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

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

1. Recall: Pass-by-value

Last week we learned the following about `C++`:

- By default, an argument is **passed by value**. This means that the function gets a copy of the variable. So any changes to it are local to the function.
- If (say) `v` is a variable, then `&v` is (a reference to) the memory address of that variable.
- To pass the variable `v`'s **reference** to a function, refer to it as `&v` in the function header/prototype and definition.
- If a variable is passed by reference to a function, f, and its value changed in f, then it is also changed in the calling function.

1. Recall: Pass-by-value

Example

00PassByValueAndReference.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; }
```

Output

2. 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, in Week 4 we wrote 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.)

2. Function overloading

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

`01Swaps.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);
```

01Swaps.cpp (main)

```
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);
}
```

01Swaps.cpp (functions)

```
// 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;
}
```

2. 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:

2. 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, `02OverloadedConvert.cpp`.
- **References**: we cannot have two functions
  `int MyFunction(int x)` and
  `int MyFunction(int &x)`

3. 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**.

3. Detailed example

03Sort.cpp (headers)

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

const int N=8;

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

3. Detailed example

03Sort.cpp (main)

```
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);
}
```

03Sort.cpp (Sort two ints)

```
// 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;
  }
}
```

3. Detailed example

03Sort.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]);

}
```

3. Detailed example

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

4. Arrays

Much of Scientific Computing involves working with data, and often collections of data are stored as **arrays**, which are list-like structures that stores a collection of values all of the same type.

Example: declare an array to store five floats:

```
float vals[5];
vals[0]=1.0;
vals[1]=2.1;
vals[2]=3.14;
vals[3]=-21.0;
vals[4]=-1.0;
```

4. Arrays

Consider the following piece of code:

04Array.cpp

```
  float vals[3];
  vals[0]=1.1;  vals[1]=2.2;  vals[2]=3.3;
  for (int i=0; i<3; i++)
    std::cout << "  vals["<<i<<"]=" << vals[i];
  std:: cout << std::endl;
  std::cout << "vals=" << vals << '\n';
```

The output I get looks like

```
  vals[0]=1.1  vals[1]=2.2  vals[2]=3.3
vals=0x7ffd9ab8ec9c
```

Can we explain the last line of output?

4. Arrays

So now it know that, if `vals` is the name of an array, then in fact the value stored in `vals` is the memory address of `vals[0]`.

We can check this with

```
std::cout << "vals=" << vals << '\n';
std::cout << "&vals[0]=" << &vals[0] << '\n';
std::cout << "&vals[1]=" << &vals[1] << '\n';
std::cout << "&vals[2]=" << &vals[2] << '\n';
```

For me, this gives

```
    vals=0x7ffc932b960c
&vals[0]=0x7ffc932b960c
&vals[1]=0x7ffc932b9610
&vals[2]=0x7ffc932b9614
```

Can we explain?

4. Arrays

And in the same piece of code, if I changed the first line from

`float vals[3];`

to

`double vals[3];`

we get something like

```
    vals=0x7ffd361abdc0
&vals[0]=0x7ffd361abdc0
&vals[1]=0x7ffd361abdc8
&vals[2]=0x7ffd361abdd0
```

Can we explain?

4. Arrays

So now we understand why C++ (and related languages) index their arrays from `0`:

- `vals[0]` is stored at the address in `vals`;
- `vals[1]` is stored at the address after the one in `vals`;
- `vals[k]` is stored at the kth address after the one in `vals`;

But there are numerous complications, not least that different data types are stored using different numbers of bytes. So the off-set depends on the data type.

To understand the subtleties, we need to know about **pointers**.

5. 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;
  int *p
  ```

  then we can set `p=&i`.

05Pointers.cpp

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

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

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

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

```
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++;
}
```

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.

6. 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];
```

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`.

7. 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 $x_0 = a$, $x_1 = a + h$, $\dots x_N = b$. In general, $x_i = a + ih$.
- Set $y_i = f(x_i)$ for $i = 0, 1, \dots, N$.
- 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]

7. 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);
```

7. 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.

7. 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];
```

7. Example: Quadrature 1

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;
```

7. Example: Quadrature 1

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

07TrapeziumRule.cpp

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

7. 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}?
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?)

7. Example: Quadrature 1

Some answers to those questions.