CS 242

C++

John Mitchell

History

- C++ is an object-oriented extension of C
- C was designed by Dennis Ritchie at Bell Labs
  - used to write Unix
  - based on BCPL
- C++ designed by Bjarne Stroustrup at Bell Labs
  - His original interest at Bell was research on simulation
  - Early extensions to C are based primarily on Simula
  - Called "C with classes" in early 1980's
  - Popularity increased in late 1980's and early 1990's
  - Features were added incrementally

    Classes, templates, exceptions, multiple inheritance, type tests...

Design Goals

- Provide object-oriented features in C-based language, without compromising efficiency
  - Backwards compatibility with C
  - Better static type checking
  - Data abstraction
  - Objects and classes
  - Prefer efficiency of compiled code where possible
- Important principle
  - If you do not use a feature, your compiled code should be as efficient as if the language did not include the feature. (compare to Smalltalk)

How successful?

- Given the design goals and constraints,
  - this is a very well-designed language
- Many users -- tremendous popular success
- However, very complicated design
  - Many specific properties with complex behavior
  - Difficult to predict from basic principles
  - Most serious users chose subset of language
    - Full language is complex and unpredictable
  - Many implementation-dependent properties
  - Language for adventure game fans

Significant constraints

- C has specific machine model
  - Access to underlying architecture
- No garbage collection
  - Consistent with goal of efficiency
  - Need to manage object memory explicitly
- Local variables stored in activation records
  - Objects treated as generalization of structs, so some objects may be allocated on stack
  - Stack/heap difference is visible to programmer

Overview of C++

- Additions and changes not related to objects
  - type bool
  - pass-by-reference
  - user-defined overloading
  - function templates
  - ...

C++ Object System

- Object-oriented features
  - Classes
  - Objects, with dynamic lookup of virtual functions
  - Inheritance
    - Single and multiple inheritance
    - Public and private base classes
  - Subtyping
    - Tied to inheritance mechanism
  - Encapsulation

Some good decisions

- Public, private, protected levels of visibility
  - Public: visible everywhere
  - Protected: within class and subclass declarations
  - Private: visible only in class where declared
- Friend functions and classes
  - Careful attention to visibility and data abstraction
- Allow inheritance without subtyping
  - Better control of subtyping than without private base classes

Some problem areas

- Casts
  - Sometimes no-op, sometimes not (esp multiple inher)
- Lack of garbage collection
  - Memory management is error prone
    - Constructors, destructors are helpful though
- Objects allocated on stack
  - Better efficiency, interaction with exceptions
  - BUT assignment works badly, possible dangling ptrs
- Overloading
  - Too many code selection mechanisms
- Multiple inheritance
  - Efforts at efficiency lead to complicated behavior

Sample class: one-dimen. points

```
class Pt {
  public:
    Pt(int xv);                 } Overloaded constructor
    Pt(Pt* pv);                 }
    int getX();                   Public read access to private data
    virtual void move(int dx);    Virtual function
  protected:
    void setX(int xv);            Protected write access
  private:
    int x;                        Private member data
};
```

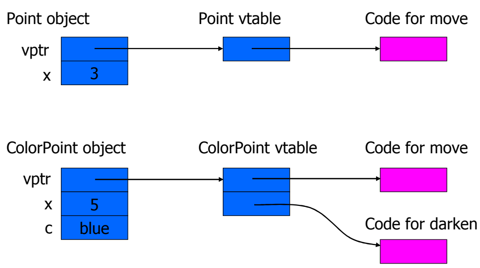
Virtual functions

- Member functions are either
  - Virtual, if explicitly declared or inherited as virtual
  - Non-virtual otherwise
- Virtual functions
  - Accessed by indirection through ptr in object
  - May be redefined in derived (sub) classes
- Non-virtual functions
  - Are called in the usual way. *Just ordinary functions.*
  - Cannot redefine in derived classes (except overloading)
- Pay overhead only if you use virtual functions

Sample derived class

```
class ColorPt: public Pt {         Public base class gives supertype
  public:
    ColorPt(int xv,int cv);        }
    ColorPt(Pt* pv,int cv);        } Overloaded constructor
    ColorPt(ColorPt* cp);          }
    int getColor();                  Non-virtual function
    virtual void move(int dx);     } Virtual functions
    virtual void darken(int tint); }
  protected:
    void setColor(int cv);           Protected write access
  private:
    int color;                       Private member data
};
```

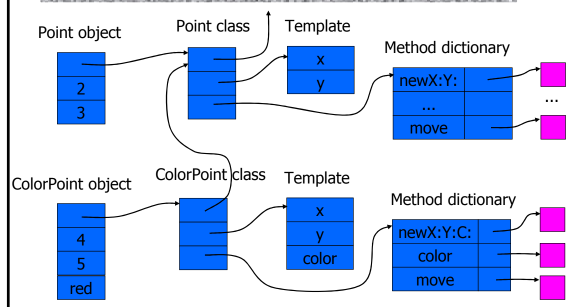
Run-time representation



Data at same offset

Function pointers at same offset

Compare to Smalltalk



Why is C++ lookup simpler?

- Smalltalk has no static type system
  - Code p message:pars could refer to any object
  - Need to find method using pointer from object
  - Different classes will put methods at different place in method dictionary
- C++ type gives compiler some superclass
  - Offset of data, fctn ptr same in subclass and superclass
  - Offset of data and function ptr known at compile time
  - Code p->move(x) compiles to equivalent of

    (*(p->vptr[1]))(p,x) if move is first function in vtable

    data passed to member function; see next slide

Calls to virtual functions

- One member function may call another

```
class A {
    public:
        virtual  int  f (int x);
        virtual  int  g(int y);
};
int A::f(int x) { … g(i) …;}
int A::g(int y) { … f(j) …;}
```

- How does body of f call the right g?
  - If g is redefined in derived class B, then inherited f must call B::g

"This" pointer (analogous to *self* in Smalltalk)

- Code is compiled so that member function takes "object itself" as first argument

  Code `int A::f(int x) { … g(i) …;}`

  compiled as `int A::f(A *this, int x) { … this->g(i) …;}`

- "this" pointer may be used in member function
  - Can be used to return pointer to object itself, pass pointer to object itself to another function, ...

Non-virtual functions

- How is code for non-virtual function found?
- Same way as ordinary "non-member" functions:
  - Compiler generates function code and assigns address
  - Address of code is placed in symbol table
  - At call site, address is taken from symbol table and placed in compiled code
  - *But* some special scoping rules for classes
- Overloading
  - Remember: overloading is resolved at compile time
  - This is different from run-time lookup of virtual function

Scope rules in C++

- Scope qualifiers
  - binary :: operator, ->, and .
  - class::member, ptr->member, object.member
- A name outside a function or class,
  - not prefixed by unary :: and not qualified refers to global object, function, enumerator or type.
- A name after X::, ptr-> or obj.
  - where we assume ptr is pointer to class X and obj is an object of class X
  - refers to a member of class X or a base class of X

Virtual vs Overloaded Functions

```
class parent { public:
    void printclass() {printf("p ");};
    virtual void printvirtual() {printf("p ");};   };
class child : public parent { public:
    void printclass() {printf("c ");};
    virtual void printvirtual() {printf("c ");};   };
main() {
    parent p;  child c; parent *q;
    p.printclass(); p.printvirtual(); c.printclass(); c.printvirtual();
    q = &p;  q->printclass(); q->printvirtual();
    q = &c;  q->printclass(); q->printvirtual();
}
Output:  p  p  c  c  p  p  p  c
```

Subtyping

- Subtyping in principle
  - A <: B if every A object can be used without type error whenever a B object is required
  - Example:

```
Point:       int getX();           } Public members
             void move(int);
ColorPoint:  int getX();
             int getColor();       } Public members
             void move(int);
             void darken(int tint);
```

- C++: A <: B if class A has public base class B
  - This is weaker than necessary Why?

Independent classes not subtypes

```
class Point {                  class ColorPoint {
  public:                        public:
    int getX();                     int getX();
    void move(int);                 void move(int);
  protected:   ...                  int getColor();
  private:     ...                  void darken(int);
};                               protected:   ...
                                 private:     ...
                               };
```

- C++ does not treat ColorPoint <: Point   as written
  - Need public inheritance ColorPoint : public Pt
  - Why??

Why C++ design?

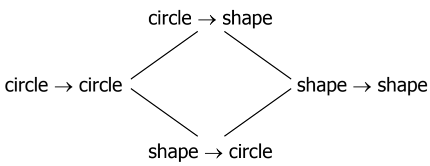
- Client code depends only on public interface
  - In principle, if ColorPoint interface contains Point interface, then any client could use ColorPoint in place of point
  - However -- offset in virtual function table may differ
  - Lose implementation efficiency   (like Smalltalk)
- Without link to inheritance
  - subtyping leads to loss of implementation efficiency
- Also encapsulation issue:
  - Subtyping based on inheritance is preserved under modifications to base class …

Function subtyping

- Subtyping principle
  - A <: B if an A expression can be safely used in any context where a B expression is required
- Subtyping for function results
  - If A <: B, then $C \to A$ <: $C \to B$
- Subtyping for function arguments
  - If A <: B, then $B \to C$ <: $A \to C$
- Terminology
  - Covariance: A <: B implies F(A) <: F(B)
  - Contravariance: A <: B implies F(B) <: F(A)

Examples

◆ If circle <: shape, then



C++ compilers recognize limited forms of function subtyping

Subtyping with functions

```
class Point {
  public:
    int getX();
    virtual Point *move(int);
  protected:  ...
  private:    ...
};
```

```
class ColorPoint: public Point {
  public:
    int getX();
    int getColor();
    ColorPoint * move(int);
    void darken(int);
  protected:  ...
  private:    ...
};
```

Inherited, but repeated here for clarity

◆ In principle: can have ColorPoint <: Point

◆ In practice: some compilers allow, others have not

This is covariant case; contravariance is another story

Details, details

◆ This is legal

```
class Point { …
    virtual Point * move(int);
 …  }
class ColorPoint: public Point {  …
    virtual ColorPoint * move(int);
 …  }
```

◆ But not legal if *'s are removed

```
class Point { …   virtual Point move(int); … }
class ColorPoint: public Point { …virtual ColorPoint move(int);… }
```

Related to subtyping distinctions for object L-values and object R-values
(Non-pointer return type is treated like an L-value for some reason)

Subtyping and Object L,R-Values

◆ If class B : public A { … }

Then

- B r-value <: A r-value
  - If x = a is OK, then x = b is OK
    provided A's operator = is public
  - If f(a) is OK, then f(b) is OK
    provided A's copy constructor is public
- B l-value ~~<:~~ A l-value
- B* <: A* on last slide because cannot assign to return value
- B** ~~<:~~ A**

Generally, X <: Y → X* <: Y* is unsound.

Review

◆ Why C++ requires inheritance for subtyping

- Need virtual function table to look the same
- This includes private and protected members
- Subtyping w/o inheritance weakens data abstraction

(This is my post facto explanation; I don't know what designers think.)

◆ Possible confusion regarding inlining

- Cannot generally inline virtual functions
- Inlining is possible for nonvirtual functions
  - These are available in C++
  - Not in Smalltalk since every lookup is through class

Inlining is very significant for efficiency; enables further optimization.

Abstract Classes

◆ Abstract class:

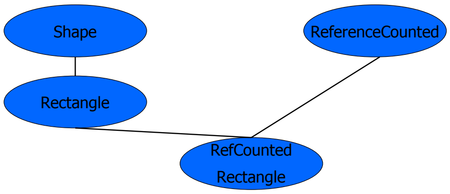
- A class without complete implementation
- Declare by =0 (what a great syntax!)
- Useful because it can have derived classes

  Since subtyping follows inheritance in C++, use abstract classes to build subtype hierarchies.
- Establishes layout of virtual function table (vtable)

◆ Example

- Geometry classes in appendix of reader
  - Shape is abstract supertype of circle, rectangle, ...

Multiple Inheritance



Inherit independent functionality from independent classes

Problem: Name Clashes

```
class A {
    public:
        void virtual f() { … }
};
class B {
    public:
        void virtual f() { … }
};
class C : public A, public B { … };
…
    C* p;
    p->f();     // error
```

same name in 2 base classes

Possible solutions to name clash

- Three general approaches
  - Implicit resolution
    - Language resolves name conflicts with arbitrary rule
  - Explicit resolution
    - Programmer must explicitly resolve name conflicts
  - Disallow name clashes
    - Programs are not allowed to contain name clashes
- No solution is always best
- C++ uses explicit resolution

Repair to previous example

- Rewrite class C to call A::f explicitly

```
class C : public A, public B {
    public:
        void virtual f( ) {
                A::f( );    // Call A::f(), not B::f();
        }
```

- Reasonable solution
  - This eliminates ambiguity
  - Preserves dependence on A
    - Changes to A::f will change C::f

vtable for Multiple Inheritance

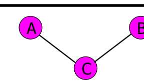
```
class A {
    public:
        int x;
        virtual void f();
};
class B {
    public:
        int y;
        virtual void g();
        virtual void f();
};
```

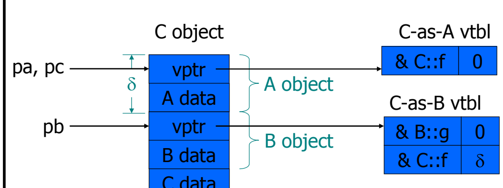
```
class C: public A, public B {
    public:
        int z;
        virtual void f();
};

    C *pc = new C;
    B *pb = pc;
    A *pa = pc;
```

Three pointers to same object, but different static types.

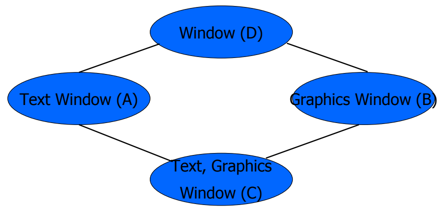
Object and classes





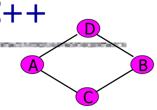
- Offset δ in vtbl is used in call to pb->f, since C::f may refer to A data that is above the pointer pb
- Call to pc->g can proceed through C-as-B vtbl

Multiple Inheritance "Diamond"

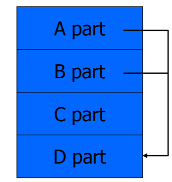


- Is interface or implementation inherited twice?
- What if definitions conflict?

Diamond inheritance in C++



- Standard base classes
  - D members appear twice in C
- Virtual base classes
  - class A : public virtual D { … }
  - Avoid duplication of base class members
  - Require additional pointers so that D part of A, B parts of object can be shared



- C++ multiple inheritance is complicated in part because of desire to maintain efficient lookup

C++ Summary

- Objects
  - Created by classes
  - Contain member data and pointer to class
- Classes: virtual function table
- Inheritance
  - Public and private base classes, multiple inheritance
- Subtyping: Occurs with public base classes only
- Encapsulation
  - member can be declared public, private, protected
  - object initialization partly enforced