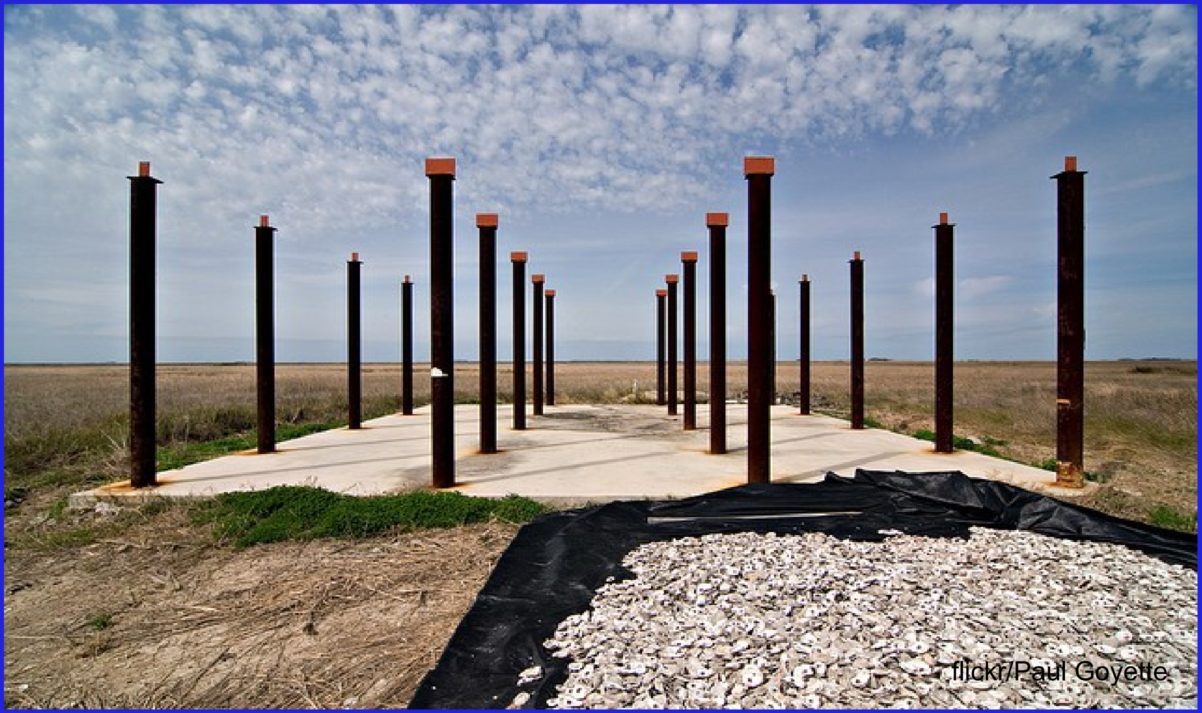
C++ Foundation



Objects, Classes, and Lifetimes

Objects, Classes, and Lifetimes

- constructors
- destructors
- the three storage classes
- new expressions
- delete expressions
- copy construction
- copy assignment
- smart pointers
- copying rule of three

Construction

- A constructor has the same name as the class

```
#include "date.hpp"

void example()
{
    date xmas(2011, 12, 25);
    assert(xmas.day() == 25);
}
```

date.hpp

```
class date
{
public:
    date(int year, int month, int day);
    ...
};
```

Construction

- Define constructors using the
  : member(*initialization*), syntax

date.hpp

```
class date
{
public:
    date(int year, int month, int day);
    ...
private:
    long time_stamp;
};
```

date.cpp

```
#include "date.hpp"

date::date(int year, int month, int day)
  : time_stamp(calc(year,month,day))
{

}
```

you *cannot* do initialization here

No Defaults

- Primitive data members *<u>don't</u>* acquire a default value - just like they don't in C

```
date::date(int year, int month, int day)
  // : time_stamp(...)
{
}
```

initialization commented out

```
void no_defaults()
{
    date xmas(2011, 12, 25);
    std::cout << xmas.time_stamp;
}
```

```
-740281080
```

assumes time_stamp is publicly accessible

Default Construction

- Value type objects can often be created without any arguments
  - If all arguments have defaults the effect is the same

```
class rope
{
public:
    rope();
    ...
};
```

```
#include “rope.hpp”

void example()
{
  rope old;
  ...
}
```

```
class rope
{
public:
    rope(const char * = “”);
    ...
};
```

Very Common Mistake

```
void correct()
{
  rope old;
  ...
}
```

this defines an object called old, of type rope

```
void incorrect()
{
  rope old();
  ...
  old.member
}
```

this declares a *function* called old, that accepts no arguments and returns a rope object

Destruction

- When an object's life ends it's destructor is *automagically* called
- The ~ “complement” of a constructor

```
#include “date.hpp”

void example()
{
    date xmas(2011, 12, 25);
    ...
} // xmas.~date()
```

```
class date
{
public:
    ~date();
    ...
};
```

the destructor never has any parameters - *you* “never” call it...

3 Types of Storage

- static - eg global variables
- automatic - eg local variables
- dynamic - eg new/delete

```
date a(2011,3,7);
static date b(2011,3,29);

date example(date c)
{
    static date d(2011,3,7);
    date e(2011,3,7);
    date * f = new date(2011,3,7);
    ...
    delete f;
    return date(2011,12,25);
}
```

a	
b	
c	
d	
e	
f	
*f	
return	

fill in the 8 storage classes

3 Types of Storage

- static - eg global variables
- automatic - eg local variables
- dynamic - eg new/delete

```
date a(2011,3,7);
static date b(2011,3,29);

date example(date c)
{
    static date d(2011,3,7);
    date e(2011,3,7);
    date * f = new date(2011,3,7);
    ...
    delete f;
    return date(2011,12,25);
}
```

a	static
b	static
c	automatic
d	static
e	automatic
f	automatic
*f	dynamic
return	automatic

fill in the 8 storage classes

Static Storage

- The order of initialization within a translation unit is defined: top to bottom, left to right
- The order of initialization across translation units is *undefined*

```
date global(2011,3,7);

void function()
{
    ...
}

// global.~date()
```

global's constructor is usually called *before* function invocation

global's destructor may be called at program termination

Automatic Storage

- A parameter object or local object is automatically destructed when it goes out of scope

```
void eg(date param)
{
    for (...)
    {
        date local;
        ...
        ...
        // local.~date()
    }
    // param.~date()
}
```

Dynamic Storage

- Create a dynamic object with a *new* expression
  - Allocates memory dynamically, like malloc()
  - Constructs an object in that memory
  - Returns a strongly typed pointer to the object

new keyword

```
void example()
{
    date * xmas = new date(2011, 12, 25);
}
```

object construction

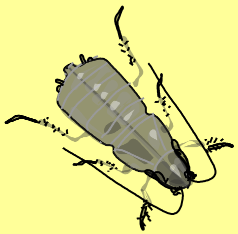
Dynamic Storage

- When a pointer goes out of scope *nothing* happens - the object pointed to does ***not*** have its destructor automatically called
- Destroy a new'd object with a delete expression
  - Calls the destructor on the pointed to object
  - Releases the dynamic memory - like free()

```
void example()
{
    date * xmas = new date(2011, 12, 25);
    ...
    delete xmas;                // xmas->~date();
                                // free(xmas);
}
```

Puzzle

- Spot the bug
- What happens in this case?



```
#include “B.hpp”

class fubar
{
public:
    fubar(int size) : elems(new B[size]) {}
    ~fubar() { delete elems; }
    ...
private:
    B * elems;
};
```

Solution

- delete[ ] calls all the destructors (in reverse) and then releases the memory

```
#include “B.hpp”

class fubar
{
public:
    fubar(int size) : elems(new B[size]) {}
    ~fubar() { delete[] elems; }
    ...
private:
    B * elems;
};
```

Consider using <vector> instead

Copying

- Copying an object...
  - As it is created is called *copy construction*
  - After it is created is called *copy assignment*

```
void example()
{
    date xmas(2011,12,25);

    date copy(xmas);

    copy = xmas;

    date another = xmas;
}
```

copy construction

copy assignment

copy construction

Copying

```
class date
{
public:
    ...
    date(const date & other);
    date & operator=(const date & rhs);
    ...
};
```

```
date::date(const date & other)
    ...
{
}

date & date::operator=(const date & rhs)
{   ...
}
```

Copy Construction

- The order of initialization is defined by the order of declaration

```
date::date(const date & other)
    : year(other.year)
    , month(other.month)
    , day(other.day)
{
}
```

```
class date
{
    ...
private:
    int year;
    int month;
    int day;
};
```

Why?

Copy Construction

- If you don't write a copy constructor the compiler will try and write one for you
  - It will do a member-by-member copy construction

```
/*
date::date(const date & other)
     : year(other.year)
     , month(other.month)
     , day(other.day)
{
}
*/
```

Copy Assignment

- If you don't write a copy assignment operator the compiler will attempt to write one for you
  - It will do a member-by-member copy assignment

```
/*
date & date::date(const date & rhs)
{
    year = rhs.year;
    month = rhs.month;
    day = rhs.day;
    return *this;
}
*/
```

Smart Pointers

- The dereference and arrow operators can be overloaded; an object can be designed to look like, feel like and smell like a plain pointer - but act smarter

```
class wibble_ptr
{
public:
    ...
    wibble & operator*() const;
    wibble * operator->() const;
    ...
private:
   wibble * raw;
};
```

Smart Pointers

- For example, consider a null-dereference checking smart pointer

```
void raw(wibble * ptr)
{
    ptr->use();
}
```

what if ptr is null...

```
void smart(wibble_ptr ptr)
{
    ptr->use();
}
```

compiler rewrites...

```
void smart(wibble_ptr ptr)
{
    ptr.operator->()->use();
}
```

Smart Pointers

```
wibble * wibble_ptr::operator->() const
{
    if (!raw)
        ...

    return raw;
}
```

throw an exception
(covered later)

Pointer Parameters

- Good design is often associated with fewer bald pointers and more smart pointers
- Aim to make ownership and lifetime an explicit part of design
- Assume reference parameter objects do not live beyond the end of the function call - don't store their address

Copying - 3 Options

- 1. Let the compiler write them and add a comment to that effect

```
class date
{
public:
    date(int year, int month, int day);
    // compiler generated copy c'tor ok
    // compiler generated copy assignment ok
    ~date();
    ...
private:
    long yyyymmdd;
};
```

Copying - 3 Options

- 2. Write your own because the compiler generated ones would be wrong

```
template<typename Type>
class shared_ptr
{
public:
    shared_ptr(Type * ptr);
    shared_ptr(const shared_ptr &);
    shared_ptr & operator=(const shared_ptr &);
   ~shared_ptr();
    ...
private:
    Type * raw;
    unsigned int * count;
};
```

Copying - 3 Options

- 3. Turn copying off by declaring them private
  - Since they are never used they don't need to be defined

```
template<typename Type>
class scoped_ptr
{
public:
    scoped_ptr(Type * ptr);
   ~scoped_ptr();
    ...
private: // inappropriate
    scoped_ptr(const scoped_ptr &);
    scoped_ptr & operator=(const scoped_ptr &);
private:
    Type * raw;
};
```