Introduction to programming techniques

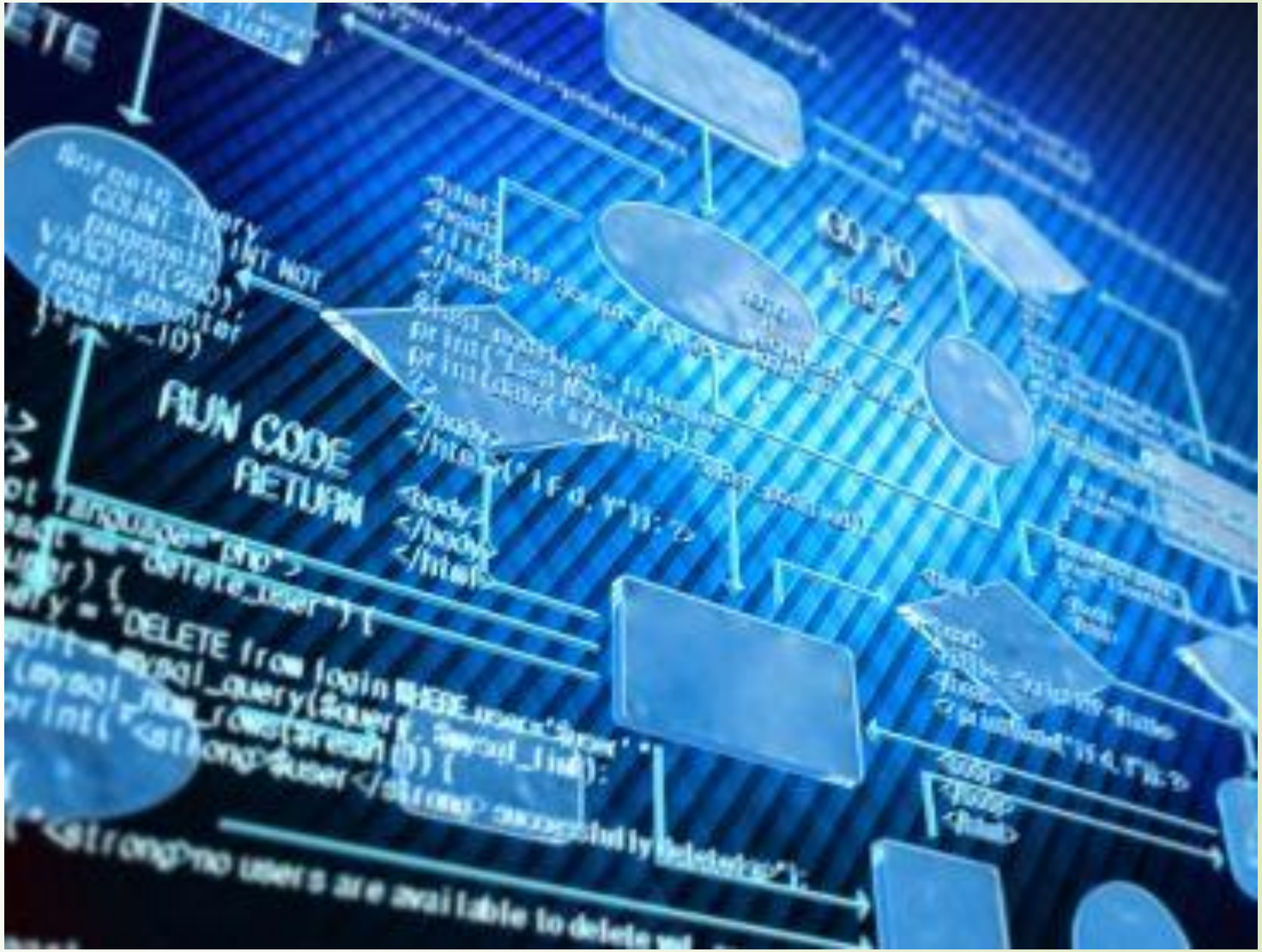
Dott. Costantino Zazza, costantino.zazza@unitus.it

IR0000032 – ITINERIS, Italian Integrated Environmental Research Infrastructures System
(D.D. n. 130/2022 - CUP B53C22002150006) Funded by EU - Next Generation EU PNRR-Mission 4 “Education and Research” - Component 2: “From research to business” - Investment 3.1: “Fund for the realisation of an integrated system of research and innovation infrastructures”

Finanziato dall'Unione europea NextGenerationEU
Ministero dell'Università e della Ricerca
Italiadomani

Introduction to Programming Techniques (1)

- History
- Programming models
- The compilation phase
- C and FORTRAN compilers in Linux
- Compilers options
- The make utility



History

The duty a any computing machine is barely to execute a set of instructions in order to solve a given numerical problem.

But, in any case, the overall workflow is always the same:

History

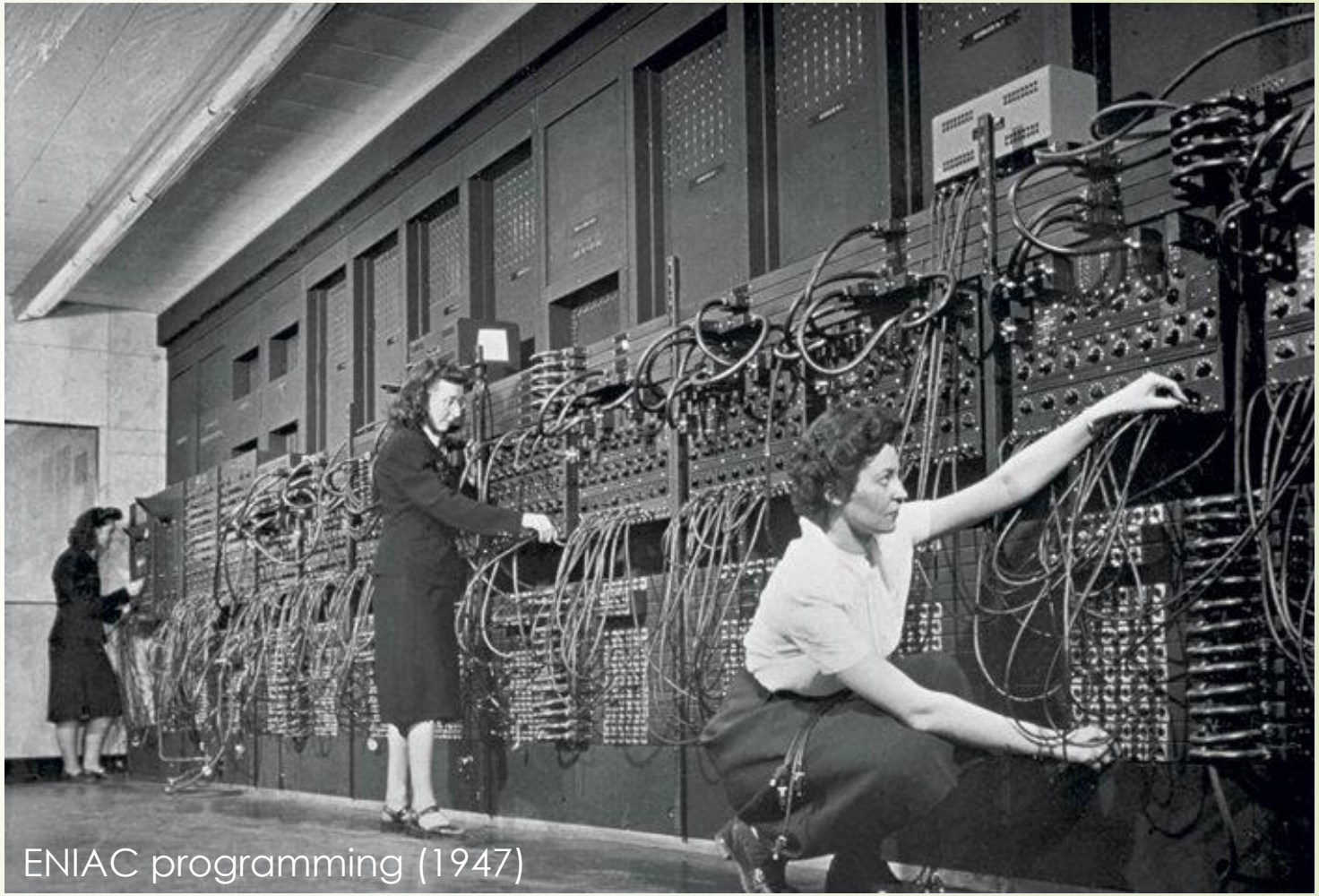
To better exploit the computing machinery the designers at the beginning of this story, though to divide the instrument onto two layers.

The **hardware** layer, including the physical (electronic) devices of the computing system and the **software** layer, that is, the whole instructions (over a given data set) the same machine has to execute.

At the beginning, the instructions were written in binary, the physical format of the digital device. Then, it was soon understood this solution was to inefficient, not flexible enough to permit a real computation and, even worse, at that time changing a code needed to modify the hardware underneath!

The engineers then designed a solution: introducing a low level coding (assembly) for the binary instructions and data.

VERY GOOD! But still nowadays, understanding a set of instructions written in binary code is arduous undertaking ☹



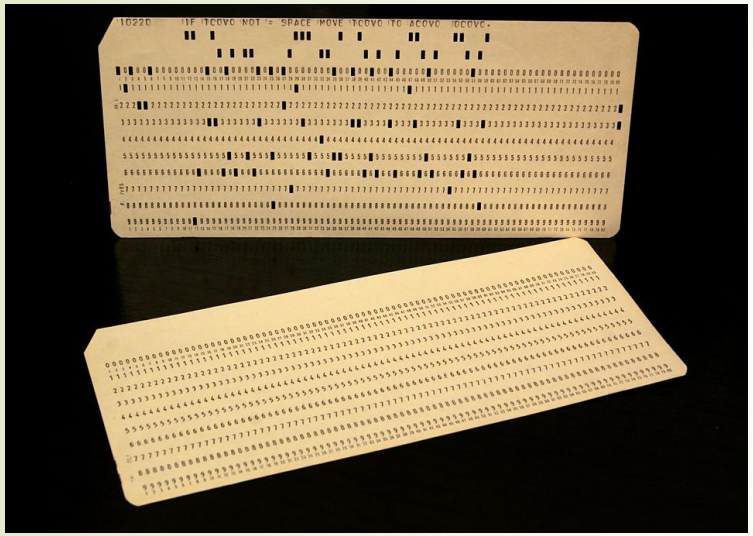
ENIAC programming (1947)

History

The in the 50's some visionary engineers understood we had to simplify the instrucions' coding in a way this would closer to the programmer rather than the physical system.
It was in those years the FORTRAN language is born, the first of its kind, with the name derived from joining the words FORmula TRANslator.
Since its first issue, FORTRAN introduced a new concept: the **programming language**, that is, a syntax to write instructions for a computer in a coded and reproducible way.
This last aspect is fundamental: for the first time we split-up the physical (electronic) devices from its instructions (and data).
Thanks to the concept of programming language(*) a given program (code) written following the syntax defined in the language itself, could be run on different computing systems and provide the same results starting from the same input data set.

(*) *Assigning a meaning to the syntactic forms of a programming language is called* ***semantics of the language***.

History



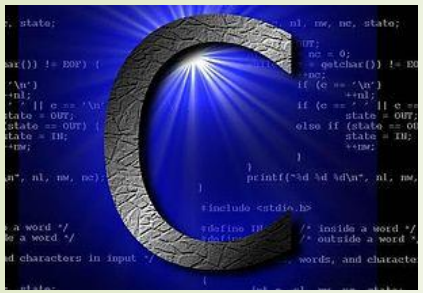
To make this feasible, it is necessary to maintain a strict adoption of precise standards defining the way of coding, that is, the procedures we must use to write instructions and data.
All in all, a language describe the procedure with which a code MUST be written and the standard of the language will outline the usage limits to programming cases over real (computing) devices.

E.g., the FORTRAN born as language at the end of 40's but its first (industrial) standard was written in the 50's: the FORTRAN4.
Afterwards, the functionalities of the language were extended and a new standard was released: FORTRAN66.
This to say that, programming languages, as any human concept, are steadly evolving and for FORTRAN we passed by the standard below:

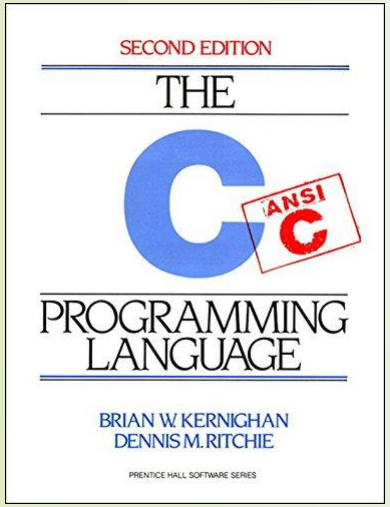
FORTRAN4 > FORTRAN66 > FORTRAN77 > **FORTRAN90** >FORTRAN95 (e HPF)…

FORTRAN programming (1947)

History



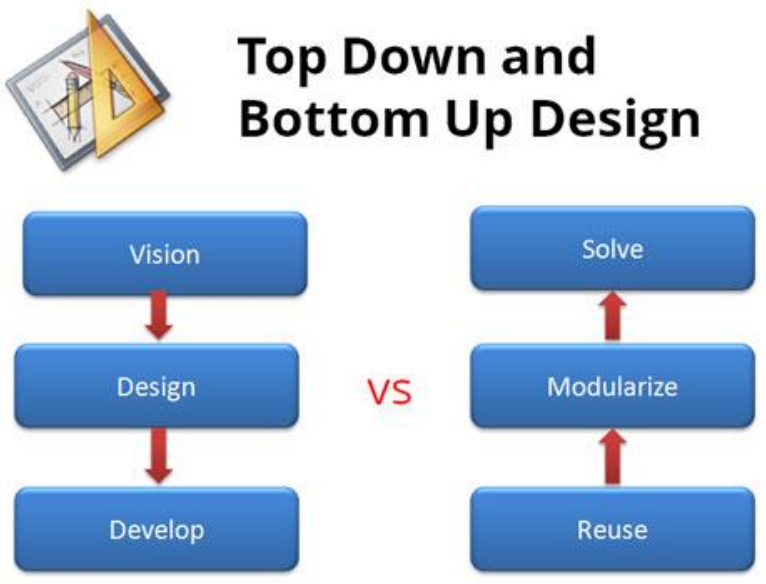
The FORTRAN was born (and has been modified) as laguage to translate scientific formulas for digital computers.
The primary objective was to have a tools to write mathematical formulas in an efficient way for the executions by a computer indipendently from physical (electronic) system underneath.



This is NOT, of course, the story of any programming language…

The C programming language, for example, was conceived by **Kerningham** e **Ritchie** to have a tool to write an new layer of software (called Operating System) to make the software/hardware separation even stronger.
Born in this way in the 70's a language which revolutionised the idea itself of a computer: *not only the formulas but also the control system (Operating System) could be made independent from hardware*.
Then, it is thanks to C that Unix was born at the Bell Labs (a USA telco) and thanks to the fact that was written with a portable language, Unix moved quickly to all existing calculators. In practice, there is no hardware platform that does not support some kind of Unix (Linux) operating system.

Programming models



So thanks to programming languages, we can give precise instructions to our computer to make sure that it solves for us a certain numerical problem otherwise difficult (or even impossible) to solve with our own intellectual means.

But what rules do we have to follow to write a good program to the computer?

There is no one-on-one answer to this question and one of the tasks of the experts in Computer Science is to study the best method of code translation, of a given scientific theory. This is done by studying new languages and programming models that are increasingly sophisticated and high-level (far from hardware) to make this translation as simple and efficient as possible.

Let's look in detail now on one of the most commonly used strategies for writing programs to the computer:
the **TOP-DOWN design**.

ENIAC programming (1947)

Programming models

When writing a program to the computer, the best strategy is to start with the main program, that is, from those instructions that generally make up the processing. Imagining our usual approach to the functionality of a calculator (input, processing and output), the main program will read the data to be processed, these will be processed according to a certain mathematical formula and finally the results will be printed for the final reading.

From the point of view of the main program (**main**) you can think of the problem as a whole and then try to identify the main sections of the entire processing. Once these sections are identified, you can divide the problem according to them and try to see if each of these components is not further divisible into simpler subsections. The process can continue until the single problem in the code subsections you choose is simple enough.

ENIAC programming (1947)

This iterative process is called top-down design or stepwise refinement.

Programming models

So we have to think about solving our problem starting from the top (the scientific problem of the user...) to get to define all the details of the calculation to be done, one by one, for subsequent approximations.

But how can we be sure that our problem breakdown is the best possible? Again the answer is not unique but one thing we need to know and to do: *write our instructions according to a precise* ***algorithm***.

In the first approximation we can think of an algorithm as the strategy used to solve a given problem. But to be more precise we should say that a solution technique in order to be an algorithm must be subject to three basic requirements:

1. It must be ***clear*** and ***uniquely defined***.

2. It must be ***effective***, that is, each step must be executed.

ENIAC programming (1947)

3. It must be ***finite***, in the sense that it must end after a finite and limited number of steps.

Programming models

Suppose we want to calculate

$$C = \pi \cdot r$$

Is there an algorithm to solve this simple equation?

NO! A solution technique that includes the multiplying operation by r and the *exact value of π is not effective*, since it is not possible to calculate the exact value of π !

ENIAC programming (1947)

Programming models

Ultimately, solving a problem with the calculator consists of two distinct steps:

In the *first*, you develop an algorithm, that is, you choose an existing one, which solves the problem.

The *second* is expressed by this algorithm as a computer program through a programming language. The latter process is called **coding**.

FORTRAN and C are high-level programming languages and as we know, these languages are designed to be independent of the underlying hardware, in the sense that after coding (writing) our program, this will run on different computers in the same way.

ENIAC programming (1947)

The compilation phase

To make it possible for a program written in high-level language to run on different computers, the program must first be translated into the machine language (low-level or assembly/assembler) suitable for the host computer.

The programs that do such a work of translation are called, **compilers**.

Of course, before you can run a given program on a certain computer, you need to write the program's text to a file that will reside generically on a file system (Unix) on disk. As we know, each file in a Unix file system must have a name that is generally divided into two parts divided by a period (.).

When we create a file, the name part before the period (.) is called root name and what comes after it its extension.

The compilation phase

Some extensions have a pre-assigned meaning: a **.c** file indicates a program file written in C language, and a **.f** file indicates the FORTRAN-language equivalent.

These files containing the text of a program are called *source files*, and the process of modifying the contents of one of them is called *program editing*.

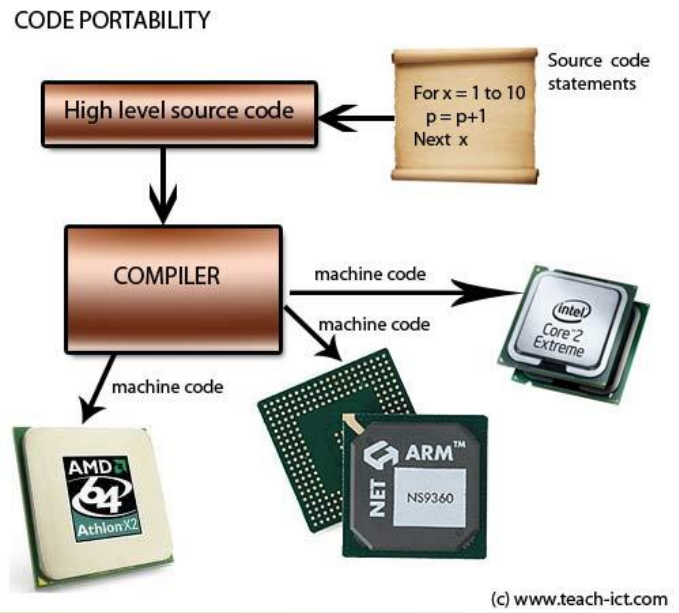
Of course, editing a source file can only be done through the word processing tools made available by the operating system and in the case of Unix we have seen that one of these tools is ***vi***.

In our day there are extremely sophisticated program development tools that allow the modification of source files through a "friendly" graphical interface and are able to guide the entire production phase of executable code.

These tools are called Interactive Development Environment (**IDE**) and are the simplest approach to programming.

The compilation phase

But before we understand how these tools work, let's see more in detail how an executable program is produced under Unix.



Once we have our source file written in the desired programming language, we have to use a ***compiler*** of that language to translate our source into a format that is understandable directly from the computer we are working on.

Of course, this translation process varies from machine to machine and is the function of the operating system present. However, in most cases and also under Unix, the compiler translates the source file into a second file, called an **object file**, which contains the appropriate instructions for our computer where a Unix operating system runs, for example.

ENIAC programming (1947)

The compilation phase

This object file is then combined with other object files to produce an **executable file**. These other object files typically include other predefined object files, called libraries, which contain different operations commonly required by user programs.

The process of combining all individual object files into an executable, is called **linking** and the command that executes it, **linker**.

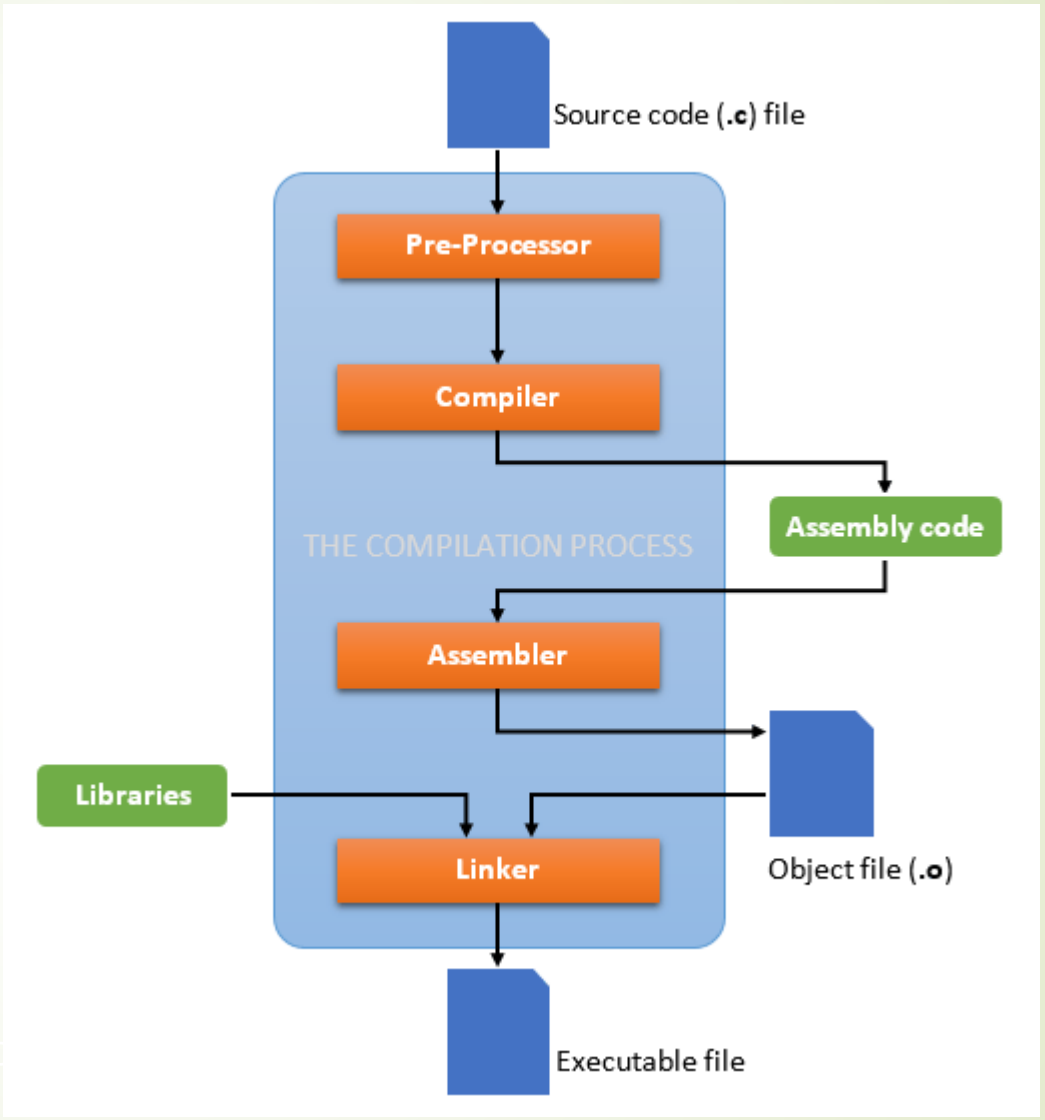
But let's look at the whole process of producing an executable through a flow-chart(*):

ENIAC programming (1947)

(*) A flowchart is the graphical representation of an algorithm expressed by uniquely defined graphs.

The compilation phase

Structure of the compilation phase



The compilation phase

To simplify, we can then say that compiling a source code in an executable is a two-step process:

(1) Translation of the source file to the object file. The object file contains meta-instructions that cannot be executed yet but can be combined with other object files compiled for the same hardware/software architecture.

(2) Converting the object file (and other objects possibly in libraries that need it) to an executable file for a given hardware/software architecture.

In Unix as on other operating systems, the command that invokes the compiler can also invoke the linker to assemble the final executable. Below we will see masked the Unix linker function (called **ld**) through the compiler-only call. The linker however operates independently of the compiler (Assembler, FORTRAN, C, Pascal, etc.) only for convenience of use it can be invoked (or not) by using an option on the command line of the compiler itself.

The compilation phase

Although we don't yet know how to write a program in a given programming language, let's see what the development environment described in the previous flow chart looks like in Linux OS.

First of all, the **compiler**.

What command should I give to invoke the compiler and have it translate (compile) my source program?

The generic syntax is similar to that of each Unix/Linux command:

```
<comp> [options] [executable-file/object] file-source
```

For e.g. (*)

```
f77 -c -o foo.o foo.f
```

translates the FORTRAN source file foo.f into the foo.o object file using the FORTRAN compiler (f77).

(*) In this example we do not invoke the linker **ld**, as we only want to produce the object file pippo.o (option –c, "compile-only").

The compilation phase

All right, but how do I know the name of the c compiler, FORTRAN, Pascal, etc. present in my Unix?

I expect them to be called differently ...

Let's see how to access some of the most popular compilers in the Unix environment:

	FORTRAN	**C**	**C++**
IBM/AIX	xlf (xlf90)	xlc	cpp
Intel/Linux	ifort	icc	icc
PGI	pgf77 (pgf90)	pgcc	pgc++
GNU	gfortran	gcc	g++

... and anyway on each operating system I'll always find a **cc** command for my C compiler and an **f77/f90** command for my FORTRAN compiler!

The compilation phase

Of course, it helps us the standardization work done in recent years in Unix environment and that allowed to produce the POSIX4 standard. In this, as in previous standards, among other things it is established that the Unix development environment must be uniformed and that *therefore compilers will also have to have similar use in different Unix dialects*.

What is even more useful is that all compilers have a subset of common options. This increases the portability of the code (from one calculator to another) and simplifies the interface to the programmer:

Let's look at some of the most commonly used options:

-o	**Object (executable) name**
-c	**Compile only**
-p (-pg)	**Profiling**
-g	**Debugging**
-On	**Optimisation level**
-Wn	**Warning level**

ENIAC programming (1947)

The compilation phase

At the same time, there are specific options regarding the language and architecture (or processor) of the machine we are using:

Language dependent options:

C

```
-cpp            Run C preprocessor
-Dxxx           Execute prepocessor directive xxx
-ansi           Support ANSI C programs
```

.....

FORTRAN

```
-align dcommons     Align commons
-col72              Use max 72 cols
-iN                 Set integers to N (2,4,8) bytes
```

ENIAC programming (1947)

The make utility

The make utility is essentially a programming language to automate the build process.

The idea behind make is that the programmer does not have to recompile a certain source file if a corresponding object file already exists. The object file is said to be current if it was compiled more recently than the last change to its source file.

As an example, consider the following situation:

We have just edited the source file program.f and want to compile it and "link" it to the inputs.f and outputs.f modules.

The corresponding command (f77) will be:

```
f77 program.f inputs.f outputs.f
```

which compiles and links the three modules correctly.

The make utility

If the modules object inputs.o and outputs.o already exist, and if they are current (*unmodified*) we could also give the command(*):

```
f77 program.f inputs.o outputs.o
```

In this case, f77 would have compiled just program.f to produce program.o and linked all together program.o, inputs.o and outputs.o.

Of course, by compiling a single program instead of three, the build-linking process will be much faster and more efficient.

Operating as in the second case, when you have many source files and objects available, it becomes a very laborious operation and hence, the need for a utility like make that can do this for us automatically.

(*) In this case, having not put the –c option, the compiler after generating the file object automatically invokes the ld linker to generate the executable. In Unix, if we do not name our executable using the –o option, the default name assigned to it is **a.out**.

The make utility

In addition, make reduces the possibility of compilation errors that can occur both for command line errors or in the choice of modules (objects) to use.

To understand how make works, we must first define some terms:

target	An operation to perform. In most cases, it contains the name of the file to be created.
dependency (depend-on)	A relationship between two targets: Target A depends on target B if a change in B produces a change in A.
up-to-date	A file that is newer than each of the files it depends on.
makefile	A file that describes how to create one or more targets. It lists all the files on which the targets depend and the rules necessary to build these targets correctly.
make	The command that executes what is in a makefile. By default make searches for a file called makefile or Makefile in the current directory. As an option you can give the target to execute.

The make utility

But let's take a practical example, bringing back in a makefile the case seen before with f77 with two different targets: e.g., we use two different options for f77.

```
#
# A simple makefile
#(Comment lines begin with #)
#
# First target:
#
program:
    f77 -o program –O program.f inputs.f outputs.f

# Second target:
#
program.db:
    f77 -o program-db -DDEBUG –g program.f inputs.f outputs.f
```

ENIAC programming (1947)

If we use this makefile with make we can choose between the two targets simply by writing them after the command make …

The make utility

For example,,

```
make program
```

will run the command

```
f77 -o program –O program.f inputs.f outputs.o
```

and will produce the executable **program**.

It is important to remember that make will create a new shell for each of the commands it encounters after the target and that these must be given after the [TAB] character (each line of commands for a target begins with a [TAB]...).

The make utility

If we want our targets to depend on one or more files, then we must indicate their dependencies. This is done by adding the name of the files on which a certain target depends after the [:]

```
program: program.o inputs.o outputs.o
        f77 –o program program.o inputs.o outputs.o
```

The first line tells us that program depends on **program.o**, **inputs.o** and **outputs.o** while, the second uses **f77** (here, **ld**) to link object files if they already exist.

If object files do not exist, they must have their dependencies in the makefile and there must be a target for each of the object files. This way make first generates the object files and then the executable.

Ultimately our complete makefile, will be

The make utility

```
program: program.o inputs.o outputs.o
    f77 -o program program.o inputs.o outputs.o

program.o: program.f
    f77 -c -o program.o program.f

inputs.o: inputs.f header.f
    f77 -c -o inputs.o inputs.f

outputs.o: outputs.f
    f77 -c -o outputs.o outputs.f
```

The make utility

You can also use shortcuts and macros in the makefile. Among the most commonly used abbreviations we have

$@ Indicates the full name of the target

$* Indicates the name of the target without suffix

Using them in a line in the previous example we will have:

```
program: program.o inputs.o outputs.o
     f77 –o $@ $*.o inputs.o outputs.o
```

Instead, a macro is used in a makefile to group several names (or commands) into a single variable. For example, if we put in the macro **DEPENDS** our object files, we can write the previous line as

```
DEPENDS = program.o inputs.o outputs.o

program: $(DEPENDS)
     f77 –o $@ $(DEPENDS)
```

The structure of a program

Then let's see how a generic C or FORTRAN program (procedural) presents itself to the programmer who intends to read it: