L34A

ECE 198KL

22 April 2013

C++ classes

review: member & class functions

inheritance

constructors & destructors

access control

overloading & references

PA2 due Tuesday (as I mentioned Friday)

```
class Derived : public Base {
    // int field;
    // static OtherClass classVariable;
    // interface & implementation member & class functions
    // initialization & teardown for an instance
};
```

(review)

Member functions / methods

- looks like a C declaration: `void method(void);`
- has an implicit pointer to class instance "this" as first argument
- invocation

```
ALPHA a;
ALPHA * ptr;

a.method();      // this is &a
ptr->method();   // this is ptr
```

ALPHA::method implied by type of a & ptr

- definition: in a source file; context must be specified...

```
void ALPHA::method(void) {
    //---
}
```

In the function, symbols from class ALPHA can be used without prefix ALPHA::
Fields & methods without prefix use "this" implicitly.

```
field++;   recursiveMethod();
```

more detail than last time

What if I don't want an implicit pointer to a class instance?

Add 'static' to front of function declaration in class definition:

```
static void classfunc (int arg);
```

Such functions are also called class functions.

Class functions are invoked just like functions not defined in a class (C functions), but remember that the class name is added to their name.

Class name may not be inferred (often can't be for these functions), so ~~you must be explicit~~:

```
TheClass::classfunc (42);
```

↑ new namespace construction/join operator

Definitions (in source file) look the same as methods, but have no implicit "this" argument...

no "static" here →

```
void TheClass::classfunc (int arg)
{
    field++; // not allowed — no "this" so no implicit fields
    method(); // not allowed
    classfunc(arg-1); //ok: "TheClass::" inferred
}
```

access control

In C, information hiding gets entangled with file management

Why? scope = visibility, but also access rights

If compiler recognizes a name, my code can use that name (call a function, access a structure's field, write a variable ...)

Alternative is file scope — can't use name outside of a file.

As a result, whole modules sometimes jammed into one file.

In C++, scope = visibility only

- Does not imply access rights
- Protects against accidents, not fraud/malice
- Access rights granted by class to class/function/field (not objects)

Access rights specified in class definition

- Nowhere else to look — unless listed there, access not allowed
- Not possible to claim access rights in another piece of code.

⇒ Code for class can be organized into source files (more than one) as desired

Types of access rights in C++

default → "private:"
- access allowed only within the class (class' functions)
- used for fields & implementation functions ~~static~~ class variables,

"protected:"
- access allowed within the class and any derived classes
- use similar to private

"public:"
- access allowed to any code
- used for interface functions, and for instance initialization/teardown

---

Access is specified for all subsequent declarations (but can be changed by another specifier):

```
class ALPHA {
    private:
        ]  all private
    public:
        ]  all public
    private:
        ]  all private
};
```

Constructors

A constructor is a subroutine called to initialize a new variable of class type (an instance of the class — often called an object).

Constructor is always called for class instances:

static variable — before main

automatic variable — start of block of code / definition point in code

dynamic variable — just after allocation

A Few Important Details...

- "Name" of constructor is name of class
- No return type
- Can have >1 if compiler can tell which one you want to use (more later, but basically args must match)

- Two created by default
  - no args: used for array element initialization; if you define any other constructor, this variant is not created, and arrays can only be declared if you define a constructor with no args
  - copy constructor: takes one argument: another instance of class

```
ALPHA a;
ALPHA b=a;   <- used here
```

more detail later

```
class ALPHA {
    private:
        int one;
        int two;
    public:
        ALPHA (int arg, int second);
};
```

In source file, the definition:

```
ALPHA::ALPHA (int arg, int second) : one(arg), two(second)
{
    // code
}
```

base class constructor would go here

optional list of initializers: field (expression)

constructor order of execution

- base class constructor (if only) can be chosen as initializer
- initializers in order of field declaration

(how you list them in constructor does not affect code — put them in same order)

NOTE: constructors called for all class type fields, even if you give no initializer.

- body of constructor

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 instruction
dynamic variable = just before free [delete in C++]

*- 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]

A Few Important Details...

Define one destructor, 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
— 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.