62. Destructors

Objectives

- Write destructors
- Understand default destructors
- Understand when to overwite destructors
- Understand when destructors are called

Review: Automatic variables and automatic reclaim of memory

Constructors let you initialize objects when they are created. Objects also need to "die" just like "regular" variables, i.e., automatic variables. When I say a variable "dies", I mean that the name of the variable goes out of scope. When a variable goes out of scope, the memory used by the variable in the local scope is reclaimed automatically. (Remember that the memory used in the free store is not reclaimed automatically – the memory in the free store was requested manually by a memory allocation command using `new` and you have to reclaim it yourself using `delete` or `delete[]` .)

Here are some examples and scenarios … MAKE SURE YOU STUDY THEM CAREFULLY!!!

```
if (x > 0)
{
    int i; // i created
    ...
} // i goes out of scope at the end of the
  // if-block and i's memory is reclaimed
  // automatically.
```

```
for (int i = 0; i < 10; i++)
{
    int x[100]; // x created
} // x goes out of scope after the body of
  // the for-loop. x is destroyed and memory
  // for x (100 integers) is reclaimed and
  // execution returns to the update section
  // of the for-loop
```

```
void f()
{
    int i;  // i created

    return; // i goes out of scope and i's memory
            // is reclaimed automatically.
}
```

```
void g()
{
    double * i; // i created
    i = new double[100]; // 100 doubles allocated
                         // in the heap (and i's
                         // value is to set to the
                         // address of the first
                         // double of this array of
                         // doubles)
```

```
    return; // i goes out of scope and i's memory is
            // reclaimed automatically. However the
            // 100 doubles allocated from the heap is
            // not deallocated ... MEMORY LEAK!
}

int main()
{
    g();
    g();
    g();
    g();
    return 0;
}
```

All the above is OLD STUFF. If you don't remember, you have to comb through the old notes and look for information on scopes on variables.

Now …

Automatic objects and automatic reclaim of memory

What happens when an **object** goes out of scope? Memory of all **automatic instance variables** of the object will be reclaimed automatically:

```
if (x > 0)
{
    Date date = Date(1970, 1, 1);
    ...
}    // date goes out of scope after the if-block.
     // date.yyyy, date.mm, date.dd are int variables
     // which are *automatic* and are therefore
     // automatically reclaimed. No memory leak.
...
```

Of course in the case where the object is in the free store, you will have to delete it manually. So this is BAD:

```
if (x > 0)
{
    Date * p = new Date(1970, 1, 1);
    ...
}    // p goes out of scope after the if-block. p is
     // automatic since it's in this local scope.
     // So the memory used for pointer p is
     // automatically reclaimed. However the object
     // in the free store that p points to is not
     // reclaimed!!!  MEMORY LEAK!!!
...
```

It should be this instead:

```
if (x > 0)
{
    Date * p = new Date(1970, 1, 1);
    ...
    delete p; // Deallocate the memory used by p,
              // i.e., the object that p points to is
              // reclaimed by the free store. In this
              // since all instance variables in Date
              // objects are automatic, their memory
              // used are automatically reclaimed.

}    // ... and p goes out of scope after the
     // if-block. p is automatic since it's in this
     // local score.
     // So the memory used for pointer p is
     // automatically reclaimed. No memory leak.
...
```

Likewise this is correct:

```
if (x > 0)
{
    Date * p = new Date[1000];
    ...
```

```
    delete []  p; // Deallocate the memory used by p,
                  // i.e., the 1000 objects that p
                  // points to is reclaimed by the
                  // free store ...

}    // ... and p goes out of scope after the
     // if-block. p is automatic since it's in this
     // local score.
     // So the memory used for pointer p is
     // automatically reclaimed. No memory leak.
...
```

Destructors!!!

However, in general, some objects are more complicated and might need special **"cleaning up".** This is because some objects **acquire resources (such as memory) which are not automatically released**.

Here's an example:

```
class Thingy
{
public:
    Thingy()
        : y_(new char) // memory allocation during
    {}                 // constructor call

private:
    int x_;      // x_ is automatic
    char * y_;   // y_ is automatic, but the *memory*
                 // that y_ points to, if allocated,
                 // is *not* automatic.
};

int main()
{
    int x = 1;
    if (x > 0)
    {
        Thingy thing; // memory for thing.x_,
                      // thing.y_ and *(thing.y_)
                      // allocated.

        // do something with thing

    } // At the end of the body of the if-statement,
      // memory for thing.x_, thing.y_ is reclaimed
      // automatically ...
      // BUT the memory that thing.y_ points to is
      // NOT deallocated!!!

    return 0;
}
```

What's the problem? Note that `thing.y_` has requested for memory in the constructor. So I really need to execute

```
delete thing.y_;
```

before `thing` goes out of scope!!! In other words I should do something like this:

```
class Thingy
```

```
{
    ...
};

int main()
{
    int x = 1;
    if (x > 0)
    {
        Thingy thing;
        // do something with thing

        // *** delete thing.y_ ***
    }

    return 0;
}
```

Of course, you can't do that because `thing.y_` is private. No big deal! I can create a method and call that method.

```
class Thingy
{
public:
    ...
    void cleanup()
    {
        delete y_;
    }
...
};

int main()
{
    int x = 1;
    if (x > 0)
    {
        Thingy thing;
        // do something with thing

        thing.cleanup();
    }

    return 0;
}
```

But what if I forget?

WAIT … don't even do that!!! The destructor comes to the rescue!!!

A **destructor** is a member function that is **automatically called** when an object is about to be destroyed (i.e., about to go out of scope):

```
class Thingy
{
```

```
    ...
};

int main()
{
    int x = 1;
    if (x > 0)
    {
        Thingy thing;
        // do something with thing

    } // thing calls its destructor here just
      // before thing goes out of scope.

    return 0;
}
```

(The above is for the case where the object is in the local scope. I'll talk about objects in the free store in a bit.)

TADA!!!

… so what???

Well, let me write a destructor for the `Thingy` class. But first ...

There are three rules for writing a destructor …

- For class `C`, the name of the destructor must be `~C`.
- Destructors have an empty list of parameters – no parameters, i.e., the prototype must be `~C()`.
- Destructors do not have return type. You can't even put `void` as a return type. (This is just like constructors.)

So I do the following:

```
class Thingy
{
public:
    Thingy()          // memory allocation
        : y_(new char) // during constructor call
    {}
    ~Thingy()
    {
        delete y_;
    }

private:
    int x_;     // x_ is automatic
    char * y_;  // y_ is automatic, but the *memory*
                // that y_ points to, if allocated,
                // is not automatic.
};

int main()
```

```
{
    int x = 1;
    if (x > 0)
    {
        Thingy thing; // memory for thing.x_,
                      // thing.y_, *(thing.y_)
                      // allocated.

        // do something with thing

    } // At the end of the body of the if-statement,
      // memory for thing.x_, thing.y_ is reclaimed
      // automatically ...
      // BUT before that ... (and this is the point):
      // thing.~Thingy() executes so that the
      // memory that thing.y_ points to is reclaimed
      // by the free store.

    return 0;
}
```

And note that in my code in `main()` ... I do not have to write code to deallocate the memory used by `thing.y_` because that's written in `Thingy.~Thingy()` and `thing` will call this method automatically just before going out of scope. I don't have to worry about forgetting to deallocate memory used by `thing` (i.e. through `thing.y_`). The danger of accidental memory leaks is reduced.

TADA!!! GET IT!!!

It's a common practice to do the following:

- A destructor is placed just after the constructor in the class definition.
- Destructors are usually inlined.

Exercise. Do a simple experiment yourself to show that the destructor is indeed called:

```
class Thingy
{
public:
    Thingy()          // memory allocation
        : y_(new char) // during constructor call
    {}
    ~Thingy()
    {
        std::cout << "Thingy::~Thingy()\n";
        delete y_;
    }

private:
    int x_;      // x_ is automatic
    char * y_;   // y_ is automatic, but the *memory*
                 // that y_ points to, if allocated,
```

```
                  // is not automatic.
};

int main()
{
    int x = 1;
    if (x > 0)
    {
        Thingy thing; // memory for thing.x_,
                      // thing.y_, *(thing.y_)
                      // allocated.

        // do something with thing

    } // At the end of the body of the if-statement,
      // memory for x, y_ is reclaimed
      // automatically ...
      // BUT before that ... (and this is the point):
      // thing.~Thingy() executes so that the
      // memory that thing.y_ points to is reclaimed
      // by the free store.

    return 0;
}
```

Exercise. What is the name of the destructor for the `WeatherCtrl` class?

Destructors are used (in classes) to include code for releasing resources acquired, not just memory. For instance, there are other resources like:

- Member variable might be a pointer
- Member variable might be holding onto the CD player
- Member variable might be holding a file
- Member variable might be a network connection
- Etc.

Let's write a destructor for `Date`. Technically we don't have to clean up anything since all instance members (or member variables) in `Date` are automatic. This is just a drill and also to check that `Date` objects do call their destructor just before they die.

```
// Date.h
...
class Date
{
public:
    ...
    ~Date()
    {
        std::cout << "Date::~Date ...\n";
    }
    ...
```

```
}
...
```

```
// main.cpp
...
int main()
{
    ...
    for (int i = 0; i < 5; i++)
    {
        std::cout << '\n' << i << "... \n";
        Date date(1970, 1, 1);
        date.print();
        // does date call its destructor ~Date?
    }

    return 0;
}
```

Default destructor

Since objects call their destructors just before they die (i.e. just before they go out of scope) and our previous version of `Date` does not have a destructor, how come the previous version can still compile and run??? Shouldn't your C++ compiler yell at you and tell you that `~Date()` is not found???

No.

Because ... if you don't specify a destructor for a class, the C++ compiler actually includes a do-nothing destructor in the class. This is the **default destructor**. The default destructor is a do-nothing method because it has an empty body:

```
class C
{
public:
    ...
    ~C() {} // default destructor of C
    ...
};
```

In the case of the earlier `Date` class, C++ included this destructor into the `Date` class when there was no destructor:

```
class Date
{
public:
    // constructors
    ~Date() {}
    // other methods
};
```

If you do write a destructor, then your C++ compiler will not insert a default destructor for you.

Note that the default destructor supplied by C++ does not do anything. Since all instance variables are automatic in `Date`, a destructor is not necessary. It's still a good (and common) practice to include destructors even if it's not necessary. Being explicit is better than leaving things implicit.

Exercise. What is the order in which objects call their destructor? (This is important!)

```
int main()
{
    Date date1(1970, 1, 1);
    Date date2(1985, 1, 1);
    Date date3(2010, 10, 10);

    return 0;
}
```

Why is knowing the order important? Because in complex systems, there might be some dependencies between objects so that some object `x` can exist only if object `y` exists, i.e., `y` must be constructed before `x` and `x` must be destroyed before `y`.

```
int main()
{
    Date date1(1970, 1, 1);
    Date date2(1985, 1, 1);
    Date date3(2010, 10, 10);
    return 0; // date3 calls its destructor,
              // *then* date2 calls its destructor,
              // *then* date1 calls its destructor.
}
```

So, remember this: **the order of destructor calls is always opposite to the order of constructor calls.**

Exercise. Is it possible to call the destructor "manually"?

```
int main()
{
    Date date(1970, 1, 1);
    date.~Date();

    return 0;
}
```

Exercise. What is the output?

```
#include <iostream>

class A
{
public:
    ~A() { std::cout << "A::~A\n"; }
};

class B
{
public:
    ~B() { std::cout << "B::~B\n"; }
};

class C
{
public:
    ~C() { std::cout << "C::~C\n"; }
};

int main()
{
```

```
    A a;
    B b;
    for (int i = 0; i < 5; i++)
    {
        C c;
        if (i > 3)
        {
            A a;
            if (i == 4) B b;
        }
    }

    return 0;
}
```

Exercise. Write a destructor for `WeatherCtrl` and test it by inserting a print statement in the destructor to check that it is called. Are all the instance variables automatic? So, is the default destructor good enough?

Exercise. Write a destructor for `Vehicle` and test it by inserting a print statement in the destructor to check that it is called. Are all the instance variables automatic? So, is the default destructor good enough?

Exercise. Write a destructor for `Being` and test it by inserting a print statement in the destructor to check that it is called. Is the default destructor good enough?

Exercise. Write a destructor for `IntArray` and test it by inserting a print statement in the destructor to check that it is called. Is the default destructor good enough?

Objects in the free store

The above is for an object in the local scope. For the case of an object in the free store, say a pointer `p` is pointing to this object, when we execute `delete p`, then the object will call its destructor automatically. For the case of a pointer `p` pointing to an array of objects in the free store, when we execute `delete [] p`, each object in the array will call its destructor. Again: You do **not** call destructors explicitly. They are called automatically. (In fact you cannot call destructors in your code.)

```
class Thingy
{
public:
    Thingy()            // memory allocation
        : y_(new char) // during constructor call
    {}

    ~Thingy()
    {
        delete y_;
    }

private:
    int x_;      // x_ is automatic
    char * y_;   // y_ is automatic, but the *memory*
                 // that y_ points to, if allocated,
                 // is not automatic.
};

int main()
{
    int x = 1;
    if (x > 0)
    {
        Thingy * pthing = new Thingy;

        // do something with *pthing

        delete pthing; // Memory used by the object
                       // that pthing points to is
                       // decallocated - reclaimed
                       // by the free store. But
                       // before that, this object
                       // (*pthing) calls its
                       // destructor
                       // (*pthing).~Thingy().
    }

    return 0;
}
```

The story is similar if you have a pointer pointing to an array of objects in the free store:

```
int main()
{
    int x = 1;
    if (x > 0)
    {
        Thingy * p = new Thingy[100];
        // do something with p[0],...,p[99]

        delete [] p; // p[0], ..., p[99] call
                     // their destructors
    }

    return 0;
}
```

IntPointer

When you want a pointer, you declare it and allocate memory (using `new`) from the heap for the pointer to point to. You never forget to `new` the pointer because your program will not work immediately if you don't `new`.

But it's easy to forget to deallocate / release the memory (using `delete`) after you're done with using the value that `p` points to. The problem is this: the program will continue to work until it runs out of memory which will be much later. By then you have to find that one bad pointer among hundreds or thousands of pointers in your code.

Here's our `IntPointer`:

```
...
class IntPointer
{
public:
    IntPointer(int x = 0)
        : p_(new int)
    {
        *p_ = x;
    }

    IntPointer(const IntPointer & intptr)
        : p_(new int)
    {
        *p_ = *(intptr.p_);
    }

    int & operator*()
    {
         return *p_;
    }

    void deallocate()
    {
        delete p_;
    }

private:
    int * p_;
};
...
```

```
...
int main()
{
    for (int i = 0; i < 5; i++)
    {
        IntPointer intptr(i);
    }
    return 0;
}
```

Note that each of the five `intptr` contain a pointer `p_` which points to an allocated integer in the free store. Note however that the integer `intptr.p_` points to is not deallocated. You can see the point in time when each `intptr` dies (and when one additional memory leak occurs). Add the following destructor:

```
...
class IntPointer
{
public:
    ...
    ~IntPointer()
    {
        std::cout << "memory leak ... :(\n";
    }
...
```

Run your program again. If you don't believe me that you'll run out of memory, resulting in program abort, try this:

```
...
int main()
{
    while (1)
    {
        IntPointer intptr(0);
    }

    return 0;
}
```

(You may want to comment out the print statement in the destructor because printing slows down your program and delays the program from crashing.)

Now we make our pointer object smart enough to deallocate memory used by itself:

```
...
class IntPointer
{
public:
    ...
    ~IntPointer()
    {
        delete p_;
    }
...
```

Run this again and see if you have memory leaks:

```
...
int main()
{
    while (1)
    {
        IntPointer intptr(0);
    }
    return 0;
}
```

You won't.

Now, if ever we need an integer pointer, we can use this class and we will never have to worry about memory leaks. (There are a few more things to add to this class to really complete it …)

Typically, if you have a class that grabs hold of some resource (not just memory) in the constructor of the class, you should probably include code in the destructor to release the resource.

Exercise. Are we done improving `IntPointer`? Not quite. Why is the following a problem? Trace it by hand and draw a diagram.

```
...
int main()
{
    IntPointer p1(42);
    IntPointer p2(0);
    p2 = p1;
    return 0;
}
```

You should be able to explain why there's a problem. There are two possible solutions. What are they? (This pretty much depends on what you want `operator=` to mean … I'll be talking about operators later.) We'll have to come back to this issue again later and redefine `operator=` in the `IntPointer` class.

Exercise. Is the default destructor good enough for the following class? Run the program below.

```
class C
{
public:
    C(int x) : u_(x)
    {
        if (u_ <= 0)
        {
            t_ = new char;
        }
        else
        {
            t_ = new char[u_];
        }
    }
private:
    char s_[100];
    char * t_;
    const int u_;
};

int main()
{
    int i = 0;
    while (1)
    {
        if (i % 2 == 0)
        {
```

```
            C c(i);
        }
        else
        {
            C c(0);
        }
    ++i;
    }

    return 0;
}
```

If the default destructor is not good enough, write a destructor that does the job. Test run it in a long while loop and see if your program crashes.

IntDynArray

Here's our `IntDynArray` with the placeholder `deallocate()` function:

```
...
class IntDynArray
{
public:
    ...

    void deallocate()
    {
        delete [] x_;
    }

    ...
private:
    int * x_;
    int size_;
    int capacity_;
};
...
```

Exercise. Replace the `deallocate()` function in `IntDynArray` with a destructor. Test it with the following to make sure that there's no memory leak:

```
...
int main()
{
    while (1)
    {
        IntDynArray a(1000);
    }

    return 0;
}
```

There's still a similar problem involving `operator=`:

```
...
int main()
{
    IntDynArry a(100);
    IntDynArry b(100);
    b = a;

    return 0;
}
```

Again, we'll come back to this issue later.

Lazy allocation

There are times when you might want to delay allocating memory (or any resource) because it might not be needed till later.

You can assign any pointer the value of NULL.

NULL (or the NULL pointer) is usually integer 0 (or something similar ... like `(void *)0`). NULL is frequently used as a value to indicate that a pointer has not been allocated memory.

```
#include <cstdlib> // or #include <cstddef>
#include <iostream>
// NULL can be the integer 0 or it can be
// ((void *) 0)

class C{};

int main()
{
    // NULL can be assigned to pointers of any type:
    int * p = NULL;
    char * q = NULL;
    double * r = NULL;
    bool * s = NULL;
    C * t = NULL;
    std::cout << p << '\n';
    std::cout << (int)NULL << '\n';

    return 0;
}
```

In this case you can do the following: In the constructor, do NOT allocate memory but set the **relevant pointer** to NULL.

Example:

```
...
class LazyIntPointer
{
public:
    LazyIntPointer()
        : p_(NULL)
    {}
...
private:
    int * p_;
};
...
```

When do you allocate memory? Only when you need to use the value that p_ points to:

```
...
class LazyIntPointer
{
public:
    ...
    int & operator*()
    {
        // Allocate memory only when p_ is NULL
        if (p_ == NULL)
        {
            p_ = new int;
        }
        return *p_;
    }
    ...
};
...
```

And in the destructor, only deallocate when `p_` is not NULL, i.e., only when `p_` has actually been allocated memory:

```
...
class LazyIntPointer
{
public:
    ...
    ~LazyIntPointer()
    {
        // De-allocate memory only when p is not NULL
        if (p_ != NULL)
        {
            delete p_;
        }
    }
    ...
};
...
```

This is a very common technique to save on memory usage. (There are many other techniques.)

Summary

Every class can have a special method called the destructor. If `C` is a class, then the prototype of its destructor is `~C()` with no return type.

The default destructor of a class is a method that has an empty body:

```
class C
{
...
    ~C(){} // default destructor of C
...
};
```

If a destructor is not defined in a class, then the C++ compiler will provide the default destructor automatically. If a destructor is defined, then the compiler will not supply the default destructor.

The destructor cannot be called manually by code – it is called automatically.

Just before an object in the local scope goes out of scope, the object calls its destructor.

```
{
    C c;
    …
    // c calls its destructor just before
    // going out of scope
}
```

When an object in the free store is deallocated, the object will call its destructor:

```
{
    C * p = new C;
    ...
    delete p; // (*p) calls its destructor ~C()
}
```

The story is similar when `p` points to an array of objects in the free store:

```
{
    C * p = new C[1000];
    ...
    delete [] p; // p[0], …, p[999] call their
                 // destructors ~C()
}
```

Memory used by automatic instance variables of objects will be automatically reclaimed.

Destructors are called automatically in the opposite order that the constructors were called.

Resources acquired by objects through constructor calls such as memory allocation for pointer instance variables (if not deallocated) must

be deallocated by destructors.