Object-Oriented Languages: from struct to class

The explosive rise of the object-oriented languages really began with Bjorn Stroustrup's C++, although the concept of representing objects in programs had been around since the 70's. The archetypal O-O language is Smalltalk, developed by the visionaries at Xerox PARC in the 70's. It remains as the purest version of the O-O paradigm, so we study it for its use of the three main aspects of O-O languages that make then different from the imperative languages. They are:

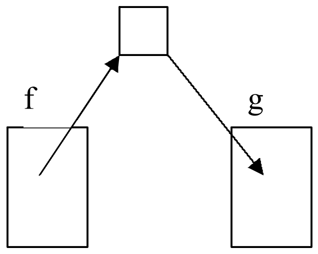
1. Encapsulation
2. Inheritance
3. Polymorphism

We will study these, first from the older but purer, Smalltalk, angle, and then using C++, with a little Java thrown in. First, though, let us briefly describe these and give a model of computation for this group of languages. Encapsulation is the ability to group variables together in a unit that can then be treated as a single entity. We can also encapsulate procedures and/or functions, and other language elements. Inheritance enables types to be related by the sub-type relation and then properties of sub-types to be inherited – basically reused without rewriting – from their super-types. Polymorphism is the ability to maker certain operations independent of the type of object they operate on. All of these three major aspects will be motivated as the solution to a particular problem that the simpler imperative languages have. All three will be extended to include more capabilities than mentioned above, giving the languages more power, but also more room for abuse. By abuse we simply mean that a programmer uses the language in ways that the designers never thought of.

ENCAPSULATION

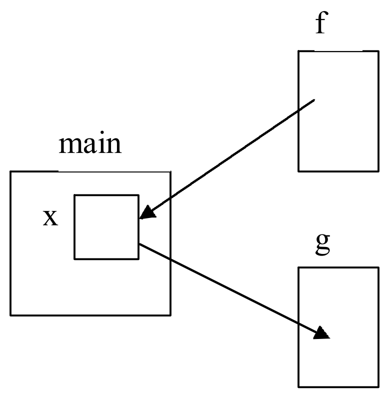
Niklaus Wirth's Pascal was an attempt to address the central issue with the state of programming at the time; programmers should be able to build programs where variables, procedures and functions can be optionally protected from access by 'rogue' procedures, i.e. procedures that should not be accessing tat information. So, Pascal can solve the following problem. Assume that two C procedures (void functions) are communicating a value through a non-local variable:

```
int x;

void f() {
      x = 1;
}

void g() {
      printf("%d", x);
}

main() {
      f();
      g();
}
```



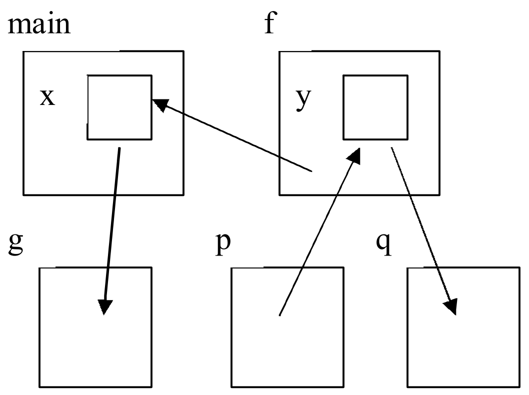
Here, f and g are using x to communicate a value. We could of course us parameters and pointers as well:

```
void f(int *x) {
        *x = 1;
    }

    void g(int x) {
        printf("%d", x);
    }

    main() {
        int x;
        f(&x);
        g(x);
    }
```



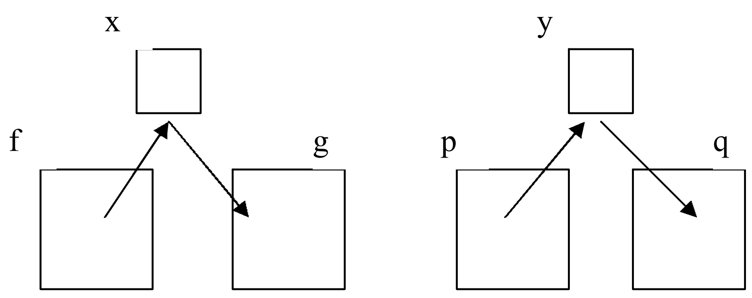
Here, main keeps the variable x and controls everything. If this intended to go to another level, say, in f, then we could write:

```
    void p(int *y) {
        *y = 1;
    }

    void q(int y) {
        printf("%d", y);
    }

    void f(int *x) {
        int y;
        p(&y);
        q(y);
        *x = y;
    }

    void g(int x) {
        printf("%d", x);
    }

    main() {
        int x;
        f(&x);
        g(x);
    }
```



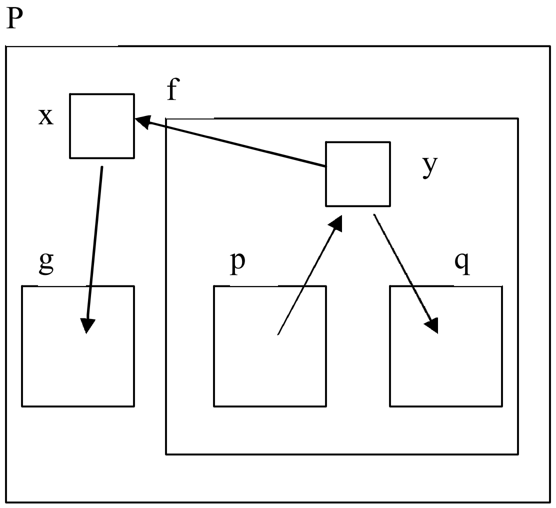
Now f does what main does at the top level – control the use of a variable, y. We cannot achieve the same result with global variables without possible confusion between the two:

```
int x, y;

void p() {
      y = 1;
}

void q() {
      printf("%d", y);
}

void f() {
      x = 1;
      p();
      q();
}

void g() {
      printf("%d", x);
}

main() {
      f();
      g();
}
```



In either case, the problem is one of indiscriminate access. We cannot stop the programmer calling p from main, or g from f, indeed we can call anything from anywhere. The simple two-level scope is incapable of helping. The Pascal solution is nested procedures:

```
program P;
    var x, y : integer;
    procedure f();
        var y : integer;
        procedure p() {
        begin
                y := 1;
        end;
        procedure q();
        begin
                write(y)
        end
    begin
        x := 1;
        p();
        q()
    end;
    procedure g();
    begin
        write(x);
    end;
begin
    f();
    g();
end.
```



or

```
program P;
    var x : integer;
    procedure f(var xp : integer);
        var y : integer;
        procedure p(var y : integer) {
        begin
                y := 1;
        end;
        procedure q(y : integer);
        begin
                write(y)
        end
    begin
        p(y);
        q(y);
        xp := y
    end;
    procedure g(x : integer);
    begin
        write(x);
    end;
begin
    f(x);
    g(x);
end.
```

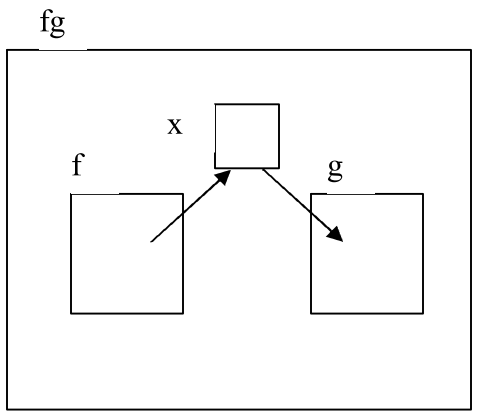
However, again there is nothing to stop me writing a ‘rogue’ procedure h at the level of f and g, or r at the level of p and q, that can interfere with the careful control of the use of variables x and y.

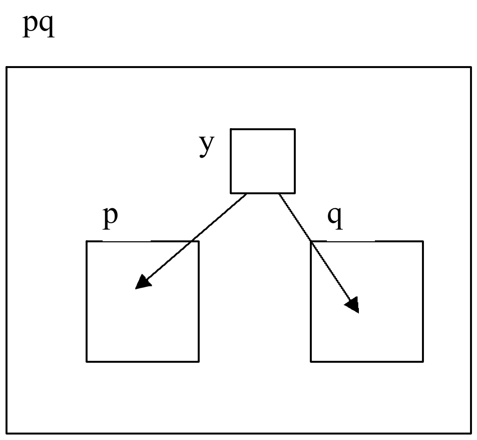
MODULES

The solution that Wirth came up with is the concept of encapsulating f and g along with their shared variable x, and also encapsulating p and q with their variable y. Modula2 and the succession of languages leading to Ada can do something like this (not exact syntax):

```
module fg;
      import p, q from pq;
      var x : integer;
      procedure f();
      begin
            pq.p();
            pq.q();
            x := 1
      end;
      procedure g();
      begin
            write(x);
      end;
begin
      f();
      g();
end.
```

```
module pq;
      export p, q;
      var y : integer;
      procedure p();
      begin
            y := 1
      end;
      procedure q();
      begin
            write(y);
      end;
begin
      /* nothing here */
end.
```

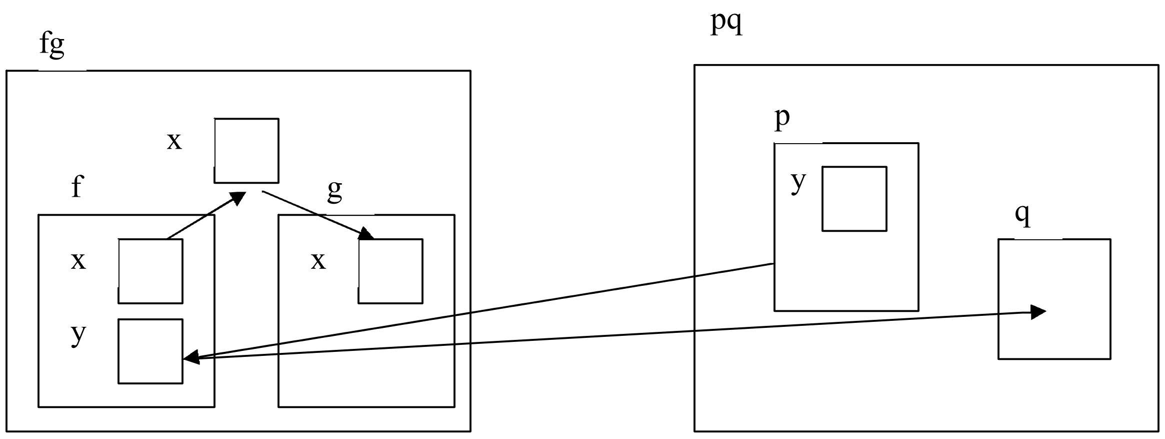




Note the use of the module name to call procedures encapsulated in that module, and, more importantly, the import/export statements that allows module fg to access procedures in pq. Now we have control over who gets access to variable y. f and g have no access to y directly, and p and q have no access to x directly. Furthermore, if we add procedures to modules fg and/or pq, as long as their names are not exported, the other modules cannot call them. The parameter version, where y is still local to f, is:

```
module fg;
      var x : integer;
      procedure f(var x : integer);
            var y : integer;
      begin
            pq.p(y);
            pq.q(y);
            x := y
      end;
      procedure g(x : integer);
      begin
            write(x);
      end;
begin
      f(x);
      g(x);
end.
```

```
module pq;
      export p, q;
      procedure p(var y : integer);
      begin
            y := 1
      end;
      procedure q(y : integer);
      begin
            write(y);
      end;
begin
      /* nothing here */
end.
```



INSTANTIATION – AN OBJECT IS AN INSTANCE OF A CLASS

The package in Ada is the encapsulation mechanism. Anything can be encapsulated in a package – variables, procedures, functions, and, in particular, types. If a type and its operations (implemented as functions and procedures) are encapsulated, then we have an Abstract Data Type. Instances of the type are then variables which can be operated on by the functions and procedures. Types can also be based on other pre-defined types, and these types can inherit the capability of these 'base' types through the use of the type's 'primitive' functions. However, the full use of inheritance only comes about if we identify the encapsulation as *defining* a type. This is what C++, and, later, Java have done through the notion of a class. In Ada, types are separate from the encapsulation mechanism; in C++ and Java, they are the same mechanism.

Thus we come to the instantiation of a class which relates an object of the type defined by the class. Stroustrup, in fact, reinvented this from Smalltalk, which was, in turn, developed from a simulation language called Simula-67. It is natural to think of a type as being an encapsulation, instead of being encapsulated, and the success of C++ and Java have proved this, although there are many other features that contributed to their success. If we start with the struct in C, we can see how this was done. A C struct is an encapsulation that defines a type:

```
struct point {
      float x;
      float y;
};
```

This can be used by declaring a variable of this type, and using procedures and/or functions to operate on it:

```
struct point p1, p2;
...
point translate(struct point p, float dx, float dy) {
      p.x += dx;
      p.y += dy;
      return p;
}
...
p1.x = 10.2;
p1.y = 21.1;
p2 = translate(p1, 2.7, 3.1);
```

The problem here is that there is no access control on how p1 and p2 are to be used. It would be nice to encapsulate the translate function inside the struct, *and* allowing the struct to govern who has access to what and how. Stroustrup's brilliant solution was to allow functions to be encapsulated and to introduce the notion of access control as either private or public. The simple struct has open access – public – to everything inside it, whereas the class encapsulation is exactly the opposite – everything is private. Of course an encapsulation that doesn't allow any access to its components is useless. The addition of the *public* and *private* keywords solved this problem. We thus have the two extremes (everything public and everything private) and the generally accepted good practice in between where all data (variables) are private and all operations (functions) are public. In addition, he dropped the parameter of the class type the encapsulates it since all functions would be passed the object on which they are operating. This is done by a hidden pointer parameter whose value is bound to the keyword 'this'. In the example, we return the object which is being operated on by dereferencing the pointer this.

```
// everything public
struct point {
      float x;
      float y;
      point translate(float dx, float dy) {
            x += dx;
            y += dy;
            return *this;
      }
};

// everything private
class point {
      float x;
      float y;
      point translate(float dx, float dy) {
            x += dx;
            y += dy;
            return *this;
      }
};

// private data, public functions
class point {
private:
      float x;
      float y;
public:
      point translate(float dx, float dy) {
            x += dx;
            y += dy;
            return *this;
      }
};
```

The advantage of private/public is that the we now have every kind of control over access; the disadvantage is the we have too much variety. In fact, as we shall see, we can use this flexibility to good effect, but beginners should always stick to the data private, operations public idea. This was the only kind of access control possible in Smalltalk, and where access to date was necessary, *accessor functions* were introduced. In C++, this is achieved by *inspectors* (read-only functions) and *mutators* (write-only):

```
class point {
private:
      float x;
      float y;
public:
      point translate(float dx, float dy) {
            x += dx;
            y += dy;
            return *this;
      }
      float getx() { return x; }
      void putx(float newx) { x = newx; }
};
```

In this example x can be changed inside an object (an instantiation of the class point), but y cannot:

```
point p;
float xx;
...
p.putx(5.6);
xx = p.getx();
p.translate(3.2, 4.5);
xx = p.gety(); // this is illegal since there is no such function in point
```

There remains one problem. If data is private, and there are no mutators, then how do we initialize the data. On solution is to set every variable in an object to zero. A better idea, and the one that C++ uses, is to have a public *constructor* function do the job. This function has two jobs. One is to actually allocate space for the new object (the user has no control over this) and the other is to initialize the space. An example with our point class might be, in fact, to set the variables to zero, but it could use any values. The constructor is not a true function. It has the hidden jobs of space allocation, but it also returns the new object and thus there is no need to specify a return type. Its name must also be the same as the class, and, to start with, it needs no parameters. Such a function is the *default* constructor.

```
class point {
private:
      float x;
      float y;
public:
      point() { x = y = 0.0; } // the constructor - could set x and y to any value
      point translate(float dx, float dy) {
            p.x += dx;
            p.y += dy;
            return *this;
      }
      float getx() { return x; }
      void putx(float newx) { x = newx; }
};
```

So a more complete example of the use of this class is:

```
point p1, p2;
float xx;
xx = p1.getx();          // xx is now 0.0
p1.putx(3.2);            // p1 now has x = 3.2, y is still 0.0
```

```
p2 = p1;                 // p2 now has x = 3.2, y = 0.0
p2.translate(5.6, 7.8); // p2 now has x = 8.8, y 7.8
xx = p2.getx();          // xx is now 8.8
```