Object-Oriented Programming

The Three Paradigms of Programming

As far as I know, there are three paradigms of programming:

1. Logic Programming.
2. Functional Programming
3. Imperative Programming

Computing $N!$ with a Logic Program

```
fact( 0, 1 ).
fact( N, M1 ) :- N > 0, N1 is N - 1,
                 fact( N, M ), M1 is M * N.
```

(The meaning of `:-` is $\leftarrow$.)

- A logic program consists of true statements about the functions that one wants to compute.
- In theory, one should not worry about efficiency, but in practice one has to: Order of formulas matters, and order inside formulas matters.
- LP supports backtracking very conveniently.

Computing $N!$ in a Functional Language

```
define fact(N) :
   if N = 0 then 1
   else
      fact( N - 1 ) * N.
```

- The program can be interpreted as characteristic equation, or definition.
- Designed by mathematicians. Among the three paradigms, it is the closest to mathematical representation.
- Computation is rewriting with the definition.

Computation of $N!$ in an Imperative Language

```
int fact( int n )
{
   int fact = 1;
   while( n != 0 )
   {
      fact = fact * n;
      n = n - 1;
   }
   return fact;
}
```

Imperative Languages

- Imperative Languages evolved from machine languages:

  Connecting Wires ⇒ Assembler ⇒ Fortran/Cobol ⇒ Algol/Pascal/C ⇒ Java/$C^{\#}$/C^{++}.

- Imperative languages have assignemnts, commands that **do** or **change** something. The computation model is based on the notion of state.

Imperative Languages versus Functional Languages

Despite the different origins, imperative programming and functional programming are not that far apart.

For programs that have as aim to compute a result, imperative programming is a subset of functional programmming.

For each point in the program, one can define a function that computes the result starting at this point in the program.

Imperative Languages versus Functional Languages (2)

For **fact**, the result is:

```
define fact(N) :
   fact_while( N, 1 ).

define fact_while( N, F ) :
   if( N == 0 ) then F
   else
      fact_while( N - 1, F * N ).
```

Imperative Programming versus Functional Programming (3)

- Modern languages, like Java, $C^{\#}$ and C^{++} are machine independent. Java and $C^{\#}$ have garbage collection.
- Since Algol, all programming language have local variables with stack like semantics. (FORTRAN does not have this)

Programming in functional style is possible in modern imperative languages, and you should do this wherever possible.

Definition of object-Oriented Programming

OOP is a buzzword. All modern programming languages have OO features.

But nobody seems to really understand what it is.

Wikipedia says (on Feb 2009): ’Object Oriented Programming (OOP) is a programming paradigm that uses ”objects” and their interactions to design applications and computer programs.’

Using this definition, object-oriented programming is restricted to imperative programming.

Standard Way of Obtaining Data Structures

In languages that are not object-oriented (like Pascal, or C), data structures can be obtained by the following, recursive constructors:

- Primitive data types: **int, bool, real, double**.
- If D is a data type then **array [1 .. N] of** D is a data type.
- If D is a data type with a natural order, then **range D1 .. D2 of D** is a a data type.
- If $D_1, \ldots, D_n$ are data types, $N_1, \ldots, N_n$ are identifiers, then **record** $N_1 : D_1$ **. . .** $N_n : D_n$ is a data type.

Difficulties with Records

Mathematically seen, the record constructor simply constructs ordered tuples of form $D_1 \times D_2 \times \cdots \times D_n$.

For many applications, (maybe even most), this is not adequate:

- Not every tuple $(d_1, \ldots, d_n)$ represents a valid object.
- Objects that are equal can be represented by distinct tuples.

Problems with Records (2)

Dates can be defined as follows:

```
day = range 1 .. 31 of int
month = range 1 .. 12 of int
year = int
data = record { day : day, month : month, year : year }
```

Not every combination is a valid date:

$(29, 2, 2009),\ (31, 4, 2008)$.

Determining which combinationa are valid dates can be very tricky.

Floating point numbers can be defined as follows:

$\textbf{double} = \textbf{sign} \times \textbf{mantisse} \times \textbf{expsign} \times \textbf{exp}$.

Representation of double is again tricky. The number 0 has two representations. Mantisses must be normalized.

Class = Record + Access Functions

The main achievement of modern programming languages is:

class = Set of Tuples + Small set of access functions.

Only the access functions can access the tuples.

Access functions gaurantee that:

- No ill-formed tuples can be constructed.
- Tuples that have the same meaning cannot be distinguished.

For **data**, access functions could be:

nextdate, previousdate.

differenceindays

adddays, subtractdays.

weakday

Theoretically, access functions can also be used in C or Pascal, but in practice this is difficult.

In C/Pascal access to the representation is not blocked, so that the user of a class can forget to use the access functions.

C^{++} automatically inserts assignments, constructors.

Other Applications of Controlled Access

- Guaranteeing correctness of algorithm. For example a sorting algorithm that is guaranteed not to forget or invent elements.
- Hiding memory management. (important in C^{++}.)
- Easy replacement of implementation. (As long as the access functions remain the same)

Polymorphism (Inheritance)

If class C_1 has all access functions which C_2 has, then a C_1 can be used wherever a C_2 can be used.

This is called inheritance.

In my view, the importance of class inheritance is overestimated, and one must be extremely careful when using it.

Even when C_1 has all access functions that C_2 has (with the same name), their behaviour need not be the same.

(For example N, R)

Inheritance should not be confused with reimplementation.

Polymorphism (Interfaces)

If some set of classes $C_1, \ldots, C_n$ shares a set of access functions, it is possible to define an interface I based on these access functions.

It is said that $C_1, \ldots, C_n$ implement I, or that $C_1, \ldots, C_n$ inherit from I.

Interface inheritance is related to class inheritance but one must not confuse them. (An interface has no direct members)

Given an interface, it is possible to define a function or a data structure on I. Such function/data structure will work for every C_i.

Polymorphism (Templates)

If some set of classes $C_1, \ldots, C_n$ shares a set of access functions, it is possible to define a data structure or a function parametrized by a C_i.

For example list$< C_i >$ or print $< C_i >$

The difference between templates and interfaces is that: With a template, a separate copy of the datastructure/function is compiled for every C_i : This is more efficient at run time, but more compilation is necessary and the resulting code is longer.

With a template, it is known at compile time which C_i will be used. It is not possible to use different C_i in the same data structure. This loss of flexibility can be a disadvantage, but it allows compile time type checking.

The current implementation of templates in C^{++} is unpleasant.

Summary

Functional Programming and Imperative Programming are closely related. You should be aware of that and use elements of functional style whenever possible.

Classes are a way of defining types that can be combined with imperative and with functional programming.

Defining the right class (right set of access functions) is hard. It requires experience, and I will give a lot of attention to this during the class.

Classes allow polymorphism. Templates are more important than interfaces are more important than class inheritance.