.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);
```

Numerical Integration — Trapezium Rule as a function

03QuadratureV02.cpp(function)

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

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

04QuadratureV04.cpp(header)

```
// 04QuadrateureV03.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

04QuadratureV04.cpp (part of main())

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

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

Functions as arguments to functions

04QuadratureV04.cpp (TrapRule())

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

  return(QFn);
}
```

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:

`05QuadratureV04.cpp` (line 10)

```
10 double TrapRule(double Fn(double), double a,
                   double b, int 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:

```
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.

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

Exercise

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.