Why are you using C++?

All your friends are using it.

You were told to.

You aren't allowed to use Lisp anymore.

You want to do object-oriented programming.

C++ concepts

- Data abstraction
- Inheritance
- Classes, virtual functions
- Public/protected/private/friend
- Constructors, destructors
- Operator overloading
- Type conversion operators
- Inline functions

Questions:

When to inherit vs. instance?

Is multiple inheritance useful?

When to use pointers vs. references?

How to make parameterized classes?

How to build a class library?

How to debug a C++ program?

Schedule

1:30	Motivation
1:40	Language choices
2:40	Break
2:50	Libraries
3:50	Break
4:00	Environments
4:20	Summary

Language issues

Single/multiple inheritance

Pointers/references/smart pointers

Ambiguity

Principles

Just because it is there
doesn't mean you have to use it

Don't let your esthetics
prevent you from finishing the job

Simplicity is power

Inheritance and instances

```
class A {
public:
   int f();
};

class B : public A { };

class C {
private:
   A a;
public:
   int f() { return a.f(); }
   operator A&() { return a; }
};
```

Why not use inheritance?

- Scatters functionality
- Creates header file dependencies
- Might cause unexpected results
- Hard to undo in next release

Reasons to use inheritance

1. Redefining virtuals

2. Redefining constructors

3. Sharing implementation

Redefining virtuals

```
class Interactor {
public:
    virtual void Highlight(boolean);
};

class Message : public Interactor {
    virtual void Highlight(boolean);
};

 ...

Interactor* i;

i->Highlight(true);
```

Redefining constructors

```
class Menu : public Control {
public:
    Menu(Interactor*);

    void SetAlign(Alignment);
};

class PulldownMenu : public Menu {
public:
    PulldownMenu(Interactor*);
    PulldownMenu(const char*);
private:
    void Init();
};

PulldownMenu::PulldownMenu(Interactor* i) : (i) {
    Init();
};

void PulldownMenu::Init() {
    SetAlign(BottomLeft);
};
```

Sharing implementation

```
class Resource {
public:
    Resource();
    virtual ~Resource();

    void Reference();
    void Unreference();
private:
    friend void Unref(Resource*);

    unsigned refcount;
};

class Color : public Resource { ... };
```

Multiple inheritance

```
class Glyph : public Resource {
public:
    virtual void draw();
};

class InputHandler {
public:
    virtual void handle(const Event&);
};

class Button : public Glyph, public InputHandler {
public:
    virtual void draw();
    virtual void handle(const Event&);
};
```

Ambiguous members

```
class A { public: int f(); };
class B { public: int f(); };
class C : public A, public B { };

C* c;

c->f(); // A::f() or B::f()?

c->A::f();
```

Virtual base classes

```
class A : public Resource { ... };
class B : public Resource { ... };

// now C gets two refcounts
class C : public A, public B { };


// Fix must be applied to A and B!!!
class A : virtual public Resource { ... };
class B : virtual public Resource { ... };
```

Pointers (T*)

- Dynamic
- Allow address arithmetic
- Can be re-assigned

References (T&)

- Fixed
- T's operations
- Single assignment

Rules of thumb

Members should be *pointers*
except when value is fixed at creation

Parameters should be *references*
except when type is always on heap
(Interactor, Font, Color)

Smart pointers

```
class pA {
public:
    A* operator ->();
    operator A*();
private:
    A* _rep;
};

class A {
public:
    int f();
};

pA a;

a->f();
```

Problems with smart pointers

Parallel class derivations

Not compatible with ancestors

Not created by operator new

Useful for iterators

```
class T_Iterator {
public:
    T_Iterator();

    void first();
    boolean more() const;
    void next();

    T* operator ->();
    operator T*();
private:
    T* _cur;
};

for (T_Iterator i; i.more(); i.next()) {
    i->f(i);
};
```

Pointers to member functions

```
typedef void (App::MenuFunc)();

class Action : public MenuItem {
public:
    Action(const char*, Alignment, App&, MenuFunc*);

    virtual void Do();
private:
    App& _app;
    MenuFunc* _func;
};

Action::Action(
    const char* str, Alignment al, App& a, MenuFunc* f
) : MenuItem(str, al), _app(a), _func(f) { }

void Action::Do() {
    (_app.*_func)();
}
```

Ambiguity resolution

- Type conversion
- Constructors
- Copying objects
- Default parameters

Type conversion

```
class MyNumber {
public:
    MyNumber(int);
    MyNumber(float);
    MyNumber(double);

    operator int() const;
    operator float() const;
    operator double() const;

    friend operator +(MyNumber, MyNumber);
};

MyNumber n, m;


n = 1 + m;
```

Solution is to order conversions

T1 -> T2 -> T3 ...

Disallow circularity

Force explicit out-of-order

int MyNumber::int_v() const

Constructors

```
class Semaphore {
public:
    Semaphore(int);

    void wait();
    void signal();
};
```

```
Semaphore mutex(1); mutex = 1;
```

Solutions

- Special init function
- Derive MutexSemaphore
- Protect assignment

Copying objects

Initialization

Assignment

Parameter passing

Function return

Member functions

```
A::A(const A&)
A::A(A&)
A& A::operator =(const A&)
A& A::operator =(A&)
```

If not defined, default is provided

Not inherited

Default parameters

```
class HGlue : public Glue {
public:
    HGlue(Coord natural = 0, Coord stretch = hfil);
    HGlue(
        Coord natural = 0, Coord shrink = hfil,
        Coord stretch = hfil
    );

}; new HGlue(1*inch, 1*inch);
```

Occasionally useful

Really want keyword parameters

C++ class libraries

- Simple
- Extensible
- Evolve over time

Keeping it simple

When in doubt, leave it out

Shallow class hierarchy

Inter-object protocols

Optimize common cases

Extensibility

Make it virtual, not inline

Protected is as open as public

Hide the implementation

Minimize interface dependencies

Use friends for protection

Hiding the implementation

```
class World : public Scene {
public:
    ...
   class WorldRep* rep()  return _rep;  private:
   WorldRep* _rep;
};

#include <InterViews/X11/Xlib.h>

class WorldRep {
public:
   char* hostname();
   Display* display();
    ...
};
```

Class names

Global across application and libraries

Part of linkage identifiers

Short, but not too generic

Can use prefix convention

Should be fixed in language/environment

Collections
(Lists, sets, tables, ...)

Dynamic type information

Templates/generics

Heterogeneous collection

```
class List : public Collection {
public:
    List();
    ~List();

    virtual void append(const Object*);
    virtual void remove(const Object*);
    class ListElement& head() const;
    class ListElement& tail() const;
};

class ListElement : public Object {
public:
    ListElement();
    ~ListElement();

    boolean end() const;
    ListElement& next() const;
    ListElement& prev() const;
    operator Object*();
};
```

Using List

```
List a;
Interactor* i;

a.append(i);

for (ListElement e = a.head(); !e.end(); e = e.next()) {
    Object* o = e;
    i = Narrow(o, class_Interactor);
};
```

Templates (proposed)

```
template<class T> class List {
public:
    virtual void append(const T);
    virtual void remove(const T);
    class ListElement<T>& head() const;
    class ListElement<T>& tail() const;
};

template<class T> class ListElement {
public:
    ListElement<T>& next() const;
    ListElement<T>& prev() const;
    operator T();

};
```

Using templates

```
List<Interactor*> a;
Interactor* i;

a.append(i);

ListElement<Interactor*> e;
for (e = a.head(); !e.end(); e = e.next()) {
    i = e;
};
```

Template power

If file1.c uses List<Interactor*>

And file2.c uses List<Interactor*>

a.out contains only one append method

This makes it harder to implement

Implicit where clause

```
template<class T> class List {
    boolean operator <(const List<T>&);
};

template<class T>
boolean List<T>::operator <(List<T>& list) {
    ListElement<T> e1 = head(), e2 = list.head();
    for (; !e2.end(); e1 = e1.next(), e2 = e2.next()) {
        if (e1.end()) {
            return true;
        }
        T t1 = e1, t2 = e2;
        if (t1 < t2) {
            return true;
        } else if (t1 > t2) {
            break;
        }
    }
    return false;
}

List<int> a, b;
List<Interactor*> c;

if (a < b) { ...
```

Another example

```
template<class T, unsigned size = 10>
class LocalArray {
public:
    LocalArray(unsigned n) {
        if (n < sizeof(_smallarray)) {
            _data = _smallarray;
        } else {
            _data = new char[n];
        }
    }

    ~LocalArray() {
        if (_data != _smallarray) {
            delete _data;
        }
    }

    T& operator[](unsigned);
private:
    T* _data;
    T _smallarray[size];
};

int f(const char* s) {
    LocalArray<char*> a(strlen(s)+1);
     ...
}
```

Using macros

- Define template as 2 macros
- *Name*declare, *Name*implement
- Use declare(name, param) in .h file
- Use implement(name, param) in .c file

Not pretty or convenient

Example: Array<T>

```
#include <generic.h>

#define Array(T) name2(Array_,T)
#define Arraydeclare(T) \
class Array(T) { \
public: \
    Array(T)(); \
 ... \
};

#define Arrayimplement(T) \
Array(T)::Array(T)() { \
 ... \
}
```

Example: I/O Stream library

```
class ios
class istream : virtual public ios
class ostream : virtual public ios
class iostream : public istream, public ostream
```

Using streams

```
class StringBuffer {
public:
    friend class ostream& operator <<(
        ostream&, const StringBuffer&
    );
};

istream& operator >>(
    istream& out, const StringBuffer& s
) {
    ...
}

ostream& operator <<(
    ostream& out, const StringBuffer& s
) {
    ...
};

StringBuffer b;
cin >> b;
cout << b;
```

InterViews (<u>Inter</u>active <u>Views</u>)

- Graphical applications
- Layer above window system
- Natural object-oriented design
- Composition objects
- Drawing framework

Examples

```
#include <InterViews/message.h>
#include <InterViews/world.h>

int main() {
    World* world = new World;
    world->InsertApplication(
        new Message("Hi mom!");
    );
    world->Run();
    delete world;
    return 0;
}
```

```
new VBox(
    new VGlue(space, vfil),
    new HBox(
        new HGlue(space, 0),
        new Message(”Time to pick up Margo and Paul”);
        new HGlue
    ),
    new VGlue(2*space, 2*vfil),
    new HBox(
        new HGlue,
        new PushButton(”OK, OK ...”, status, false),
        new HGlue(space, 0)
    ),
    new VGlue(space)
);
```

InterViews libraries

libInterViews – composition

libgraphic – structured graphics

libUnidraw – drawing framework

libTask – time-sliced tasks

Lessons for C++ libraries

- Complexity is always knocking on the door
- Use indirection for flexibility
  - Virtual functions
  - Access functions to private members
  - Indirect representation
  - Classes versus concrete types (Coord)
- Preserve interfaces when possible
- Use destructors only for deallocation

Application-driven design

- Work from application down
- Generalize solutions, not problems
- Look at power from bottom up
- Allow access to lower layers

Building C++ programs

- Compiling and linking
- Name mangling
- Debugging
- Browsing
- Documenting

Compilation

CC -g -c -I/usr/include/InterViews/Std interactor.c

Suffixes: .c, .C, .cc, .cxx

Produces interactor.o

Automatic -I CC include directory

stream.h, string.h, ...

Automatic -D__cplusplus=1 -Dc_plusplus=1

/lib/cpp -B

Linking with static objects

```
// test.c

class A {
public:
    A();
    ~A();
private:
    int x;
};

A global_a;
```

Special initializer functions for constructing, destructing objects

Link steps

Resolve symbols

Look for initializer functions

Create table of functions

Patch table into a.out

Or relink with table

Have _main call functions in table

Name mangling

Make unique link symbol

void Interactor::Read(Event&);
Read__10InteractorFR5Event

Tools should hide linker names, but ...

Compiler output

Linker error messages

Debugger symbols

Debugging

Line numbers mapped by compiler/translator

Symbols may be mangled

Debugger may not understand inheritance

Use vtbl to identify types

Example

```
% dbx a.out
Reading symbolic information...
Read 2039 symbols
warning: main routine not compiled with the -g option
(dbx) ^Z
Stopped
% nm a.out | grep SquaresView
00005cec T _Redraw__11SquaresViewFiN31
00006090 T _Update__11SquaresViewFv
00005b88 T ___ct__11SquaresViewFP7Squares
00004900 T ___ct__12SquaresFrameFP11SquaresView
00005c58 T ___dt__11SquaresViewFv
0004d8e0 D ___ptbl__11SquaresView_____view_c
0004d860 D ___vtbl__11SquaresView
% fg
dbx a.out
(dbx) stop in Redraw__11SquaresViewFiN31
(2) stop in Redraw__11SquaresViewFiN31
```

```
(dbx) r
Running: a.out
stopped in Redraw__11SquaresViewFiN31 at line 51
in file "../view.c"
   51       output->ClearRect(canvas, x1, y1, x2, y2);
(dbx) t
Redraw__11SquaresViewFiN31(
    __0this = 0x5fd40,
    __0x1 = 0, __0y1 = 0, __0x2 = 199, __0y2 = 199
), line 51 in "../view.c"
SendRedraw__10InteractorFiN41() at 0x259b0
getEvent__11EventReaderFUiR5Event() at 0x25468
gotEvent__11EventReaderFUiR5Event() at 0x2f67c
read__11EventReaderFR5Event() at 0x2f278
run__11EventReaderFv() at 0x2f374
Run__10InteractorFv() at 0xd860
`main() at 0x2c44
```

```
(dbx) p *__0this
*`view`Redraw__11SquaresViewFiN31`__0this = {
    shape__10Interactor       = 0x5fd88
    canvas__10Interactor      = 0x6c9c0
    perspective__10Interactor = (nil)
    xmax__10Interactor        = 199
    ymax__10Interactor        = 199
    input__10Interactor       = (nil)
    output__10Interactor      = 0x5ebf0
    parent__10Interactor      = 0x61b40
    classname__10Interactor   = (nil)
    instance__10Interactor    = (nil)
    toplevel__10Interactor    = (nil)
    left__10Interactor        = 1
    bottom__10Interactor      = 1
    cursensor__10Interactor   = (nil)
    __vptr__10Interactor      = 0x4d860
    subject__11SquaresView    = 0x5fc10
}
(dbx) p &__vtbl__11SquaresView
&__vtbl__11SquaresView = 0x4d860
(dbx) dump
__1h = 541098001
__1w = 134217728
__1y = 202
__1x = 0
__1s = 0xf7fff780
__0y2 = 199
__0x2 = 199
__0y1 = 0
__0x1 = 0
```

Documenting

- Manual page per class
- Replicate inherited methods
- Examples, lots of examples

We want on-line browsing!

Summary

Paying the inheritance tax

Virtuals, constructors, implementation

Linking to other objects

Members and heap types point,
parameters and fixed links reference

Resolving ambiguity

Be explicit with constructors, assignment
Sort direction of type conversions

Stocking libraries is hard work

- Simple to use
  - Focus on specific solutions
- Easy to add functionality
  - Use virtuals plentifully
- Grows and changes over time
  - Make implementation private
- Hang on until templates

Use existing libraries

- I/O streams
- InterViews

Keep in touch with your environment

- Static objects
- Mangled names
- Use debugger however possible
- Use a browser

Conclusions

C++ offers the power

Programmer provides the brains

Implementation/interface code reuse

Think–design–code–repeat

Build experience from others