Compiling OO Languages

CSE413
Autumn 2007

Agenda for Today
- Object representation and layout
- Field access
- What is this?
- Object creation - new
- Method calls
  - Dynamic dispatch
  - Method tables
  - Super
  - Runtime type information

Notes: 1) The student class inherits from (is a subclass of) the Person class
2) Both classes have their own implementation of print_mailing_label();

C++ Example

```cpp
Person aPerson;
Student aStudent;
// Static binding done at compile time
// Function called depends on the declared type
// of the variable used:
aPerson.print_mailing_label();
aStudent.print_mailing_label();

Person *p_ptr;
p_ptr = &aPerson; // OR p_ptr = &aStudent;
// Dynamic binding done at runtime
// Function called depends on what type the object
// p_ptr points to is, could be a Person or a Student
p_ptr->print_mailing_label();
```

Dynamic vs. Static binding

- **Static method binding:**
  - The compiler can figure out at compile time what function to call.
  - C++ has this by default, can designate functions as virtual to get dynamic binding.

- **Dynamic method binding:**
  - Compiler must instead generate code that will figure out **at run-time** what function to call.
  - Java uses dynamic binding for functions by default.

Object Representation

- The naïve explanation is that an object contains:
  - **Fields** declared in its class and in all superclasses
  - **Methods** declared in its class and in all superclasses
- When a method is called, the method inside that particular object is called.
- But we don’t want to really implement it this way—we only want **one** copy of each method’s code

Actual representation

- Each object contains:
  - An entry for each field (variable)
  - A pointer to a runtime data structure “describing the class”
- Key component: method dispatch table
Method Dispatch Tables

- Often known as “vtables”
- One pointer per method
- Offsets fixed at compile time
- One instance of this per class, (not per object)

Member Lookup - vtable

Method Tables and Inheritance

- One possible simple implementation:
  - Method table for extended class has pointers to methods declared in it
  - Method table also contains a pointer to parent class method table
  - Method dispatch
    - Look in current table and use it if there
    - Look in parent class table if not there
    - Repeat until found
  - Actually used in some dynamic systems (e.g. SmallTalk, etc.)

O(1) Method Dispatch

- Better Idea: First part of method table for extended class has pointers in same order as parent class
  - BUT pointers actually refer to overriding methods if these exist
  - ∴ Method dispatch is indirect using fixed offsets known at compile time – O(1)
  - In C: *(object->vtbl[offset])(parameters)
  - Pointers to additional methods in extended class are included in the table following inherited/overridden ones

Single Inheritance
Single Inheritance

```java
class Bird {
    private:
        int age;
        double weight;
        char favorite_letter;
    public:
        void eat();
        virtual int sleep();
        virtual void speak();
    }
}
class Eagle : public Bird {
    private:
        int zip_code;
    public:
        virtual void speak(); // overrides Bird version
        virtual double findfish();
        void look_important();
        virtual void buildnest();
    }
Bird B;
Eagle E;
```

Method Dispatch Footnotes

- Still want pointer to parent class method table for other purposes
- Casts and instanceof

What if we had to generate code for objects in a language like Java??

- Need to explore
  - Object layout in memory
  - Compiling field references
    - Implicit and explicit use of “this”
  - Representation of vtables
  - Object creation – new
  - Code for dynamic dispatch
    - Including implementing “super.f”
  - Runtime type information – instanceof and casts

Object Layout

- Typically, allocate fields sequentially
- Follow processor/OS alignment conventions when appropriate
- Use first 32 bits of object for pointer to method table/class information
- Objects are allocated on the heap
- No actual representation in the generated code

Local Variable Field Access

- Source
  ```
  int n = obj.field;
  ```
- X86
  ```
  (Assuming that obj is a local variable in the current method):
  mov eax, [ebp + offset_obj]
  mov eax, [eax + offset_field]
  mov [eax + offset_field], eax
  ```
Example: Local Variable Field Access

- Source
  ```
  Bird b_ptr = new Eagle();
  int n = b_ptr.age;
  ```
- X86
  ```
  mov eax, [ebp + offset_b_tr] ; get address of object
  mov eax, [eax + offset_age] ; get value of field
  mov [eax + offset_n], eax ; store result in n
  ```

Local Fields

- A method can refer to fields in the receiving object either explicitly as “this.f” or implicitly as “f”
  - Both compile to the same code – an implicit “this.” is assumed if not present
- **Mechanism**: a reference to the current object is an *implicit* parameter to every method
  - Can be in a register or on the stack

Implementing the *this* pointer

- When you write:
  ```
  void setIt(int it) {
  this.it = it;
  }
  ... obj.setIt(42);
  ```
- You really get:
  ```
  void setIt(ObjectType this, int it) {
  this.it = it;
  }
  ... setIt(obj, 42);
  ```

x86 Conventions (C++)

- ecx is traditionally used as “this”
- Add to method call
  ```
  mov ecx, receivingObject ; ptr to object
  ```
- Do this after arguments are evaluated and pushed, right before dynamic dispatch code (more about that to come)

x86 Local Field Access

- Source
  ```
  int n = fld; or int n = this.fld;
  ```
- X86: ???
class foo {
    int a;
    double b;
    char c;
public:
    virtual void k();
    virtual int l();
    virtual void m();
    virtual double n();
    ...}

class bar : public foo {
    int w;
public:
    void m(); // overrides version in foo
    virtual double s();
    virtual char *t();
} B;

Method Tables For Foo and Bar

class foo {
    ... foo$$ dd 0 ; no superclass
    virtual void k();
    virtual int l();
    virtual void m();
    virtual double n();
    }
class bar : public foo {
    ... bar$$ dd foo$$ ; parent
    virtual double s();
    virtual char *t();
} B;

Method Table Footnotes

- **Key point:** First four method entries in bar’s method table are pointers to methods declared in foo in exactly the same order.
  ∴ Compiler knows correct offset for a particular method regardless of whether that method is overridden.

Object Creation – new

- Steps needed
  - Call storage manager (malloc or similar) to get the raw bits
  - Store pointer to method table in the first 4 bytes of the object
  - Call a constructor (pointer to new object, this, in ecx)
  - Result of new is pointer to the constructed object

Method Calls

- Steps needed
  - Push arguments as usual
  - Put pointer to object in ecx (new this)
  - Get pointer to method table from first 4 bytes of object
  - Jump indirectly through method table

Method Call

- Source
  - obj.meth(...);
- X86
  - <push arguments from right to left> ; (if needed)
  - mov ecx, [ebp+offset]
  - mov eax, [ecx]
  - call [eax+offset]
  - call indirect via method tbl
  - <pop arguments> ; (if needed)
Runtime Type Checking

- Use the method table for the class as a "runtime representation" of the class
- The test for "o instanceof C" is:
  - Recursively, get the superclass's method table pointer from the method table and check that
  - Stop when you reach Object (or a null pointer, depending on how you represent things)
  - If no match when you reach the top of the chain, result is "false"

More Code Generation

Other Control Flow: switch

- Naive: generate a chain of nested if-else if statements
- Better: switch is designed to allow an O(1) selection, provided the set of switch values is reasonably compact
- Idea: create a 1-D array of jumps or labels and use the switch expression to select the right one
  - Need to generate the equivalent of an if statement to ensure that expression value is within bounds

Switch

Source

```java
switch (exp) {
  case 0: stmts0;
  case 1: stmts1;
  case 2: stmts2;
}
```

x86

```x86
<put exp in eax>
"if (eax < 0 || eax > 2)
jmp defaultLabel"
mov eax, swtab[eax*4]
jmp eax
.data
swtab dd L0

.code
L0: <stmts0>
```

Arrays

- Several variations
- C/C++/Java
  - 0-origin; an array with n elements contains variables a[0]...a[n-1]
  - 1 or more dimensions; row major order
- Key step is to evaluate a subscript expression and calculate the location of the corresponding element

0-Origin 1-D Integer Arrays

Source

```java
exp1[exp2]
```

x86