601.220 Intermediate Programming

Destructors

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

```cpp
// new_eg1.cpp:
#include <iostream>

using std::cout;
using std::endl;

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

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

C++ dynamic memory allocation revisited

```
$ g++ -std=c++11 -pedantic -Wall -Wextra -c new_eg1.cpp
$ g++ -o new_eg1 new_eg1.o
$ ./new_eg1
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]`

C++ dynamic memory allocation revisited

If `T` is a `class`, 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

```cpp
// new_eg2.cpp:
#include <iostream>

using std::cout;
using std::endl;

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++) {
        cout << s_array[i].get_i() << " ";
    }
    cout << endl;
    delete[] s_array;
    return 0;
}
```

C++ dynamic memory allocation revisited

```
$ g++ -std=c++11 -pedantic -Wall -Wextra -c new_eg2.cpp
$ g++ -o new_eg2 new_eg2.o
$ ./new_eg2
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 initalize 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
- 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 name of the class 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:
#include <cassert>

// What does this class do?  Anything wrong with it?
class Sequence {
public:
    Sequence() : array(NULL), size(0) { }

    // Note: constructor can have both an initializer
    // list and statements in its body
    Sequence(int sz) : array(new int[sz]), size(sz) {
        for(int i = 0; i < sz; i++) {
            array[i] = i;
        }
    }

    int at(int i) {
        assert(i < size);
        return array[i];
    }

private:
    int *array;
    int size;
};
```

C++ classes: destructors

```
// sequence_main.cpp:
#include <iostream>
#include "sequence.h"

using std::cout;
using std::endl;

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

```
$ g++ -std=c++11 -pedantic -Wall -Wextra -c sequence_main.cpp -g
$ g++ -o sequence_main sequence_main.o
$ ./sequence_main
0 1 2 3 4 5 6 7 8 9
```

C++ classes: destructors

`valgrind` finds a leak!

```
==12306== HEAP SUMMARY:
==12306==     in use at exit: 40,185 bytes in 431 blocks
==12306==   total heap usage: 510 allocs, 79 frees, 46,345 bytes allocated
==12306==
==12306== 40 bytes in 1 blocks are definitely lost in loss record 29 of 83
==12306==    at 0x100009EAB: malloc (in 3.11.0/lib/valgrind/vgpreload_memcheck-amd64-darwin.so)
==12306==    by 0x10004E43D: operator new(unsigned long) (in /usr/lib/libc++.1.dylib)
==12306==    by 0x10000130A: Sequence::Sequence(int) (in ./sequence_main)
==12306==    by 0x10000111A: Sequence::Sequence(int) (in ./sequence_main)
==12306==    by 0x10000107C: main (in ./sequence_main)
==12306==
==12306== LEAK SUMMARY:
==12306==    definitely lost: 40 bytes in 1 blocks
==12306==    indirectly lost: 0 bytes in 0 blocks
==12306==      possibly lost: 0 bytes in 0 blocks
==12306==    still reachable: 4,096 bytes in 1 blocks
==12306==         suppressed: 36,049 bytes in 429 blocks
==12306== Reachable blocks (those to which a pointer was found) are not shown.
==12306== To see them, rerun with: --leak-check=full --show-leak-kinds=all
```

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:
#include <cassert>

class Sequence {
public:
    Sequence() : array(NULL), size(0) { }

    Sequence(int sz) : array(new int[sz]), size(sz) {
        for(int i = 0; i < sz; i++) {
            array[i] = i;
        }
    }

    // *** destructor ****
    ~Sequence() { delete[] array; }

    int at(int i) {
        assert(i < size);
        return array[i];
    }
private:
    int *array;
    int size;
};
```

C++ classes: destructors

```
$ g++ -std=c++11 -pedantic -Wall -Wextra -c sequence_main.cpp -g
$ g++ -o sequence_main sequence_main.o
$ ./sequence_main
0 1 2 3 4 5 6 7 8 9
```

C++ classes: destructors

```
==12568== HEAP SUMMARY:
==12568==     in use at exit: 40,121 bytes in 429 blocks
==12568==   total heap usage: 509 allocs, 80 frees, 46,321 bytes allocated
==12568==
==12568== LEAK SUMMARY:
==12568==    definitely lost: 0 bytes in 0 blocks
==12568==    indirectly lost: 0 bytes in 0 blocks
==12568==      possibly lost: 0 bytes in 0 blocks
==12568==    still reachable: 4,096 bytes in 1 blocks
==12568==         suppressed: 36,025 bytes in 428 blocks
==12568== Reachable blocks (those to which a pointer was found) are not shown.
==12568== To see them, rerun with: --leak-check=full --show-leak-kinds=all
```

C++ classes: destructors

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

```
// sequence.h:
#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!

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 to an object

D. None of the above