ECE150KL

L41PA
26 April 2013

ICES

destructors

new & delete

overloading & references

(certainly into Monday)

ICES Q's

#1 & #2 — Is having ability to work on Android worthwhile? What if you could use multi-touch camera & accelerometer in some programs (and were thus forced to test on Nexus — gdb exists over USB)?

→ just use both boxes — no need to treat as separate questions

---

mock exam 4

- will put out solutions
- people who turned in mostly did well, with general issues on...
  - #4 was on W's material — will review today
  - #1A.. I'll review — seems my lecture wasn't clear enough

L39 P8
~~L40 P4~~
L41 P1

destructors

A destructor is a subroutine called to destroy a variable of a class type (an object)

Destructor is always* called for objects:

static variable: after main

automatic variable: just ~~before~~ RET instructio

dynamic variable: just before free [delete in

review

get to here Wednesday

*- possibly skipped if program crashes

[- supposed to execute if exception thrown from child to parent ... may depend on use of one compiler for all code]

(mentioned Wed)

Start

A Few Important Details...

Define one ~~de~~structor, and make it virtual

```
virtual ~ALPHA();   // no args to destructor
```

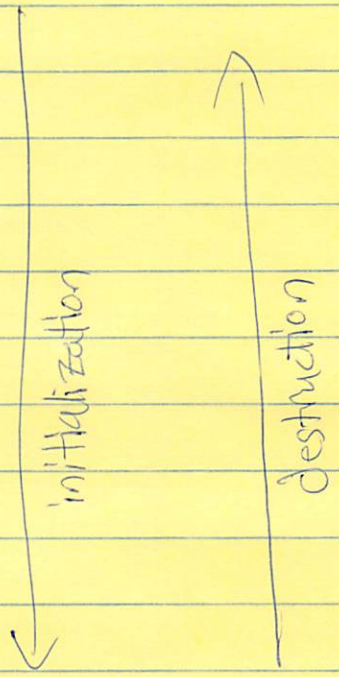
Why virtual? Avoid calling only base class' destructor (for dynamic variables)

Order of execution

- body of destructor
- destructors for fields with class type (bottom to top of order in class definition)
- base class destructor (if any)

NOTE: A pointer is not an object. If you have a pointer to a private, dynamically allocated object, your destructor must explicitly delete that object.

```
Class ALPHA : public BETA {
    int x;
    GAMMA y;
    double z;
    .
    .
    .
};
```



You cannot override these orders.

You can list initializers out of order, but they will simply be put in correct order (as above) by compiler — that was the bug in Mock exam #4.

If you leave out an initializer for an object (for y in class above), ~~a default (no args) ctor~~ Constructor (with no arguments) will be called in order above (after x before z).

Note: Do not call y's destructor in ALPHA's — will be called after destructor body.

~~L4OPS~~
L4 P3

allocating memory

malloc & free do not call constructors/destructors

⇒ do not use for class instances
(in new C++ code, best just not to use them)

create a new object:

```
Class* ptr = new Class (arg1, arg2, ...);
```

(arg1, arg2, ...) — passed to Class' constructor (omitted ⇒ no arguments)

returns pointer to constructed Class instance (Class*)

On failure, throws an exception. We probably won't get to exceptions in our class, but by default kills your program.

If you want call to return NULL instead (no constructor called in that case),

```
#include <new>
Class* ptr = new (std::nothrow) Class (arg1, arg2, ...);
```

~~L4OPC~~
L4IP4

If you want to allocate an array, use---

```
Class *ptr = new Class[42];
```

↑ array of 42 objects of type Class, each constructed using constructor with no args (which must be defined)

add "(std::nothrow)" & #include <new> at top of file if you want new to return NULL on failure

Deleting objects

You (the programmer) must remember whether ~~an~~ a pointer points to an object or an array.

Call the right form of delete!
(Bug likely to be hard to find if you do not.)

```
Class* ptr;
delete ptr;   // object — no args allowed to destructor
delete[] ptr; // array — also no args allowed
```

It's ok to delete NULL (has no effect) in C++

Lecture Topics

- overloading & references
  - purpose, pros, & cons
  - matching an overloaded call
  - pitfalls of overloading & conversions
  - copying vs. constructing
- variable declarations

[trimmed down version of Lectures 6, 7, and 8 from ECE498SL/ECE409]

Overloading

- one function name, multiple definitions
- extends to operators (not just redefining, but providing multiple definitions)

- original goal of overloading
  - support “natural” use of operators for user-defined types
  - canonical example: complex numbers

- let P and Q be complex numbers and calculate R = P^2 + Q^2
  - in C, taking some small liberties with the stack…

```
R = complex_add (complex_multiply (P, P),
                 complex_multiply (Q, Q));
```

  - in C++,

```
R = P * P + Q * Q;
```

- also expect to fit in with “natural” conversions from integer, double, etc.
  - complex * int
  - complex * double
  - int * complex
  - double * complex
  - etc.

- define all such functions? absurd…

- instead
  - create new implicit casts
  - use friend functions for symmetry

- example

```
class complex {
    complex (int real_part);
    complex (double real_part);
    friend complex operator+ (const complex& a,
                              const complex& b);
    friend complex operator* (const complex& a,
                              const complex& b);
};
```

- a few things to notice
  - single-argument constructor creates implicit cast path
    - complex P = 4; // P = 4 + 0i -- why not 0 + 4i?
    - to prevent implicit casts, use keyword “explicit”: `explicit complex (int real_part);`
    - in case above, compiler will still use int to double to complex implicit path without warnings…
  - trailing ampersand indicates a reference type
    - implementation equivalent to pointer
    - syntactic use and rules slightly different (discussed later)
  - return type is the whole structure!
    - can’t use reference (pointer) to local variable inside function
    - copy is returned on the stack
  - all of these functions can be inlined
    - including friend functions
    - but some copying hard to optimize away
  - arguments to operators are constants (casts do not work otherwise)

References

- want syntactically equivalent yet efficient forms for user-defined types
  - syntax
    - recall: `R = P * P + Q * Q;`
    - not: `R = *(&P * &P + &Q * &Q);`
  - but class instance may be quite large
    - avoid copying to stack all the time
    - avoid returning on stack if possible

- reference
  - pointer implementation (identical!)
  - syntactically equivalent to base type
  - possible ambiguity with reference-to-reference assignment
    - copy pointer or copy contents?
    - to avoid, C++ disallows changes to reference value (that is, to a new pointer) after initialization
    - assignment thus results in copy of contents

- references allow
  - redefine argument semantics on a per-argument basis.
  - can use “pass by reference” instead of “pass by value!”

- my take
  - as Stroustrup says elsewhere, any language can be misused
  - bad idea to rewrite language in a way that obscures intent

- simple example:
  - Where is “a” initialized?
  - Yes, the compiler can tell and warn you.

```
int a;
foo (a);
```

- This loop seems to hang. Can you help?

```
while (42 != i) {
    foo (i);
    x = bar (i);
    zap (i, x);
}
```

- worst part in my view
  - can change argument style (for example, **`int`** to **`int&`**)
  - with no compiler warnings (not a problem when variable is passed)

- solutions? either
  - pick function names that make it obvious which arguments might change value (???)
  - or just mark arguments in the code instead

- What about arguments that don’t change?
  - most of your data is class instances
  - don’t want copied onto the stack
  - but a little clunky to write “&” everywhere

- but use **`const`** with reference arguments!
  - copying struct to stack can be useful, but leave decision to callee
  - **`const`** came from C++ (wasn’t in early C)

Matching

- How different do two definitions of a function really have to be for the compiler to distinguish them?

- How does the compiler decide which function you meant to call?

- C++ allows for extremely minor distinctions; use at your own risk.

- For example, C's default type conversions are not assumed:
  - char/short to int
  - float to double
  - thus the following operators are different

```
operator+= (int i);
operator+= (char c);
```

- also allows overloaded variants based on other implicit conversions
  - signed to unsigned
  - non-const to const

- selecting between overloaded matches
  - the basics: pick the "most derived" class
  - not always unique: compiler reports ambiguity as error

- more exact rules? (see 498SL/409 notes for a longer discussion)
  - they exist, but they don't seem to be widely known, understood, portable, and so forth
  - so, as with operator precedence, you should simply avoid making a set of overloaded calls that require you to know more: use different names instead

- not all operators can be overloaded
  - member access (“.”)
  - pointer to member function invocation (“.*”)
  - conditional expressions (?:)
  - scope identification (::)

- overloading can break C’s duality
  - pointer-like objects and array-like objects not necessarily equal
  - pointer vs. array
    - `array[10]`
    - `*(array+10)`
  - pointer dereference
    - `inst->member`
    - `(*inst).member`
    - `inst[0].member`
    - not possible to change definitions equivalently because “.” can’t be overloaded

- copying vs. constructing
  - What’s the difference between the two assignments below?

```
ALPHA a;
ALPHA b = a; // copy constructor
b = a;       // assignment
```

  - declaration has no “old version”
    - may need work to destroy previous version
    - e.g., rehash instance in a lookup table
  - these two are **NOT** equivalent in C++
    - default version is memberwise copy for both
    - overriding one does **NOT** catch the other (other version will use default copy)
    - compiler will **NOT** warn you

- copying vs. constructing
  - What's the difference between the two assignments below?

```
ALPHA a;
ALPHA b = a; // copy constructor
b = a;       // assignment operator
```

  - declaration has no "old version"
    - may need work to destroy previous version
    - e.g., rehash instance in a lookup table
  - these two are **NOT** equivalent in C++
    - default version is memberwise copy for both
    - overriding one does **NOT** catch the other (other version will use default copy)
    - compiler will **NOT** warn you

[just to be clear: in second line (copy constructor), ONLY the copy constructor is called; no default (no arg) constructor is called first]

Variable Declarations and Single-Assignment

- objectives
  - clean up minor scoping issues
    - e.g., loop variables, test conditions
    - scope inside loop / inside then/else code blocks
  - provide single-assignment style choice that is easier to optimize
    - avoid "empty" constructor for uninitialized objects
    - avoid need to optimize away reads of dead initialization

- extension (to C)
  - support more flexible variable declarations
  - interleaved with statements
  - note: doesn't really solve the problem
    - `if (…) {var = …;} else {var = …;}`
    - still need outside (dead) constructor
  - can obscure type information (buried somewhere in code)

- also encourage use of op-assign operators (+=, -=, *=, etc.)
  - optimizer in theory may be able to transform
  - in practice, has to worry about aliasing
    - A = A + B;
    - Can I update A safely before calculating sum?
    - not clear
      - if compiler has both functions, can analyze interaction
      - can't insert call to op_and_assign (A, B, A);
    - in contrast
      - A += B
      - implementation obviously expects aliasing