CS246 Midterm Review Notes

Fall 2019

Oct 17 2019

Contents

1 Linux Shell

Shell: interface to the operating system.

- Graphical shell: easy to learn but inflexible
- Command-line shell: flexible but steep learning curve

1.1 Linux File System

- Files are organized as a tree structure.
- The root directory is the root (/) of the file system.
- **Directory:** a file that contains files
- **Path:** Used to specify the location of a file
  - E.g. **/usr/share/dict/words** (first / specifies root directory, other /s are directory separators)
  - Absolute path: starts at the root file
  - Relative path: starts relative to the "current directory"
- Special directories:
  - . - current directory
  - .. - parent directory
  - - home directory
  - ˜<userid> - userid's home directory

1.2 Linux Commands

Structure of Linux commands: <command> <arguments>

- echo $0 - tells you which shell you are running
- bash - switch shell to bash

1.3 Output Redirection

- Output redirection will redirection the output of a program to a file using >
- *ctrl* + *d* tells the cat command that there is no more input, sends an EOF signal, program gets to "react" unlike *ctrl* + *c*

1.4 Input Redirection

Input redirection sends input to a program from a file

- *cat* $<$ *file* - feature of the shell (shell opens the file and makes content available to the program)
- *cat file* - given a file name, cat finds and opens a file
- *cat* $-n$ *argument* $<$ *in.txt* $>$ *out.txt*

1.5 IO Streams

	Standard Input (stdin)	Standard Output (stdout)	Standard Error (stderr)
Default	keyboard	screen	screen
Redirection	Use < to redirect	Use > to redirect	Use 2> to redirect

1.6 Linux Pipes

A Linux pipe connects stdout of a program to the stdin of another program
Note: Using the output of a program as an argument is called **Embedded Command**

- e.g. echo "Today is $(date) and I am $(whoami)"

1.7 Wildcard Matching (Globbing Patterns)

Quotes: Allows embedded commands and suppresses globbing
Single Quotes: Does not allow embedded commands and suppresses globbing patterns

1.8 Searching within text files

egrep <pattern> <file> : extended global regular expression print

- looks at one line at a time and outputs lines that match the **patterns**
- Patterns are regular expressions that specify regular language
- egrep is case sensitive

1.9 Regular Expressions

Regular expressions are only used with egrep and are different from globbing patterns

Symbol	Meaning
a|b|c|d	equivalent to [abcd], ”choose one character from the set”
[!abcd]	any 1 character not in the set
*	0 or more of the preceding
+	1 or more of the preceding
?	0 or 1 occurrences of the preceding
.	match any one character
.*	match any number of any character
^*pattern*	force a match to begin at the start of the line
pattern$	force a match to be the last pattern in the line

Anything written in + can be written with *

1.10 File Permissions

Use ls l for a long listing that shows file permissions.
Example of file's permissions: $rw\_|r\_x|r\_\_$ (user/owner bits | group bits | other bits)

- x for execute. For ordinary files, this means the file can run. For directories, it means you can enter the directory
- **Command to change permissions:** chmod <mode> <file>. How to write the mode:
  1. ownership class: **u**ser, **g**roup, **o**ther, **a**ll
  2. + (add), **-** (remove) or = (switch)
  3. **r**, **w**, or **x**
- Shortform e.g: `chmod 744 file`

2 Shell Scripts

A shell script is a text file containing a sequence of commands that is executed as a program.

- #!*/bin/bash* (shebang line) should appear in every script you write
- tells the shell that this is a sequence of bash lines and to ”switch to bash” and start executing

2.1 Shell Variables

- Variables contain strings e.g. $x = 1$ (no spaces)
- Use $\$x$ to access value

- $\$PATH$ gives list of paths
- Scripts can be given arguments which is available as variables \$1, \$2, etc. as the first, second, args respectively
- **Arithmetic operations syntax**: e.g. $((x+1))
  - Bad idea: \$x + 1 because it will give "1 + 1"

2.2 Test Program

Every process sets a global status code (\$?) which will either by 0 for success or a non-zero for a failure (not just 1 because there are many ways something can fail).
A test program evaluates a condition and sets the status code.
e.g. `[ 1 -eq 2 ]` outputs 1. **Need spaces between because they are arguments**

2.3 If Statements

```
if [ cond ]; then
    # code
elif [ cond ]; then
    # code
else
    # code
fi
```

2.4 While Loop

```
while [ cond ]; do
    #code
done
```

2.5 For Loop

```
for x in 1 2 3 ... N
do
    #code
done
```

3 Basic C++

Compiling from the command line: `g++ -std=c+14 hello.cc -o myprog`
or `g++14 hello.cc`

3.1 C++ I/O Streams

When you `#include <iostream>`, you gain access to 3 global variables.

stream	variable	type	code
stdin	std::cin	istream	cin >> var;
stdout	std::cout	ostream	cout << var;
stderr	std::cerr	ostream	cerr << var;

- If a read fails, a default value is used and the program continues (as of C++11).
- It's okay to not initialize a variable as long as you write to it before you read from it
- If a read fails, the expression `cin.fail()` is tru
- If a read fails due to EOF, the `cin.fail()` and `cin.eof()` are true
- An istream variable (cin) can be automatically be converted to a bool.
  - `cin` is considered true if `!cin.fil()`
  - `cin` is false if `cin.fail()`
- **Formatting I/O** `<iomanip>`
  - We use I/O manipulators to change the mode/format of the I/O
  - e.g `ccout << hex << x` : prints in hexadeciamal
  - `ccout << dec;` : back to decimal
  - Other manipulators include `showpoint, setprecision, boolalpha`

3.2 C++ Strings

`#include <string>` Strings in C++ are arrays of characters in memory that is null-terminated

comparisons	==, !=, <, >
concatenations	s1 + s2
length	.length()
access chars	s[i]

Reading strings:

- `cin` into a string: read from first non-whitespace char until the first whitespace char
- `getline(cin, s);`: read until newline

3.3 File I/O Streams

```
#include <fstream>
```

- `ifstream` : read from file
- `ofstream` : write to file
- e.g. `ifstream file suite.txt`
- Anything we can do with an istream variable like cin, we can do with an ifstream variable

String Stream `<sstream>` We can use a C++ string as a source/destination for a file using `istringstream` and `ostringstream`

3.4 Constants

Constants are variables whose value cannot be changed.

```
const int n = 5;
const Node n = {5, nullptr};


int const *p;
p=&n; p=&m; // valid
*p = 5; // not valid


// where the const is changes what can be modified
int * const p;
*p = 5; // valid
p = &n; // no longer valid
```

3.5 References

C and C++ pass by value.

```
void inc(int n) {
   n += 1;
}
int k = 5;
inc(k);
cout << k; // prints 5
```

In C, pointers were used to change values, and addresses were passed. In C++, references are used. References are constant pointers with automatic dereferencing. For example,

```
int y = 5;
int &z = y;
```

Whenever `z` is written, it means the value pointed at by `z`.

- References are also referred to as *alias*.
- We cannot have pointers to references
  - `int &*x = &n;` is wrong
  - `int *&x = something;` is fine
- References of an object must be initialized (e.g. `int &x;` is wrong)
- You cannot have a reference of an array (e.g. `int &x=(y,y,y);` is wrong)
- `int &&x = n;` is wrong since `&&` has a different meaning in C++

3.5.1 Pass by Reference

Although references seems useless since we would usually just use the original, it's good for pass by reference.

```
void inc(int &n) {
    n += 1;
}

int k = 5;
inc(k);
cout << k; // prints 6
```

Pass by reference is useful especially when we're passing in very large structs, since we will not need to copy every thing in the struct.

3.5.2 l-values

An l-value is:

- something that has a permanent place in memory
- something that appears on the left hand sign of the assignment operator
- something that has a name
- E.g. `int k = 5;`, `k` is an l-value
- E.g. `2+2` is not an l-value (we can't do `int &x = 2 + 2;`)

With the previous `inc()` example, we cannot call `inc(2)` as it's written. However, if we pass by const ref, e.g. `int inc(const int &n);`, then we can pass in non l-values as well.

It is strongly recommended that if a parameter does not need to be modified, and is bigger than an `int`, pass by const ref. Some trivia, cin passes by const ref.

3.6 Dynamic Allocation

```
// C style
int *k = malloc(sizeof(int));
free(k)
```

```
// C++ style
int *k = new int;
delete k;
```

Pointes can be set to nullptr, but refs cannot (called nullable)

3.7 Operator Overloading

We can write operator functions, which is useful for custom objects

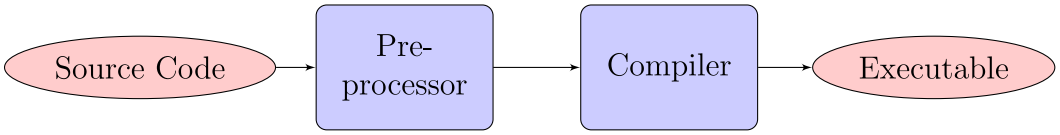
```
struct Vec {int x, y};
// the motivation is that the + operator should work for Vecs
Vec operator+(const Vec &v1, const Vec &v2) {
    return Vec{v1.x + v2.x, v1.y + v2.y};
}
Vec v{1, 2}, w{3, 4};
v + w; // returns a Vec with 4 and 6

//another example
Vec operator*(int k, const Vec &v1) {
    return Vec{k * v1.x, k * v1.y};
}

Vec operator*(const Vec &v1, int k) {
    return k*v1;
}
```

4 Preprocessors and Compilation

4.1 Preprocessors



`#include` is a preprocessor directive

- Copy and paste the header file
- `#include <library>` : include from library file
- `#include "file"` : include from current directory

Other preprocessor directives:

- `g++ 14 -E file.cc` : shows the outputs of the preprocessor before it goes through the compiler
- `#define var value` : tells the preprocessor to define var with value using search and replace
  - e.g. `#define MAX 0` : We can use MAX as a constant

4.2 Conditional Compilation

```
#define SECURITY 1 // or 2
#if SECURITY == 1
    // short int
#elif SECURITY == 2
    // long long int
#endif
    // public key
```

```
#define FLAG
#ifDef VAR // if VAR is defined
    // code
#endif
#ifndef VAR // if VAR is NOT defined
    // code
#endif
```

- **Preprocessor variable from the command line**: `g++14 -D SECURITY=2 file.cc`

- **Debug variable**: `g++ -DEBUG debug.cc`
  - compile with the debug variables
  - recompile with fixes

4.3 Separate Compilation

Separate compilation breaks programs into :

1. Interface files / header files (.h) which includes type definitions and function declarations / prototypes
2. Implementation files (.cc) which includes function definitions

Separate compilation is for when there are many .cc files. How to compile: `g++14 main.cc vec.cc`
Never compile header files and never include .cc files
How to compile main.cc & vec.cc separately:

- `g++ main.cc` : by default, compiler compiles & produces executable
- `g++14 -c main.cc` and `g++14 -c vec.cc` : gives one compilation
  - Produces output file (.o) `g++ main.o vec.o`
  - When we change one .cc file, we just make one single new object file and them all of them after

4.4 Makefiles

Build tools keep track of updated files so that we only recompile what is necessary. With Linux, we use **make**, and we use it by giving a **Makefile** A Makefile provides all the dependencies in the program, and is useful for compiling two or more files separately and merging their executables.

Once the .o files are created, the `g++14 main.o otherfile.o ...` will merge the executables. The Makefile is usefule for compiling **only what is necessary**. Expect to know how to read a Makefile, but knowing how to write one is not necessary for examinations.

```
myprogram: main.o vec.o
   g++ main.o vec.o -o myprogram

vec.o: vec.cc vec.h
   g++ -std=c++14 -c vec.cc

main.o: main.cc vec.h
```

```
   g++ -std=c++14 -c main.cc

filename: dependency dependency
    recipe to make filename

.PHONY: clean

clean:
   rm *.o myprogram

//////////////////////////////////////

CXX = g++
CXXFLAGS = -std=c++14 -Wall
OBJECTS = main.o vec.o
EXEC = myprogram

${EXEC}: main.o vec.o
   ${CXX} ${CXXFLAGS} ${OBJECTS} -o ${EXEC}

vec.o: vec.cc vec.h

main.o: main.cc vec.h

.PHONY: clean

clean:
   rm ${OBJECTS} ${EXEC}
```

4.5 Include Guards

Oftentimes, multiple files including each other will repeat an include. For example, if file1 includes file2, and both file1 and file2 include file3, nothing would compile. We use include guards to avoid this.

```
#ifndef _VECTOR_H_
#define _VECTOR_H_

struct Vec {
    int x;
    int y;
}

Vec operator+(const Vec &v1, const Vec &v2);
```

```
#endif
```

Random Note: Never put "using namespace std" in header files.

5 C++ Classes

The big idea in Object Oriented Programming (OOP) is that we can put functions in structs.

```
// student.h
# ifndef STUDENT_H
# define STUDENT_H
struct Student {
    int assns, mt, final;
    float grade();
};
#endif

// student.cc
#incldue "student.h"
float Student::grade() {
    return assns * 0.4 + mt * 0.2 + final * 0.4;
}

// main.cc
Student billy{70, 50, 75};
billy.grade()
```

- In OOP, a class is a struct that contains functions
- An instance/value of a class is called an object
- A function defined inside a class is called a "member function"/"method"
  - Methods can only be called using objects
  - Methods have access to the fields of the object on which the method was called
  - Every method has a hidden/automatic parameter. This parameter is called `this`
  - `this` is a pointer to the object used to call the method, so `this == &billy` or `*this == billy`
  - When accessing fields of the object, we do `this->field`

5.1 Object Initialization

The following are both C-style initialization syntax, which are compile time constant (TODO find out what that means).

```
Student billy{70, 50, 75};
Student billy = {70, 50, 75};
```

5.2 Constructors

C++ allows writing special methods to construct objects, called constructors. Constructors have no return type, the constructor's name is the class name, and they do not return objects (they construct them)

```
// student.h
struct Student {
    // code
    Student(int assns, int mt, int final);
};
// student.cc
#include "student.h"
Student::Student(int assn, int mt, int final) {
    this->assns = assns;
    this->mt = mt;
    this->final = final;
}
```

5.2.1 Advantages of Constructors

- Constructors are methods, which means they are functions, so constructors get all the nice properties of functions
- Arguments need not be compile time constant
- Within the constructor body, we can write whatever code we want
  - Sanity checks
- We can overload constructors
- We can put in default arguments

```
// student.h
struct Student {
    // code
    Student(int assns = 0, int mt = 0, int final = 0);
    // note that the default values go in the declaration
};

// student.cc
Student::Student(int assns, int mt, int final) {
    this->assns = assns < 0 ? 0 : assns;
    this->mt = mt < 0 ? 0 : mt;
    this->final = final < 0 ? 0 : final;
}

// main.cc
Student s1{70, 60}; // s1.final == 0
Student s2{}; // all fields 0
```

5.3 Default Constructor

Every class comes with a free default constructor, which is a constructor that takes 0 parameters.

- If you write *any* constructor yourself, you lose the default constructor as well as C-style initialization.
- The default constructor will call constructors on fields that are objects

A small example:

```
// MyStruct.h
struct MyStruct {
    int x;
    Student y;
    Vec *z;
};

// main.cc

// this calls the default constructor
MyStruct s;
```

The above will compile. `x` and `z` will not be initialized. `y` will be "default" constructed.

Another example:

```
// Vec.h
```

```
struct Vec{
    int x, y;
    Vec(int, int);
};

// Vec.cc
Vec::Vec(int x, int y) {
    this->x = x;
    this->y = y;
}

// main.cc
Vec v{1, 2}; // is not C-style initialization (but it is not)
Vec v1 = {1, 2};
Vec v2; // will not compile since we lost the default constructor
Vec v3 = {1}; // will not work as well
```

5.4 Initializing Fields that are const/lvalue references

```
int m;
struct MyStruct {
    const int x = m; // in-class initialization (new as of C++11)
    int &y = m;
};

struct Student {
    const int id = 21234567;
    int assns;
    int mt;
    int final;
};
```

In this Student example, we would like to initialize the id field and then keep it constant afterwards. To understand how to do this, we must look at how objects are created.

5.5 Steps of Object Creation

1. Allocate space for the object
2. Field initialization: default construct fields that are objects
3. Constructor body runs

The solution is to "hijack" step 2 using member initialization list (MIL)

5.6 Member Initialization List (MIL)

```
// student.cc
Student::Student(int id, int assns, int mt, int final) : id{id}, assns{assns},
    mt{mt}, final{final} {}
```

- MIL is only available for constructors
- MIL can be used to initialize all fields
- `this` is not required to disambiguate between fields and parameters.
- MIL initializes fields in declaration order
- MIL can be more efficient than using the constructor body for initializing fields that are objects
- MIL takes precedence over in-class initialization (see example below)

```
// vec.h
struct Vec {
    int x = 0;
    int y = 0;
    Vec(int, int);
};

// Vec.cc
Vec::Vec(int x, int y) : x{x}, y{y} {}

// main.cc
Vec v{1, 2}; // the values of x and y are 1 and 2
```

6 The Big 5

6.1 Copy Constructors

Constructing an object as a copy of another object uses the **copy constructor**

```
Student billy{70, 50, 75};
Student bobby{billy};
```

The copy constructor that comes for free does a field-for-field copy:

```
Student::Student(const Student &other): assns{other.assns}, mt{other.mt},
    final{other.final} {}
```

Sometimes this copy constructor is not "correct". For example, with a linked list:

```
// Node.h
struct Node {
    int data;
    Node *next;
    Node(int, Node *);
    Node(const Node &);
};

// Node.cc
Node::Node(int data, Node *next) : data{data}, next{next} {}
Node::Node(const Node &other) : data{other.data}, node{other.next} {} // this is
    what the free copy constructor would look like

// main.cc
Node *p = new Node{1, new Node {2, new Node {3, nullptr}}};
Node m{*p}; // calling the copy constructor, which is now stack allocated
Node *q = new Node{*p} // same as m, but heap allocated
```

The free copy constructor will make a "shallow copy", and all of the lists will share the same nodes.

- What happened was that `m` created the value 1 correctly, and then copied the *pointer* to the next node, not the next node

We must implement our own constructor to do a "deep copy".

```
Node::Node(const Node &other) : data{other.data}, next{new Node{*other.next}} {}
```

The above example creates a new node correctly by recursively calling the copy constructor. However, it is slightly wrong since there is no base case, and we need to handle when other.next is nullptr.

```
Node::Node(const Node &other) : data{other.data}, next{other.next ? new
    Node{*other.next} : nullptr} {}
```

A copy constructor is called whenever

- an object is created as a copy of another
- there is pass by value - this is the reason why copy constructors are done pass by reference
- there is return by value

6.1.1 Constructors with 1 value

Let's create a function with one parameter.

```
Node::Node(int data) : data{data}, next{nullptr} {}
```

One parameter constructors create automatic/implicit conversions. For instance, `Node m = 4;` is a valid line of code. If we have a function `void foo(Node);`, the line of code `foo(4);` is also valid. Another example is `string s= "Hello";`. The string constructor has one parameter and implicitly converts the C-style array of characters that is null-terminated into a C++ string. We can disable this automatic conversion by declaring the constructor `explicit`.

```
struct Node {
    // code
    explicit Node(int);
};
```

6.2 Destructors

- Stack allocated variables: destroyed when the identifier goes out of the scope.
- Heap allocated variables: `delete` the pointer to the object
  - A method that runs when an object is destroyed (either stack or heap)
- The reverse of the 3 steps of object creation:
  1. Destructor body runs
  2. Fields are destroyed: destructors are called for fields that objects in reverse declaration order
  3. Memory is deallocated
- The "free" destructor has an empty destructor body. Sometimes this the wrong thing to do. For example, for a Node struct:

  ```
      Node *p = new Node{1, Node{2, new Node{3, nullptr}}};
      delete p;
  ```

  node 2 and node 3 are not deleted, and thus, we leak memory. We must make our own destructor:

  ```
          // Node.h
          struct Node {
              // code
  ```

```
    ~Node();
};

// Node.cc
Node::~Node() {
    delete next;
}
```

6.3 Copy Assignment Operator

```
Student billy{10, 10, 10};
Student jane = billy;
// this line of code is actually jane.operator=(billy);
```

In contrast with the copy constructor, this operator is called when an object is declared, and then assigned the value of another object.

Example with linked list and nodes:

```
Node &Node::operator=(const Node &other) {
    if (this == &other) return *this // self assignment check

    data = others.data

    delete next; // we must have this line, since there is an existing object and
        we have to avoid a memory link

    // similar to the copy constructor now
    next = other.next ? new Node {*other.next} : nullptr;
    return *this;
}
```

Whenever possible, pass a constant reference. This method should return the updated node and update the node itself. We do this for the same reason we can do this with integers: `a = b = c = 0;`. The return type is a reference since it is faster than returning just the object.

Here is a better implementation:

```
Node &node::operator=(const Node &other) {
    if (this == &other) return *this;
    Node *temp = next; // in case new throws an exception
    next = other.next ? new Node{*other.next} : nullptr;
    // if new throws an exception then all the code below will not run
    data = other.data;
    delete temp;
    return *this;
}
```

6.3.1 Copy and Swap Idiom

```
#include <utility>

struct Node {
    // code
    void swap(Node &);
    Node &operator = (const Node &);
};

void Node::swap(Node &other) {
    using std::swap;

    swap(data, others.data);
    swap(next, other.next);
}

Node &Node::operator=(const Node &other) {
    Node tmp{other}; // call the copy constructor
    swap(tmp);
    return *this;
    // we do not have to delete tmp since tmp is stack allocated
}
```

6.4 Move Constructor

6.4.1 Rvalues and Rvalue references

```
Node plusOne(Node n) {
    // code
    return n;
}

Node n{1, new Node{2, nullptr}};
Node n1{plusOne(n)}
```

In the above code, `n` is passed by value. This calls the copy constructor once initially, and one more time recursively. At the end of `plusOne`, it calls the copy constructor (two times again) when returning the value. This returned value "sits" on the right hand side of the assignment temporarily, and then we copy that value (twice again) into `n1`. This makes 6 calls in total to the copy constructor.

An rvalue is:

- a temporary
- anything not an lvalue
- Anything that cannot be referenced, that doesn't have a name, etc.
- something that is going to/about to be destroyed

The move constructor:

- has one parameter, which is an rvalue reference - a reference to an object that is about to be destroyed
- Instead of copying from that object, we "steal" their fields since they are going to die anyway
- In general, used to construct an object from an rvalue

```
Node::Node(Node &&other) : data{other.data}, next{other.next} {
    other.next = nullptr;
}
// we copy with MIL, and the function body deletes the old content
```

Recall that an rvalue is a value that has no name, exists momentarily, and is destroyed once it serves its purpose. Note that in the signature of the move constructor, we use the *reference* of an rvalue.

Let `other` be a reference to an rvalue (say a linked list with 3 nodes). If we ran the copy constructor, it would be less efficient to copy every node of the list. The move constructor instead just copies the first node, and steals the rest, which is more efficient than copying everything.

6.5 Move Assignment Operator

Similar to the copy assignment operator, we are assigning to a pre-existing value. Say we have a linked list called `l` with nodes 4, 5, 6, and 7. We can swap the values we don't want anymore to the rvalue that is going to be destroyed.

```
Node &Node::operator=(Node &&other) {
    swap(other); // see copy and swap idiom from oct 10
    return *this;
}
```

6.6 Rule of 5

If you need a custom version of any one of copy constructor, copy assignment operator, desctructor, move constructor, or the move assignment operator, then usually you need all five.

7 Short Topics

7.1 Copy/move Elijsion

Turn off -fno-elide-constructors, which would normally skip copying into a temporary value.

```
Vec makeVec() {
    return {1, 2};
}
Vec v1 = makeVec();
```

In the code above, the compiler knows where the final resting place of `1, 2` and will directly place the values in `v1.x` and `v1.y`, without calling the copy constructor. C++ allows compilers to elide/avoid calls to copy/move constructors if there is a more efficient way.

7.2 Overloading Operators

Should we be implementing operators as methods or stand alone functions?
Rules:

1. `operator=` is implemented as a method. This is because the left hand side will always be referred to as `this`
2. `operator[]` must also be implemented as a method
3. `operator()`, `operator->`, `operatorT()` should be written as methods as well.

7.3 Arrays of Objects

How can we write arrays of objects? The below code will not compile.

```cpp
struct Vec {
    int x;
    int y;
    Vec(int, int);
};
Vec arr[10]; // stack array won't compile, no default constructor
Vec *p = new Vec[10]; // heap array won't compile, no default constructor
```

Option 1: Write a default constructor
Option 2: Stack arrays, explicitly initialize each element

```cpp
Vec arr[3] = {Vec{1, 2}, Vec{3, 4}, Vec{5, 6}}
```

Option 3: Use an array of pointers to objects

```cpp
Vec *arr[10];
arr[0] = new Vec{1, 2};

Vec **p = new Vec*[10];
p[i] = new Vec{1, 2};

for (int i = 0; i < 10; ++i) {
    delete p[i];
}
delete [] p;
```

7.4 Const Methods

```
struct Student {
    int assns, mt, final;
    float grade() const {
        return assn * 0.4 + mt * 0.2 + final * 0.4;
    }
};

const Student billy{80, 50, 80};
cout << billy.grade(); // will not work without const on the method name
```

The compiler will not allow calling methods on `const` objects, even if the method wouldn't change the object's fields anyway. We must append `const` to the end of the function declaration, which promises to not change fields of `*this`. Note: `const` objects can only call `const` methods.

8 Invariants

```
struct Node {
    int data;
    Node *next;
    ~Node() {delete next;}
};

Node n1{1, new Node {2, nullptr}};
Node n2{3, nullptr};
Node n3{4, &n2};
```

The above three nodes will be destructed when they're gone out of scope (they're stack allocated). They will be destroyed with the destructor. Thus, we are attemping to deallocate a stack address, which will crash.

Node assumed that `next` is either `nullptr` or points to the heap. When that isn't true, we get undefined behaviour (probably a crash).

An invariant is a statement/assumption that must be true for code to function correctly. We must prevent the line containing `n3` to not compile. In general, we must prevent users from breaking the invariants of our program.

9 Encapsulation

- Treating objects as capsules/black box.
- Implementation is hidden and a few selected methods are exposed.

```
struct Vec {
    Vec(int, int); // default visibility is "public"
    private:
    int x, y;
    public:
    Vec operator+(const Vec &rhs) {
        return {x+rhs.x, y+rhs.y};
    }
};
```

Note that in the example, we can do `rhs.x, rhs.y`. As long as we're within the `Vec` code, we can use x and y.

```
int main() {
    Vec v{1,2};
    Vec v1= v + v;
    cout << v.x << v.y; // will not compile, x and y are private
}
```

Advice: keep fields private, make select methods public.

9.1 "Class" keyword

- Default visibility is private
- Very similar to struct

```
class Vec {
    int x, y;
    public:
    Vec(int,int);
    Vec operator+(const Vec&);
};
```

9.2 Node Invariant

Prevent direct access to Nodes by creating a wrapper List class

```
// list.h
class List {
    struct Node; // declaration of private, nested class
    Node *theList = nullptr;
    public:
    void addToFront(int i);
    int ith(int i);
    ~List();
};

// list.cc
struct List::Node {
    int data;
    Node *next;
    Node(int, Node*);
    ~Node() { delete next; }
};

void List::addToFront(int i) {
    theList = new Node{i, theList};
}

int List::ith(int i) {
    Node *curr = theList;
    for(int j = 0; j < i && curr; ++j,curr=curr->next);
    return curr->data;
}

List::~List() { delete theList; }

///////
List l;
// fill the list with stuff
for (int i = 0; i < n; ++i) {
    cout << l.ith(i);
} // O(n^2) time

Node *curr = theList;
while(curr) {
    // code
    curr = curr->next;
} // O(n) time
```

Idea: create an abstraction of a pointer into the list

```
// arr an array
for (int *p = arr; p != arr+size; ++p) {
    ////*p//////
} //TODO: !=, ++, *, begin, end
```

10 Iterator Design Pattern

Helps us support linear time traversal while still having encapsulation.

```
class List {
    struct Node;
    Node *theList = nullptr;
    public:
    //////
    ////// same methods as before

    class Iterator {
        Node *curr
        public:
        Iterator(Node *p) : curr{p} {}
        int &operator*() {
            return curr->data;
        }
        Iterator &operator++() {
            curr = curr->next;
            return *this;
        }
        bool operator!=(const Interator &other) {
            return curr != other.curr;
        }
    };
    Iterator begin() { return Iterator{theList};}
    Iterator end() {return Iterator{nullptr; }}

};
```

What was the point of this? Now, if we want to traverse the list, this is what we can do:

```
for (List::Iterator it = list.begin(); it != list.end(); ++it) {
    cout << *it << endl; //print 1 2 3 in linear time, and without exposing nodes
    *it += 1; // updates to 2,3,4
}
```

We can also use the keyword `auto`.

```
for (auto it = list.begin(); it != list.end(); ++it) {
    cout << *it << endl; //print 1 2 3 in linear time, and without exposing nodes
    *it += 1; // updates to 2,3,4
}
```

C++ has built in support for the iterator design pattern in the form of what is called Range-based for loops.

```
for (auto &n: list) {
    cout << n; // n is the result of dereferencing
    n += 1;
}
```

Note: omtting the `&` will create a by value declaration, and will not update curr.