**BACK TO CLASSES**

* -> operator combines “ \* ” (dereference) and “ . ”

MyClass \*p;

p = new MyClass;

p->grade = “A”; (Equivalent to: (\*p).grade = “A”;)

I have to use parantheses because precedence of . is higher than \*.

**The this Pointer**

2 ways for member functions to access:

1. cout << stuff;
2. cout << this->stuff.

At the beginning of each member function I have this:

* PFArrayD \* const this;

If parameter of your member function has the same name as the data member, then you can do like this:

//used is data member

const double& PFArrayD::operator[](int used) const{

if (used == this -> used) …

…

}

DON’T DO THIS ANYWAY.

*If your object is dynamic, constructer is called with the new operator and destructor is called when you delete the object.*

*There is only 1 destructor.*

**Copy Constructors**

There is always copy constructor available.

* Automatically called when:
  1. Class object declared and initialized to other object ( Money m2(m1); )
  2. When func returns class type object

Money f(int k){

…

Money m;

…

return m;

}

After I return m, m is killed. So what am I returning?

* Compiler automatically calls the copy constructor (which takes m as a parameter) to make a temporary object to return.
* If you don’t have a copy constructor in your class, you won’t be able to return an object of your class.
  1. When argument of class type is “plugged in” as actual argument to call-by-value parameter

When we do call by value, copy of our object will be made. Who is making that copy : copy constructor which is automatically called by compiler.

void f(Money m){ … }

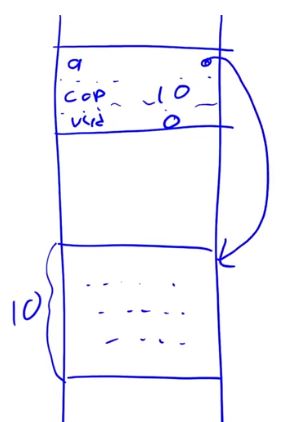
* Requires “temporary copy” of object
  + Copy constructor creates it
* Default copy constructor
  + Like default “=”, performs member-wise copy
* Pointers 🡪 write own copy constructor!
* If you write your copy constructor, you say that you are not gonna use default copy constructor.

**Destructors**

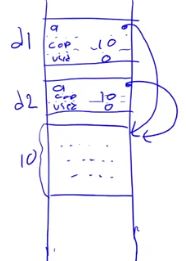
* Automatically called when object is out of scope.
* Default version only removes ordinary variables, not dynamic variables.
* One class can only have one destructor.

**Partially Filled Array (Primitive version of vector) CHECK 10-12**

**DEFAULT COPY CONSTRUCTOR (SHALLOW COPY)**

PFArrayD d1(10);

sizeof(d1) 🡪 sizeof(double\*) + sizeof(int) + sizeof(int)

If you do PFArrayD d2(d1); ----------------------------------------------->

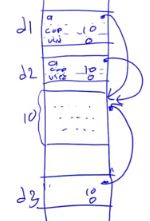
WE ARE USING DEFAULT COPY CONSTRUCTOR HERE, ASSUME THAT WE DON’T WRITE ANY COPY CONSTRUCTOR.

If you change one of the elements of a in d1, that element that is in d2 will also change.

If object d1 dies, my destructor will deallocate the sized 10 location in the heap. If d2 continues living, it’s a member will point to the location that doesn’t belong to me anymore.

So we can say default behaviour of the copy constructor doesn’t work.

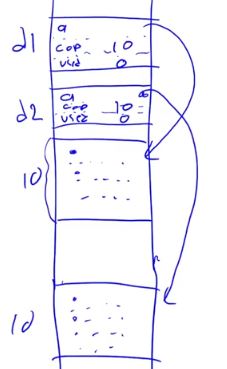
(There is same problem with DEFAULT assignment operator, when we assign one class to another.)



If you do PFArrayD d3; d3 = d1; ---------------------------------------->

*IT IS ALWAYS GOOD IDEA TO PUT EXPLICIT IN FRONT OF SINGLE PARAMETER CONSTRUCTORS BECAUSE THEY MAY ACT LIKE CONVERSION CONSTRUCTOR!*

**DEEP COPY**

This way of copying is much better, we did this in 10-12 in PFArrayD class with copy constructor and overloading “=” operator.

You need to write your own copy constructor and “=” operator whenever you need to do this kind of deep copy.

If you have a vector as the data member, then you should be fine. Don’t write the copy constructor.

If you need to implement memory allocation, deep copying etc. you need to implement big three.

Usually big three is implemented when there is a pointer in your class.

Usually if one of them is needed, other 2 are needed too.

If you don’t use big three you should comment in the class like “//big three is not needed”, so that your customer know that you didn’t implement on purpose.

**BIG THREE:**

1. Copy constructor

2. Assignment operator

3. Destructor

**Overloaded = Operator Definition**

Assignment operator returns reference so assignment chains are possible.

Operator must return same type as it’s left-hand side to allow chains to work

* s1 = s2 must return object of s1’s type.
* The this pointer will help

Uses string class example:

StringClass& StringClass::operator=(const StringClass& rtSide)

{

if (this == &rtSide) *//if right side same as left* side

return this;

else

{

capacity = rtSide.length;

length = rtSide.length;

delete [] a;

a = new char[capacity];

for (int l = 0; l < length; l++)

a[l] = rtSide.a[l];

return \*this;

}

}