Linux shell

Shell

- interface to the operating system
- Two type of shells:
    1. **Graphical shell** - Click to select, drag and drop. Easy to learn but restrictive.
    2. **Command-line shell** - Type commands on a prompt. Powerful but steep learning curve.

Unix

- built in 70s
- Came with command-line interface (shell)
- Stephen Bourne
- Other popular shells
    - C Shell (csh) -> Turbo C Shell (tcsh)
    - Korn Shell (ksh)
    - Original *Shell* became Bourne Again Shell (bash)
- Run `echo $0` to see which shell you are using. You should be using `bash` (or `zsh` which is mostly bash compatible)

Linux file system

- Directories - just files that can contain other files
- Non-directory files - *ordinary files*
- File system is a tree structure
    - The root of the tree is the *root directory* (`/`)
- **Path**: Way to specify the location of any file in the file system
    - Eg. /usr/share/dict/words

Commands

- `ls` - listings
    - lists all non-hidden files in the directory
    - **hidden files**: name starts with a dot
        * `ls -a` to view hidden files too (all files)
    - **Special Files**:
        * `.` is the current directory
        * `..` is the parent directory
- `pwd` - present working directory

- `cd` - change directory
    - `cd {path}` where {path} is a path to a directory
        * Allows absolute paths or relative paths

Sept 13 - CS246

Using output of program as arg to another program

- Eg. `echo "Today is $(date) and I am $(whoami)`
    - `date` and `whoami` are both commands
- Using double quotes for arguments
    1. echo Today is $(date) and I am $(whoami)
    - This is multiple arguments but they still all get printed
    2. echo "Today is $(date) and I am $(whoami)"
    - Single arg
    3. echo 'Today is $(date)'
    - Output: Today is $(date)
    - Single quotes will suppress embedded commands
    4. echo `date`
    - It will print the date. `date` is treated as a command and is evaluated
    - Older syntax, won't be used much

Searching within text files

`man grep`

- tool: egrep (grep -E)
- print every line in files provided as args that contain the regex pattern

Regex additional examples

- Match zero or one of the precending identifier
    1. To match `cs246` and `cs 246`
    - `cs ?246`
    2. To match `cs246` and `246`
    - `(cs)?246`

3. Match zero or more of the preceding identifier

- `*`
- Not the same as globbing wildcard
- `(cs)*246`
    - Match `cs246 cscs246 246`

4. Match any character (only a single char)

- `.`
- To match any number of anything
  - `.*`

5. To search for special identifiers you can escape with `\`

- `\(\)` will match `()`

6. Match one or more occurences

- `+`
- `sc+` will match `sc` `sccc` `scccccc` but not `s`

7. Match the start of the line

- `^`
- `^tobe` will match `tobesssss` but not `sssstobe`

8. Match end of the line

- `$`
- `tobe$` will match `dddddtobe$` but not `dddtobeddd`
- `^tobe$` will match `tobe` but not `dddtobeddd`

Examples

1. `grep searchtext myfile.txt`

- Search for `searchtext` in myfile.txt

2. `grep "searchtext|Searchtext" myfile.txt`

- Or `grep "(s|S)earchtext" myfile.txt`
- Search for `searchtext` or `Searchtext` in myfile.txt

3. `grep "[sS]earchtext" mufile.txt`

- Same as above, `[]` mean to match any char between them
- `[^sS]` means to match not `s` and not `S`

3. `grep -i "searchtext" myfile.txt`

- Same as above, but `-i` forces case-insensitivity

4. Print all words from /usr/share/dict/words that start with an e and consist of 5 characters

- `egrep "^e.{4}$" /usr/share/dict/words`
- `egrep "^e....$" /usr/share/dict/words`

5. Lines of even length

- `egrep "^(..)*$" /usr/share/dict/words`

6. Filenames from cur directory whose name contains exactly one `a`

- `ls | egrep "^[^a]*a[^a]*$"`

File Permissions

- `ls -l` - long listing
  - Eg.

```
-rw-r--r--  1 rethy  staff  2433 Sep 13 15:29 sept_13.md
-rw-r--r--  1 rethy  staff  1351 Sep  9 15:37 sept_6.md
```

  - see below - shortcut - owner - group - size - date last modified - filename
  - `-rw-r--r--`
    - first bit - type of file - ordinary or directory (`-` for ordinary file, `d` for directory)
    - Nine following bits of three bits each (owner bits - single owner, group bits - all users that belong to the same group to which the file belongs, other bits - all other users not in group)
    - r (read) w (write) x (execute)
  - Ordinary files
    - `r` read the file, cat the file
    - `w` modify the contents
    - `x` can execute the program
  - Directories
    - `r` ls, tab completion
    - `w` add/remove files
    - `x` can search the directory, enter it, navigate into it

Changing permissions

- `chmod {mode} {file}+`
  - Add or remove mode to file
  - `u` (owner) `g` (group) `o` (other) `a` (all three)
  - `chmod a+x {file}+` - add executable permission for everyone
  - `chmod u-rw {file}+` - remove read/write permission from owner
  - `chmod u= {file}+` - remove all permissions from owner (still has permission to re-add these permissions)

Shell Scripts

- **Script** - a test file containing a sequence of commands executed as a program

Assume we have a shell script named basic

- Starts with `#!/bin/bash` (she-bang slash bin slash bash slash) - shebang line
- To run script `basic`: `./basic`.
  - Must put path to file since bash will only look through your `$PATH`
  - Can also use `$(pwd)/basic`

Command line arguments

- `$ ./script arg1 arg2`
  - $1 contains arg1
  - $2 contains arg2
  - …
- `${#}` is the number of args to the script
- `$0` is the name of the script (what comes before the first command which is also `$1`)

Example

Assume we have a script isItAWord

```bash
#!/bin/bash

# Search for the first arg in the dictionary
# Redirect output to blackhole
# egrep is smart enough to know $1 should be expanded
egrep "^$1$" /usr/share/dict/words > /dev/null

if [ $? -eq 0 ]; then
    echo $1 is not a good password
else
    echo $1 might be a good password
fi
```

- **Notes**: User needs to escape whatever special characters that get used
- **Notes**: Every command exits with a status code. Zero for success, non-zero for failure
  - `$?` is the status code for the last process
- **Notes**: The square bracket are themselves a program try `[ 1 -ne 2 ] ; echo $?`
  - The white space in the square brackets are **important**
  - Name of program is `[`, the reset are all arguments
- Take a look at the shell scripts dir

Scripts

- See 1189/lectures/..../goodPasswordUsage
  - the functions (subroutines) are not type safe
- All variables are of type string

More examples

```
if [ condition ]; then
    block
elif [ condition ]; then
    block
else
    block
fi

# No spaces between variable assignment
x=1
while [ ${x} -le $1 ]; do
    echo ${x}
    x=$((x + 1))
done

for x in a b c d; do echo ${x}; done

for x in a b c d; do
  echo ${x}
done

# Rename all .cpp files to .cc

# Use globbing patterns to find files
for name in *.cpp; do
    # Remove the trailing pp and add a c
    mv ${name} ${name%pp}c
done
```

Testing

- Testing is not the same as debugging
- Testing does not guarantee correctness
  - Can prove presence of defects

Writing Tests

- Write small tests
- Check various classes of input
  - numeric ranges, positive, negative, zero
  - boundaries between ranges (edge cases)
  - simultaneous boundaries (corner cases)

C++

- Simula67 - first OO language
- Created `C with classes`
  - Became `C++`
- Including `iostream` gives us access to 3 i/o variables
  1. std::cin
  - Read from stdin
  - Use the input operator `>>`
  - `char x; std::cin >> x;`
  2. std:cout
  - Write from stdout
  - Use the output operator `<<`
  - `std::out << "Hello World";`
  3. std:cerr
  - Write to stderr
  - Same as std::cout

cin

- If a read fails due to invalid input OR if EOF is reached, variable will be set to zero
- Ignores white space
- If a read fail, the expression cin.fail() is true
- If the read fails due to EOF, both cin.fail() and cin.eof() are true
- c++ defines an automatic conversion from a istream type to booleans
  - cout and cerr have type ostream
- cin is true if the last read succeeded
  - instead of writing `cin.fail()` we can use `!cin`
- `cin >> i` will produce `cin` which is mapped to a boolean
- If a read fails, all subsequent reads will fail unless failure is acknowledged
  - use `cin.clear(); cin.ignore();`
    - acknowledged then throw away offending input
    - `cin.ignore()` ignores a single char, if “Hello” is typing in, then only “H” is ignored

Side note

- In c++, » can act as both the bit shift AND input operator
  - **operator overloading**

Compiling

`$ g++ -std=f++14 hello.cc -o myprog`

Note: Without `-o`, the default executable is `a.out`

More I/O, stream

Strings in C++

- C used null terminated char arrays
  - Manual memory management
- C++ provides a header
  - Automatic resizing as needed
  - `string s = "hello";`
    * `string s{"Hello"}:`
    * `"hello"` is a C style string since it is just an array of char
  - Compare with ==, ~=, <, >, <=, >=
  - Length: `str.length();`
  - Characters: `str[0];` (Same as C with char[])
  - `cin` will stop after reading seeing a space
  - `getline(cin, s)` will get the entire line until a `\n` is encountered

I/O manipulators (iostream, iomanip)

- Print `x` in hexadecimal
  - Remember the code is evaluated left-to-right
  - `cout << hex << x;`
  - Change how cout works
    * All future `cout` will print in hex
  - `cout << dec;`
    * Change `cout` back to decimal
  - `cout << showpoint << setprecision(3);`
    * Print to three decimal points

Reading/writing files

- The *stream abstraction* can work on other sources of data
- Header `<fstream>`
  - `ifstream` - read from files
  - `ofstream` - write to files
- Anything we can do with cin(type istream) we can do with myfile(type ifstream)
  - Same for cout & variable of type ofstream
- When is this file close?
  - myfile is stack allocated and thus destroyed when it goes out of scope
  - When it is destroyed, it closes the file

Reading/writing to strings

- Header `<sstream>`
  - istringstream – read from strings
  - ostringstream – write to strings
- `buildStrings.cc`
  - Using ostringstream and writing to a string

```
#include <iostream>
#include <string>
#include <sstream>
using namespace std;

int main () {
  ostringstream ss;
  int lo {1}, hi {100};
  ss << "Enter a # between " << lo << " and " << hi;
  string s {ss.str()};
  cout << s << endl;
}
```

- `getNum.cc`
  - Insist user inputs a number

```
#include <iostream>
#include <string>
#include <sstream>
using namespace std;

int main () {
  int n;
  while (true) {
    cout << "Enter a number:" << endl;
```

```
    string s;
    cin >> s;
    // if (cin == EOF) return;
    istringstream ss{s};
    if (ss >> n) break;
    cout << "I said, ";
  }
  cout << "You entered " << n << endl;
}
```

- readIntsSS.cc

```
#include <iostream>
#include <sstream>
using namespace std;

int main () {
  string s;
  while (cin >> s) {
    istringstream ss{s};
    int n;
    if (ss >> n) cout << n << endl;
  }
}
```

Default Arguments

- New in C++

```
void printFile(string file = "myfile.txt") {
  ifstream f {file};
}

printFile();
printFile("hello.txt"):
```

- Parameters with default arguments **must** appear last
  - `void test(int num = 5, string str) {}`
    - **Illegal**
  - `void test(int num = 5, string str = "foo") {}`
    - **Legal**
    - `test(10, "bar");`
    - `test(10);`
    - `test();`
    - ^ Legal
    - This is illegal: `test("bar");` and `test(,"bar");`
  - Functions can have the same name in C++

    * Function overloading
    * Looks at function signature (looks at parameters types and/or number of parameters)
    * Differing on return type is not enough
    * default args act as if the function with that param only was implemented so it conflicts if you try to overload it

```
int negate(int i) { return -i; }
bool negate(bool b) { return !b; }
```

Review

- You can also do operator overloading
  - `cin >> x` where x can be `int` or `bool` or something else
    * `operator>>(cin, x);` is the same as before. Operator is implemented as a functon

Structs

```
struct Node {
  int data;
  struct Node *next;
};

//or

struct Node {
  int data;
  Node *next;
};

Node n = { 2, NULL };

//or

Node n = { 2, nullptr }; // c++11 and onwards
```

Note: The struct keyword above is optional in this line `struct Node *node;`

Constants

```
const int MAX = 100;
const Node n1{ 2, nullptr }; // Cannot change the fields of n1
int x = 10;

const int *p = &x; // p is a pointer to an constant int
p = &y; // VALID
*p = 10; // INVALID

int * const p1 = &x; // p is a constant pointer to an int;
p = &y; // INVALID
*p = 10; // VALID
```

```
// pass-by-value
void inc(int n) {
  n = n + 1;
}

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

```
// pass-by-pointer
void inc(int *n) {
  *n = *n + 1;
}

int x = 5;
inc(&x):
cout << x; // prints 6
```

pass-by-reference

- Lvalue Reference
- Two types of references

```
int y = 10;
int &z = y; // z is an lvalue reference to y

z = 15; // note not *z = 15;
// Changes the value of y

int *p = &z; // p now points to y
sizeof(z); // gives the size of y
```

- An lvalue reference is a constant ptr with automatic dereferencing
- z will continue to "point" to y forever
- `z` has no concept of itself, it is all changed at compile time
  - Thus, lvalue reference is just syntactic sugar

Things you cannot do

- Cannot leave it uninitialize
  - Since it is a const ptr under the hood
- Must initialize with an `lvalue`
  - `lvalue`: Something that can appear on the LHS of an assignment. Something that has a memory address.
    - `int &x = 5; // illegal since 5 cannot have a memory address, rvalue`
    - `int &x = y + z; // illegal since "y + z" cannot have a memory address, rvalue`
    - `int &x = y; // legal since "y" has a memory location`
- Cannot create a pointer to a reference
- Cannot create a reference to a reference
- Cannot create an array of references

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

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

- The signature of `cin`:

```
// @param in: Cannot make a copy of a stream so we must pass-by-reference
//            Want changes to stream to be visible when function is done
// @return: cin >> x evaluates to cin, so it must return {@param in}
std::istream &operator>>(std::istream &in, int &data) {
  // Method body
}
```

- Use case for big structs

```
struct RellyBig {  };

void f(ReallyBig rb) {
  // Copy of ReallyBig is created, very expensive
  // Cannot modify the original copy
}
```

```
// C++ style
void g(ReallyBig &rb) {
  // pass-by-reference to avoid copying the whole struct
  // Can modify the original copy
}

// C style
void h(ReallyBig *rb) {
  // pass-by-pointer to avoid copying the whole struct
  // Can modify the original copy
}

// C++ style
void l(const ReallyBig &rb) {
  // pass-by-reference to avoid copying the whole struct
  // Cannot modify the original copy
}

g(5); // ILLEGAL
int y = 5;
g(y + y); // ILLEGAL
g(y); // LEGAL

// BUT IN C++11
// Since the pass-by-reference param is const
l(5); // LEGAL
int y = 5;
l(y + y); // LEGAL
l(y); // LEGAL
```

Note: Always prefer passing by reference to const

Dynamic Memory

```
// C style (not valid in this class
int *p = malloc(5);
free(p);

// C++ style
struct Node {
  int data;
  Node *next;
}
Node *np = new Node;
```

```
// np is a variable on the stack that contains the address to memory in the heap
// When np goes out of scope (popped off the stack), the heap memory will continue to exist
// Use this to "free" the memory in the heap allocated for np
delete np; // np must be created from a call "new" call
```

Arrays

```
int *arr = new int[10]; // Allocated from the heap, same as before
delete [] arr; // Syntax for deallocating arrays
delete arr; // WILL COMPILE but will cause memory leak
```

Review

```
Node *np = new Node;
delete np;

int *arr = new int[10];
delete [] arr;

// A copy of n is returned and thus will be inefficient
Node getANode() {
  Node n;
  return n;
}

// BAD
// The pointer is returned which is more efficient than above.
// However, it is a dangling pointer since it points to memory than is not
//   yours to access
// n is allocated on the stack, never return a prt/reference to data the
//   function allocated on the stack
Node *getANode() {
  Node n;
  return &n;
}

// Correctly returns a pointer (not a reference) to heap allocated memory
Node *getANode() {
  Node *np = new Node{...};
  return np;
}
```

Operator Overloading

- Idea: we can give meaning to C++ operators for user-defined types

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

Vec vA = new Vec{1, 2};
Vec vB = new Vec{1, 2};

// Does not compile
Vec v1 = vA + vB;

// Make an overload for the plus operator for struct Vec
Vec operator+(const Vec &v1, const Vec &v2) {
  Vec newV{v1.x + v2.x, v1.y + v2.y};
  return newV;
}
// This DOES compile
Vec v = vA + vB;

// Another example
// Commutivity does NOT apply
Vec operator*(const int k, const Vec &v1) {
  // C++ compiler is smart as fuck
  // It will know to construct a Vec
  return {k * v1.x, k * v1.y};
}
// Works
Vec v = 3 * vA;
// Does NOT work
// Need to write a function with the following signature
// Vec operator*(const Vec &v1, const int k);
Vec v = vA * 3;
```

Another set of examples

```
Grade g = new Grade{52};
cout << g << endl;

ostream &operator<<(ostream &out, const Grade &g) {
  out << g.grade << "%";
```

```
  return out;
}
```

```
Grade g = new Grade{52};
cin >> g;

istream &operator>>(istream &in, Grade &g) {
  in >> g.grade;
  if (g.grade < 0) g.grade = 0;
  if (g.grade > 100) g.grade = 100;
  return in;
}
```

C/C++ Preprocessor

- A program that runs before the compiler gets the code
- Has the ability to change code
- All code that starts with `#` is a preprocessor directive
  - `#include <...>`
    - Look first in standard library for file inside `<>`
  - `#include "..."`
    - look in local directory
- To only run the preprocessor
  - `g++14 -E -P {file}`

```
#define VAR VALUE
// Preprocessing directive to replace all instance of VAR with VALUE

#define MAX 100
int array[MAX];
```

- `#define` are used for conditional compilation
  - Supposed different clients want different security strength

```
#define SECLEVEL 5 // could be something else
#if SECLEVEL == 1
short int
#elif SECLEVEL == 2
long long int;
#endif
publicKey;
```

Example

```
#define LEN 2;

int main() {
  #if LEN == 1;
  short int
  #elif
  long int
  #endif
  myNum;
}
```

Compiles to:

```


int main() {



  long int

  myNum;
}
```

Problems

- Requires manual changing of `LEN` in the define statement

Solution

- We can set preprocessor variables from the command-line

`g++ -DVAR=VALUE {file}`

Eg.

`g++ -LEN=1 {file}`

ifdef

```
#define BANANA

// true if BANANA is defined
#ifdef BANANA
```

```
#endif

// true if BANANA is not defined
#ifndef BANANA
#endif
#define ever ;;

for (ever) {
  // code
}

// COMPILES TO...
for (;;) {
  // code
}
```

Tricks to comment out debug statements

```
#include <iostream>
using namespace std;

// file: debug.cc

int main() {
  #ifdef DEBUG
    cout << "setting x=1" << endl;
  #endif
  int x = 1;
  while (x < 10) {
    ++x;
    #ifdef DEBUG;
      cout << "x is now " << x << endl;
    #endif
  }
  cout << x << endl;
}
```

- Show the debug logs
  - `g++ -DDEBUG debug.cc`
- Don't show debug logs
  - `g++ debug.cc`

Seperate Compilation

- Interface (.h file)
  - Type definitions
  - function declarations
- Implementation (.cc file)
  - function implementations
- Look at files in dir `~/cs246/1189/lectures/c++/separate/example1/`

Compilation for many files

- Can use globbing pattern `g++ *.cc`
- Better to not use globbing `g++ main.cc vec.cc`
- header files are **never** compiled, they are included and compiled due to the preprocessing directive

  vim:tw=78:ts=2

Separate Compilation

- Don't want to compile each file cuz that shit is tedious
- If you compile dependant files with two linux commands, you can have linker error
- By default g++ tries to compile, link and produce an executable
  - `g++ -c main.cc` to only compile `main.cc`, but it will not create an executable
    - Produces `main.o` which contains compiled code, and info about what is defined and what is required
  - `g++ main.o vec.o` will now do the linking process. Linker will simply check that all *promises* about what each program requires is kept
    - Allows you to recompile only the files that have changed
    - Means you need to keep track of what needs to be recompiled

make

Makefile

- Specify project dependencies
- Compile command

```
# The full program
myprogram: main.o vec.o
```

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

# Recompile vec.o when vec.cc or vec.h changes
vec.o: vec.cc vec.h
  g++ -std=c++14 -c vec.cc

# Recompile main.o when main.cc or vec.h changes
main.o: main.cc vec.h
  g++ -std=c++14 -c main.cc

.PHONY: clean

clean:
  rm *.o myprogram
```

- Make knows what needs to be recompiled by looking at linux time stamp

  We can make the Makefile a bit nicer

```
# Throw stuff into variables
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
# Can omit the recompile lines since make is smart

main.o: main.cc vec.h
  # Code here will be executed as a shell command, try echo "hello";pwd

.PHONY: clean

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

- make is not C/C++ specific
  - All it does is look at the time stamp of files, and if it is old, then recompile based on the commands we provide
- We can use `g++ -MMD vec.cc` to create a `.d` file which is a dependency file
  - It will generate a file with `vec.o: vec.cc vec.h`

```
CXX = g++
CXXFLAGS = -std=c++14 -Wall -MMD
```

```
EXEC = myprogram
OBJECTS = main.o vec.o
DEPENDS = ${OBJECTS:.o=.d}

${EXEC}: ${OBJECTS}
    ${CXX} ${CXXFLAGS} ${OBJECTS} -o ${EXEC}

-include ${DEPENDS}

.PHONY: clean

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

- Use this final example for make file

  TODO: Figure out exact syntax for files in nested sub directories

Include guards

- How to avoid multiple includes
  - Avoid conflicting definitions
- We use include guard to avoid this:

  vec.h

```
#ifndef VEC_H
#define VEC_H

struct Vec {
  int x;
  int y;
}

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

- You can now `#include "vec.h"` in multiple files and don't have a compiler error
- Must be unique to the program, so use a form of the name of the file
  - **Convention**: upper case, replace dot with an underscore
- *Include guards* only in header files

Some rules

- Never compile header files
- Never include .cc files
- Do not put `using namespace std;` in header files
  - In the industry, you won't even put this in .cc files to avoid cluttering the namespace
    - Always use fullname in real life, eg. `std::string`

C++ Classes

student.h

```
struct Student {
  int assns, mt, final;
  float grade(); // declaration
};
```

student.cc

```
#include "student.h"

float Student::grade() {
  return assns * 0.4 + mt * 0.2 + final * 0.4;
}
```

- `::` is the scope resolution operator
  - `Student::` means that in the scope of the Student struct there is a function grade
- Use the function by using the dot operator
  - `student.grade()` will evaluate to a float
- In OOP, a class is a struct type that can contain functions
  - These functions are called **methods** or member function
- An instance of a class is called an object
- Within a method, you have access to the fields of the object on which this method was called
- Within a method, you have access to `this` which is a pointer to the object on which this method was called
  - In the above case, `this == &student`

student.cc explicit version

```
#include "student.h"

// Super verbose and not necessary
// Only used if you have conflicting names with a parameter
float Student::grade() {
```

```
  return this->assns * 0.4 + this->mt * 0.2 + this->final * 0.4;
}
```

Initializing Objects

```
// C-style initializing
// Integers below are constants
Student student{60, 50, 70};
```

- We can write special objects to construct objects
  - `constructors`

  student.h

```
struct Student {
  Student(int assns, int mt, int final);
};
```

  student.cc

```
Student::Student(int assns, int mt, int final) {
  this->assns = assns;
  this->mt = mt;
  this->final = final;
}
```

Initializing Objects (cont'd)

- `Student billy{70, 50, 75};`
  - Allocated on the stack **not** the heap
- If an appropriate constructor is present, it is called
- `Student billy = Student(80, 50, 75);`
  - Allocated on the stack **not** the heap

Uniform Initializing Syntax

1. `int x = 5;`
2. `int x(5);`
3. `string s = "hello";`
4. `string s("string");`
5. `int x{5};`
6. `string s{"hello"};`
7. `Student billy{...};`

- You can always use curly braces (5, 6, 7) since C++11.

- They added it in C++11 to have a single initializing syntax that is can be used everywhere

```
// pbilly is allocated on the heap NOT the stack
Student *pbilly = new Student{85, 50, 75};
// Some code
delete pbilly;
```

Advantages of writing constructors

- Execute arbitrary code
- Default arguments
- Overload constructors
- Sanity checks
- Anything you can do with methods, you can do with a constructor

```
// Note: Default values go in the declaration
struct Student {
  Student(int assns = 0, int mt = 0, int final = 0);
  int assns = 0;
  int mt = 0;
  int final = 0;
};

// This constructor has some sanity checks
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;
}

Student s1{50, 50, 50};
Student s2{50, 50}; // {assns, mt}
Student s3{50}; // {assns}
Student s3; // use ALL default values (note: no curly braces)
```

- Every class comes with a default (0 param) constructor which initializes fields that that are objects by calling its default constructor

```
struct A {
  int x;
  Student y;
  Vec *z;
};
A a; // Calls the default constructor
// x, z are uninitialized
// y is initialized
```

- **Note**: Above there is no constructor declared, so `A a;` calls the default constructor
- As soon as you implement any constructor, you lose the default constructor and C-stlye initialization

```
// No default constructor
struct A {
  // Bad style, used for brevity
  A::A(int x, Vec *z) {
    this->x = x;
    this->z = z;
  }
  int x;
  Vec *z;
};

A a; // Won't compile (if you have default params it will compile)
A b(5, nullptr); // Will compile
```

- Initializing constant/reference fields

```
// Option 1
// in class initialization
int z;
struct MyStruct {
  const int myConst = 10;
  int &myRef = z;
};

// Option 2
// Do not do in class initializing
// Rule: const/refs must be initialized before constructor body runs
// TODO
```

Steps for object construction

1. allocate space
2. field initialization
3. constructor body runs

- Lets hijack step 2: Member initialization List (MIL)

```
Student::Student(const int id, int assns, int mt, int final) :
  id{id}, assns{assns}, mt{mt}, final{final < 0 ? 0 : final} {
}
```

- MIL cna be used for ALL fields
- no need to use `this`

  - outside the braces the identifier is a field
  - inside the braces normal scope rules apply
- in the MIL field initialization occurs in field declaration order
- Initializing fields in the MIL *can be* more efficient than Initializing in the constructor body
  - It would get default value at step 2, then reassigned in constructor body.

Initializing objects as copies of others

```
Student billy{80, 50, 75};
Student bobby{billy}; // copy constructor (get for free)
```

A class comes with...

1. default constructor
2. copy constructor
3. copy assignment operator
4. destructor
5. move constructor (C++ onwards)
6. move assignment operator (C++ onwards)

- 2-6 are called the big five **(1/4 of the midterm is on this)**

```
// The free copy constructor
Student::Student(const Student &other) :
  assns{other.assns}, mt{other.mt}, final{other.final} {
}
```

- Sometimes the free copy constructor does not work the way we want it
  - So we can override it

```
struct Node {
  int data;
  Node *next;
  Node(int data, Node *next);
  Node(const Node &other);
};

Node::Node(int data, Node *next) :
  data{data}, next{next} {
}

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

```
Node *np = new Node{1, new Node{2, new Node{3, nullptr}}}; // heap
```

- *Stack* np -> | *heap* -> Node(1) -> Node(2) -> Node(3) -> nullptr
- `Node m{*np};`
  - *Stack* m(1) -> | *heap* -> Node(2) *same node as above*
- `Node *n1 = new Node{*np};`
  - *Stack* n1 -> | *heap* -> Node(1) -> Node(2) *same node as above*
- This is not a true copy of a linked list since it should share nodes
  - This is called a **Shallow copy**
- What we want is a **deep copy**

```
// Deep copy
// Recursively call the copy constructor

// Incorrect: it has no base case for next
//   segmentation fault
Node::Node(const Node &other) :
  data{other.data}, next{*other.next} {
}

// Correct: has base case
Node::Node(const Node &other) :
  data{other.data}, next{other.next ? new Node{*other.next} : nullptr} {
}
```

A copy constructor is called when...

1. an object is constructed as a copy of another
2. pass by value
3. returning by value

- Due to #2, the param of a copy constructor **must** be a reference
  - or else we will have infinite recursion

Last time

- Copy constructor: used to create copies of objects
- Note on single parameter constructors

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

```
Node n{4};
Node n = 4; // Calls the syntax above

void foo(Node n) {
}

foo(4); // Legal

// Explains why this is legal:
string s = "Hello";
string s{"Hello"}; // Same as above
```

- One parameter constructors create implicit/automatic conversions
- Disallow the single parameter conversion, use the `explicit` keyword:

```
struct Node {
  explicit Node(int data);
};

Node n = 4; // Illegal now

void foo(Node n) {
}
foo(4); // Illegal
foo(Node{4}); // Legal
```

Destructor

- A method called the destructor runs whenever objects are destroyed
- When is it called
  - stack allocated -> when it goes out of scope
  - heap allocated -> destroyed when you call `delete` on a pointer to that object
- Steps for object destruction
  1. destructor body runs
  2. fields that are objects are destroyed in reverse declaration order
  3. deallocate space
- The destructor body of the "free" destructor is empty
  - This might not be sufficient

```
// ptr np -> Node(0) -> Node(1) -> Node(2)
// stack     heap       heap       heap

delete np;
```

- Node(0) is deallocated

- Node(1) is not deallocated
- Node(2) is not deallocated
- Node(1) and Node(2) are both leaked
- We need to recursively deallocate the nested nodes

```
struct Node {
  ~Node(); // no parameters and cannot be overloaded
  Node n;
  int data;
};

Node::~Node() {
  delete n; // recursively calls the destructor
}
```

Note: You are able to call `delete` on `nullptr` **Note:** The destructor above will crash if `n` is allocated on the stack

Copy assignment operator

```
Student billy{80, 50, 75};
Student bobby = billy; // copy constructor
Student jane;
jane = billy; // copy assignment operator
jane.operator=(billy); // Same as above
```

```
// Return type is explained at bottom of code
// Example of a question on the midterm
// 5/5 on exam for this question
Node &Node::operator=(const Node &newN) {
  data = newN.data;

  // see #1 below for explanation
  if (this == &newN) return *this;

  delete n;

  // relying on the copy constructor to make sure a true copy is made
  // See #2 below for when this fails
  n = newN.n ? new Node(*newN.n) ? nullptr;

  return *this;
}

Node n1{1};
```

```
Node n2{2};
Node n3{3};

n1 = n2;
n1.operator=(n2);

n1 = n2 = n3; // we need to return a Node from the operator=() method
```

1. Self assignment could be problematic:

```
Node n{1, new Node{2, nullptr}};
n = n;
```

- The code above will delete `this->n` and then try accessing it through `newN.n` which will be a dangling pointer since you no longer own that memory location
  - Use self-assignment check to avoid dangling pointer

2. If call to `new Node()` fails (heap has no more memory), then `n` is not assigned, but it was already deleted and thus has become a dangling pointer.

- Can be fixed with the following implementation of the copy assignment operator

```
// 5/5 on exam for this question, better than above
// Important on final
Node &Node::operator=(const Node &newN) {
  if (this == &newN) return *this;
  Node *tmp = n;

  // If this line fails, no lines after this will be run
  n = newN.next ? new Node(*newN.next) : nullptr;

  data = newN.data;
  delete tmp; // delaying the deleting to see if the above line fails
  return *this;
}
```

Alternate method (need to know both this and the above method)

Copy and Swap Idiom

node.h

```
struct Node {
  void swap(Node &other);
};
```

node.cc

```
#include <utility>

void Node::swap(Node &other) {
  std::swap(data, other.data);
  std::swap(next, other.next);
}

Node &Node::operator=(const Node &other) {
  Node tmp{other}; // deep copy. Memory allocated on the stack
  swap(tmp);
  return *this;
}
```

Complicated example

```
// pass-by-value
Node plusOne(Node n) {
  for (Node *p {&n}; p ; p = p->next) {
    ++p->data;
  }
  // return by value
  return n;
}

Node n{1, new Node{2, nullptr}};
Node n2{plusOne(n)}; // 6 calls to copy constructor
```

- C++11 introduced rvalue reference
  - `Node &other` - lvalue reference
  - `Node &&other` - rvalue reference
- rvalue reference is a reference to a tmp value (one that is about to be destroyed)

Move constructor

- Takes one parameter - rvalue reference

```
other -> Node(2) -> Node(3)
this -> Node(2) -/^
```

- `Node(2)` is copied, `Node(3)` is shared
- **Shallow copy**

```
Node::Node(Node &&other) :
    data{other.data},
    next{other.next}, {
  // Old Node(2) above has the ptr removed
  other.next = nullptr;
}
```

Move assignment operator

```
Node m{1, new Node{2, nullptr}}; // notice Node(1) is stack while Node(2) is heap
m = plusOne(m); // m would otherwise get copied then destroyed
```

Memory diagram for above code

```
other -> Node(2) -> Node(3)
m -> Node(1) -> Node(2)

AFTER move assignment operator

other -> Node(1) -\ Node(3)
m -> Node(2) -/^ Node(2)
```

Move assignment operator

```
Node &Node::operator=(Node &&other) {
  std::swap(other); // copy-swap idiom from oct_16.md
  return *this;
}
```

- If a move constructor/operator= is available, it will be used whenever the RHS is an rvalue reference
- The default move constructor/operator= go away if you write any of the big 5

Rule of 5

- If you need to implement any of the following, then usually need to implement all 5:
  1. copy constructor
  2. copy operator
  3. destructor
  4. move constructor
  5. move operator=

Copy/move elision

```
Vec makeVec() { return {0, 0}; }
Vec v = makeVec();
```

- Intuitively either move or copy constructor would be called
  - But it doesn't
- The compiler optimizes this so Vec{0, 0} is created in the space for variable `v`
- C++ *allows* compilers to avoid calling copy/move constructors even if this would change program behaviour
  - You are able to turn off optimizations: `g++14 -fno-elide-constructors`

Operators - functions or methods?

- `operator=` - must be implemented as a method
  - `n1 = n2` — `n1.operator=(n2)`
- The LHS is represented by `*this`

```
Vec v1 = {1, 2};
Vec v2 = {3, 4};
v1 + v2;
v1 * 5;
5 * v2;
```

```
struct Vec {
  int x,y;
  Vec operator+(const Vec &);
  Vec operator*(const int);
};

// Cannot go inside the Vec struct
Vec operator*(const in, const Vec &);
```

These are **methods**, these are **not functions**

```
Vec Vec::operator*(const int k) {
  return { x * k, y * k };
}

ostream &Vec::operator<<(ostream &out) {
  out << x << "  " << y;
  return out;
}
```

- The above must be called as follows: `v1 << cout` and `v2 << (v1 << cout)`

– Don't do this

These are **functions**

```
Vec operator*(const int k, const Vec &v) {
  return { v.x * k, v.y * k };
}

ostream operator<<(ostream &out, const Vec &v) {
  out << v.x << "  " << v.y;
  return out;
}
```

- However, the following must be implemented as methods (not too important)
  1. operator=
  2. operator[]
  3. operator->
  4. operator()
  5. operator T()

Arrays of objects

```
struct Vec {
  // free no parameter constructor goes away
  // Vec v[10] also doesn't work since no default constructor
  Vec(int x, int y) : x{x}, y{y};
};
```

- How to fix?
  1. implement a 0 parameter constructor
  2. for stack arrays, use array initialization: `Vec v3 = {Vec{0, 0}, Vec{0, 0}, Vec{0, 0}}`
  3. use array of pointers to objects

number 3 from above

```
// stack
Vec *arr[3];

// heap
Vec **arr = new Vec*[3];
array[0] = new Vec{0, 0};
for (int i = 0; i < 3; ++i) {
  delete array[i];
}
delete [] array;
```

const methods

```
struct Student {
  int assns, mt, final;
  float grade() {
    return 0.4 * assns + 0.2 * mt + 0.f * final;
  }
};

const Student billy{80, 50, 75};
billy.grade(); // won't compile since grade() doesn't promise it won't modify fields

// Must use const as follows
struct Student {
  int assns, mt, final;
  float grade() const {
    return 0.4 * assns + 0.2 * mt + 0.f * final;
  }
};
```

- a const method promises to not change field value of `*this`
- const objects can only call const methods
- **const** applies to the thing on the left, unless there is nothing on the left, then it applies to the thing on the right
  - `const int k` == `int const k`

Invariants and Encapsulation

Invariants

- In Node class `next` always points to heap memory or is nullptr
  - `Node n{...};`
  - `Node m{2, &n};` // should crash since n is on the stack not the heap

Encapsulation

- treat object as black boxes
- hide the implementation
- provide and interface (select number of methods)

```
// by default for struct, things are pubic
struct Vec {
  Vec(int x, int y);
```

```
  private:
    int x, y;
  public:
    Vec operator+(const Vec &other) {
      return {x + other.x, y+other.y};
    }
};

Vec v{1, 2}; // legal
Vec v1 = v + v; // legal
std::cout << v.x << v.y << std::endl; // illegal
```

- **Advice**: at a minimum, keep all fields private (use setters/getters)
- In a `class`, default visibility is `private` not `public`, prefer `class` over `struct`

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

Node Invariant

- Prevent client code from creating Nodes, and accessing the next ptr
  - Create a wrapper `List` class which will have sole access to creating nodes or updating the next ptr

  list.h

```
class List {

  struct Node;
  Node *head = nullptr;

  public:
    void addToFront(int);
    int ith(int); // returns the ith data value at the ith node
    ~List();
};
```

  list.cc

```
struct List::Node {
  int data;
  Node *next;
```

```
  ~Node() {
    delete next;
  }
}

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

void List::addToFront(int n) {
  head = new Node{n, head};
}

/**
 * The ith node must exist
 */
int List::ith(int i) {
  Node *cur = head;
  for (int j = 0; j < i && cur; ++j, cur = cur->next);
  return cur->data;
}
```

- Suppose we want to print the list, what is the Big O time complexity
  - If done outside the List class, it will be O(n^2)
  - If done inside the List class, it can be O(n)
    - Avoid this since you will need a custom method for each thing, thus we can create some abstract list function and iterators

Design Patterns

Iterator Design Pattern

- For O(n) traversal, we will need to track where we are inside the List
  - **Challenge**: Without using a public Node ptr
  - **Solution**: Create another class that keeps track of where we are, but does so privately
    - This Iterator class will act as an abstraction of a ptr inside the List

```
// arr is an array
for (int *p = arr; p != arr + arraySize; ++p) {
}
```

```
class List {

  struct Node;
```

```
  Node *head = nullptr;

  public:
    class Iterator {

      Node *cur;

      public:

        Iterator(Node *cur) : cur{cur} {
        }

        int &operator*() const {
          return cur->data;
        }

        // unary prefix
        Iterator &operator++() {
          cur = cur->next;
          return *this;
        }

        // unary postfix
        //Iterator &operator++(int) {
          // ...
        //}

        bool operator!=(const Iterator &other) {
          return cur != other.cur;
        }

    };
    void addToFront(int);
    int ith(int); // returns the ith data value at the ith node
    ~List();

    Iterator begin() {
      return Iterator{head};
    }

    Iterator end() {
      return Iterator{nullptr};
    }
};
```

client code to interact with the above code snippet

```cpp
list lst;
list.addtofront(2);
list.addtofront(3);
// order o(n) traversal
for (auto it = list.begin(); it != list.end(); ++it) {
  cout << *it << endl;
  *it = *it + 1; // changes the node data
}
```

Nicer way of writing the above

```cpp
list lst
list.addtofront(2);
list.addtofront(3);
// order o(n) traversal
for (auto x : lst) {
}
for (auto &x : lst) {
}
```

Last time

```cpp
List lst;
lst.addToFront(1);
for (List::Iterator it = list.begin(); it != list.end(); ++i) {
  cout << *it << endl;
}
for (auto it = list.begin(); it != list.end(); ++i) {
  cout << *it << endl;
}
```

```cpp
int y = 10;
// similar to var in kotlin or js
auto x = y; // compiler will know y is type int, it will make x the same type as y
```

- C++ has builtin support for the iterator design pattern (so does Java)
  - **Range-based for loops**

```cpp
for (auto n : lst) {
  cout << n << endl; // n is a copy of whatever operator* returns
}
for (auto &n : lst) {
  cout << n << endl; // n is a reference
  ++n; // valid now
}
```

- To use a range-based for loop for a class, MyClass

1. MyClass must implement `begin()` and `end()` that must return some Iterator (name of iterator class doesn't matter) object
2. That Iterator class must implement `operator*`, `operator++`, `operator!=`

friend keyword

- Constructor for List::Iterator is public, List::begin && List::end call it
- Want to make List::Iterator constructor private, but then begin && end cannot access it
- Iterator can declare List to be a friend

```
class List {
  class Node;
  Node *theList = nullptr;

 public:
  class Iterator {
    Node *p;
    explicit Iterator(Node *p);
   public:
    int &operator*() const;
    Iterator &operator++();
    bool operator==(const Iterator &other) const;
    bool operator!=(const Iterator &other) const;
    friend class List;
  };

  Iterator begin();
  Iterator end();

  void addToFront(int n);
  int ith(int i);
  ~List();
};
```

- `friend` breaks encapsulation
  - Thus, try to use `friend` as rare as possible
- Keep fields private, use setters/getters for them

```
class Vec {
  int x, y;
  public:
    int getX() const { return x; }
    int getY() const { return y; }
```

```
    void setX(int x) { this->x = x; }
    void setY(int y) { this->y = y; }

    friend ostream &operator<<(ostream &, const Vec &);
};

ostream &operator<<(ostream &out, const Vec &v) {
  out << v.x << v.y; // won't compile
  return out;
}
```

- Single point of entry into the variable
    - Useful for invariance
- This comes default in a bunch of languages, see `Kotlin`

mutable

```
struct Student {
  int assns, mt, final;
  mutable int numCalls = 0;
  float grade() const {
    ++numCalls; // legal modification since numCalls is mutable
    return 0.4 * assns + 0.2 * mt + 0.4 * final;
  };
};
```

static

We can make fields static

- A static field belongs to the class & not each object of the class

```
struct Student {
  int assns, mt, final;
  static int numObjects; // in-class initialisation is not allowed
  Student() : ..... { ++numObjects; } {
  }
  float grade() const {
    ++numCalls; // legal modification since numCalls is mutable
    return 0.4 * assns + 0.2 * mt + 0.4 * final;
  };
};
```

- static fields must be defined external to the file defining the class

Student.cc

```
int Student::numObjects = 0; // now memory is allocated
```

We can make static member functions

```
struct Student {
  int assns, mt, final;
  static int numObjects; // in-class initialisation is not allowed
  Student() : ..... { ++numObjects; } {
  }
  float grade() const {
    ++numCalls; // legal modification since numCalls is mutable
    return 0.4 * assns + 0.2 * mt + 0.4 * final;
  };
  // Static functions not static methods since it doesn't belong to an instance of a class
  // Does not have a this ptr
  static int objCreated() {
    return numObjects;
  }
};
```

- Call be called as follows `int num = Student::objCreated()`
  - Don't need an instance of the class Student

System Modelling

- A good design requires:
  1. determining the major abstractions (classes)
  2. relationship between classes
- **UML**: Unified Modelling Language
  - Standard

Modelling a class

1. Name of class
2. Fields
3. Methods

   - is for private, + is for public

name of class	Vec
fields	- x: Integer

name of class	Vec
fields	- y: Integer
—	——
methods	+ Vec(Integer, Integer)
methods	+ getX(): Integer

- Relationship 1: Composition

```
class Vec {
  int x, y;
  public:
    Vec(int x, int y);
};

class Basic {
  Vec v1, v2;
  public:
    Basis() : v1{0, 0}, v2{1, 1} {
    }
};

Basic b;
```

- Embedding an object (Vec) within another (Basic) is composition. We say: `Basic owns-a Vec`
  - If A owns-a B, then B does not exist outside A
  - copying A, copies B (deep copies)
  - destroying A, destroys B
- For the List class, List owns-a Node

Showing the owns-a relationship

Draw each class as a table, similar to above. Star means owns-a

`Basic *->v1,v2-> Vec` # Composition, Aggregation, Inheritance

- Composition creates a OWNS-A relationship

Car **OWNS-A** CarParts	Catalog **HAS-A** CarParts
	A class A HAS-A class B B is not copied if A is copied (shallow copy) B is not destroyed if A is

```
// Has parts, but the parts are not super dependant on the Catalog
// ie. Catalog is not responsible for copying/destroyed Part array
```

```
class Catalog {
  Part *[10]; / ptr to objects
};
```

```
UML Diagram
===========

Catalog []-> parts[10] -> Part
```

- The [] refers to HAS-A

Inheritance

Book
title: String author: String numPages: Integer

Text
title: String author: String numPages: Integer topic: String

Comic
title: String author: String numPages: Integer hero: String

- How to create an array for different types of Books?
- In C, use union types, void ptrs

1. Text **IS-A** Book (with an additional topic)
2. Comic **IS-A** Book (with an additional hero)

Book
title: String author: String numPages: Integer

Text Inherits from Book

Text
topic: String

Comic Inherits from Book

Comic
hero: String

- Book would be the base/superclass/parent, while text and comic is the derived/subclass/child
- **Note**: Use an arrow in UML to show inheritance

Book class

```
// book.h
class Book {
  std::string title, author;
  int length;
 public:
  Book(const std::string &title, const std::string &author, int length);
  std::string getTitle() const;
  std::string getAuthor() const;
  int getLength() const;
  bool isHeavy() const;
};

// book.cc
Book::Book(const string &title, const string &author, int length):
  title{title}, author{author}, length{length} {}

string Book::getTitle() const { return title; }
string Book::getAuthor() const { return author; }
int Book::getLength() const { return length; }
bool Book::isHeavy() const { return length > 200; }
```

Text class

```
// text.h
class Text: public Book {
  std::string topic;
 public:
  Text(const std::string &title, const std::string &author, int length, const std::string &t
  std::string getTopic() const;
};
```

```
// text.cc
Text::Text(const string &title, const string &author, int length, const string &topic):
  Book{title, author, length}, topic{topic} {}

string Text::getTopic() const { return topic; }
```

Comic class

```
// comic.h
class Comic: public Book {
  std::string hero;
 public:
  Comic(const std::string &title, const std::string &author, int length, const std::string &
  std::string getHero() const;
};

// comic.cc
Comic::Comic(const string &title, const string &author, int length, const string &hero):
  Book{title, author, length}, hero{hero} {}

string Comic::getHero() const { return hero; }
```

- public inheritance
- Derived classes inherit members (fields/methods) from the base class
- Any method that we can call on Book, can be called on Text/Comic
- It is because of inheritance that ifstream/istringstream objects like istream objects

Inheriting private members

- Text inherited the private fields from Book

```
int main() {
  Text t = ...;
  t.author = ....; // won't compile since author is private
}
```

- But what if we are inside the Text class

```
// WON'T COMPILE
Text::Text(const string &title, const string &author, int numPages, const string &topic) :
  title{title}, author{author}, numPages{numPages}, topic{topic} {
}
```

- This will not compile

   1. These inherited fields are private && MIL can only refer to fields declared in the class
   2. Steps of Object creation (for inherited classes):
   3. space is allocated - *MIL*
   4. superclass part is constructed (this step will fail) - *MIL*
   5. subclass field initialization
   6. constructor body runs
- Step 2 fails as Book does not have a default constructor

```
Text::Text(const string &title, const string &author, int numPages, const string &topic) :
  Book(title, author, numPages), topic{topic} {
}
```

Protected access

```
class Book {
  protected:
    string author;
};

class Text : public Book {
  void addAuthor(string auth) {
    author += auth;
  }
};
```

- Protected gives access to class and subclass
- Private > Protected usually to maintain invariant
- A class is responsible to maintain its invariant
- Protected breaks encapsulation (like friend)
- Better to keep fields private and provide protected methods for subclass

```
class Book {
  string author;
  protected:
    void addAuthor(string auth) {
      // invariant check
      author += auth;
    }
};
```

Method overriding

- `isHeavy`
  - Book > 200
  - Text > 300
  - Comic > 30

- `isHeavy` should be overridden in each subclass

  Book class

```
// Book.h
class Book {
  std::string title, author;
  int length;
 protected:
  int getLength() const;
 public:
  Book(const std::string &title, const std::string &author, int length);
  std::string getTitle() const;
  std::string getAuthor() const;
  bool isHeavy() const;
};

// Book.cc
Book::Book(const string &title, const string &author, int length):
  title{title}, author{author}, length{length} {}

int Book::getLength() const { return length; }
string Book::getTitle() const { return title; }
string Book::getAuthor() const { return author; }
bool Book::isHeavy() const { return length > 200; }
```

Comic class

```
// comic.h
class Comic: public Book {
  std::string hero;
 public:
  Comic(const std::string &title, const std::string &author, int length, const std::string &
  bool isHeavy() const;
  std::string getHero() const;
};

// comic.cc
Comic::Comic(const string &title, const string &author, int length, const string &hero):
  Book{title, author, length}, hero{hero} {}

bool Comic::isHeavy() const { return getLength() > 30; }
string Comic::getHero() const { return hero; }
```

Text class

```
// text.cc
class Text: public Book {
  std::string topic;
```

```
 public:
  Text(const std::string &title, const std::string &author, int length, const std::string &t
  bool isHeavy();
  std::string getTopic();
};

// text.h
Text::Text(const string &title, const string &author, int length, const string &topic):
  Book{title, author, length}, topic{topic} {}

bool Text::isHeavy() { return getLength() > 400; }
string Text::getTopic() { return topic; }
```

```
Book b{..., ..., 100};
b.isHeavy(); // false

Comic c{..., ..., 40};
c.isHeavy(); // true
```

- Making use of the **IS-A** relationship

```
Book b = Comic{..., ..., 40, "Batman"}; // legal
b.isHeavy(); // false
```

- Inherited fields in Comic get saved and that's what Book has, but the rest is discard (such as "batman")
- Called **object coercion/slicing**

Dynamic Dispatch

- Method Overriding

```
bool Book::isHeavy() { return numPages > 200; }
bool Text::isHeavy() { return numPages > 300; }
bool Comic::isHeavy() { return numPages > 30; }

Book b = Comic{..., ..., 40, ...};
b.isHeavy(); // calls Book::isHeavy() due to object slicing

Comic c{..., ..., 40, ...};
Comic *cptr{&c};
cptr->isHeavy(); // calls Comic::isHeavy()

Book *bptr{&c};
bptr->isHeavy(); // calls Book::isHeavy(), no slicing has occured
// Bool:isHeavy() is called since the compiler looks at the declared type of the ptr
```

- To change this behaviour, we want to use Dynamic Dispatch using `virtual`

```
// Book class
class Book {
  std::string title, author;
  int length;
 protected:
  int getLength() const;
 public:
  Book(const std::string &title, const std::string &author, int length);
  std::string getTitle() const;
  std::string getAuthor() const;
  virtual bool isHeavy() const;
};
```

```
// Comic class
class Comic: public Book {
  std::string hero;
 public:
  Comic(const std::string &title, const std::string &author, int length, const std::string &
  bool isHeavy() const override;
  std::string getHero() const;
};
```

Now the following code has different behaviour due to `virtual` and `override` in the above classes

```
Comic c{..., ..., 40, ...};
Comic *cp{&c};
Book *bp{&c};
Book &br{c};
cp->isHeavy(); // calls Comic::isHeavy()
bp->isHeavy(); // calls Comic::isHeavy()
br.isHeavy(); // calls Comic::isHeavy()
```

- A virtual method is dispatched dynamically
  - The decision on which method to call is based on the runtime type of the object
  - dynamic dispatch has a cost since it is determined at runtime
  - CS744

```
// The objects will be sliced, but the correct isHeavy will be called
Book *collection[20];
for (int i = 0; i < 20; ++i) cout << collection[i]->isHeavy();
```

Polymorphism

> The ability to accommodate multiple types within the same abstraction

- collection is a polymorphic array

```
// Must use references to avoid slicing to occur on the params
// ostream is polymorphic
ostream &operator<<(ostream &, Student &);
```

Destructors

- When an object of a derived class is destroyed:

1. subclass destructor body runs
2. subclass fields are destroyed(reverse declaration order, opposite to construction)
3. superclass destructor is called
4. memory is deallocated

```
// Run with valgrind
// leaks memory

class X {
  int *x;
 public:
  X(int n): x{new int [n]} {}
  ~X() { delete [] x; }
};

class Y: public X {
  int *y;
 public:
  Y(int n, int m): X{n}, y{new int [m]} {}
  ~Y() { delete [] y; }
};

int main () {
  X x{5};
  Y y{5, 10};

  X *xp = new Y{5, 10};

  // calls class X destructor but not class Y destructor
  // destructor is statically dispatched
```

```
  delete xp;
}

// Run with valgrind
// no memory leak

class X {
  int *x;
 public:
  X(int n): x{new int [n]} {}
  virtual ~X() { delete [] x; }
};

class Y: public X {
  int *y;
 public:
  Y(int n, int m): X{n}, y{new int [m]} {}
  ~Y() { delete [] y; } // doens't need virtual keyword since it was declared in class X
};

int main () {
  X x{5};
  Y y{5, 10};

  X *xp = new Y{5, 10};

  // calls class Y destructor since X destructor is virtal
  // destructor is dynamically dispatched
  delete xp;
}
```

- If you don't want your class to be subclassed, declare the class `final`
    - This is default in `kotlin` but not `java`

```
class Y final : public X {
};
```

- Note that final can be a field name, it is one of the keywords that only counts as a keyword if it is in a specific location, this is to maintain backwards compatibility

```
class Student {
  public:
    // fees has no default impl
    virtual int fees();
};

class Coop : public Student {
```

```
  public:
    int fees() override {
    }
};

class Regular : public Student {
  public:
    int fees() override {
    }
};
```

- We want Student::fees to not have an implementation
  - Make it Pure Virtual

```
class Student {
  public:
    // Pure Virtual method
    virtual int fees() = 0;
};
```

- `virtual` methods MAY be overridden by subclasses
- `Pure Virtual` methods MUST be overridden by subclasses to be considered concrete
- Student is not a **concrete class** since it has a Pure Virtual method
  - `abstract` class
- A class with even a single Pure Virtual method is abstract and cannot be instantiated

```
Student s: // will not compile
```

- Abstract classes are used to organize types
  1. shared fields/methods
  2. polymorphism
- A concrete class declares no new Pure Virtual methods and overrides **all** inherited Pure Virtual methods

UML

- Both `virtual` and `Pure Virtual` use italics
- `Abstract` class name use italics
- `static` uses bold

Templates

```
// A non generic Stack class
class Stack {
  int count;
  int capacity;
  int *contents;

  public:
    int pop();
    void push(int);
    int top();
    ~Stack();
};
```

- To make this Stack class generic, we can use C++ templates
  - Template class is parameterized on a type
    - The type would be `int` in the above class

```
// A generic Stack class
template <typename T>
class Stack {
  int count;
  int capacity;
  T *contents;

  public:
    T pop();
    void push(T);
    T top();
    ~Stack();
};

Stack<int> sInts;
Stack<string> sStrings;
```

Templates, STL, Exceptions

Last time

```
template <typename T>
class Stack {
  int size;
  int capacity;
```

```
  T *contents;
  public:
    void push(T x) {}
    T top() {}
    void pop() {}
    ~Stack() {}
};
```

```
Stack<int> sInts;
Stack<string> sStrings;
```

Template List Class

```
template <typename T>
class List {
  struct Node {
    T data;
    Node *next;
  };
  Node *thelist = nullptr;

  public:
    class Iterator {
      Node *cur;
      Iterator(...) {...}
      public:
        T &operator*() {...}
        Iterator &operator++() {...}
        friend class List<T>;
    }
    T ith(int i) {...}
    void addToFront(T &data) {...}
};

List<int> loi;
List<string> los;

for (List<int>::Iterator it = loi.begin(); it != loi.end(); ++it)
  cout << *it << endl;
}

List<List<int>> loloi;
```

STL: Standard Template Library

- std::vector

  - Java equivalent would be `ArrayList`
  - Dynamically resizing array of generic contents, but it avoids costly copying
  - Can shrink accordingly as well, `std::vector::resize()`

```
include <vector>
vector<int> v{1, 2}; // contains vector that has an array with elements [1, 2]
v.emplace_back(3); // add to end, [1, 2, 3], uses move ctor, prefer this
v.push_back(4); // add to end, [1, 2, 3, 4], uses copy ctor
vector<int> v2(4, 5); // contains vector that has an array with elements [5, 5, 5, 5]

for (int i = 0; i < v.size(); ++i) {
  cout << v[i] << endl;
}

for (vector<int>::iterator it = v.begin(); it != v.end(); ++it) {
  cout << *it << endl;
}

for (auto n : v) {
  cout << n << endl;
}

for (vector<int>::reverse_iterator rit = v.rbegin(); rit != v.rend(); ++rit) {
  cout << *rit << endl;
}

v.pop_back(); // remove last element, now we have essentially a stack
```

- Many STL methods take iterators as parameters
  - `v.erase(v.begin());` to remove first, like a queue
    - time: O(n)
    - now old iterator is wrong since elements are shift
    - returns an iterator pointing to the element after the element that was reversed
  - `v.erase(v.end() - 1);` to remove last
- `v[i]` gets the ith element, unchecked access
  - `i` can be out of range, no index bounds checking
- `v.at(i)` gets the ith element, checked access
  - `i` cannot be out of range, an exception will be thrown

Exceptions

- `v.at(i);`
  - out of bounds gets discovered in the `vector::at(int)` method, that needs to get bubbled up the call stack

- if `i` is out of bounds, a error will be thrown, you must catch the exception at the call site or the program will terminate

```
// Program will terminate prematurely due to the exception
int main () {
  vector<int> v;
  v.emplace_back(2);
  v.emplace_back(4);
  v.emplace_back(6);

  cout << v.at(3) << endl;  // out of range
}
```

```
#include <stdexcept>

// error is caught, program will print err msg
int main () {
  vector<int> v;
  v.emplace_back(2);
  v.emplace_back(4);
  v.emplace_back(6);

  try {
    cout << v.at(3) << endl;  // out of range
    cout << "Done with try" << endl;
  } catch (out_of_range r) { // out_of_range is a class that inherits from exception in stde
    cerr << "Bad range " << r.what() << endl;
  }

  cout << "Done" << endl;
}
```

```
// call chain and throwing an exception
// stack unwinding
void f () {
  cout << "Start f" << endl;
  throw (out_of_range("f")); // exception is thrown
  cout << "Finish f" << endl;
}

void g() {
  cout << "Start g" << endl;    f ();    cout << "Finish g" << endl;
}

void h() {
  cout << "Start h" << endl;    g ();    cout << "Finish h" << endl;
}
```

```
int main () {
  cout << "Start main" << endl;
  try {
    h();
  }
  catch (out_of_range) { cerr << "Range error" << endl; }
  cout << "Finish main" << endl; // this will be triggered
}
```

- This is called stack unwinding
- When an exception is thrown, the program begins to look for a catch block. This might cause stack frames to be popped.
  - If a catch block is found, code resumes there, otherwise program gets killed

```
// Can have multiple catch blocks
try {
} catch (out_of_range e) {
} catch (bad_alloc e) {
}
```

Error recovery can be done in stages

```
try {
} catch (some_exception e) { // catch by value, slicing can occur to due to polymorphism
  throw some_unrelated_exception{};
  throw e; // throws whatever was caught in e
  throw; // throws the original exception, no slicing
}
```

- All c++ library exceptions inherit from `exception` class
  - but in c++ you can throw anything (`int`, `string`, `exception`, etc.)
- To catch **all** exceptions, use `...`

```
try {
} catch (...) { // not very descriptive however, very hacky, don't do this please
}
```

Exceptions, Design Patterns

Last time

- In C++, you can throw anything
- **Good practice**: using existing exception classes or create your own

```
class BadInput {};
int n;
try {
  if (!(cin >> n)) throw BadInput{};
} catch (BadInput &e) {
  n = 0;
}
```

Destructors & exceptions

- By default a program will terminate if a destructor throws an exception (std::terminal is called)
- we can change this default behaviour
  - `~class noexcept(false) {}`
- Supposed there is an exception and the stack is unwinding
  - a destructor is running
    - what if an exception occurs
  - this produces 2 simultaneous exceptions
    - `std::terminate` is called
- **Advice**: destructors shouldn't throw exceptions

Design Patterns

- **Principle**: program to an interface, not an implementation
- Create abstract base classes to provide an interface
- Use base class ptrs and call interface methods
  - subclasses can be swapped in or out to change behaviour

Iterator Pattern Revisited

- operator*, operator++, operator!=

```
class AbsIter {
  public:
    virtual ~AbsIter() {} // You must have an implementation since this gets called when a s
    virtual int &operator*() const = 0;
    virtual AbsIter &operator++() = 0;
    virtual bool operator!=(const AbsIter &) = 0;
};
```

```
class List {
  struct Node;
  Node *head = nullptr;
  public:
```

```
    class Iterator : public AbsIter {
      Node *cur;
      Iterator() {
      }
      public:
        int &operator*() const override {
        }

        AbsIter &operator++() override {
        }

        bool operator!=(const AbsIter &other) override {
        }
    };
};

class Set {
  public:
    class Iterator : public AbsIter {
      // same methods get implemented
      // We can write code that is no longer tied to any specific data structure
    };
};

template <typename Fn>
void forEach(AbsIter &start, const AbsIter &end, Fn f) {
  while (start != end) {
    f(*start);
    ++start;
  }
}

void addFive(int &x) {
  x += 5;
}

List<int> l;
List::Iterator b = l.begin();
forEach(b, l.end(), addFive);
```

Observer Pattern

- used in a publish-subscribe system
  - **Subject** - publishing/generating data
  - **Observer** - subscribe to data

AbsSubject
+ attach(AbsObserver) + detach(AbsObserver) + notifyObservers() + ~*AbsSubject*

[] —↓

AbsObserver
+ *notify()*

- We would have a Subject inherit from AbsSubject, and an Observer (which inherits from AbsObserver) **HAS-A** Subject
- Want AbsSubject to be abstract
  - no obvious Pure Virtual method
  - make the destructor Pure Virtual, but still implement it
- **Pure Virtual Method**: A Pure Virtual method must be implemented by the subclass for it to be concrete

Decorator Pattern

- trying to update/add functionality to existing object

Component
+ *operation()*

ConcreteComponent inherits from Component

ConcreteComponent
+ operation()

Decorator inherits from Component, but it is abstract Decorator **HAS-A** component

Decorator

ConcreteDecorator1 inherits from Decorator

ConcreteDecorator1
+operation()

ConcreteDecorator2 inherits from Decorator

ConcreteDecorator2
+operation()

Big 5 revisited

```cpp
class Book {
  // imagine big 5 implemented
};

class Text : public Book {
  // big 5 not implemented
};

Text t1 = {...};
Text t2 = t1; // calls Text's copy ctor, the one we get for free

// this is what the default would do
Text::Text(const Text &other)
  : Book(other), topic{other.topic} {
}

// this is what the default would do
Text &Text::operator=(const Text &other) {
  Book::operator=(other);
  topic = other.topic;
  return *this;
}
```

Move constructor

Incorrect, this makes a deep copy

```cpp
Text::Text(Text &&other) : Book{other}, topic{other.topic} {
}
```

Correct

```
Text::Text(Text &&other) : Book{std::move(other)}, topic{std::move(other.topic)} {
}
```

- `std::move(T t)` allows you to get `t` (an **lvalue**) as an **rvalue**

Move assignment operator

```
Text &Text::operator=(Text &&other) {
  Book::operator=(std::move(other));
  topic = std::move(other.topic);
  return *this;
}
```

Copy assignment operator

```
Text t1{"abc", "Nomair", 400, "CS246"};
Text t2{"xyz", "Dave", 200, "CS136"};
Book *pb1 = &t1;
Book *pb2 = &t2;
*pb1 = *pb2; // object assignment through Base class ptrs
// Book::operator= is called
// t1 will become {"xyz", "Dave", 200, "CS246"}
```

- Partial Assignment Problem
  - `topic` was not copied
- We could make operator= virtual

```
class Book {
  public:
    virtual Book &operator=(const Book &);
};

class Text : public Book {
  public:
    Text &operator=(const Text &) override; // does not compile, incorrect method signature

    Text &operator=(const Book &) override; // does compile, signatures match
    // We don't know that the parameter is of type Text, call Mixed Assignment Problem
};
```

- **Side Note**: method signature is name and parameters, not the return type
- **Mixed Assignment Problem**: This allows assignment to Texts from any type of Book

    - later how to avoid this will be discussed, for now, lets not make operator= virtual

- T prevent Partial assignment, lets prevent assignment through base class pointers
    - Could make Book::operator= private, but subclasses still need access
    - Could make Book::operator= protected, but we still can't assign a Book to another Book
    - The base class should have been abstract (recall Book was not abstract), this solves the above problem

AbstractBook
title author numPages
-
+ *AbstractBook* # operator=

- Each class inherits from AbstractBook
- Now you can assign with the exception of going through base class pointers as above

Factory Method Pattern

Enemy and Level are abstract classes

- Enemy
    - Turtle
    - Bullet
- Level
    - Normal
    - Castle

```
Player *p = ...;
Level *l = ...;
Enemy *e = nullptr;
while (p->isNotDead()) {
  // generate enemy
  e = l->createEnemy(); // see below code snippet
  // attack player
}
```

- The type of enemy to generate next should come from a factory method which is part of the level

```
class Level {
  public:
    virtual Enemy *createEnemy() = 0; // Pure Virtual
};

class Normal : public Level {
  public:
    Enemy *createEnemy() override {
      // send in more turtles
    };
};

class Castle : public Level {
  public:
    Enemy *createEnemy() override {
      // send in more bullets
    };
};
```

- **Note**: Factory Pattern is also called the Virtual Constructor Pattern
  - The factory method acts like a constructor
  - **Eg.**
    - Iterator::begin and Iterator::end
    - LinkedList::addToFront

Template Method Pattern

- Used when a class wants subclass to override a proper subset of its methods

Base
+
+

- A subclass can only override a proper subset of the base class' method

```
class Turtle {
  public:
    void draw() {
      drawHead();
      drawShell();
      drawFeet();
    }
  private:
    void drawHead();
```

```cpp
    void drawFeet();
    virtual void drawShell() = 0;
};

class GreenTurtle : public Turtle {
  void drawShell() override {
  }
};
```

TODO

Visitor design pattern

- TODO: get these notes from elsewhere

NVI Idiom (Non-virtual interface)

1. all public methods are non virtual
2. all virtual methods should be private/protected

- **Exception**: the destructor which is public/protected

```cpp
// not using NVI
class DigitalMedia {
  public:
    virtual void play() = 0;
    virtual ~DigitalMedia();
};

// using NVI
class DigitalMedia {
  public:
    virtual void play() {
      // we could update analytics here or whatever we want
      doPlay();
    }
    virtual ~DigitalMedia();

  private:
    virtual void doPlay() = 0;
};
```

std::map

- generalization of an array
- template class parametrized on two types: the key and the value

```
#include <map>

map<string, int> m;
m["abc"] = 1;
m["def"] = 2;
std::cout << m["abc"] << std::endl;
m.erase("abc");
std::cout << m["xyz"] << endl; // key not in map, it gets inserted with a default value (0)

// to search for a key in a map
if (m.count("pqr")) { // 0 if not found, 1 if found
}

for (auto &p : m) {
  // p is of type std::pair
  std::cout << p.first << p.second << std::endl;
}
```

- map is a binary tree, the key needs to support the `<` operator

Visitor Design Pattern

- used to do **double dispatch**
- **virtual**: choose the method to execute based on runtime type of the object
- what if the choice of the method to run depends on 2 objects
- Assume we have an `Enemy` class with subclasses `Turtle` and `Bullet`
- Assume we have an `Weapon` class with subclasses `Stick` and `Rock`

```
virtual void Enemy::strike(Stick &) = 0;
virtual void Enemy::strike(Rock &) = 0;

while (player->notDead()) {
  e = l->createEnemy();
  w = player->chooseWeapon();
  e->strike(*w);
}
```

- VDP uses a combination of overriding and overloading

```cpp
class Enemy {
  public:
    virtual void strike(Weapon &) = 0;
};

class Turtle : public Enemy {
  public:
    void strike(Weapon &w) override {
      w.useOn(*this);
    }
};

class Bullet : public Enemy {
  public:
    void strike(Weapon &w) override {
      w.useOn(*this);
    }
};

class Weapon {
  public:
    virtual void useOn(Turtle &) = 0;
    virtual void useOn(Bullet &) = 0;
};

class Stick : public Weapon {
  public:
    void useOn(Turtle &) override {
    }

    void useOn(Bullet &) override {
    }
};

class Rock : public Weapon {
  public:
    void useOn(Turtle &) override {
    }

    void useOn(Bullet &) override {
    }
};
```

Another VDP

1. separation of concerns
2. adding functionality without cluttering classes with new virtual methods

- requires class hierarchy to be setup to accept visitors

```
class Book {
  public:
    virtual void accept(BookVisitor &v) {
      v.visit(*this);
    }
};

class Text : public Book {
  public:
    void accpet(BookVisitor &v) override {
      v.visit(*this);
    }
};

class Comic : public Book {
  public:
    void accept(BookVisitor &v) override {
      v.visit(*this);
    }
};

class BookVisitor {
  public:
    virtual void visit(Book &) = 0;
    virtual void visit(Comic &) = 0;
    virtual void visit(Text &) = 0;
};
```

Catalog Books

- count Books based on author
- count Text based on topic
- count Comic based on hero

```
class Catalog : public BookVisitor {
  std::map<string, int> catalog;

  public:
    void visit(Book &b) {
```

```
    ++cat[b.getAuthor()];
  }
  void visit(Text &b) {
    ++cat[b.getTopic()];
  }
  void visit(Comic &b) {
    ++cat[b.getHero()];
  }
};
```

Visitor Design Pattern and include vs forward declaration

- Files: `se/visitor`
  - Has a cyclic include and won't compile
  - An include creates a compilation dependency
  - Often, a forward declaration of a class is all we need (rather than the include)
    - `class XYZ; // you are telling the compiler to trust that class XYZ is getting compiled`
  - **Whenever possible**, prefer forward declaration to include:

1. reduces compilation dependencies and reduce circular dependencies
2. reduce compile time
3. reduces frequency of compilation for a file

File: a.h

```
class A {
};
```

File: b.h

```
#include "a.h"

class B : public A {
};
```

File: c.h

```
#include "a.h"

class C {
  A a;
};
```

File: d.h

```
class A;

class D {
  A *ptrA;
};
```

File: d.cc

```
#include "a.h"

void D::foo() {
  ptrA->someMethod();
}
```

File: e.h

```
class A;

class E {
  A f(A x);
};
```

File: e.cc

```
A E::f(A x) {
}
```

Reducing Compilation Dependencies

File: window.h

```
#include <xlib/x11.h>

class XWindow {
  Display *d;
  Window w;

  public:
    void draw();
};
```

File: client.cc

```
#include "window.h"

myXWindow->draw() {
}
```

- `client.cc` has to recompile even if private members in window.h change

Solution: Ptr to Implementation (pImpl idiom)

- Take the private implementation out of window.h

  File: XWindowImpl.h

```
struct XWindowImpl {
  Display *d;
  WIndow w;
};
```

File: window.h

```
struct XWindowImpl;

class XWindow {
  XWindowImpl *pImpl;
  public:
    void draw();
};
```

File: window.cc

```
#include "xwindowimpl.h"
#include "window.h"

XWindow::XWindow() :
  pImpl{new XWindowImpl()} {
  pImpl->d = ...;
}
```

- If private implementation changes, window.h and therefor client.cc are not affected
- In UML:

XWwindow

- ↓

XWindowImpl

Generalization: Bridge Pattern

XWwindow

- ↓

XWindowImpl

- *XWindowImpl* will have bunch of implementations

Terms

- **Coupling**: degree to which modules/classes interact with each other
  - **low coupling**: interaction through a public interface - **preferred**
  - **high coupling**: interaction through a public implementation
- **Cohesion**: how related are things within a module
  - **low cohesion**: loosely connected, such as `<utility>` which has loads of unrelated things
  - **high cohesion**: module/class achieves exactly a single task - **preferred**

Decoupling the interface

```
class ChessBoard {
  void foo() {
    cout << "Your move";
  }
};
```

- `ChessBoard` is now coupled with `stdout`
- Slightly better would be to use an `ostream` variable
- `ChessBoard` class has 2 responsibilities: game state, communication
- Violates single responsibility principle
- It would be better to have a separate communication class

MVC (Model-View-Controller)

- Multiple opinions on implementation

1. `Controller` tells the `Model` something has changed, the `Model` then notifies the `View` which then queries from the `Model`
2. `Controller` tells the `Model` something has changed, the `Model` then updates the `Controller` and the `Controller` notifies the `View`

Exception Safety

```
void f() {
  MyClass *p = new MyClass();
  MyClass c;
  g(); // if g() throws an exception, the next line won't get hit and p will leak
  delete p;
}
```

Exception Safety

- Want our program to recover from exceptions
  - Program to not leak memory
  - Want to stay in a consistent state (no dangling ptrs, no broken invariants)
  - C++ guarentee: during stack unwinding all stack allocated objects are destroyed, destructors run and memory is reclaimed

  Old f method

```
void f() {
  MyClass *p = new MyClass();
  MyClass m;
  g(); // g() might throw an exception
  delete p;
}
```

  Refactor f method

```
void f() {
  MyClass *p = new MyClass();
  try {
  MyClass m;
  } catch (...) {
    delete p;
    throw;
  }
  delete p;
}
```

- But this code is super complex, tedious, and very error prone
- Want a way to guarantee that some code runs irrespective of whether an exception occurs or not

Language	Mechanism
Java	finally

Language	Mechanism
Scheme	dynamic-wind
C++	nothing

- in C++, we need to rely on the guarantee for stack objects
  - Maximize stack usage

C++ idiom: RAII: Resource Acquisition Is Initialization

- Every resource should be wrapped within a stack allocated object whose destructor released the resource

```
int main() {
  ifstream f{"file.txt"}; // automatically opens the file
}
```

- The file resource is released (closed) when the stack object goes out of scope

RAII for heap memory

- Wrap the heap object within a stack object whose destructor deallocates the heap object
- STL provides a template class to do this
  - `std::unique_ptr<T>`
    - Constructor takes a T* as an argument
    - destructor deletes the ptr
    - overloads operator*, operator->

```
void f() {
  std::unique_ptr<MyClass> p{new MyClass};
  // auto p = std::make_unique<MyClass>();
  MyClass m;
  g();
  // Don't need to delete p since it is stack allocated and will clean the heap allocated My
}
```

- TODO: `c++/unique_ptr/example1.cc`

Copying vs Moving `unique_ptr`

```
class c{...};
auto p = make_unique<c>();
```

```
std::unique_ptr<c>;
unique_ptr<c> q = p; // won't compile
```

- `unique_ptr` does not have a copy constructor
- `unique_ptr` cannot be copied as then you would have multiple stack objects pointing tot he same heap object
    - double `delete` will get called
- Sample implementation of `unique_ptr`

  TODO: `c++/unique_ptr/basicimpl.cc`

- Note that `void f() = delete;` disables method `f()`

Ownership

- Who owns the heap resource
- If code does not own the heap resource, it can be provided the "raw" pointer
    - Use `get()` to access the raw ptr
- If there are multiple owners, use `std::shared_ptr`

```
void g() {
  std::shared_ptr<MyClass> p1 = make_shared<MyClass>();
  if () {
    auto p2 = p1; // copy constructor
  } // p2 goes out of scope (won't deleted MyClass object as p1 still points to it)
} // p1 out of scope, destructor will delete MyClass ptrs
```

- `shared_ptr` uses reference counting
    - destructor only deletes heap memory when ref count reaches 0

3 Levels of Exception Safety

1. Basic Guarantee: if an exception occurs the program is in a valid but unspecified state (no memory leaks, no dangling ptrs, etc.)
2. Strong Guarantee: if an exception occurs during f, it is as if f was never called
3. No-throw Guarentee: f does not throw exceptions, achieves its task

Exception Safety

```
class A {};
class B {};
class C {
  A a;
```

```
  B b;

  void f() {
    a.method1(); // strong guarantee
    b.method2(); // no strong guarantee
  }
};
```

- if method1 throws, f has a strong guarantee
- if method2 throw, we need to undo what method1 did
  - often undos are not possible
  - `f()` does not have a strong guarantee
- Suppose method1/method2 only have local side effects can we rewrite f to provide a strong guarantee?
- Lets call these methods on copies of a and b

```
void C::f() {
  A atemp{a};
  B btemp{b};
  atemp.method1();
  btemp.method2();
  a = atemp;
  b = btemp; // this can throw an exception and thus f() does not have a strong guarantee
}
```

- `f()` would have a strong guarantee if B::operator= is no throw

```
struct CImpl { A a; B b; };
class C {
  unique_ptr<CImpl> pImpl;

  /**
   * f() now has a strong guarantee
   */
  void f() {
    unique_ptr<CImpl> temp{new CImpl(*pImpl)};
    // auto temp = make_unique<CImpl>(*pImpl);

    temp->a.method1();
    temp->b.method2();

    swap(pImpl, temp);
  }
};
```

Exception Safety in the STL

- `std::vector` - uses RAII
- wraps around a heap array
- destructor deallocates array

```
void f() {
  std::vector<MyClass> v;
} // When v is destroyed, the elements (Objects) are also destroyed

void g() {
  vector<MyClass *> v;
} // When v is destroyed, the ptrs are not deleted

// g() is not exception safe
void g() {
  vector<MyClass *> v;
  // We have to explicitly delete the contents of v
  for (auto p : v) delete p; // But now we are calling delete
}

void g() {
  vector<unique_ptr<MyClass>> v;
}
```

- `std::vector::emplace_back` provides a strong guarantee
- **Easy case**: size < capacity
- **Hard case**: size == capacity
  - This is exception safe:
  - Create larger array
  - copy from old to new
  - swap old and new and delete old
  - delete is no throw
  - **This is inefficient though since copying not moving**
- We should move elements to new array
  - if an exception occurs during the move, we lose our strong guarantee
  - `emplace_back` checks if the move has a no throw guarantee (`noexcept`) and uses it, otherwise copies are used

Casting

C-style casting

```
Node n;
int *p = (int *) &n;
```

- Four different kinds of C++ casting:

1. `static_cast` - sensible cast since behaviour is well-defined (compile checks that it is possible)
2. `reinterpret_cast` - anything goes, relies on compiler dependant decisions on how objects appear in memory
3. `const_cast` - Used to remove `const`
4. `dynamic_cast` - see next notes

`static_cast`

```
void f(int);
void f(double);

double a = 12.2;
f(a); // calls f(double)
f(static_cast<int>(a)); // calls f(int)

Book *bp = new Text{...};
// To call getTopic, we need a Text *
Text *tp = static_cast<Text *>(bp);
// There must be an IS-A relationship between the current type and the requested type
// This is a downcast, unchecked cast
// If your assumption is not correct, behaviour is undefined
```

`reinterpret_cast`

```
Student *s = ...;
Turtle *t = reinterpret_cast<Student *>(s);
t->draw();
// Casting an Object rather than a ptr would cause slicing
```

`const_cast`

```
void g(int *p);
const int *q = ...;
g(q); // won't compile due to const
g(const_cast<int *>(q)); // will compile
// But if q is in read-only memory, then if g() modifies q, then the program will crash
```

Casting

- 4 kinds of casts (see last time)

`dynamic_cast`

```
vector<Book *> myBooks;
Book *bp = myBooks[0];
Comic *cp = dynamic_cast<Comic *>(bp); // If this fails, then nullptr is returned
```

```
if (cp) cout << cp->getHero();
else cout << "Not a Comic";
```

- Tentatively try the cast
  - if successful, cp is a valid Comic ptr
  - if not successful, cp becomes nullptr

Casting with share ptrs

- used to cast a `shared_ptr` to another `shared_ptr` object

1. `static_pointer_cast`
2. `const_pointer_cast`
3. `dynamic_pointer_cast`

Runtime-Type Information (RTTI)

```
void whatIsIt(shared_ptr<Book> b) {
  if (dynamic_pointer_cast<Comic>(b)) cout << "Comic";
  else if (dynamic_pointer_cast<Text>(b)) cout << "Text";
  else cout << "Book";
}
```

- The code is tightly coupled with the class hierarchy, the above code would therefore be bad practice. Prefer `virtual` methods.

  `dynamic_cast` for references

```
Comic c{};
Book &b{c};
Comic &c2 = dynamic_cast<Comic &>(b);
// if successful, c2 is a valid reference to the Comic
// if not successful, a bad_cast exception is thrown
```

  Remember from TODO: figured out which notes this example belongs to

```
Text t1{}, t2{};
Book *bp1{&t1};
Book *bp2{&t2};
*bp1 = *bp2;
```

- Resolve partial assignment by making operator= virtual

```
class Book {
  public:
    virtual Book &operator=(const Book &other);
};
```

```
class Text : public Book  {
  public:
    Text &opeator=(const Book &other) override {
      const Text &temp = dynamic_cast<const Text &>(other);
      Book::operator=(other);
      topic = temp.topic;
      return *this;
    }
};

Comic c{};
Book *bp3{&c};
*bp1 = *bp3; // This could throw an exception
```

Template Functions

```
template<typename T>
T main(T x, T y) {
  return x < y ? x : y;
}

int x = 5, y = 7;
int result = min(x, y); // min<int>(x, y);
// type of T is automatically inferred
```

```
template<typename Func>
void foreach(AbsIter &start, AbsIter &finish, Func f) {
  while (start != finish) {
    f(*start);
    ++start;
  }
}
```

- Instead of hard coding the type `AbsIter`, we can make it a template parameter as well

```
template<typename Iter, typename Func>
void foreach(Iter start, Iter finish, Func f) {
  while (start != finish) {
    f(*start);
    ++start;
  }
}
```

```
void foo(int n) {
  cout << n << endl;
}

int a[] = { 1, 2, 3, 4 };
foreach(a, a + 4, foo);
```

std::algorithm

1. `std::for_each`
2. `std::find`
3. `std::find_if`
4. `std::find_if_not`
5. `std::copy`
6. `std::transform`

   `std::find`

```
template<typename Iter, typename T>
Iter find(Iter first, Iter last, const T &val) {
  // Search for val within the range [first, last)
  // If val is not found, then return last
}
```

   `std::copy`

```
template<typename InIter, typename OutIter>
OutIter copy(InIter first, InIter last, OutIter result) {
  // copy one container's range [first, last) into another starting at result
  // Require: result has enough space, no memory allocation will be performed
}

vector<int> v{ 1, 2, 3, 4, 5, 6, 7 };
vector<int> w(4); // Reserve space for 4 ints
copy(v.begin() + 5, v.begin() + 5, w.begin());
```

   `std::transform`

```
template<typename InIter, typename OutIter, typename Func>
OutIter transform(InIter first, InIter last, OutIter result, Func f) {
  while (first != last) {
    *result = f(*first);
    ++first;
    ++result;
  }
  return result;
}
```

```
int add1(int n) {
  return n + 1;
}

vector<int> v{ 1, 2, 3, 4, 5, 6, 7 };
vector<int> v2(v.size());
transform(v.begin(), v.end(), v2.begin(), add1);
// Now v2 = { 2, 3, 4, 5, 6, 7, 8 };
```

How Virtual works

CS444 will implement this

```
class C {
  int x;
  public:
    virtual void foo();
    void bar();
    ~C();
};

C c;
sizeof(c); // atleast 12 due to ptr to foo()
```

- every time a class has a virtual method, objects of that class contain a ptr "virtual pointer" (vptr)
  - Needed to implement dynamic dispatch
- `c.bar();`
  - Compiler finds the memory address for bar function at compile time
- `c.foo();`
  - For every class that has a virtual method, a single virtual table is created (vtable)
  - vptrs point to virtual tables

virtual table, contains address of different virtual methods

"C"
addresses of foo addresses of destructor

- Object C has a vptr to the table above (inside the UML for Object C)
- `p->foo();`
  - Follow object to the Object, then follow the vptr to the vtable, then find the function address

- much more complex than a non virtual method
- different children of Object C would have different vptrs
  * If the virtual method is no overridden in a child, then the address to `foo()` would be the same as the address in the parent