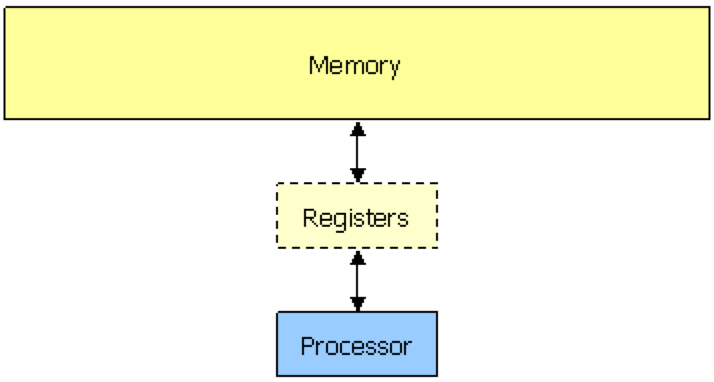
Lecture 1: Intro and Machine Model

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).

    ```
    #include <iostream>

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

    int main(){
        int d = example();
        std::cout << d << std::endl;
        std::cout << c << std::endl;
    }
    ```

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

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

- 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?

```
#include <iostream>

int main(){
    int x = 1;
    int z = 2;
    int &y = x;
    std::cout << x << "," <<  y << "," <<  z << std::endl;
    x = 3;
    std::cout << x << "," <<  y << "," <<  z << std::endl;
    y = z;
    std::cout << x << "," <<  y << "," <<  z << std::endl;
    z = 4;
    std::cout << x << "," <<  y << "," <<  z << std::endl;
}
```

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>`.

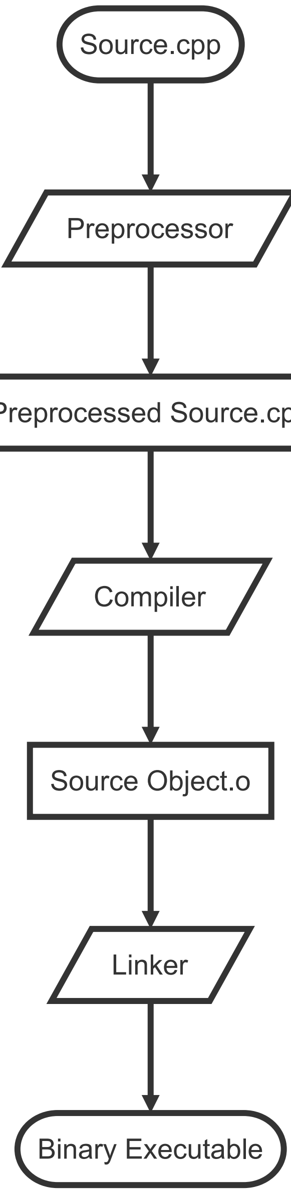
Lecture 3: Developing Programs

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.

Lecture 4: C++ Basics

Value Category

L/R-value refers to memory location which identifies an object. A simplified definition for beginners are as follows.

- L-value may appear as **either left hand or right hand** side of an assignment operator(=).
  - In memory point of view, an l-value, also called a *locator value*, represents an object that occupies some identifiable location in memory (i.e. has an address).
  - Any non-constant variable is lval.
- R-value may appear as **only right hand** side of an assignment operator(=).
  - Exclusively defined against L-values.
  - Any constant is an rval.

Consider the following code.

```
int main(){
    int arr[5] = {0, 1, 2, 3, 4};
    int *ptr1 = &arr[0];
    int *ptr2 = &(arr[0]+arr[1]);
    cout << ptr1 << endl;
    cout << ptr2 << endl;
    arr[0] = 5;
    arr[0] + arr[1] = 5;
}
```

The compiler will raise errors.

```
...\main.cpp:7:32: error: lvalue required as unary '&' operand
     int *ptr2 = &(arr[0]+arr[1]);
                                ^
...\main.cpp:11:21: error: lvalue required as left operand of assignment
     arr[0] + arr[1] = 5;
```

Function Declaration and Definition

Consider the following codes.

```
// Function Declaration
int getArea(int length, int width);

int main()
{
    cout << getArea(2, 5) << endl;
}

// Function Definition
int getArea(int length, int width)
{
    return length*width;
}
```

Function Declaration

Function declaration (prototype) shows how the function is called. It must appear in the code before function can be called.

A Function declaration `int getArea(int length, int width);` tells you about:

- Type signature:
  - Return type: `int`;
  - # arguments: 2;
  - Types of arguments: `int`*2;
- Name of the function: `getArea`;
- Formal Parameter Names (*): `length` and `width`.

However, formal parameter names are not necessary. Try to replace:

```
int getArea(int length, int width);
```

with:

```
int getArea(int l, int w);
```

or even:

```
int getArea(int, int);
```

The program still works. Yet, it is considered good coding style to keep the formal parameter names, so that your potential collaborators can understand what the function is for.

Function Definition

Function definition describes how a function performs its tasks. It can appear in the code before or after function can be called.

```
int getArea(int length, int width)  // ----> Function Header
{                                   // -+
    return length*width;            //  |--> Function Body
}                                   // -+
```

Argument Passing Mechanism

Consider the following example.

```
void pass_by_val(int x){
    x = 2;
}

void pass_by_ref(int &x){
    x = 2;
}

void mixed(int x, int &y){
    x = 3;
    y = 3;
}

int main(){
    int y = 1;
    int z = 2;
    pass_by_val(y);
    pass_by_ref(z);
    cout << y << " " << z << endl;
    mixed(y, z);
    cout << y << " " << z << endl;
}
```

The output of the above code is

```
1 2
1 3
```

This example demonstrates the 2 argument passing mechanism in C++:

- Pass by Value;
- Pass by Reference.

The difference of above mechanisms can be interpreted from the following aspects:

- Language point of view: reference parameter allows the function to change the input parameter.
- Memory point of view:
    - Pass-by-reference introduce an extra layer of indirect access to the original memory object. In fact, many compilers implement references with pointers.

    - Pass-by-value needs to copy the argument.
    - Can both expensive, in terms of memory and time.

Choosing argument passing methods wisely:

- Pass atomic types by value (`int`, `float`, `char*`...).
    - `char` is 1B, `int32` is 4B, `int64` is 8B, and a pointer is 8B (x64 System).
- Pass large compound objects by reference (`struct`, `class`...).
    - For `std` containers, passing by value costs 3 pointers, but passing by reference costs only 1;
    - What about structures/classes you created?

Arrays and Pointer

Your familiarity of arrays and pointers is assumed in this course. Test yourself with the following examples.

Pointers

What is the output of the following example?

```
int main(){
    int x = 1;
    int y = 2;
    int *p = &x;
    *p = ++y;
    p = &y;
    y = x++;
    cout << x << " " << y << " " << *p << endl;
}
```

Arrays

What is the output of the following example?

```
void increment(int arr[], int size){
    for (int i = 0; i < size; ++i){
        (*(arr + i))++; // correct
        //*(arr + i)++; // wrong
    }
}

int main(){
    int arr[5] = {0, 1, 2, 3, 4};
    increment(arr, 5);
    for (int i = 0; i < 5; ++i){
        cout << *(arr + i);
    }
    cout << endl;
}
```

Two important things to keep in mind here:

- Arrays are naturally passed by pointer;

- Conversion formula between arrays and pointers: `*(arr + i) = arr[i]`

References and Pointers

L-vals always corresponds to a fixed memory region. This gives rises to a special construct called references.

Your familiarity of **non-constant references** is assumed in this course. Test yourself with the following example. What is the output?

```
int main(){
    int a = 1;
    int b = 5;
    int *p1 = &a;
    int *p2 = &b;
    int &x = a;
    int &y = b;
    p2 = &a;
    x = b;
    a++;
    b--;
    cout << x << " " << y << " " << *p1 << " " << *p2 << endl;
}
```

Non-constant references must be initialized by a variable of the same type, and cannot be rebounded. Try resist the temptation to think reference as an alias of variables, but remember they are alias for the memory region. References must be bind to a memory region when created: there is no way to re-bind of an existing reference.

Structures

Your familiarity of structures is assumed in this course.

```
struct Student{
    // represents a JI student.
    string name;
    string major;
    long long stud_id;
    bool graduated;
};

int main(){
    // Initialize a structrue
    struct Student s = {"martin", "undeclared", 517370910114, false};
    struct Student *ptr = &s;

    // Use . and -> notation to access and update
    cout << s.name << " " << ptr->stud_id << endl;
    s.major = "ece";
    ptr->graduated = true;
}
```

Two facts to take away here:

- `struct` is in fact totally the same as `class`, instead the default is `public`.

- To save memory, place larger attributes first. You will understand and get used to this when you take VE/EECS 281 and VE/EECS 370.

Undefined Behaviors

Undefined behaviors (UB) are program whose output depends on a specific platform, or a specific implementation of the compiler. Therefore, the outcome/answer of a process may be YES, NO...or UB.

Why UBs are still there throughout the years? It's not that the committee doesn't know how to eliminate them, but they leave room for compiler optimizations.

For a programmer, keep in mind that:

- It's an absolute waste of time trying to figure out what will happen given an code that contains UB.
- It's dangerous and to write code that contains UB, and it is your job to avoid them.

Any (zero or more) of the following may happen if you trigger any of UBs:

- Compiler refuse to compile;
- Compiler compiles but warns (Again, warnings are bugs);
- The compiler compiles silently, but...
    - Program crashes when executed;
    - Your program works out, but the output is wrong;
    - The compiler deletes your files in the systems;
    - Your program works perfectly on your computer but gets rejected by the JOJ.

Common examples of UBs:

- Integer overflow:

  ```
  int x = INT_MAX;
  x++; // UB
  ```

  Many compilers would work out `x = -2147483648`, but that is not guaranteed.

- Dereferencing `nullptr`:

  ```
  int* x = nullptr;
  *x = 1; // UB
  ```

  Most compilers compile it, but the program crashes. Yet, it is not guaranteed that the program crashes.

- Array out-of-bound:

  For most platforms, the following works perfectly, but is bad and not guaranteed to.

  ```
  int main(){
      int x[5] = {0};
      x[5] = 1; // UB
      cout << x[5] << endl;
  }
  ```

  Merely taking the address is UB.

```
int main(){
    int x[5] = {0};
    int *p = &(x[5]); // UB
    *p = 1;
    cout << *p << endl;
}
```

- Dangling references

  You could still get correct value, or not.

```
int main(){
    auto x = new int[10];
    x[3] = 1;
    delete[] x;
    cout << x[3]; // UB
}
```

There are more UBs out there. Please be aware and remember them when you encounter any.

Lecture 5: `const` Qualifier

Constant Modifier

Immutability of `const`

Whenever a type something is `const` modified, it is declared as immutable.

```
struct Point{
    int x;
    int y;
};

int main(){
    const Point p = {1, 2};
    p.y = 3; // compiler complains
    cout << "(" << p.x << "," << p.y << ")" << endl;
}
```

Remember this immutability is enforced by the compiler at compile time. This means that `const` in fact does not guarantee immutability, it is an intention to. The compiler does not forbid you from changing the value intentionally. Consider the following program:

```
int main(){
    const int a = 10;
    auto *p = const_cast<int*>(&a);
    *p = 20;
    cout << a << endl;
}
```

What's the output? This is actually an UB. It depends on your compiler and platform whether the output is 10 or 20. Therefore, be careful of undefined behaviors from type casting, especially when `const` is involved.

Const Global

Say when we declare a string for jAccount username, and want to ensure that the max size of the string is 32.

```
int main(){
    char jAccount[32];
    cin >> jAccount;
    for (int i = 0; i < 32; ++i){
        if (jAccount[i] == '\0'){
            cout << i << endl;
            break;
        }
    }
}
```

This is bad, because the number 32 here is of bad readability, and when you want to change 32 to 64, you have to go over the entire program, which, chances are that, leads to bugs if you missed some or accidentally changed 32 of other meanings.

This is where we need constant global variables.

```
int MAX_SIZE = 32;

int main(){
    char jAccount[MAX_SIZE];
    cin >> jAccount;
    for (int i = 0; i < MAX_SIZE; ++i){
        if (jAccount[i] == '\0'){
            cout << i << endl;
            break;
        }
    }
}
```

Note that const globals must be initialized, and cannot be modified after. For good coding style, use UPPERCASE for const globals.

Const References

Const Reference vs Non-const Reference

Define a constant reference: `const int& iref`

There is something special about const references:

- Const reference are allowed to be bind to right values;
- Normal references are not allowed to.

Consider the following program. Which lines cannot compile?

```
int main(){
    // which lines cannot compile?
    int a = 1;
    const int& b = a;
    const int c = a;
    int &d = a;
    const int& e = a+1;
    const int f = a+1;
    int &g = a+1;       // x
    b = 5;              // x
    c = 5;              // x
    d = 5;
}
```

Normally, if a const reference is bind to a right value, the const reference is no difference to a simple const. In above example, you may consider line 4 and 5 identical.

Argument Passing

Then why do we need const references? See the following example.

```
class Large{
    // I am really large.
};

int utility(const Large &l){
    // ...
}
```

Reasons to use a constant reference:

- Passing by reference -> avoids copying;
- `const` -> avoids changing the structure;
- const reference -> rvals can be passed in.
    - That's why we don't use a const pointer.

Const Pointers

There are many ways to define a `const` reference, which are NOT identical. Personally my trick of identification is to read from the right hand side.

- Pointer to Constant (PC): `const int *ptr`;
- Constant Pointer (CP): `int *const ptr;`
- Constant Pointer to Constant (CPC): `const int *const ptr;`

Type	Can change the value of pointer?	Can change the object that the pointer points to?
Pointer to Constant	Yes	No
Constant Pointer	No	Yes
Constant Pointer to Constant	No	No

See the following example. Which lines cannot compile?

```
int main(){
    // which lines cannot compile?
    int a = 1;
    int b = 2;
    const int *ptr1 = &a;
    int *const ptr2 = &a;
    const int *const ptr3 = &a;
    *ptr1 = 3;      // x
    ptr1 = &b;
    *ptr2 = 3;
    ptr2 = &b;      // x
    *ptr3 = 3;      // x
    ptr3 = &b;      // x
}
```

Const and Types

Type Definition

When some compound types have long names, you probably don't want to type them all. This is when you need `typedef`.

The general rule is `typedef real_name alias_name`.

For example, you probably want:

```
typedef std::unordered_map<std::string, std::priority_queue<int,
std::vector<int>, std::greater<int> > > string_map_to_PQ;
```

Mind that you can define a type based on a defined type. See following example. Which lines cannot compile?

```
typedef const int_ptr_t Type1;
typedef const_int_t* Type2;
typedef const Type2 Type3;

int main(){
    // which lines cannot compile?
    int a = 1;
    int b = 2;
    Type1 ptr4 = &a;
    Type2 ptr5 = &a;
    Type3 ptr6 = &a;
    *ptr4 = 3;
    ptr4 = &b;      // x
    *ptr5 = 3;      // x
    ptr5 = &b;
    *ptr6 = 3;      // x
    ptr6 = &b;      // x
}
```

Type Coercion

The following are the **Const Prolongation Rules**.

- `const type&` to `type&` is incompatible.
- `const type*` to `type*` is incompatible.
- `type&` to `const type&` is compatible.
- `type*` to `const type*` is compatible.

In one word, only from non-const to const is allowed.

Consider the following example:

```
void reference_me(int &x){}
void point_me(int *px){}
void const_reference_me(const int &x){}

void main() {
    int x = 1;
    const int *a = &x;
    const int &b = 2;
    int *c = &x;
    int &d = x;

    // which lines cannot compile?
    int *p = a;             // x
    point_me(a);            // x
    point_me(c);
    reference_me(b);        // x
    reference_me(d);
    const_reference_me(*a);
    const_reference_me(b);
    const_reference_me(*c);
    const_reference_me(d);
}
```

Lecture 6: Procedural Abstraction

Abstraction

Abstraction is the principle of separating what something is or does from how it does it.

- Provide details that matters (what)
- Eliminate unnecessary details (how)

There are 2 types of abstractions:

- Data Abstraction
- Procedural Abstraction

The product of *procedural abstraction* is a procedure, and the product of *data abstraction* is an abstract data type (ADT).

Procedural Abstraction

Properties

Functions are mechanism for defining procedural abstractions.

Difference between *abstraction* and *implementation*:

- Abstraction tells what and implementation tells how.
- The same abstraction could have different implementations.

There are 2 properties of proper procedural abstraction *implementation*:

- Local: the implementation of an abstraction does not depend of any abstraction implementation.
- Substitutable: Can replace a correct implementation wit another.

Composition

There are 2 parts of an abstraction:

- Type signature
- Specification

There are 3 clauses in the specification comments:

- REQUIRES: preconditions that must hold.
- MODIFIES: how inputs will be modified.
- EFFECTS: what the procedure is computing.

```
void log_array(double arr[], size_t size)
// REQUIRES: All elements of `arr` are positive
// MODIFIES: `arr`
// EFFECTS: Compute the natural logarithm of all elements of `arr`
{
    for (size_t i = 0; i < size; ++i){
        arr[i] = log(arr[i]);
    }
}
```

Completeness of functions are defined as follows:

- If a function does not have any `REQUIRES` clauses, then it is valid for all inputs and is complete.
- Else, it is partial.
- You may convert a partial function to a complete one.

Lecture 7: Functions and Recursion

Function Pointers

Definitions

Variable that stores the address of functions are called *function pointers*. By passing them around we could pass functions into functions, return them from functions, and assign them to variables.

Consider when you only need to change one step in a larger function, like changing “adding” all elements in the matrix to “multiplying” all the elements. It is a waste of time and space to repeat the code, thus programmers would consider using a function pointer.

See the following example:

```
int max(int arr[], size_t size) {
    int maximum = -INT_MAX;
    for (size_t i = 0; i < size; i++){
        maximum = max(maximum, arr[i]);
    }
    return maximum;
}

int min(int arr[], size_t size) {
    int minimum = INT_MAX;
    for (size_t i = 0; i < size; i++){
        minimum = min(minimum, arr[i]);
    }
    return minimum;
}

int avg(int arr[], size_t size) {
    int average = 0;
    for (size_t i = 0; i < size; i++){
        average += arr[i];
    }
    average /= size;
    return average;
}

int get_stats(int arr[], size_t size, int (*fun)(int[], size_t)){
    return fun(arr, size);
}

int main(){
    int arr[] = {1,2,3,4,5};
    cout << get_stats(arr, 5, min) << endl;
    cout << get_stats(arr, 5, max) << endl;
    cout << get_stats(arr, 5, avg) << endl;
}
```

You may consider functions as a bunch of numbers in the memory, and we can refer to the function by referring to the numbers. These numbers has an address in the memory as well, so we could use that address to refer to the function.

Properties

Given the function declarations,

```
void foo();

int foo(int x, int y);

int foo(int, int);

int *foo(int, char*);

char* foo(int[], int);
```

Write the corresponding function pointers.

```
void (*bar)();

int (*bar)(int, int);

int (*bar)(int, int);

int *(*bar)(int, char*);

char *(*bar)(int[], int);
```

Function pointers are invariant under `*`: Dereferencing a function pointer still gives back a function pointer. Consider the following example.

Given that:

```
int max(int x, int y) { return x > y ? x : y;}
```

What are the values of `a`, `b` and `c`?

```
int (*cmp)(int, int) = max;
int a = cmp(1, 2);
int b = (*cmp)(1, 2);
int c = (*******cmp)(1, 2);
```

It's possible to have an array of function pointers. Consider the following example:

```
int add (int x, int y) {return x + y;}
int multiply(int x, int y) {return x * y;}

int (*fun[])(int, int) = {add, multiply};

int main() {
    int op = 1, x = 0, y = 0;
    cout << "Select your operation (1 = add, 2 = multiply): ";
    cin >> op;
    cout << "Numbers: ";
    cin >> x >> y;
    cout << "ANS = " << fun[op-1](x, y) << endl;
}
```

Function Call Mechanism

To fully understand function call mechanism, you need to understand the memory organization of system, which, is beyond the scope of VE280 (Learn more about it in VE370 / EECS370). For now, you only have to get familiar with the concept of *call stack*.

At each function call, the program does the following:

1. Evaluate the actual arguments.

   For example, your program will convert `y = add(1*5, 2+2)` to `y = add(5, 4)`.

2. Create an activation record (stack frame)

   The activation record would hold the formal parameters and local variables.

   For example, when `int add(int a, int b) { int result = a+b; return result; }` is called, your system would create an activation record to hold:

   - `a`, `b` (formal parameters)
   - `result` (local parameters)

3. Copy the actual values from step 1 to the memory location that holds formal values.
4. Evaluate the function locally.
5. Replace the function call with the result.

   For the same example, your program will convert `y = add(5, 4)` to `y = 9`.

6. Destroy the activation record.

Typically, we come across situations with multiple functions are called and multiple activation records are to be maintained. To store these records, your system applies a data structure called *stack*.

A *stack* is a set of objects. When popping out an element from it, you always get the last element that is pushed into it. This property is referred to as *last in first out* (LIFO).

Consider the following example.

```
int add_2 (int x, int y) {return x + y;}

int add_3 (int x, int y, int z) = {return add_2(x, add_2(y, z));};

int main() {
    int x = 1, y = 2, z = 3;
    int a = add_3(x, y, z);
    cout << a << endl;
}
```

Readers are encouraged to simulate how stack frames are generated and destroyed on paper.

Recursions

Recursion simply means to refer to itself.

For any recursion problem, you may focus on the 2 compositions:

- Base cases: There is (at least) one “trivial” base or “stopping” case.
- Recursive step: All other cases can be solved by first solving one smaller case, and then combining the solution with a simple step.

A trivial example would be:

```
int factorial (int n) {
// REQUIRES: n >= 0
// EFFECTS:  computes n!
    if n == 0
        return 1;                   // Base case
    else
        return n * factorial(n-1);  // Recursive step
}
```

A recursive function could be simply recursive or tail recursive.

Recall from the previous section that when a function is called, it gets an activation record.

- A function is simply recursive if it generates a stack frame for each recursive call.
- A function is tail recursive if there is no pending computation at each recursive step.
    - It "re-use" the stack frame rather than create a new one.

The above `factorial` function is an example of a simply recursive function. It can also be redesigned as a tail recursive function.

```
int factorial(int n, int result = 1) {
    if (n == 0)
        return result;
    else
        return factorial(n -1, res * n);
}
```

Sometimes it is hard to implement a recursive function directly due to lack of function arguments. In this case, you may find a *helper* function useful.

Instead of

```
recursion(){
    ...
    recursion()
    ...
}
```

One may consider:

```
recursion(){
    ...
    recursion_helper()
    ...
}

recursion_helper(){
    ...
    recursion_helper()
    ...
}
```

An example is to check palindromes.

```
bool is_palindrome_helper(string s,
int begin, int end)
// EFFECTS: returns true if the subtring of s that
// starts at `begin` and ends at `end`
// is a palindrome.
{
    if(begin >= end)
        return true;
    if(s[begin] == s[end])
        return is_palindrome_helper(s, begin+1, end-1);
    else
        return false;
}

bool is_palindrome(string s)
// EFFECTS: returns true if s is a palindrome.
{
    return is_palindrome_helper(s, 0, s.length()-1);
}
```

Lecture 8: Enumeration Types

Why `enum`

`enum` is a type whose values are restricted to a set of values.

Consider the example in your project 2.

```
enum Error_t {
    INVALID_ARGUMENT,
    FILE_MISSING,
    CAPACITY_OVERFLOW,
    INVALID_LOG,
};
```

The advantages of using an `enum` here are:

- Compared to constant `strings`: more efficient in memory.
    - Even the minimum size of a `std::string` is larger than an `int`.
- Compared to constant `int`s or `char`s: more readable and limit valid value set.
    - Numbers or single chars are less readable than a string;
    - When passed to a function, you will always have to specify which integers/chars are valid in `REQUIRES`.

Properties

Enum values are actually represented as an integer types (`int` by default). If the first enumerator does not have an initializer, the associated value is zero.

```
enum Error_t {
    INVALID_ARGUMENT,   // 0
    FILE_MISSING,       // 1
    CAPACITY_OVERFLOW,  // 2
    INVALID_LOG,        // 3
};
```

Making use of this property, one usually writes:

```
int main(){
    const string error_info[] = {
            "The argument is invalid!",
            "The file is missing!",
            "Reaching the maximum capacity!",
            "The log is invalid!"
    };
    enum Error_t err = INVALID_LOG;
    cout << error_info[err] << endl;
}
```

Here are a few important properties of `enum`s.

Comparability:

Enum values are comparable. This means that you can apply <, >, ==, >=, <=, != on enum values.

Designable:

The constant values can also be chosen by programmer.

Consider the following example, what would the output be?

```
enum QAQ {
    a, b, c = 3, d, e = 1, f, g = f+d
};

int main(){
    QAQ err = g;
    cout << static_cast<int>(err) << endl;
}
```

Note that if you directly use `cout << err << endl`, the compiler would complain. You will need to statically cast the enum value to a printable type like `static_cast<int>`.

The correct answer would be:

```
enum QAQ {
    a,      // 0
    b,      // 1
    c = 3,  // 3
    d,      // 4
    e = 1,  // 1
    f,      // 2
    g = f+d // 6
};
```

The underlying integer type can be modified as well. If the range of `int` is too big or too small, you can also use a different underlying integer type.

```
enum Error_t : char {
    INVALID_ARGUMENT,
    FILE_MISSING,
    CAPACITY_OVERFLOW,
    INVALID_LOG,
};
```

Note that `char` is an integer type (Recall your knowledge about ASCII).

Enum Class

You may also define the `enum` as an enum class, *i.e.*

```
enum class Error_t {
    INVALID_ARGUMENT,   // 0
    FILE_MISSING,       // 1
    CAPACITY_OVERFLOW,  // 2
    INVALID_LOG,        // 3
};
```

When defining an variable, do not forget about `Error_t::`.

```
enum Error_t err = Error_t::INVALID_LOG;
```

Lecture 9: Program Argument

Program arguments are passed to the program through `main()`:

```
#include <iostream>
using namespace std;
int main(int argc, char* argv[]) {
    cout << argc << endl;
    for(int i = 0; i < argc; ++i){
        cout << argv[i] << endl;
    }

    return 0;
}
```

`argv`: all arguments including program name; an array of C-strings

`argc`: the number of strings in `argv`

Call:

```
g++ -o test test.cpp
./test ve280 midterm
```

Output:

```
3
./test
ve280
midterm
```

Manipulate C-strings:

```
int integer = atoi(argv[1]);

char carr_err[strlen(argv[1])] = argv[1]; // incorrect
char *carr = new char[strlen(argv[1])+1];
strcpy(carr, argv[1]);
delete[] carr;

std::string str = argv[1];
```

Many Linux commands are programs that take arguments:

```
diff file1 file2
g++ test.cpp
```

You could take a look at `/usr/bin` and `/bin` directories if you like.

Lecture 10: I/O Streams

cin and cout

>> and <<

In C++, streams are **unidirectional**, which means only `cin >>` and `cout <<`.

If we look into `cin`, it's an object of class `istream` (input stream). `operator>>`(the extraction operator) is one of it's member function.

Check `std::istream::operator>>`, similar for `cout` (`ostream`)

```
istream& operator>> (bool& val);
istream& operator>> (short& val);
istream& operator>> (unsigned short& val);
istream& operator>> (int& val);
istream& operator>> (unsigned int& val);
istream& operator>> (long& val);
istream& operator>> (unsigned long& val);
istream& operator>> (long long& val);
istream& operator>> (unsigned long long& val);
istream& operator>> (float& val);
istream& operator>> (double& val);
istream& operator>> (long double& val);
istream& operator>> (void*& val);
```

Many type of parameter it takes -> it knows how to convert the characters into values of certain type

Return value also a reference of `istream` -> it can be cascaded like `cin >> foo >> bar >> baz;`

###Other functions

```
istream& getline (istream& is, string& str);
std::ios::operator bool // member of istream -> if(cin)
istream& get (char& c); // member of istream
```

Buffered

`cout` and `cin` streams are buffered.

(4) (5%) When using a buffered output stream, list the events that make the buffer written in the output stream.

File Stream

```
ifstream iFile; // inherit from istream
ofstream oFile; // inherit from ostream
iFile.open("myText.txt");
iFile.close(); // important, remember to close

iFile >> bar; //explained by inheritance
if(!iFile) ...
while(getline(iFile, line)) // simple way

oFile << bar; //explained by inheritance
```

String Stream

```
istringstream iStream; // inherit from istream
iStream.str(line); // assigned a string it will read from, often used with
getline
iStream >> foo >> bar; //explained by inheritance

ostringstream oStream; // inherit from ostream
oStream << foo << " " << bar; //explained by inheritance
result = oStream.str(); //string str() const;
```

Lecture 11: Testing

Difference between testing and debugging

- Debugging: fix a problem
- Testing: discover a problem

Five Steps in testing:

1. Understand the specification
2. Identify the required behaviors
3. Write specific tests
4. Know the answers in advance
5. Include stress tests

General steps for Test Driven Development

1. Think about task specification carefully
2. Identify behaviors
3. Write one test case for each behavior
4. Combine test cases into a unit test
5. Implement function to pass the unit test
6. Repeat above for all tasks in the project

A simple example

Step 1: Specification

```
Write a function to calculate factorial of non-negative integer,
return -1 if the input is negative
```

Step 2: Behaviors

```
Normal: return 120 for input = 5
Boundary: return 1 for input = 0
Nonsense: return -1 for input = -5
```

Step 3: Test Cases

```
void testNormal() {
    assert(fact(5) == 120);
}

void testBoundary() {
    assert(fact(0) == 1);
}

void testNonsense() {
    assert(fact(-5) == -1);
}
```

Step 4: Unit Test

```
void unitTest() {
    testNormal();
    testBoundary();
    testNonsense();
}
```

Step 5: Implement

```
int fact(int n) {
    if (n < 0) return -1;
    int ret = 1;
    for (int i = 2; i <= n; ++i)
        ret *= i;
    return ret;
}
```

Lecture 12: Exception

Exception propagation: where to find the handler

1. the remaining part of the function, if not found, proceed to step 2
2. caller of the function, if not found, proceed to step 3
3. caller of the caller...

```
void foo() {
    throw 1;
}
void bar() {
    foo();
}
int main() {
    try {
        bar();
    }
    catch (...) {
        //default constructor
        std::cout << "Caught!";
        return 0;
    }
    return 0;
}
```

Examples

1. the **first** catch block with the **same** type as the thrown exception object will handle the exception
2. `catch(...) {}` is the default handler that matches any exception type

```
int foo(int i) {
    try {
        if (foo) throw 2.0;
        if (bar) throw 1;
        if (baz) throw 'a';
    }
    catch (double v) {
        // catch exceptions thrown by if (foo)
    }
    catch (int a) {}
    catch (char c) {}
    catch (...) {
        //default handler
    }
    ...// Do something next
}
```

3. Use EFFECTS clause to tell the caller that a function can throw an exception:

```
int factorial (int n) {
    // EFFECTS: return n! if n >=0, throws int n if n < 0
}
```

Lecture 13: Abstract Data Type

Why need ADT?

Abstraction hides detail and makes users' life simpler

Procedural abstraction: Function
A handy black box, processes whatever you give it

Data abstraction: Abstract Data Type (ADT)
A handy object, stores information, easy to use

Key Components

1. Specification
2. Member Type
3. Data
4. Methods

IntSet Example

```
const int MAXELTS = 100;
class IntSet {
   public:
    IntSet();
    void insert(int v);
    void remove(int v);
    bool query(int v) const;
    int size() const;

   private:
    int elts[MAXELTS];
    int numElts;
    int indexOf(int v) const;
};
```

Specification

Comments to explain things

- **OVERVIEW**
  Short description of class
- **RME**
  What the function does
- **REP INVARIANT**
  What the variable stands for

Member Type

Different view from user and developer (outside and inside)

- **Public**
  What users can see and use, no need to care about actual implementation
  “Insert 5 to the set! Easy!”
- **Private**
  What developers have to think of to meet users’ requirement
  “The user wants to insert 5, I have to first check if 5 already exists! Ahh, and also the total number may change!”

Data

Variables to store information

- **Representation**
  “The user want a set, so it can be an array, or maybe a linked list.”
- **Representation Invariant**
  What the variable stands for and can be taken as granted immediately before and after an operation
  “numElts always equals to the number of elements in the set, so size() can use this and doesn’t need to count every time!”

Method

Functions to get things done

- **Maintenance**
  Constructor for initialization
- **Operation**
  For users to use
- **Helper**
  Useful because other methods may want to use the same functionality
- **Const Qualifier**
  const object, const function, const this
- **Implementation**
  Detect error, get things done,
  repair invariant