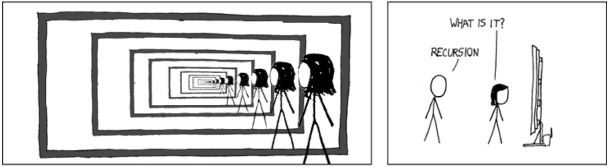
0. Annotated slides from Friday

CS319: Scientific Computing

Functions and Quadrature

Dr Niall Madden

Week 4: 4th and 6th, February, 2026



Slides and examples: https://www.niallmadden.ie/2526-CS319

0. Outline

1. Overview of this week's classes
2. Numerical Integration (again)
3. The Trapezium Rule algorithm
   - The code
4. V02: Trapezium Rule as a function
5. V03: Functions as arguments to functions
6. ~~V04: Functions~~ with default arguments
7. Pass-by-value
8. Function overloading
   - Detailed example
9. Exercises

Slides and examples:
https://www.niallmadden.ie/2526-CS319

7. Pass-by-value

In C++ we need to distinguish between

"identifier" = " name "

- 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.

7. Pass-by-value

01MemoryAddresses.cpp

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

a = 10
b = 11
⋮
f = 15

0x means "Hexadecimal".
-base 16.

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f

7. Pass-by-value

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.

7. Pass-by-value

01MemoryAddresses.cpp

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

7. Pass-by-value

Finally, let's call this function:

01MemoryAddresses.cpp

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

7. 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. (Another important case use is if that data is stored in a very large array which we don't want to duplicate).

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.

7. Pass-by-value

02SwapByValue.cpp

```cpp
#include <iostream>
void Swap(int a, int b); // swap values of a and 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); // ⟵ how to code this ??
  std::cout << "After Swap: a=" << a << ", b=" << b
            << std::endl;

  return(0);
}
```

7. 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...

7. 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.

We'll do that presently, but fist an example of the effect of using &...

7. Pass-by-value

Example

03PassByValueAndReference.cpp

```cpp
void DoesNotChangeVar(int X);
void DoesChangeVar(int &X);

int main(void)
{
  int q=34;
  std::cout << "main: q=" << q << std::endl;
  std::cout << "main: Calling DoesNotChangeVar(q)...";
  DoesNotChangeVar(q);
  std::cout << "\t Now q=" << q << std::endl;
  std::cout << "main: Calling DoesChangeVar(q)...";
  DoesChangeVar(q);
  std::cout << "\t And now q=" << q << std::endl;
  return(0);
}

void DoesNotChangeVar(int X){  X+=101; }
void DoesChangeVar(int &X){  X+=101; }
```

7. Pass-by-value

Output

```
main: q=34
main: Calling DoesNotChangeVar(q)...      Now q=34
main: Calling DoesChangeVar(q)...    And now q=135
```

8. Function overloading

C++ has certain features of **polymorphism**: where a single identifier can refer to two (or more) different things. A classic example is when two different functions can have the same name, but different argument lists.

This is called **function overloading**.

There are lots of reasons to do this. For example, we earlier had a function called `Swap()` that swapped the value of two `int` variables.

However, we can write a function that is also called `Swap()` to swap two `float`s, or two `string`s.

(Note: this can also be done with something called `templates`: we'll look at that in a few weeks.)

8. Function overloading

As a simple example, we'll write two functions with the same name: one that swaps the values of a pair of ints, and that other that swaps a pair of floats. (Really this should be done with templates...)

04Swaps.cpp (headers)

```
#include <iostream>

// We have two function prototypes with same name!
void Swap(int &a, int &b);   // note use of references
void Swap(float &a, float &b);
```

8. Function overloading

04Swaps.cpp (main)

```cpp
int main(void){
    int a, b;
    float c, d;

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

    std::cout << "a=" << a << ", b=" << b <<
      ", c=" << c << ", d=" << d << std::endl;
    std::cout << "Swapping ...." << std::endl;

    Swap(a,b);
    Swap(c,d);

    std::cout << "a=" << a << ", b=" << b <<
      ", c=" << c << ", d=" << d << std::endl;
    return(0);
}
```

8. Function overloading

04Swaps.cpp (functions)

```cpp
// Swap(): swap two ints
void Swap(int &a, int &b)
{
  int tmp;

  tmp=a;
  a=b;
  b=tmp;

}

// Swap(): swap two floats
void Swap(float &a, float &b)
{
  float tmp;

  tmp=a;
  a=b;
  b=tmp;
}
```

8. Function overloading

What does the compiler take into account to distinguish between overloaded functions?

C++ distinguishes functions according to their **signature**. A signature is made up from:

- **Type of arguments**. So, e.g., `void Sort(int, int)` is different from `void Sort(char, char)`.
- **The number of arguments**. So, e.g., `int Add(int a, int b)` is different from `int Add(int a, int b, int c)`.

Examples:

8. Function overloading

However, the following to not impact signatures:

- **Return values**. For example, we cannot have two functions `int Convert(int)` and
  `float Convert(int)`
  since they have the same argument list.
- **user-defined types** (using `typedef`) that are in fact the same. See, for example, [redacted]`OverloadedConvert.cpp`.
- **References**: we cannot have two functions
  `int MyFunction(int x)` and
  `int MyFunction(int &x)`

Also, having different variable names is not enough to distinguish.
That is, having
   int MyFunc(int x);   and
   in MyFunc(int ABC);
would not be allowed.

8. Function overloading

Detailed example

In the following example, we combine two features of C++ functions:

- Pass-by-reference,
- Overloading,

We'll write two functions, both called `Sort`:

- `Sort(int &a, int &b)` – sort two integers in ascending order.
- `Sort(int list[], int n)` – sort the elements of a list of length n.

The program will make a list of length 8 of random numbers between 0 and 39, and then sort them using **bubble sort**.

8. Function overloading

Detailed example

05Sort.cpp (headers)

```
#include <iostream>
#include <stdlib.h> // contains rand() header

const int N=8; ← ignore for now

void Sort(int &a, int &b);
void Sort(int list[], int length);
void PrintList(int x[], int n);
```

8. Function overloading

Detailed example

05Sort.cpp (main)

```cpp
int main(void )
{
    int i, x[N];

    for (i=0; i<N; i++)
      x[i]=rand()%40;

    std::cout << "The list is:\t\t";
    PrintList(x, N);
    std::cout << "Sorting..." << std::endl;

    Sort(x,N);

    std::cout << "The sorted list is:\t";
    PrintList(x, N);
    return(0);
}
```

8. Function overloading

Detailed example

05Sort.cpp (Sort two ints)

```cpp
// Sort(a, b)
// Arguments: two integers
// return value: void
// Does: Sorts a and b so that a <= b.
void Sort(int &a, int &b)
{
  if (a>b)
  {
    int tmp;
    tmp=a;     a=b;     b=tmp;
  }
}
```

8. Function overloading

Detailed example

05Sort.cpp (Sort list)

```
// Sort(int [], int)
// Arguments: an integer array and its length
// return value: void
// Does: Sorts the first n elements of x
void Sort(int x[], int n)
{
  int i, k;
  for (i=n-1; i>1; i--)
    for (k=0; k<i; k++)
      Sort(x[k], x[k+1]);

}
```

8. Function overloading

Detailed example

```cpp
void PrintList(int x[], int n)
{
  for (int i=0; i<n; i++)
    std::cout << x[i] << "  ";
  std::cout << std::endl;
}
```

9. 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* `01MemoryAddresses.cpp` *so that the variable* `i` *is passed by reference.*
How does the output change?