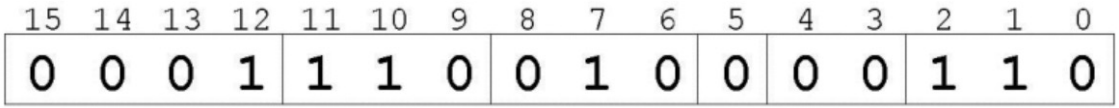
Hello C!

Hyung-Sin Kim

Computing for Data Science

Graduate School of Data Science, Seoul National University

ISA-based machine code is perfect for computers



What's wrong with it?

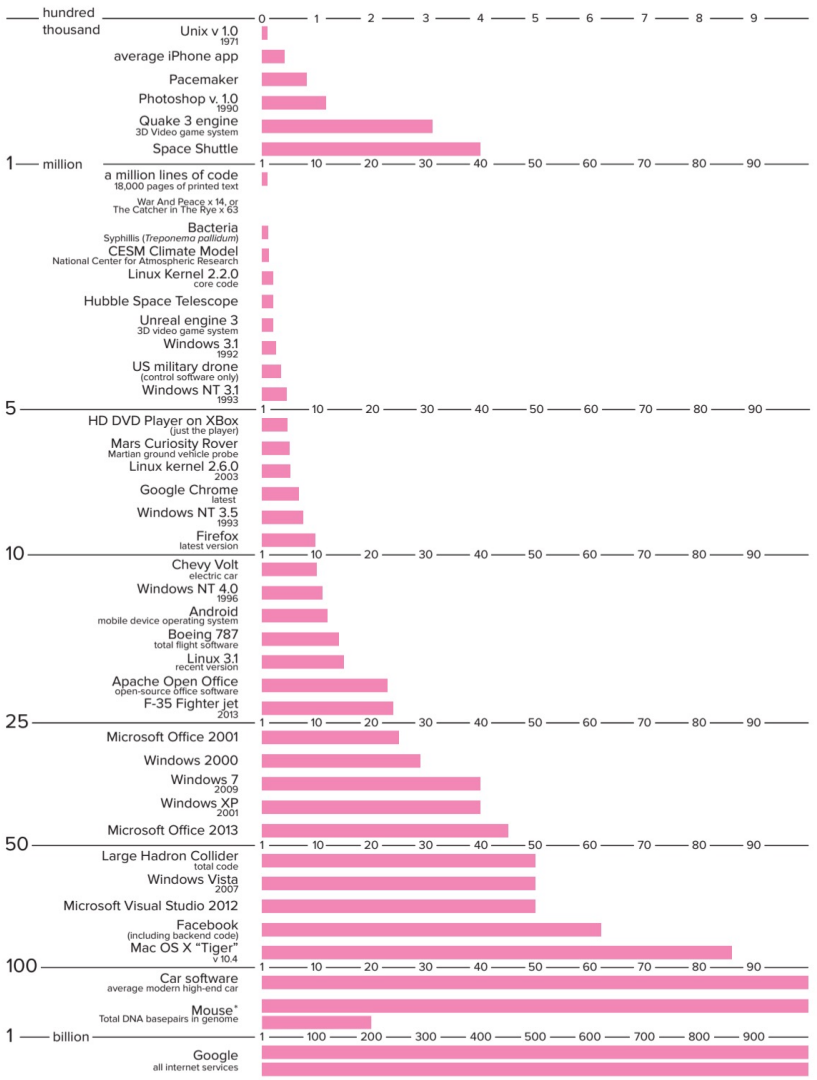
Computer scientist Margaret Hamilton poses with the Apollo guidance software she and her team developed at MIT.

Photos: MIT Museum

You need to write only this much code ☺
(and this is not even a machine code, but an assembly code)

Code Size …

- **Gap**: Computers love machine codes (ISA) but humans **don't**
- As software size increases significantly, improving **programmer productivity** becomes more important!
  - Teams of programmers should be able to more rapidly and maintainably develop correctly working code
- Filling the gap between C and H
  - Assembler (but still like Apollo code…)
  - **High-level languages**! (C/C++, Python)



Why High-level Languages?

- Easily **manage the values** upon which we are computing
  - Low-level languages take care of values and where they are stored
    - But do **NOT** consider what the values mean
  - A value is represented as a meaningful symbolic name (e.g., temperature)
  - Language takes care of allocating storage and performing data movement operations

- **Human-friendly expression** of computation
  - Programmers can express complex tasks with a smaller amount of code since the code looks more like a human language
    - Symbolic names (e.g., temperature) and control structures (e.g., if/else and for)

Why High-level Languages?

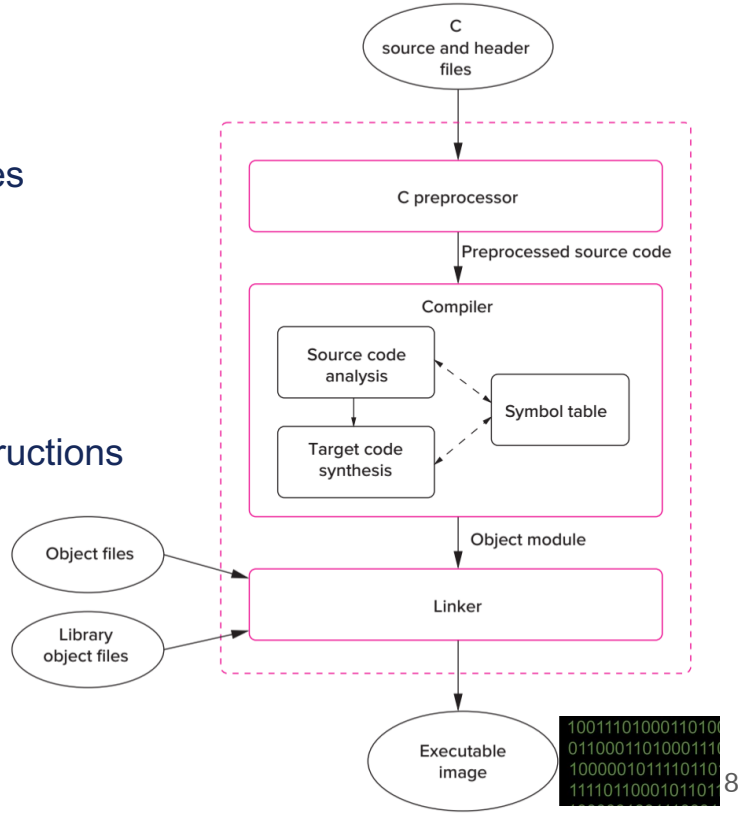
- **Abstraction** of the underlying hardware
  - A uniform programmer interface regardless of the underlying hardware
    - **Portability**: easily and efficiently targeted for various different devices
    - **Diverse operations**: More operations than those supported by ISA
- Better **maintainability**
  - Since common control structures are expressed using simple, English-like statements, the program becomes easier to read and for others to modify and fix
- **Safeguards** against bugs
  - Make the programmer adhere to a stricter set of rules
  - If certain rules or conditions are violated, an error message will direct the programmer to the spot in the code where the bug is likely to exist

Translating High-Level Languages

- **Interpreter** (program)
  - Receives a high-level language program as a set of commands
  - Translate and execute the program one line, command, or subroutine at a time
  - **Pros**: Easy debugging and developing (examine intermediate results and modify code on the fly), portability
  - **Cons**: Slow execution due to the intermediary step
- **Compiler** (program)
  - Receives an entire high-level language program as an input
  - Translate the whole program into a machine language program that can be directly executed on the hardware (i.e., an executable image), but does **NOT** execute it
  - **Pros:** Program execution is fast, repeatable, and memory efficient (production software)
  - **Cons:** Harder to debug based on execution, compiled programs may not be executed on a different hardware (less portable)

C Compiler

- Preprocessor
  - Gets source and header files (xx.c, xx.h)
  - Substitute all preprocessors with their real values
    - #include stdio.h, #define DAYS_THIS_MONTH 30
  - Output is still C
- Compiler
  - Gets the processed source code
  - Output is an object file comprising machine instructions
    - One object file per c file
- Linker
  - Combine the generated object files and library object files
  - Output is a single executable image



Let’s dive into C now!

Our First C Program – Create a C File

- $cd ~/Desktop // This is like “바탕화면 폴더” in Windows
- $mkdir ComputDS // make a folder
- $cd ComputDS // go to the folder
- $gedit helloC.c // create a C file

Our First C Program – Source Code

```
#include <stdio.h>                  // Preprocessor directive

int main(void)                      // Function main
{                                   // Function block marked by {…}
    printf("Hello C!\n");           /* Printing function */
    return 0;
}
```

- Save and close the gedit window
- Check if your source file has been created
  - $ls

Our First C Program – Compile and Execute

- Compilation with gcc compiler (generate an executable from a source file)
  - $gcc [source file name] –o [executable image file name]
  - $gcc helloC.c –o helloC
- Check if your binary image file has been created
  - $ls
- Execute the binary file
  - $./helloC
- Do you see something on the terminal? ☺

C (Compiler) vs. Python (Interpreter)

- Let’s do the same thing using Python
  - $gedit helloPy.py
    - write print(“hello Python!”)
    - Save and close gedit
  - $python3 helloPy.py
  - You executed the python source file itself directly!

- In contrast, as for C,
  - You made a source file (helloC.c)
  - You compiled and made an executable image (helloC) – **one more step!**
  - And you executed the executable image, not the source code!

Code Analysis – Main

```
#include <stdio.h>

int main(void)
{
    printf("Hello C!\n");
    return 0;
}
```

- Function **main** is a special function in C
  - Where the program execution begins
  - Returns an integer
  - All C programs starts at the first statement of main and progresses until it returns
- A function block is not indicated by indentation but **brackets {}**
  - In C, indentation means nothing! But indentation is still important for readability
- All C statements except preprocessor macros end with semicolon **;**
  - One of the most frequent mistake for beginner
- **printf** is an output function, such as **print** in Python

Code Analysis – Preprocessor Macros

```
#include <stdio.h>

int main(void)
{
    printf("Hello C!\n");
    return 0;
}
```

- Preprocessor macro starts with "#" and does not end with ;
  - This is replaced by other C codes in the C preprocessor stage of compilation
  - #include <xx> or #include "xx" will be replaced by file xx's contents
    - #include <stdio.h> or #include "stdio.h"
    - Looks similar to **import** in Python but slightly different in that #include literally copies the content of the file

- xx.h is a **header file** that holds declarations useful among multiple source files
  - stdio.h has declarations of standard I/O functions, such as printf
  - It is necessary to include stdio.h to use printf

Code Analysis – Comment

- In C, comments do not start with #
  - # is used for preprocessor macros

- One line comment: //
- Multi-line comment: /* */

```
#include <stdio.h>                  // Preprocessor directive

int main(void)                      // Function main
{                                   // Function block marked by {…}
    printf("Hello C!\n");           /* Printing function */
    return 0;
}
```

Our Second C Program – Source Code

```
#include <stdio.h>                    // Preprocessor directive
#define MY_CONSTANT 10                // Preprocessor directive

int main(void)                        // Function main
{                                     // Function block marked by {…}
    printf("You defined a fixed value %d\n", MY_CONSTANT);   // Printing function
    return 0;
}
```

Code Analysis

- Another preprocessor macro **#define**
  - **#define X Y** makes X get substituted with Y, used to create fixed values within a program
  - #define NUMBER_OF_STUDENTS 25
  - #define COLOR_OF_EYES brown

- **printf** requires a **format string** in which we provide two things
  - text to print out
    - printf(“You defined a fixed value 10\n”);
  - Some specifications on how to print out program values within the text
    - printf(“You defined a fixed value **%d**\n”, **MY_CONSTANT**);
    - When you use %d in the format string, the value after the format string is embedded in the output as a decimal number in place of the %d

Questions?

Thanks!