CIS 330
C++ and Unix

Lecture 15

Name Control

Copy Constructors Summary

- A constructor for creating a new object when you make copy of an existing object
- Enables correct copying of objects
- Compiler creates one for you if you don't define your own
- You can also "enforce" users to always pass objects by reference by "disabling" pass by value (which results in a copy) by creating a **private** definition of a copy constructor (you don't even have to implement it)

Pointers to Members

- A pointer is a variable that holds the address of some location
- It can point to either data or function, and it can be changed to point at different things at runtime
- In C++, pointer-to-member follows this same concept, but what it selects is a **location inside a class**
  - But there is no "address" inside a class
  - Selecting a member means find an **offset** into that class
  - You can't produce an actual address until the offset is combined with the starting address of of an object
- These cannot be incremented or compared (like regular pointers)

Questions

Namespace

- Global functions are in a single global namespace
- The `static` keyword can give you some control over this (allowing you to give same variable/function names in different files)
- However, this can still cause problems in a large project with multiple files (and multiple coders)
- Creating a namespace can give you better control over names

Properties of a namespace

- Namespace definition can only appear at global scope, or nested within another namespace
- No terminating semicolon is necessary (you can still put it there, if you want)
- You can alias namespaces (e.g., if the original name is too long)
- You can also use "friendship" – any `friend` declaration in a given namespace means that it is also part of the namespace

Example

Notice how members inside a namespace is referenced - similar to how class members are referenced

```
1.  namespace X {
2.    class Y {
3.      static int i;
4.    public:
5.      void f();
6.    };
7.    class Z;
8.    void func();
9.  }
```

```
10.  int X::Y::i = 9;
11.  class X::Z {
12.    int u, v, w;
13.  public:
14.    Z(int i);
15.    int g();
16.  };
17.  X::Z::Z(int i) { u = v = w = i; }
18.  int X::Z::g() {
19.    return u = v = w = 0;
20.  }
21.  void X::func() {
22.    X::Z a(1);
23.    a.g();
24.  }
```

Is this necessary?

Using a namepsace

- Use the directive “using”
- using namespace std;

Example

```
namespace Int {
  enum sign { positive, negative };
  class Integer {
    unsigned int i;
    sign s;
  public:
    Integer(int ii = 0) : i(ii), s(i >= 0 ? positive : negative) {}
    sign getSign() const { return s; }
    void setSign(sign sgn) { s = sgn; }
  };
}
namespace Math {
  using namespace Int;
  Integer a, b;
  Integer divide(Integer, Integer);
}
```

Example

```
void arithmetic() {
  using namespace Int;
  Integer x;
  x.setSign(positive);
}
```

Example

```
namespace supercalifragilisticexpialidocious {

}

namespace MaryPoppins = supercalifragilisticexpialidocious;
```

Example

```
namespace Int {
  enum sign { positive, negative };
  class Integer {
    unsigned int i;
    sign s;
  public:
    Integer(int ii = 0) : i(ii), s(i >= 0 ? positive : negative) {}
    sign getSign() const { return s; }
    void setSign(sign sgn) { s = sgn; }
  };
}
namespace Math {
  using namespace Int;
  Integer a, b;
  Integer divide(Integer, Integer);
}
```

```
using namespace Math;
Integer a;
a.setSign(negative);
Math::a.setSign(positive);
```

Why is this necessary?

namespace Override

- What happens if you use namespace to introduce a new name, but you "override" that name by declaring another name in the same scope

Example

Will this run?
If so, what would happen?

```
namespace Math {
    using namespace Int;
    Integer a, b;
    Integer divide(Integer, Integer);
}
namespace Calculation {
    using namespace Int;
    Integer divide(Integer, Integer);
}
```

```
void testFive()
{
        using namespace Math;
        using namespace Calculation;
        Integer a(1);
        Integer b(2);
        Integer c = divide(a, b);
}
```

Error

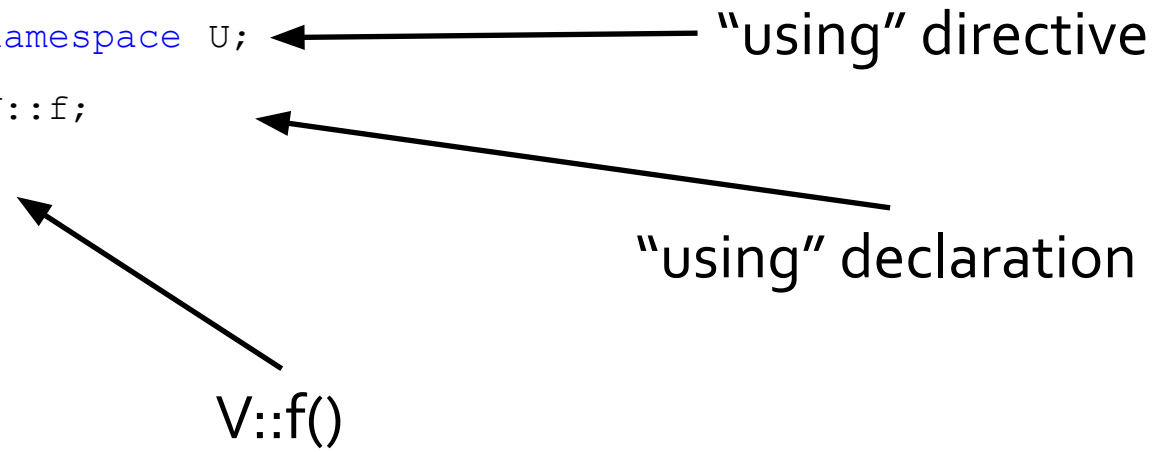
```
lecture12.cc: In function 'void testFive()':

lecture12.cc:166:28: error: call of overloaded
'divide(Int::Integer&, Int::Integer&)' is ambiguous

     Integer c = divide(a, b);
                              ^
```

`using` Declaration

- You can "inject" names one at a time with a `using` **declaration**
  - `using` **declaration** is different from the `using` **directive**
  - You can use the `using` declaration to specify a name (as opposed to the entire namespace with the `using` directive)

Example

```
namespace U {
  inline void f() {}
  inline void g() {}
}
namespace V {
  inline void f() {}
  inline void g() {}
}
using namespace U;
using V::f;
f();
U::f();
```



Questions?

static Members in C++

- When you need a single storage space for use by every object of a class
  - Use a global variable, but this allows other functions to change its value
  - Use #define, but this does not provide type checking
  - **Declare the member variable `static`**
- Every object for that class shares a single variable (i.e., any change to the static variable by a single object will be visible to every other object of that class)

Example

```
1.  class WithStatic {
2.    static int x;
3.    static int y;
4.  public:
5.    void print() const {
6.      cout << "WithStatic::x = " << x << endl;
7.      cout << "WithStatic::y = " << y << endl;
8.    }
9.    void inc() { x++; y++; }
10. };

11. int WithStatic::x = 1;
12. int WithStatic::y = x + 1;
```

You **must** define static data members outside of the class

Example

Will this run?
If so, what would happen?

```
1.  class WithStatic {
2.    static int x;
3.    static int y;
4.  public:
5.    void print() const {
6.      cout << "WithStatic::x = " << x <<
      endl;
7.      cout << "WithStatic::y = " << y <<
      endl;
8.    }
9.    void inc() { x++; y++; }
10. };

11. int WithStatic::x = 1;
12. int WithStatic::y = x + 1;
```

```
13. WithStatic a;
14. WithStatic b;
15. WithStatic c;
16. a.print();
17. a.inc();
18. b.print();
19. b.inc();
20. c.print();
```

Example

```
WithStatic::x = 1
WithStatic::y = 2

WithStatic::x = 2
WithStatic::y = 3

WithStatic::x = 3
WithStatic::y = 4
```

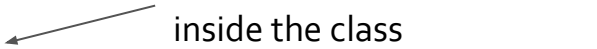
static const

- Remember from previous lecture that,
  - A static const variable inside a class is a non-changeable (const) variable that is shared across all objects of that class type

Example

Will this compile?

```
1.  class Values {
2.    static const int scSize = 100;
3.    static const long scLong;
4.  public:
5.    void print() { cout << scSize <<  " " << scLong << endl; }
6.  };

7.  void testSeven()
8.  {
9.      Values x;
10.     Values y;
11.     x.print();
12.     y.print();
13. }
```



Error

- /usr/bin/ld: /tmp/ccBEw3NM.o: in function `Values::print()':
- lecture12.cc:(.text._ZN6Values5printEv[_ZN6Values5printEv]+0x32): undefined reference to `Values::scLong'
- collect2: error: ld returned 1 exit status

Example

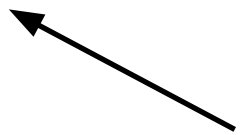
This will compile

```
1.   class Values {
2.     static const int scSize = 100;
3.     static const long scLong;
4.   public:
5.     void print() { cout << scSize <<  " " << scLong << endl; }
6.   };
7.  const long Values::scLong = 1000000;
8.   void testSeven()
9.   {
10.        Values x;
11.        Values y;
12.        x.print();
13.        y.print();
14.   }
```

What about arrays?

- Works similarly to regular variables

```
class Values {
    static const int scInts[];
};
const int Values::scInts[] = {
  99, 47, 33, 11, 7
};
```



Arrays must be initialized **outside the class** (i.e., not inline) - C++ standard

What about static (const) **objects** inside a class?

- You can also create static const **objects** and arrays of such objects
- However, you cannot initialize them inline – they must be **initialized externally (like arrays)**
- Objects behaves similarly to other static variables (i.e., they are persistent across all objects of the same type)

Example

```
class X {
  int i;
public:
  X(int ii) : i(ii) {}
};
```

```
class Stat {
  // This doesn't work
  //!  static const X x(100);
  // Both const and non-const static class
  // objects must be initialized externally:
  static X x2;
  static X xTable2[];
  static const X x3;
  static const X xTable3[];
};
X Stat::x2(100);
X Stat::xTable2[] = {
  X(1), X(2), X(3), X(4)
};
const X Stat::x3(100);
const X Stat::xTable3[] = {
  X(1), X(2), X(3), X(4)
};
int main() { Stat v; } ///:~
```

static Functions

- You can also declare member functions `static`
  - This function "works" for the class as a whole, rather than for a particular object
  - Because it is associated with the class, and not with objects, **it does not get passed the `this` pointer**
- What does this mean for a function exactly?
  - **A `static` function can only access static variables**
  - It **cannot** access the regular data members
  - It can also only call other `static` functions

Example

It is more common to call it using the **class name and the scope operator** (rather than associating it with an object via -> or .)

```
class X {
public:
  static void f(){};
};

int main() {
  X::f();
} ///:~
```

Example

```
class X {
  int i;
  static int j;
public:
  X(int ii = 0) : i(ii) {
    // Non-static member function can access
    // static member function or data:
    j = i;
  }
  int val() const { return i; }
  static int incr() {
    //! i++; // Error: static member
    // function cannot access non-
    // static member data
    return ++j;
  }
  static int f() {
    //! val(); // Error: static function
    // cannot access non-static function
    return incr(); // OK
  }
}; // class X
int X::j = 0;
int main() {
  X x;
  X* xp = &x;
  x.f();
  xp->f();
  X::f(); // Only works with static
          // member functions
} ///:~
```

Questions?

Variable Scope

- Variable created inside a function exists only within the function
  - The scope of the variable is the function it is defined in
  - Because it is created on the stack, it is `destroyed' when the function ends
- What if you want it to exist outside of the function?
  - Make it global
  - Make it ***static***
- ***static*** variable inside a function
  - NOT created on the stack
  - Created on the program's static data area
  - Initialized only ONCE (the first time the function is called)

Example

Will this run?
If so, what would happen?

```
1.  char oneChar(const char* charArray =
    nullptr) {
2.      static const char* s = nullptr;
3.      if(charArray) {
4.          s = charArray;
5.          return *s;
6.      } else if(!s) {
7.          cout << "Uninitialized char* s\n”;
8.          exit(EXIT_FAILURE);
9.      }
0.      if(*s == '\0') {
1.          return 0;
2.      }
3.      return *s++;
4.  }
```

```
15. const int sz = 100;
16. char a[sz] =
    "abcdefghijklmnopqrstuvwxyz";
17.

18. int main() {
19.     oneChar(a);
20.     char c;
21.     while((c = oneChar()) != 0) {
22.         cout << c << endl;
23.     }
24. }
```

Example

Will this run?
If so, what would happen?

```
1.  char oneChar(const char* charArray =
    nullptr) {
2.      static const char* s = nullptr;
3.      if(charArray) {
4.          s = charArray;
5.          return *s;
6.      } else if(!s) {
7.          cout << "Uninitialized char* s\n";
8.          exit(EXIT_FAILURE);
9.      }
0.      if(*s == '\0') {
1.          return 0;
2.      }
3.      return *s++;
4.  }
```

```
15. const int sz = 100;
16. char a[sz] =
    "abcdefghijklmnopqrstuvwxyz";
17.

18. int main() {
19.     oneChar(a);
20.     char c;
21.     while((c = oneChar()) != 0) {
22.         cout << c << endl;
23.     }
24. }
```

Example

Will this run?
If so, what would happen?

```
1.  char oneChar(const char* charArray =
    nullptr) {
2.      static const char* s = nullptr;
3.      if(charArray) {
4.          s = charArray;
5.          return *s;
6.      } else if(!s) {
7.          cout << "Uninitialized char* s\n”;
8.          exit(EXIT_FAILURE);
9.      }
0.      if(*s == '\0') {
1.          return 0;
2.      }
3.      return *s++;
4.  }
```

```
15. const int sz = 100;
16. char a[sz] =
    "abcdefghijklmnopqrstuvwxyz";
17.

18. int main() {
19.     oneChar(a);
20.     char c;
21.     while((c = oneChar()) != 0) {
22.         cout << c << endl;
23.     }
24. }
```

static variables

- Be careful when using static variables.
- Because they retain their values between function calls, you need to keep track of it carefully
- This is particularly true when using multi-threaded programming – if multiple threads are accessing the static variable, it's difficult to know if it's in the correct 'state.'
- For example, say a static variable is keeping track of how many times you rang a bell. If you are the only one pressing it, you can keep track of this count accurately. However, if someone else is also pressing the bell (i.e., a multi-threaded code), you've just lost track of how many times you've pressed the bell.

static class objects

- Same rules apply for static objects inside functions

Example

Will this run?
If so, what would happen?

```
1.  class X {
2.    int i;
3.  public:
4.    X(int ii = 0) : i(ii) {}
5.    ~X() { cout << "X::~X()" << endl; }
6.    void print_X() { cout << i << endl; i++;
    }
7.  };
8.  void testTwo()
9.  {
10.     static X x(47);
11.     x.print_X();
12.     x.print_X();
13. }
```

```
14.    int main() {
15.        testTwo();
16.        testTwo();
17.        testTwo();
18.    }
```

Example

```
47
48
49
50
51
52
X::~X()
```

static class Object Destructor

- Remember,
  - Destructors are called for objects that has been constructed
  - Global objects are created before main and destroyed when main ends
- If a function containing a local static object is never called, the constructor is never executed, and therefore, the destructor is not executed

Example

Will this run?
If so, what would happen?

```
1.  class Obj {
2.    char c;
3.  public:
4.    Obj(char cc) : c(cc) {
5.      cout << "Obj::Obj() for " << c << endl;
6.    }
7.    ~Obj() {
8.      cout << "Obj::~Obj() for " << c << endl;
9.    }
10. };
11. Obj aa('a');
12. void f() {
13.   static Obj b('b');
14. }
15. void g() {
16.   static Obj c('c');
17. }
```

```
18. void
    testThree()
19. {
20.   f();
21. }
22. int main() {
23.   testThree();
24. }
```

Different Meaning to static

- We've covered this before, but…
  - Any name at file scope (i.e., not inside a class or a function, so essentially global) is visible to other files at link time – this is called "external linkage"
    - Exception to this is the `const` variables
  - However, sometimes you want these names to be only visible to the file it resides in (maybe it clashes with names in a different file)
  - Name it `static` to make it invisible to files outside – i.e., it has "internal linkage"
- This has **different meaning** from our usage of static from the previous few slides
  - Global variables are already "static" in nature (it persists across the lifetime of the program), so adding static in front of it has a different meaning (i.e., internal/external linkage)

Example

- In file a.cc,
- `static int a = 1;`
-
- `int main() {`
- `    return 0;`
- `}`
- The variable a is only visible within a.cc

Questions?

Object Creation

Remember that….

when an object is created in C++, two events occur

- Storage is allocated
- Constructor is called to initialize the object

Storage Allocation

Storage allocation can occur in one of several ways

- Before the program begins, allocated in the static area (i.e., exists for the lifetime of the program, such as global variables)
- Allocated in the stack (e.g., opening braces, function, etc.), and released at the end of its scope
- Allocated on the heap

This is similar to how things are done in C

C++

Dynamic memory in C++

- `new` **operator**
  - Not a function (e.g., malloc)
  - `MyType *fp = new MyType(1,2);`
    - Equivalent to `malloc(sizeof(MyType))`, and
    - Its constructor will be called
- `delete` **operator**
  - Not a function (i.e., free)
  - Can be called for any object created with `new`
  - Destructor will first be called, and then the memory will be released
  - **Undefined if allocated with `malloc` (or other variations)**

Example

```
class Tree {
int height;
public:
  Tree(int treeHeight) : height(treeHeight) {}
  ~Tree() { std::cout << "*"; }
  friend std::ostream&
  operator<<(std::ostream& os, const Tree* t) {
    return os << "Tree height is: "
              << t->height << std::endl;
  }
};
using namespace std;
int main() {
  Tree* t = new Tree(40);
  cout << t;
  delete t;
} ///:~
```

Example

```
Tree height is: 40
*
```

`delete`

Deleting a `void*` is probably a bug

- It only okay for simple data (e.g., does not require a destructor)
- Only the storage is released, and the destructor is not called (the program does not know which destructor to call, since it can be anything)

Example

```
class Object {
  void* data; const int size; const char id;
public:
  Object(int sz, char c) : size(sz), id(c) {
    data = new char[size];
    cout << "Constructing object " << id
         << ", size = " << size << endl;
  }
  ~Object() {
    cout << "Destructing object " << id << endl;
    delete []data; // OK, just releases storage,
  }
};
int main() {
  Object* a = new Object(40, 'a'); delete a;
  void* b = new Object(40, 'b');   delete b;
} ///:~
```

Example

```
Constructing object a, size = 40
Destructing object a
Constructing object b, size = 40
```

Delete

If you have a memory leak (and you are deleting `new` allocated memory), check the type for the pointer being deleted

When you are using `void* data` to hold different types of objects, cast it to the proper type before using it (and deleting it)

`new` and `delete` for Arrays

To create an array of objects

```
MyType* fp = new MyType[100];
```

- This allocates enough memory on the heap for 100 `MyType` objects

Let's say you also have

```
MyType* fp2 = new MyType;
```

What if you do

```
delete fp2;
delete fp;
```

What happens?

`new` and `delete` for Arrays

```
MyType* fp = new MyType[100];
delete fp;
```

This DOES free up storage for all 100 `MyType` objects, BUT the constructor for only the FIRST element will be called

- Just as in `free()`, the OS keeps track of the memory allocated
- 99 other elements will NOT have their constructors called

Under what scenario could this cause a problem?

`new` and `delete` for Arrays

```
MyType* fp = new MyType[100];
delete [] fp;
// old syntax - delete [100]fp;
```

Questions?

Running Out of Memory

When `new` cannot find enough memory to allocate, it calls a special function called `new-handler`

- More accurately, there is a function pointer, and if the pointer is non-zero, this function is called

Example

```
int count = 0;
void out_of_memory() {
  cerr << "memory exhausted after " << count
    << " allocations!" << endl;
  exit(1);
}
int main() {
  set_new_handler(out_of_memory);
  while(1) {
    count++;
    new int[1000]; // Exhausts memory
  }
} ///:~
```

Arrays of Objects

With **arrays**, you can only call the default constructor - what if you need to use a non-default constructor

- Use a **vector** - the easier solution
- Use an **array of pointers** - for each element, use `new` with a non-default constructor to create a pointer and save it there
- **Placement-new** - a special case of `new` where you can pre-allocate the memory, then initialize it with a non-default constructor by passing in the pre-allocated memory location
- **Overload** the `new` operator (operator overloading will be covered later)

Arrays of Objects

Use an array of pointers

```
MyClass** list = new MyClass*[100];
for(int i = 0; i < 100; i++) {
  list[i] = new MyClass(i);
}

for(int i = 0; i < 100; i++) {
  delete list[i];
}
```

delete [] list;

Placement-New Example

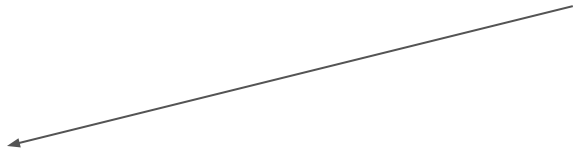
```
class X {
  int i;
public:
  X(int ii = 0) : i(ii) { cout << "this = " << this << endl; }
  ~X() { cout << "X::~X(): " << this << endl; }
};
int main() {
  int L[10];
  cout << "L = " << L << endl;
  X* xp = new(L) X(47); // X at location L[0]
  X* xp = new(L+1) X(53); // X at location L[1]
  // etc.
  xp2->X::~X(); // Explicit destructor call required
  xp->X::~X(); // Explicit destructor call required
}
```

Separates memory allocation and initialization

Another Example

```
class Car {
    int _no;
public:
    Car( int no ) : _no( no ) { }
};

int main() {
    void* raw_memory = operator new(NUM_CARS * sizeof(Car));
    Car* ptr = static_cast<Car*>(raw_memory);
    for( int i = 0; i < NUM_CARS; ++i ) {
        new (&ptr[i]) Car( i ); // placement-new
    }
    // destruct in reverse order
    for( int i = NUM_CARS - 1; i >= 0; --i ) {
        ptr[i].~Car();
    }
    operator delete [] ( raw_memory ); // only frees memory
    return 0;
}
```

ONLY allocates memory



Questions?