

`inline`, and `enums` in classes

- `inline` – an alternative to macros
 - Explicit – use `inline` keyword
 - Necessary if outside class definition
 - Implicit – any complete function in class definition
 - Both types should be placed in header files
 - But note: just a request – compiler may ignore
- Encapsulated `enums` should be public
 - Just a type – no risk to implementation
 - Means no need for multiple get and set functions – just need two: `setFoo(type t)` and `type getFoo()`
 - See `clock` example in `.../demo06/`

Constructors - ctors

- A ctor is invoked *every* time an instance is created (whenever a class is instantiated)
 - Includes objects on the stack and the free store
 - Also includes temporary objects (like result of expression), and copies passed to/from functions
 - But not invoked by creation of pointer or reference
- Compiler supplies default ctor if no ctor written
 - Compiler-supplied version can take no arguments
 - Also default ctor if default values for all parameters –
`Square(int = 1);`
 - Can invoke with 0-1 arguments – `Square a, b(5);`

Destructors - dtors

- A dtor is invoked whenever an object goes out of scope, or by `delete` for objects on free store
 - Compiler supplies one if you don't
 - But it doesn't do any work – so write one if need to release free store space or other resources
- Defined like a ctor, but with a `~` in front, and it may not take any arguments
 - `~Foo();` // syntax in header file
 - `Foo::~~Foo() { ... }` // syntax in implementation file
- Can be invoked directly (unlike ctors)
 - `myFooPtr->~Foo();` // must do if use new placement syntax

Manager functions

- 4 important functions every class should have – “the orthodox canonical class form”
 - A **default ctor**, a **copy ctor** (*below*), a **dtor**, and an **assignment operator** (*next week*)
 - Compiler supplies all 4 – but usually should write own
- Copy ctor – `Foo(Foo const &other);`
 - Compiler-supplied version makes a “shallow copy”
 - Invoked when initializing with object as argument:
`Foo foo(aFoo);` // or C-style: `Foo foo = aFoo;`
 - Also when passing an object to/from a function by value
 - See `String` example in `.../demo07`

Wednesday, November 11



Implicit type conversion

- **Converting ctors** – for conversion of primitive types to user-defined types
 - Any ctor that takes exactly one argument
 - Invoked whenever an argument of that type is passed to a function that expects an object
 - See Assignment 4, part 4: testcomplex.cpp
- **Operator conversion functions** – for conversion of user-defined to primitive (usually)
 - e.g., `operator int() const; // in class Foo`
 - Means a `Foo` instance should be implicitly converted to `int` in certain circumstances, like `cout << myFoo;`
 - But can be risky – better to overload `<<` instead

Base member initialization

- Only applies to ctors – to initialize data

```
Foo::Foo(int v) : value(v) { }
```

- No function body in this case, but still need { }
- Separate the items in the list (after :) by commas
- Note: initialization order same as declaration order, not order in list (usually match though)
- *Must* use for constants and reference variables
- *Should* use for user-defined types if default ctor not appropriate (otherwise extra ctor *and* dtor calls)
- Always a better style than assignment – “PITA”

Dealing with arrays of objects

- Oops – can't initialize array elements *in* a class
 - Only choice: default initialize; then assign in body

- Fortunately, can initialize if outside a class

```
Foo a[] = { Foo(5), Foo(), Foo(-2) }
```

- Default constructor is only called for `a[1]`

- Initialization restriction also applies to free store
 - Decent “trick” is to use an array of pointers instead

```
Foo **ap = new Foo *[3]; // no ctors called yet
```

```
ap[0] = new Foo(5); // construct each element
```

- Remember: delete each element when done; then delete array

this

- A *pointer* to the **invoking instance**
 - i.e., it has the same *meaning* as Java's `this`
 - But used differently: `this->x`, instead of `this.x`
 - It is passed as a *hidden* first parameter to non-static member functions
- So `*this` is the instance itself
 - Use `*this` for making copies

```
Foo myClone(*this);
```
 - Also `return *this` to return a self-reference or copy
 - Returning a self-reference allows for “function chaining”
 - See `String::refresh()` example in `.../demo07`