8.7 Memory leaks

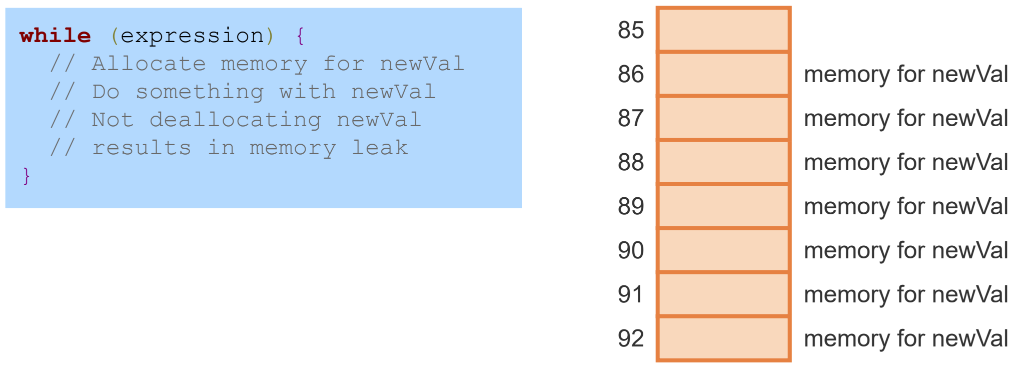
Memory leak

A ***memory leak*** occurs when a program that allocates memory loses the ability to access the allocated memory, typically due to failure to properly destroy/free dynamically allocated memory. A program's leaking memory becomes unusable, much like a water pipe might have water leaking out and becoming unusable. A memory leak may cause a program to occupy more and more memory as the program runs, which slows program runtime. Even worse, a memory leak can cause the program to fail if memory becomes completely full and the program is unable to allocate additional memory.

A common error is failing to free allocated memory that is no longer used, resulting in a memory leak. Many programs that are commonly left running for long periods, like web browsers, suffer from known memory leaks — a web search for "<your-favorite-browser> memory leak" will likely result in numerous hits.

PARTICIPATION ACTIVITY 8.7.1: Memory leak can use up all available memory.

1 2 2x speed



```
while (expression) {
  // Allocate memory for newVal
  // Do something with newVal
  // Not deallocating newVal
  // results in memory leak
}
```

Each loop iteration allocates more memory, eventually using up all available memory and causing the program to fail.

Feedback?

Garbage collection

Some programming languages, such as Java, use a mechanism called ***garbage collection*** *wherein a program's executable includes automatic behavior that at various intervals finds all unreachable allocated memory locations (e.g., by comparing all reachable memory with all previously-allocated memory), and automatically frees such unreachable memory. Some non-standard C++ implementations also include garbage collection. Garbage collection can reduce the impact of memory leaks at the expense of runtime overhead. Computer scientists debate whether new programmers should learn to explicitly free memory versus letting garbage collection do the work.*

PARTICIPATION ACTIVITY 8.7.2: Memory leaks.

Unusable memory	Memory locations that have been dynamically allocated but can no longer be used by a program. Unusable memory is created when a memory leak occurs.	**Correct**
Memory leak	Occurs when a program allocates memory but loses the ability to access the allocated memory. To prevent a memory leak, dynamically allocated memory must be freed before the memory becomes unusable.	**Correct**
Garbage collection	Automatic process of finding and freeing unreachable allocated memory locations. Garbage collection negatively affects a program's performance but can resolve a memory leak before the leak causes the program to fail.	**Correct**

Reset

Feedback?

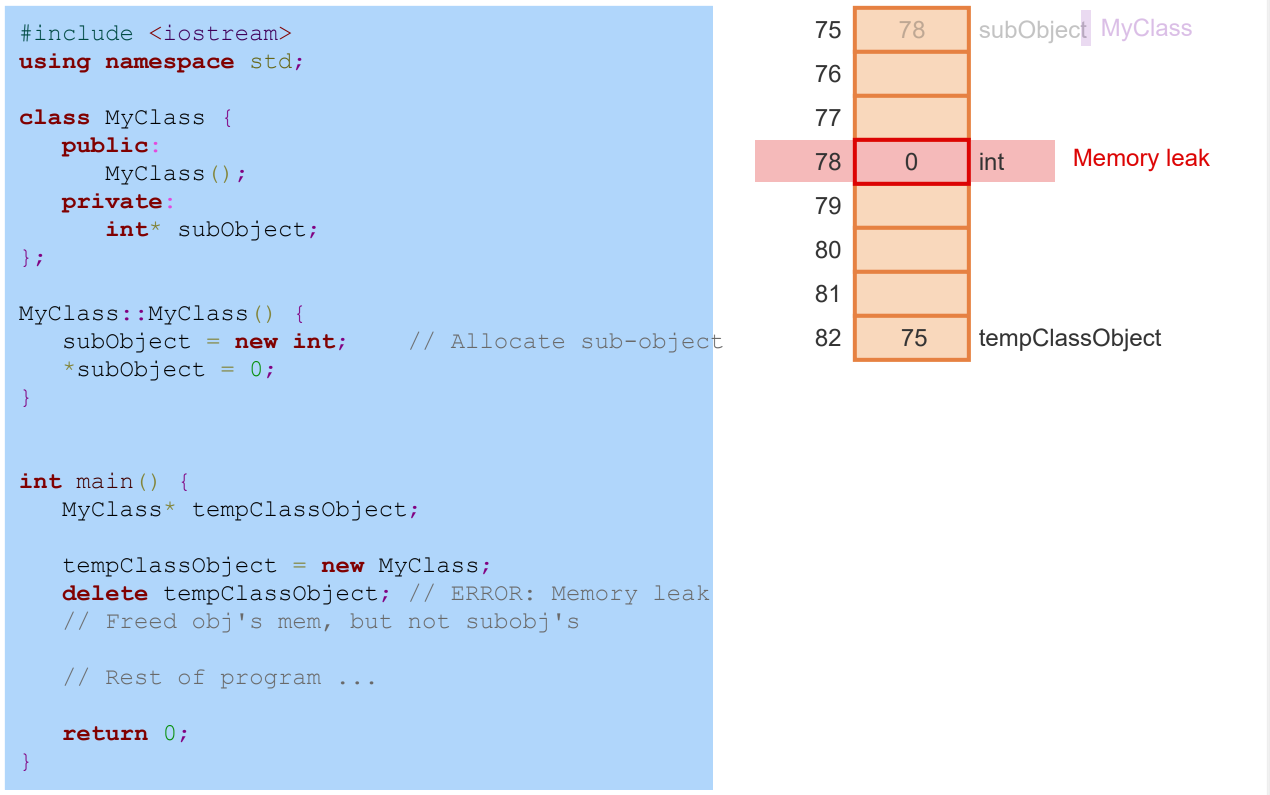
Memory not freed in a destructor

Destructors are needed when destroying an object involves more work than simply freeing the object's memory. Such a need commonly arises when an object's data member, referred to as a sub-object, has allocated additional memory. Freeing the object's memory without also freeing the sub-object's memory results in a problem where the sub-object's memory is still allocated, but inaccessible, and thus can't be used again by the program.

The program in the animation below is very simple to focus on how memory leaks occur with sub-objects. The class's sub-object is just an integer pointer but typically would be a pointer to a more complex type. Likewise, the object is created and then immediately destroyed, but typically something would have been done with the object.

PARTICIPATION ACTIVITY 8.7.3: Lack of destructor yields memory leak.

1 2 3 2x speed



```
#include <iostream>
using namespace std;

class MyClass {
   public:
      MyClass();
   private:
      int* subObject;
};

MyClass::MyClass() {
   subObject = new int;     // Allocate sub-object
   *subObject = 0;
}


int main() {
   MyClass* tempClassObject;

   tempClassObject = new MyClass;
   delete tempClassObject; // ERROR: Memory leak
   // Freed obj's mem, but not subobj's

   // Rest of program ...

   return 0;
}
```

Deleting tempClassObject frees the memory for the tempClassObject, but not subObject. A memo leak results because memory location 78 is still allocated, but nothing points to the memory alloca

Feedback?

PARTICIPATION ACTIVITY | 8.7.4: Memory not freed in a destructor. ✓

1) In the above animation, which object's memory is not freed?

- ○ `MyClass`
- ○ `tempClassObject`
- [✓ `subObject`]

✓ **Correct**

`subObject` is a sub-object of `tempClassObject` that has also been allocated memory. When `tempClassObject`'s memory is freed, `subObject`'s memory remains allocated, resulting in a memory leak.

2) Does a memory leak remain when the above program terminates?

- ○ Yes
- [✓ No]

✓ **Correct**

Memory leaks occur at the program level, so when a program terminates, all memory allocated by the program is freed.

3) What line must exist in `MyClass`'s destructor to free all memory allocated by a `MyClass` object?

- [✓ `delete subObject;`]
- ○ `delete tempClassObject;`
- ○ `delete MyClass;`

✓ **Correct**

When a `MyClass` object is created, the object allocates memory for `subObject`. Thus, when a `MyClass` object is deleted, the destructor must also free the memory allocated for `subObject` by deleting `subObject`.

Feedback?

PARTICIPATION ACTIVITY | 8.7.5: Which results in a memory leak? ✓

Which scenario results in a memory leak?

1)
```
int main() {
   MyClass* ptrOne = new
MyClass;
   MyClass* ptrTwo = new
MyClass;

   ptrOne = ptrTwo;
   return 0;
}
```

✓ **Correct**

ptrOne no longer points to the initial MyClass object, thus the object is inaccessible, causing a memory leak.

[✓ Memory leak]

No memory leak

2)
```
int main() {
   MyClass* ptrOne = new
MyClass;
   MyClass* ptrTwo = new
MyClass;
   MyClass* ptrThree;

   ptrThree = ptrOne;
   ptrOne = ptrTwo;
   return 0;
}
```

Memory leak

[✓ No memory leak]

Correct

Pointing ptrThree to ptrOne's object before re-pointing ptrOne keeps track of both MyClass objects. ptrOne and ptrThree point at the first MyClass object before ptrOne is set to point elsewhere.

3)
```
class MyClass {
   public:
      MyClass() {
         subObject = new
int;
         *subObject = 0;
      }

      ~MyClass() {
         delete
subObject;
      }

   private:
      int* subObject;
};

int main() {
   MyClass* ptrOne = new
MyClass;
   MyClass* ptrTwo = new
MyClass;
   ...

   delete ptrOne;
   ptrOne = ptrTwo;
   return 0;
}
```

Memory leak

[✓ No memory leak]

Correct

`delete ptrOne` causes the MyClass destructor to delete the MyClass sub-object, so memory for the object and sub-object are both freed before ptrOne is reassigned.

Feedback?