Object-Oriented Programming in C++

Pre-Lecture 6: Copy and move

Prof Niels Walet (Niels.Walet@manchester.ac.uk)

Room 7.07, Schuster Building

Version: February 24, 2020

Prelecture 6

Outline

In today's lecture: focus on how to replicate objects, covering

- The assignment operator
- The copy constructor
- Deep and shallow copying
- The `this` pointer

Two advanced aspects:

- lvalues and rvalues (simplified!)
- move semantics

A basic example

Class example

This week's example: a simple class for dynamic (1D) arrays

```
4  #include<iostream>
5  class dynamic_array
6  {
7  private:
8    size_t size {};
9    double *array {nullptr};
10 public:
11   dynamic_array()
12      {std::cout<<"Default constructor called"<<std::endl;}
13   dynamic_array(size_t s);
14   ~dynamic_array(){delete array; std::cout<<"Destructor called"<<std::endl;}
15   size_t length() const {return size;}
16   double & operator[](size_t i);
17 };
18 // Parameterized constructor implementation
19 dynamic_array::dynamic_array(size_t s)
20 {
21   std::cout<<"Parameterized constructor called"<<std::endl;
22   if(s<1)
23     {
24       std::cout<<"Error: trying to declare an array with size < 1"<<std::endl;
25       throw("size not positive");
26     }
27   size = s;
28   array = new double[size];
29   for(size_t i{}; i<size; i++) array[i]=0;
30 }
```

Class example

This week's example: a simple class for dynamic (1D) arrays ct'd

```
32 // Overloaded element [] operator implementation
33 double & dynamic_array::operator[](size_t i)
34 {
35   if(i<0 || i>=size)
36     {
37       std::cout<<"Error: trying to access array element out of bounds"<<std::endl;
38       throw("Out of Bounds error");
39     }
40   return array[i];
41 }
42 int main()
43 {
44   std::cout<<"Declaring array a1 with parameterized constructor"<<std::endl;
45   dynamic_array a1{2};
46   std::cout<<"Length of a1 = "<<a1.length()<<std::endl;
47   a1[0] = 0.5;
48   a1[1] = 1.0;
49   std::cout<<"a1[0] = "<<a1[0]<<std::endl;
50   std::cout<<"a1[1] = "<<a1[1]<<std::endl;
51   std::cout<<std::endl;
52   return 0;
53 }
```

Listing 1 : selection of PL6/dynarr.cpp

Class example

Diversion: `operator[]`

We have overloaded the *subscript* operator `double & operator[](int i);` which allowed us to write

```
a1[1] = 1.0;
```

- In line with standard C arrays - use square brackets when referring to an individual element (subscripting).
- Can also overload `operator()`- has advantage that it generalises to more than one parameter, e.g. multi-dimensional arrays, `my3dArray(0,0,0)`.
- Note that we use return by reference so that we can write `a1[0] = 0.5;`: The LHS returns a reference to the value of the first element in `a1` which can then be set to a different value!
- Closely linked to the idea of lvalues and rvalues, see below!

Replication: assignment

Replicating objects:

Now let's declare a 2nd object `a2` . We can then copy its values from `a1` by assignment

```
39 int main()
40 {
41   std::cout<<"Declaring array a1 with parameterized constructor"<<std::endl;
42   dynamic_array a1{2};
43   std::cout<<"Length of a1 = "<<a1.length()<<std::endl;
44   a1[0] = 0.5;
45   a1[1] = 1.0;
46   std::cout<<"a1[0] = "<<a1[0]<<std::endl;
47   std::cout<<"a1[1] = "<<a1[1]<<std::endl;
48   std::cout<<std::endl;
49   std::cout<<"Declaring array a2 with default constructor"<<std::endl;
50   dynamic_array a2;
51   std::cout<<"Length of a2 = "<<a2.length()<<std::endl;
52   std::cout<<"Now copy values from a1 by assignment"<<std::endl;
53   a2=a1;
54   std::cout<<"Length of a2 = "<<a2.length()<<std::endl;
55   std::cout<<"a2[0] = "<<a2[0]<<std::endl;
56   std::cout<<"a2[1] = "<<a2[1]<<std::endl;
57   std::cout<<std::endl;
58   return 0;
59 }
```

Listing 2 : selection of PL6/assignment.cpp

Replicating objects:

The code now outputs

```
Declaring array a1 with parameterized constructor
Parameterized constructor called
Length of a1 = 2
a1[0] = 0.5
a1[1] = 1

Declaring array a2 with default constructor
Default constructor called
Length of a2 = 0
Now copy values from a1 by assignment
Length of a2 = 2
a2[0] = 0.5
a2[1] = 1

Destructor called
Destructor called
```

Replicating objects:

assignment operator

ANALYSIS:

- The statement `a2=a1` copies the member data of `a1` to `a2` so they both have the same length and values after the operation.
- Since `a2` is already instantiated, this is known as an assignment operation.
- Handled by the assignment operator `=`.
- If not provided by the class, the compiler creates a default function `operator=` that overloads this operator for any class.
- We will see there are good reasons why we usually want to do this ourselves.

Replication: shallow copy

Replicating objects:

- We can also copy the values while creating new objects (using initialisation)
- Remember, there are two ways to do this (as with simple data types like `int` and `double`)

add the following (using a3 and a4):

```
56    a2=a1;
57    std::cout<<"Length of a2 = "<<a2.length()<<std::endl;
58    std::cout<<"a2[0] = "<<a2[0]<<std::endl;
59    std::cout<<"a2[1] = "<<a2[1]<<std::endl;
60    std::cout<<std::endl;
61    std::cout<<"Declare array a3 and initialize"<<std::endl↩
          ;
62    dynamic_array a3=a1;
63    std::cout<<"Length of a3 = "<<a3.length()<<std::endl;
64    std::cout<<"a3[0] = "<<a3[0]<<std::endl;
65    std::cout<<"a3[1] = "<<a3[1]<<std::endl;
66    std::cout<<std::endl;
67    std::cout<<"Using other C++ way to declare and ↩
          initialize"<<std::endl;
68    dynamic_array a4{a1};
69    std::cout<<"Length of a4 = "<<a4.length()<<std::endl;
70    std::cout<<"a4[0] = "<<a4[0]<<std::endl;
71    std::cout<<"a4[1] = "<<a4[1]<<std::endl;
72    std::cout<<std::endl;
73    return 0;
74  }
```

Listing 3 : selection of PL6/initialise.cpp

```
Declaring array a1 with parameterized ↩
      constructor
Parameterized constructor called
Length of a1 = 2
a1[0] = 0.5
a1[1] = 1

Declaring array a2 with default constructor
Default constructor called
Length of a2 = 0
Now copy values from a1 by assignment
Length of a2 = 2
a2[0] = 0.5
a2[1] = 1

Declare array a3 and initialize
Length of a3 = 2
a3[0] = 0.5
a3[1] = 1

Using other C++ way to declare and initialize
Length of a4 = 2
a4[0] = 0.5
a4[1] = 1

Destructor called
Destructor called
Destructor called
Destructor called
```

Replicating objects:

copy constructor

- The result may seem surprising: the objects `a3` and `a4` did not need one of our constructors!
- Instead they invoked the default copy constructor.
- This performs a bitwise (or like-for-like) copy of the data from one object to another.
- Also known as a shallow copy (since it copies addresses, rather than the data being pointed to!).
- There are three main situations when the copy constructor is used:
  - When declaring a new object as a copy of an old object (as above).
  - When passing an object to a function by value (need to make local copy of object in function).
  - When creating a temporary object (e.g. in a `return` statement when returning by value).

Replicating objects:

problems with shallow copying

- The above examples (using the default assignment operator and copy constructor) are fine if the objects are simple and we want to create like-for-like copies by value.
- What happens if we do the following?

```
a1[1] = -2.5;
std::cout<<"a1[1] = "<<a1[1]<<std::endl;
std::cout<<"a2[1] = "<<a2[1]<<std::endl;
std::cout<<"a3[1] = "<<a3[1]<<std::endl;
std::cout<<"a4[1] = "<<a4[1]<<std::endl;
return 0;
```

Replicating objects:

problems with shallow copying

- You may (or may not!) be surprised that all first entries (`a1[1]`, `a2[1]`, `a3[1]`, and `a4[1]`) equal -2.5!
- Thus all 4 objects are modified...
- When an object's member data contains a pointer, the address is copied by the default assignment operator and copy constructor, **not** the data it points to.
- That is why it is called a shallow copy.
- So all shallowly copied objects contain a pointer to the same data.
- This can cause serious problems!

Replicating objects:

problems with shallow copying

- Currently our constructor assigns memory that never gets deleted!
- It is very good practice to `delete` array in destructor.
- If we modify the code as follows

```
~dynarr(){cout<<"Destructor called"<<endl; delete[] array;}
```

- we get lots in runtime errors!
- `a4` is destroyed first - the destructor is called and the array is deleted.
- When the destructor for `a3` is called, there is no array left to delete!
- Rule: for all but the simplest classes (no pointers/dynamic memory), it is much better to write our own functions to overload the assignment operator and copy constructor.
- We can control how dynamic arrays are copied - either just the pointer or copy the whole array...
- Latter style is known as deep copying.

Replication: deep copy

Replicating objects:

writing our own rules

- Copy constructor: similar to ordinary constructor but with class type as sole parameter

```
20 // Copy constructor for deep copying
21 dynamic_array::dynamic_array(dynamic_array &arr)
22 {
23   // Copy size and declare new array
24   array=nullptr; size=arr.length();
25   if(size>0)
26     {
27       array=new double[size];
28       // Copy values into new array
29       for(size_t i{};i<size;i++) array[i] = arr[i];
30     }
31 }
```

Listing 4 : selection of PL6/deep.cpp

- Used when a new object is declared as a copy of an existing object `dynarr a3=a1; dynarr a4{a1};`.

Replicating objects:

writing our own rules

- Assignment operator - similar to copy constructor except that we assume the object is already constructed!
- We must therefore delete existing data first before copying

```
32 // Assignment operator for deep copying
33 dynamic_array & dynamic_array::operator=(dynamic_array &arr)
34 {
35   if(&arr == this) return *this; // no self assignment
36   // First delete this object's array
37   delete[] array; array=nullptr; size=0;
38   // Now copy size and declare new array
39   size=arr.length();
40   if(size>0)
41     {
42       array=new double[size];
43       // Copy values into new array
44       for(size_t i{};i<size;i++) array[i] = arr[i];
45     }
46   return *this; // Special pointer!!!
47 }
```

Listing 5 : selection of PL6/deep.cpp

- Used when an existing object is assigned to another `dynarr a2; a2=a1;`.

Replicating objects:

writing our own rules

- This is now used in the code PL6/deep.cpp; the statements

```
107   a1[1] = -2.5;
108   std::cout<<"a1[1] = "<<a1[1]<<std::endl;
109   std::cout<<"a2[1] = "<<a2[1]<<std::endl;
110   std::cout<<"a3[1] = "<<a3[1]<<std::endl;
111   std::cout<<"a4[1] = "<<a4[1]<<std::endl;
```

now give the output

```
a1[1] = -2.5
a2[1] = 1
a3[1] = 1
a4[1] = 1
```

- Each object now has its own memory and copies of the data (as we performed deep copies)

Replicating objects:

this pointer this

- The assignment operator returns a *reference* to the basic type, `dynarr&`
- This is usually true for operators, so one can do things like

```
a=b=c; // same as a=(b=c) so b=c must have same type as a
```

- For the operation `b=c` the object returned is the identical to the object calling the function (contrast this with the operation `b+c` where `b` calls function and `b+c` is returned)
- For this purpose, all member functions have access to a special pointer called `this` which points to object itself!
- Example: we can access member data in a different way `int length() const return this->size;`.
- Better use: `return *this` when we just want to return back the object calling the function.

Replicating objects:

this pointer this

- Another example of using the this pointer:
- When we are overloading operator=, we need to protect against possible self-assignment. Trivial dangerous example in our code: a2=a2
- In this case, our code would crash since it could delete the object's data (and allocated memory) before trying to copy itself!
- Simplest way to avoid that was to write

```
dynamic_array & dynamic_array::operator=(dynamic_array &arr↩
    )
{
  if(&arr == this) return *this; // no self assignment
  // First delete this object's array
```

- Here, the code simply compares the address of the object (this) and the address of the argument arr to check if they are the same.

Replicating objects:

a word of caution

- When defining a function to overload the assignment operator, we returned the object by reference

```
myclass & myclass::operator=(myobject){... return *this;}
```

- Why? When returning an object by value a copy is made and returned; the original object is then destroyed!
- Shallow-copied objects may then point to data that no longer exists.
- Returning by reference avoids this when a (deep) copy constructor is not defined.
- Even when it is, returning by reference is faster (but only works for objects that do not go out of scope).

Summary, part 1

Prelecture 6

Summary, part 1

In the first part of today's lecture: we looked at how and when to replicate objects, covering

- The assignment operator
- The copy constructor
- Deep and shallow copying
- The `this` pointer

Part2: advanced aspects

Lvalues and Rvalues

Lvalues and Rvalues

Normal and temporary variables

- If you ever read any advanced material on C++ (e.g., the C++ specification) a lot of time is spent on discussing **rvalues** and **lvalues**;
- Actually the discussion here is substantially simplified!
- An lvalue is *originally* a variable that can appear on the left-hand side of an expression, and an rvalue one that can only occur on the right-hand side
- More specifically, and lvalue is something where we can take the address, something in (semi)permanent memory. They don't have to be variables, e.g., a[i]=10 or more complicated functions are allowed (as long as we have a referable object at the left).
- An rvalue, on the other hand refers to a temporary object; in order to capture these in permanent memory, the only way is to copy them into an lvalue.
- That can be quite inefficient!

Lvalues and Rvalues

lvalue and rvalue references

- Like we said we know the address of an lvalue, so we can write `lvalue&`; C++11 introduces the idea of an rvalue reference as well, `rvalue&&` (note the double ampersand!)
- Why? What is this useful for? It allows us to write functions that specifically deal with "mutable" temporary variables. Consider the following two statements

```
print_reference (const String& str) {cout << str; }
print_reference (String&& str) {cout << str; }
```

  The first one accept any constant lvalue–it actually accepts any argument it is given, lvalue or rvalue. The second overload actually picks up a mutable rvalue (no const, &&), so the general function is left with the remainder,

- So we now have a way to differentiate a mutable rvalue (temporary) from all the other forms of a variable, and we can act on that. But why is that useful? The answer is ...

Move semantics

Move semantics

Move constructor and move assignment

- The most common way of using rvalue references is in the “move constructor” and “move assignment”. In many senses these parallel the copy constructors discussed before.
- These are specified in almost the same way, but they take an rvalue reference
- Their implementation is very different
- See the next slide for an example

move vs copy: constructor

```
22 // Copy constructor for deep copying
23 dynamic_array::dynamic_array(dynamic_array &arr)
24 {
25   // Copy size and declare new array
26   std::cout <<"copy constructor\n";
27   array=nullptr; size=arr.length();
28   if(size>0) {
29       array=new double[size];
30       // Copy values into new array
31       for(size_t i{};i<size;i++) array[i] = arr[i];
32    }
33 }
```

```
34 // Move constructor
35 dynamic_array::dynamic_array(dynamic_array &&arr)
36 { // steal the data
37   std::cout <<"move constructor\n";
38   size=arr.size;
39   array=arr.array;
40   arr.size=0;
41   arr.array=nullptr;
42 }
```

Listing 6 : selection of PL6/move.cpp

move vs copy: assignment

```
43 // Assignment operator for deep copying
44 dynamic_array & dynamic_array::operator=(↩
       dynamic_array &arr)
45 {
46   std::cout <<"copy assignment\n";
47   if(&arr == this) return *this; // no self ↩
       assignment
48   // First delete this object's array
49   delete[] array; array=nullptr; size=0;
50   // Now copy size and declare new array
51   size=arr.length();
52   if(size>0){
53       array=new double[size];
54       // Copy values into new array
55       for(size_t i{};i<size;i++) array[i] = arr[i];
56     }
57   return *this; // Special pointer!!!
58 }
```

```
59 // Move assignment operator
60 dynamic_array & dynamic_array::operator=(↩
       dynamic_array&& arr)
61 {
62   std::cout <<"move assignment\n";
63   std::swap(size,arr.size);
64   std::swap(array,arr.array);
65   return *this; // Special pointer!!!
66 }
```

Listing 7 : selection of PL6/move.cpp

Move semantics

std::move

- Suppose I know an lvalue object is no longer useful, and I do want to use the move assignment to reassign its data
- Is there a way to do this?
- Need to turn (cast?) an lvalue to an rvalue
- Can be done by using a `static_cast` using an rvalue reference, but nicer is the `std::move` function defined in C++
- Misnamed, because it turns an lvalue into something that can be used like an rvalue (and thus its data can be moved, and the objects content destroyed). Thus `std::move` itself moves nothing!

move vs copy: assignment

```
113   dynamic_array a3(2);
114   std::cout<<"Length of a3 = "<<a3.length()<<std::endl;
115   a3[0] = 0.5;
116   a3[1] = 1.0;
117   std::cout<<"a3[0] = "<<a3[0]<<std::endl;
118   std::cout<<"a3[1] = "<<a3[1]<<std::endl;
119   std::cout<<std::endl;
120   std::cout<<"Now move values from a1 by assignment"<<↩
          std::endl;
121   dynamic_array a4;
122   a4= std::move(a3);
123   std::cout<<"Length of a4 = "<<a4.length()<<" and of ↩
          a3 ="<<a3.length()<<std::endl;
124   std::cout<<"a4[0] = "<<a4[0]<<std::endl;
125   std::cout<<"a4[1] = "<<a4[1]<<std::endl;
126   std::cout<<std::endl;
```

Listing 8 : selection of PL6/move.cpp

```
Declaring array a3 with parameterized constructor
Parameterized constructor called
Length of a3 = 2
a3[0] = 0.5
a3[1] = 1

Now move values from a1 by assignment
Default constructor called
move assignment
Length of a4 = 2 and of a3 =0
a4[0] = 0.5
a4[1] = 1
```

Listing 9 : selection of PL6/move.out

Summary, part 2

Prelecture 6

Summary, part 2

In the second part of today's lecture: we looked at how and when to replicate objects, covering

- lvalues, rvalues
- move semantics