CS 246 Course Notes

Object-Oriented Software Development

Brad Lushman • Fall 2019 • University of Waterloo

Table of Contents

1 September 5, 2019

Grading: Assignments: 40% (5%, 7%, 7%, 9%, 12%), Midterm: 20%, Final: 40%

Modules: Linux Shell, C++, Tools, Software Engineering

1.1 Linux Shell

Shell:

- interface to the OS - gets OS to run programs, manage files, etc.
- graphical (click/touch)
- command line - type commands at a prompt, more versatile

This course uses bash, a command line shell.

To check this, login and type `echo $0`.

`cat` – displays the contents of a file (e.g. `cat /usr/share/dict/words`)

- first `/` is called the root (top) directory
- `usr`, `share`, `dict` are directories
- `words` is the file

In Linux, a directory is a special kind of file.

In Linux, every line of a valid text file must end with a newline character, **including the last one**. Marmoset checks for this.

Use `^C` (Ctrl + C) to stop.

`ls` – list files in current directory (non-hidden files)

`ls -a` – list all files (including hidden ones - name begins with `.`)

`pwd` – prints the current directory

`wc` – shows the number of lines, words, and characters in a file

1.2 Output and Input Redirection

Typing `cat` on its own makes it wait and prints out everything you type onto the screen.

This can be useful if we can capture the output in a file.

To do this, we can use `cat > myfile.txt`.

To stop, use `^D` at the beginning of a line. This sends an end-of-file (EOF) signal to `cat`.

In general, using `command args > file` executes `command args` and captures the output in `file` instead of the screen. This is called **output redirection**.

We can also redirect input. Consider `cat < file`.

- input comes from `file` instead of the keyboard
- displays content of `file`
- seems equivalent to `cat file`

The difference between the two (important):

`cat file`

- passes the name `file` as an **argument** to `cat`
- `cat` opens the file and displays its contents

`cat < file`

- **the shell** opens `file` and passes its contents to `cat` in place of keyboard input

Both input and output redirection can be done: `cat < infile > outfile`.

- takes characters from `infile` and sends them to `outfile`
- copies the file

1.3 Globbing Patterns

`*.txt` is called a globbing pattern. `*` means to match any sequence of characters. The shell finds all files in the current directory whose names match the pattern and substitutes them on the command line.

For example, `cat *.txt` transforms into `cat a.txt b.txt c.txt`. `cat` opens all three and displays them.

More globbing patterns can be found on the Linux sheet.

1.4 Streams

Every process is attached to 3 **streams**.

`stdin` is the input, and `stdout` and `stderr` are the outputs.

By default, `stdin` is the keyboard, and `stdout` and `stderr` are on the screen.

Redirection: `<` connects `stdin` to a file, `>` connects `stdout` to a file, and `2>` connects `stderr` to a file.

`stderr` is a separate output stream for error messages. We can send output and error messages to different places, and error messages don't corrupt the format of output files.

Also, `stdout` may be **buffered**. The system may wait to accumulate output before actually displaying (flushing) it. `stderr` is never buffered – error messages are received immediately.

1.5 Pipes

`|` – use output of one program as input to another, and sets second program's `stdin` to first programs `stdout`.

Example. How many words occur in the first 20 lines of `myfile.txt`?

`head -n file` gives the first `n` lines of `file`.

`wc -w` gives the number of words.

Hence we can pipe the former into the latter with `head -20 myfile.txt | wc -w`.

We can also use `cat myfile.txt | head -20 | wc -w`.

Example. Suppose `words1.txt`, `words2.txt`, etc. contains lists of words, one per line. Produce a duplicate-free list of all words that occur in any of these files.

`uniq` – removes *adjacent* duplicate entries

`sort` – sorts the lines of a file

Solution: `cat words*.txt | sort | uniq`

Example. Print the 10 most frequent words in a file and their frequency.

Solution: `sort myfile | uniq -c | sort -n | tail -10`, if each word is on a different line

`tr ' ' '\n' myfile | sort | uniq -c | sort -n | tail -10` otherwise

2 September 10, 2019

2.1 Embedded Commands

Consider `echo "Today is $(date) and I am $(whoami)"`.

The shell executes `date` and `whoami`, and substitutes the result into the second line.

Note that using `echo Today is $(date) and I am $(whoami)`, with no quotes, takes everything as separate arguments.

Inside single quotes, such as `echo 'Today is $(date) and I am $(whoami)'`, results in no `$`-expansion.

Using `echo *` returns all the files.

Both single and double quotes will suppress glob expansion. For example, `echo "*"` returns `*`.

2.2 Pattern-Matching in Text Files

`grep/egrep` – (extended) global regular expression print

`egrep pattern file` prints every line in `file` that contains `pattern`.

Example. Print every line in `myfile` that contains `cs246`.

Solution: `egrep cs246 myfile`

Example. How many lines in `myfile` contain `cs246` or `CS246`?

Solutions: `egrep "cs246|CS246" myfile | wc -l` or `egrep "(cs|CS)246" myfile | wc -l`

`(c|C)(s|S)246` or `[cC][sS]246` will also match `cS246` and `Cs246`.

`[...]` means any one character between `[` and `]`.

`[^...]` means any one character *except* `...`.

The available patterns are called **regular expressions** (different from globbing patterns).

We can also add an optional space with `[cC][sS] ?246`.

`?` matches 0 or 1 of the preceding pattern (in this case, 0 or 1 spaces).

`*` matches 0 or more of the preceding pattern. For example, `(cs)*246` will match `246`, `cs246`, `cscs246`, etc.

`+` means 1 or more of the preceding pattern.

`.` matches any single character.

`.*` matches anything. `egrep "cs.*246" myfile` would match any lines containing `cs...(anything)...246`.

`^.$` means the beginning or end of line. `^cs246` matches lines that start with `cs246`, and `^cs246$` matches lines exactly equal to `cs246`.

Example. Print all lines of even length.

Solution: `^(..)*$`

Example. Print all files in the current directory whose names contain exactly one `a`.

Solution: `ls | egrep "^[^a]*a[^a]*$"`

Example. Find all words in the global dictionary that start with `e` and have 5 characters.

Solution: `egrep "^e....$" /usr/share/dict/words`

2.3 Permissions

`ls -l` – long form listing

Suppose we have a line `-rw-r----- 1 j2smith j2smith 25 Sep 9 15:27 abc.txt`.

The first character, `-`, is the type. This can be `-`, an ordinary file, or `d`, a directory.
The following three characters, `rw-`, is the (owner) user permissions.
The next three characters after that, `r--` is group permissions.
The last three characters, `---`, are other permissions.
The next number, `1`, is the number of links.
After that, `j2smith`, is the owner.
Then, `j2smith`, is the groups.
The number after that, `25`, is the size.
Next, `Sep 9 15:27` is the last modified date.
Finally, `abc.txt` is the name of the file.

2.4 Groups

A user can belong to one or more groups, and a file can be associated with one group.

User bits apply to the file's owner.

Group bits apply to members of the file's group (other than the owner).

Other bits apply to everyone else.

`r` is the read bit, `w` is the write bit, and `x` is the execute bit.

Bit	Meaning for ordinary files	Meaning for directories
`r`	file's contents can be read	directory's contents can be read (e.g. `ls`, globbing, tab completion)
`w`	file's contents can be modified	directory's contents can be modified (can add/remove files)
`x`	file can be executed as a program	directory can be navigated (i.e. can `cd` into the directory)

If the directory's `x` bit is not set, then there is no access at all, nor to any file/directory within it.

`chmod mode file` – change permissions

`mode: usertypes operator perms`

`usertypes:` `u` (user), `g` (group), `o` (other), `a` (all)

`operator:` `+` (add person), `-` (remove person), `=` (set perms exactly)

`perms:` `r`, `w`, `x`

Example. Give others read permission.

Solution: `chmod otr file`

Example. Make everyone's permission `rx`.

Solution: `a=rx`

Example. Give the owner full control.

Solution: `u+rwx` (or `u=rwx`)

3 September 12, 2019

3.1 Shell Scripts

A shell script is a file containing sequences of shell commands, executed as a program.

Example. Print the date, current user, and current directory.

`myscript` contains

```
#!/bin/bash
date
whoami
pwd
```

`#!/bin/bash` is called the "shebang line", and says to execute this file as a bash script.

First, give the file execute permission with `chmod u+x myscript`.

Then run the file with `./myscript`.

To debug a script, use `bash -x myscript`, or add `-x` to the shebang line.

3.2 Variables

We can assign a variable with `x=1` (no spaces).

Use `$` when fetching the value of a variable.

No `$` is used when setting a variable.

For good practice, use `${x}`.

To remove suffixes, `${A%foo}` removes `foo` from the end of `A` (this does not change the variable itself, just the output).

All variables contain **strings**. Here, `$x` is the **string** 1, not the number.

You can test with `[` and `]`. For example, `[ "hi" = "hi" ]`, or `[ "$A" -le "$B" ]`.

Example. Consider `dir=~/cs246`.

`echo ${dir}` returns `/u/userid/cs246`.

`ls ${dir}` returns the contents of the above directory.

There are some "global" variables.

Important: `PATH` is a list of directories. When you type a command, the shell searches these directories, in order, for a program with that name.

`echo "$PATH"` returns the path, while `echo '$PATH'` returns `$PATH`. `$`-expansion is only in double quotes.

There are special variables for scripts: `$1`, `$2`, etc. – command line args.

Example. Check whether a word is in the dictionary.

Solution: `./isItAWord hello` where `isItAWord` contains

```
#!/bin/bash
egrep "^$1$" /usr/share/dict/words
```

This prints nothing if the word is not found, and prints the word if found.

Example. A good password is not in the dictionary. Determine if the password is good.

Solution: First consider

```
#!/bin/bash
egrep "^$1$" /usr/share/dict/words > /dev/null
```

(Useful: `/dev/null` suppresses output.)

Every program returns a status code when finished.

In Linux, `0` is success, `1` is failure. For example, `egrep` returns `0` if found, `1` if not found.

`$?` is the status of the most recently executed command.

Script continued:

```
if [ $? -eq 0 ]; then
  echo Bad password
else
  echo Maybe a good password
fi
```

Alternatively, we can replace the first line with

```
if egrep ... > /dev/null ; then
```

To verify that the correct number of arguments are there, print an error message if wrong.

```
#!/bin/bash
usage() {
  echo "usage: $0 password" 1>&2
}
if [ $# -ne 1 ]; then
  usage
  exit 1
fi
(as before)
```

`$0` is the name of the program/script, and `1>&2` redirects `stdout` to `stderr`. `$#` is the number of arguments.

3.3 Loops

Example. Print all numbers from 1 to `$1`.

Solution:

```
#!/bin/bash
x=1
while [ $x -le $1 ]; do
  echo $x
  x=$((x+1))
done
```

Use `$((...))` for arithmetic.

A simple `for` loop

```
for x in a b c; do
  echo $x
done
```

would return

```
a
b
c
```

3.4 Applications of Shell Scripts

Example. Looping over a list: rename all `.cpp` files to `.cc`.

Solution:

```
for name in *.cpp; do
  mv ${name} ${name%cpp}cc
done
```

`name%cpp` is the value of `name`, without the trailing `cpp`.

Example. How many lines does word `$1` occur in file `$2`?

Solution:

```
for word in $(cat "$2"); do
  if [ $word = "$1" ]; then
    x=$((x+1))
  fi
done
echo $x
```

Example. Payday is the last Friday of the month. When is this month's payday?

Solution: We have two tasks: compute payday, and report the answer.

The command `cal` returns the calendar for the current month. Then, the line

```
cal | awk '{print $6}'
```

returns

```
Fr
6
13
20
27
```

Then

```
cal | awk '{print $6}' | egrep "[0-9]"
```

gives

```
6
13
20
27
```

and taking the tail with

```
cal | awk '{print $6}' | egrep "[0-9]" | tail -1
```

we get

```
27
```

Then our script would be

```
report() {
  if [ $1 -eq 31 ]; then
    echo "This month: the 31st"
  else
    echo "This month: the ${1}th"
  fi
}
report $(cal | awk '{print $6}' | egrep "[0-9]" | tail -1)
```

Generalizing to any month, note that

```
cal October 2019
```

gives that month's calendar. We can replace the last line of the above script with

```
report $(cal $1 $2 | awk '{print $6}' | egrep "[0-9]" | tail -1) $1
```

Note that `$1` and `$2` are blank if not supplied, hence reverting to previous behaviour. Here, the month name, `$1` is the second argument. Below, `$2` is the second argument.

```
report() {
  if [ $2 ];
    echo -n $2
  else
    echo -n "This month"
  fi
  if [ $1 -eq 31 ]; then
    echo ": the 31st"
  else
    echo ": the ${1}th"
  fi
}
```

Example. Write a script called `mean`. It takes `filename` as an argument, computes the mean of the numbers in that file, and prints it to `stdout`.

Solution:

```
#!/bin/bash
if [ $# -ne 1 ], then
  echo "$0 requires exactly one argument"
  exit 1
fi
A="0" # sum of all numbers
B="0" # number of numbers
for n in $(cat $1); do
  A=$(($A + $n))
  B=$(($B + 1))
done
echo $(($A / $B))
```

4 September 17, 2019

4.1 Testing

Testing is an essential part of program development. It begins *before* you start coding (i.e. creating test suites for expected behaviour) and continues while you are coding. It is not the same as debugging – you cannot debug without testing first. Ideally, the programmer is not the tester (there is no psychological barrier).

Human Testing: Humans look for flaws (i.e. code inspections). Humans cannot check everything.

Machine Testing: Runs program on selected inputs and compares with expected outputs. Again, this cannot check everything, and test cases must be chosen carefully.

Black/White/Grey Box Testing: No/full/some knowledge of program input.

Start with black box testing, and then supplement with white box.

Some tips:

- Test various classes of input (e.g. numeric ranges, positive versus negative).
- Test boundaries of valid ranges (edge cases).
- Test multiple simultaneous boundaries (corner cases).
- Use intuition and experience to guess at likely errors.
- Test extreme cases (within reason).

When white box testing, execute all logical paths through the program, and make sure every function is called.

Do not test invalid input, unless a behaviour has been prescribed. If the input is invalid, there is no corresponding correct output, so any such test case would be meaningless.

Regression Testing: Ensures that new changes to program don't break the old test cases (test suites, testing scripts).

4.2 C++

Hello world in C++:

```
#include <iostream>
using namespace std;
int main() {
    cout << "Hello world" << endl;
    return 0;
}
```

Notes:

- `void main() ...` is illegal in C++, unlike C

- `stdio.h`, `printf` are still available in C++ (but banned in this course)
- preferred C++ I/O `<iostream>`

  `std::cout << ___ << ___ << ___ ...` where each `___` is data

  `std::endl` means the end of line
- `using namespace std` lets you omit the `std::` prefix
- `return` statement can be omitted from `main` (`0` assumed)
- `$?` returns the status code

Compiling a C++ program: `g++ -std=c++14 -Wall program.cc -o program`

`-std=c++14` is the current version, `-Wall` means to warn all, `program.cc` is the name of the program, and `-o program` is the name of the output file (if not given, `a.out`).

Alternatively: `g++14 program.cc -o program`

Most C programs are valid C++ – there will be no C review in this course.

4.3 Input/Output

There are 3 I/O streams:

`cout/cerr` – for printing to `stdout/stderr`
`cin` – for reading from `stdin`

I/O operators:

`<<` – "put to" (output)
`>>` – "get from" (input)

To remember this, the operator "points" in the direction of information flow.

Example. Add two numbers.

Solution:

```
#include <iostream>
using namespace std;
int main() {
    int x, y;
    cin >> x >> y;
    cout << x + y << endl;
}
```

We will omit the first two lines from now on (assume that they are always there).

Note: `cin >>` ignores whitespace (space, tab, newline).

If the number is too big, the program will return the largest number possible.

If the input is exhausted before we get 2 integers, then the statement fails.

If the read failed, `cin.fail()` will be true.

If `EOF`, `cin.eof()` and `cin.fail()` are both true, but not until the attempted read fails.

Example. Read all integers from `stdin` and `echo` them, one per line, to `stdout`. Stop on bad input or `EOF`.

Solution:

```
int main() {
    int i;
    while(true) {
        cin >> i;
        if(cin.fail()) break;
        cout << i << endl;
    }
}
```

Note: There is an implicit conversion from `cin` to `bool` (true if success, false if failure). This lets `cin` be used as a condition.

Using this, we can rework the previous solution:

```
int main() {
    int i;
    while(true) {
        cin >> i;
        if(!cin) break;
        cout << i << endl;
    }
}
```

Note: In C, `>>` is the right bitshift operator. `a >> b` shifts `a`'s bits to the right by `b` spots.

For example, consider `21 >> 3`:
$21 = 10101_2$, removing last 3 digits, `21 >> 3` returns $10_2 = 2$

Essentially, it computes $a/2^b$.

This is the same in C++. But when the left hand side is `cin`, `>>` is the "get from" operator.

The operator `>>` inputs `cin` (istream) and some data (varying types), and outputs `cin` back (istream).

From the previous example, another solution is

```
int main() {
    int i;
    while(true) {
        if(!(cin >> i)) break;
        cout << i << endl;
```

```
    }
}
```

We can also write

```
int main() {
    int i;
    while(cin >> i) {
        cout << i << endl;
    }
}
```

Example. Read all integers and `echo` to `stdout` until `EOF`. Skip non-integer input.

Solution:

```
int main() {
    int i;
    while(true) {
        if(!(cin >> i)) {
            if(cin.eof()) break; // EOF - done
            cin.clear(); // cin stops working until cleared
            cin.ignore(); // ignore next character
        } else cout << i << endl;
    }
}
```

5 September 19, 2019

5.1 I/O Manipulators

Consider printing a number in hexadecimal.

```
cout << hex << 95 << endl; // prints 5f
```

`hex` is an I/O manipulator. It says to print all subsequent ints in hexadecimal.

To go back to decimal, use

```
cout << dec;
```

There are other manipulators as well (see C++ reference `<iomanip>`).

5.2 Strings

Recall that in C, a string is an array of characters (`char*` or `char[]`) terminated by `\0`. You must explicitly manage memory, allocating more memory as strings get larger. It is easy to overwrite `\0` and corrupt memory.

In C++, it is comparatively easier to use strings. Use `#include <string>`. It grows as needed and is safer to manipulate.

Example. `string s = "Hello";`

`"Hello"` is a C-style string (char array). `s` is a C++ string created from a C string on initialization.

String Operations

Equality/inequality: `s1 == s2`, `s1 != s2`

Comparison: `s1 <= s2` (lexicographic)

Length: `s.length()`

Get individual characters: `s[0]`, `s[1]`, etc.

Concatenation: `s3 = s1 + s2`, `s3 += s4`

Example. A program to read in and print a string:

```
int main() {
    string s;
    cin >> s;
    cout << s << endl;
}
```

This reads a string, skips leading whitespace, and stops at whitespace (i.e. reads one word).

If we want the whitespace, we can use `getline(cin, s)`. This reads from the current position to next newline into `s`.

5.3 Stream Abstraction

Stream abstraction applies to other sources of data.

Files

Read from a file instead of `stdin`.
`std::ifstream` reads from a file.
`std::ofstream` writes to a file.

File access in C:

```c
#include <stdio.h>
int main() {
    char s[256];
    FILE *f = fopen("file.txt", "r"); // r means open for reading
    while(true) {
        fscanf(f, "%255s", s);
        if(feof(f)) break;
        printf("%s\n", s);
    }
    fclose(f); // give file access back to OS
}
```

File access in C++:

```cpp
#include <iostream>
#include <fstream>
#include <string>
using namespace std;
int main() {
    ifstream f {"file.txt"}; // declaring and initializing an ifstream opens the
        file
    string s;
    while(f >> s) {
        cout << s << endl;
    } // file is closed when the ifstream (f) goes out of scope
}
```

Anything you can do with `cin/cout`, you can do with an `ifstream/ofstream`.

For example, for a string, attach a stream to the string variable and read from or write to the string.

```
string intToString(int n) {
    ostringstream sock;
    sock << n;
    return sock.str();
}
```

Example. Convert a string to a number.

Solution:

```
int main() {
    int n;
    cout << "Hello, " << endl;
    while(true) {
        cout << "enter a number." << endl;
        cin >> s;
        istringstream sock{s};
        if(sock >> n) break;
        cout << "I said, ";
    }
    cout << "You entered " << n << endl;
}
```

Example. Revisited: `echo` all numbers and skip all non-numbers.

Solution:

```
int main() {
    string s;
    while(cin >> s) { // only way to fail is EOF
        istringstream sock{s};
        int n;
        if(sock >> n) cout << n << endl;
    }
}
```

Differences between the two solutions: this one will take the first number and throw the rest away, while the previous one will take all numbers inside. (e.g. `123abc456` gives only `123` for the above, while for the previous it would return `123` and `456`.)

For the above two examples, there is no need to reset because it is a new sock each time.

5.4 Default Value in Parameters

Consider the following:

```
void printSuiteFile(string name = "suite.txt") {
```

```
    ifstream f{name};
    string s;
    while(f >> s) cout << s << endl;
}
```

Calling `printSuiteFile();` prints from `suite.txt` by default.

Note: parameters with defaults must be last.

5.5 Overloading

In C++, functions with different parameter lists can share the same name.

For example, to negate both integers and booleans, we can write

```
int neg(int a) { return -a; }
bool neg(bool b) { return !b; }
```

This is known as **overloading**. The correct version of `neg`, for each function call, is chosen *by the compiler* (i.e. at compile time). It is based on the number and the types of the arguments; just differing in the return type is not enough.

The operators `>>` and `<<` are overloaded – the behaviour depends on the types of arguments.

5.6 Structs

Consider the following struct:

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

In particular, `struct` is not needed in front of `Node *next`, and the pointer is necessary.

5.7 Constants

Declare as many things `const` as possible. This helps to catch errors.

```
const int maxGrade = 100; // must be initialized
```

Let us declare

```
Node n{5, nullptr};
```

`nullptr` is the syntax for null pointers in C++. Do not use `NULL` or `0`.

```
const Node n2 = n;
```

Then `n2` is an immutable copy of `n`.

6 September 24, 2019

6.1 Parameter Passing

Recall:

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

Pass-by-value: `inc` gets a *copy* of `x` and mutates the copy; the original is unchanged.

If a function needs to mutate an argument, pass a pointer.

```
void inc(int *n) { ++*n; }
int x = 5;
inc(&x); // x's address passed by value, changes value at that address
cout << x << endl; // prints 6
```

6.2 References

Why `cin >> x` and not `cin >> &x`?

C++ has another pointer-like type – `reference`.

```
int y = 10;
int &z = y; // z is an lvalue reference to int
            // like a constant pointer, similar to int *const z = &y;
```

References are like `const` pointers with automatic dereferencing.

```
z = 12; // (NOT *z = 12;)
        // now y == 12
```

```
int *p = &z; // gives the address of y
```

In all cases, `z` behaves exactly like `y`; `z` is an alias (another name) for `y`.

Things you cannot do with `lvalue` references:

- leave them uninitialized (e.g. `int &x;`)
  – must be initialized to something with an address (an `lvalue`), since references are pointers

```
int &x = 3;     // not allowed
int &x = y + z; // not allowed
int &x = y;     // allowed
```

- create a pointer to a reference (`int &*x;`)
  – a reference to a pointer is allowed (`int *&x = p;`)
- create a reference to a reference (`int &&r = ...;`)
  – this compiles, but it means something different (later)
- create an array of references (`int &r[3] = { ... };`)

What can be done?

They can be passed as function parameters:

```
void inc(int &n) { ++n; } // no pointer dereference
                          // const pointer to the arg (x)
int x = 5;
inc(x);                   // no &
cout << x << endl;        // prints 6
```

`cin >> x` works because it takes `x` as a reference.

```
istream &operator >> (istream &in, int &n); // cannot copy streams
```

A pass-by-value copies the arguments. If the argument is big, the copy is expensive.

```
struct ReallyBig { ... };
int f (ReallyBig rb) { ... } // copy - slow
int g (ReallyBig &rb) { ... } // alias - fast, but could change rb in the caller
int h (const ReallyBig &rb) { ... } // alias - fast, cannot be changed
```

Advice: Prefer pass-by-const-reference over pass-by-value for anything larger than a pointer, unless the function needs to make a copy anyway (in which case, use pass-by-value).

```
int f (int &n) { ... }
int g (const int &n) { ... }
f(5); // invalid, can't initialize an lvalue reference (n) to a literal value
      // if n changes, can't change the literal 5
g(5); // valid, since n can never be changed, compiler allows this
      // puts 5 in a temporary location, so that the reference n has something to
         point at
```

6.3 Dynamic Memory Allocation

In C:

```
int *p = malloc(... * sizeof(int));
...
free(p);
```

Don't use these in C++. Instead, use `new/delete` – type-aware and less error-prone.

```
struct Node {
    int data;
    Node *next;
};
Node *np = new Node;
...
delete np; // remove from the heap
```

`*np` is on the stack, pointing to some `Node` on the heap.

All local variables reside on the stack. Variables are deallocated when they go out of scope (stack is popped).

Allocated memory resides on the heap. It remains allocated until `delete` is called. If allocated memory is not deleted, that causes a **memory leak** – the program will eventually fail.

6.4 Allocating Arrays on the Heap

Consider the following:

```
Node *nodeArray = new Node[10];
...
delete[] nodeArray;
```

Memory allocated with `new` must be deallocated with `delete`.

Memory allocated with `new []` must be deallocated with `delete[]`.

Mixing these has undefined behaviour.

6.5 Returning by Value/Pointer/Reference

```
Node getMeANode() {
    Node n;
    return n;
} // expensive - n is copied to caller's stack frame on return
```

Returning a pointer (or reference) instead:

```
Node &getMeANode() {
```

```
    Node n;
    return n;
} // bad - dangling reference
```

```
Node *getMeANode() {
    return new Node;
} // OK - returns a pointer to keep data
  // must remember to delete it when done
```

Recommendation: return by value – not as expensive as it looks (later).

6.6 Operator Overloading

Gives new meanings to C++ operators.

```
struct vec {
    int x, y;
};
vec operator + (const vec &v1, const vec &v2) {
    vec v { v1.x + v2.x, v1.y + v2.y };
    return v;
}
vec operator * (const vec k, const vec &v) { // k * v
    return { k * v.x, k * v.y };
} // valid - compiler can figure out the return type
vec operator * (const vec &v, const int k) { // v * k
    return k * v;
}
```

Special case: overloading <<, >>

```
struct Grade {
    int theGrade;
};
ostream &operator << (ostream &out, const Grade &g) {
    return out << g.thegrade << '%';
}
istream &operator >> (istream &in, const Grade &g) {
    in >> g.theGrade;
    if(g.theGrade < 0) g.theGrade = 0;
    if(g.theGrade > 100) g.theGrade = 100;
    return in;
}
```

7 September 26, 2019

7.1 The Preprocessor

The preprocessor transforms the program before the compiler sees it.

`# ...` – preprocessor directive (e.g. `#include`)

There is a new naming convention when including C headers. Instead of `#include <stdio.h>`, use `#include <cstdio>`. Other patterns apply the same way.

```
#define VAR value
```

This sets a preprocessor variable. From that point on, all occurrences of VAR in the source are replaced with value.

```
#define MAX 10
#define FLAG // sets variable FLAG, value is the empty string
int x[MAX]; // transformed to int x[10];
```

Generally, it is better to use `const` definitions instead.

Defined constants are useful for conditional compilation.

```
#define SECURITYLEVEL 1 (or 2)
#if SECURITY LEVEL == 1
    short int // removed if SECURITYLEVEL != 1
#elif SECURITYLEVEL == 2
    long long int // removed if SECURITYLEVEL != 2
#endif
    publickey;
```

A special case of this:

```
#if 0   // never true, all code inside is suppressed
    ... // heavy-duty "comment-out"
#endif  // #if's will nest properly
```

We can also define symbols on the command line.

```
// define.cc
int main() {
    cout << X << endl;
}
```

Then

```
g++14 -DX=1000 define.cc -o define
```

```
./define
```

prints 1000.

`#ifdef NAME` – true if `NAME` has been defined
`#ifndef NAME` – true if `NAME` has not been defined

Note that

```
g++14 -DDEBUG loop.cc
```

enables debugging output.

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

7.2 Separate Compilation

Split programs into composable modules, while each provide:

- **an interface:** type definitions, prototypes for functions – `.h` file
- **an implementation:** full definitions for every provided function – `.cc` file

Recall that a declaration asserts existence, while a definition includes full details and allocates space (variables and functions). An entity can be declared many times, but defined only once.

Interface (`vec.h`):

```
struct vec {
    int x, y;
};
vec operator + (const vec &v1, const vec &v2);
```

Main file (`main.cc`):

```
#include "vec.h"
```

```
int main() {
    vec v {1, 2};
    v = v + v;
    ...
}
```

Implementation (`vec.cc`):

```
#include "vec.h"
vec operator + (const vec &v1, const vec &v2) {
    return { v1.x + v2.x, v1.y + v2.y };
}
```

Compiling separately:

```
g++14 -c vec.cc
g++14 -c main.cc
g++14 main.o vec.o -o main
```

`-c` means to compile only – do not link, do not build an executable. This produces an object file (`.o`). After obtaining the object files, we can link them together.

If we change `vec.cc`, we only need to recompile `vec.cc` and relink.

If we change `vec.h`, we need to recompile both `vec.cc` and `main.cc` (because they both include `vec.h`) and relink.

How can we keep track of dependencies and perform minimal recompilation?

Linux tool: `make`

Create a `Makefile` that says which files depend on which other files.

An example of a `Makefile`:

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

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

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

.PHONY: clean

clean:
    rm *.o main
```

`main` depends on `main.o`, `vec.o`.

`g++ main.o vec.o -o main` is a recipe for building `main` from these files.
`main`, `main.o`, `vec.o`, `clean` are targets.
`-c` is an optional target for removing all binary files.

To use a `Makefile`, type `make` on the command line. It recursively builds the dependencies (if necessary), and then builds `main` (again, if necessary).

For example, if `vec.cc` changes, `make` rebuilds `vec.cc` only, not the whole program, and then relinks.

`make clean` results in a full rebuild.

8 October 1, 2019

8.1 Makefiles, Continued

Generalizing with variables:

```
CXX = g++                       # compiler's name
CXXFLAGS = -std=c++14 -Wall     # compiler options
OBJECTS = main.o vec.o
EXEC = main

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

# can omit recipes from the following
# make guesses that the recipe is ${CXX} ${CXXFLAGS} -c ___.cc

main.o: main.cc vec.h

vec.o: vec.cc vec.h

.PHONY: clean

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

The biggest problem with writing `Makefiles` is writing dependencies and keeping them up to date.

We can get help from `g++`:

```
g++14 -MMD -c vec.cc
```

This creates two files: `vec.o` and `vec.d`. Then `vec.d` contains

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

So we can just include this in the `Makefile`.

```
CXX = g++
CXXFLAGS = -std=c++14 -Wall -MMD -g
OBJECTS = main.o vec.o
DEPENDS = ${OBJECTS:.o=.d}
EXEC = main

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

```
-include ${DEPENDS}

.PHONY: clean

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

As the project expands, we only have to add `.o` files to the `Makefile`.

8.2 Global Variables in a Module

Consider, in `abc.h`:

```
int globalNum; // declaration and definition
```

Every file that includes `abc.h` defines a separate `globalNum` – the program will not link.

Solution: Put the variable in the `.cc` file.

`abc.cc`:

```
int globalNum; // definition
```

`abc.h`:

```
extern int globalNum; // declaration, but not a definition
```

8.3 Headers Included Multiple Times

Suppose we write a linear algebra module.

`linalg.h`:

```
#include "vec.h"
...
```

`linalg.cc`:

```
#include "linalg.h"
#include "vec.h"
...
```

`main.cc`:

```
#include "linalg.h"
#include "vec.h"
```

```
...
```

This does not compile. This is because `main.cc`, `linalg.cc` each get two copies of `vec.h`, so `struct vec` is defined twice.

Solution: `#include guards`

`vec.h`:

```
#ifndef VEC_H
#define VEC_H
// file contents
#endif
```

The first time `vec.h` is included, the symbol `VEC_H` is not defined, so the file is included.

Subsequently, `VEC_H` is defined, so contents of `vec.h` are suppressed.

Always:

– Put `#include guards` in `.h` files.

Never:

– Put `using namespace std;` in header files. The `using` directive will be forced upon any client that includes the file. Always say `std::cin`, `std::string`, etc. in headers.

– Compile `.h` files.

– Include `.cc` files.

8.4 Classes

We can put functions inside of `structs`.

`student.h`:

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

`student.cc`:

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

`client`:

```
Student s{60, 70, 80}
cout << s.grade() << endl;
```

A **class** is essentially a structure type that can contain functions. C++ has a `class` keyword.

An **object** is an instance of a class.

In the above example:

- `Student` is a class.
- `s` is an object.
- The function `grade` is called a **member function** (or **method**).
- `::` is called the **scope resolution operator**. `C::f` means `f` in the context of the class (or namespace) `C`. `::` is like `.`, where the left side is a class (or namespace), not an object.

What do `assns`, `mt`, `final` mean inside of `Student::grade`?

They are the fields of the *current* object – the object upon which `grade` was called.

```
Student billy{...};
billy.grade(); // method call, uses billy's assns, mt, final
```

Formally: Methods take a hidden extra parameter called `this`, a pointer to the object on which the method was called.

```
billy.grade(); // hidden parameter this == &billy
```

Then we can write

```
struct Student {
    ...
    // Method body can be written inside the class
    // Done in lecture for brevity
    // Should put implementations in .cc files
    float grade() {
        return this->assn * 0.4 + this->mt * 0.2 + this->final * 0.4;
    }
};
```

8.5 Initializing Objects

C-style `struct` initialization. This is okay, but limited.

```
Student billy{60, 70, 80};
```

Something better: write a method that initializes: a **constructor**.

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

Implementation:

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

Then we can initialize using

```
Student billy{60, 70, 80};
```

This looks the same as C-style initialization, but this calls the constructor instead.

9 October 3, 2019

9.1 Initializing Objects, Continued

Alternatively:

```
Student billy = Student{60, 70, 80};
```

Heap allocation:

```
Student *pBilly = new Student{60, 70, 80};
```

Advantages of constructors: default parameters, overloading, sanity checks.

```
struct Student {
    ...
    // Default parameters
    Student(int assns = 0, int mt = 0, int final = 0) {
        this->assns = assns;
        this->mt = mt;
        this->final = final;
    }
};
```

Then

```
Student jane{70, 80}; // 70, 80, 0
Student newKid;       // 0, 0, 0
```

Note: Every class comes with a default (i.e. no argument) constructor, which just default-constructs all fields that are objects.

Consider the following:

```
vec v;
```

This calls a default constructor, and does nothing in this case (`int` is not an object).

But the built-in default constructor goes away if you write any constructor.

```
struct vec {
    int x, y;
    vec(int x, int y) {
        this->x = x;
        this->y = y;
    }
};
```

We now have a constructor (not default).

```
vec v{1, 2}; // valid
vec v;       // error
```

9.2 Member Initialization List

Q: What if a struct contains constants or references?

```
struct MyStruct {
    const int myConst;
    int &myRef;
};
```

They must be initialized:

```
int z;
struct MyStruct {
    const int myConst = 5;
    int &myRef = z;
};
```

A more concrete example:

```
struct Student {
    ...
    const int id;
    ...
};
```

`id` is a constant (doesn't change), but not the same for all students.

Q: Where do we initialize them?

Not the constructor body – this is too late, fields must be fully constructed by then.

Q: What happens when an object is created?

1. Space is allocated
2. Fields are constructed in declaration order (constructors run for fields that are objects)
3. Constructor body runs

We need to put our initialization at step 2. We can do this with a **member initialization list (MIL)**.

```
Student::Student(int id, int assns, int mt, int final): id{id}, assns{assns},
    mt{mt}, final{final} { }
```

`id`, `assns`, `mt`, `final` (as in the front `id` in `id{id}`, etc.) are fields. The second `id` in `id{id}`, etc. are parameters.

Notes:

- We can initialize *any* field this way, not just constants and references.
- Fields are initialized *in the order in which they were declared in the class*, even if the MIL orders them differently.
- The MIL is sometimes more efficient than setting fields in the body (otherwise, runs default constructor in step 2, and then reassigns in step 3).

Consider:

```
struct Student {
    string name;
    // in step 2, calls default constructor (name = "")
    Student(string name) {
        // in step 3, name is reassigned
        this->name = name;
    }
};
```

Using MIL:

```
struct Student {
    string name;
    // Step 2: name{name}, Step 3: { } - no reassign
    Student(string name): name{name} { }
};
```

It is better to use the MIL.

Q: What if a field is initialized in line *and* in the MIL?

```
struct vec {
    int x = 0, y = 0;
    vec(int x): x{x} { }
};
```

The MIL takes precedence (field `x` is not initialized twice).

9.3 Copy Constructor

Now consider:

```
Student billy{60, 70, 80};
Student bobby = billy;
```

Q: How does this initialization happen?

The **copy constructor**. It is used for constructing one object as a copy of another.

Note: Every class comes with:

- a default constructor (default-constructs all the fields that are objects; last if you write any constructor)
- copy constructor (copies all fields)
- copy assignment operator
- destructor
- move constructor
- move assignment operator

Building your own copy constructor:

```
struct Student {
    int assns, mt, final;
    ...
    Student(const Student &other): assns{other, assns}, mt{other, mt},
        final{other, final} { } // equivalent to built-in
};
```

Q: When is the built-in copy constructor not correct?

Consider:

```
struct Node {
    int data;
    Node *next;
    Node(int data, Node *next): data{data}, next{next} { }
};
// linked list
Node *n = new Node{1, new Node{2, new Node{3, nullptr}}};
Node m = *n; // *n: copy constructor
Node *p = new Node{*n};
```

In memory:

Simple copy of fields: only the first node is actually copied; *shallow* copy.

If you want a *deep* copy (copies the whole list), you must write your own copy constructor:

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

Notes:

- We have `.next` first, then `*` is applied after
- `new node{*other.next}` calls the copy constructor – recursively copies the whole list

The copy constructor is called:

1. when an object is initialized by another object of the same type
2. when an object is passed by value
3. when an object is returned by value

for now – this will change soon (and there are exceptions to these rules).

Warning: Be careful with constructors that can take one parameter.

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

Single argument constructors set up implicit conversions.

```
Node n{4};
```

But we also have

```
Node n = 4; // implicit conversion from int to Node
```

Consider the following:

```
int f(Node n);
f(4); // this works - 4 implicitly converted to Node
```

Danger: accidentally passing an `int` to a function expecting a `Node`. It is a silent conversion and the compiler does not signal an error, so potential errors are not caught.

To disable the implicit conversion: make the constructor *explicit*.

```
struct Node {
    ...
    explicit Node(int data): data{data}, next{nullptr} { }
};

Node n{4};   // valid
Node n = 4;  // invalid
f(4);        // invalid
f(Node {4}); // valid
```

10 October 8, 2019

10.1 Destructors

When an object is destroyed (stack-allocated: goes out of scope; heap-allocated: is deleted), a method called the **destructor** runs.

Classes come with a destructor (just calls destructors for all fields that are objects).

When an object is destroyed:

1. the destructor body runs
2. the fields' destructors are invoked in reverse declaration order (for fields that are objects)
3. the space is deallocated

Q: When do we need to write a destructor?

Consider:

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

If `np` goes out of scope:

- the pointer `np` is reclaimed (stack-allocated)
- the list is leaked

If we say `delete np;` – calls `*np`'s destructor, which does nothing.

Write a destructor to ensure the whole list is freed.

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

Now: `delete *np;` frees the whole list (recursively calls `*next`'s destructor, deallocating the whole list).

10.2 Copy Assignment Operator

```
Student billy{60, 70, 80};
Student jane = billy; // copy constructor

Student joey; // default constructor
joey = billy; // copy, but not a constructor
```

The last = is a **copy assignment operator** – uses compiler-supplied default.

You may need to write your own copy assignment operator.

Consider:

```
struct Node {
    ...
    // return type is Node so that cascading works (e.g. a = b = c = d)
    Node &operator=(const Node &other) {
        data = other.data;
        delete next;
        next = other.next ? new Node{*other.next} : nullptr;
        return *this;
    }
};
```

The above code is dangerous – consider the following code:

```
Node n{1, new Node{2, new Node 3{3, nullptr}}};
n = n; // deletes n and then tries to copy n to n, undefined behaviour
```

When writing `operator=`, *always* watch out for self-assignment.

```
struct Node {
    ...
    Node &operator=(const Node &other) {
        if(this == &other) return *this;
        data = other.data;
        delete next;
        next = other.next ? new Node{*other.next} : nullptr;
        return *this;
    }
};
```

`new` can fail – if it does, the method will abort. In the above example, `next` has been deleted, but not reassigned. So it points at dead memory, resulting in a corrupted list.

A better option:

```
struct Node {
    ...
    Node &operator=(const Node &other) {
        if(this == &other) return *this;
        Node *tmp = next;
        next = other.next ? new Node{*other.next} : nullptr;
        data = other.data;
        delete tmp;
```

```
        return *this;
    }
};
```

Here, if `new` fails, `Node` will still be in its original state (not corrupted).

Alternative: copy-and-swap idiom.

```
#include <utility>
struct Node {
    ...
    void swap(Node &other) {
        using std::swap;
        swap(data, other.data);
        swap(next, other.next);
    }
    Node &operator=(const Node &other) {
        Node tmp = other;
        swap(tmp);
        return *this;
    }
};
```

11 October 10, 2019

11.1 `Rvalues` and `Rvalue` References

Recall: An `lvalue` is anything with an address. An `lvalue` reference (`&`) is like a constant pointer with auto-deferencing, and is always initialized to an `lvalue`.

Now consider:

```
Node plusOne(Node n) {
    for(Node *p = &n; p; p = p->next) {
        ++p->data;
    }
    return n;
}
Node n{1, new Node{2, nullptr}};
Node n2 = plusOne(n); // copy construction
```

The compiler creates a *temporary* object to hold the result of `plusOne`.

`other` is a reference to this temporary object – copy constructor deep-copies from this temporary.

It is wasteful to have to copy the data from the heap.

In order to know if it is appropriate to steal data, we need to be able to determine whether `other` is a reference to a temporary object or a standalone object.

C++ provides `rvalue` references – `Node&&` is a reference to a *temporary* object (`rvalue`) of type `Node`. Write a version of the constructor that takes a `Node&&`.

```
struct Node {
    ...
    // move constructor, steals other's data
    Node(Node &&other): data{other.data}, next{other.next} {
        other.next = nullptr;
    }
};
```

Similarly:

```
Node m;
m = plusOne(n); // assignment from temporary
```

Move assignment operator:

```
struct Node {
    ...
    // steal other's data, destroy my old data
    // to do this, swap without copy
```

```
    Node &operator=(Node &&other) {
        using std::swap;
        swap(data, other.data);
        swap(next, other.next);
        return *this; // temporary will be destroyed and takes old data with it
    }
}
```

11.2 Copy/Move Elision

Consider:

```
Vec makeAVec() {
    return {0, 0}; // invokes basic constructor
}
Vec v = makeAVec();
```

In g++: this invokes just the basic constructor – no copy constructor nor move constructor.

In some cases, the compiler is allowed to skip calling copy/move constructors (but doesn't have to).

In this example, `makeAVec` writes its result (`{0, 0}`) directly into the space occupied by `v` in the caller, rather than copy it later.

Another example:

```
void doSomething(Vec v) { ... } // pass-by-value - copy/move constructor
doSomething(makeAVec()); // result of makeAVec written directly into the parameter
                         // no copy or move
```

This is allowed, even if dropping constructor calls would change the behaviour of the program (e.g. if constructors print something).

Not expected to know exactly when copy/move elision is allowed, just that it is a possibility. Always assume that they could happen.

If you need all of the constructors to run:

```
g++14 -fno-elide-constructors
```

Note: This can slow your program considerably.

In summary: Rule of 5 (Big 5)

If you need to write any one of:

(1) Copy constructor

(2) Copy assignment operator

(3) Move constructor

(4) Move assignment operator

(5) Destructor

then you usually need to write all five.

Note: `operator=` is a member function, not a standalone function.

When an operator is declared as a member function, `*this` plays the role of the first operand.

```
struct Vec {
    int x, y;
    ...
    Vec operator+(const Vec &other) {
        return {x + other.x, y + other.y};
    }
    Vec operator*(const int k) {
        return {x * k, y * k}; // implements v * k
    }
};
```

Q: How do we implement `k * v`?

Not possible – cannot be a member function, since the first operand is not a `Vec`. This must be written externally.

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

I/O operators:

```
struct Vec {
    ...
    ostream &operator<<(ostream &out) {
        return out << x << ' ' << y;
    }
};
```

Do not write this – it makes `Vec` the first operand, not the second. As a result, it must be used as `v << cout`.

You should define operators `<<`, `>>` as standalone.

Certain operators *must* be members.

- `operator=`
- `operator[]`
- `operator->`

- `operator()`
- `operatorT`, where `T` is a type

11.3 Arrays of Objects

Consider:

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

Vec *vp = new Vec[10];
Vec moreVecs[10];
```

This results in an error: the compiler wants to call the default constructor on each item, and there isn't one.

Some options:

- Provide a default constructor.
- For stack arrays: `Vec moreVecs[3] = { {0, 0}, {1, 1}, {2, 4} };`
- For heap arrays: `Vec *vp = new Vec[3] { {0, 0}, {1, 1}, {2, 4} };`

  Create an array of pointers:

  ```
  Vec **vp = new Vec*[5];
  vp[0] = new Vec{0, 0};
  ...
  for(int i = 0; i < 5; ++i) { delete vp[i]; }
  delete[] vp;
  ```

11.4 Const Objects

```
int f(const Node &n) { ... }
```

Const objects arise often, especially as parameters.

Q: What is a const object?

Fields cannot be mutated.

Q: Can we call methods on a const object? (Issue: method may mutate fields and violate `const`.)

Yes – as long as the methods "promise" to not mutate fields.

```
struct Student {
    int assns, mt, final;
    float grade() const; // <-- const placed here
};
```

The compiler checks that const methods don't modify fields. Only const methods can be called on const objects.

12 October 22, 2019

12.1 Mutable Objects

Suppose we want to collect usage statistics on Student objects.

```
struct Student {
    ...
    int numMethodCalls = 0;
    float grade() { // <-- no longer const
        ++numMethodCalls;
        return ...;
    }
};
```

Then the method can no longer be const, since it modifies `numMethodCalls`. If we remove `const`, we can no longer call `grade` on const `Students`.

But mutating `numMethodCalls` affects only the *physical constness* of `Student` objects, not the *logical constness*.

In order to update `numMethodCalls`, even if the object is const, declare the field `mutable`.

```
struct Student {
    ...
    mutable int numMethodCalls = 0;
    float grade() const {
        ++numMethodCalls;
        return ...;
    }
};
```

12.2 Static Fields and Methods

`numMethodCalls` tracked the number of times a method was called on a *particular* object.

What if we want to track the number of times a method is called over all `Student` objects, or how many students are created?

Static members are associated with the class itself, and not any particular instance (object).

```
struct Student {
    ...
    static int numInstances;
    Student( ... ): ... {
        ++numInstances;
    }
};
```

```
int Student::numInstances = 0; // this goes in .cc file
```

Static fields must be defined external to the class.

Static member functions do not depend on the specific instance (no `this` parameter). They can only access static fields and call other static member functions.

```
struct Student {
    ...
    static int numInstances;
    ...
    static void howMany() {
        cout << numInstances << endl;
    }
};

Student billy{60, 70, 80}, jane{70, 80, 90};
Student::howMany(); // 2
```

12.3 Invariants and Encapsulation

Consider:

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

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

Q: What happens when these go out of scope?
`n1` – The destructor runs and the entire list is deleted. This works fine.
`n2`, `n3` – `n3`'s destructor tries to delete `n2`, but `n2` is on the stack, not the heap. This results in undefined behaviour.

The class `Node` relies on an assumption for its proper operation: that `next` is either `nullptr`, or was allocated by `new`. This assumption is an example of an **invariant**, a statement that must hold true – upon which `Node` relies on.

But we can't guarantee this invariant – we cannot trust the user to use `Node` properly.

For example, in a stack, an invariant could be that the last item pushed is the first item popped – but not if the client can rearrange the underlying data.

It is hard to reason about programs if you cannot rely on invariants.

To enforce invariants, introduce **encapsulation**:

- Want clients to treat objects as black boxes – capsules.
- Creates an abstraction: seal away implementation, and only interact via provided methods.
- Regains control over our objects.

```
struct Vec {
    Vec(int x, int y); // default visibility is public
    private: // cannot be accessed outside of struct Vec
        int x, y;
    public:  // anyone can access
        Vec operator+(const Vec &other);
        ...
};
```

In general, we want private fields – only methods should be public. It is better that default visibility is private.

Switch from `struct` to `class`.

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

The only difference between `struct` and `class` is default visibility – it is public in `struct`, and private in `class`.

Let's fixed our linked list class:

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

```
// list.cc
#include "list.h"

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

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

int List::ith(int i) {
    Node cur = theList;
    for(int j = 0; j < i; ++j) cur = cur->next;
    return cur->data;
}
```

Only `List` can manipulate `Node` objects. Therefore, we can guarantee that the invariant that `next` is always `nullptr` or allocated by `new`.

But now we cannot traverse the list from node to node as we would be able to in a linked list – we can only repeatedly call `ith`, which takes $O(n^2)$ time.

13 October 24, 2019

13.1 Design Patterns

Certain programming problems arise often. We can keep track of good solutions to these problems, and reuse and adapt them.

The solution to the above situation: *iterator pattern*.

Create a class that manages access to nodes. This is an abstraction of a pointer. We can traverse the list without exposing the actual pointers.

Recall:

```
for(int *p = arr; p != arr + size; ++p) {
    cout << *p << endl;
}
```

Now consider:

```
class List {
        struct Node;
        Node *theList;
    public:
        class Iterator {
                Node *p; // "bookmark"
            public:
                explicit Iterator(Node *p) : p{p} { }
                Iterator &operator++() { p = p->next; return *this; }
                int &operator*() { return p->data; }
                bool operator!=(const Iterator &other) { return p != other.p; }
        };

        Iterator begin() { return Iterator{theList}; }
        Iterator end() { return Iterator{nullptr}; }
        // List methods here
};
```

Then:

```
int main() {
    List l;
    l.addToFront(1);
    l.addToFront(2);
    l.addToFront(3);
    for(List::Iterator it = l.begin(); it != l.end(); ++it) {
        cout << *it << endl;
    } // 3 2 1
}
```

This traversal runs in $O(n)$ time.

A shortcut: *automatic type deduction*.

```
auto x = y; // gives x the same type as the value of y
```

We can instead write:

```
for(auto it = l.begin(); it != l.end(); ++it) {
    cout << *it << endl;
}
```

Even shorter: *range-based for loop*.

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

Range loops are available for any class with methods `begin()` and `end()` that produce iterators. The iterator must support `!=`, prefix `++`, and unary `*`.

Note: In the above example, `n` is a *copy* of the list item.

If you want to modify list items (or save copying):

```
for(auto &n : l) {
    ++n;
}
```

13.2 Encapsulation, Continued

The `List` client can create iterators directly:

```
auto it = List::Iterator{nullptr};
```

This violates encapsulation – the client should be using `end()`.

We cannot make `Iterator`'s constructor private – the client cannot call `List::Iterator`, but then neither can `List`.

Solution: Give `List` privileged access to `Iterator` – make `List` a *friend*. `List` will have access to all members of `Iterator`.

```
class List {
        ...
    public:
        class Iterator {
                Node *p;
```

```
            explicit Iterator(Node *p);
        public:
            ...
            friend class List;
    };
    ...
};
```

Now, `List` can still create iterators, but the client can only create iterators by calling `begin()` and `end()`.

Give classes as few friends as possible, otherwise encapsulation will be weakened.

In order to provide access to private fields, use accessor/mutator methods.

```
class Vec {
    int x, y;
  public:
    ...
    int getX() const { return x; } // accessor
    void setY(int z) { y = z; }    // mutator
};
```

What about `operator<<`? This needs `x` and `y`, but it cannot be a member.

If `getX()` and `getY()` are defined, this is fine.

If you don't want to provide `getX()` and `getY()`, make `opeator<<` a friend function.

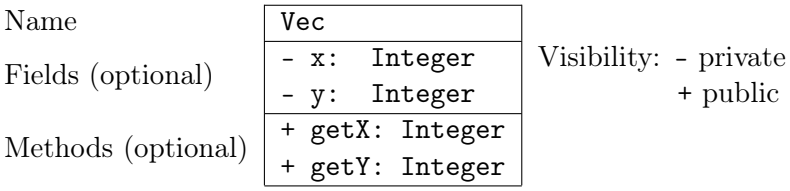
```
// .h
class Vec {
  ...
  friend std::ostream &operator<<(std::ostream &out, const Vec &v);
};

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

13.3 System Modelling

Visualize the structure of the system (abstractions and relationships among them) to aid design and implementation.

A popular standard is UML (Unified Modelling Language).

Modelling classes:



13.4 Relationship: Composition of Classes

```
class Vec {
        int x, y, z;
    public:
        Vec(int, int, int);
};
// Two Vecs define a plane:
class Plane {
        Vec v1, v2;
    ...
};
```

We cannot write:

```
Plane p;
```

This is invalid, we cannot initialize `v1` and `v2` without help. There is no default constructor for `Vec`.

Instead:

```
class Plane {
        Vec v1, v2;
    public:
        Plane() : v1{1, 0, 0}, v2{0, 1, 0} { }
        ...
};
```

Embedding an object (in this case, `Vec`) inside another (in this case, `Plane`) is called *composition*.

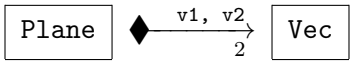
The relationship: a `Plane` "owns" a `Vec` (in fact, it owns two of them).

If `A` owns a `B`, then typically:

- `B` has no identity outside `A` (no independent existence)
- If `A` is destroyed, `B` is destroyed.
- If `A` is copied, `B` is copied (deep copy).

This relationship is typically implemented as a composition of class.

Modelling:



14 October 29, 2019

14.1 Aggregation

Compare car parts in a car ("owns `A`") versus car parts in a catalogue.

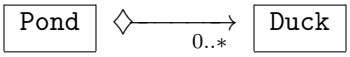
The catalogue contains the parts, but the parts hae an independent existence.

This is called a "has a" relationship (aggregation).

If `A` "has a" `B`, then typically:

- `B` exists apart from its association with `A`.
- If `A` is destroyed, `B` still exists.
- If `A` is copied, `B` is not (shallow copy) – copies of `A` share the same `B`.

UML:



Typical implementation: pointer fields.

```
class Pond {
    Duck *ducks[maxDucks];
    ...
};
```

14.2 Specialization

Suppose you want to track a collection of books.

```
class Book {
       string title, author;
       int length;
    public:
       Book( ... );
       ...
};
```

For textbooks, there is typically a topic.

```
class Text {
       string title, author;
       int length;
       string topic;
    public:
```

```
        Text( ... );
        ...
};
```

For comic books, there is a hero:

Comic
- title: String
- author: String
- length: Integer
- hero: String

This is okay, but this doesn't capture the relationship among `Book`, `Text`, and `Comic`.

How do we create an array (or linked list) that contains a mixture of these?

We could:

(1) Use a union

```
Union BookTypes { Book *b; Text *t; Comic *c; };
BookTypes myBooks[20];
```

(2) Create an array of `void *` – pointer to anything

These two options are not good solutions, since they subvert the type system.

Rather, observe that `Texts` and `Comics` are kinds of `Books` – `Books` with extra features.

To model this in C++: *inheritance*.

```
// Book is unchanged - the base class or superclass.
class Book {
        string title, author;
        int length;
    public:
        Book( ... );
        ...
};

// Text and Comic are called derived classes or subclasses.
// They inherit fields and methods from the base class.
// Text and Comic get title, author, and length.
// Moreover, any method than can be called on Book can be called on Text and Comic.
class Text: public Book {
        string topic;
    public:
        Text( ... );
        ...
};
```

```
class Comic: public Book {
        string hero;
    public:
        Comic( ... );
        ...
};
```

Who can see Book's fields?

- Private in Book – outsiders cannot see them.
- Even subclasses Text and Comic cannot see them.

How do we initialize Text? We need title, author, length, and topic, where the first tree are to initialize the Book part.

```
class Text: public Book {
        string topic;
    public:
        Text(string title, string author, int length, string topic):
            title{title}, author{author}, length{length}, topic{topic} { }
        ...
};
```

This does not compile, for two reasons:

(1) title, author, and length are not accessible in Text, since they are private.

(2) Once again, when an object is created:

1. Space is allocated.
2. Superclass part is constructed **(new)**.
3. Fields are constructed.
4. Constructor body runs.

Step 2 will fail – Book has no default constructor.

To fix this, invoke Book's constructor in Text's MIL.

```
class Text: public Book {
        string topic;
    public:
        Text(string title, string author, int length, String topic):
            Book{title, author, length}, topic{topic} { } // <--- Step 4
         // |          Step 2          |  |   Step 3   |
        ...
};
```

If the superclass has no default constructor, the subclass *must* invoke a superclass constructor in its MIL.

There are good reasons to keep superclass fields inaccessible to subclasses. For instance, you don't know what the subclass might do – cannot guarantee any invariants.

If you want to give subclasses access to certain members, use *protected* visibility.

```
class Book {
    protected: // accessible to Book and its subclasses, no one else
        string title, author;
        int length;
    public:
        Book( ... );
        ...
};
```

This allows the following:

```
class Text: public Book {
        string topic;
    public:
        ...
        void addAuthor(string newAuthor) { author += new Author; } // valid
};
```

However, this is generally not a good idea.

A better option: keep fields private, but provided protected accessors.

```
class Book {
        string title, author;
        int length;
    protected: // subclasses can call these
        string getTitle() const;
        void setAuthor(string newAuthor);
        ...
    public:
        Book( ... );
        bool isHeavy() const;
        ...
};
```

The relationship among `Text`, `Comic`, and `Book` is called a "is a".

A `Text` "is a" `Book`, and a `Comic` "is a" `Book`.

UML:

Now consider the method `isHeavy` – when is a book heavy?

- for an ordinary `Book` – more than 200 pages
- for a `Text` – more than 500 pages
- for a `Comic` – more than 30 pages

```
class Book {
        ...
    protected:
        int length;
    public:
        bool isHeavy() const { return length > 200; }
        ...
};

class Text: public Book {
        ...
    public:
        bool isHeavy() const { return length > 500; }
        ...
};

class Comic: public Book {
        ...
    public:
        bool isHeavy() const { return length > 30; }
        ...
};

Book b {"A small book", ..., 50};
Comic c {"A big comic", ..., 40, "HeroMan"};

cout << b.isHeavy() << endl; // false
cout << c.isHeavy() << endl; // true
```

15 October 31, 2019

15.1 Virtual Keyword, Polymorphism

Now, since inheritance defines an is-a relationship, we can do this:

```
Book b = Comic{"A big comic", ..., 40, "HeroMan"};
```

Q: Is `b` heavy? That is, if we call `b.isHeavy()`, is this true or false?

A: No, `b` is not heavy – `Book::isHeavy` runs, not `Comic::isHeavy`.

This is because it tries to fit a `Comic` object where there is only space for a `Book` object. The `Comic` is *sliced* – the `hero` field is chopped off, and the `Comic` is coerced into a `Book`. So

```
Book b = Comic{"A big comic", ..., 40, "HeroMan"};
```

creates a `Book` (although `Comic`'s constructor is run), and `Book::isHeavy` runs.

When accessing objects through pointers, slicing is unnecessary and doesn't happen.

```
Comic c {..., ..., 40, ...};
Book *pb = &c;
Comic *pc = &c;
pc->isHeavy(); // true
pb->isHeavy(); // false
```

`Book::isHeavy` still runs when we access `pb->isHeavy`.

The compiler uses the type of the pointer (or reference) to decide which `isHeavy` to run – it does not consider the actual type of the object.

The same object behaves differently depending on what type of pointer accesses it.

How do we make `Comic` act like a `Comic` even when pointed to by a `Book` pointer? Declare the method *virtual*.

```
class Book {
    string title, author;
    protected:
        int length;
    public:
        Book(...);
        virtual bool isHeavy() const { return length > 200; }
};

class Comic: public Book {
    ...
    public:
        bool isHeavy() const override { return length > 30; }
};
```

Now, we get

```
Comic c {..., ..., 40, ...};
Comic *pc = &c;
Book *pb = &c;
Book &rb = c;
Book b = c;
pc->isHeavy(); // true
pb->isHeavy(); // true
rb.isHeavy(); // true
b.isHeavy(); // false - sliced
```

Virtual methods choose which class method to run, based on the actual type of the object at run-time. This does not prevent slicing.

Consider a book collection:

```
Book *myBooks[20];
...
for(int i = 0; i < 20; ++i) {
    cout << myBooks[i]->isHeavy() << endl;
}
```

This uses the correct `isHeavy` every time – `Book::isHeavy` for `Books`, `Text::isHeavy` for `Texts`, and `Comic::isHeavy` for `Comics`.

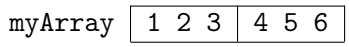
Accommodating multiple types under one abstraction is known as *polymorphism*.

Note: This is why a function `void f(istream &in)` can be passed as an `ifstream` – `ifstream` is a subclass of `istream`.

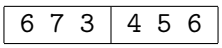
Danger: Consider

```
class One {
    int x, y;
};

class Two: public One {
    int z;
    ...
};

void f(One *a) {
    a[0] = {6, 7};
    a[1] = {8, 9};
}

Two myArray[2] = { {1, 2, 3}, {4, 5, 6} };
f(myArray);
```

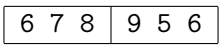
What happens? We start with



When `a[0] = {6, 7};` occurs:



Then when `a[1] = {8, 9};` occurs:



The data is misaligned. *Never* use arrays of objects polymorphically.

15.2 Destructor, Revisited

Consider:

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

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

When an object is destroyed:

1. Destructor body runs.
2. Fields are destructed in reverse declaration order.
3. Superclass part is destructed.
4. Space is deallocated.

In the above example, `X`'s destructor will get called (step 3).

```
X *myX = new Y{10, 20};
delete myX;
```

This causes a memory leak: calls `X`'s destructor, but not `Y`'s destructor, so only `x` is freed, not `y`.

How can we ensure that deletion through a pointer to the superclass will call the subclass destructor? Declare the destructor virtual.

```
class X {
    ...
    public:
        ...
        virtual ~X() { delete[] x; }
};
```

Always declare the destructor virtual in classes that are meant to have subclasses, even if the destructor doesn't do anything.

If a class is not meant to have subclasses, then declare it `final`.

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

15.3 Pure Virtual Methods and Abstract Classes

We take another look at the `Student` class:

```
class Student {
    ...
    public:
        virtual float fees() const;
};
```

Suppose there are two kinds of `Student`: regular and co-op.

```
class Regular: public Student {
    public:
        float fees() const override; // regular student fees
};

class CoOp: public Student {
    public:
        float fees() const override; // co-op student fees
};
```

What should we put for `Student::fees`? Not sure – every student should be `Regular` or `CoOp`.

We can explicitly give `Student::fees` *no* implementation.

```
class Student {
    ...
    public:
        virtual float fees() const = 0; // this method has no (*) implementation
};
```

This is called a *pure virtual method*.

A class with a pure virtual method is called an *abstract class* and cannot be instantiated. For example, the following is invalid:

```
Student s;
```

The purpose of an abstract class is to organize subclasses.

Subclasses of an abstract class are also abstract, unless they implement all pure virtual methods.

Non-abstract classes are called *concrete*.

```
class Regular: public Student {
    public:
        int fees() const override { return 700 * numCourses; }
}
```

UML diagrams:

- **virtual and pure virtual methods** – italics
- **abstract classes** – class name in italics
- **protected** – #
- **static** – underline

16 November 5, 2019

16.1 Inheritance and Copy/Move

Consider the following scenario:

```
class Book {
    public:
        // Defines copy/move constructor, copy/move assignment operator
};

class Text: public Book {
    string topic;
    public:
        // Does not define copy/move operations
};
```

Now define the following:

```
Text t {"Algorithms", "CLRS", 500, "CS"};
Text t2 = t; // No copy constructor in Text
```

This calls `Book`'s copy constructor, then goes field-by-field (i.e. default behaviour) for the `Text` part. This is the same for the other operations.

These operations are the same as what the default would do, adapt accordingly when needed (e.g. for nodes).

```
// Copy constructor
Text::Text(const Text &other): Book{other}, topic{other.topic} {}

// Copy assignment operator
Text &Text::operator=(const Text &other) {
    Book::operator=(other);
    topic = other.topic;
    return *this;
}

// Move constructor
Text::Text(Text &&other): Book{std::move(other)}, topic{std::move(other.topic)} {}

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

A note about the move constructor and move assignment operator:
Even though `other` points at an rvalue, `other` itself is an lvalue (so is `other.topic`). `std::move(x)` forces an lvalue `x` to be treated as an rvalue, so that the "move" versions of the operations run.

Now, consider:

```
Text t1 { ... };
Text t2 { ... };
Book *pb1 = &t1;
Book *pb2 = &t2;
```

What happens if we do `*pb1 = *pb2`? `Book::operator=` runs.

This is known as a *partial assignment* – only the `Book` part is copied.

How can we fix this? Try making `operator=` virtual, and make `Book` a parameter instead of `Text`.

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

class Text: public Book {
    ...
    public:
        Text &operator=(const Book &other) override { ... } // Book instead of Text
};
```

Note: Different return types are okay, but the parameter types must be the same, otherwise it is not an override (and won't compile). This violates "is-a".

By doing this, assignment of a `Book` object to a `Text` object would be allowed.

```
Text t { ... };
Book b { ... };
t = b; // uses a Book to assign a Text - bad (but compiles)
```

Also:

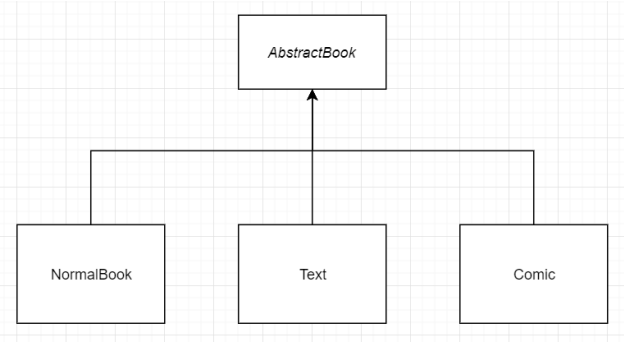
```
Comic c { ... };
t = c; // assigns Comic to a Text - really bad
```

The above is known as *mixed assignment*.

In either case, there are problems. If `operator=` is non-virtual, there is partial assignment throuh base class pointers. If it is virtual, there is mixed assignment.

Recommendation: All superclasses should be *abstract*.

Rewrite the book hierarchy:




```
class AbstractBook {
    string title, author;
    int length;
    protected:
        // Prevents assignment through base class pointers from compiling, but
            implementation still available to subclasses.
        AbstractBook &operator=(const AbstractBook &other);
    public:
        ...
        // Need at least one virtual method. If you don't have one, use the
            destructor.
        virtual ~AbstractBook() = 0;
};

// Other classes are similar
class NormalBook: public AbstractBook {
    public:
        ...
        NormalBook &operator=(const NormalBook &other) {
            AbstractBook::operator=(other);
            return *this;
        }
        ~NormalBook() {}
};
```

Note: The virtual destructor **must** be implemented, even though it is pure virtual.

```
AbstractBook::~AbstractBook() {}
```

17 November 7, 2019

17.1 Templates

Consider:

```
class List {
    struct Node;
    Node *theList;
    ...
};

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

What if you want to store something else? We can write a new class – or use a *template*.

```
template <typename T> class List {
    struct Node {
        T data;
        Node *next;
    };
    Node *theList;
    public:
        class Iterator {
            Node *p;
            explicit Iterator(Node *p): p{p} {}
            public:
                T &operator*() { return p->data; }
                ...
        };
        T &ith(int i) { ... }
        void addToFront(T x) { ... };
        ...
};
```

In the client, we can write:

```
List<int> l1;
List<List <int>> l2;
l1.addToFront(3);
l2.addToFront(l1);
for(List<int>::Iterator it.l1.begin(); it != l1.end(); ++it) {
    cout << *it << endl;
}
for(auto n : l1) { ... }
```

The compiler specializes templates at the source code level, before compilation.

For templates, the entire implementation must go in the header.

17.2 The Standard Template Library (STL)

The STL contains a large number of useful templates.

For example, dynamic-length arrays: `vector`.

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

vector<int> v {4, 5}; // creates the vector 4 5
v.emplace_back(6); // 4 5 6
v.emplace_back(7); // 4 5 6 7
```

Note: brace initialization is different from round brackets in this case.

```
vector<int> v (4, 5); // produces 5 5 5 5
```

Looping over vectors:

```
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) { ... }
```

To iterate in reverse:

```
for(vector<int>::reverse_iterator it = v.rbegin(); it != v.rend(); ++it) {
    ...
}
```

`rbegin` is one past the last element of the array, while `rend` is one before the first element of the array.

To remove the last element:

```
v.pop_back();
```

Using iterators to remove items from inside a vector:

```
auto it = v.erase(v.begin()); // erases first item
it = v.erase(v.begin() + 3); // erases 4th item
it = v.erase(it); // erases at current spot
v.erase(v.end() - 1); // erases last item
```

`erase` returns an iterator pointing to the first item after the erase.

`v[i]` returns the i-th element of `v`. This is unchecked – if you go out of bounds, undefined behaviour occurs.

`v.at(i)` is a checked version of `v[i]`. What happens when you go out of bounds?

17.3 Exceptions

The problem: `vector`'s code can detect the error, but it doesn't know what to do about it. The client can repond, but cannot detect the error.

The solution: Have functions return a status code, or set the global variable `errno`. However, this style of programming encourages programmers to ignore error checks.

In C++, when an error condition arises, the function raises an *exception*. By default, execution stops. But we can write *handlers* to catch exceptions and deal with them.

`vector<T>::at` throws `std::out_of_range` when it fails. We can handle it as follows:

```
#include <stdexcept>
...
try {
    cout << v.at(10000);
} catch(out_of_range r) {
    cerr << "Range error: " << r.what() << endl;
}
... // Execution resumes
```

Now consider:

```
void f() {
    throw out_of_range{"f"}; // constructor argument is what()
}
void g() { f(); }
void h() { g(); }
int main() {
    try {
        h();
    } catch(out_of_range) {
        ...
    }
}
```

What happens?

`main` calls `h`
`h` calls `g`
`g` calls `f`
`f` raises `out_of_range`

Control goes back through the call chain (*unwinds* the stack) until a handler is found. This is all the way back to `main` – `main` handles the exception.

If there is no matching handler in the entire call chain, the program terminates.

A handler might do part of the recovery job (i.e. execute some corrective code and throw another exception).

```
try { ... }
catch(SomeErrorType s) {
    ... // corrective action
    throw SomeOtherError{ ... };
}
```

It could rethrow the same exception:

```
try { ... }
catch(SomeErrorType s) {
    ...
    throw; // as opposed to throw s
}
```

Comparing `throw` with `throw s`:

`throw`: The actual type of `s` is retained.

`throw s`: `s` may be a subtype of `SomeErrorType`, and thus throws an exception of type `SomeErrorType`.

A handler can act as a catch-all.

```
try { ... }
catch(...) { // literally "..." in round brackets, catches all exceptions
    ...
}
```

You can throw anything you want, not just objects.

18 November 12, 2019

18.1 Exceptions, Continued

Never let a destructor throw an exception. If this happens, the program will abort immediately.

If you want a destructor to throw, you can tag it with `noexcept(false)`, but this is not recommended. If a destructor is running during stack unwinding while dealing with another exception, and it throws, you now have *two* active, unhandled exceptions. The program will abort immediately.

18.2 Design Patterns, Continued

Guiding principle:

- Program to the interface, not the implementation.
- Abstract base classes define the interface. Work with base class pointers and call their methods.
- Concrete subclasses can be swapped in and out: abstraction over a variety of behaviours.

For example, the Iterator pattern:

```
class AbstractIterator {
    public:
        virtual int &operator*() = 0;
        virtual AbstractIterator &operator++() = 0;
        virtual bool operator!=(Abstract Iterator &other) = 0;
        virtual ~AbstractIterator();
};

class List {
    ...
    public:
        class Iterator : public AbstractIterator {
            ...
        };
};

class Set {
    ...
    public:
        class Iterator : public AbstractIterator {
            ...
        };
};
```

Then, we can write code that operates over iterators. The following code works over `List` and `Sets`.

```
void for_each(AbstractIterator &start, AbstractIterator &finish, int(*f)(int)) {
    while(start != finish) {
        f(*start);
        ++start;
    }
}
```

18.3 Observer Pattern

Consider a publish/subscribe model:

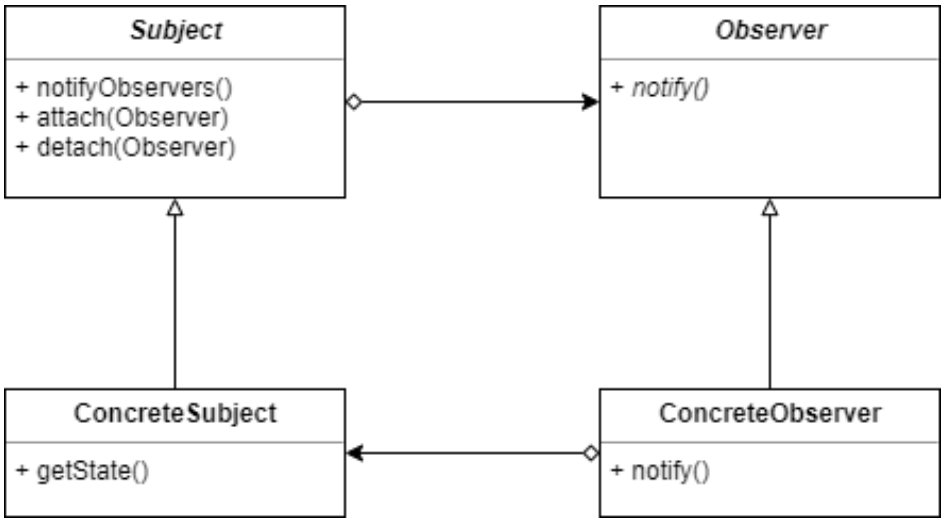
One class: publisher/subject – generates data.

One or more subscriber/observer class – receive data and react to it.

For example, the publisher could be spreadsheet cells, while the observers could be graphs. When cells change, the graphs update.

There can be many different kinds of observer objects – subject should not need to know all the details.

UML for the observer pattern:



Sequence of method calls:

1. `Subject`'s state is updated.
2. `Subject::notifyObservers()` – calls each observer's `notify`.
3. Each observer calls `ConcreteSubject::getState` to query *the state*, and reacts accordingly.

For example: horse races. The subject publishes winners, and the observers are individual bettors: declare victory when their horse wins.

```
// Subject and Observer classes are boilerplate code
class Subject {
    vector<Observer*> observers;
    public:
        void attach(Observer *ob) { observers.emplace_back(ob); }
        void detach(Observer *ob); // remove from observers
        void notifyObservers() { for(auto &ob : observers) ob->notify(); }
        virtual ~Subject() = 0;
};
Subject::~Subject() {}

class Observer {
    public:
        virtual void notify() = 0;
        virtual ~Observer() {}
};

// The following classes are specialized
class HorseRace : public Subject {
    ifstream in; // source of data
    string lastWinner;
    public:
        HorseRace(string source): in{source} {}
        bool runRace() {
            return in >> lastWinner;
        }
        string getState() { return lastWinner; }
};

class Bettor : public Observer {
    HorseRace *subject;
    string name, myHorse;
    public:
        Bettor( ... ): ... {
            subject->attach(this);
        }
        ~Bettor() { subject->detach(this); }
        void notify() {
            string winner = subject->getState();
            if(winner == myHorse) cout << "Win!" << endl;
            else cout << "Lose." << endl;
        }
};
```

Now in main:

```
HorseRace hr;
```

```
Bettor Larry {&hr, "Larry", "RunsLikeACow"};
... // other bettors
while(hr.runRace()) {
    hr.notifyObservers();
}
```

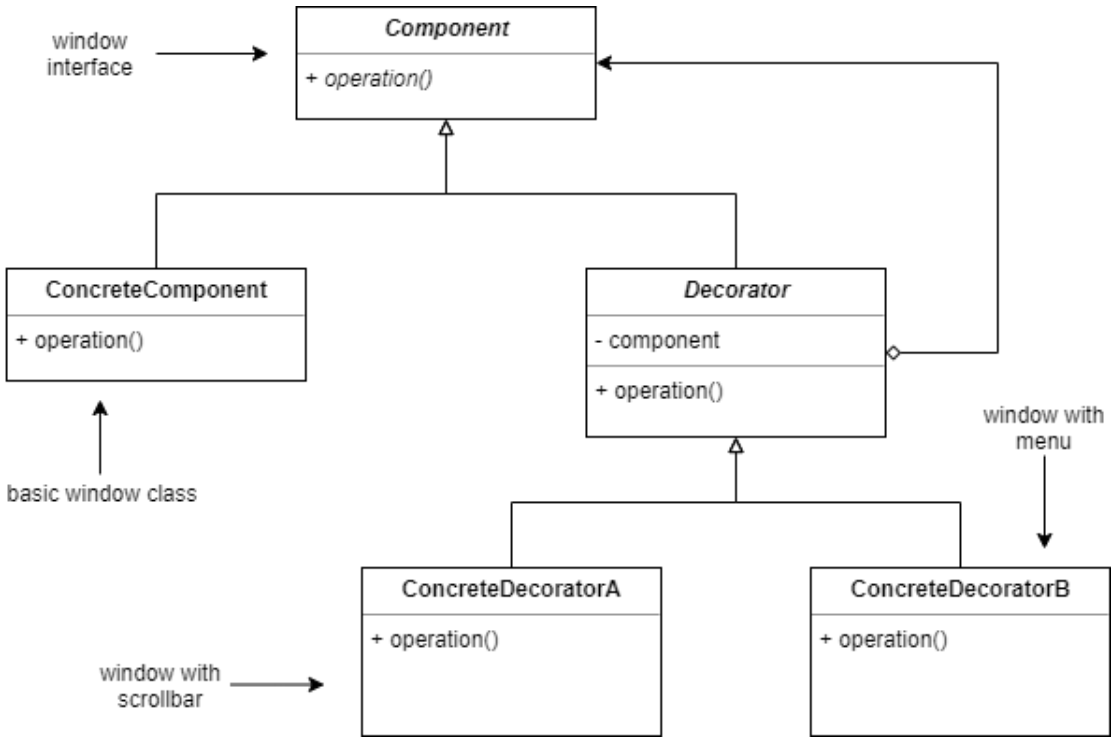
18.4 Decorator Pattern

The Decorator pattern is used if you want to enhance an object at runtime – add functionality or features.

For example, consider a windowing system: start with a basic window. If you add enough text, add a scrollbar. If you press a certain key, add a menu.

We want to choose these enhancements at runtime.

UML for decorator pattern:



`Component`: Defines the interface – operations your objects will provide.

`ConcreteComponent`: Implements the interface.

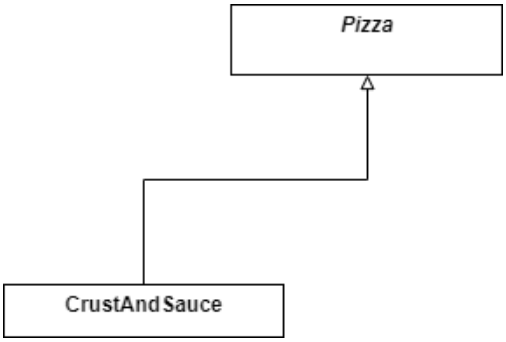
`Decorators`: All inherit from `Decorator`, which inherits from `Component`. Each decorator *is* a `Component` and *has* a component.

For example, `WindowWithScrollbar` is a kind of window and has a pointer to the underlying plain window.

`WindowWithScrollbarAndMenu` *is* a window and has a pointer to a `WindowWithScrollbar`, which has a pointer to a window.

All inherit from `WindowInterface`, so `Window` methods can be used polymorphically on all of them.

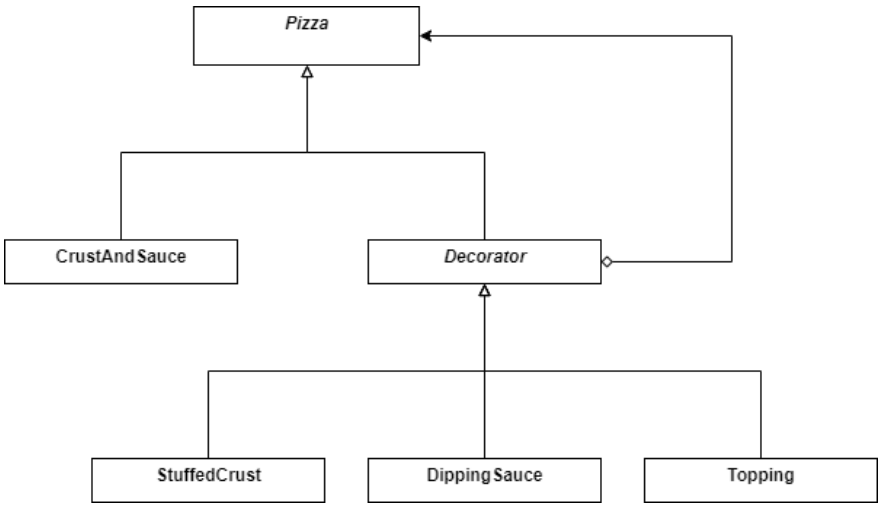
For example, pizza:



Basic pizza is crust and sauce.

```
class Pizza {
    public:
        virtual float price() const = 0;
        virtual string desc() const = 0;
        virtual ~Pizza() {}
};

class CrustAndSauce : public Pizza {
    public:
        float price() const override { return 5.99; }
        string desc() const override { return "pizza"; }
};
```

Using the decorator pattern:

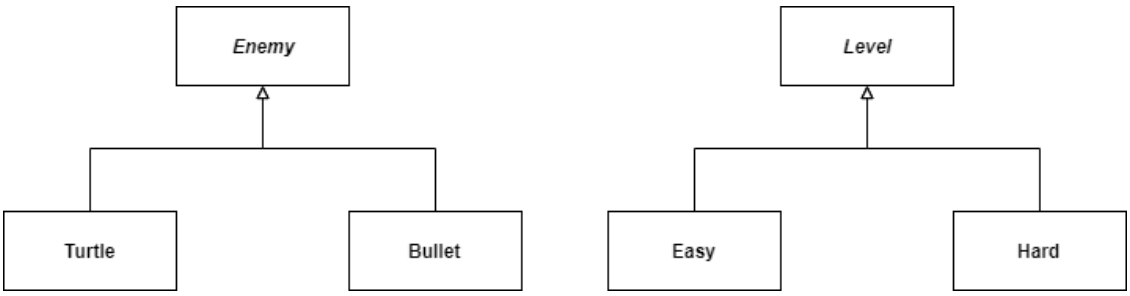


```
class Decorator : public Pizza {
    protected:
        Pizza *component;
    public:
        Decorator(Pizza *p): component{p} {}
        virtual ~Decorator() { delete component; }
};

class StuffedCrust : public Decorator {
    public:
        StuffedCrust(Pizza *p): Decorator{p} {}
        float price() const override { return component->price() + 2.69; }
        string desc() const override { return component->desc() + " with stuffed
            crust"; }
};
```

19 November 14, 2019

19.1 Factory Method Pattern

Suppose we are making a video game with 2 kinds of enemies: turtles and bullets. The system randomly sends turtles and bullets, but bullets are more frequent in harder levels.



Instead, put a factory method in `Level` that creates enemies.

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

class Easy : public Level {
    public:
        Enemy *createEnemy() override {
            // create mostly turtles
        }
};

class Hard : public Level {
    public:
        Enemy *createEnemy() override {
            // create mostly bullets
        }
};
```

Then:

```
Level *l = new Easy; // or Hard
Enemy *e = l->createEnemy();
```

19.2 Template Method Pattern

Suppose we want subclasses to override superclass behaviour, but some aspects must stay the same.

For example, let's say there are red turtles and green turtles.

```
class Turtle {
    public:
        void draw() {
            drawHead();
            drawShell();
            drawFeet();
        }
    private:
        void drawHead() { ... }
        void drawFeet() { ... }
        virtual void drawShell() = 0;
};

class RedTurtle : public Turtle {
    void drawShell() override {
        // draw red shell
    }
};

class GreenTurtle : public Turtle {
    void drawShell() override {
        // draw green shell
    }
};
```

Subclasses cannot access the private methods of the superclass, but they may override them.

In the above example, subclasses cannot change the way a turtle is drawn (head, shell, feet), but can change the way the shell is drawn.

Generalization: the non-virtual interface (NVI) idiom.

A public virtual method is really two things:

- public: An interface to the client. It indicates provided behaviour with pre- and post-conditions.
- virtual: An interface to subclasses. A "hook" to insert specialized behaviour.

These are conflicting goals.

NVI says that:

- all public methods should be non-virtual
- all virtual methods should be private (or at least protected), except the destructor

The following example does not conform to NVI:

```
class DigitalMedia {
    public:
        virtual void play() = 0;
};
```

We can modify it so that it does:

```
class DigitalMedia {
    public:
        void play() {
            doPlay();
        }
    private:
        virtual void doPlay() = 0;
};
```

This code does the same thing, except we have more control now. We can also add before and after code within `play()` (e.g. copyright check, updating play count).

This generalizes the template method pattern: it puts every virtual method inside a template method.

19.3 STL Maps For Creating Dictionaries

Consider "arrays" that map strings to integers.

```
#include <map>
using namespace std;
map<string, int> m;
m["abc"] = 1;
m["def"] = 4;
cout << m["ghi"] << m["def"]; // 04
```

If the key is not present, it is inserted and the value is default-constructed (in this case, 0).

```
m.erase("abc");
if(m.count("def")) ... // 0 means not found, 1 means found
```

Iterating over a map: sorted key order.

```
for(auto &p : m) {
    cout << p.first << ' ' << p.second << endl;
}
```

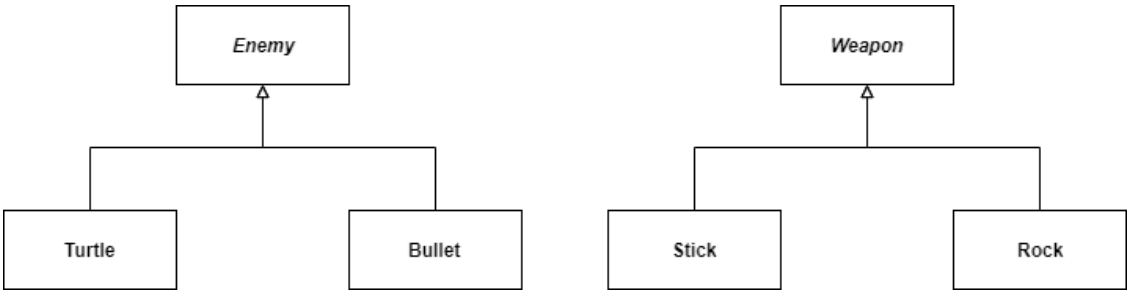
p's type is `std::pair<string, int>` from `<utility>`. Also note that `first` and `second` are public fields, which makes sense since it is not meant to be an abstraction.

19.4 Visitor Pattern

The visitor pattern is used for implementing **double dispatch**.

Virtual methods are chosen based on the actual type (at runtime) of the receiving object. What if you want to choose based on two objects?

Consider once again the video game example: striking enemies with weapons.



We would like to write something like:

```
virtual void(Enemy, Weapon)::strike();
```

However, this is not legal C++.

If `strike` is a method of `Enemy`, we choose based on an enemy, but not a weapon.

If `strike` is a method of `Weapon`, we choose based on a weapon, but not an enemy.

The trick to get dispatch on both (double dispatch): combine overriding and overloading.

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

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

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

They appear the same, but in the case of `Turtle`, `*this` is of type `Turtle&`, while in `Bullet`, `*this` is of type `Bullet&`. These are two different overloads of `strike`.

```
class Weapon {
    public:
        virtual void strike(Turtle &t) = 0;
        virtual void strike(Bullet &b) = 0;
};

class Stick : public Weapon {
    public:
        void strike(Turtle &t) override { /* strike turtle with stick */ }
        void strike(Bullet &b) override { /* strike bullet with stick */ }
};

class Rock : public Weapon {
    public:
        void strike(Turtle &t) override { /* strike turtle with rock */ }
        void strike(Bullet &b) override { /* strike bullet with rock */ }
};
```

Now, consider:

```
Enemy *e = new Bullet; // don't know type of enemy, assume Bullet
Weapon *w = new Rock; // don't know type of weapon, assume Rock
e->beStruckBy(w);
```

What happens?

`Bullet::beStruckBy` runs (virtual method)
It calls `Weapon::strike`, `*this` is of type `Bullet`
`Bullet` version of `strike` is chosen at compile-time
Virtual method `strike` on `Weapon` resolves to `Rock::strike(Bullet &)`

20 November 19, 2019

20.1 Visitor Pattern, Continued

The visitor pattern can be used to add functionality to existing classes, without changing or recompiling the classes themselves.

For example, consider adding a visitor to the `Book` hierarchy.

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

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

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

We claim that this is the same as the striking enemies with weapons example above. If we replace `Book` with `Enemy`, `accept` with `beStruckBy`, and `Visitor` with `Weapon`, they are indeed the same.

Application: Track how many of each type of `Book` we have:

`Book`: by author

`Text`: by topic

`Comic`: by hero

We can use a `map<string, int>`. We could write a virtual `updateMap` method, or write a visitor.

```
struct Catalogue : public BookVisitor {
    map<string, int> theCat;
    void visit(Book &b) override { ++theCat[b.getAuthor()]; }
    void visit(Text &t) override { ++theCat[t.getTopic()]; }
    void visit(Comic &c) override { ++theCat[c.getHero()]; }
};
```

20.2 Compilation Dependencies

When does a file *need* to include another file?

Remark: A long sequence of includes in the beginning of compile errors is an indicator that there is a cycle of includes.

Consider:

```
// a.h
class A { ... };
```

The following are modules that have `A` in them:

```
// b.h
#include "a.h"
class B : public A {
    ...
};

// c.h
#include "a.h"
class C {
    A a;
};

// d.h
class A;
class D {
    A *pa;
};

// e.h
class A;
class E {
    A f(A a);
};

// f.h
#include "a.h"
class F {
    A f(A a) { a.g() }
};
```

For `B` and `C`, we need to know the size of `A`, so we need to include `a.h`.

For `D` and `E`, there is no compilation dependency: they just need to know that `A` exists. In the case of `D`, all pointers are the same size. In the case of `E`, we don't need to know anything about `A`.

For `F`, we need details about `A`, so we need an include.

If the code doesn't need an include, don't create a needless compilation dependency by including unnecessarily.

When `A` changes, only `A`, `B`, `C`, and `F` need recompilation.

In the *implementations* of `D` and `E`:

```
// d.cc
#include "a.h"
void D::f() {
    pa->something();
}
```

Do the include in the `.cc` file instead of the `.h` file (where possible).

20.3 Bridge Pattern

Consider the `XWindow` class:

```
class XWindow {
    Display *d;
    Window w;
    int s;
    GC gc;
    unsigned long colours[10];
public:
    ...
};
```

What if we need to add or change a private member? All clients must recompile. It would be better to hide these details away.

Solution: `pImpl` idiom ("pointer to implementation")

Create a second class `XWindowImpl`:

```
// XWindowImpl.h
#include <X11/Xlib.h>
struct XWindowImpl {
    Display *d;
    Window w;
    int s;
    GC gc;
    unsigned long colours[10];
};

// Window.h
class XWindowImpl;
class XWindow {
```

```
    XWindowImpl *pImpl;
public:
    ... // no change
};
```

Notes on the `Window.h` class:

- There is no need to include `Xlib.h`.
- We forward declare the `Impl` class (`class XWindowImpl;`).
- There is no compilation dependecy on `XWindowImpl.h`.
- Clients also don't depend on `XWindowImpl.h`.

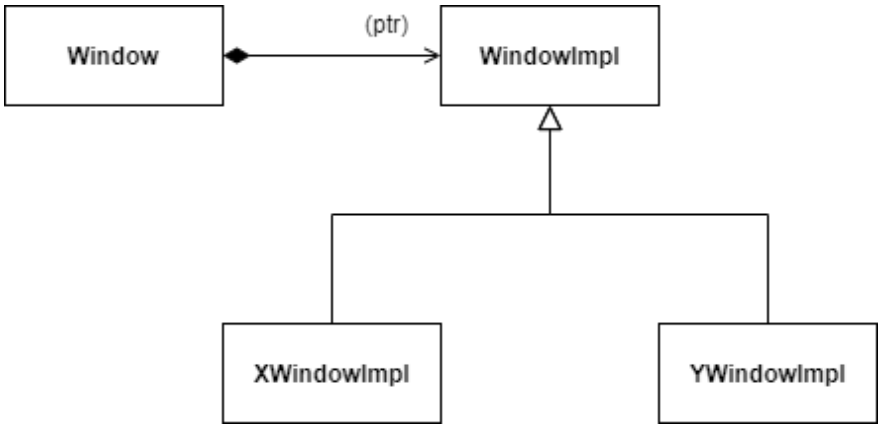
```
// Window.cc
#include "Window.h"
#include "XWindowImpl.h"
XWindow::XWindow( ... ): pImpl{new XWindowImpl} {}
XWindow::~XWindow() { delete pImpl; }
```

For other methods, replace fields `d`, `w`, `s`, `gc`, `colours` with `pImpl->d`, `pImpl->w`, `pImpl->s`, `pImpl->gc`, and `pImpl->colours`.

When changing private fields, we only need to recompile `Window.cc` and relink.

Generalization: What if there are several possible window implementations, say `XWindow` and `YWindow`.

Then make `Impl` struct a superclass.



The `pImpl` idiom and hierarchy of implementations is called the *bridge pattern*.

20.4 Measures of Design Quality

Coupling: How much distinct modules depend on each other.

Low coupling:

- Communicate via function calls with basic parameters and results.
- Modules pass arrays and structs back and forth.
- Modules affect each other's control flow.

High coupling:

- Modules have access to each other's implementation (friends).

Cohesion: How closely elements of a module are related to each other.

Low cohesion:

- Arbitrary grouping of unrelated elements (e.g. `<utility>`).
- Slightly more cohesive: elements share a common theme, otherwise unrelated, maybe share some base code (e.g. `<algorithm>`).
- Elements manipulate state over the lifetime of an object (e.g. open/read/close files).
- Elements pass data to each other.

High cohesion:

- Elements cooperate to perform exactly one task.

Low cohesion results in poorly organized code.

The goal is to achieve low coupling and high cohesion.

21 November 21, 2019

21.1 Decoupling the Interface (MVC)

Your primary program class should not be printing things.

```
class Chessboard {
    ...
    cout << "Your move" << endl;
};
```

This is bad design: it inhibits code reuse.

What if you want to reuse `Chessboard`, but not have it communicate via stdout?

We could give the class stream objects, where it can send its input/output.

```
class Chessboard {
    istream &in;
    ostream &out;
    ...
    out << "Your move" << endl;
};
```

This is better, but what if we don't want to use streams at all?

`Chessboard` should not be doing any communication at all.

Single Responsibility Principle: "A class should have only one reason to change."

The game state and communication are *two* reasons.

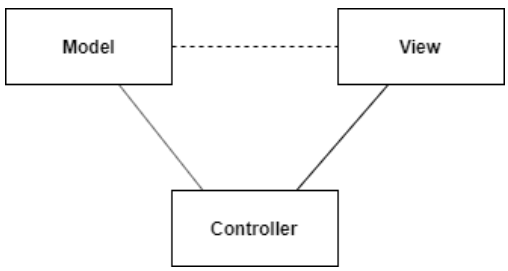
A better option: communicate with the chessboard via parameters or results. Confine user communication to outside the game class.

`main` should not be communicating – it is hard to reuse code if it is in `main`.

There should be a class to manage interaction, separate from the state class.

Architecture: Model-View-Controller (MVC)

Separate the distinct notions of the data ("model"), the presentation of the data ("view"), and the control or manipulation of the data ("controller").



The model can have multiple views (e.g. text/graphics). It doesn't need to know their details. This is a classic Observer pattern (or could communicate through controller).

The controller mediates control flow between model and view. It might handle input (or could be the view).

This decouples presentation and control, and as a result, promotes reuse.

21.2 Exception Safety

Consider:

```
void f() {
    MyClass mc;
    MyClass *p = new MyClass;
    g();
    delete p;
}
```

There are no leaks – but what if `g` throws?

During stack unwinding, all stack-allocated data is cleaned up. The destructors run, and memory is reclaimed.

However, heap-allocated objects are not destroyed. Therefore, if `g` throws, `*p` is leaked (`mc` is not).

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

This works: but there is duplication of code. How else can we guarantee that something (e.g. `delete p`) will happen no matter how we exit `f` (normal or exception)?

In some languages, "finally" clauses guarantee certain final actions – C++ is not one of them.

The only thing we can count on in C++: destructors for stack-allocated data will run.

Therefore, use stack-allocated data with destructors as much as possible: use the guarantee to your advantage.

C++ idiom: Resource Acquisition Is Initialization (RAII)

Every resource should be wrapped in a stack-allocated object whose job is to delete it.

For example, files:

```
void foo() {
    ifstream f{"file"};
    ...
}
```

Acquiring the resource (`"file"`) is initializing the object.

The file is guaranteed to be released when `f` is popped from the stack (`f`'s destructor runs).

This can be done with dynamic memory:

```
#include <memory>
class std::unique_ptr<T>
```

This takes a `T*` in the constructor. The destructor will delete the pointer. In between, we can deference, just like a pointer.

```
void f() {
    MyClass mc;
    std::unique_ptr<MyClass> p{new MyClass};
    g();
}
```

Now, we are guaranteed to have no leaks.

We can phrase the above slightly differently:

```
void f() {
    MyClass mc;
    auto p = make_unique<MyClass>(); // constructor arguments in parenthesis
    g();
}
```

Difficulty:

```
unique_ptr<C> p{new C{...}};
unique_ptr<C> q = p;
```

What happens when a `unique_ptr` is copied? We don't want to delete the same pointer twice (double free).

Instead: copying is disabled for `unique_ptr` – they can only be moved.

However, we can have one `unique_ptr` and many regular pointers to an object, and we can think of this as ownership of the resource.

A simple implementation of `unique_ptr` (repo):

```
template<typename T> class unique_ptr {
    T *ptr;
public:
    unique_ptr(T *p): ptr{p} {}
    ~unique_ptr() { delete ptr; }
    unique_ptr(const unique_ptr &other) = delete;
    unique_ptr &operator=(const unique_ptr &other) = delete;
    unique_ptr(unique_ptr &&other): ptr{other.ptr} {
        other.ptr = nullptr;
    }
    unique_ptr &operator=(unique_ptr &&other) {
        std::swap(ptr, other.ptr);
        return *this;
    }
    T &operator*() { return *ptr; }
}
```

`unique_ptr` also has a method called `get`:

```
T *q = p.get();
```

This can also be done:

```
T *q = &*p;
```

Copying Pointers

First, we need to answer the question of ownershiip.

Who will own the resource? Who will have responsibility for freeing it?

That pointer should be a `unique_ptr` – all other pointers can be raw pointers (can fetch the underlying raw pointer with `p.get()`).

If there is true shared ownership (i.e. any of several pointers might need to free the resource), use `std::shared_ptr`.

```
void f() {
    auto p1 = std::make_shared<MyClass>();
    if( ... ) {
        auto p2 = p1;
    } // p2 popped, pointer not deleted
} // p1 popped, pointer is deleted
```

Shared pointers maintain a *reference count*, the count of all `shared_ptr`s pointing at the same object. Memory is freed when the number of `shared_ptr`s pointing to it is about to hit 0.

22 November 26, 2019

22.1 Back to Exception Safety

Three levels of exception safety for a function `f`:

1. **Basic guarantee:** If an exception occurs, the program will be left in some valid state (no leaks, no corrupted data structures, and invariants are maintained).
2. **Strong guarantee:** If an exception is raised while executing `f`, the state of the program will be as it was before `f` was called.
3. **No-throw guarantee:** The function `f` will never throw or propogate an exception, and will always accomplish its task.

For example:

```
class A { ... };
class B { ... };
class C {
    A a;
    B b;
public:
    void f() {
        a.g(); // may throw (strong guarantee)
        b.h(); // may throw (strong guarantee)
    }
};
```

Is `C::f` exception safe?

If `a.g()` throws, nothing has happened yet, so there are no problems.

If `b.h()` throws, the effects of `a.g()` must be undone to offer the strong guarantee. This is very hard or impossible to do if `a.g()` has non-local side effects.

So `C::f` is not exception safe.

If `A::g` and `B::h` do not have non-local side effects, we can use copy-and-swap.

```
class C {
    ...
    void f() {
        // If the first four lines throw, original is intact
        A atemp = a;
        B btemp = b;
        atemp.g();
        btemp.h();
        // What happens if copy assignment throws?
        a = atemp;
        b = btemp;
```

```
    }
};
```

It would be better if the "swap" part was no-throw. Copying pointers cannot throw.

Solution: Use the `pImpl` idiom.

```
struct CImpl {
    A a;
    B b;
};

class C {
    unique_ptr<CImpl> pImpl;
    ...
    void f() {
        auto temp = make_unique<CImpl>(*pImpl);
        temp->a.g();
        temp->b.h();
        std::swap(pImpl, temp); // no throw
    }
};
```

Now, `C::f` has strong guarantee.

22.2 Exception Safety and the STL: Vectors

Vectors encapsulate a heap-allocated array. They follow RAII: when a stack-allocated vector goes out of scope, the internal heap-allocated array is freed.

```
void f() {
    vector<MyClass> v;
    ...
}
```

When `v` goes out of scope, the array is freed, and the `MyClass` destructor runs on all objects in the vector.

However:

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

The array is freed. Pointers don't have destructors, so any objects pointed at by the pointers are not deleted.

`v` doesn't know whether the pointers in the array *own* the objects they point at.

Now consider:

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

The array is freed, the `unique_ptr` destructor runs, so the objects *are* deleted. We don't have to do any explicit deallocation.

Now, consider `vector<T>::emplace_back`, which offers the strong guarantee. This is because if the array is full (i.e. `size == cap`):

- New array is allocated.
- Copy objects over (copy constructor) – if a copy constructor throws, destroy the new array, and the old array is still intact.
- Delete old array (no-throw).

But copying is expensive and the old data will be thrown away. Wouldn't moving the objects be more efficient?

The old array is no longer intact if the move constructor throws, so we cannot offer the strong guarantee.

If the move constructor offers the no-throw guarantee, then `emplace_back` will use the move constructor. Otherwise, it will use the copy constructor, which will be slower.

So, move operations should offer the no-throw guarantee, and you should indicate that they do.

```
class MyClass {
public:
    ...
    MyClass(MyClass &&other) noexcept { ... }
    MyClass &operator=(MyClass &&other) noexcept { ... }
};
```

If you know a function will never throw or propagate an exception, declare it `noexcept`, which facilitates optimization.

At the minimum, moves and swaps should be `noexcept`.

23 November 28, 2019

23.1 Casting

In C:

```
struct Node n;
int *ip = (int*) &n;
```

The above is a cast: it forces C to treat a `Node*` as an `int*`.

C-style casts should be avoided in C++. If you must cast, use a C++-style cast.

There are four kinds:

- `static_cast` – "sensible casts"

  For example, `double` to `int`:

  ```
  double d;
  void f(int x);
  void f(double x);
  f(static_cast<int>(d));
  ```

  Superclass pointer to subclass pointer:

  ```
  Book *b = new Text{ ... };
  Text *t = static_cast<Text*>(b); // t->getTopic() valid
  ```

  Note: You are taking responsibility that `b` actually points at a `Text`. If this is wrong, it results in undefined behaviour.

- `reinterpret_cast` – unsafe, implementation-specific, "weird" casts

  ```
  Student s;
  Turtle *t = reinterpret_cast<Turtle*>(&s);
  ```

- `const_cast` – for converting between const and non-const

  This is the only C++ cast that can "cast away const".

  ```
  void g(int *p); // know that g doesn't modify p
  void f(const int *p) {
      ...
      g(const_cast<int*>(p));
      ...
  }
  ```

- `dynamic_cast` – unsure if it is safe to convert, better to do a tentative cast and see if it works

  ```
  Book *pb;
  Text *pt = dynamic_cst<Text*>(pb);
  ```

If the cast works (`*pb` really is a `Text` or a subclass of `Text`), then `pt` points at the object. Otherwise, `pt` will be `nullptr`.

```
if(pt) cout << pt->getTopic();
else cout << "Not a text";
```

But these are all operations on raw pointers. Can we do this with smart pointers?

Yes: `static_pointer_cast`, `const_pointer_cast`, and `dynamic_pointer_cast` all cast `shared_ptr`s to `shared_ptr`s.

We can use dynamic casting to make decisions based on an objects run-time 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 << "Normal book";
}
```

However, this code is tightly coupled to the `Book` hierarchy, and may indicate bad design.

Better options: use virtual methods or visitor pattern (if possible).

Dynamic casting also works with references:

```
Text t { ... };
Book &b = t;
Text &t2 = dynamic_cast<Text&>(b);
```

If `b` points to a `Text`, then `t2` is a reference to the same `Text`.

If not, the exception `bad_cast` is raised (no such thing as a null reference).

Note: Dynamic casting only works on classes with at least one virtual method.

With dynamic reference casting, we can now solve the polymorphic assignment problem.

```
Text &Text::operator=(const Book &other) { // virtual
    const Text &textOther = dynamic_cast<const Text&>(other);
    if(this == &textOther) return *this;
    Book::operator=(other);
    topic = textOther.topic;
    return *this;
}
```

If `other` is not a text, then dynamic casting throws, and none of the operations after occur.

23.2 How Virtual Methods Work

Consider the following two classes:

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

class Vec2 {
    int x, y;
public:
    virtual void f() { ... }
};
```

What is the difference?

```
Vec v {1, 2};
Vec2 w {1, 2};
```

Do they look the same in memory?

```
cout << sizeof(v) << ' ' << sizeof(w); // 8 16
```

First, note that 8 is the space for 2 ints, and there is no space for the `f` method.

The compiler turns methods into ordinary functions and stores them separately from objects.

Recall that if a method is virtual, the choice of which version to run is based on the type of the actual object, which the compiler can't know at compile-time.

The correct method must be chosen at run-time.

For each class with virtual methods, the compiler creates a table of functions pointers (the vtable).

```
class C {
    int x, y;
    virtual void f();
    virtual void g();
    void h();
    virtual ~C();
};
```

The vtable then looks like

C
f
g
~C

with each of the entries pointing to the actual code `C::f`, `C::g`, and `C::~C`.

`C`'s objects have an extra pointer (the vptr) that points to `C`'s vtable.

What occurs when a virtual method is called:

- Follow vptr to vtable.
- Fetch the pointer to the action method from table.
- Follow the function pointer and call the function.

All of these occur at run-time.

Virtual function calls incur a small overhead cost. Also, declaring at least one virtual method adds a pointer to the object, a vptr. Thus, classes with no virtual methods produce smaller objects than if some functions were virtual.

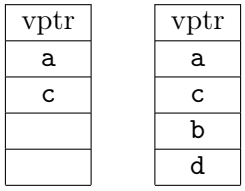
24 December 3, 2019

24.1 How Objects are Laid Out

Concretely, how is an object laid out? This is compiler dependent.

In g++:

```
class A {
    int a, c;
    virtual void f();
};

class B: public A {
    int b, d;
};
```



This is better so that a pointer to `B` looks like a pointer to `A` if you ignore the last two fields. Also, you will always know where the vptr is (always first).

24.2 Multiple Inheritance

A class can inherit from more than one class.

```
class A {
public:
    int a;
};

class B {
public:
    int b;
};

class C: public A, public B {
    row f() {
        cout << a << ' ' << b;
    }
};
```

Challenges

Suppose `b` and `c` inherit from `A`.

```
class D: public B, public C {
public:
    int d;
};
```

If `B` and `c` from `A`, should there be one `A` prt of `D`< or two (default)?

Should `B::a` and `C::a` be the same or different?

Consider the "deadly diamond".

Make `A` a virtual base class – virtual inheritance.

```
class B: virtual public A { ... };
class C: virtual public A { ... };
```

How will this be laid out?

vptr
`A` fields
`B` fields
`C` fields
`D` fields

g++:

vptr
`B` fields
vptr <– `C*`
`C` fields
`D` fields
vptr <– `A*`
`A` fields

`B` needs to be laid out so that we can find its `A` part, but these diagrams show that the distance to the `A` part isn't necessarily there.

SOLUTION: The location of the base class is stored in the vtables.

The diagram doesn't look like all of `A`, `B`, `C`, `D` simultaneously, but slices of it look like `A`, `B`, `C`, `D`.

Pointer assignment among `A`, `B`, `C`, `D` changes the address stored in the pointer.

```
D *d = ...;
A *a = d; // changes its address to point at the A part
```

Note: `static_cast`, `dynamic_cast` will do this, while `reinterpret_cast` will not.

24.3 Template Functions

Consider:

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

int f() {
    int x = 1, y = 2;
    int z = min(x, y); // don't have to say min<int>
}
```

C++ can infer that `T` is an `int` from the types of `x` and `y`. This applies to template functions only.

If C++ cannot determine `T`, you can always tell it explicitly: `z = min<int>(x, y);`.

```
char w = min('a', 'c'); // T = char
auto f = min(1.0, 3.0); // T = double
```

For what types `T` can `min` be used? Any type for which `operator<` is defined.

Recall:

```
void for_each(AbstractIterator start, AbstractIterator finish, int(*f)(int)) {
    while(start != finish) {
        f(*start);
        ++start;
    }
}
```

This works as long as `AbstractIterator` supports `!=`, `*`, `++`. Also, `f` can be called as a function.

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

Now, for example, we can write

```
void f(int n) { cout << n << endl; }
int a[] = {1, 2, 3, 4, 5};
for_each(a, a + 5, f); // prints the array
```

C++ STL Library `<algorithm>`

```
template<typename InIter, typename OutIter>
```

```
OutIter copy(InIter first, InIter last, OutIter result) {
    /* Copies one container range [first, last) to another,
       starting at result.
       Note: does not allocate new memory. */
}
```

For example:

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