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!

2. 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.

2. Recall: Pass-by-value

Example

00PassByValueAndReference.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; }
```

2. Recall: Pass-by-value

Output

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

3. 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.)

3. 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);
```

3. Function overloading

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);
}
```

3. Function overloading

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;
}
```

3. 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:

3. 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)`

4. 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**.

4. 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);
```

Ignore for now.

4. 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);
}
```

x is an integer array: x[0], x[1],..., x[N-1].

rand() is a pseudo-random number between 0 and "RAND_MAX".

So rand()%40 is a random int between 0 & 39.

4. Detailed example

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;
  }
}
```

4. 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]);

}
```

x = {5, 0, 3, 7, 6}

n = 5

Step 1 : i = 4.

Step 1.1 k = 0

Sort x[0], x[1]

x = {0, 5, 3, 7 6}

Step 1.2 sort x[1], x[2] x = {0, 3, 5, 7, 6}

Step 1.3 " x[2], x[3] x = {0, 3, 5, 7, 6}.

Step 1.4 " x[3] x[4] x = {0, 3, 5 6, 7}.

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

The list is: 23 6 17 35 33 15 26 12
Sorting...
The sorted list is: 6 12 15 17 23 26 33 35

5. 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;
```

5. 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?

It is a hexadecimal number representing a memory address.

5. 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
```

These are the same, and are the memory address of vals[0], the first item in the array.

Can we explain?

There is a difference of 4 between addresses.

5. 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
```

diference of 8

Can we explain?

because a double is stored in 8 bytes (= 64 bits).

5. 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**.

Finished here Wed @ 5pm.