CS319: Scientific Computing

Week 6: Pointers, Arrays, and Quadrature again

Dr Niall Madden

9am and **4pm**, 14 February, 2024

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

Annotated slides from 9am class.

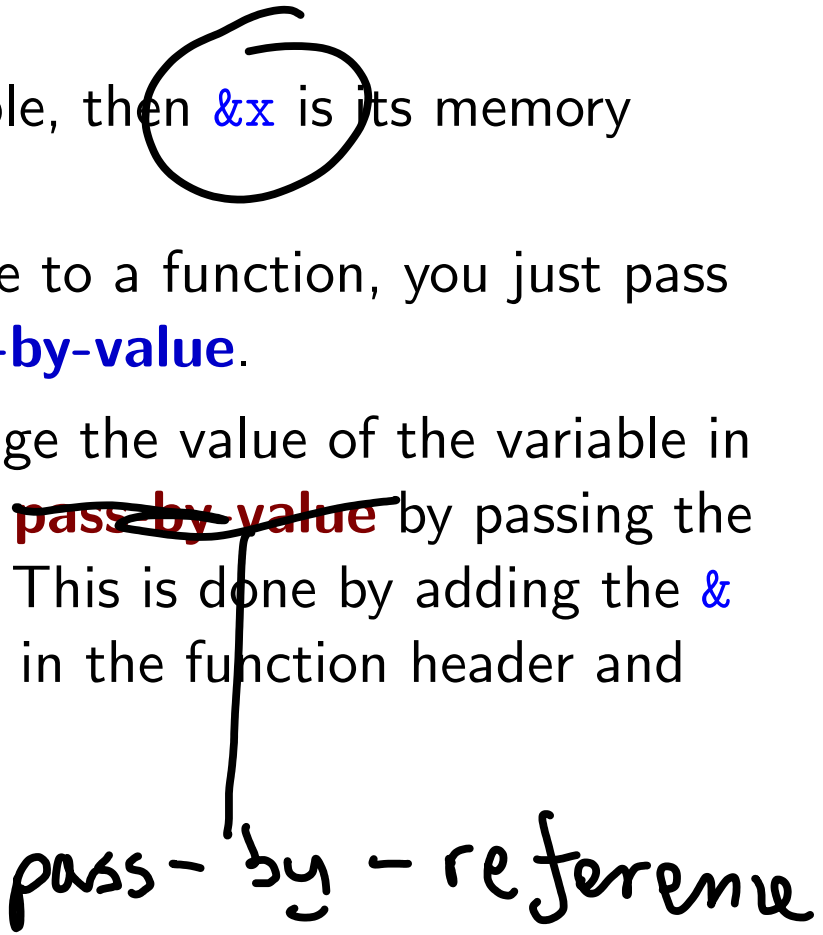
Outline

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

Recall: memory addresses

In Week 5 we learned that...

- If (for example) `x` is some variable, then `&x` is its memory address.
- Usually, when you pass a variable to a function, you just pass a local copy. This is called **pass-by-value**.
- If you want the function to change the value of the variable in the calling function, you have to ~~pass-by-value~~ by passing the memory address of the variable. This is done by adding the `&` symbol before the variable name in the function header and definition.



"reference" = "memory address"

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:

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

1: vals is an array that stores five floats.

Each element of the array is a single variable.

Arrays

Consider the following piece of code:

00Array.cpp

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

The output I get looks like

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

"0x" = "hex" This is a memory address, but of what?

Can we explain the last line of output?

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

Hex	dec
10	10
1a	11
1b	12
1c	13
1e	14
1f	15
20	16

For me, this gives

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

→ Same!!

0c, 0d, 0e, 0f → A single float

10

→ start of vals[0]

→ start of vals[1]

Can we explain?

Each address is of a single byte
And a float is stored in 4 bytes

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
```

difference of 8
" of 8

Can we explain?

A double is stored in 8 bytes, so there is a difference of 8 in each successive memory address.

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 *k*th 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 between addresses changes.

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

vals[k] means "vals plus an offset of k"

Also, right now I know how to find the address of a variable.
But how do we find out what value is stored at a given address???

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`.

Think of this as p is of type int*

Ok to write

```
int* p;
```

but not

```
int* p, q;
```

Pointers

01Pointers.cpp

```
10  int a=-3, b=12;
    int *where;  // where is a pointer to type int.

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

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

If "where" is a pointer storing the memory address of an int, then "*where" evaluates as the value stored there.

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. [Finished here at 10am]

In particular, if
int vals[3], *p;
vals[0]=1.1; vals[1]=2.222;

We can set
p=vals;

Then "*p" evaluates as 1.1 and "*(p+1)" as 2.222;
because "p+1" is the "memory address after p".
where the compiler works out the number of bytes for the data type.