CS319: Scientific Computing

Function Overloading and Memory Allocation

Dr Niall Madden

Week 5: 12th and 14th of February, 2025

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

0. Outline

1. Recall: Pass-by-value
2. Function overloading
3. Detailed example
4. Arrays
5. Pointers
   - Pointer arithmetic
   - Warning!
6. Dynamic Memory Allocation
   - `new`
   - `delete`
7. Example: Quadrature 1

Slides and examples:

https://www.niallmadden.ie/2425-CS319

1. Stuff...

See announcement...

1. Lab 2: due tomorrow at 10.
2. Class test - next week!

Next Friday

"Sample paper on Monday morning"

6. Pointers

To properly understand how to use arrays, we need to study **Pointers**.

- We already learned that if, say, `x` is a variable, then `&x` is its memory address.
- A **pointer** is a special type of variable that can store memory addresses. We use the `*` symbol before the variable name in the declaration.
- For example, if we declare
  `int i;` i is of type int
  `int *p` → p is of type "pointer to int"
  then we can set `p=&i`.

  p can store the memory address of i

6. Pointers

05Pointers.cpp

```
10   int a=-3, b=12;
     int *where;

     std::cout << "The variable 'a' stores " << a << std::endl;
14   std::cout << "The variable 'b' stores " << b << std::endl;
     std::cout << "'a' is stored at address " << &a << std::endl;
16   std::cout << "'b' is stored at address " << &b << std::endl;

18   where = &a;
     std::cout << "The variable 'where' stores "
20             << (void *) where << std::endl;
     std::cout << "... and that in turn stores " <<
22     *where << '\n';
```

The variable 'a' stores -3 ← From line 13
The variable 'b' stores 12
'a' is stored at address 0x7ffd192c6a48 (line 15)
'b' is stored at address 0x7ffd192c6a4c
The variable 'where' stores 0x7ffd192c6a48
... and that in turn stores -3

One can actually do calculations on memory addresses. This is called **pointer arithmetic**. One can't (for example) add two addresses, or compute their product, but you can, for example, increment them.

06PointerArithmetic.cpp

```cpp
int vals[3];
vals[0]=10;  vals[1]=8;  vals[2]=-4;

int *p;
p = vals;

for (int i=0; i<3; i++)
{
  std::cout << "p=" << p << ", *p=" << *p << "\n";
  p++;
}
```

Recall "vals" is the base (memory) address of the array.

p++; → pointer increment: p=p+1

p=0x7ffcfe3df84c, *p=10
p=0x7ffcfe3df850, *p=8
p=0x7ffcfe3df854, *p=-4

Being able to manipulate memory addresses is one of the reasons C++ is considered a very **powerful** language. It is possible to preform (low-level) operations in C++ that are impossible in, say, Python.

But it is also possible to write programmes that will crash, or even crash your computer, since memory addresses are not well protected.

7. Dynamic Memory Allocation

In all examples we've had so far, we've specified the size of an array at the time it is defined.

In many practical cases, we don't have that information. For example, we might need to read data from a file, but not know the file size in advance.

It would be useful if, on the fly, we could set the size of an array.

Furthermore, for efficiency, we may want to free up memory allocated.

To add this functionality, we will use two new (to us) C++ operators for dynamic memory allocation and deallocation:

- `new` and
- `delete`.

(There are also functions `malloc()`, `calloc()` and `free()` inherited from C, but we won't use them).

The `new` operator is used in C++ to allocate memory. The basic form is

var = new *type*

where *type* is the specifier of the object for which you want to allocate memory and *var* is a pointer to that type.

If insufficient memory is available then `new` will return a NULL pointer or generate an exception.

To dynamically allocate an array:

- First declare a pointer of the right type:
  ```
  int *data;
  ```
- Then use `new`
  ```
  data = new int[MAX_SIZE];
  ```

New returns a memory address which is stored in "data".

When it is no longer needed, the operator `delete` releases the memory allocated to an object.

To "delete" an array we use a slightly different syntax:

```
delete [] array;
```

where *array* is a pointer to an array allocated with `new`.

8. Example: Quadrature 1

In Week 4, we introduced the idea of **numerical integration** or **quadrature**.

We computed estimates for $\int_a^b f(x)dx$ by applying the Trapezium Rule:

- Choose the number of intervals N, and set $h = (b-a)/N$.
- Define the quadrature points $x_0 = a$, $x_1 = a + h$, $\ldots x_N = b$. In general, $x_i = a + ih$. note: $x_N = a + \frac{N}{n} = a + \frac{N}{N}(b-a) = b$
- Set $y_i = f(x_i)$ for $i = 0, 1, \ldots, N$.
- Compute $\displaystyle\int_a^b f(x)dx \approx Q_1(f) := h\Big(\frac{1}{2}y_0 + \sum_{i=1:(N-1)} y_i + \frac{1}{2}y_N\Big).$

[Take notes for the next few slides]

Here we use MATLAB notation:

1:(N-1) is 1, 2, 3, ... N-1

8. Example: Quadrature 1

07TrapeziumRule.cpp

```
#include <iostream>
#include <cmath>   // For exp()
#include <iomanip>

double f(double x) {  return(exp(x)); } // definition
double ans_true = exp(1.0)-1.0; // true value of integral

double Quad1(double *x, double *y, unsigned int N);
```

$f(x) = e^x$ is the function to be integrated.

Prototype for Quad1();

It takes 3 arguments:

- pointer to double, called x

→ stores base address of a array of points.

- pointer to double, called y
- An integer, N.

Next we skip to the function code...

N is number of intervals.

07TrapeziumRule.cpp

```
double Quad1(double *x, double *y, unsigned int N)
{
  double h = (x[N]-x[0])/double(N);
  double Q = 0.5*(y[0] + y[N]);
  for (unsigned int i=1; i<N; i++)
    Q += y[i];
  Q *= h;
  return(Q);
}
```

x and y are of length N+1.

Source of confusion: * is used in two very different contexts here.

- we use it for multiplication, eg Q*=h (which is Q=Q*h)
- in header, to give that x & y are pointers.

8. Example: Quadrature 1

Next we skip to the function code...

07TrapeziumRule.cpp

```
double Quad1(double *x, double *y, unsigned int N)
{
  double h = (x[N]-x[0])/double(N);
  double Q = 0.5*(y[0] + y[N]);
  for (unsigned int i=1; i<N; i++)
    Q += y[i];
  Q *= h;
  return(Q);
}
```

Source of confusion: * is used in two very different contexts here.

Note : *x same as x[0]

and y[i] same as *(y+i)

8. Example: Quadrature 1

Back to the main function: declare the pointers, input N, and allocate memory.

07TrapeziumRule.cpp

```
int main(void )
{
  unsigned int N;
  double a=0.0, b=1.0; // limits of integration
  double *x; // quadrature points
  double *y; // quadrature values

  std::cout << "Enter the number of intervals: ";
  std::cin >> N; // not doing input checking

  x = new double[N+1];
  y = new double[N+1];
```

DMA .

Initialise the arrays, compute the estimates, and output the error.

07TrapeziumRule.cpp

```
double h = (b-a)/double(N);
for (unsigned int i=0; i<=N; i++)
{
  x[i] = a+i*h;
  y[i] = f(x[i]);
}
double Est1 = Quad1(x,y,N);
double error = fabs(ans_true - Est1);
std::cout << "N=" << N << ", Trap Rule="
          << std::setprecision(6) << Est1
          << ", error=" <<  std::scientific
          << error << std::endl;
```

initialise x & y.

→ call Quad1().

8. Example: Quadrature 1

Finish by de-allocating memory (optional, in this instance).

07TrapeziumRule.cpp

```
  delete [] x;
  delete [] y;
  return(0);
}
```

8. Example: Quadrature 1

Although this was presented as an application of using arrays in C++, some questions arise...

1. What value of N should we pick the ensure the error is less than, say, 10^{-6}?
   , of N.
2. How could we predict that value if we didn't know the true solution?
3. What is the smallest error that can be achieved in practice? Why?
4. How does the time required depend on N? What would happen if we tried computing in two or more dimensions?
5. Are there any better methods? (And what does "better" mean?)

8. Example: Quadrature 1

Some answers to those questions.

Answer to Q5:

A better method, such as
Simpson's Rule
would give smaller Error
for some cost.