Imperative Programming Languages (IPL)

Definitions:

- The imperative (or procedural) paradigm is the closest to the structure of actual computers.

- It is a model that is based on moving bits around and changing machine state

- Programming languages based on the imperative paradigm have the following characteristics:

    - The basic unit of abstraction is the PROCEDURE, whose basic structure is a sequence of statements that are executed in succession, abstracting the way that the program counter is incremented so as to proceed through a series of machine instructions residing in sequential hardware memory cells.

    - The sequential flow of execution can be modified by conditional and looping statements (as well as by the very low-level goto statement found in many imperative languages), which abstract the conditional and unconditional branch instructions found in the underlying machine instruction set.

- Variables play a key role, and serve as abstractions of hardware memory cells. Typically, a given variable may assume many different values of the course of the execution of a program, just as a hardware memory cell may contain many different values. Thus, the assignment statement is a very important and frequently used statement.

Examples of imperative languages:

- FORTRAN, Algol, COBOL, Pascal, C (and to some extent C++), BASIC, Ada - and many more.

PL/I

- PL/I (1963-5): was one of the few languages that attempted to be a general purpose language, rather than aiming at a particular category of programming.
- PL/I incorporated a blend of features from FORTRAN, ALGOL, and COBOL, plus allowed programmers to create concurrent tasks, handle run-time exceptions, use recursive procedures, and use pointers.
- The language development was closely tied to the development of the IBM/360, a line of "general use" computers.
- The main problems with the language were its large size and the interaction of so many complex features.

↳ Simula 67:

- SIMULA 67: yet another descendant of ALGOL, SIMULA was the first language to support data abstraction, through the class concept.

↳ Pascal:

- PASCAL (1971): an extension of the ALGOL languages, it survived as a teaching language for structured programming, it still has widespread (though rapidly declining) use in the teaching community, but comparatively little commercial use.
- It has stronger type and error checking than Fortran or C and more restrictive syntax, hence enforces some fundamental programming concepts better than C (perhaps).

↳ C:

- C (1972): C presented relatively little that was new or remarkable in terms of programming language design, but used and combined established features in a very effective manner for programming.
- It was designed for systems programming, and initially spread through close ties with UNIX.
- C has numerous and powerful operators, and extensive libraries of supporting function.

- It has (comparatively) little in the way of type checking, which makes the language more flexible for the experienced user but more dangerous for the inexperienced.

⮱ **Ada**

- Ada (1975-1983): Ada, like COBOL, had its development sponsored by the Department of Defense, and survived as a language largely because of mandated use by the DoD.
- In design, Ada's developers tried to incorporate everything known about software engineering to that time. It supports object oriented programming, concurrency, exception handling, etc.
- The design and implementation of the language suffered through being perhaps too ambitious.

⮱ IPL Characteristics:

- Variable and Storage
- Commands:

  - ✓ Assignments
  - ✓ Procedure call
  - ✓ Sequential commands
  - ✓ Collateral commands
  - ✓ Conditional commands
  - ✓ Iterative commands
  - ✓ Block commands

⮱ Assignments

- Simple assignment:

  x = y +1;

- Multiple assignment:

  v1=v2=v3=v4=200;

- Simultaneous assignment:

  n1,n2,n3,n4 =m1,m2,m3,m4

- Operator-assignment commands:

  m +=n

Procedure Calls

- The effect of a procedure call is to apply a procedure abstraction to some arguments

- The net effect of a procedure call is to update variables (local or global).

Sequential commands

- Much of imperative languages are concerned with control flow, making sure that commands are executed in a specific order.

- A sequential command is a set of commands executed sequentially.

- In the sequential command:

  ‘C1; C2;’

  C2 is executed after C1 is finished.

➩ **Collateral commands**

- A computation is deterministic if we can predict in advance exactly which sequence of steps will be followed. Otherwise the sequence is nondeterministic.

- A collateral command is a set of nondeterministic commands.

- In the command:

  ‘C1; C2;’

  C1 and C2 are executed in no particular order.

➩ **Conditional commands**

- A conditional command has a number of subcommands, from which exactly one is chosen to be executed.
- Example: the most elementary if command:

```
if E then
    C1
else
    C2
end if;
```

- Conditional commands can also be nondeterministic:

```
If    E1 then C1
    | E2 then C2
      …
    | En then Cn
end if;
```

- Nondeterministic conditional commands are available in concurrent programming languages (such as Ada).

- Another conditional command is the Case statement.

➪ Iterative commands

- An iterative command, also known as loop, has a set of commands that is to be executed repeatedly and some kind of phrases that determines when the iteration will stop.

- Control variable in the definite loops:
    - Predefined variable
    - The loop declares the variable
    - The initial value is atomic or comes an expression.

- Two types of iterations:
  - ✓ Definite (For loop)
  - ✓ Undefinite (While loop)

Side-effects in IPL

- In some IPL, the evaluation of expressions has the side effect of updating variables.

```
/* A program in C-like syntax, with side-effects
*/
int i=1;
main() {
        int y = 5;
        printf("%d\n",f(y)+g(y));
        printf("%d\n",g(y)+f(y));
}
int f(int x) {
        i = i*2;
        return i*x;
}
int g(int x) {
        return i*x;
}
```

- The two printf statements will not print the same answer. This means that, for this program

f(y) + g(y) is different from g(y) + f(y)

Is it bad programming?
or

side-effect of f on variable i?

➯ What is wrong with side-effects in sequential execution ?

- Program is not readable: The result from a function depends on what happened during the execution of another function.

- Reusability: A program fragment depends on a global environment

- Correctness of a program becomes almost impossible

- Good programming: ensure that side-effects never occur.
- How can we enforce programmers to avoid side-effect?

➷ How can side-effects be avoided ?

- The problem is destructive assignment.

- Whenever a statement like

```
x = 8;
```

is executed then the old value of x is destroyed and the new value, 8, substituted. To be safe this implies that the previous value of x cannot be needed again.

- So to avoid side-effects, abolish **destructive assignment**!

➷ Is there an alternative to imperative programming languages ?

Other programming paradigms.

Case Study - C

- **History**
  - Kernighan and Ritche designers
  - language designed to implement operating system (Unix)
  - terse, compact, but can write really fast code
  - free, ported with Unix
- **types (minor difference with Pascal)**
  - static typing
  - weak typing
  - standard primitive types - but no booleans
  - enumerated types (in ANSI C)
  - composite types
  - arrays
  - records (structs)
  - variant records (unions)
  - no sets

↳ **expressions**

- literals
- aggregate expressions (a[] = {2, 3, 4})
- function calls (limited to returning primitive types, so composite values are not first-class)
- conditional expression ((2 < 3)? 1 : 0)
- constants (in ANSI C) and variables

↳ **storage**

- classic run-time storage model
- selective and total updating of composite

↳ **Variables**

- static and dynamic arrays
- heap storage for values allocated by calling malloc()
- "uncontrolled use" of pointers, pointer craziness

Commands

- structured programming constructs (e.g., if-then-else, for-loop)
- assignment is an expression
- multiple, composite assignment
- procedure (void functions) and function calls

Bindings

- static scoping
- nested name spaces (can declare vars after a {)
- new-type and type declarations
- new-variable, but not variable declarations
- limited recursive declarations

Abstractions

- user-defined function and procedure abstractions
- built-in selector abstractions only
- parameter passing call-by-value and call-by-pointer
- eager evaluation of parameters

↳ **encapsulations**

- no packages, objects, ADTs

↳ **type systems**

- built-in operator overloading/coercion
- type coercion via casting
- no user-defined overloading
- no polymorphic types
- no parameterised types

↳ **sequencers**

- **gotos**
- escapes via **returns**
- **break** escapes from containing block
- no exception handlers (setjmp, longjmp are in library)