What is Programming?

Wikipedia:

> Computer programming (often simply programming or coding) is the craft of writing a set of commands or instructions that can later be compiled and/or interpreted and then inherently transformed to an executable that an electronic machine can execute or “run”. Programming requires mainly logic, but has elements of science, mathematics, engineering, and — many would argue — art.
>
> In software engineering, programming (*implementation*) is regarded as one phase in a software development process.

- *logic:* programs are unambiguous
- *science:* programs reflect theories about the real world
- *mathematics:* programs can be complex — **abstraction!**
- *engineering:* systematic approach necessary
- *art/craft:* programs should be well-written for **human readers**

Historical Development of Programming Languages

Emphasis has changed:

– **from** making life easier for the **computer**

– **to** making life easier for the **programmer.**

Easier for the programmer means:

– Use languages that facilitate writing **error-free programs**

– Use languages that facilitate writing programs that are **easy to maintain**

Goal of language development:

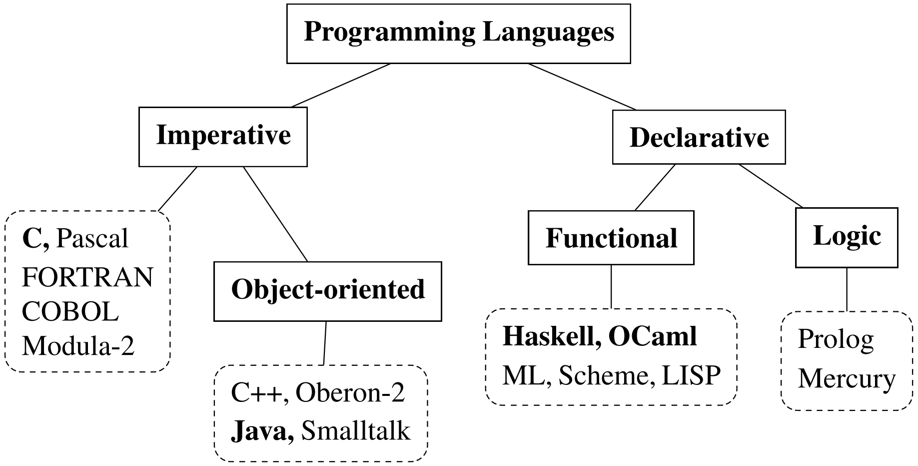
– Developers concentrate on **design** (or even just **specification**)

– Programming is trivial or handled by computer
(*executable specification languages, rapid prototyping*)

What Kinds of Programming Languages are There?

Imperative — “telling the machine what to **do**”

Declarative — “telling the machine what to **achieve**”



SE 2S03, Principles of Programming: Calendar Description

> **Fundamental concepts of imperative programming languages; (Assertion, Assignment, Control flow, Iteration, recursion, exceptions); Data representations; Basic concepts of operating systems; Composing and analyzing small programs.**

- *“Fundamental concepts”:* the execution model of imperative programming
- *“… operating systems”:* the execution environment of your programs
- *“… Assignment, Control flow, …”:* standard imperative program constructs
- *“Data representations”:*
  how to implement and use application-specific data structures
- *“Composing and analyzing small programs”:*
  systematic and principled approach to software development

Goals

- producing good-quality imperative programs in **C**
- **firm understanding** of the execution model
- **coding-level software engineering practices**
- common programming patterns and data structures
- **solving** programming **problems**

Skills

- **Programming** is a **skill**
- **Problem solving** is a **skill**
- **Debugging** is a **skill**
- …

Acquiring skills is not a spectator sport.

Language Learning

- **Syntax:** Rules of Grammar
  — ***how to write correct sentences/programs***
- **Semantics:** Meaning of Grammar
  — ***how to understand sentences/programs***
- **Vocabulary:** Words and their meanings
  — ***standard library***
- **Pragmatics**
  — ***how people use the language***
- **Practice, practice, practice**

Pair Programming

- Collaborative learning
- Two persons work together on one computer
- “Designated **driver**” at keyboard, actively creating code
- “**Reviewer**” constantly watching, identifying deficiencies
- **Switch roles** after designated period of time!
- **Joint ownership** of every single character in result
- Studies show improved quality of code and learning

Teaching Assistants

- Salvador Garcia Martinez
- Jinrong Han
- Scott West
- Shiqi (Steve) Cao

Grading

Surprise Quiz	**+2%**
Midterm 1	**10% / 20%**
Midterm 2	**10% / 20%**
Midterm 3	**10% / 20%**
Final	**40% – 70%**

$\mathit{midterm\_weight}[i] = (\mathit{midterm}[i] < \mathit{final})\ ?\ 10\ :\ 20$

SE 2S03 — Exercises and Tutorials

- Weekly **exercise sheets**
- Some exercise questions will be similar to exam questions
- **Complete** the exercises **before** the tutorial!!!
- Tutorials are intended for ***discussion*** of ***student solutions***
- **Practice** is **essential** for acquiring ***skills!***
- Three days before an exam is **too late for acquiring skills!**

Quiz 2005, Simulation of Program Execution

What is the output ot the following C program:

```
#include <stdio.h>                                   // Q2005_1.c
void main(void) {
  int n = 22;
  int k = 0, d = 1, s = 1;
  while ( s ≤ n ) {
    d = d + 2;
    s = s + d;
    k = k + 1;
    printf("k = %d\t d = %d\t s = %d\n", k, d, s);
  }
  printf("The result is %d.\n", k);
}
```

Can you state a general mathematical relation between k, d, and s that holds at each *printf* call inside the loop?

Quiz 2005, Program Execution with Assertions

```
#include <stdio.h>                               // Q2005_1_assert.c
#include <assert.h>                 // See Textbook 13.10
int square(int k) { return k * k; }
void main(void) {
  int n = 22;
  int k = 0, d = 1, s = 1;
  while ( s ≤ n ) {    //  Loop invariant:
                      assert( d == 2 * k + 1          &&   s == square(k + 1)  );
    d = d + 2;
                      assert( d == 2 * (k + 1) + 1    &&   s == square(k + 1)  );
    s = s + d;
                      assert( d == 2 * (k + 1) + 1    &&   s == square(k + 2)  );
    k = k + 1;
                      assert( d == 2 * k + 1          &&   s == square(k + 1)  );
    printf("k = %d\t d = %d\t s = %d\n", k, d, s);
  }                   assert( s > n                   &&   s == square(k + 1)  );
  printf("The result is %d.\n", k);
}
```


Assertions

- From The Free On-line Dictionary of Computing:

  assertion: An expression which, if false, indicates an *error*. Assertions are used for *debugging* by catching *can't happen* errors.

  can't happen: The traditional program comment for code executed under a condition that should never be true […] Although "can't happen" events are genuinely infrequent in production code, programmers wise enough to check for them habitually are often surprised at how frequently they are triggered during development and how many headaches checking for them turns out to head off.
- `#include <assert.h>`
- `void assert(scalar expression);`
- if the macro *NDEBUG* was defined when <*assert.h*> was last included, *assert*() generates no code
- if *expression* evaluates to false, *assert*(*expression*) prints an error message and terminates the program
- for debugging, not intended for users!

Quiz 2005, Testing for Sortedness

Design and implement a C function *is_sorted* that, for a fixed positive integer *N*, and an integer array of size *N* tests whether the elements of that array are in strictly ascending order.

Document all your assumptions and decisions!

- "testing whether" — return type bool most appropriate
- *N* is assumed to be defined outside our function

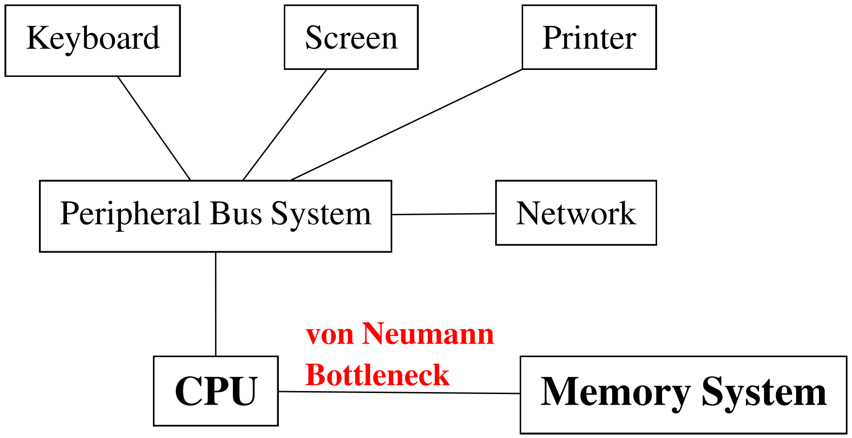
```
#include <stdbool.h>                                  // sorted.c
bool is_sorted(int array[]) {
                          assert( N > 0 ); // non-empty array
  int i;
  for ( i = 0; i < N – 1; i++) {
    if (array[i] ≥ array[i+1]) return false;
  }
  return true;
}
```


Read: Textbook Chapter 1

- **Computer Organization**
- **Machine Languages, Programming Languages**
- **Programming Language Translation**
- **C Standard Library**

  In analogy with natural languages:
  - *Grammar:* Syntax and semantics rules of the programming language C
  - *Vocabulary:* C standard library functions

How Does a Computer Work?



How Does a Computer Run Your Program?

- You edit `myprogram.c`
- You **compile:** `cc -o myprogram myprogram.c`
  - **Preprocessor** generates **preprocessed source** (`myprogram.i`)
  - **Compiler proper** generates **assembly program** (`myprogram.s`)
  - **Assembler** generates **object code** (`myprogram.o`)
  - **Linker** generates **executable** (`myprogram`)
  - You "**run**" it: `./myprogram`
    - **Operating system** generates a new process
    - **Dynamic linker** resolves references to shared libraries
    - **Loader** generates **executable in-memory image**
    - **CPU** runs machine code