A Start to the Road Ahead

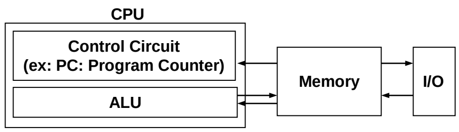
1

Programming

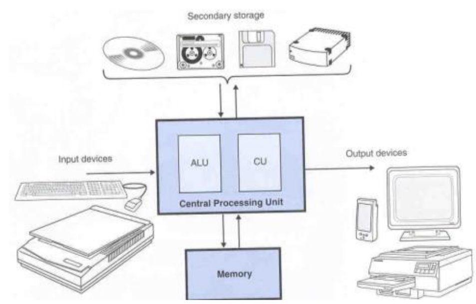
3

Computer Organization

- A typical Von-Neumann Architecture



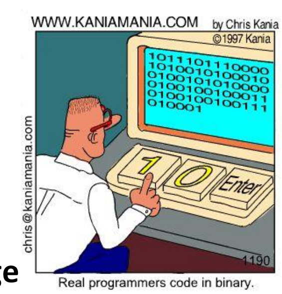
- Example:
  - Input unit
  - Output unit
  - Memory unit
  - Arithmetic and logic unit (ALU)
  - Central processing unit (CPU)
  - Secondary storage unit



fit@hcmus

Why Do Programming?

- Humans communicate in a **natural language**
  - Large vocabulary (10 000s words)
  - Complex syntax
  - Semantic ambiguity
    - *The man saw the boy with the telescope.*



- Machines communicate in **binary code / machine language**
  - Small vocabulary (2 words… 1, 0)
  - Simple syntax
  - No semantic ambiguity

fit@hcmus

So?

Humans - natural language
- Large vocabulary
- Complex syntax
- Semantic ambiguity

Machines - binary language
- Small vocabulary
- Simple syntax
- No semantic ambiguity

Programming language
- Vocabulary: restricted
- Syntax: small and restricted
- Semantic: no ambiguity (almost)

fit@hcmus

Machine Languages

- Machine dependent.
- Native tongue of a particular kind of computer.
- Each instruction is **a binary string**. The code is used to indicate the operations to be performed and the memory cells to be addressed. This form is easiest form of computers to understand, but most difficult for a person to understand.
- Strings of numbers giving machine specific instructions
  - +1300042774
  - +1400593419
  - +1200274027

fit@hcmus

Assembly Languages

- Machine dependent.
- English-like abbreviations representing elementary computer operations (translated via assemblers)
- Again specific to only one type of computer. Uses descriptive names for operations and data, e.g. , “LOAD value”, “ADD delta”, “STORE value”.
- **Assemblers** will translate these to machine languages.
  - LOAD BASEPAY
  - ADD OVERPAY
  - STORE GROSSPAY

fit@hcmus

High-level Languages

- Machine independent
- Codes similar to everyday English.
- Write program instructions called statement that resemble a limited version of English.
- Portable: can be used on different types of computers without modifications.
- Use mathematical notations

```
grossPay = basePay + overTimePay
a = a + b
```

Before	a	10
	b	7
After	a	17
	b	7

High-level Languages

Language	Application Area	Origin of Name
FORTRAN	Scientific programming	Formula Translation
COBOL	Business data Processing	Common Business-Oriented Language
Lisp	Artificial Intelligence (AI)	List Processing
C	System Programming	Predecessor B
Prolog	AI	Logic Programming
Ada	Real-time distributed systems	Ada Augusta Byron & Charles Babbage
Smalltalk	GUI, OOP	Objects "talk" via message
C++	Supports object & OOP	C (++ is the increment operator)
JAVA	Supports Web programming	Originally named "Oak"

Semantic Gap

- A "semantic gap" exists between the amount of information conveyed in assembly language vs high level languages.
- Consider the following C++ single statement:

```
x = x + 3;
```

- This single statement may require many assembly language statements (operations):

```
Load memory location 24 into accumulator
Add a constant 3 to the accumulator
Store accumulator in memory location 24
```

- The number of executable statement expands greatly during the translation process from a high-level language into assembly language.

Programing Paradigms

Programming Paradigms

- A programming paradigm is a style, or “way”, of programming.
- Programming paradigms:
  - Declarative
    - Focus on WHAT the computer should do
    - Programming by specifying the result you want, not how to get it.
  - Imperative
    - Focus on HOW the computer should do.
    - Programming with an explicit sequence of commands that update state.

Programming Paradigms

declarative	
functional	Lisp/Scheme, ML, Haskell
dataflow	Id, Val
logic, constraint-based	Prolog, spreadsheets
template-based	XSLT
imperative	
von Neumann	C, Ada, Fortran, . . .
scripting	Perl, Python, PHP, . . .
object-oriented	Smalltalk, Eiffel, Java, . . .

Programming Paradigms

```
int gcd(int a, int b) {                         // C
    while (a != b) {
        if (a > b) a = a - b;
        else b = b - a;
    }
    return a;
}

(define gcd                                     ; Scheme
  (lambda (a b)
    (cond ((= a b) a)
          ((> a b) (gcd (- a b) b))
          (else (gcd (- b a) a)))))

gcd(A,B,G) :- A = B, G = A.                     % Prolog
gcd(A,B,G) :- A > B, C is A-B, gcd(C,B,G).
gcd(A,B,G) :- B > A, C is B-A, gcd(C,A,G).
```

Compilation and Interpretation

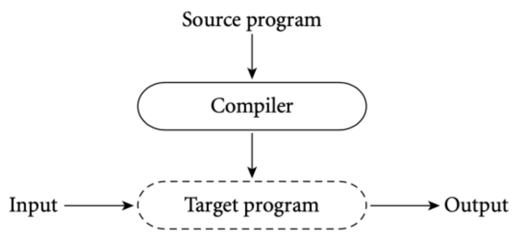
Compilation and Interpretation

- **properties** of the *implementation* of a language.
- It's not accurate to say that a language is interpreted or compiled because interpretation and compilation are **both properties of the implementation** of a particular language, and **not a property of the language itself**.
- Any language can be compiled or interpreted. It just depends on what the particular implementation that you are using does.

Compilation

fit@hcmus

- A **compiler** will translate the program directly into code that is specific to the target machine (known as **machine code** – basically code that is specific to a given processor and operating system). The computer will run the machine code on its own.



Compilers

fit@hcmus

- Usually
  - the compiler translates directly into machine language
  - But each type of CPU uses a different machine language

  … so same executable file will **not work on different platforms**

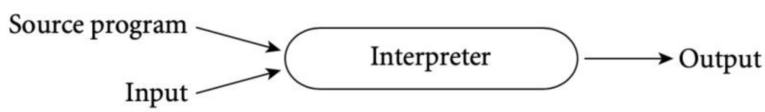
  … need to **re-compile** the original source code on different platforms

- Some programming languages used this way: C, C++, Erlang, Haskell, Rust, and Go.

fit@hcmus

Interpretation

- The source code is not directly run by the target machine.
- Another program (the ***interpreter***) reads and then executes the original source code.
- Some programming languages used this way: PHP, Ruby, Python, and JavaScript.



fit@hcmus

Comparison

BASIS FOR COMPARISON	COMPILER	INTERPRETER
Input	It takes an entire program at a time.	It takes a single line of code or instruction at a time.
Output	It generates intermediate object code.	It does not produce any intermediate object code.
Working mechanism	The compilation is done before execution.	Compilation and execution take place simultaneously.
Speed	Comparatively faster	Slower
Memory	Memory requirement is more due to the creation of object code.	It requires less memory as it does not create intermediate object code.
Errors	Display all errors after compilation, all at the same time.	Displays error of each line one by one.
Error detection	Difficult	Easier comparatively

More Reading

- Notes:
  - https://www.guru99.com/difference-compiler-vs-interpreter.html
- Video Ref:
  - https://www.youtube.com/watch?v=I1f45REi3k4

Error Messages

- ***Syntax error****:* A grammatical mistake in a program
  - Detected by the compiler
- ***Bug***: A mistake in a program
  - cannot be detected by the compiler
  - The process of eliminating bugs is called *debugging*

(who found the first COMPUTER bug?)

Types of Errors

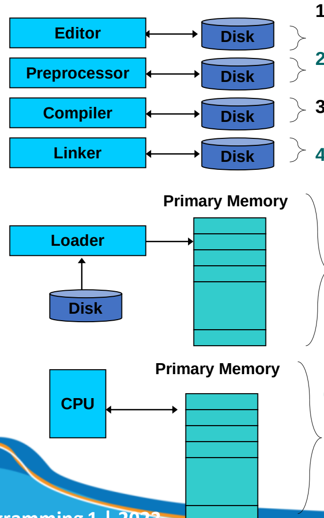
- **compile-time errors**
  - The compiler will find syntax errors and other basic problems
  - An executable version of the program is not created
- **run-time errors**
  - A problem can occur during program execution
  - Causes the program to terminate abnormally
- **logical errors**
  - A mistake in the *algorithm*
  - Compiler cannot catch them
  - A program may run, but produce incorrect results

Development Environments

- IDE (Integrated Development Environment)
- you will edit, compile and run

fit@hcmus

A Typical Program Development Environment



1. Program is created in the editor and stored on disk
2. Preprocessor program processes the code
3. Compiler creates object code and stores it on disk.
4. Linker links the object code with the libraries
5. Loader puts program in memory.
6. CPU takes each instruction and executes it, possibly storing new data values as the program executes

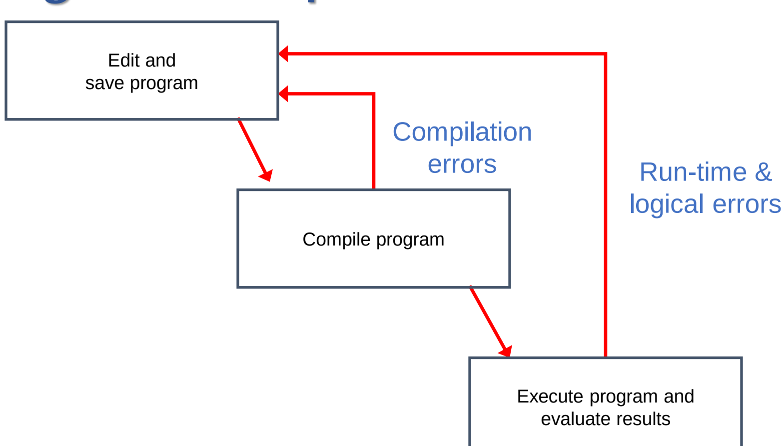
1. *Edit*
2. *Preprocess*
3. *Compile*
4. *Link*

5. *Load*
6. *Execute*

26

fit@hcmus

Basic Program Development



27

Algorithm

Algorithm

- A finite **sequence of well-defined steps** to solve a problem
  (can be) expressed in natural language.
- Other definition:
  - A finite sequence of precise instructions which leads to a solution.
- Equivalent words:
  - recipe, method, directions, procedure, routine.

Characteristics of Algorithm

fit@hcmus

- Finiteness
  - For any input, the algorithm must terminate after a finite number of steps.
- Correctness
  - Always correct. Give the same result for different run time.
- Definiteness
  - All steps of the algorithm must be precisely defined.
- Effectiveness
  - It must be possible to perform each step of the algorithm correctly and in a finite amount of time.

Algorithm | An Example

fit@hcmus

```
Algorithm that determines how many times a name occurs in a list of names:

1.    Get the list of names.
2.    Get the name being checked.
3.    Set a counter to zero.
4.    Do the following for each name on the list:
      Compare the name on the list to the name being checked,
      and if the names are the same, then add one to the counter.
5.    Announce that the answer is the number indicated by the counter.
```

Algorithm | Another Example

fit@hcmus

- **`Input:`** `No`
- **`Output:`** what do you think about the output?

- `Step 1. Assign sum = 0. Assign i = 0.`
- `Step 2.`
  - `Assign i = i + 1`
  - `Assign sum = sum + i`
- `Step 3. Compare i with 10`
  - `if i  < 10, back to step 2.`
  - `otherwise, if i ≥ 10, go to step 4.`
- `Step 4. return sum`

Pseudocode

fit@hcmus

- An algorithm
  - expressed in a **more formal language,** code-like
  - but does not necessarily follow a specific syntax

fit@hcmus

Pseudocode | An Example

```
Step 1:  Input M1,M2,M3,M4
Step 2:  GRADE ← (M1+M2+M3+M4)/4
Step 3:  if (GRADE <50) then
             Print "FAIL"
         else
             Print "PASS"
         endif
```


fit@hcmus

Program

- An algorithm
    - expressed in a programming language
    - follows a specific syntax

Program | An Example

- A source code expressed in C++:

```
double grade = (m1 + m2 + m3 + m4)/4;

if (grade < 50)
   std::cout << "Fail" << std::endl;
else
    std::cout << "Pass" << std::endl;
```


Flowchart

fit@hcmus

Flowchart

- A flowchart is a diagram that depicts the “flow of control” of a program.

fit@hcmus

Flowchart

- Use a separate slide.

fit@hcmus

Problem Solving

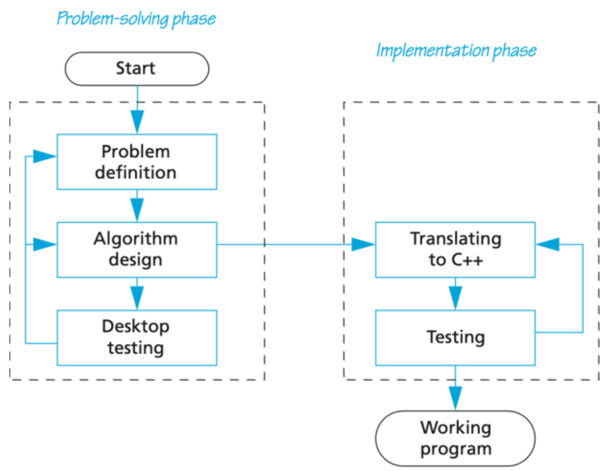
fit@hcmus

Problem Solving

- The purpose of writing a program is **to solve a problem**.

- The general steps in problem solving are:
  - Understand the problem
  - Design a solution (find an algorithm)
  - Implement the solution (write the program)
  - Test the program and fix any problems

Program Design Process



45

Another Algorithm

Give a non-negative integer **N**:

Make a variable called **x**, set it equal to (N+2)

Count from **0** to **N** (include both ends), call each number **i**:

Write down the value (**x** * **i**)

Update x to be equal to (**x** + **i** * **N**)

When you finish counting, write down the value of **x**

46

Another Algorithm

Give a non-negative integer **N**:

Make a variable called **x**, set it equal to (N+2)

Count from **0** to **N** (include both ends), call each number **i**:

Write down the value (**x** * **i**)

Update x to be equal to (**x** + **i** * **N**)

When you finish counting, write down the value of **x**

N	
x	
i	
OUTPUT	

Another Algorithm

Give a non-negative integer **N**:

Make a variable called **x**, set it equal to (N+2)

Count from **0** to **N** (include both ends), call each number **i**:

Write down the value (**x** * **i**)

Update x to be equal to (**x** + **i** * **N**)

When you finish counting, write down the value of **x**

N	
x	
i	
OUTPUT	

Another Algorithm

fit@hcmus

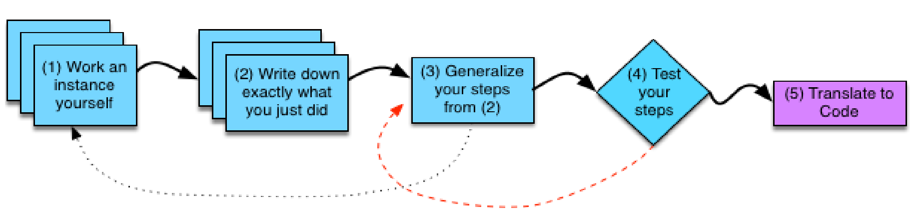
```
Give a non-negative integer N:
      Make a variable called x, set it equal to
(N+2)
      Count from 0 to N (include both ends), call
each number i:
            Write down the value (x * i)
            Update x to be equal to (x + i * N)
      When you finish counting, write down the
value of x
```

N	2
x	4
i	0

OUTPUT

Designing an Algorithm

fit@hcmus



Designing an Algorithm

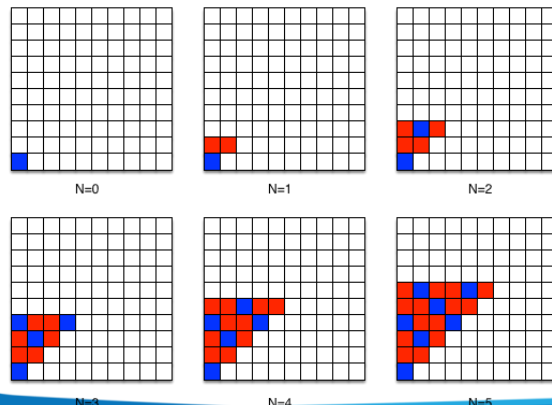
- Design an algorithm to check whether *n* is a prime number or not.

Designing an Algorithm

- Design an algorithm to compute *x* to the power of *y*.

Designing an Algorithm

- Design an algorithm to produce a pattern of blue and red squares like in the following examples, which demonstrates *N* from 0 to 5.



Questions and Answers