Destructors, object assignments and copy constructors

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 the pointer to them
- Calling *delete* on a reference is legal, but is considered bad practice
  - References should be bound to objects whose storage management is handled elsewhere
  - For stack allocated objects, elsewhere is the C++ system

The destructor calls in this example

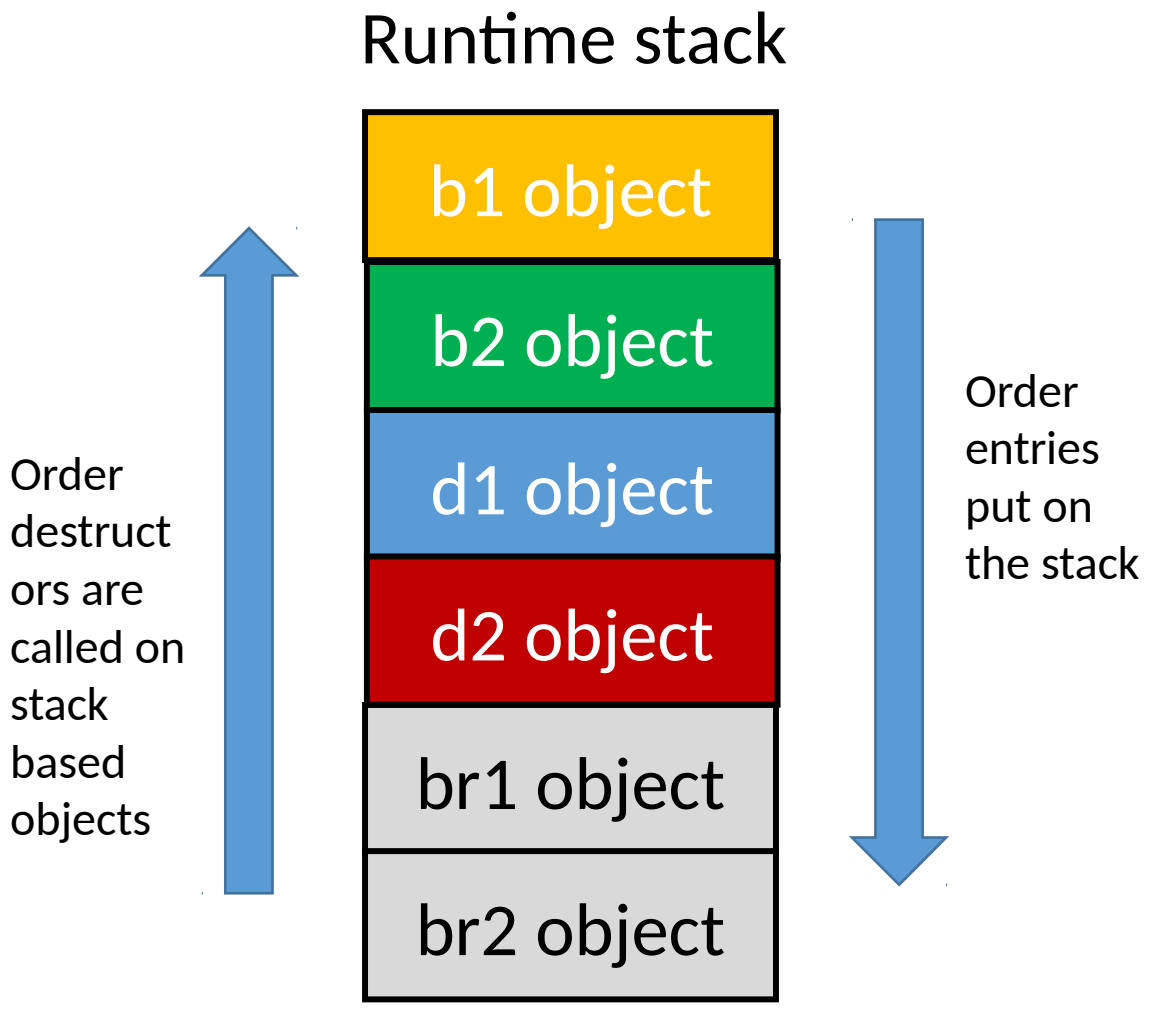
The full output from running this program is shown below. Red items have already been discussed.

object 50
object 100
object 150
object 102 103
object 150 103
object 150 103
deleting object 150 103
deleting object 150
deleting object 100 101
deleting object 100
deleting object 150
deleting object 150

- Destructors called as object leave the stack, i.e., the destructor on the last object on the stack is called first
- Destructors always call the base class destructor automatically -- you don't have to do this.

```
int main(int argc, char * argv[ ]) {
   B b1(50);
   B b2(150);
   D d1(100, 101);
   D d2(102, 103);
   . . .
   B& br1 = b1;
   . . .
   B& br2 = (B&) d2;
```

deleting object 150 103 *D part of d2 object*
deleting object 150 B part of d2 object
deleting object 100 101 *D part of d1 object*
deleting object 100 *B part of d1 object*
deleting object 150 *B part of b2 object*
deleting object 150 *B part of b1 object*



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 objects that are pointed-to to be deleted
  - Other cleanup actions can also be performed, e.g., decrementing a global counter of items represented by the object, closing a file, etc.

Destructor example (see Destructor)

```
class X {
public:
   X( );
   virtual ~X( );

private:
   int *ages;
};
```

```
X::X( ) {
   ages = new int[1000];
   std::cout << "allocated ages" << std::endl;
}

X::~X( ) {
   delete ages;
   std::cout << "freed ages" << std::endl;
}
```

- When an object is deleted (either b calling *delete* on a pointer to the object or it is popped from the stack) the destructor is called.
- The destructor frees the array allocated by the constructor

Destructor example (see Destructor)

```
int main(int argc, char * argv[ ]) {
  X xStack;

  std::cout << "before new" << std::endl;
  X *xHeap = new X( );

  std::cout << "delete xHeap" << std::endl;
  delete xHeap;
  std::cout << "delete xHeap finished" << std::endl;

  return 0;
}
```

```
allocated ages
before new
allocated ages
delete xHeap
freed ages
delete xHeap finished
freed ages
```

Copying objects, copy constructors and overload = operators in C++

NoCopyConstructor

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

int main(int argc, char * argv[ ]) {
  X x1;
  X x2;
  X x3;
  x2 = x1;
  x3 = x2;
}
```

- By default, *x2 = x1* says to copy x1, bit by bit, into x2's memory
- By default, *x3 = x2* says the same thing
- For simple object, i.e., objects that don't point to other objects, arrays, structs, etc., this often ok
- Sometimes we need something more sophisticated

What if X has pointers? (copyConstructor)

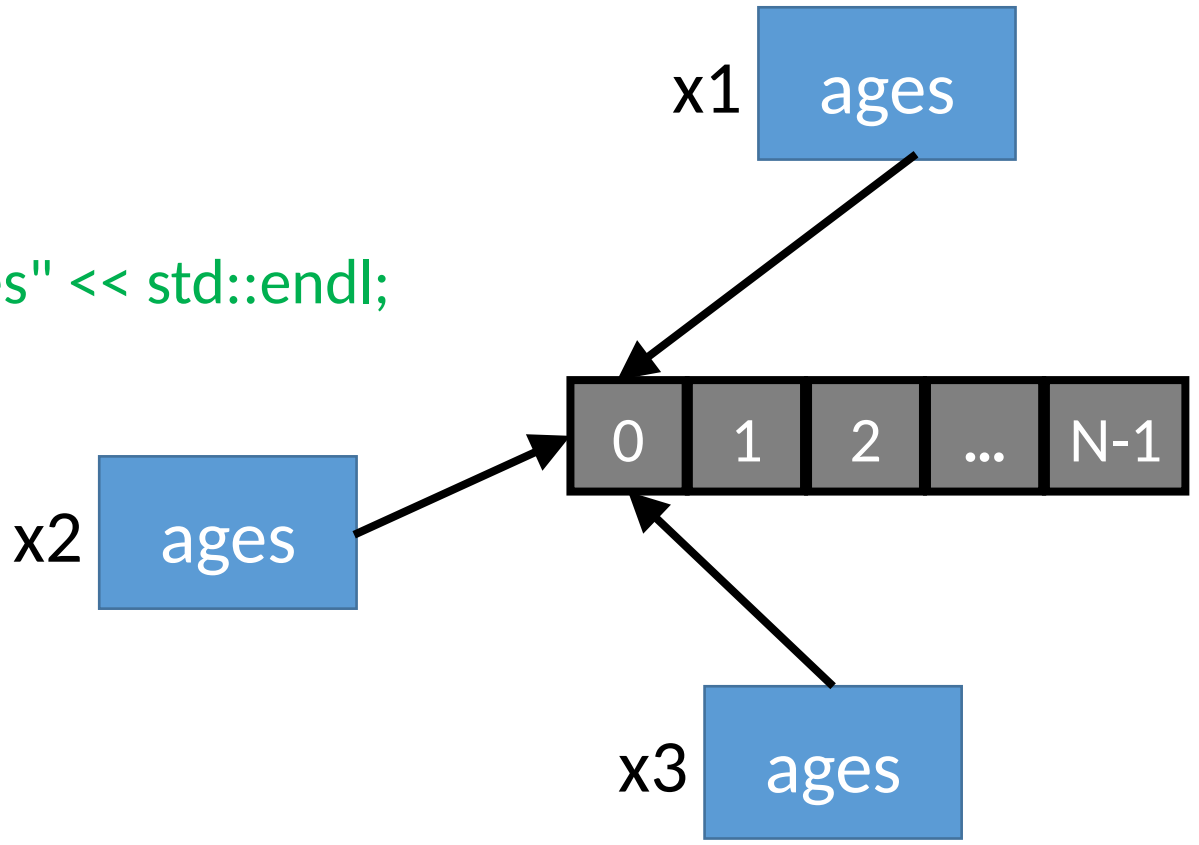
```
class X {
public:
   X( );
   virtual ~X( );

private:
   int *ages;
};

int main(int argc, char * argv[ ]) {
   X x1;
   X x2;
   X x3;
   x2 = x1;
   x3 = x2;
}
```

```
X::X( ) {
   ages = new int[1000];
   std::cout << "allocated ages" << std::endl;
}

X::~X( ) {
   delete ages;
   std::cout << "freed ages" << std::endl;
}
```

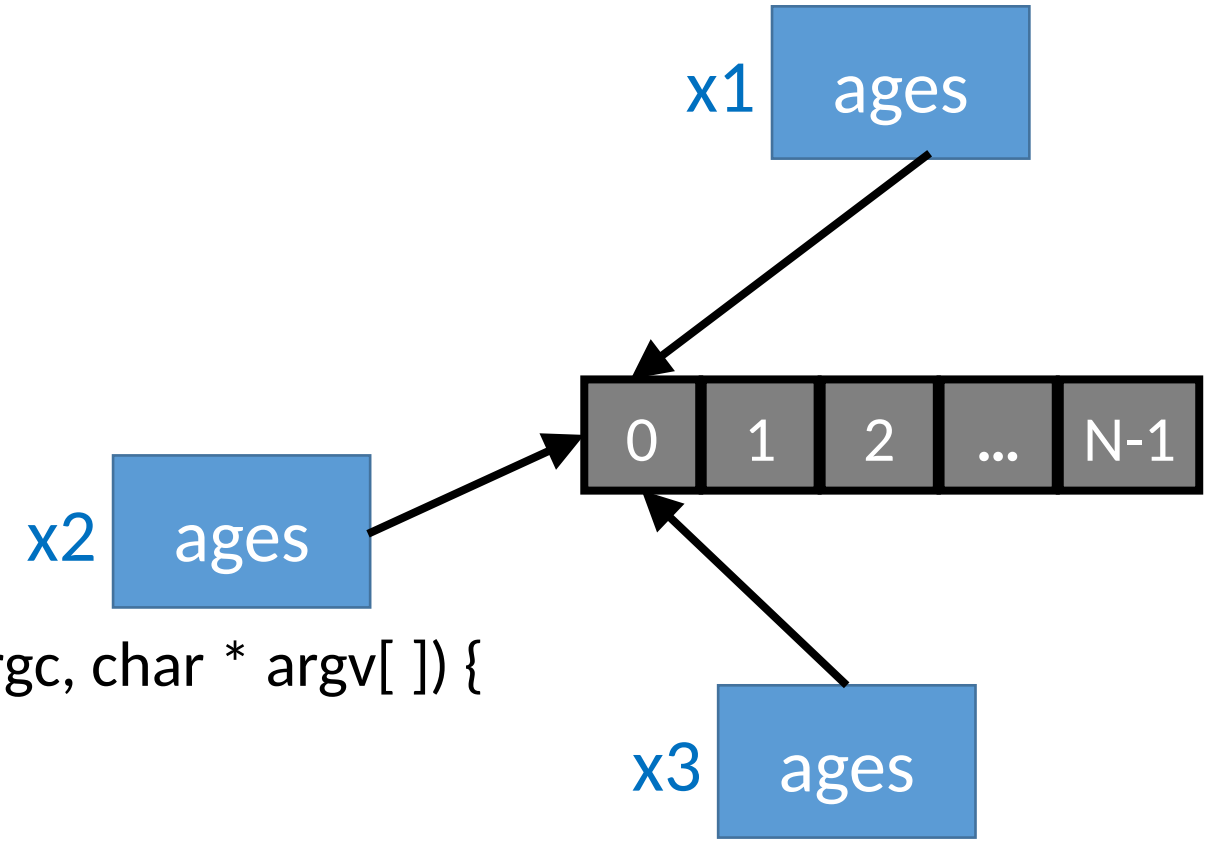


What if X has pointers? (needCopyConstructor)

```
class X {
public:
   X( );
   virtual ~X( );

private:
   int *ages;
};

int main(int argc, char * argv[ ]) {
   X x1;
   X x2;
   X x3;
   x2 = x1;
   x3 = x2;
}
```



- Because of the bit for bit copy, then x1.age, x2.age and x3.age all point to the same array.
- ~X for x1, x2 and x3 all try and delete the same array pointed to by their age pointers
- 3 attempts will be made to delete memory pointed to by age
- This is an undefined operation and can cause the program to crash or worse.

How can 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 *x1* to *x2*, and *x2* to *x3* do a bit for bit copy, including the value of *ages*.
- Need to specify a different semantics for what it means to copy an object.
- 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.
- An overloaded = operator and a *copy constructor* are the C++ mechanisms to define these actions

```
int main(int argc, char * argv[ ]) {
   X x1;
   X x2;
   X x3;
   x2 = x1;
   x3 = x2;
}
```

Zero arg constructor used to initialize x1, x2 and x3

C++ supplied equal operator used for the equal. It does a bit for bit copy from rhs into lhs. ***No constructor called!***

```
*** Error in `./a.out': double
free or corruption (!prev):
0x0000000000c1d010 ***
```

Need to override the “=“ operator

- +, -, ++, << etc. can be overridden
- In the context of copy constructors, overriding the "=" operator is important
- When overriding an operator, say “=”, for objects t1 and t2 of type T, t1 = t2 invokes the overloaded “=” function
- If the overloaded function is defined within the T class then t2 is passed as an argument to the function and T1 is passed as the *this* parameter
- Call is analogous to (but this syntax is not correct C++!) t1.operator=(t2), i.e., invoking the “operator=” function on t1 and passing t2 as an argument

```
X& X::operator=(const X& x) {
  if (this != &x) {
    delete [ ] ages;
    ages = new int[x.size];
    for (int i; i < x.size; i++)
      ages[i] = x.ages[i];
  }
  std::cout << "X::operator=" << std::endl;
  return *this;
}
```

- x1 = x2: the *this* pointer refers to x1, i.e., it is like a call to x1.operator=(x2);
- ages refers to x1.ages.
- Why the blue test?
- Why do we delete this.ages and reallocate it?

Why not return a void instead of an X reference?

```
X& X::operator=(const X& x) {
  if (this != &x) {
    delete [ ] ages;
    ages = new int[x.size];
    for (int i; i < x.size; i++)
      ages[i] = x.ages[i];
  }
  std::cout << "X::operator=" << std::endl;
  return *this;
}
```

- C++ allows chained assignments
- *X1 = x2 = x3 = x4;* is legal
- All assigns after the rightmost use the value resulting from the previous assignment

3 2 1

X1 = x2 = x3 = x4;

The other place object copies occur is with parameter passing

(CopyConstructor main2)

```
void f(X x) { }

int main(int argc, char * argv[ ]) {
   X x1;
   f(x1);
}
```

- When a copy of the object is made, a new object is created and initialized by a constructor
- C++ gives us a default constructor that does a bit-for-bit copy.
- We can replace that with our own *copy constructor*

Copy Constructors *T(const T&)*

- Given a class T, a constructor declared as *T(const T&)* is *a copy constructor*
- *const* not necessary, but it makes the constructor work with *const* objects
- Sometimes we need to change the parameter and in these cases const would not be appropriate.

A copy constructor for X (CopyConstructor)

If array size is different for different objects, copied object should have a getter function to get the array size, e.g. *getAgesLen( );*

```
#define UB 1000

X::X(const X& x) {
   ages = new int[UB];
   for (int i=0; i < UB; i++)
      ages[i] = x.ages[i];
   std::cout << "X::X(const X&)" << std::endl;
}
```

Allocate new *ages* array for the receiving object

Copy the contents of the *ages* vector for the old X object into the *ages* vector of the new X object

This constructor will be called to make the copy

```
void f(X x) { }

int main(int argc, char * argv[ ]) {
   X x1;
   f(x1);
}
```

- Whenever a copy of an object of type T is needed, C++ looks for a *copyconstructor* , i.e., a constructor whose signature is T(T&) or T(const T&), and if found, calls it.
- If not found, C++ uses the default constructor which does a bit-for-bit copy of the object to be copied.

```
X::X( )// constructor call for x1
X::X(const X&)// Copy constructor called to pass x1 to f by value
~X( )  // destructor called for copy of x1 passed to f
~X( )  // constructor called for x1 when main exited
```

Why a *const T& X* parameter?

- The *const* allows assignments like *X(x2)* where *x2* is a *const* variable
- What if the signature was *X& operator=(const X xobj)*; (not a reference)?
- Assume *x1(x2);*
  - To invoke the constructor, a copy of x2 to pass to x1's operator= and then copy parts of it again into x1. This is inefficient
  - But even worse, to make this copy the copy construct would again be invoked on x2.
  - This isn't going to finish quickly!

When do we need to define our own copy constructors?

- If a shallow copy is ok, we don’t need to define
  - A shallow copy is when we don’t make copies of objects pointed to from the object, only the object itself
- If a copy constructor is needed, you want to have the same functionality for assignment between objects, and so an operator= needs to be defined.