DESIGN ISSUES FOR OBJECT-ORIENTED LANGUAGES

The following are the design issues of Object oriented languages

- Exclusivity of objects
- Type checking of objects used to invoke dynamically bound methods
- Single versus multiple inheritance
- Allocation of objects
- Deallocation of objects
- Dynamic binding of messages to methods

Exclusivity of Objects

- Primitive type – data type that is built into the language; often has a corresponding type on the machine (example: int in C)
- Exclusivity of objects answers the question: is everything (including primitive types) an object?

Object syntax

Java

```
Car c1;    // c1 isn't an object – it's a handle, reference, pointer to an          //object
c1 = new Car();   //the new car is an object
```

C++

```
Car c1;      //c1 is an object
Car * c2;   //c2 is a pointer to a Car object
c2 = new Car();  //the new car is an object
c1.paint("red");   //how to call paint if c1 is an object
c2->paint("red");  //how to call paint if c2 is a pointer
```

//-> is used for “dereferencing” -- get the object

//c2 points to

Object Allocation

- Stack dynamic – object allocated space on *stack* at runtime
- Static – object allocated space by compiler in the *data segment*
- Heap dynamic – object allocated space on *heap* at run time

Memory Allocation

- languages that use a stack for execution (which is nearly all languages) allocate basically three sections of memory
    1) stack
    2) heap
    3) data/text segment

Stack

- frame (activation record) for a function created during execution time when function is called and pushed onto stack
- frame is popped from stack when function is exited
- frame contains:
    - local variables and parameters
    - return address
    - saved registers, old frame pointer

Heap

- heap contains data dynamically allocated via a new, malloc, alloc, etc.
- dynamic means “runtime”

- note that space on the stack is also dynamically allocated (we call this stack dynamic) but is not allocated explicitly via a new

Data/text segment

- data/text segment contains items that are allocated by the compiler
- This includes:
  - program text (instructions)
  - string literals (also called c-strings) - “hello”
  - large constants – those that are too large to be encoded in an instruction

Object deallocation

- When and how are heap allocated objects deallocated?

Polymorphism/Dynamic Binding

- Are all messages to methods dynamically bound?
  - Dynamically bound messages are less efficient than statically bound messages

Assignments between object references

- Consider classes called Parent, Child where Child inherits from (extends) Parent
- There exists an IS-A relationship between Child objects and Parent objects (a Child object is a Parent object)
- The IS-A relationship indicates what assignments between references are valid

parentRef = childRef #allowed

childRef = parentRef #compiler error

Java casts of object references

- a cast of an object reference doesn't change the object; it is only an indication to the compiler what the object will be at run time

- downcast - cast a reference along the class hierarchy in a direction from the root class towards the children or subclasses; (Child) parentRef
- upcast - cast a reference along the class hierarchy in a direction from the sub classes towards the root; (Parent) childRef
- compile-time casting rule - catch attempted casts in cases that are simply not possible; happens when we try to attempt casts on objects that are *totally* unrelated

Assume Male, Female both inherit from Gender:

(Male) femaleRef #error because femaleRef can

#never reference a Male object

- run-time casting rules – the cast must actually be correct (according to IS-A) relationship or runtime error occurs

Parent parentRef = new Parent();

((Child)parentRef).method();

Assignments between object references

- Consider classes called Parent, Child where Child inherits from (extends) Parent
- There exists an IS-A relationship between Child objects and Parent objects (a Child object is a Parent object)
- The IS-A relationship indicates what assignments between references are valid

parentRef = childRef #allowed

childRef = parentRef #compiler error

C++ casts of objects or object references

- casts of pointers don't change object or create new object
- casts of objects will cause new object to be created; needed constructor automatically called
- compile-time casting rule – when casting pointers, same rules as Java apply

- run-time casting rule – no rule, because C++ doesn't do dynamic type checking

Dynamic binding of method calls to method bodies in Java

- by default, method calls to method bodies are dynamically bound (meaning JVM determines which method to call, not compiler)
- exception to this is methods that are declared to be static; these aren't called with an object thus compiler can determine which method is called

Dynamic binding of method calls to method bodies in C++

- by default, method calls to method bodies are statically bound (determined by the compiler)
- A dynamic binding requires
    a) polymorphic variable
    b) virtual method

Polymorphism

- assignment of a different meaning to a method call in different context
- Requires
    - method overriding – method in parent class has same signature as a method in a child class
    - polymorphic variable – can reference a parent class object and a child class object

dynamic binding – which method is called is determined at runtime

Polymorphism in Java

```
class Parent
{
   void foo() { ... }
}
class Child extends Parent
{
```

```
    void foo() { ... }
}
..
..
Parent obj;
..
obj.foo();   //JVM determines which foo is called based upon the type
             //of object referenced by obj
```

Polymorphism in C++

```
class Parent
{
    virtual void foo() {  ...  }
    void goo() { ... }
}
class Child: public Parent
{
   virtual void foo() { ... }
   void goo() { ... }
}
..
Parent * obj;
obj->foo();   //which foo is called depends upon what obj points to
obj->goo();  //call to goo is statically bound (compiler determines it)
              //based on type of obj (Parent *)
```

Single or multiple inheritance

- Supporting multiple inheritance adds complexity
  - Name collision

  - Diamond inheritance

Name Collision

- Child class inherits from two Parent classes where each define the same name
  - example: Parent1 and Parent2 both have a method called display
  - example: Parent1 and Parent2 both have a data member named number
- Language designer needs to come up with a technique (scope resolution) to resolve the ambiguity
- if method is overrided, which is overridden?

Diamond Inheritance

- Both Parent classes are derived from a common Grandparent class
- Should the child class inherit two copies of the Grandparent's data members?

Type checking of objects used to invoke dynamically bound methods

- Type checking – ensuring that operands of an operator are of compatible type (includes ensuring whether the appropriate object is used to invoke a specific method)
- Strongly typed language – a language in which all type errors can be detected either at compile time or at run time
- Static type checking (static typing) – type checking performed at compile type
- Dynamic type checking (dynamic typing) type checking performed at runtime

More C++/Java differences

- C++ has header files
  - These allow classes to be compiled in isolation even if one class references another class
- C++ has a reference type and a pointer type

- Java object handles are always references thus an explicit reference type isn't needed

More C++/Java Differences Parameter Passing

- In mode semantics – formal parameter receives data from actual parameter
- Out mode semantics – formal parameter transmits data to the actual parameter
- Inout mode semantics – supports both
- Pass-by-value – value of actual parameter is used to initialize formal parameter; supports in mode semantics
- Pass-by-reference – access path is passed to formal method; supports inout semantics