

CS560

LECTURE 8
March 3, 2005

Slides: From John Mitchell, Concepts in Programming Languages, 2003.

Announcements

- ◆ Homework 3 is due.
- ◆ Homework 4 is posted.
- ◆ Term project due next week.
 - [What needs to be turned in?](#)
- ◆ Any questions?

Overview of Class Tonight

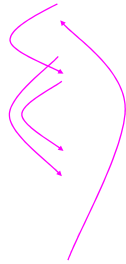
- ◆ Control
 - [Exceptions](#)
- ◆ Object-oriented programming
 - [Primary object-oriented language concepts](#)
 - dynamic lookup
 - encapsulation
 - Inheritance
 - subtyping
 - [C++](#)

Topics → Chapter 8

- ◆ Structured Programming
 - [Go to considered harmful](#)
- ◆ Exceptions
 - [“structured” jumps that may return a value](#)
 - [dynamic scoping of exception handler](#)

Fortran Control Structure

```
10 IF (X .GT. 0.000001) GO TO 20
11 X = -X
   IF (X .LT. 0.000001) GO TO 50
20 IF (X*Y .LT. 0.000001) GO TO 30
   X = X-Y-Y
30 X = X+Y
   ...
50 CONTINUE
   X = A
   Y = B-A
   GO TO 11
   ...
```



Similar structure may occur in assembly code

GO TO Statement Considered Harmful --Dijkstra

For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of **go to** statements in the programs they produce. More recently I discovered why the use of the **go to** statement has such disastrous effects, and I became convinced that the **go to** statement should be abolished from all "higher level" programming languages (i.e. everything except, perhaps, plain machine code). At that time I did not attach too much importance to this discovery; I now submit my considerations for publication because in very recent discussions in which the subject turned up, I have been urged to do so.

Historical Debate

- ◆ Dijkstra, Go To Statement Considered Harmful
 - Letter to Editor, *C ACM*, March 1968
 - Google: paper title
- ◆ Knuth, Structured Prog. with go to Statements
 - You can use goto, but do so in structured way ...
- ◆ General questions
 - Do syntactic rules force good programming style?
 - Can they help?

Advance in Computer Science

- ◆ Standard constructs that structure jumps
 - if ... then ... else ... end
 - while ... do ... end
 - for ... { ... }
 - case ...
- ◆ Modern style
 - Group code in logical blocks
 - Avoid explicit jumps except for function return
 - Cannot jump *into* middle of block or function body

Exceptions: Structured Exit

◆ Terminate part of computation

- Jump out of construct
- Pass data as part of jump
- Return to most recent site set up to handle exception
- Unnecessary activation records may be deallocated
 - May need to free heap space, other resources

◆ Two main language constructs

- Declaration to establish exception *handler*
- Statement or expression to *raise* or *throw* exception

Often used for unusual or exceptional condition, but not necessarily

ML Example

```
exception Determinant; (* declare exception name *)
fun invert (M) =      (* function to invert matrix *)
  ...
  if ...
    then raise Determinant (* exit if Det=0 *)
    else ...
end;
...
invert (myMatrix) handle Determinant => ... ;
```

Value for expression if determinant of myMatrix is 0

C++ Example

```
Matrix invert(Matrix m) {
  if ... throw Determinant;
  ...
};

try { ... invert(myMatrix); ...
}
catch (Determinant) { ...
  // recover from error
}
```

C++ vs ML Exceptions

◆ C++ exceptions

- Can throw any type
- Stroustrup: "I prefer to define types with no other purpose than exception handling. This minimizes confusion about their purpose. In particular, I never use a built-in type, such as int, as an exception." -- The C++ Programming Language, 3rd ed.

◆ ML exceptions

- Exceptions are a different kind of entity than types.
- Declare exceptions before use

Similar, but ML requires the recommended C++ style.

ML Exceptions

◆ Declaration

`exception (name) of (type)`

gives name of exception and type of data passed when raised

◆ Raise

`raise (name) (parameters)`

expression form to raise an exception and pass data

◆ Handler

`(exp1) handle (pattern) => (exp2)`

evaluate first expression

if exception that matches pattern is raised,

then evaluate second expression instead

General form allows multiple patterns.

Which handler is used?

```
exception Ovfllw;  
fun reciprocal(x) =  
  if x < min then raise Ovfllw else 1/x;  
(reciprocal(x) handle Ovfllw => 0) / (reciprocal(y) handle Ovfllw => 1);
```

◆ Dynamic scoping of handlers

- First call handles exception one way
 - Second call handles exception another
 - General dynamic scoping rule
- Jump to most recently established handler on run-time stack

◆ Dynamic scoping is not an accident

- User knows how to handle error
- Author of library function does not

Exception for Error Condition

- datatype 'a tree = LF of 'a | ND of ('a tree)*('a tree)

- exception No_Subtree;

- fun lsub (LF x) = raise No_Subtree

 | lsub (ND(x,y)) = x;

> val lsub = fn : 'a tree -> 'a tree

- This function raises an exception when there is no reasonable value to return
- We'll look at typing later.

Exception for Efficiency

◆ Function to multiply values of tree leaves

`fun prod(LF x) = x`

 | `prod(ND(x,y)) = prod(x) * prod(y);`

◆ Optimize using exception

`fun prod(tree) =`

`let exception Zero`

`fun p(LF x) = if x=0 then (raise Zero) else x`

 | `p(ND(x,y)) = p(x) * p(y)`

`in`

`p(tree) handle Zero => 0`

`end;`

Dynamic Scope of Handler

```

exception X;
  (let fun f(y) = raise X
    and g(h) = h(1) handle X => 2
  in
    g(f) handle X => 4
  end) handle X => 6;
  
```

scope

handler

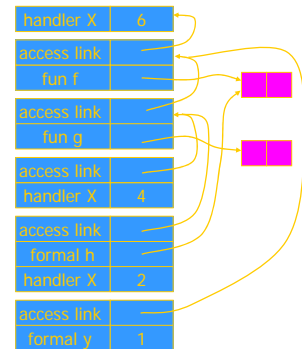
Which handler is used?

Dynamic Scope of Handler

```

exception X;
(let fun f(y) = raise X
  and g(h) = h(1) handle X => 2
in
  g(f) handle X => 4
end) handle X => 6;
  
```

Dynamic scope:
find first X handler,
going up the
dynamic call chain
leading to raise X.



Compare to static scope of variables

```

exception X;
  (let fun f(y) = raise X
    and g(h) = h(1) handle X => 2
  in
    g(f) handle X => 4
  end) handle X => 6;
  
```

```

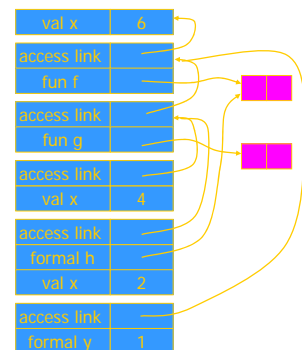
val x=6;
  (let fun f(y) = x
    and g(h) = let val x=2 in
      h(1)
    in
      let val x=4 in g(f)
    end);
  
```

Static Scope of Declarations

```

val x=6;
(let fun f(y) = x
  and g(h) = let val x=2 in
    h(1)
  in
    let val x=4 in g(f)
  end);
  
```

Static scope: find
first x, following
access links from
the reference to X.



Typing of Exceptions

◆ Typing of raise $\langle \text{exn} \rangle$

- Recall definition of typing
 - Expression e has type t if normal termination of e produces value of type t
- Raising exception is not normal termination
 - Example: $1 + \text{raise } X$

◆ Typing of handle $\langle \text{exn} \rangle \Rightarrow \langle \text{value} \rangle$

- Converts exception to normal termination
- Need type agreement
- Examples
 - $1 + ((\text{raise } X) \text{ handle } X \Rightarrow e)$
 - $1 + (e_1 \text{ handle } X \Rightarrow e_2)$

Exceptions and Resource Allocation

```
exception X;
(let
  val x = ref [1,2,3]
in
  let
    val y = ref [4,5,6]
  in
    ... raise X
  end
end); handle X => ...
```

◆ Resources may be allocated between handler and raise

◆ May be “garbage” after exception

◆ Examples

- Memory
- Lock on database
- Threads
- ...

General problem: no obvious solution

Questions

Exception Propagation

```
class CTest
{
public:
  char *ShowReason() { return "Something really bad happened."; }
};

class CDemo
{
public:
  CDemo() { cout << "Constructing CDemo." << endl; }
  ~CDemo() { cout << "Destructing CDemo." << endl; }
};

void MyFunc()
{
  CDemo D;
  cout << "In MyFunc(). Throwing CTest exception." << endl;
  throw CTest();
}
```

Exception Propagation

```

void main()
{
    cout << "In main." << endl;
    try {
        cout << "In try block, calling MyFunc()."
        << endl;
        MyFunc();
    }
    catch( CTest E ) {
        cout << "In catch handler." << endl;
        cout << "Caught CTest exception type:
        ";
        cout << E.ShowReason() << endl;
    }
    catch(...) {
        cout << "Caught some other exception."
        << endl;
    }
    cout << "Back in main. Execution resumes
    here." << endl;
}
    
```

Program output:

```

In main.
In try block, calling MyFunc().
Constructing CDemo.
In MyFunc(). Throwing CTest exception.
Destructing CDemo.
In catch handler.
Caught CTest exception type: Something
really bad happened.
Back in main. Execution resumes here.
    
```

Exercise

exception Excpt of int;

```

fun twice(f,x) = f(f(x)) handle Excpt(x) => x;
fun pred(x) = if x = 0 then raise Excpt(x) else x-1;
fun dumb(x) = raise Excpt(x);
fun smart(x) = 1 + pred(x) handle Excpt(x) => 1;
    
```

```

twice(pred, 1);
twice(dumb, 1);
twice(smart ,0);
    
```

What is the result and which exception gets raised and where?

Outline of lecture

- ◆ Object-oriented design
- ◆ Primary object-oriented language concepts
 - dynamic lookup
 - encapsulation
 - inheritance
 - subtyping
- ◆ Program organization
 - Work queue, geometry program, design patterns
- ◆ Comparison
 - Objects as closures?

Objects

◆ An object consists of

- **hidden data**
instance variables, also called member data
hidden functions also possible
- **public operations**
methods or member functions
can also have public variables in some languages

hidden data	
msg ₁	method ₁
...	...
msg _n	method _n

◆ Object-oriented program:

- Send messages to objects

Object-Orientation

- ◆ Programming methodology
 - organize concepts into objects and classes
 - build extensible systems
- ◆ Language concepts
 - dynamic lookup
 - encapsulation
 - subtyping allows extensions of concepts
 - inheritance allows reuse of implementation

Dynamic Lookup

- ◆ In object-oriented programming,
object → message (arguments)
code depends on object and message
- ◆ In conventional programming,
operation (operands)
meaning of operation is always the same

Example

- ◆ Add two numbers $x \rightarrow \text{add}(y)$
different `add` if `x` is integer, complex
- ◆ Conventional programming `add(x, y)`
function `add` has fixed meaning

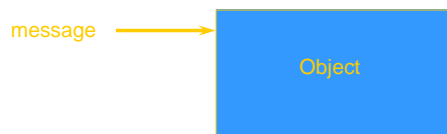
Very important distinction:
Overloading is resolved at compile time,
Dynamic lookup at run time

Language concepts

- ◆ "dynamic lookup"
 - different code for different object
 - integer "+" different from real "+"
- ◆ encapsulation
- ◆ subtyping
- ◆ inheritance

Encapsulation

- ◆ Builder of a concept has detailed view
- ◆ User of a concept has "abstract" view
- ◆ Encapsulation is the mechanism for separating these two views



Comparison

- ◆ Traditional approach to encapsulation is through abstract data types
- ◆ Advantage
 - Separate interface from implementation
- ◆ Disadvantage
 - Not extensible in the way that OOP is

We will look at ADT's example to see what problem is

Language concepts

- ◆ "dynamic lookup"
 - different code for different object
 - integer "+" different from real "+"
- ◆ encapsulation
- ◆ subtyping
- ◆ inheritance

Subtyping and Inheritance

- ◆ Interface
 - The external view of an object
- ◆ Subtyping
 - Relation between interfaces
- ◆ Implementation
 - The internal representation of an object
- ◆ Inheritance
 - Relation between implementations

Object Interfaces

- ◆ Interface
 - The messages understood by an object
- ◆ Example: point
 - x-coord : returns x-coordinate of a point
 - y-coord : returns y-coordinate of a point
 - move : method for changing location
- ◆ The interface of an object is its *type*.

Subtyping

- ◆ If interface contains all of interface , then objects can also be used objects.

Point	Colored_point
x-coord	x-coord
y-coord	y-coord
move	color
	move
	change_color

- ◆ Colored_point interface contains Point
 - Colored_point is a *subtype* of Point

Inheritance

- ◆ Implementation mechanism
- ◆ New objects may be defined by reusing implementations of other objects

Example

```
class Point
  private
    float x, y
  public
    point move (float dx, float dy);

class Colored_point
  private
    float x, y; color c
  public
    point move(float dx, float dy);
    point change_color(color newc);
```

◆ Subtyping

- Colored points can be used in place of points
- Property used by client program

◆ Inheritance

- Colored points can be implemented by reusing point implementation
- Property used by implementor of classes

Language concepts

- ◆ “dynamic lookup”
 - different code for different object
 - integer “+” different from real “+”
 - ◆ encapsulation
 - ◆ subtyping
 - ◆ inheritance
- ◆ How are these features implemented in C++?

CS 560

C++

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 members
 - Are 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:
    ColorPt(int xv,int cv);
    ColorPt(Pt* pv,int cv);
    ColorPt(ColorPt* cp);
    int getColor();
    virtual void move(int dx);
    virtual void darken(int tint);
protected:
    void setColor(int cv);
private:
    int color;
};
```

Public base class gives supertype

Overloaded constructor

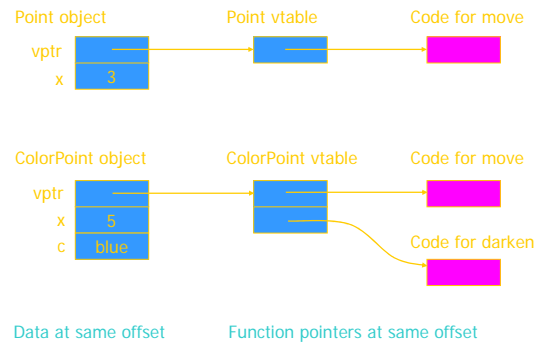
Non-virtual function

Virtual functions

Protected write access

Private member data

Run-time representation



Why is C++ lookup simpler?

- ◆ Smalltalk has no static type system
 - Code `p message:args` 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 fctn 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() ...; }
int A::g(int y) { ... f() ...; }
```
- ◆ 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

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

```
Code      int A::f(int x) { ... g(l) ...;}
compiled as int A::f(A *this, int x) { ... this->g(l) ...;}
```

- ◆ "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();
}
```

Exercise 12.1 book