CS319: Scientific Computing

Functions: overloading and pass-by-reference

Dr Niall Madden

Week 5: **9am**, 7 February, 2024

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

Outline

1. Pass-by-value
2. Function overloading
3. Detailed example
4. Arrays and memory allocation
   - Arrays
   - Pointers
5. Dynamic Memory Allocation
   - `new`
   - `delete`

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

Pass-by-value

In C++ we need to distinguish between

- 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.

int val = 24;

variable identifier

value

Pass-by-value

00MemoryAddresses.cpp

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

Typical output might be something like:

value stored in i

```
main: The value stored in i  is 12
main: The address of i is 0x7ffcd1338314
```

value of &i,
ie memory address of i

Address is stored in hexadecimal ("hex")., i.e, base 16.

Pass-by-value

A while back we learned that, 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.

Pass-by-value

00MemoryAddresses.cpp

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

Typical output:

Line	
20	Address: Value stored in i: 12
21	Address: address of i: 0x7ffc471e18ac
23	Address: New val stored in i: 22
24	Address: address of i: 0x7ffc471e18ac

memory address does not change.

Pass-by-value

Finally, let's call this function:

00MemoryAddresses.cpp

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

10+11
main: Value stored in i: 12
main: address of i: 0x7ffc471e18c4

12
Address: Value stored in i: 12
Address: address of i: 0x7ffc471e18ac
Address: New val stored in i: 22
Address: address of i: 0x7ffc471e18ac

13+4
main: Value stored in i: 12
main: address of i: 0x7ffc471e18c4

different memory addresses.

Value of i changed in Address, but not main()

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.

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.

Pass-by-value

01SwapByValue.cpp

```
#include <iostream>
void Swap(int a, int 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);
  std::cout << "After Swap: a=" << a << ", b=" << b
            << std::endl;

  return(0);
}
```

← tries to swap values in a & b.

```
                         5       10
void Swap(int x, int y)
{
  int tmp;
   5
  tmp=x;
 10
  x=y;
 5
  y=tmp;
}
```

x	y	tmp
5	10	5
10	10	5
10	5	5

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...

~~x = y~~
~~y = x~~

Pass-by-value

...we just change the declaration and prototype from

```
void Swap(int x, int y) // Pass by value
```

pass variables

to

```
void Swap(int &x, int &y) // Pass by Reference
```

pass memory address

the pass-by-reference is used.

Pass-by-value

Exercise

Change the `Address()` function in `00MemoryAddresses.cpp` so that the variable `i` is passed by reference.
How does the output change?

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, just now we wrote a function called `Swap()` that swapped the value of two `int` variables. But suppose we wanted to write a function that swapped two `float`s, or two `string`s. Would we have to give a different name to each function? No!

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...)

02Swaps.cpp

```
#include <iostream>

// We have two function prototypes!
void Swap(int &a, int &b);
void Swap(float &a, float &b);
```

Function overloading

02Swaps.cpp (continued)

```
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);
```

Function overloading

02Swaps.cpp (continued)

```
void Swap(int &a, int &b)
{
  int tmp;

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

void Swap(float &a, float &b)
{
  float tmp;

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

Function overloading

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

C++ takes the following into account: "function signature".

- **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)`.

But not

- **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, `03OverloadedConvert.cpp`.

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**.

Detailed example

04Sort.cpp (i)

```
#include <iostream>
#include <stdlib.h>

const int N=8;  ← ignore this

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

Detailed example

04Sort.cpp (ii)

```
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);
}
```

Returns a very large (pseudo) random int.

Detailed example

04Sort.cpp (iii)

```
// 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;
  }
}
```

if a > b swap their values.

Detailed example

04Sort.cpp (iii)

```
// Arguments: an integer array and its length
// return value: void
// Does: Sorts the 1st 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]);
}
```

23 6 17 35 33 15 26 12

6 23 17 35 33 15 26 12

6 17 23 35 33 15 26 12

6 17 23 35 33 15 26 12

6 12 23 33 35 15 26 12

Detailed example

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

Finished here 10am

Arrays and memory allocation

A while back we mentioned that one can create a list-type variable that stores a collection of values all of the same type. In C++ this is called an **array**.

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;
```

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

- We already learned that if, say, `var` is a variable, then `&var` 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 *p`
  then we can set `p=&var`.

05Pointers.cpp

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

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

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

Dynamic Memory Allocation

In many practical cases, we may not know the size of an array at the time it is declared. 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).

Dynamic Memory Allocation `new`

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`.