601.220 Intermediate Programming

Virtual destructors and iterators

Virtual destructors

```
// virt_dtor.h:
class Base {
public:
    Base() : base_memory(new char[1000]) { }

    ~Base() { delete[] base_memory; }

private:
    char *base_memory;
};

class Derived : public Base {
public:
    Derived() : Base(), derived_memory(new char[1000]) { }

    ~Derived() { delete[] derived_memory; }

private:
    char *derived_memory;
};
```

Virtual destructors

```
// virt_dtor.cpp:
#include "virt_dtor.h"

int main() {
    // Note use of base-class pointer
    Base *obj = new Derived();
    delete obj; // calls what destructor(s)?
    return 0;
}
```

`new Derived()` calls `Derived` default constructor, which in turn calls `Base` default constructor; that's good

Which destructor is called?

- Destructor is not `virtual`
- Does that mean `~Base` is called but not `~Derived`?

Virtual destructors

```
$ g++ -o virt_dtor virt_dtor.cpp
$ valgrind ./virt_dtor
==3961== Memcheck, a memory error detector
==3961== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==3961== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==3961== Command: ./virt_dtor
==3961==
==3961==
==3961== HEAP SUMMARY:
==3961==     in use at exit: 1,000 bytes in 1 blocks
==3961==   total heap usage: 4 allocs, 3 frees, 74,720 bytes allocated
==3961==
==3961== LEAK SUMMARY:
==3961==    definitely lost: 1,000 bytes in 1 blocks
==3961==    indirectly lost: 0 bytes in 0 blocks
==3961==      possibly lost: 0 bytes in 0 blocks
==3961==    still reachable: 0 bytes in 0 blocks
==3961==         suppressed: 0 bytes in 0 blocks
==3961== Rerun with --leak-check=full to see details of leaked memory
==3961==
==3961== For counts of detected and suppressed errors, rerun with: -v
==3961== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

`~Derived` is *not* called; `derived_memory` is leaked

Virtual destructors

```
// virt_dtor2.h:
class Base {
public:
    Base() : base_memory(new char[1000]) { }

    // Now *** virtual ***
    virtual ~Base() { delete[] base_memory; }

private:
    char *base_memory;
};

class Derived : public Base {
public:
    Derived() : Base(), derived_memory(new char[1000]) { }

    // Now *** virtual ***
    virtual ~Derived() { delete[] derived_memory; }

private:
    char *derived_memory;
};
```

Virtual destructors

```
// virt_dtor2.cpp:
#include "virt_dtor2.h"

int main() {
    // Note use of base-class pointer
    Base *obj = new Derived();
    delete obj; // calls what destructor(s)?
    return 0;
}
```

Virtual destructors

```
$ g++ -o virt_dtor2 virt_dtor2.cpp
$ valgrind ./virt_dtor2
==3971== Memcheck, a memory error detector
==3971== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==3971== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==3971== Command: ./virt_dtor2
==3971==
==3971==
==3971== HEAP SUMMARY:
==3971==     in use at exit: 0 bytes in 0 blocks
==3971==   total heap usage: 4 allocs, 4 frees, 74,728 bytes allocated
==3971==
==3971== All heap blocks were freed -- no leaks are possible
==3971==
==3971== For counts of detected and suppressed errors, rerun with: -v
==3971== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Fixed; thanks to dynamic binding, `delete obj` calls `~Derived`, which in turn calls `~Base`

Derived-class destructor always implicitly calls base-class destructor at the end

Virtual destructors

To avoid this in general: *Any `class` with virtual member functions* should also have a `virtual` destructor, even if the destructor does nothing

Quiz!

Assume class C is derived from classes A and B and class D is derived from B. At the very least, the destructors of which classes must be `virtual`?

A. C and D

B. A and B

C. A, B and C

D. A, B, C and D

E. D only