CS202 Compiler Construction

April 22, 2003

Assignment 12
Written problems
Relevant topics not thoroughly covered in programming assignments

What's the difference between Pond compiler and a “real” compiler?

Advanced topics: language-specific platforms (OO)

What's different from lakec and a “real” compiler?

Testing and reliability
  Very important (and very boring)
Command line wrapper
  Calls preprocessor, assembler (as), linker
Runtime library
  e.g. malloc/free, garbage collector, array bounds checking, dynamic type checks
Driver program
Processes command line arguments
  Checks for legality
Calls tools
  preprocessor (cpp)
  compiler (ccom)
  assembler (as)
  linker (ld)

Example:
```
cc -o prog -g x.c -includes y.c
  cpp x.c > /tmp/xxx1
  ccom -g /tmp/xxx1 /tmp/xxx2
  rm /tmp/xxx1
  as /tmp/xxx2 -o x.o
  rm /tmp/xxx2
  ccom -g /tmp/xxx3 /tmp/xxx4
  rm /tmp/xxx3
  as /tmp/xxx4 -o y.o
  rm /tmp/xxx4
  ld -o prog /lib/crt0.o x.o y.o /lib/libc.a /lib/crt1.o
```

Preprocessor handles directives:
  #include
  #define
  #if, #ifdef, #else, ...
Simple text processing
  No understanding of C,C++,lake
Mostly tokenizer
Simple parsing
  Only lines starting with #
Assembler translates assembly code:

``` assembler
mov %i0, %i2
``` into binary machine code

Not everything known at as time
- Starting address of code, data
- Address of external names

Format contains relocation info:
- Mostly elf format now
- List locations where ext. addr. must be resolved

Linker combines relocatable binaries:
- Produced by as
- Pulls out pieces from .o files
- Combines program code/data
- Resolves unresolved names
- Updates locations in code/data

More things on ld command line:
- `$/lib/crt0.o`
- `$/lib/crt1.o`
- `$/lib/libc.a`
- `crt0,crt1`
  - Startup, cleanup code
  - main is NOT entry point; _enter is
- `libc.a`
  - Standard library (e.g. basic math functions)
  - Runtime library (malloc, printf, etc.)
For most platforms, linker
   Loads everything from .o
   Loads only what's referenced from .a
May need to make multiple passes
   Over list of libraries
   If libraries are mutually recursive
May be linker's or user's responsibility
   Depends on platform
   Usually user's job to list libs twice

Libraries basically contain multiple .o's
   May be literally true
   May be identical format to single .o
Distinction for linker
   Load by .o
   Load by function
Library always has:
   Table of contents
   Relocation info

Source level debugging
   Expected piece of language tool set

Debuggers let the developer
   Single step a source statement at a time
   Set breakpoints at source statements
   Evaluate expressions
   Assign values to variables
   Call functions
   Explore type errors (recent research)
Supporting OO

C++ has its own difficulties
   Tougher parsing
   More context sensitive lexing
   Multiply nested name spaces

Same as what we did
   Just harder

C++ also supports common OO features
   1. Encapsulation (e.g. private fields)
   2. Operator overloading
   3. Inheritance
   4. Method dispatching

1 and 2 supported by typechecking
3 and 4 supported by IR translation

Quick review of inheritance
   What does compiler need to support

Class types arranged in hierarchy
   Subclass inherits from superclass
   All instances of subclass are also instances of superclass
   If believed to be superclass, only superclass operations allowed
   Even if actually instance of subclass
   But uses subclass implementation
Compiling inheritance: three major issues
1. typechecking
2. data layout
3. method dispatching

Multiple inheritance complicates type analysis, data layout
Start with only single inheritance
NB: Java punt on multiple inheritance altogether!

Type analysis for OO programs very advanced topic

Uses subtyping:
Rough approximation: if A subclass of B, then A subtype of B
If o : A, then can be used as o : B

Many issues still being dealt with in research community
C++ uses simple analysis

Data layout is simple
Assume B is subclass of A

Layout of instances of B must look:
like instance of A when dereferencing A* pointer
like instance of B when dereferencing B* pointer

Make Bs start with A fields
B fields come after
Consider C++ code

```cpp
class A { public: int a1; int a2; }

A* pointer expects to point to

```

Consider C++ code

```cpp
class A { public: int a1; int a2; }
class B : public A { public: int b1; int a2; }

B* pointer expects to point to

```

Multiple inheritance is messier

```cpp
class A { public: int a1; int a2; }
class B : public A { public: int b1; int b2; }
class C { public: int c1; int c2; }

A* points to a1, C* points to c1

```

What does B* point to?
C++ solution
B* points to different mem than C*

Because B* points to different memory location than C*
Assignment no longer simple

IR for c = b was
MOVE c,b
IR now becomes
MOVE c,b,+8

Many issues now arise
Need to add offset for casts
b = (B*)c becomes MOVE b,c,-8
All +/- must be conditional
cannot do if NULL
Adds lots more branches to code
More problems with virtual functions
Details coming
Also problems with ==, hashing, ...

IR for c = b was
MOVE c,b
IR now becomes
MOVE c,b,+8

Many issues now arise
Need to add offset for casts
b = (B*)c becomes MOVE b,c,-8
All +/- must be conditional
cannot do if NULL
Adds lots more branches to code
More problems with virtual functions
Details coming
Also problems with ==, hashing, ...

Because B* points to different memory location than C*
Assignment no longer simple

IR for c = b was
MOVE c,b
IR now becomes
MOVE c,b,+8

Many issues now arise
Need to add offset for casts
b = (B*)c becomes MOVE b,c,-8
All +/- must be conditional
cannot do if NULL
Adds lots more branches to code
More problems with virtual functions
Details coming
Also problems with ==, hashing, ...
Java avoids all this with interfaces

Data only singly inherited
No equivalent layout problem

Class may implement multiple interfaces
Similar design patterns possible

Still has virtual function issues

"Diamonds" introduce more issues

```
class A { public: int a1; };
class B : public A { public: int b1; };
class C : public A { public: int c1; };
class D : public B, public C { public: int d1; };
```

How many copies of a1 in D?

Default C++ behavior is 2 copies
one from B
one from C
C++ virtual inheritance changes this
provides indirection to superclass data

```cpp
class A { public: int a1; }
class B : virtual public A  
  { public: int b1;         }
class C : virtual public A  
  { public: int c1;         }
class D : public B, public C
  { public: int d1;         }
```

```
B* b
  ^   ^
  b4  b8
  Aptr
  ^   ^
  b4  b8
  a1
  ^   ^
  b
```

b->b1 becomes MEM +(b,8)
However b->a1 not MEM +(b,0)

Instead access through Aptr
b->a1 becomes MEM +(MEM +(b,4),0)

```
B* b
  ^   ^
  b4  b8
  Aptr
  ^   ^
  b4  b8
  a1
  ^   ^
  b
```

Multiple subclasses now share a1

```
C* c
  ^   ^
  c4  c8
  A* Aptr
  ^   ^
  c4  c8
  b1
  ^   ^
  c4  c8
  A* Aptr
  ^   ^
  c4  c8
  a1
  ^   ^
  c4  c8
  b
```
Compiler must insert code in beginning of constructors for B,C,D to initialize these pointers.

Must be disabled for B,C if already done for D.

Constructors have extra (hidden) param to enable/disable ptr inits.

Dynamic method dispatching
Virtual functions in C++ lingo
Select most appropriate implementation

When accessing instance of subclass through superclass pointer, still get subclass function implementation.

Compiler cannot know actual class
How to find code at runtime?

C++ uses dispatch tables
As do most other OO languages
Pointed to by hidden field
May be first in object
Last in object
“Before” object…
Compiler dependent
Dispatch table has type:
void (*vtab[])();
(it's an array of function pointers)
Each virtual function has one slot

For single inheritance:
Single table pointer for all classes
Single table for all instances of class

Consider C++ code
```cpp
class A { public: int a1;
  virtual int fa1(void) {}
  virtual int fa2(void) {};
};
class B : public A { public: int b1;
  virtual int fa1(void) {};
  virtual int fb(void) {};
};
```
Generates following layout

Calling dispatched functions:
- Gets pointer to table
- Gets function pointer from table
- Casts function type
- Instantiates implicit this pointer
- Makes call

This instantiated for all object methods
- Not just dispatched calls
- Value is pointer to object itself
Given C++ classes

```cpp
class A { public: int a1;
    virtual int fa1(void);
    virtual int fa2(void);
}
class B : public A { public: int b1;
    virtual int fb(void);
    virtual int fba(void);
};
```

B * b; b->fa1();

Becomes

```
(*((int*)(B*))((b->vtab[0])))(b)
```

Multiple inheritance confuses each superclass has own vtab ptr

One more issue here
Consider call to b->fc2()
Standard conversion:

```
(*((int*)(B*))((b->c_vtab[0])))(b)
```

But b is a B*, not C*
Need to convert this

```
(*((int*)(B*))((b->c_vtab[0])))((C*)b)
```

Or finally

```
(*((int*)(B*))((b->c_vtab[0])))(b+b)
```