Assigning objects and copying constructors

copyConstructorClones.key (newWeek5_6_7)
Destructors in C++

• A destructor is called when an object is deleted

• Objects on the stack (e.g. instance fields declared as $B b()$, are deleted when the stack for the function they are declared in is popped.

• Objects on the heap (e.g. allocated with new as in $B* b = new B()$) are deleted when delete is called on them
What are they good for?

- Objects often point to other heap allocated objects

- An object being deleted is often the only thing pointing to a heap allocated object
  - If the object is deleted without deleting what it points to a memory leak will result

- Destructors allow pointed to objects to be deleted

- Other cleanup actions can also be performed, i.e. decrementing a global counter of items represented by the object
An example of a destructor

class X {
    int* ptr;
    int size
public:
    X() { allocate ptr array }
    ~X() {delete[ ] ptr;}
};

When the object is deleted the destructor deletes the array pointed to by ptr
Copy constructors in C++

By default, \( x_2 = x_1 \) says copy \( x_1 \), byte by byte, into \( x_2 \)’s memory

By default, \( x_3 = x_2 \) says the same thing

Easy and works because \( X \) has no pointers

Shallow copy is ok

```cpp
class X {
    int n;
public:
    X( ... ) { ... }
}

void f( ) {
    X x1( ... );
    X x2 = x1;
    X x3;
    x3 = x2;
}
```
What if X has pointers?

class X {
    int* ptr;
    int size
public:
    X( ) { allocate ptr array}
    ~X( ) {delete[ ] ptr;}
};

void f( ) {
    X x1( );
    X x2 = x1;
    X x3;
    x3 = x2;
}

• Let x2 be initialized as before.
• then x1.ptr, x2.ptr and x3.ptr all point to the same memory
• ~X for x1, x2 and x3 all try and delete the same memory
• 3 attempts will be made to delete memory pointed to by ptr
• This is an undefined operation and can cause the program to crash or worse.
How do we fix this?

- Assume we want each \( X \) object to have its own array, even when copied from another \( X \) object.
- The problem is that the default assignment of \( x_1 \) to \( x_2 \), and \( x_2 \) to \( x_3 \) do a bit for bit copy, including the value of ptr.
- Need to specify a different semantics for what it means to copy an object.
- Again, the language/compiler/runtime cannot read our mind as to what needs to be done, so defining what needs to be done is left as an exercise for the programmer.
- A copy constructor is the C++ mechanism to define these actions.
Copy constructors $T(const\ T&)$

- $T(const\ T&)$; is the prototype for the copy constructor for type (class) $T$
- $const$ not necessary but makes the constructor work for $const$ and non-$const$ objects
- Sometimes we need to change the parameter, e.g. if some concept of ownership changes with the assignment of a referenced object -- $const$ will not be appropriate in this case
- $T&\ \text{operator}=(const\ T&)$; is the prototype for the overloaded $=$ operator for type $T$. 
A copy constructor for X

X( const X& obj) {
    size = xobj.size;
    int* ptr = new int[size];
    for (int i = 0; i < size; i++) {
        ptr[i] = xobj.ptr[i];
    }
}

- **copy primitive size**
- **allocate new pointed-to data**
- **copy the contents of the old pointed-to data to the new pointed-to data**
Should also override the 

\[ \times \& \ \text{operator} \] 

\[ \times \& \ \text{operator}=(\text{const} \ \times \& \ \text{xobj}) \{ \]

\begin{verbatim}
if (this != &xobj) {
  delete [ ] ptr;
  size = xobj.size;
  ptr = new int[size];
  for (int i; i < size; i++)
    ptr[i] = xobj.ptr[i]
}
return *this;
\end{verbatim}

Note the syntax (use of the keyword "operator") for defining the "method name"

- Let's look at this in more detail
The `=` operator

```cpp
X& operator=(const X& xobj) {
    if (this != &xobj) {
        delete [] ptr;
        size = xobj.size;
        ptr = new int[size];
        for (int i; i < size; i++)
            ptr[i] = xobj.ptr[i]
    }
    return *this;
}
```

*This* refers to `xobj_1`  Stated differently, the "=" operator method called belongs to the left hand side object

Kind of like `xobj_1.assign(xobj_2);`

`size` and `ptr` refer to `xobj_1`

`xobj_1`'s pointer and make `ptr` refer to fresh memory (why?)

Do the assignment of values from lhs to rhs `xobj`
The \texttt{= operator}

\begin{verbatim}
X& operator=(const X& xobj) {
    if (this != \&xobj) {
        delete [ ] ptr;
        size = xobj.size;
        ptr = new int[size];
        for (int i; i < size; i++)
            ptr[i] = xobj.ptr[i]
    }
    return *this;
}
\end{verbatim}

Why the line
\begin{verbatim}
    if (this != xobj)?
\end{verbatim}

Consider the legal statement:
\begin{verbatim}
    xojb_1 = xobj_1;
\end{verbatim}
The = operator

X& operator=(const X& xobj) {
    if (this != &xobj) {
        delete [] ptr;
        size = xobj.size;
        ptr = new int[size];
        for (int i; i < size; i++)
            ptr[i] = xobj.ptr[i]
    }
    return *this;
}

Why not return a `void`?

C++ allows chained assigns:

```
xobj_1 = xobj_2 = ... = xobj_n;
```

All assigns after the first (rightmost) use the this value from the previous assignment as the lhs
Why a \textit{const} \texttt{T\&} parameter?

- What if the prototype was just
  \begin{verbatim}
  X\& \texttt{operator\textbar{}=(const X xobj)};
  \end{verbatim}

- To invoke the equal operator would need to do a copy of \texttt{xobj} to pass to \texttt{operator\textbar{}}, and then copy it again into the left hand side.

- What if the prototype was just
  \begin{verbatim}
  X(const X xobj);
  \end{verbatim}

- To invoke the equal operator would need to do a copy of \texttt{xobj} to pass to the constructor that is supposed to make a copy of \texttt{xobj}!
When to use copy constructor and operator=

- If a shallow copy is not sufficient, you need to use a copy constructor.
- If a copy constructor is needed for a copy, you will want to have the same intelligence when assigning an object to another object.