CSCI-UA.0480-001

Special Topics:

Multicore Programming

Lecture 2

C++ Crash Course for C Programmers

Christopher Mitchell, Ph.D.

cmitchell@cs.nyu.edu || http://z80.me

Lecture 2 Outline

- I/O
- Classes and Structs
  - Constructors/Destructors
  - Visibility
  - Inheritance & Polymorphism
- Memory Allocation
- Pointers
- Casting
- Strings
- Templates
- Data Structures
- Const & References

Take-Home Exercises

- Inheritance and Virtual
- Diamond and Multiple Inheritance
- Templated Containers
- CRTP
- Templates: SFINAE

Foundational Concepts

- C is not C++ with extra bits glued on top
  - Change your design patterns
- `const` correctness (more on this later)
- References (more on this too)
- Data structures are there to help
- Avoid bare pointers and bare casts
  - Including C-style strings (char *)
- Template metaprogramming
- C++11 and later – for a future discussion

I/O

- `printf()` and `scanf()` are deprecated (sort of…)
- Preferred: cout and cin
  - #include <iostream>
  - Introducing stream operators: << and >>

```
int i = 3;
std::cout << "string here: " << i << std::endl;
int j;
std::string s;
std::cin >> j;
std::cin >> s;
std::cout << "int: " << j << ", string: " << s
          << std::endl;
```

Classes & Structs: Containers for Everything

- Structs (and classes) (and enums) are first-class members!
  - Don’t add “`class`”/“`struct`”/”`enum`” prefix
- Classes and structs are the same thing, except…
  - `classes` default to private visibility
  - `structs` default to public visibility
- Member variables
- Member functions: “methods”
- Instantiation: classes/structs as “cookie cutter”

Ease-In Example: Structs

- C structs:

```
struct MyStruct {
    int member1;
    char* member2;
    struct OtherStruct member3;
};
```

- C++ structs:

```
struct MyStruct {
    int member1;
    std::string member2;
    OtherStruct member3;
};
```

Constructors and Destructors

- The "cookie" is reserved in memory
  - Same as C structs
- New special functions when creating/destroying.

```
class Foo {
    int width_, height_;
  public:
    Foo();
    Foo(const int width, const int height);
    ~Foo();
};
```

Constructors and Destructors

```
class Foo {
    int width_, height_;
  public:
    Foo();
    Foo(const int width, const int height);
    ~Foo();
};

void myfunc() {
    Foo foo1; // uses Foo(); constructor
    Foo foo2(32, 16); // uses Foo(int, int);
    // ... Use foo1 and foo2
} // ~Foo(); will be called on both foo1 and foo2
```

Inheritance

```
struct Parent {
    int i;
    std::string s;
};
struct Child : public Parent {
    double f;
};
```

Child has i, s, and f.

Demo: Access members

Visibility

- Who can see what's in a class or struct
- Public
  - Anyone can access it
- Protected
  - Only derived classes can access it
- Private
  - Only the class itself can access it

Who Can See It?

```
class Foo {
    double bar_;
  protected:
    int parentId();
};

class FooChild : public Foo
{
    FooChild();
    func() {
        bar_ += 3.0;
        parentId();
    }
};
```

```
struct Baz {
    double bar_;
 protected:
    int parentId();
};

class BazChild : public Baz
{
    BazChild();
    func() {
        bar_ += 4.0;
        parentId();
    }
};
```

Polymorphism (1)

```
// Example program
#include <iostream>
#include <string>

class Parent {
  public:
    Parent() {
        printName();
    }
    void printName() {
      std::cout << "Parent"
                << std::endl;
    }
};
```

```
class Child : public Parent {
  public:
    void printName() {
      std::cout << "Child“
                << std::endl;
    }
};

int main() {
    Parent parent;
    Child child;
    parent.printName();
    child.printName();
}
```

Polymorphism (2): Virtual

```
class Parent {
  public:
    Parent() {
        printName();
    }
    virtual void printName() {
      std::cout << "Parent"
                << std::endl;
    }
};
```

```
class Child : public Parent {
  public:
    void printName() {
      std::cout << "Child" <<
std::endl;
    }
};

int main() {
    Child* child = new Child;
    child->printName();
    Parent* parent =
        (Parent*)child;
    parent->printName();
}
```

Inherited Animals

```
struct Animal {
    unsigned int legs;
    double height;
    double mass;
};
struct Elephant : public Animal {
    double trunk_length;
};
struct Bird : public Animal {
    double wingspan;
    bool migratory
};
struct Swallow : public Bird {
    enum {
        AFRICAN,
        EUROPEAN
    } locale;
};
```

Demo: print the animal's information

Memory Allocation

- Don't use `malloc()` and `free()`!
- OLD:

```
struct Foo* foo = (struct Foo*)malloc(sizeof(struct Foo));
//…operations on foo
free(foo);
```

- NEW:

```
Foo* foo = new Foo();
//…operations on foo
delete foo;
```

Pointers

- You already know bare pointers:
  `struct MyStruct*`
- Meet smart pointers (C++11 and higher)
  - `std::shared_ptr<MyStruct> my_ptr = std::make_shared<MyStruct>(arg, args);`
    - Automatically destroyed *and* freed with destruction of last reference
  - `std::unique_ptr<MyStruct> my_ptr2 = std::make_unique<MyStruct>(arg, args);`
    - Can only be stored in one pointer instance at a time.
    - Must use `std::move()` if you want to move or return it!

Casting

- Say we have a `Child` class, derived from `Parent` class, a `parent_ptr` pointer to a `Parent`, and `child_ptr` to a `Child`
- Old and busted:
```
Parent* parent_ptr = (Parent*)child_ptr;
```
- New hotness:
```
Parent* parent_ptr =
    dynamic_cast<Parent*>(child_ptr);
Child* child_ptr2 =
    dynamic_cast<Child*>(parent_ptr);
```

Types of Casts

- Especially for casts up and down inheritance hierarchy, can also be used for primitive types (e.g. char -> int)
- static_cast<type>(input):
  Performed at compile type. Error?
- dynamic_cast<type>(input):
  Performed at runtime. Error?
- Let’s not worry about:
  - const_cast<type>(input)
  - reinterpret_cast<type>(input)

Strings

- C++ class to store, well, strings
- Common methods on `std::string s`:
  - `s = "Hello, World"`: Initialize/replace stored string
  - `s.length()`: Get length of string
  - `s.substr(3, 10)`: Get 10-character substring starting at position 3, returned as a new std::string
  - `s += ", today";`: Append string to string
  - `s.c_str()`: Get char* pointer to data, useful to use strings with `printf()`/`fprintf()`.
- https://en.cppreference.com/w/cpp/string

Templates

- We'll barely scratch the surface of the surface.
- Basic concept: allow functions/structs/classes to be reused with multiple types
  - Common example: containers (we'll see some soon)
- Function with template parameter:

```
template<typename T>
bool function(const T& arg);
```

- Class with template parameter:

```
template <typename T>
class MyClass { … };
```

Template Example: Comparison

```
template<typename C>
bool getMax(const C a, const C b) {
    return a > b ? a : b;
}

std::cout << getMax(1.5, 2.5) << std::endl;
unsigned long long int i = 1e10, j = 1e11;
std::cout << getMax(i, j) << std::endl;
std::string a = "hello", b = "world";
std::cout << getMax(a, b) << std::endl;
```

Will this compile? What will it print?

Template Example: Container

```
template<typename Elem>
class MyPair {
    Elem first_;
    Elem second_;
  public:
    MyPair(const Elem& first, const Elem& second)
    : first_(first), second_(second)
    {}
    const Elem& getFirst() const { return first_; }
    const Elem& getSecond() const { return second_; }
    void print() { std::cout << first_ << ", "
                             << second_ << std::endl; }
};

MyPair<int> pair1(1, 10);
MyPair<std::string> pair2("Multicore", "Programming");
pair1.print();
Pair2.print();
```

Data Structures

Common data structures

- **`std::vector<element>`**
- **`std::map<key, value>`**
- `std::set<key>`
- `std::unordered_map<key, value>`
- `std::unordered_set<key>`
- `std::list<element>`

`std::vector`

- Ordered list of elements
- O(1) random access, O(1) append, O(n) insert, O(n) delete
- Common methods on `std::vector<double> vec`:
  - `vec.push_back(3.14)`: append an element
  - `vec[2]`: Access element 2 (read or write)
  - `vec.size()`: Return the size of the vector
  - `vec.clear()`: Empty the vector
  - `vec.front()`, `vec.back()`: Access first (or last) element
  - `vec.insert(it, 2.7)`: Insert an element within the vector
- https://en.cppreference.com/w/cpp/container/vector
- Demo

`std::vector` Iteration

- Iterator methods
  - `.begin()` and `.end()`: Iterators to first and after-last elements
  - `.cbegin()` and `.cend()`: *Constant* iterators to first and after-last elements
  - `iterator++`: Increment to next element

std::vector Iteration

Example:

```
std::vector<std::string> strings;
// … insert elements into strings
for(std::vector<std::string>::iterator it =
    strings.begin(); it != strings.end(); it++)
{
    std::cout << *it << std::endl;
}

for(auto it = strings.begin(); it != strings.end(); it++) {
    std::cout << *it << std::endl;
}
```

Aside: Typedefs

- Syntactic sugar: save typing a long type
- Example:

```
typedef std::vector<std::string> StringVec;

StringVec strings;
// … insert elements into strings
for(StringVec::iterator it =
    strings.begin(); it != strings.end(); it++)
{
    std::cout << *it << std::endl;
}
```

`std::map`

- Ordered map of key->value pairs
- O(log(n)) random access, O(log(n)) insert and delete
- Common methods on `std::map<std::string, double> mymap`:
  - `mymap.insert("pi", 3.14)`: insert an element
  - `mymap["e"]`: Read or write value for key “e”
  - `mymap.size()`: Return the size of the map
  - `mymap.clear()`: Empty the map
  - `mymap.find("akey")`: Return iterator to (key, value) pair if mymap contains key “akey”, or return `mymap.end()` otherwise.
- https://en.cppreference.com/w/cpp/container/map
- Demo

Finding Elements in Maps

Example:

```
typedef std::map<std::string, std::string> StringStringMap;
StringStringMap phones;
phones.insert("Sandra", "19175554321");
phones.insert("John", "12125551234");
const StringStringMap::iterator thomas_phone =
    phones.find("thomas");
const auto sandra_phone = phones.find("Sandra");
```

What do `thomas_phone` and `sandra_phone` contain?

Const Correctness

- New C++ keyword: const
  - Usually refers to something that can't be modified
- `void myfunc(const MyClass instance) { … }`
  - Can't modify instance inside myfunc
- `const int i = 5;`
  - Can't modify i after initialization
- `const double* d = new double(3.14);`
  - Can't modify *contents* that d points to
- `double * const d2 = new double(2.717);`
  - Can't modify the pointer address
-
```
class MyClass {
    void mymethod(args) const {…}
};
```
  - Can't modify any state stored in an instance of MyClass

References

- Like pointers, but better
  1. References cannot be null
  2. References cannot be uninitialized
  3. References cannot be *reseated*
  4. Don't need to dereference

- ```
  MyClass instance;
  MyClass& ref = instance; // reference to instance
  instance.method();
  ref.method();         // these are the same
  ```

- ```
  void myFunc(const std::string& string_arg) {
      // Avoid copying the entire instance onto stack
  }
  ```

Take-Home Exercises

Inheritance & Virtual

- Consider class A and B
- Which fields and funcs will be accessed from:
  - aa_ref?
  - ab_ref?
  - bb_ref?

```
class A {
  public:
    int field;
    bool func();
    virtual func_virtual();
};
class B {
  public:
    int field;
    bool func();
    func_virtual();
};
A a;
B b;
A& aa_ref = a;
A& ab_ref = b;
B& bb_ref = b;
```

Templated Linked List

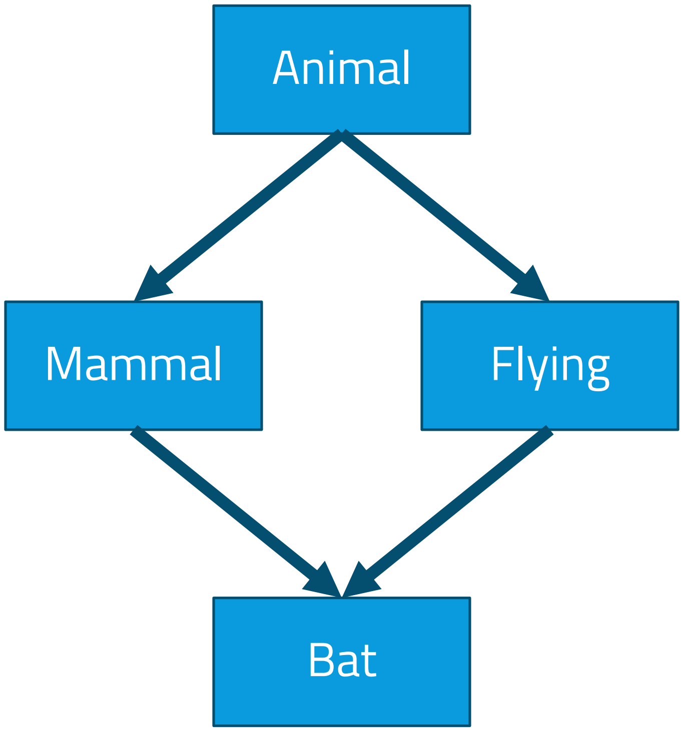
- Create a templated linked list, using unique pointers, that announces its values when the destructor on each node is called

Templated DAG

- Create a templated DAG with shared pointers that announces its values when the destructor is called on each node.

Diamond & Multiple Inheritance

- Consider the inheritance of four classes as shown
- Explain how you can get:
  - A bad memory layout with two Animal subobjects
  - A good memory layout with one Animal suboject



Templates: SFINAE

SFINAE: Substitution Failure Is Not An Error. Explain how/why this works.

```
template<typename T>
class is_class {
    typedef char yes[1];
    typedef char no [2];
    // selected if C is a class type
    template<typename C> static yes& test(int C::*);
    // selected otherwise
    template<typename C> static no&  test(...);
  public:
    static bool const value = sizeof(test<T>(0)) == sizeof(yes);
};
class random_class { };

std::cout << "int is class? " << is_class<int>::value << ", “
          << "class is class? " << is_class<random_class>::value << std::endl;
```

CRTP:
Curiously Recurring Template Pattern

Observation: Method of class template instantiated only when needed.

```
#include <iostream>

template<class T>
struct Lazy {
    void func() { std::cout << "func" << std::endl;}
    void func2(); // not defined
};

int main() {
    Lazy<int> lazy;
    lazy.func();
}
```

CRTP:
Curiously Recurring Template Pattern

```
#include <iostream>

template <typename Derived>
struct Base {
  void interface() {
    static_cast<Derived*>(this)->impl();
  }
  void impl() {
    std::cout << "Impl Base" << std::endl;
  }
};

struct Derived1: Base<Derived1>{
  void impl() {
    std::cout << "Impl Derived1" << std::endl;
  }
};

struct Derived2: Base<Derived2>{
  void impln() {
    std::cout << "Impln Derived2" << std::endl;
  }
};
```

```
struct Derived3: Base<Derived3>{};

template <typename T>
void execute(T& base) {
    base.interface();
}

int main(){
    Derived1 d1;
    execute(d1);
    Derived2 d2;
    execute(d2);
    Derived3 d3;
    execute(d3);
}
```

A CRTP Calculator

- Build a simple calculator that uses CRTP to build an AST of operations.