

Structs vs. Blocks

Many programming languages, including C, C++, Pascal and Ada, have a notion of grouping data together into *structs* or *records*.

For example:

```
struct complex { float re, im; }
```

There is also the notion of grouping statements and declarations into *blocks*:

```
{ float re, im;  
  re = 0.0; im = 1.0;  
}
```

Blocks and structs look similar, but there are significant differences:

Structs are *data*,

- As originally designed, structs contain only data (no functions or methods).
- Structs can be dynamically created, in any number, and included in other data structures (e.g., in an array of structs).
- All fields in a struct are visible outside the struct.

Blocks are *code*,

- They can contain both code and data.
- Blocks *can't* be dynamically created during execution; they are “built into” a program.
- Locals in a block *aren't* visible outside the block.

By adding functions and initialization code to structs, we get *classes*—a nice blend of structs and blocks.

For example:

```
class complex{  
  float re, im;  
  complex (float v1, float v2){  
    re = v1; im = v2; }  
}
```

Classes

- Classes can be created as needed, in any number, and included in other data structure.
- They include both data (fields) and functions (methods).
- They include mechanisms to initialize themselves (constructors) and to finalize themselves (destructors).
- They allow controlled access to members (private and public declarations).

Type Equivalence in Classes

In C, C++ and Java, instances of the same struct or class are type-equivalent, and mutually assignable.

For example:

```
class MyClass { ... }
MyClass v1, v2;
v1 = v2; // Assignment is OK
```

We expect to be able to assign values of the same type, including class objects.

However, sometimes a class models a data object whose size or shape is set upon creation (in a constructor). Then we may *not* want assignment to be allowed.

```
class Point {
    int dimensions;
    float coordinates[];
    Point () {
        dimensions = 2;
        coordinates = new float[2];
    }
    Point (int d) {
        dimensions = d;
        coordinates = new float[d];
    }
}
Point plane = new Point();
Point solid = new Point(3);
plane = solid; //OK in Java
```

This assignment is allowed, even though the two objects represent points in different dimensions.

Subtypes

In C++ and Java we can create *subclasses*—new classes derived from an existing class.

We can use subclasses to create new data objects that are similar (since they are based on a common parent), but still *type-inequivalent*.

Example:

```
class Point2 extends Point {
    Point2() {super(2); }
}
class Point3 extends Point {
    Point3() {super(3); }
}
Point2 plane = new Point2();
Point3 solid = new Point3();
plane = solid; //Illegal in Java
```

Parametric Polymorphism

We can create distinct subclasses based on the values passed to constructors. But sometimes we want to create subclasses based on distinct *types*, and types can't be passed as parameters. (Types are not values, but rather a **property** of values.)

We see this problem in Java, which tries to create general purpose data structures by basing them on the class **object**. Since any object can be assigned to **object** (all classes must be a subclass of **object**), this works—at least partially.

```

class LinkedList {
    Object value;
    LinkedList next;
    Object head() {return value;}
    LinkedList tail(){return next;}
    LinkedList(Object O) {
        value = O; next = null;}
    LinkedList(Object O,
                LinkedList L){
        value = O; next = L;}
}

```

Using this class, we can create a linked list of any subtype of `Object`.

But,

- We can't guarantee that linked lists are *type homogeneous* (contain only a single type).
- We must cast `Object` types back into their "real" types when we extract list values.

- We must use wrapper classes like `Integer` rather than `int` (because primitive types like `int` aren't objects, and aren't subclass of `Object`).

For example, to use `LinkedList` to build a linked list of `ints` we do the following:

```

LinkedList l =
    new LinkedList(new Integer(123));
int i =
    ((Integer) l.head()).intValue();

```

This is pretty clumsy code. We'd prefer a mechanism that allows us to create a "custom version" of `LinkedList`, based on the type we want the list to contain.

We can't just call something like `LinkedList(int)` or `LinkedList(Integer)` because types can't be passed as parameters.

Parametric polymorphism is the solution. Using this mechanism, we *can* use type parameters to build a "custom version" of a class from a general purpose class.

C++ allows this using its template mechanism. Pizza, an extension of Java, also allows type parameters.

In both languages, type parameters are enclosed in "angle brackets" (e.g., `LinkedList<T>` passes `T`, a type, to the `LinkedList` class).

In Pizza we have

```

class LinkedList<T> {
    T value; LinkedList<T> next;
    T head() {return value;}
    LinkedList<T> tail() {
        return next;}
    LinkedList(T O) {
        value = O; next = null;}
    LinkedList(T O,LinkedList<T> L)
        {value = O; next = L;}
}

LinkedList<int> l =
    new LinkedList(123);
int i = l.head();

```

Overloading and Ad-hoc Polymorphism

Classes usually allow overloading of method names, if only to support multiple constructors.

That is, more than one method definition with the same name is allowed within a class, as long as the method definitions differ in the number and/or types of the parameters they take.

For example,

```
class MyClass {  
    int f(int i) { ... }  
    int f(float g) { ... }  
    int f(int i, int j) { ... }  
}
```

Overloading is sometimes called “ad hoc” polymorphism, because, to the programmer, it appears that one method can take a variety of different parameter types. This isn’t true polymorphism because the methods have different bodies; there is no sharing of one definition among different parameter types. There is no guarantee that the different definitions do the same thing, even though they share a common name.