Lecture 1: Intro and Machine Model

Logistics

Time

- Tuesday 10:00 am -11:40 am
- Thursday 10:00 am -11:40 am
- Friday 10:00 am -11:40 am (odd weeks)

Grading

Composition	Percentage
Quizzes	10%
Labs * 11	10%
Projects * 5	8% * 5 = 40%
Midterm	20%
Final	20%

Project Deadlines

Hours Late	Scaling Factor
(0, 24]	0.8
(24, 48]	0.6
(48, 72]	0.4
(72, ∞)	0

Textbook

- *C++ Primer*
- *Problem Solving with C++*
- *Data Structures and Algorithm Analysis*

Personal Recommendation

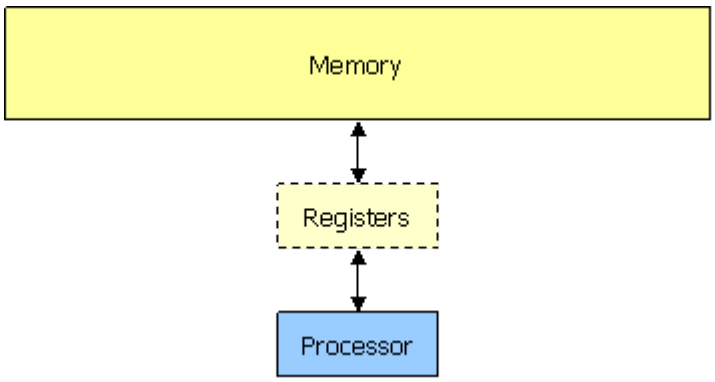
- *Expert C Programming: Deep C Secrets*
- *C Traps and Pitfalls*
- *Pointers on C*

Machine Models

Definition

An abstract machine, also called an abstract computer, is a theoretical model of a computer hardware or software system used in automata theory.

According to Stanford Sequoia Group, C's abstract machine model can be summarized as follows.



To fully understand this model, you will need to complete JI VE 370 / UM EECS 370, Computer Organization. For now, we will consider a simplified machine model, which consider memory only as a linear structure with slots for different objects.

Example

Consider the following example, what would be the value of `c` at the end of main function?

```
int example() {
    int a = 1;
    int b = 2;
    int c = a;
    a = 3;
    return c;
}
```

For each of the variables `a, b, c` declared within `example()`, the value of the variable is stored somewhere in the memory. To figure out what's going on in the program, we can sketch the machine model and observe the changes by lines.

At line 3, variable a and b are declared. They are stored in some location in the memory.	At line 4, variable c is declared, and stored in a different memory location.	At line 5, the value of variable a is modified, yet it does not affect the value of variable c.
/	c -> 1	c -> 1
/	/	/
b -> 2	b -> 2	b -> 2
/	/	/
a -> 1	a -> 1	a -> 3

Terminologies

- A *name* refers to some entity such as a *variable*, function, or type.
- A *variable* is a *name* that refers to an *object* in memory.
- A *declaration* is what introduces a *name* into the program and begins its *scope*.
- A *name* has a *scope*, which determines what region of code can use that *name* to refer to an entity.
    - In The following example, the scope of `c` begins at the declaration (line 6) and ends at the end of the function definition for `example()` (line 8).

```
1  #include <iostream>
2
3  int example() {
4      int a = 1;
5      int b = 2;
6      int c = a;
7      a = 3;
8      return c;
9  }
10
11 int main(){
12     int d = example();
13     std::cout << d << std::endl;
14     std::cout << c << std::endl;
15 }
```

If you attempt to use c outside `example()`, you will run into a compiler error.

```
1 error: 'c' was not declared in this scope
2   std::cout << c << std::endl;
3                ^
```

- At runtime, an *object* is a piece of data in memory, and it is located at some *address* in memory.

- An *object* has a *lifetime* during which it is legal to use that object. It is created at some point in time, and at some later point in time it is destroyed.
- The *storage duration* of an *object* determines its *lifetime*. There are three options:
  - *static*: lifetime = the whole program, controlled by compiler;
  - *automatic (local)*: lifetime = a particular scope, usually a block of code, controlled by compiler;
  - *dynamic*: lifetime is explicitly decided by the programmer.

Semantics

The semantics of an initialization or assignment `x = y` has 2 options.

- *value semantics*: modify the value of the object that x refers to;
- *reference semantics*: modify the object that x refers to.
  - C++ supports reference semantics only when initializing a new variable.
  - Since C++ only supports *reference semantics* in initialization, the association between a variable and a memory object can never be broken, except when the variable goes out of scope.

Consider the following program. What would be the output?

```
1  #include <iostream>
2
3  int main(){
4      int x = 1;
5      int z = 2;
6      int &y = x;
7      std::cout << x << "," <<  y << "," <<  z << std::endl;
8      x = 3;
9      std::cout << x << "," <<  y << "," <<  z << std::endl;
10     y = z;
11     std::cout << x << "," <<  y << "," <<  z << std::endl;
12     z = 4;
13     std::cout << x << "," <<  y << "," <<  z << std::endl;
14 }
```

To understand what's going on, we consider the machine model.

At line 6,	At line 8,	At line 10,	At line 12,
/	/	/	/
/	/	/	/
x, y -> 1	x, y -> 3	x, y -> 2	x, y -> 2
/	/	/	/
z -> 2	z -> 2	z -> 2	z -> 4

The output should be:

```
1,1,2
3,3,2
2,2,2
2,2,4
```

Coding Style

Good coding

- Meaningful variable names;
- Consistent indentation;
- Well tested, documented and commented;
- Rule of D-R-Y: Don't repeat yourself;
- High Coherence / Low coupling;
- Open for extension, but closed for modification.

The following is a good function example.

```
class Student{
    // represents a JI student.
    string name;
    string major;
    int stud_id;
    bool graduated;

public:
    Student(string name="default", string major="ece", int stud_id=0, bool graduated=false);
    // EFFECTS  : create a new student.

    bool compMajor(const Student &stud) const;
    // EFFECTS  : return true if "this" student has the same major as "stud",
    //            return false otherwise.

    bool hasGraduated() const;
    // EFFECTS  : return true if "this" student has graduated,
    //            return false otherwise.

    void changeMajor(string new_major);
    // MODIFIES : "major",
    // EFFECTS  : set "major" to "new_major".
};

```

Bad coding

- Vague variable names;
- Arbitrary indentation;
- Repeat part of your code or have codes of similar function;
- Long function. Say 200+ lines in a one function;

- Too many arguments. Say functions of 20+ arguments;
- ...There are many ways that you run into bad style.

The following is a bad function example. Readers are encouraged to modify the code into neat coding style.

```
int poly_evaluation(int x, int *coef, unsigned int d)
{
    int r = 0, p = 1;
for(int i = 0; i<= d; i++){
    r += coef[i] * p;
        p *= x;}
    return r;}
```

Lecture 2: Linux

Linux Filesystem

Directories in Linux are organized as a tree. Consider the following example:

```
/                       //root
├── home/               //users's files
    ├── username1
    ├── username2
    ├── username3
    └── ...
├── usr/                //Unix system resources
    ├── lib
    └── ...
├── dev/                //devices
├── bin/                //binaries
├── etc/                //configuration files
├── var/
└── ...
```

There are some special characters for directories.

- root directory: `/`
  - The top most directory in Linux filesystem.
- home directory: `~`
  - Linux is multi-user. The home directory is where you can store all your personal information, files, login scripts.
  - In Linux, it is equivalent to `/home/<username>`.
- current directory: `.`
- parent directory: `..`

Shell

The program that interprets user commands and provides feedbacks is called a *shell*. Users interact with the computer through the *shell*. For details interested, I would recommend a tour over UM EECS 201, Computer Pragmatics official website.

The general syntax for shell is `executable_file arg1 arg2 arg3 ...`.

- Arguments begin with `-` are called "switches" or "options";
  - one dash `-` switches are called short switches, e.g. `-l`, `-a`. Short switch always uses a single letter and case matters. Multiple short switches can often be specified at once. e.g. `-al` = `-a -l`.
  - Two dashes `--` switches are called long switches, e.g. `--all`, `--block-size=M`. Long switches use whole words other than acronyms.
- There are exceptions, especially, `gcc` and `g++`.

Useful Commands

The following are some useful commands. Try them.

- `pwd`: print working directory.
- `cd`: change directory.
  - For example, `cd ../` brings you to your parent directory.
- `ls`: list files and folders under a directory.
  - Argument:
    - If the argument is a directory, list that directory.
    - If the argument is a file, show information of that specific file.
    - If no arguments are given, list working directory.
  - Options:
    - `-a` List hidden files as well. Leading dot means "hidden".
    - `-l` Use long format. Each line for a single file.
- `tree`: recursively list the directory tree.
- `mkdir`: make directory.
- `rm`: remove.
  - This is an extremely dangerous command. See the famous bumblebee accident.
  - `-i`: prompt user before removal.
  - `-r` Deletes files/folders recursively. Folders requires this option.
  - `-f` Force remove. Ignores warnings.
- `rmdir`: remove directory.
  - Only empty directories can be removed successfully.
- `touch`: create an new empty file.
  - Originally designed to change time stamp though.
- `cp`: copy.
  - Takes 2 arguments: `source` and `dest`.
  - Be very careful if both source and destination are existing folders.
  - `-r` Copy files/folders recursively. Folders requires this option.
- `mv`: move.

    - Takes 2 arguments: `source` and `dest`.
    - Be very careful if both source and destination are existing folders.
    - Can be used for rename by making `source` = `dest`.
- `cat`: concatenate.
    - Takes multiple arguments and print their content one by one to `stdout`.
- `less`: prints the content from its `stdin` in a readable way.
- `diff`: compare the difference between 2 files.
    - `-y` Side by side view;
    - `-w` Ignore white spaces.
- `nano` and `gedit`: command line file editor.
    - Advanced editors like `vim` and `emacs` can be used also.
    - If you try `vim`: just in case you get stuck in this beginner-unfriendly editor...the way to exit `vim` is to press `ESC` and type `:q!`. See how do i exit vim.
- `grep`: filter input and extracts lines that contains specific content.
- `echo`: prints its arguments to `stdout`.

IO Redirection

Most command line programs can accept inputs from standard input and display their results on the standard output.

- `executable < input` Use input as `stdin` of executable.
- `executable > output` Write the `stdout` of executable into output.
    - Note this command always truncates the file.
    - File will be created if it is not already there.
- `executable >> output` Append the `stdout` of executable into output.
    - File will be created if it is not already there.
- `exe1 | exe2` Pipe. Connects the `stdout` of exe1 to `stdin` of exe2.

They can be used in one command line. Consider `executable < input > output`. What is this line for?

File Permissions

The general syntax for long format is `<permission> <link> <user> <group> <file_size> <modified_time> <file_name>`.

```
$ ls -l
total 88
-rw-r--r--  1 marstin marstin   13 Jan 29 16:32 ans.txt
drwxr-xr-x  2 marstin marstin  4096 Feb 21 21:27 bigWigLiftOver
drwxr-xr-x  3 marstin marstin  4096 Oct  2  2019 Desktop
drwxr-xr-x 16 marstin marstin  4096 May  9 15:14 Documents
drwxr-xr-x  2 marstin marstin  4096 May 11 01:31 Downloads
-rw-r--r--  1 marstin marstin  8980 Oct  2  2019 examples.desktop
drwxr-xr-x  2 marstin marstin  4096 Feb 14 22:13 igv
-rw-r--r--  1 marstin marstin 11733 Jan 17 14:33 index.html
-rw-------  1 marstin marstin   464 Feb 10 18:47 key1
```

```
-rw-r--r--  1 marstin marstin   99 Feb 10 18:47 key1.pub
drwxr-xr-x  2 marstin marstin  4096 May 11 01:30 Music
-rw-r--r--  1 marstin marstin   0 Jan 29 16:32 out.txt
drwxr-xr-x  2 marstin marstin  4096 Nov 10  2019 Pictures
drwxr-xr-x  2 marstin marstin  4096 Oct  2  2019 Public
drwxr-xr-x 10 marstin marstin  4096 Feb 16 21:29 pycharm-2019.3.1
drwxr-xr-x  3 marstin marstin  4096 Oct  3  2019 PycharmProjects
drwxr-xr-x  2 marstin marstin  4096 Oct  2  2019 Templates
drwxr-xr-x  2 marstin marstin  4096 Oct  2  2019 Videos
drwxr-xr-x  8 marstin marstin  4096 Feb 17 17:03 WebStorm-193.6494.34
```

File permission

In total, 10 characters:

- char 1: Type. `-` for regular file and `d` for directory.
- char 2-4: Owner permission. `r` for read, `w` for write, `x` for execute.
- char 5-7: Group permission. `r` for read, `w` for write, `x` for execute.
- char 8-10: Permission for everyone else. `r` for read, `w` for write, `x` for execute.

Give executable permission to a file: `chmod +x <filename>`.

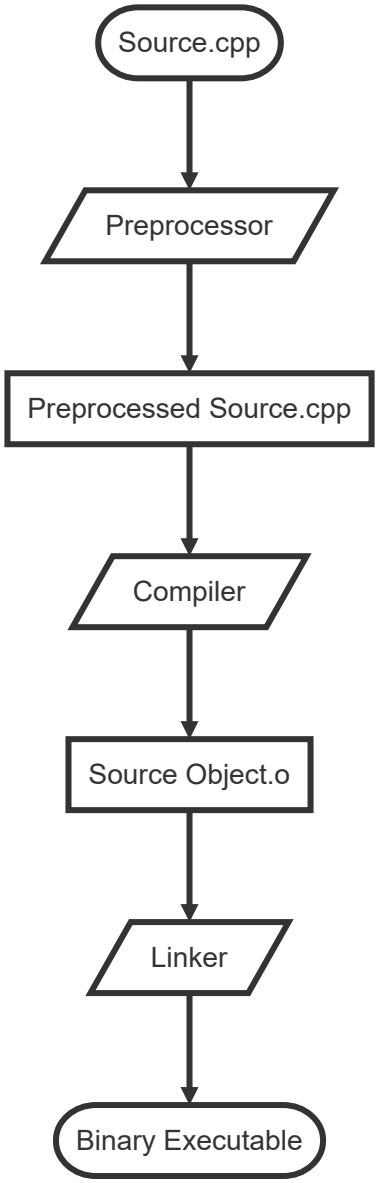
Lecture 3: Developing Programs

Compilation

Compilation Process

For now just have a boarder picture of what's going on. Details will be discussed in the upper level courses.

- **Preprocessing** in `g++` is purely textual.
  - `#include` simply copy the content
  - Conditional compilation (`#ifdef`, `#ifndef`, `#else`, ...) directives simply deletes unused branch.
- **Compiler**: Compiles the `.c` / `.cpp` file into object code.
  - Details of this part will be discussed in UM EECS 483, Compiler Structure. Many CE students with research interest in this field also take EECS 583, Advanced Compiler.
- **Linker**: Links object files into an executable.
  - Details of this part will be discussed in JI VE 370 / UM EECS 370, Computer Organization, where your project 2 is to write a linker (I personally hate this project because it is extremely hard to debug).



g++

Preprocessor, compiler and linker used to be separate. Now `g++` combines them into one, thus is an all-in-one tool. By default, `g++` takes source files and generate executable. You can perform individual steps with options.

Compile in one command: `g++ -o program source.cpp`.

In steps:

- Compile: `g++ -c source.cpp`;
- Link: `g++ -o program source.o`.

Some options for g++:

- `-o <out>` Name the output file as out. Outputs a.out if not present.
- `-std=` Specify C++ standard.
    - Recommend `-std=c++11`.

- `-Wall` Report all warnings. Do turn `-Wall` on during tests. **Warnings are bugs.**
- `-O{0123}` Optimization level.
  - `-O2` is the recommended for release.
- `-c` Only compiles the file (Can not take multiple arguments).
- `-E` Only pre-processes the file (Can not take multiple arguments).

Header Guard

Everything in C++ is allowed to have at most one definition during compilation. Problem arises for multiple inclusion: what will happen during preprocessing if `point.h` is included in both `square.cpp` and `circle.cpp`?

```
// point.h

struct Point {
    double x, y;
};
```

That's why we need a *header guard*.

```
// point.h
#ifndef POINT_H
#define POINT_H

struct Point {
    double x, y;
};

#endif
```

Note: Be careful when naming you header. Double check if a library shares the same name. This may lead to unexpected errors when using header guards, which are extremely hard to detect.

Build

Why do we need a build system?

- Build process is complicated, avoid type every command.
- Project have dependence, need to manage dependence
- Compile minimum amount of code possible upon update.
- Many other reasons, abstract out actual compiler, compile for different platform / target.

GNU Make and Makefile

Makefile is made up of *targets*. A *target* can depend on other *target*, or some files.

```
target: prerequisites
    recipe # <- actual tab character, not spaces!```
```

I have uploaded some demos on piazza. It's also available here. The following Makefile is from demo1.

```
all: main

main: main.o add.o minus.o
    g++ -o main main.o add.o minus.o

main.o: main.cpp
    g++ -c main.cpp

add.o: add.cpp
    g++ -c add.cpp

minus.o: minus.cpp
    g++ -c minus.cpp

clean:
    rm -f main *.o

.PHONY: all clean
```

Try to run with `make`, `make all` and `make clean` on your own system and observe what's going on in your working directory.

If you are interested in further details, you can have a look over EECS 201 Lecture 7.

CMake

CMake is a modern make system used by Clion and many other projects. It is very flexible, reliable and is a cross platform solution. If you use Clion as your IDE you should at least know some basics of it.

Example `CMakeList.txt`:

```
cmake_minimum_required(VERSION 3.8)
project(surfaces)
set(CMAKE_CXX_STANDARD 11)
find_package(OpenCV REQUIRED)
set(CMAKE_CXX_FLAGS_RELEASE
    "${CMAKE_CXX_FLAGS_RELEASE} -march=native
    -ffast-math")

add_executable(main main.cpp utils.cpp)
target_link_libraries(main ${OpenCV_LIBS})
```

Credit

SU2019 & SU2020 VE280 Teaching Groups.

VE 280 Lecture 1-3 and EECS 201 Lecture 7