SUPA COO (C++)
Lecture 3 – 29 Oct 2015

Dr. Adrian Buzatu
University of Glasgow, adrian.buzatu@glasgow.ac.uk

Thanks to to some usage of previous lecturer's materials
(S. Allwood-Spiers & W. H. Bell)

Reference

double a=3.0;
Creates a new object of type double, name a, and value 3.0.
a=5.0;
The same object at the same address has a new value 5.0.

const double a=3.0;
Creates a new object of type double, name a, and value 3.0.
a=5.0;
Compilation error. We are not allowed to change its value.

double& r=a;
No new object created. r must be initialized to an existing object.
r=5.0;
Changes the value of a to 5. . r usage syntax as if instead of r is a.

Pass arguments to functions by value

void change (double a) { a = 7.0;}
double a = 3.0;
change(a);
std::cout<<a<<std::endl; What do we see?
Answer: 3!
An object is created in memory inside the function by copying the original object a, by cloning it. also called a, also with a value 3.0. but a different object. That object is changed from 3.0 to 7.0. That object is then destroyed when the function gets out of scope. So no effect on the original a.

If a is a very large object (not double as here), then this takes time, making C++ running slower!

Pass arguments to functions by reference

void change (double& a) { a = 7.0;}
double a = 3.0;
change(a);
std::cout<<a<<std::endl; What do we see?
Answer: 7!
We passed the reference to a, meaning the address in memory where a resides, which means we change a!

Pass arguments to functions by reference

void change (double& a, double& b) { a = 7.0; b=8.0;}
double a = 3.0;
Double b = 4.0;
change(a,b);
std::cout<<a<<" "<<b<<std::endl; What do we see?
Answer: 7.0 8.0
We can change to or more variables. That is when we want our function to return several variables changes.

Pass arguments to functions by reference

void change (double& a, double& b) { a = 7.0; b=8.0;}
double a = 3.0;
Double b = 4.0;
change(a,b);
std::cout<<a<<" "<<b<<std::endl; What do we see?
Answer: 7.0 8.0
We can change to or more variables. That is when we want our function to return several variables changes.

Pass args to functions by const reference

```
void average (const double& a, const double& b)
{
    double average=(a+b)/2;
    std::cout<<"average="<<average<<std::endl;
}
```

double a=3; double b=4; average(a,b);

If we don't want to change a and b, force the reference to be to a const object. For double, int, bool, does not make a difference on the speed.

But for larger objects we create in our own class, it does.

```
void read_string (const std::string& s)
{ std::cout<<s<<std::endl; }
```

Pointers

double p= new double(3.0);*
First creates reserves memory for a double and fills it with the value 3.0. So creates a new object of type double, with no name, and the value 3.0.
Then creates a new object o type “double*” (not “double” !), called p which has the value 0xfg… (the address of the first created object).

Access that object by *p, like change its value.
**p = 4.0;*
*std::cout<<*p<<std::endl;*

We can make a pointer point to a new object!

This is not possible for references, which are stuck to the object they were defined to!

double a = 6.0;

What is the address of a? It is “&a”, and looks like 0x…

Now let’s point p to the new object a.

p=&a;

Now if I do

**p=5.0;*

I have changed the value of a, as *p is the object located in the memory of the computer at the address represented by the value of p, which is thanks to p=&a, the address of a, so in other words p points to a!

std::cout<<a<<std::endl;

Answer 5:

If we want the object to never be changed

If I know I don't want to change the object, then I force the reference or point to be to an object of the type const.

```
const double a = 3.0;
const double& r = a;
const double* p = &a;
```

All these will get compilation error:

```
a = 4.0;
r = 4.0;
*p = 4.0;
```

If we want the pointer to point to only one object!

We force the point to be of the type const, but the object can still be changed.

double a = 3.0;
double b = 4.0;
double const p = &a;*
p is of the type “double*” and is “const”
**p=5.0; //allows me to change the value of a*
p=&b; //compilation error can not point p to b now.

You can have both.
const point const p = &a;*
p is “const” and of the type “const point*”
**p=5.0; //compilation error here too now*

“new” needs “delete”

double p = new double(3.0);*
delete p;

Or you can use smart pointers that delete automatically as it happens for objects and references.

std::shared_ptr
std::unique_ptr

Arrays and pointers

double a[3] = {1.2, 4.4, 5.5};

std::cout<<a[0]<<" "<<a[1]<<" "<<a[2]<<std::endl;

1.2 4.4 5.5

std::cout<<&a[0]<<" "<<&a[1]<<" "<<&a[2]<<std::endl;

0x7fff5bd8d660 0x7fff5bd8d668 0x7fff5bd8d670

std::cout<<&a<<std::endl;

0x7fff5eadc660

The address of the entire array is identical with the address of the first element!

double p = &a[0];* //create p and assign to first element

*std::cout<<*p<<std::endl;*// will print 1.2

p++; // increasing the value of the pointer by 1

moves you to the next element!

*std::cout<<*p<<std::endl;*// will print 4.4

std::vectors and iterators

vectors are arrays, but also have extra functionality,
For example sort(), push_back(), pop_back(), at()
iterator is a pointer, but also has extra functionality
This allows us to loop over elements in a vector easily

```
 std::vector<double>::iterator iter_double;
for(iter_double=v_double.begin();
     iter_double!=v_double.end();
     iter_double++)
   {
     //iter is a pointer to the element
     // *iter is the value
     std::cout<<"current="<<*iter_double<<std::endl;
   }
```

std:maps and iterators

Because of templating, all containters from the Standard Template Library can be iterated on in the same way, using iterators. Here's for a map of string to double.

```
std::map<std::string,double>::iterator iter;
 for(iter=m.begin(); iter!=m.end(); iter++)
   {
    //notice how iter is an iterator, that's why we use iter->
    //we access the first element of the map (key) by iter->first
    //and the second element of the map (value) by iter->second
    std::cout<<"current_first="<<iter->first
            <<" current_second="<<iter->second
            <<std::endl;
   }
```