C++ dynamic memory allocation revisited

- `new` and `delete` are essentially the C++ versions of `malloc` and `free`
- Big difference: `new` not only allocates the memory, it also calls the appropriate constructor if needed

C++ dynamic memory allocation revisited

```
// destructors1.cpp

#include <iostream>

class DefaultSeven {
public:
    DefaultSeven() : i(7) { }
    int get_i() { return i; }
private:
    int i;
};

int main() {
    DefaultSeven s;
    DefaultSeven *sptr = new DefaultSeven(); // using new
    std::cout << "s.get_i() = " << s.get_i() << std::endl;
    std::cout << "sptr->get_i() = " << sptr->get_i() << std::endl;
    delete sptr;  // free the memory before exiting
    return 0;
}
```

C++ dynamic memory allocation revisited

```
$ g++ -o destructors1 destructors1.cpp -std=c++11 -pedantic -Wall -Wextra

$ ./destructors1

s.get_i() = 7
sptr->get_i() = 7
```

- `new` called the default constructor for us in both cases
- `delete` releases the memory, but we should also set `sptr` to `NULL`

C++ dynamic memory allocation revisited

- `T* fresh = new T[n]` allocates an array of `n` elements of type `T`
- Use `delete[] fresh` to deallocate – always use `delete[]` (not `delete`) to deallocate a pointer returned by `new T[n]`
- If `T` is a class type, then `T`'s default constructor is called for **each** element allocated
- If `T` is a "built-in" type (`int`, `float`, `char`, etc.), then the values are **not initialized**, like with `malloc`

C++ dynamic memory allocation revisited

```
// destructors2.cpp

#include <iostream>

class DefaultSeven {
public:
    DefaultSeven() : i(7) { }
    int get_i() { return i; }
private:
    int i;
};

int main() {
    DefaultSeven *s_array = new DefaultSeven[10];
    for(int i = 0; i < 10; i++) {
        std::cout << s_array[i].get_i() << " ";
        }
    std::cout << std::endl;
    delete[] s_array;
    return 0;
}
```

C++ dynamic memory allocation revisited

```
$ g++ -o destructors2 destructors2.cpp -std=c++11 -pedantic -Wall -Wextra
$ ./destructors2
7 7 7 7 7 7 7 7 7 7
```

- Confirming that default constructor was indeed called for all 10 elements

C++ classes: destructors

- A class *constructor*'s job is to initialize the fields of the object
  - It's common for a constructor to obtain a resource (allocate memory, open a file, etc.) that should be released when the object is destroyed (deallocate memory, close a file, etc.)
- A class *destructor* is a method called by C++ when the object's lifetime ends or it is otherwise deallocated (ie, with `delete`)
- A destructor's name is the class name prepended with ~, e.g. `~Rectangle()`
- The destructor is **always automatically** called when object's lifetime ends, including when it is deallocated
  - It's a convenient place to clean up

C++ classes: destructors

```
    // sequence.h

1   #ifndef SEQUENCE_H
2   #define SEQUENCE_H
3   #include <cassert>
4   class Sequence { // What does this class do?  Anything wrong with it?
5   public:
6       Sequence() : array(NULL), size(0) { }
7       // Note: constructor can have both an initializer
8       // list and statements in its body
9       Sequence(int sz) : array(new int[sz]), size(sz) {
10          for(int i = 0; i < sz; i++) array[i] = i;
11      }
12      int at(int i) {
13          assert(i < size);
14          return array[i];
15      }
16  private:
17      int *array;
18      int size;
19  };
20  #endif // SEQUENCE_H
```

C++ classes: destructors

```
// destructors3.cpp

#include <iostream>
#include "sequence.h"

int main() {
    Sequence seq(10);
    for(int i = 0; i < 10; i++) {
        std::cout << seq.at(i) << ' ';
    }
    std::cout << std::endl;
    return 0;
}
```

```
$ g++ -o destructors3 destructors3.cpp -std=c++11 -pedantic -Wall -Wextra

$ ./destructors3

0 1 2 3 4 5 6 7 8 9
```

C++ classes: destructors

```
$ valgrind --leak-check=full ./destructors3

==4559== Memcheck, a memory error detector
==4559== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==4559== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==4559== Command: ./destructors3
==4559==
==4559== error calling PR_SET_PTRACER, vgdb might block
==4559==
==4559== HEAP SUMMARY:
==4559==     in use at exit: 72,744 bytes in 2 blocks
==4559==   total heap usage: 3 allocs, 1 frees, 76,840 bytes allocated
==4559==
==4559== 40 bytes in 1 blocks are definitely lost in loss record 1 of 2
==4559==    at 0x4C2E80F: operator new[](unsigned long) (in /usr/lib/valgrind/vgpreload_memcheck-amd64-lin
==4559==    by 0x400AD2: Sequence::Sequence(int) (in /d/Study/PhDCS(JHU)/JHU2020/2020-09-12(Fall) - Interm
==4559==    by 0x4009E9: main (in /d/Study/PhDCS(JHU)/JHU2020/2020-09-12(Fall) - Intermmediate Programming
==4559==
==4559== LEAK SUMMARY:
==4559==    definitely lost: 40 bytes in 1 blocks
==4559==    indirectly lost: 0 bytes in 0 blocks
==4559==      possibly lost: 0 bytes in 0 blocks
==4559==    still reachable: 72,704 bytes in 1 blocks
==4559==         suppressed: 0 bytes in 0 blocks
==4559== Reachable blocks (those to which a pointer was found) are not shown.
==4559== To see them, rerun with: --leak-check=full --show-leak-kinds=all
==4559==
==4559== For counts of detected and suppressed errors, rerun with: -v
==4559== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

C++ classes: destructors

- Allocates `new int[sz]` in constructor, but never `delete[]`s it
- It's common for a constructor to obtain a resource (allocate memory, open a file, etc) that should be released when the object is destroyed
- *Destructor* is a function called by C++ when the object's lifetime ends, or is otherwise deallocated (i.e. with `delete`)
- It's common for a destructor to release the resource (deallocate memory, close a file, etc)

C++ classes: destructors

```
// sequence.h

1  #ifndef SEQUENCE_H
2  #define SEQUENCE_H
3  #include <cassert>
4  class Sequence { // What does this class do?  Anything wrong with it?
5  public:
6      Sequence() : array(NULL), size(0) { }
7      Sequence(int sz) : array(new int[sz]), size(sz) {
8          for(int i = 0; i < sz; i++) array[i] = i;
9      }
10     ~Sequence() { delete[] array; }
11     int at(int i) {
12         assert(i < size);
13         return array[i];
14     }
15 private:
16     int *array;
17     int size;
18 };
19 #endif // SEQUENCE_H
```

C++ classes: destructors

```cpp
// destructors3.cpp

#include <iostream>
#include "sequence.h"

int main() {
    Sequence seq(10);
    for(int i = 0; i < 10; i++) {
        std::cout << seq.at(i) << ' ';
    }
    std::cout << std::endl;
    return 0;
}
```

```
$ g++ -o destructors3 destructors3.cpp -std=c++11 -pedantic -Wall -Wextra

$ ./destructors3

0 1 2 3 4 5 6 7 8 9
```

C++ classes: destructors

```
$ valgrind --leak-check=full ./destructors3

==4571== Memcheck, a memory error detector
==4571== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==4571== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==4571== Command: ./destructors3
==4571==
==4571== error calling PR_SET_PTRACER, vgdb might block
==4571==
==4571== HEAP SUMMARY:
==4571==     in use at exit: 72,704 bytes in 1 blocks
==4571==   total heap usage: 3 allocs, 2 frees, 76,840 bytes allocated
==4571==
==4571== LEAK SUMMARY:
==4571==    definitely lost: 0 bytes in 0 blocks
==4571==    indirectly lost: 0 bytes in 0 blocks
==4571==      possibly lost: 0 bytes in 0 blocks
==4571==    still reachable: 72,704 bytes in 1 blocks
==4571==         suppressed: 0 bytes in 0 blocks
==4571== Reachable blocks (those to which a pointer was found) are not shown.
==4571== To see them, rerun with: --leak-check=full --show-leak-kinds=all
==4571==
==4571== For counts of detected and suppressed errors, rerun with: -v
==4571== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

C++ classes: destructors

Destructors are nearly always a better option than creating a special member function for releasing resources; e.g.:

```
#include <cassert>

class Sequence {
public:
    ...
    // User must call clean_up when finished with Sequence
    void clean_up() { delete[] array; }
    ...
};
```

C++ classes: destructors

User forgets to call `clean_up`:

```
{
    Sequence s(40);
    // ... (no call to s.clean_up())
} // s lifetime ends and memory is leaked
```

More subtly

```
{
    Sequence s(40);
    if (some_condition) {
        return 0; // memory leaked!
    }
    s.clean_up();
}
```

C++ classes: destructors

Destructor is **always automatically** called when object's lifetime ends or it is deallocated

You don't have to go hunting for all the places to put `object.clean_up()`

Quiz- answers

The destructor of an object is NOT necessarily called if ...

A an object's lifetime is over

B an object is deallocated

C there are no references or pointers to an object

D None of the above