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

Admin

- Quiz 1
class LinkedList {
}; // C++ programmers note the ;

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

class LinkedList {
private:
    class LLNode { <Node class can be private
        public:
            void * data;
            LLNode * next;
            LLNode(void * d, LLNode *n) : data(d), next(n) {
            }
            LLNode * head;
    };
public:
    void * data;
    LinkedList * next;
    LinkedList(void * d, LinkedList *n) : data(d), next(n) {};
    Constructor with initializer list
    LinkedList * head;
};

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

What You Write
void addFront(void * d) {
    myList->addToFront(x);
}

What Really Happens
void addFront(LinkedList * this, void *d) {
    myList->addToFront(myList, x);
}
Templates

• Now, to fix the use of `void *` everywhere: templates
template<class T>
class LinkedList {
    class LLNode {
        T data;
        LLNode * next;
    };
    LLNode * head;
public:
    void addFront(T d) {
        head = new LLNode(d, head);
    }
    T peekFront() const {
        return head->data;
    }
};

ECE 590.01 (Hilton): C -> C++

Templates

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>>`
    • 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)

ECE 590.01 (Hilton): C -> C++

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

ECE 590.01 (Hilton): C -> C++

Name Mangling

• addFront mangled into two names for two types:
    _ZN10LinkedListIPKcE8addFrontES1_
    _ZN10LinkedListIiE8addFrontEi
    * 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);

ECE 590.01 (Hilton): C -> C++

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”

ECE 590.01 (Hilton): C -> C++

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.

ECE 590.01 (Hilton): C -> C++
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 << 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
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= BigClassWithLotsOfData
  - And have overloaded < on it so that it works
  - Now will involve a lot of value copying...

A painful problem

- Template expands into:
  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:

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:

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

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

```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");}
};
```

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

ECE 590.01 (Hilton): C -> C++

25

Make a B, and call its foo function.
```cpp
B * b = new B();
b->foo();
```

ECE 590.01 (Hilton): C -> C++

26

B's inherit bar from A

ECE 590.01 (Hilton): C -> C++

27

An A is A B, so we can
treat it like one.
```cpp
A * a = b;
```

ECE 590.01 (Hilton): C -> C++

28

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

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

Virtual keyword asks for method to be dynamically dispatched. Needs to go in parent class

Which do each of these call?

C * c = new C();
B * b = c;
A * a = b;
c->foo();
b->foo();
a->foo();
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");
    a->foo();
}
};
class C: public B {
void foo() {printf("C:f\n");}
};
```

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

A calls A's foo

Statically, a is an A. A's foo is not virtual, so static dispatch is used.

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!
- 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

```cpp
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");
    a->foo calls A's foo
}
};
```

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

ADTS with abstract classes

```cpp
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

```cpp
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;
};
```

A templated LL

```cpp
template<class T> class LinkedList {
...
    void addFront(const T & data) {
        head = new LNode(data, head);
    }
    T popFront() {
        T ans = head->data;
        LNode * temp = head->next;
        delete head;
        head = temp;
        return ans;
    }
    const T & peekFront() const {
        return head->data;
    }
};
```
**Abstract Stack**

```cpp
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();
    }
    ...;
};
```

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**

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

```
x->push(32);
x->push(4);
x->push(12);
const int & y = x->peek();
printf("%d", x->peek());
```

Push onto the stack

```
const int & y = x->peek();
printf("%d", x->peek());
printf("%d", x->pop());
```

Prints 12

```
printf("%d", y);
```

Can anyone guess what this prints?
ADTS with abstract classes

```cpp
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?

---

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
    - (Lots of templates etc rely on these)
- Copy constructor:
  - \texttt{X(const X & rhs)}
  - Used for initialization situations
- Assignment operator:
  - \texttt{X & operator = (const X & rhs)}
  - Specifies how to do assignment copy (\texttt{x = y;})
  - Handle case of \texttt{x = x}: usually if (this != &rhs) (...) Should return \texttt{*this}

---

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.