ECE 590.01
C++ Programming, Data structures, and Algorithms

C -> C++: Classes and Templates
Admin

- Quiz 1
Remind us where we left off last time?

- Who can remind us what we talked about?
Remind us where we left off last time?

• Who can remind us what we talked about?
  • ADTs: interface
    • Stack
    • Queue
    • Deque
  • Implementations
    • Array
    • Linked List (singly)
    • Linked List (doubly)

• What was ugly about these?
Remind us where we left off last time?

- Who can remind us what we talked about?
  - ADTs: interface
    - Stack
    - Queue
    - Deque
  - Implementations
    - Array
    - Linked List (singly)
    - Linked List (doubly)

- What was ugly about these?
  - void *: any type
  - As long as you have a pointer to it... and are careful
Now: C++ (where we can fix this)

- C++ started life as “C with classes”
  - C++ compiler translated C++ into C
    - Class = structs with function pointers
    - More on this in April (object layout etc)

- Some of C++’s design decisions driven by this translation
  - Ugly language choices for translation convenience? Ick
C++: Classes

class LinkedList {

};  ← Java programmers note the ;
C++: Classes

class LinkedList {
    private:  <- Access restrictions by section
    class LLNode {
        public:
            void * data;
            LLNode * next;
            LLNode(void * d, LLNode * n) : data(d),
                                 next(n) {}
        }
    LLNode * head;
};
C++: Classes

class LinkedList {
    private:
    class LLNode {
        public:
            void * data;
            LLNode * next;
            LLNode(void * d, LLNode * n) : data(d), next(n) {}
        }
    LLNode * head;
};

<-Node class can be private
C++: Classes

class LinkedList {
    private:
    class LLNode {
        public:
            void * data;
            LLNode * next;
            LLNode(void * d, LLNode * n) : data(d),
            Constructor with initializer list
            next(n) {}
        }
    LLNode * head;
};
class LinkedList {
  private:
  class LLNode {...};
  LLNode * head;
  public:
  LinkedList(): head(NULL) {}
  void addFront(void * d) {
    head = new LLNode(d, head);
  }
  new instead of malloc
  Behaves a bit more like java: new T returns a T*
  Also runs constructor with arguments passed in
  Still needs to be freed, but use delete instead of free
}
OOP: Why is it good?

• So far, classes have given us two things
  • Encapsulation: data packaged up with the functions that operate on them
  • Information hiding:
    • Public/private allow for restrictions in visibility
    • Helps separate interface from implementation

• How does this work?

<table>
<thead>
<tr>
<th>What You Write</th>
<th>What Really Happens</th>
</tr>
</thead>
<tbody>
<tr>
<td>void addFront(void * d)</td>
<td>void addFront(LinkedList * this, void *d)</td>
</tr>
<tr>
<td>myList-&gt;addToFront (x);</td>
<td>addToFront (myList, x);</td>
</tr>
</tbody>
</table>
Now, to fix the use of `void *` everywhere: templates

```cpp
template<class T>
class LinkedList {

class LLNode {
    T data;
    LLNode * next;
    LLNode(T d, LLNode *n) : data(d), next(n) {}
};
LLNode * head;

void addFront(T d) {
    head = new LLNode(d, head);
}

T peekFront() const {
    return head->data;
}
};
```
Templates

```cpp
template<class T> class LinkedList {
...

• Now, LinkedList is a template, not a class
  • But when given its template parameter, it makes a type:
    • LinkedList<int>
    • LinkedList<double>
    • LinkedList< LinkedList < int > >
      • Note: > (space) > to close two template arg lists
• Templates can be parameterized over multiple/any things
  template<class T, int X, class S> class Whatever{...};
• We can have template functions outside of a class:
  template<class T> T max(T a, T b) { return a < b ? b : a; }
  (Note: would want to change this: see why shortly)
```
Templates: the ugliness

- Ugliness of templates
  - Entire definition must be visible at point of use
  - Not just interface!
  - Basically, “write all your code for templated classes in the .h files”
  - Why? How they are implemented (and C++ -> C translation)

- Templates get expanded at use
  - Compiler instantiates the template at the type its used on
    - Then type checks it: consequence only has to be valid for types used on!
      - C++ designers: YAY!
      - Me: ICK!
  - Name mangling: prevent duplication of names between instances
  - Annotations in .o files (“weak linkage”): prevent duplication on same instance
Name Mangling

- `addFront` **mangled** into two names for two types:
  - `_ZN10LinkedListIPKcE8addFrontES1_`
  - `_ZN10LinkedListIIIE8addFrontEI`
- Don’t need to care about specifics, just good to know that it happens
- Prevents duplicate names
- Also, how function/operator overloading works
  - `int foo(int x);`
  - `double foo(double x);`
Overloading

- Function overloading: same names, different arg lists
  - int foo(int x)
  - double foo(double x)
    - Pretty reasonable for some things (Java’s SOP)
      - Should basically do the same thing

- Operator overloading: takes one step further
  - Can overload operators: +, -, *, [], (), /, <<, >>...
  - Decent idea: + should work on ints, doubles, etc..
    - Add a “BigNum”, reasonable for + to work on them to (do add)
  - Ridiculous: the abuses this gets with “hey I want an operator to do something whacky so I don’t have to write a whole 6 letters”
Operator Overloading (cont’d)

• General wisdom: operator overloading isn’t too bad as long as programmer’s stick to the same meaning the operator had

• C++ designers: Yeah, so how about we use << for “print” and >> for “read”

• Rest of the world: Those are LEFT SHIFT and RIGHT SHIFT

• Thing I hate most about C++: C++ style IO.
C++ style IO

• C’s IO (printf, fprintf, scanf,...) all quite sane
• C++’s IO
  • #include <iostream>  //C++ decided not to use .h for system hdrs

  • << = print
  • >> = read
  • cout << “Hello World x = “ << x << endl;
    • endl: because “\n” is too hard?
  • Want formatting?  << the functions that do it into the stream!
    • cout << “x = “ << hex() << x << dec() << endl;
    • And “put it back to normal” when done!
C++ copying values

• Remember

```c++
template<class T> max(T a, T b) { return a < b ? b : a; }
```

• I said we wouldn’t want to do this.. Why not?

• Works great for T = int

• But suppose we have T = BigComplexClassWithLotsOfData
  • And have overloaded < on it so that it works
  • Now will involve a lot of value copying...
A painful problem

- Template expands into:

```cpp
Bccwlod max(Bccwlod a, Bccwlod b) {
    return a < b ? b : a;
}
```

- Calling this function

```
Bccwlod x = max(y, z)
```

- Requires 3 Bccwlod copies (more if < passes by value!)
  - Copy y to a
  - Copy z to b
  - Copy (a or b) to x

- Painfully slow!
C++: References (but not like Java)

We could use pointers:

```c
Bccwlod * max(Bccwlod * a, Bccwlod * b) {
    return (*a) < (*b) ? b : a;
}
```

Except:
- < gets arguments passed by value
  - < already defined on pointers, can’t redefine
- Kind of cumbersome to add extra *s everywhere (?)
C++: References (but not like Java)

We could use pointers:

```cpp
Bccwlod & max(Bccwlod & a, Bccwlod & b) {
    return a < b ? b : a;
}
```

Instead, C++ introduces references.

- Like pointers, but automatically de-referenced at every used
- No such thing as NULL
- Must be initialized at declaration (except arguments)
- Can’t change what is referenced after initialization
  - Basically “another name for”
C++ references

```cpp
int x = 3;
int & a = x;  // a is another name for x
int y = 4;
a = 3;        // x = 3
a = y;        // x = 4
```
Inheritance

• With classes, comes inheritance

class A {
public:
    void foo() {printf("a:f\n");}
    void bar() {printf("a:b\n");}
};

class B: public A {
public:
    void foo() {printf("B:f\n");}
};

In java “extends A”
In C++ can make access more
Restrictive in things inherited
from A. Generally want “public”
to keep them the same
Inheritance

• With classes, comes inheritance

```cpp
class A {
public:
    void foo() {printf("a:f\n");}
    void bar() {printf("a:b\n");}
};

class B: public A {
public:
    void foo() {printf("B:f\n");}
};
```

Make a B, and call its foo() function. (nothing special here)
Inheritance

• With classes, comes inheritance

class A {
public:
    void foo() {printf("a:f\n");}
    void bar() {printf("a:b\n");}
};

class B: public A {
public:
    void foo() {printf("B:f\n");}
};

B * b = new B();
b->foo();
b->bar();

B’s inherit bar() from A
Inheritance

- With classes, comes inheritance

```cpp
class A {
public:
    void foo() {printf("a:f\n");}
    void bar() {printf("a:b\n");}
};

class B: public A {
public:
    void foo() {printf("B:f\n");}
};
```

B * b = new B();
b->foo();
b->bar();
A * a = b;

An A is-A B, so we can treat it like one..
Inheritance

- With classes, comes inheritance

```cpp
class A {
public:
    void foo() { printf("a:f\n"); }
    void bar() { printf("a:b\n"); }
};

class B: public A {
public:
    void foo() { printf("B:f\n"); }
};
```

```cpp
B * b = new B();
b->foo();
b->bar();
A * a = b;
a->foo();
```

Which `foo()` does this call?
Inheritance

• With classes, comes inheritance

```cpp
class A {
public:
    void foo() {printf("a:f\n");}
    void bar() {printf("a:b\n");}
};

class B: public A {
public:
    void foo() {printf("B:f\n");}
};
```

Java: `a` is REALLY pointing at a `B`, so we use `B`’s `foo()` [always]

This is called dynamic dispatch
Inheritance

• With classes, comes inheritance

```cpp
class A {
public:
    void foo() { printf("a:f\n"); }
    void bar() { printf("a:b\n"); }
};

class B: public A {
public:
    void foo() { printf("B:f\n"); }
};
```

B * b = new B();
b->foo();
b->bar();
A * a = b;
a->foo();

C++: dynamic dispatch has a performance overhead, so you get static dispatch unless you ask for dynamic... We didn’t so this uses the static (declared) type of a (A) and calls A’s foo
Dynamic Dispatch

• With classes, comes inheritance

class A {
public:
    virtual void foo() {printf("a:f\n");}
    void bar() {printf("a:b\n");}
};

class B: public A {
public:
    void foo() {printf("B:f\n");}
};

Virtual keyword asks for method to be dynamically dispatched..

Needs to go in parent class
Dynamic Dispatch

• With classes, comes inheritance

```cpp
class A {
public:
    virtual void foo() { printf("a:f\n"); }
    void bar() { printf("a:b\n"); }
};

class B: public A {
public:
    virtual void foo() { printf("B:f\n"); }
};
```

Nice to write it again for clarity
Dynamic Dispatch

• With classes, comes inheritance

class A {
public:
    void foo() {printf("a:f\n");}
};

class B: public A {
public:
    virtual void foo() {printf("B:f\n");}
};

class C: public B {
    void foo() {printf("C:f\n");}
};

C * c = new C();
B * b = c;
A * a = b;
c->foo();
b->foo();
a->foo();

Which do each of these call?
Dynamic Dispatch

• With classes, comes inheritance

class A {
public:
    void foo() {printf("a:f\n");}
};

class B: public A {
public:
    virtual void foo() {printf("B:f\n");}
};

class C: public B {
    void foo() {printf("C:f\n");}
};

C * c = new C();
B * b = c;
A * a = b;
c->foo();
b->foo();
a->foo();

c->foo calls C’s foo [easy]
Dynamic Dispatch

• With classes, comes inheritance

```cpp
class A {
public:
    void foo() {printf("a:f\n");}
};

class B: public A {
public:
    virtual void foo() {printf("B:f\n");}
};

class C: public B {
    void foo() {printf("C:f\n");}
};
```

`C * c = new C();
B * b = c;
A * a = b;
c->foo();
b->foo();
a->foo();`

C * c = new C();
B * b = c;
A * a = b;
c->foo();
b->foo();
a->foo();

b->foo calls C’s foo

Statically, b is a B. B’s foo is virtual, so dynamic dispatch is used
Dynamic Dispatch

- With classes, comes inheritance

```cpp
class A {
public:
    void foo() {printf("a:f\n");}
};

class B: public A {
public:
    virtual void foo() {printf("B:f\n");}
};

class C: public B {
    void foo() {printf("C:f\n");}
};
```

C * c = new C();
B * b = c;
A * a = b;
c->foo();
b->foo();
a->foo();

ECE 590.01 (Hilton): C -> C++ 37
Design tradeoffs

• Java: always dynamic dispatch
  • Easy to understand
  • Easily readable
  • Always pay cost, even if not useful
  • Or do you?.... Hey compilers can be pretty smart!
  • Throw CFA at it, problem solved [maybe not in Java though]

• C++
  • Complicated rules
  • Harder to read: trace through code to figure out dispatch type
  • Pay cost when useful
  • Early 80s: computers, much slower. Extra insns = expensive
  • Now? Faster/wider/OOO computers, smarter compilers...
Abstract Classes

class A {
public:
    virtual void foo() = 0;
    void bar() {printf("a:b\n");}
};
class B: public A {
public:
    virtual void foo() {printf("B:f\n");}
};

- Abstract ("pure virtual") functions: = 0
- Cannot instantiate abstract classes, can only have ptrs to them.
ADTS with abstract classes

template<class T> class AbstractStack {
    public:
        virtual void push(const T & x) = 0;
        virtual const T & peek() const = 0;
        virtual T pop() = 0;
        virtual int size() = 0;
};
ADTS with abstract classes

template<class T> class AbstractStack {
    public:
        virtual void push(const T & x) = 0;
        virtual const T& peek() const = 0;
        virtual T pop() = 0;
        virtual int size() = 0;
};

This may be a bad idea: we’ll see why shortly
A templated LL

template<class T> class LinkedList {
    ...
    void addFront(const T & data) {
        head = new LLNode(data, head);
    }
    T popFront() {
        T ans = head->data;
        LLNode * temp = head->next;
        delete head;
        head = temp;
        return ans;
    }
    const T & peekFront() const {
        return head->data;
    }
    ...
}
ADTS with abstract classes

template<class T> class LLStack : public AbstractStack<T> {
private:
    LinkedList<T> list;
public:
    virtual void push(const T & x) {
        list.addFront(x);
    }
    virtual const T& peek() const {
        return list.peekFront();
    }
    virtual T pop() {
        return list.popFront();
    }
    ...

ADTS with abstract classes

AbstractStack<int> * x = new LLStack<int>();

Polymorphism lets us do this. Could “new” any subclass of AbstractStack here (any implementation). Rest of code should be oblivious: Uses x as AbstractStack<int>
ADTS with abstract classes

AbstractStack<int> * x = new LLStack<int>();
x->push(32);
x->push(4);
x->push(12);

Push onto the stack
ADTS with abstract classes

AbstractStack<int> * x = new LLStack<int>();
x->push(32);
x->push(4);
x->push(12);
const int & y = x->peek();
printf("%d\n", x->peek());

Peek and print it out
Prints 12
AbstractStack<int> * x = new LLStack<int>();
x->push(32);
x->push(4);
x->push(12);
const int & y = x->peek();
printf("%d\n", x->peek());
printf("%d\n", x->pop());
ADTS with abstract classes

AbstractStack<int> * x = new LLStack<int>();
x->push(32);
x->push(4);
x->push(12);
const int & y = x->peek();
printf("%d\n", x->peek());
printf("%d\n", x->pop());
printf("%d\n", y);  Can anyone guess what this prints?
ADTS with abstract classes

AbstractStack<int> * x = new LLStack<int>();
x->push(32);
x->push(4);
x->push(12);
const int & y = x->peek();
printf("%d\n", x->peek());
printf("%d\n", x->pop());
printf("%d\n", y);

No, nobody can guess.
We are using freed memory!
Remember when I said we were asking for trouble?

Freed! (deleted)
ADTS with abstract classes

AbstractStack<int> * x = new LLStack<int>();
x->push(32);
x->push(4);
x->push(12);
const int & y = x->peek();
printf("%d\n", x->peek());
printf("%d\n", x->pop());
printf("%d\n", y);
printf("%d\n", x->pop());
printf("%d\n", x->pop());

These do what you expect:
4
32
References: careful

• C++ programmers love to use references
• ...but as we just saw, you have to be careful
  • Really: pointers in disguise
  • References to invalid memory just as bad as pointers to invalid memory!
Other C++ things

• Default constructor
  • No arguments
  • Every class should have one! Should be sane
  • (Lot’s of templates etc rely on these)

• Copy constructor:
  • X(const X & rhs)
  • Used for initialization situations
  • Default: shallow copy (not appropriate? must write)
    • Many people like to write all the time anyways

• Assignment operator:
  • X & operator = (const X & rhs)
  • Specifies how to do assignment copy ( x = y;)
  • Handle case of x = x: usually if (this != &rhs) {...}
  • Should return *this
Destructors

- Constructors
  - Named same as class
  - Specify how to initialize when “new”ing

- Destructors
  - Named same as class but with a ~ on front
  - Specify how to release resources when “delete”ing
    - May delete other objects pointed to in this class
    - That may also run destructors...
  - May do other cleanup: close files, etc.
  - Generally public and takes no arguments

- Java actually has something like this called a “finalizer” which runs when an object is Gced.
Summary

• Now we are all C++ experts right?
  • Ha!
  • C++: giant hairball of a language...
  • So many cans of worms to open

• Later this semester:
  • Multiple inheritance
  • Virtual inheritance

• Not touching with a 10 foot pole:
  • Promotion/resolution rules for in-exact matches of overloaded functions
  • My view: if this comes into play, you are doing it wrong.