Main topics of the week:
- Information Hiding and Data Encapsulation
- Constructors and Destructors
- Inheritance
- Polymorphism and Dynamic Binding
- Practical Language Design Issues

Object Oriented Programming became popular during the late 1980's along with C++ coming into wide use. OOP is a programming paradigm that contrasts sharply to the imperative and functional programming paradigms. The roots of OOP go back to languages like SmallTalk.

The main idea in OOP is that the focus of the programming is on the data, rather than on the actions as in imperative programs. That is, an OO program is viewed as a collection of objects, or “things”, that have behavior and interact with each other. The paradigm of an imperative program is that the program is viewed as a sequence of actions, and data gets manipulated along the way of executing those actions. In OOP, the actions are the behavior (methods of objects) and conceptually “belong” to the objects. The advantage of an OOP approach to problem solving is that it is a more natural correspondence of the problem statement to the program design. The OOP approach can be used in any language, but a so-called Object Oriented language provides more support at the language level to make it easier to express an OO design. The fundamental idea at the implementation level in OOP is that of abstracting the data into object entities and packaging the actions of the object with the object. Objects have an interface, and an OO language allows the interface to be plainly seen while hiding the implementation details.

Information hiding and data encapsulation

Java and C++ are examples of languages that provide good support for Object Oriented Programming. It is certainly possible to write a program in an imperative style in each of these languages. Obviously since C is more or less a subset of C++, any traditional imperative C program from years ago could be compiled with a C++ compiler, but that doesn’t make it an object oriented program. Likewise, a Java program could have all of its code in a single class so that the class variables are effectively global, and then the program would just be viewed a typical C program – a main function, global variables, and a collection of functions.

What makes a program object oriented is its use of data abstraction. Languages like C++ and Java provide language syntax and semantics to support the idea of data encapsulation and control of access to the data. The C language has always had structures as a way of collecting related data items together into user defined types, but all functions in C are global entities, and the data structures must be passed to functions – the language has no way of expressing that the functions “belong” to the data. In C++ and Java, a class not only defines a data structure, but it also includes the functions (or methods as they are called) that operate on the structure. These methods implicitly have access to the data in the structure. Moreover, the languages provide access control so that data can be private, which means that it can only be accessed by the methods defined in the class. This provides a degree of data hiding, so that the layout of the data is a hidden implementation detail, the class is defined publicly by its interface – those properties
(methods) that it chooses to make public. Thus the implementation can change without logically affecting any users of the object as long as the interface stays the same. In the pure view of OOP, all data is private – the interface is the object as a whole, with its public methods to indicate what behavior it makes available. The data becomes the internal state of the object. The objects can be classified according to their type, since a program may have many instances of objects of the same type. This is reflected in the use of the term class to define the different types of objects in a program. At execution, objects of various class types are instantiated to create specific object instances. The instances of the objects interact with each other through the methods of their interfaces – this is often thought of as the objects passing messages back and forth to each other (by calling each other’s methods).

C++ and Java have similar scope rules for class methods and similar access control. The syntax for access control is slightly different. In C++, the keywords private, public, and protected define the access control type for everything up to the next access keyword:

```cpp
class A {
public:
    A() {. . .}
    ~A() {. . .}
    int foo() {. . .}
private:
    int bar() {. . .}
};
```

In Java, each class element is modified with a control access keyword:

```java
class A {
    public A() {. . .}
    public finalize() {. . .}
    public int foo() {. . .}
    private int bar() {. . .}
}
```

Also in Java, packages are used to determine default access control, i.e., classes compiled in the same package can access unmodified members; classes outside the package cannot.

**Constructors and Destructors.**

The variables in an OO program are thus thought of as objects – the primitive types in the language can be thought of as very simple objects, but the term more typically refers to the objects of a class type defined by the user. Otherwise, objects are just another form of variable with the same issues of scope and lifetime. Since we think of objects as more complete or complex entities, the notion of an object’s lifetime takes on more significance – we may be more concerned about what is done when an object is created and when it goes out of existence. For a simple integer variable, these are not great concerns – we may want an integer initialized, and we have syntax to do that, but that initialization is so simple as to not require any great attention. For a more complex object, say one that corresponds to the idea of a file on the disk, there are more things to be concerned with, e.g., does initialization mean the file is opened, does going out of existence mean the file is closed, etc. In general, objects may correspond to entities in the problem space that have some degree of resource management – things that must always be done to keep the object in a meaningful state. Implementing this sort of thing in a language without OO support is error prone since the programmer must always remember
to provide the code to common tasks such as initialization and cleanup that must always be performed. An OO language provides mechanisms to allow the programmer to specify such actions as special in that they are things which should automatically be done. In C++, the language provides **constructors** and **destructors**. These are essentially initialization and cleanup methods that are not called explicitly – the language definition takes care of seeing that they are called at the appropriate times. This provides support for a central OO idea – the data encapsulation embodied in the notion that an object should know how to take care of itself. Java provides constructors as well, and a form of a destructor in a **finalizer** method. However, the automatic garbage collection of Java means that we don’t really know if an object will be destroyed, and when that might happen. Thus, constructors control the way object comes into existence – its initialization. Instead of having to remember to call an initialization procedure, a constructor method is automatically invoked at the object’s creation. This means that no uninitialized errors can occur if the object is designed properly.

Although constructors and destructors are often used for an object’s memory allocation needs, the general idea is really one of resource management. C++ also provides the concept (and syntax) for **copy constructors**. Since C++ has default call-by-value, an object may have special requirements about what it means to make a copy of it as happens in call-by-value. In particular, if the object contains pointers, it may be necessary to clone data that is pointed to so as to avoid conflicting multiple pointers to the same data. Java allows objects to define a **clone** routine, but since Java is call-by-reference (more or less) for objects, the programmer must know to explicitly use the clone method when it is necessary.

All of these language constructs are ways to allow the programmer to express the OO design intent in ways that the compiler/interpreter can understand and enforce.

**Inheritance**

Another one of the fundamental ideas in OOP is that of class **inheritance**. This allows base classes to be used as building blocks and effectively permits the extension of class definitions through derivation. In practice, good use of inheritance maximizes reuse of code. The access protection allows for opaque inheritance as well as limited access for an inheritance hierarchy where the base classes are explicitly designed to be inherited from. Methods in derived class objects may be new methods in addition to methods inherited from the base, or may override the base methods, so the derived class extends and enhances the base class. Good base classes allow the design of more useful libraries since often library routines cannot be used exactly they are, but most of their functionality is desirable. Inheritance is a mechanism for capturing the common aspects of classes of objects, and factors out the **commonality** so that it can be reused.

In C++ and Java, class inheritance eases the notation significantly; the idea of inheritance could be accomplished in C with nested structures, but the syntax required can obscure the fact that inheritance is the underlying concept. The support of C++ and Java allows a clear distinction between the inheritance (“is a”) and containment (“has a”) design principles.

C++ also allows multiple inheritance, however the complexity of virtual base classes makes this tricky. Multiple inheritance can be useful for some designs that compose
classes from several definitions. Java addresses this through the use of interfaces, allowing a class to implement multiple interfaces.

Inheritance facilitates data and implementation hiding – in a good design, base class implementations can be changed without breaking the derived “client” classes. By having inheritance at the level of a language construct, the programmer can predict the impact of changes to the base class, or if there will be any impact.

C++ and Java use similar, but slightly different syntax for inheritance. In C++, inheritance is indicated in a class definition with the symbol ‘:’ and in Java with the keyword ‘extends’. Also, in C++, the inheritance has an access qualifier of public, private, or protected, e.g.,

```c++
class A : public B {
}
```

where in Java all inheritance is public, that is, it is part of the interface:

```java
class A extends class B {
}
```

The protected access is essentially the same for both languages in that protected elements may be accessed by a derived class (superclass).

**Polymorphism and dynamic binding**

In an object oriented design, it is common to have places in the design where there are many different objects that share some common interface, and the commonality is all that matters. In a non OOP language, you often see some “type-switching” occur where some embedded type information is checked to determine how the object is to be treated. The approach using large case statements to check type indicators, and possibly do type casting, tends to spread what should be an encapsulated object design through a program. It is usually very error prone as the software evolves and is maintained. Virtual functions provide language support for this design problem by keeping the design encapsulated, but allowing the desired polymorphic treatment of class objects. Static checking by the compiler can verify that objects have the interface needed, but dynamic binding at runtime is necessary to find the correct implementation of the interface if only the common base is known by static analysis. Because of the overhead of dynamic binding, C++ requires explicit declarations to get virtual function treatment. This is a compromise to the purity of an OOP design, but keeps with the design intent of C++ to not impose costs for features that are not used.

Polymorphism naturally occurs for built in types. For example, integers and floats are treated polymorphically – they both support arithmetic, but the way it is done for each is very different, and the language supports this by using the same notation for arithmetic, but resolving to different implementations depending on the actual type.

A typical example of polymorphism for objects is having heterogeneous lists of objects that have a common base. For example, a desktop might have a list of windows, and want to repaint the desktop. All the desktop knows is that the objects in its list are windows and thus have repaint methods; they will actually be various different types of objects derived from a common window base, and each can implement a repaint method suitable to its own needs.

**Overloading**

Constructors are natural overloaded methods – different initialization parameters often need different implementation methods. Method overloading allows the same
names to be used, and the distinct signature of a method includes its parameters, both
type and count. Thus the commonality of name becomes another way of expressing
design intent. C++ also allows operator overloading – this permits objects to appear
more as first class objects, but it really is just syntactic sugar for cumbersome method call
notation to express operations. Java does not allow operator overloading – named
methods must be used.

It is common to overload the assignment operator in C++ because there may be class
specific semantics in object copying and assignment overloading provides a hook to
implement these semantics. The semantics of assignment are usually closely related to
the semantics of copy construction and destruction – it is common to implement
assignment as the same actions in the destructor (release the resources of the assigned-to
object) followed by the same actions as in the copy constructor (allocate resources
equivalent to the assigned-from object and duplicate its values). It is also possible in C++
to overload the increment and decrement operators, and distinguish between prefix and
postfix. This can greatly simplify notation and give more intuitive meaning where it is
wisely used. It is even possible in C++ to overload pointer operators to implement
smart pointers.

**Operator overloading** gives the effect of making extensions to the language. To see
how desirable this can be, although Java does not implement operator overloading, it
does allow the `+` operator to be used on String objects because of the convenience,
simplicity, and intuitive connotation. In C++, this can be done for any user defined class,
ideally with the intent of improving readability and the expressiveness of the class.

**Exceptions**

Exceptions are a mechanism to handle errors where simple function/method return
values are not sufficient. C has always allowed non-local goto’s for error handling, but
this can be hard to understand, and is particularly disastrous in an OOP design that relies
on the proper handling of objects that go out of scope. Proper exception handling requires
an elaborate scheme of control flow, and is best done with language support that can
ensure the consistency and correctness.

**Parameterized Types**

Parameterized types are similar to the polymorphism concept – the same algorithm
concept (e.g., sort) can be applied to different things. It also promotes code reuse, since it
allows the creation of patterns for constructing class that vary only according to the types
to which they apply. C++ defines a syntax for the definition of templates. The
implementation by various compilers is a tricky business if minimal object code is to be
produced.

**Summary of Object Oriented Language Features**

Some of the language features to support OO design are:

1. Class methods to restrict scope to class
2. Constructors, destructors to encapsulate initialization, cleanup
3. Public, protected, and private access control to support data hiding/encapsulation
The basic principle of the access protection model in C++ is that the class interface is the focus of control to all the class data. That is, syntactically, all the access to the class can be determined by examining the class definition – access cannot be obtained independently from outside the class definition.

4. Overloading of constructors and methods
5. Overloading of operators (C++ only) to support appearance of objects as complete types, particularly assignment (Java provides clone method)
   References were added to the C++ language to ease the notation for operator overloading, but prove to be useful for call-by-reference and aliasing as well.
6. Constant class methods to support readonly/readwrite method use
7. Copy constructors get called on call-by-value passing (C++ only) to guarantee objects are treated consistently
8. Inheritance of classes to support object extension with overriding
9. Access protection to allow base methods to only be called from derived object
10. Virtual methods for polymorphic treatment of objects - C++ requires methods to explicitly be declared virtual (this is a compromise to avoid a performance hit for code that doesn’t need dynamic binding)
11. Pure virtual methods in C++, interface and abstract classes in Java
12. Exceptions
13. Templates in C++ and generics in Java – allows another form of polymorphism as well as standard container class creation.

Practical Language Design Issues

Design of C++

Overriding design goal of compatibility with C
   And comparable efficiency
   Wide user base
   C well understood, powerful, efficient
   Totally new language not likely to be adopted
Don’t try to fix C
   C solves the “computation” problem
   C++ designed to allow better organization of programs
   Syntax not perfect, but cost of incompatibility too high to fix it
   Add to C to allow better design – C with Classes
Don’t force a programming style or paradigm
   Programmers always try to get around restrictions anyway
   Stroustrup had great ideas, but who is to say what is the best?
   Provide language support, not a straight jacket
   Users should not have to pay cost for features they don’t use
Success of C++
   Evolutionary path from C
   Capitalize on C’s flexibility, efficiency, availability, portability
   Interaction with “real” users and responsiveness to suggestions
   Object compatibility with C
Easy enough to attract users and implementers  
Significant value added over C  

How C++ differs from C  
Stronger type checking  
References  
Constants  
Classes, constructors, destructor, access control  
Inheritance, virtual functions  
Function, operator overloading  
Templates, Exceptions, Multiple Inheritance  

Features not in C++  
Garbage collection – deemed too expensive at the time,  
Still probably not appropriate for real time applications  
Any class can implement its own garbage collection  
Concurrency – library approach favored  

Example Design Issues  
Allowing call of an undeclared function was originally in the language for compatibility with C (where any undeclared function is assumed to return an int, which often is okay). The problem is that this becomes a major hole in the type system for C++, so it was eventually disallowed. A design problem here is that continuing to allow the undeclared calls, though permitting compatibility, undermines the confidence in the type checking that C++ does provide. In fact, as users become more reliant on the C++ type checking, it actually becomes more difficult to find the errors caused by allowing these functions through, as well as losing the edge of preventing such errors that is learned in C. A similar phenomenon occurs with auto-prototyping (proto-typing the function according to the first call seen and guaranteeing subsequent calls are consistent). Unfortunately, this only works if the compiler sees two or more calls – a single call is consistent with itself. Although this seems like a nice automation that a compiler ought to perform, it undermines the whole type checking system since the programmer can’t know when consistency can be guaranteed. So this is a case where type checking needs to be as strong as possible if it’s going to be done at all. Similarly, errors should really be errors and warnings should be serious indications of errors lest the real problems be ignored because of too many false alarms.  

In this sense, C++ had a tougher row to hoe than a new language – the mass of C programmers had fervent beliefs and expectations of what was right and wrong, but not everyone was in agreement.  

Another example is tightening of coercions that lose information – e.g., assigning a double to int, or long to int, or int to char. The experiment to disallow these failed, since there was so much existing code that did these types of assignments (and apparently worked).  

Another design issue is the difference between classes and C structures. Stroustrup felt that it was important that these be a single concept – same object layout. This provided for smoother integration of features and kept the user community from fracturing into struct users that avoided classes as inefficient or too new fangled. Of
course, struct had to be retained for compatibility. One could debate whether the class keyword was really necessary, but it probably helped introduce the new features without looking like it was breaking existing code. A class is really nothing more than a struct with default private access. Keeping struct and class the same was in keeping with the “you only pay for what you use” philosophy.

Overloading of operators is an interesting design issue. Current belief at the time held that (1) it would be too hard to implement and make the compiler too big, (2) it was hard to teach and define, (3) such code would be inherently inefficient, and (4) it would make code incomprehensible. For the last objection, Stroustrup correctly reasoned that code doesn’t make programs incomprehensible, programmers do, and no language could ever hope to be usable but un-abusable. The focus should be on how a feature can be used well rather than on how it can be misused. Some blackboard design was convincing enough that overloading would not be unbearably inefficient. So that left the implementation and teaching issue. He added less than two manual pages to describe overloading, and did a first implementation in a few hours.

Although it is certainly true that overloading can be used to make code obscure, it can also be used to make code much more understandable by getting a lot of functional notation out of the way when it should not be the design focus. To the uninitiated, overloading can cause some surprises, but probably no more so than any useful feature. Particularly for things like user defined numeric like types, subscripting, and the now accepted I/O operators << and >>, operator overloading is a good thing. Assignment overloading is a particularly sensitive area (and one of the few operators restricted to be a member function), but extremely useful for controlling the copy semantics of operators. Equality and comparison operators are also able to make for cleaner implementations since they have very intuitive meanings.

Note that some limits are drawn with overloading of operators – namely, you cannot change the syntax of the language (including precedence rules).