C++ - Pre-lecture 6

Assigning and moving objects in memory

Caterina Doglioni, Charanjit Kaur, PHYS30762 - OOP in C++ 2024

The University of Manchester

Part 0.1: in this lecture

In this lecture

- We will see the various ways one can **assign** and **copy** objects
- [video 1] conceptual video: **memory management** 101
  - how memory works, in a nutshell
  - the heap and the stack
- [video 2] **replication** and **assignment**
- [video 3] **shallow** and **deep copying**
- [video 4] more advanced concepts:
  - **`lvalue`** and **`rvalue`**
  - **`move`** semantics (assignment/constructors)

Video 1: memory management 101

MANCHESTER 1824
The University of Manchester

Prerequisite: memory management 101

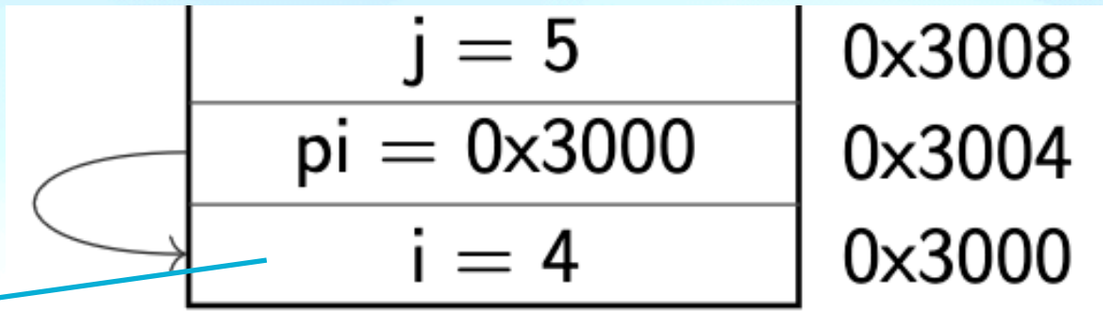
TRM feedback: more concept, slightly fewer examples

- How are objects stored in memory?
- Example of pointers discussed in Lecture 4

Your code

```
int i = 4;
int *pi = &i;
int j = *pi + 1;
```

Memory layout



this is a contiguous array of bytes

- One “block” corresponds to a space in memory
- The variable name is used to retrieve its content
- Pointers also occupy space in memory
  - When dereferencing a pointer (using *) you can get the content of the memory space having that address

Prerequisite: memory management 101

TRM feedback: more concept, slightly fewer examples

- Memory **allocation** = memory is requested to store something (e.g. a variable)
    - It doesn't mean that the thing you want is stored there, unless you define your variable! It is important to initialise your variables
- Memory **de-allocation**: memory is released and can be filled again
- **Automatic** memory allocation: python, Java
- **Dynamic** memory management: C++, (C)
    - You can request memory (so far, at your own risk and responsibility!) with `new/delete/(malloc)`
    - You can also use automatic memory management paradigm: smart pointers (see lectures 7-8)
        - Why would you not do this? When you're running on resource-constrained systems (e.g. computers on cars…)

The heap and the stack

- Simple TLDR: heap and stack are memory spaces located on your computer's RAM
  - Regularly instantiated objects go on the **stack**
    - When object goes out of scope, memory gets freed automatically
    - When a class goes out of scope, its *destructor* is called
  - New (+ delete) objects go on the **heap**
    - Memory does not get freed automatically
    - Beware of memory leaks if you don't delete → smart pointer preferred (see lecture 7)
- In both cases you can run out of memory if you put too much content on your memory (*"stack overflow"* comes from this issue)

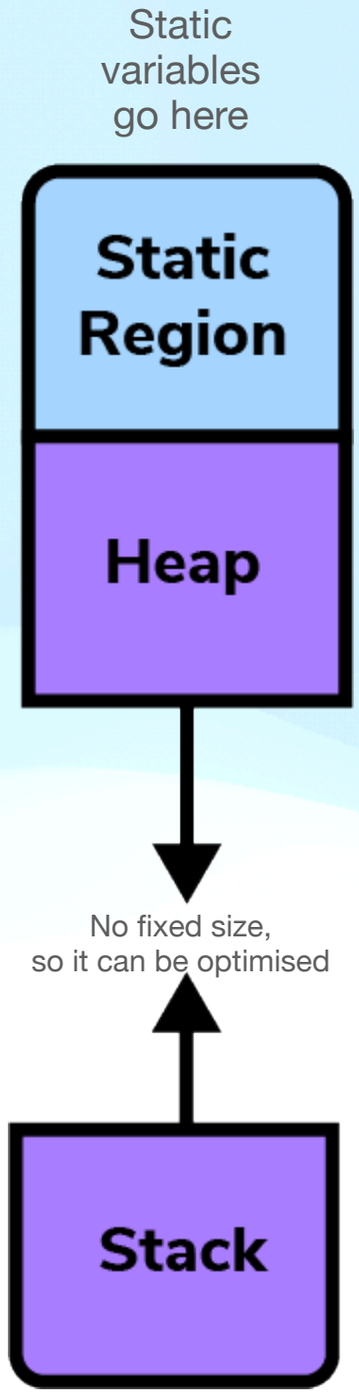


Image source link

What happens when you instantiate a class?

- `MyClass theClass("a"); //parameterised constructor`
- A memory location on the stack corresponding to the size of MyClass (data members & functions) is reserved, and its data member is initialised w/ "a"
- There is only one copy of theClass, and your code modifies this
- when theClass goes out of scope:
  - the destructor is called
    - If anything inside the class has been allocated dynamically, e.g. in the constructor (on the heap), then this is the place to delete it!
      - Otherwise that block of memory remains orphan of links to its original class and can create memory leaks
  - the memory on the stack is freed and can be used for other things

Video 2: examples of assignment and replication

MANCHESTER 1824
The University of Manchester

Assignment: the concept

- Assignment means we assign a right-hand side value to a left-hand side variable
  - Easiest way to do this: using “=“
  - More generic concepts in C++: `lvalue` and `rvalue` (see later)
- We are going to rewrite our own std::vector as an example
  - *(note: don’t use this in assignments, use std::vector)*

Assignment: the example

```
#include<iostream>
class dynamic_array
{
private:
  size_t size {};
  double *array {nullptr};
public:
  dynamic_array()
     {std::cout<<"Default constructor called"<<std::endl;}
  dynamic_array(size_t s);
  ~dynamic_array(){delete array; std::cout<<"Destructor called"<<std::endl;}
  size_t length() const {return size;}
  double & operator[](size_t i);
};
// Parameterized constructor implementation
dynamic_array::dynamic_array(size_t s)
{
  std::cout<<"Parameterized constructor called"<<std::endl;
  if(s<1)
    {
      std::cout<<"Error: trying to declare an array with size <
          1"<<std::endl;
      throw("size not positive");
    }
  size = s;
  array = new double[size];
  for(size_t i{}; i<size; i++) array[i]=0;
}
```

Health warning: the examples don't follow our suggestion to name classes starting with a capital and then using CamelCase as we haven't updated them yet, but you should start with good practices and keep them in the future!

This is an **exception** - covered in lectures 9 and 10

Find this code on GitHub at: Prelecture6/dynarr.cpp

Assignment: the example

```
// Overloaded element [] operator implementation
double & dynamic_array::operator[](size_t i)
{
  if(i<0 || i>=size)
    {
      std::cout<<"Error: trying to access array element out of
          bounds"<<std::endl;
      throw("Out of Bounds error");
    }
  return array[i];
}
int main()
{
  std::cout<<"Declaring array a1 with parameterized constructor"<<std::endl;
  dynamic_array a1{2};
  std::cout<<"Length of a1 = "<<a1.length()<<std::endl;
  a1[0] = 0.5;
  a1[1] = 1.0;
  std::cout<<"a1[0] = "<<a1[0]<<std::endl;
  std::cout<<"a1[1] = "<<a1[1]<<std::endl;
  std::cout<<std::endl;
  return 0;
}
```

can overload anything!
e.g. think of a sum or dot product of 4-vector classes…

This is an **exception** - covered in lectures 9 and 10

Find this code on GitHub at: Prelecture6/dynarr.cpp

left-hand side of overloaded operator function: `double &`

returning by reference: we are actually modifying the class data members

assignment via "=": assigning the value 0.5 to the left-hand side of the operator[] (a reference)

Demo in video: see what happens in the destructor (that gets called when `a1` goes out of scope at the end of `main()`), **we'll see what's wrong with it!**

Replication by assignment

- Conceptually: replication by assignment is when you assign a right-hand-side value to a left-hand side value
  - When both are objects, what happens to the data members?
  - Naive expectation on how it should behave:
    - both objects should have the same data members and content
  - Who is actually doing this in the class?
    - Default assignment operator =
      - If it’s not defined by you, the compiler will add it for you (like the default constructor)
  - What happens to the memory? We’ll see…

Replication by assignment: example

```
int main()
{
  std::cout<<"Declaring array a1 with parameterized
      constructor"<<std::endl;
  dynamic_array a1{2};
  std::cout<<"Length of a1 = "<<a1.length()<<std::endl;
  a1[0] = 0.5;
  a1[1] = 1.0;
  std::cout<<"a1[0] = "<<a1[0]<<std::endl;
  std::cout<<"a1[1] = "<<a1[1]<<std::endl;
  std::cout<<std::endl;
  std::cout<<"Declaring array a2 with default constructor"<<std::endl;
  dynamic_array a2;
  std::cout<<"Length of a2 = "<<a2.length()<<std::endl;
  std::cout<<"Now copy values from a1 by assignment"<<std::endl;
  a2=a1;
  std::cout<<"Length of a2 = "<<a2.length()<<std::endl;
  std::cout<<"a2[0] = "<<a2[0]<<std::endl;
  std::cout<<"a2[1] = "<<a2[1]<<std::endl;
  std::cout<<std::endl;
  return 0;
}
```

declare a2 (reserve memory for it)

copy values of a1 to a2 by assignment

Find this code on GitHub at: [Prelecture6/assignment.cpp]

Replication by assignment: example output

```cpp
int main()
{
  std::cout<<"Declaring array a1
      constructor"<<std::endl;
  dynamic_array a1{2};
  std::cout<<"Length of a1 = "<<
  a1[0] = 0.5;
  a1[1] = 1.0;
  std::cout<<"a1[0] = "<<a1[0]<<
  std::cout<<"a1[1] = "<<a1[1]<<
  std::cout<<std::endl;
  std::cout<<"Declaring array a2
  dynamic_array a2;
  std::cout<<"Length of a2 = "<<
  std::cout<<"Now copy values fr
  a2=a1;
  std::cout<<"Length of a2 = "<<
  std::cout<<"a2[0] = "<<a2[0]<<
  std::cout<<"a2[1] = "<<a2[1]<<
  std::cout<<std::endl;
  return 0;
}
```

```
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
```

Find this code on GitHub at: [Prelecture6/assignment.cpp]

Replication by assignment: what happens

- 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 **copy assignment** operator=).

- If not provided by the class, the compiler creates a **default function operator=** that **overloads** this operator for any class.

- In the next video: good memory-related reasons why we usually want to write this ourselves.

- Example: let's try to change `a1`, and see what happens to `a2` (demo)

Video 3: shallow and deep copying

MANCHESTER 1824
The University of Manchester

Copying: the concept

- Copying is conceptually close to assigning, but in this case we don’t yet have a “fresh” object already ready for it
- For copying, we do the memory allocation and the assignment in one go
  - `dynamic_array a3 = a1` / `dynamic_array a3{a1}`
- The difference wrt assignment that a new function is called for the new objects instead of their parameterised/default constructors: the **copy constructor**
  - As before, if you don’t have a copy constructor, the compiler will add it for you

What does the copy constructor do?

- The copy constructor performs a bitwise (or like-for-like) copy of the data from one object to another.
- This process is also known as a **shallow copy** (since it copies addresses, rather than the data being pointed to!)
  - So, if you modify the original object, the change propagates
- Three main situations when you want this to happen: you want to **save memory** and…
  - declare a new object as an actual copy of an old object
  - pass an object to a function by value and need to make a local copy of the object in function (this goes out of scope anyway at the end of the function, so no point in wasting memory on it)
  - create a temporary object (e.g. in a return statement when returning by value).

Shallow copying: the example

```
a2=a1;
std::cout<<"Length of a2 = "<<a2.length()<<std::endl;
std::cout<<"a2[0] = "<<a2[0]<<std::endl;
std::cout<<"a2[1] = "<<a2[1]<<std::endl;
std::cout<<std::endl;
std::cout<<"Declare array a3 and initialize"<<std::endl;
dynamic_array a3=a1;
std::cout<<"Length of a3 = "<<a3.length()<<std::endl;
std::cout<<"a3[0] = "<<a3[0]<<std::endl;
std::cout<<"a3[1] = "<<a3[1]<<std::endl;
std::cout<<std::endl;
std::cout<<"Using other C++ way to declare and initialize"<<std::endl;
dynamic_array a4{a1};
std::cout<<"Length of a4 = "<<a4.length()<<std::endl;
std::cout<<"a4[0] = "<<a4[0]<<std::endl;
std::cout<<"a4[1] = "<<a4[1]<<std::endl;
std::cout<<std::endl;
```

Find this code on GitHub at: [Prelecture6/initialise.cpp]

Shallow copying: the example

```
a2=a1;
std::cout<<"Length of a2 = "<<a2.lengt
std::cout<<"a2[0] = "<<a2[0]<<std::end
std::cout<<"a2[1] = "<<a2[1]<<std::end
std::cout<<std::endl;
std::cout<<"Declare array a3 and initi
dynamic_array a3=a1;
std::cout<<"Length of a3 = "<<a3.lengt
std::cout<<"a3[0] = "<<a3[0]<<std::end
std::cout<<"a3[1] = "<<a3[1]<<std::end
std::cout<<std::endl;
std::cout<<"Using other C++ way to dec
dynamic_array a4{a1};
std::cout<<"Length of a4 = "<<a4.lengt
std::cout<<"a4[0] = "<<a4[0]<<std::end
std::cout<<"a4[1] = "<<a4[1]<<std::end
std::cout<<std::endl;
```

```
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
```

Find this code on GitHub at: [Prelecture6/initialise.cpp]

Shallow copying: the downside

```
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;
```

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

Find this code on GitHub at: [Prelecture6/shallow.cpp]

All the 4 objects now point to the same data!

Shallow copying: even bigger problems

```
// Parameterized constructor implementation
dynamic_array::dynamic_array(size_t s)
{
  std::cout<<"Parameterized constructor called"<<std::endl;
  if(s<1)
    {
      std::cout<<"Error: trying to declare an array with size <
          1"<<std::endl;
      throw("size not positive");
    }
  size = s;
  array = new double[size];
  for(size_t i{}; i<size; i++) array[i]=0;
}
```

Find this code on GitHub at: Prelecture6/initialise.cpp

Our parameterised constructor allocates memory dynamically via `new`

- We've seen earlier that we should have freed the memory using a delete, so let's do that in the destructor

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

- The problem is that every destructor from the shallow copies tries to delete the same pointer → errors!

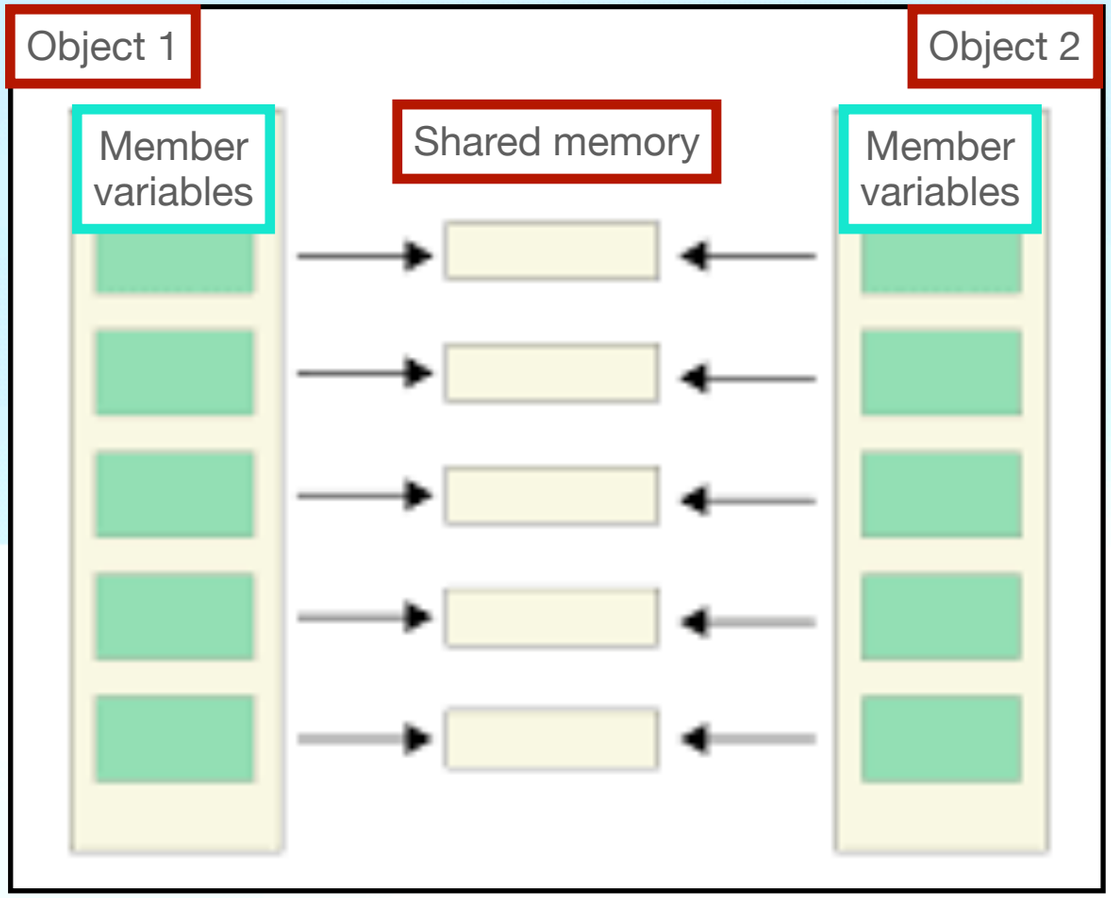
- Solution: overload assignment operator & copy constructor

What happens when you shallow-copy a class?

- `MyClass theClassA("a"); //parameterised constructor called`
  - A memory location on the stack corresponding to the size of MyClass (data members & functions) is reserved, and its data member is initialised with "a"
- `MyClass theClassB = theClassA; //implicit copy constructor called`
  - Memory is only allocated for pointers to data members and it's shared between B and A, including any dynamically allocated memory
- when `theClassA` goes out of scope:
  - the destructor of A is called
- when `theClassB` goes out of scope:
  - is the destructor of B called as well, and can cause trouble

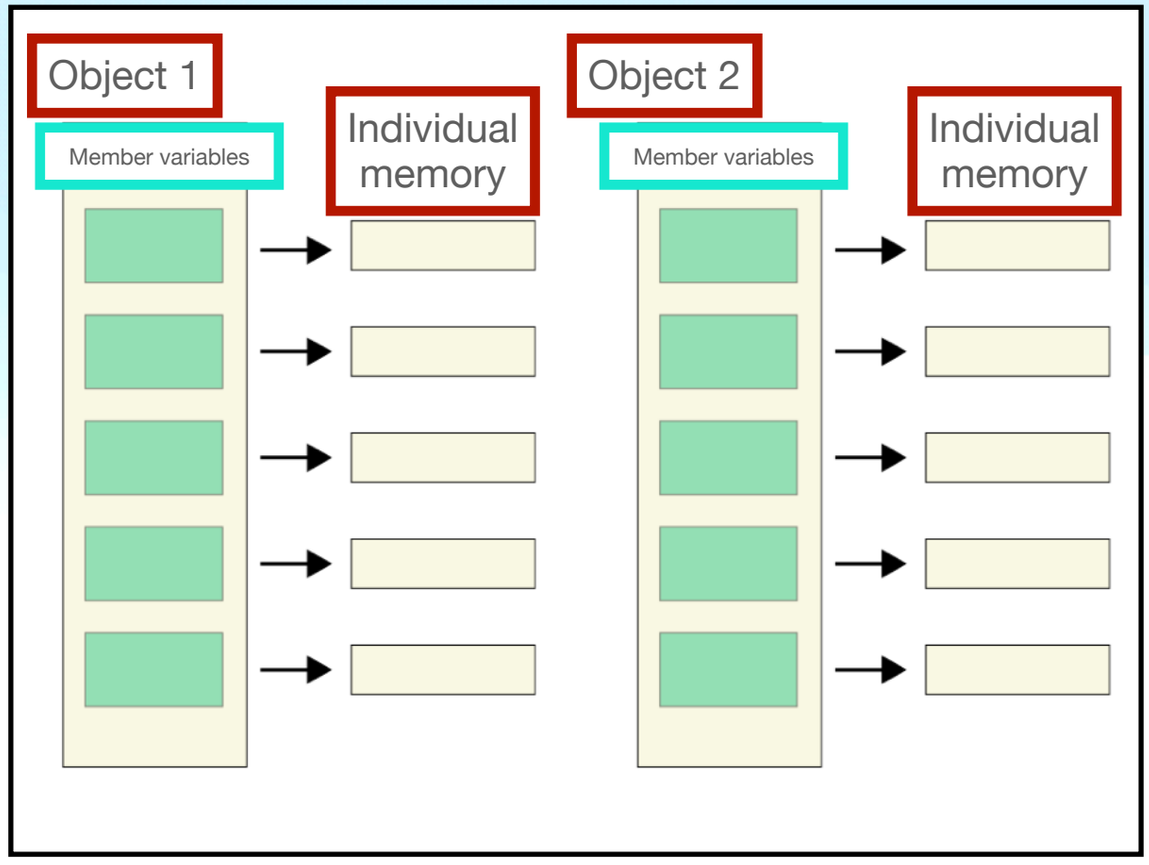
Shallow vs deep copying

Shallow copy: share memory for the same content



When changing object 1 or object 2, the other object also changes
Clashes when freeing the same memory space can happen

Deep copy: separate memory with the same content



Object 1 and object 2 change independently
Memory gets freed separately

Deep copying: copy constructor

- Good practice for classes that aren't trivial: explicitly perform what you want to do in your copy constructor
- Especially important: allocate memory in the same way as your copied object

```
// Copy constructor for deep copying
dynamic_array::dynamic_array(dynamic_array &arr)
{
  // Copy size and declare new array
  array=nullptr; size=arr.length();
  if(size>0)
    {
      array=new double[size];
      // Copy values into new array
      for(size_t i{};i<size;i++) array[i] = arr[i];
    }
}
```

Find this code on GitHub at: Prelecture6/deep.cpp

This is a constructor with a reference to a class (of the same type) as parameter

Video demo: now the output is different!

Deep copying: assignment operator

See also: **rule of three** https://en.wikipedia.org/wiki/Rule_of_three_(C++_programming)

- Good practice for classes that aren't trivial: explicitly perform what you want to do in your assignment operator (same as copy constructor)

- Only difference: here you have an object (and its memory) already being allocated on the left-hand side of the =, so you must free any dynamically allocated memory first

```
// Assignment operator for deep copying
dynamic_array & dynamic_array::operator=(dynamic_array &arr)
{
  if(&arr == this) return *this; // no self assignment
  // First delete this object's array
  delete[] array; array=nullptr; size=0;
  // Now copy size and declare new array
  size=arr.length();
  if(size>0)
    {
      array=new double[size];
      // Copy values into new array
      for(size_t i{};i<size;i++) array[i] = arr[i];
    }
  return *this; // Special pointer!!!
}
```

Find this code on GitHub at: Prelecture6/deep.cpp

This is an operator with a reference to a class (of the same type) as parameter and return value

What happens when you deep-copy a class?

- `MyClass theClassA("a"); //parameterised constructor called`
  - A memory location on the stack corresponding to the size of MyClass (data members & functions) is reserved, and its data member is initialised with "a"
- `MyClass theClassB = theClassA; //user defined copy constructor called`
  - Memory is dynamically allocated for any data members that need it, in a way that exactly mirrors what is done for class A but in a different place
- when `theClassA` goes out of scope:
  - the destructor of A is called
- when `theClassB` goes out of scope:
  - the destructor of B called and frees the right memory!

What is *this?

- TLDR: `*this` is a pointer you can access in an object that points to the object itself
- Why is it there? It is related to the assignment operator
- The assignment operator returns a reference to the basic type, `dynamic_array&`
- 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 identical to the object calling the function (in contrast with the `b+c` where `b` calls the function and `b+c` is returned).
- For this purpose, all member functions have access to a special pointer called `*this`

How do I use *this?

- For example, when accessing member data

```
int length() const return this->size;
```

- When is it used in functions: return *this when you want to return back the object calling the function
- Another use: chaining member functions
  - If the first function does something and returns *this, you can call another function on the return value as:
  - objectA->doTheThing()->doTheOtherThing();
  - Suggestion: be aware of this use, but don't use in assignments unless you find it strictly necessary

this* and the copy constructor

- A problem when using the copy constructor is self-assignment (a=a) wouldn't make sense memory and code-wise, and crash
- Principles of OOP: *abstraction* & *encapsulation*
  - You don't know how a user will use your class, as they only see the interface
  - Better to **protect against things that shouldn't be happening**, like self-assignment → *this can be used to compare the address of the new & existing object in memory (using &), and return the object itself if they are the same

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

Summary so far

Find these codes on GitHub at: [Prelecture6/[code].cpp]

Class example: dynamic 1D array
Two constructors, destructor, a1

dynarr.cpp

Default assignment operator
`a2=a1`

assignment.cpp

Default copy constructor
`a3=a4`

initialise.cpp

Shallow copying
Default copy constructor & assignment op.

shallow.cpp

Deep copying
User defined rules for copying
Copy constructor / assignment op. (`*this` ptr)

deep.cpp

Video 4: more advanced concepts: lvalue/rvalue and move semantics

Note: we will not explicitly test on these C++ concepts apart from asking you to implement them, but understanding how they work helps you become a more advanced programmer!

lvalue and rvalue

- If you read any advanced material on c++ (e.g., the C++ specifications) a lot of time is spent on discussing `rvalues` and `lvalues` —> here we simplify!

- An `lvalue` is originally a variable that can appear on the LHS of an expression, and an `rvalue` is one that can only occur on the RHS.

- More specifically, `lvalue` is something where **we can take the address**, stored in (semi) **permanent** memory**: a `lvalue` is an object reference**

  - They don't have to be variables.- more complicated functions are allowed (as long as we have a readable object at the left).

- An `rvalue` refers to anything else (operator results, numbers used in the code) and it is a **temporary** object, **it does not have to have any associated memory**

  - in order to capture these in permanent memory, the only way is to copy them into an `lvalue`.

Uses of lvalue and rvalue

For more: https://www.internalpointers.com/post/understanding-meaning-lvalues-and-rvalues-c

- Try (in a main()): `x=4;` vs `4=x;`
  - x is the lvalue, 4 is the rvalue. The second (obviously) does not work, and the compiler will tell us off
- Similarly, assigning something straight away to a function returning a reference works, while assigning something to a function returning a number does not
- Converting lvalues to rvalues works, the other way around does not (by design!)

- Try (in a main()):
  - `int x = 4;` `int x = 4;` `int z = x+y;` ✅ implicit lvalue-to-rvalue conversion
  - `int y = 10;` `int& yref = y;` ✅ lvalue reference (both are lvalues)
- Try (in a main()):
  - `int& yref = 10;` ❌ forbidden: can't convert rvalue to lvalue reference because the rvalue does not have an address. But complaint has to do with *const-ness*…

const lvalue & rvalue reference

For more: https://www.internalpointers.com/post/c-rvalue-references-and-move-semantics-beginners

- Try: `const int& myRef = 10;` ✅ allowed!
  - With this, you can **never** modify the reference, and therefore the value it points to. The pointer "allows" this by creating a new lvalue / rvalue pair, with a new variable where 10 (called a *literal*) is stored, and pointing myRef to that.
- C++0x introduced something for doing this that is non-const: a **reference to a `rvalue` (&&)**
  - Try: `int&& myRef = 10; myRef++;`
- Where does this matter?
  - When you want to **move** all data from a class to another (to save memory), we will see that you have to convert a `lvalue` to a `rvalue`
  - This is the basis of the **move semantics & operator**

Reasons to move

- Suppose we know an `lvalue` object is no longer useful, and we do want to save memory and **reassign** (steal) **its data**.
- Why would we do this? To save memory. For our initial example of the dynamic vector, we may want to:
  - allocate an initial chunky vector:
    - `dynamic_array myFirstLargeArray(100000);`
  - at some point later, copy that vector into another vector using the assignment operator:
    - `dynamic_array mySecondLargeArray;`
    - `mySecondLargeArray = myFirstLargeArray:`
- If we are sure we don't need the first chunky vector anymore, no point in keeping memory around for both!

How can we move?

- It would be convenient to have this with a syntax that is close to the one of the assignment operator/copy constructor:
  - `dynamic_array my2ndLargeArray =steal myIstLargeArray;`

This does not exist! We will use std::move instead

- This means that in our "move syntax" we need to turn a `lvalue` into a `rvalue` as we want to modify `myIstLargeArray`
- Part of the solution: use the **&&** syntax to reference the `rvalue`, taking the assignment operator / copy constructor as examples

Copy vs move assignment

```
// Assignment operator for deep copying
dynamic_array & dynamic_array::operator=(dynamic_array &arr)
{
  std::cout <<"copy assignment\n";
  if(&arr == this) return *this; // no self assignment
  // First delete this object's array
  delete[] array; array=nullptr; size=0;
  // Now copy size and declare new array
  size=arr.length();
  if(size>0){
      array=new double[size];
      // Copy values into new array
      for(size_t i{};i<size;i++) array[i] = arr[i];
    }
  return *this; // Special pointer!!!
}
```

Find this code on GitHub at: Prelecture6/deep.cpp

```
// Move assignment operator
dynamic_array & dynamic_array::operator=(dynamic_array&& arr)
{
  std::cout <<"move assignment\n";
  std::swap(size,arr.size);
  std::swap(array,arr.array);
  return *this; // Special pointer!!!
}
```

Find this code on GitHub at: Prelecture6/move.cpp

Copy vs move constructors

```
// Copy constructor for deep copying
dynamic_array::dynamic_array(dynamic_array &arr)
{
  // Copy size and declare new array
  std::cout <<"copy constructor\n";
  array=nullptr; size=arr.length();
  if(size>0) {
      array=new double[size];
      // Copy values into new array
      for(size_t i{};i<size;i++) array[i] = arr[i];
    }
}
```

Find this code on GitHub at: Prelecture6/deep.cpp

```
// Move constructor
dynamic_array::dynamic_array(dynamic_array &&arr)
{ // steal the data
  std::cout <<"move constructor\n";
  size=arr.size;
  array=arr.array;
  arr.size=0;
  arr.array=nullptr;
}
```

Find this code on GitHub at: Prelecture6/move.cpp

…but now they look the same?

- If both custom copy and move are provided, this is where the compiler's **overload resolution** comes into place
  - in case of default copy/move, the compiler optimises and picks the one that is most convenient for what one is doing - we trust it!
  - `copy` takes priority if the argument is a `lvalue` (a named object that is in use), or there is any passing by value (we actually need to make a copy)
  - `move` takes priority if the `std::move` function is called, or if the argument is an `rvalue`
    - Our examples (and your code should) explicitly use `std::move`, which in turn calls the move constructor

Copy vs move in action

```
std::cout<<"Declaring array a3 with parameterized constructor"<<std::endl;
dynamic_array a3(2);
std::cout<<"Length of a3 = "<<a3.length()<<std::endl;
a3[0] = 0.5;
a3[1] = 1.0;
std::cout<<"a3[0] = "<<a3[0]<<std::endl;
std::cout<<"a3[1] = "<<a3[1]<<std::endl;
std::cout<<std::endl;
std::cout<<"Now move values from a1 by assignment"<<std::endl;
dynamic_array a4;
a4= std::move(a3);
std::cout<<"Length of a4 = "<<a4.length()<<" and of a3 ="<<a3.length()<<std::endl;
std::cout<<"a4[0] = "<<a4[0]<<std::endl;
std::cout<<"a4[1] = "<<a4[1]<<std::endl;
std::cout<<std::endl;
```

```
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
```

std::move turns an lvalue into something that can be used like an rvalue (and thus its data can be moved, and the object's content destroyed). Thus std::move itself moves nothing!

Overall summary

- In this part, we talked about the `lvalues`, `rvalues` and `move` semantics to understand how and when to replicate objects.

- This also completes our discussion of **special member functions** of the class which we have been discussing from week 4 onwards. Those are:

- Default constructor
- Destructor
- Copy Assignment
- Copy Constructor
- Move Constructor
- Move Assignment

C++ Rule of Three:

The rule of three (also known as *the law of the big three* or *the big three*) is a rule of thumb in C++ (prior to C++11) that claims that if a class defines any of the following then it should probably explicitly define all three [source: Wikipedia + Bjarne Stroustrup, C++, 3rd ed (2000)]

C++ Rule of Five:

Same as rule of three, but since C++11 introduces move semantics, it includes the move constructor/assignment