LIHORNE.COM

CS 246

Object-Oriented Software Development

Dr. Mark Prosser • Fall 2013 • University of Waterloo

Last Revision: December 13, 2013

Table of Contents

Abstract

These notes are intended as a resource for myself; past, present, or future students of this course, and anyone interested in the material. The goal is to provide an end-to-end resource that covers all material discussed in the course displayed in an organized manner. If you spot any errors or would like to contribute, please contact me directly.

Brad Lushman: Any such notes have no official status in this course and we do not endorse their contents, nor will we point out or correct any errors in them. Use at your own risk.

1 Introduction

This course uses Linux. Thus one of the following are necessary

- a bash shell
- a Linux install
- Linux virtual machine
- ssh into linux.student.cs.uwaterloo.ca
- PuTTy, WinSCP, Cygwin, etc

This course is structured into understanding Linux shells (bash), using the C++ programming language, studying various tools and some software engineering concepts.

2 The Linux Shell

The Linux shell is an interface to the operating system. It provides a way to get the operating system to accept commands and carry out the necessary work, or to manage things for us. There are two kinds of interfaces; **graphical** and **command-line**.

Remark 2.1. A graphical shell is common on Windows and Mac. Operations typically include things like double-clicking, dragging, hovering, etc. In Linux, Gnome and KDE are examples. GUIs make simple tasks easy and intuitive. The shortshide however are that tasks can become harder to carry out as complexity increases.

Remark 2.2. Command line interfaces accept commands at a prompt and are traditional for UNIX/Linux. Windows has the "DOS" Command Prompt, and Mac is UNIX based. These interfaces have a steeper learning curve but are much more powerful than graphical interfaces. A typical command line interface would look something like this:

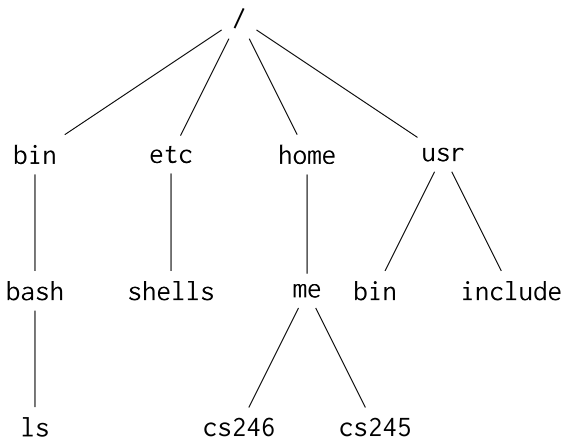
```
[mycomputer /var/nsm/] $
```

In this course we use **bash** (short for "Bourne Again Shell"). To check what shell you're using, run

```
echo $0
```

2.1 Linux File System

The file system consists of **files** (e.g., programs, data) and **directories**. Directories can contain other directories as well as files. Directories are arranged in hierarchical tree structure. The typical hierarchy is:



We can specify any file or directory by its **path** from the root. For example, /var/nsm/bin/.

Definition 2.1 (Current Directory)**.** There is a notion of present working directory that, when in a bash shell, is the current directory that will provide an environment for commands to run in. Usually, a new bash shell will open to the home directory. The command pwd will return this directory. As well, each directory has a shortcut within itself, and is symbollicaly represented by ".".

There are two kinds of paths.

Definition 2.2 (Relative Path)**.** Relative path is a path that is referring to your current directory. For example, if pwd return /home/ then any file I refer to or directory refers to what is in the /home/ folder.

```
[var] $ pwd
/var
[var] $ cd mail/
[mail] $ pwd
/var/mail
```

Definition 2.3 (Absolute Path)**.** An absolute path starts with the root directory. For example /home/ will always refer to exactly that directory, even if there is a home directory in some other directory.

Remark 2.3. The following special directories exist:

- . refers to the current directory. (relative)
- .. refers to the parent directory. (relative)
- ~ refers to the home directory. (absolute)
- ~userid refers to the home directory of userid. (absolute)

Change Directory To change the current directory, use the cd command. For example, cd /home/ takes you to the /home/ directory.
List Files/Directories To list all files in the current directory, use the ls command. For example, ls /home/ will list all files or directories in /home/. To show hidden files, pass the -a flag.
Create Directory Use mkdir dirname
Remove Directory Use rmdir dirname
Remove Directory AND Contents rm -r dirname

2.2 Wildcard Matching / Globbing

What if I want to see just the text files? (.txt extension). The * character symbolizes a "globbing" pattern, matching any string at all to it. Thus,

```
ls *.txt
```

will match all .txt files and list them. The act of globbing by the command line is that the shell will substitude every file matching the expression and replaces the single term with multiple terms of matching files. If there is no match, the original glob considers the * to be a literal character. For example,

ls *.txt ⟺ ls abc.txt def.txt ghi.txt

This is a bash constant, and therefore works for every command.

2.3 Variables

Declaration of a variable is done like so: x=1 (no spaces!) and accessing variables in other contexts uses a dollar sign, so echo $x will return 1. For example

```
$ dir=~/cs246
$ echo ${dir}
/home/rmprosse/cs246
$ ls ${dir}
a0 a1 a2 etc
```

Note that the ∼ is expanded before the variable is assigned.

In bash there are several "global" variabels that are already set up for you. To list these variables, use the env command. For example,

```
$ env
SHELL=/bin/bash
USER=lihorne
PATH=/usr/local/Cellar/emacs/24.3/bin:/usr/local/bin:/bin:/usr/sbin:/sbin:/usr/bin:/usr/↩
   texbin
PWD=/Users/lihorne
LANG=en_CA.UTF-8
HOME=/Users/lihorne
LOGNAME=lihorne
_=/usr/bin/env
```

The most important of these global variables is the PATH. The path is a colon (":") delimited list of directories that are searched in order when a command is issued.

2.4 Quoting

Quotes are used to control how the shell interprets strings. Note that globbing does not occur inside quoted strings, they are treated as literals. As well, single quotes protect everything except a single quote, and double quotes protect everything except double quotes ("), back quotes (‘), and dollar signs ($). When inside double quotes, these characters have to be escaped. Escaping is done using the backslash (\).

Example 2.1.

```
$ echo $PATH
/usr/local/bin:/bin:/usr/sbin:/sbin:/usr/bin:/usr/texbin
```

```
$ echo "$PATH"
/usr/local/bin:/bin:/usr/sbin:/sbin:/usr/bin:/usr/texbin
$ echo '$PATH'
$PATH
```

What if I wanted to use the strijng "*.txt" including the quotes? Simply put it all inside single quotes, or double quotes with escaping like this

```
$ echo '"*.txt"'
"*.txt"
$ x="\"*.txt\""
$ echo ${x}
"*.txt"
```

2.5 Input/Output Redirection

There is a command called cat that displays the contents of a file or returns input. So, cat hello.txt returns hello, world. If there is no argument cat waits for input and returns that input after it had completed. Hit ˆC to stop the process, and ˆD to stop the input. If however we wanted to redirect the output of cat then we can use a redirection operator like so:

```
$ cat hello.txt
hello, world
$ cat > hello.txt
my name is liam
^D
$ cat hello.txt
my name is liam
```

In general, command args > file will redirect the output from command args to file. This is called **output redirection**. We can do both input and output redirection using redirection, for example cat < inputfile.txt > outputfile.txt will take the contents of inputfile.txt and overwrite outputfile.txt with it.
Note that every process has 3 streams. They are

1. Standard Input - which is input into a program
2. Standard Output - that a program outputs
3. Standard Error - that a program outputs, it is a seperate output stream for error messages and is seperate from regular output

By default, stdin is conected to the keyboard, stdout and stderr are connected to the screen. Input redirection connects stdin to a file and output redirection connects stdout to a file. We can redirect stderr as well, for example program < in.txt > out.txt 2> errlog.txt. The 2> will only output errors to the file. Note that redirection will create the file if necessary and **replace entire contents**. Additionally, you can use >> to **append** output.

2.6 Pipes

Pipes let you connect the output of one program to the input of another (just like a pipe, wow). Connecting two commands is done using a vertical bar ("|"). This sets the second program's stdin to the first program's stdout.

Example 2.2. How many words occur in the first 20 lines of myfile.txt? We can use this series of commands

```
$ head -20 myfile.txt | wc -w
42
```

This example illustrates using the output of `head -20 myfile.txt` which is the first 20 lines of a file, and using it as the input for `wc -w` which counts the number of words in the input.

Example 2.3. Suppose `words.txt`, `words2.txt`, ... contains lists of words, one word per line. Print a duplicate-free list of all the words in these files, `words*.txt`. We will consider two programs that are useful in this problem.

i. `uniq` - removes adjacent duplicate lines

ii. `sort` - sorts lines

Therefore if the input is sorted, uniq removes all duplicates. Our solution is the command

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

2.7 Command Substitution

Can we use the output of a command as a **parameter** of another? The answer is yes, we simply put the command in **backquotes** (or quasiquotes). For example, echo `‘date‘` will execute the `date` command and use it as input into echo.

Remark 2.4. Recall that double quotes do not protect back quotes, shell expands back quotes inside double quotes. For example echo ῾̈‘date‘ ̈ has the same meaning. Alternatively, we can use `$()` instead of ‘ ‘. For example, echo `$(date)` has the same effect.

2.8 Grep - pattern matching in text files

We now study a new tool called `grep`, which stands for "Global Regular Expression Print". There is also an extended version called `egrep` which is equivalent to `grep -E`, which interprets a pattern as an extended regular expression (i.e. forces grep to behave as egrep). The general format is

`grep pattern file`, which prints lines in `file` which match `pattern`.

Example 2.4. Suppose we want to print all lines in our websites homepage, `index.html` that contain the pattern `cs246`. We simply call `grep cs246 index.html`.

How many lines match?

`grep cs246 index.html | wc -l` or `grep -c cs246 index.html`

We can use expressions called **regular expressions**; note that this is a completely seperate idea to globbing, and although there are many things in common and the syntax is similar, they are different.

Example 2.5. Search for `cs246` or `CS246`. The solution involes a regex that checks on the first alphabetic letters.

`grep -E ”cs246|CS246” index.html` or `grep -E ”(cs|CS)246” index.html`

Observe that the parentheses allow grouping for the "or" operator "|". Some more identical commands include egrep `”(c|C)(s|S)246” index.html` (which included mismatches of upper or lowercase characters) or egrep `”[cC][sS]246” index.html`.

Notation 2.1.

- [$a_1 a_2 a_3 \ldots a_n$] will match any character a_i in between [and]. So, [abc] $\equiv$ a|b|c.

- [ˆ$a_1 \ldots a_n$] will match anything **except** the characters mentioned.
- To denote an optional preceding character, use the ? character. For example `cs ?246` will match both `cs246` and `cs 246`.
- To denote optional preceding characters repeated 0 or more times, use `*`. For example, `cs*246` will match `cs246` and `cssssssssss246` and `c246`.
- To denote a single instance of any character, use `.` (for example, `.*` denotes 0 or more of any character, which mimics globbing)
- The caret ˆ has a different meaning at the beginning of a line. In the case

  ```
  egrep "ˆcs246” index.html
  ```

  indicates lines starting with `cs246`.
- Similarly, the dollar sign $ indicates the end of a line. So,

  ```
  egrep”ˆcs246$” index.html
  ```

  will match lines containg **only** `cs246` and nothing else.
- The + sign will match 1 or more of the preceding pattern

Example 2.6. Print all lines of even length.

```
egrep "ˆ(..)*$” index.html
```

Example 2.7. Print all files in current directory whose names contain **exactly one a**.

```
ls | grep -E "ˆ[ˆa]*a[ˆa]*$”
```

2.9 Permissions

Recall the `ls -l` will print the **long form** directory listing. For example

```
$ ls -l
total 0
drwxr-xr-x+  4 lihorne  staff   136 16 Sep 20:57 Desktop
drwxr-xr-x+ 10 lihorne  staff   340 14 Sep 10:35 Documents
drwxr-xr-x+ 12 lihorne  staff   408 17 Sep 11:32 Downloads
drwx------\ 9 lihorne staff 306 15 Sep 16:37 Dropboxdrwx------ 60 lihorne staff 2040 14 Sep 02:06
    Librarydrwxr-xr-x+ 6 lihorne staff 204 14 Sep 04:55 Moviesdrwxr-xr-x+ 6 lihorne staff 204 14 Sep 04:55
    Musicdrwxr-xr-x+ 9 lihorne staff 306 14 Sep 04:55 Picturesdrwxr-xr-x+ 6 lihorne staff 204 21 Aug 19:28
    Public
```

These listings mean,

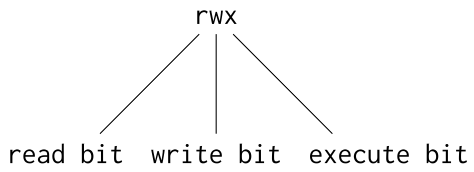
$$\underbrace{\texttt{d}}_{\texttt{type}}\ \underbrace{\texttt{rwxr-xr-x+}}_{\texttt{permissions}}\ \underbrace{\texttt{6}}_{\texttt{\#links}}\ \underbrace{\texttt{lihorne}}_{\text{owner}}\ \underbrace{\texttt{staff}}_{\text{group}}\ \underbrace{\texttt{204}}_{\texttt{size}}\ \underbrace{\texttt{21 Aug 19:28}}_{\texttt{last modified}}\ \underbrace{\texttt{Public}}_{\text{name}}$$

We're interested in the permissions. The permissions are split into three pieces,

$$\underbrace{\texttt{rwx}}_{\texttt{user}}\ \underbrace{\texttt{r-x}}_{\texttt{group}}\ \underbrace{\texttt{r--}}_{\texttt{other}}$$

- **user bits:** what the file's owner can do with it
- **group bits:** what members of the file's group can do with it (other than owner)
- **other bits:** what other users can do

Each set has



What these mean depends on the type of file.

Bit	Ordinary File	Directory
r	file can be read	contents can be read (i.e., `ls`)
w	file can be modified	contents can be modified (i.e., add / remove)
x	files can be executed	directory can be navigated (i.e., in cd)

Note. If a directory's `x` bit is not set, it is not accessible whatsoever; no file or subdirectory within it is reachable.

Remark 2.5. We list some more features of `grep`:

- `grep -v` is an inverted grep, it displays the lines that **don't** match.
- `ls` **pipes** output as a column (not line) (detects if `stdout` is a terminal).
- `pcrgrep` is a `grep` variant with a multiline option (`-m`).
- Use a dash within `[]` to specify range. For example, `[0-9]`,`[a-z]` are ranges over the first 10 digits and the 26 lowercase letters.

We can use the command chmod which stands for "change permissions" to alter some of these bits. The general format is

`chmod mode file`

The term `mode` is formatted such that there are three ownership groups,

Ownership Class	Operator	Permissions
u for user (owner)	+ to add permission	r for read
g for group	- to remove permission	w for write
o for other	= to set permission exactly	x to execute
a for all (u,g, and o)		

2.10 Shell Scripts

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

Example 2.8.

```
#!/bin/bash      # <-"she-bang"
date             # (tells Linux to execute
whoami           #  this file as a bash script)
pwd
```

We first give the file execute permissions using

```
chmod u+x myscript
```

Then run the file with `./myscript`.

Note. Command-line arguments are accessed as **variables** such as `${1}`, `${2}`, etc

Example 2.9. Check whether a word is in the dictionary. We create a file `isItAWord` that contains the following:

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

The script checks a dictionary file for the regex that is a line of only the word given as an argument to the script. Thus,

`./isItAWord hello` prints `hello` if it is there, nothing otherwise.

Example 2.10. A "good" password is not in the dictionary. Determine whether a word might be a "good" password.

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

where `/dev/null` is a directory that suppresses output.

Note. Every program returns a status code when finished. For example, `grep` returns 0 if a match was found and 1 otherwise. In Linux, 0 means success and nonzero means a failure. Essentially,

$$\texttt{program} = \begin{cases} \texttt{success} & \text{if status code is 0} \\ \texttt{failure} & \text{otherwise} \end{cases}$$

The variable `${?}` is the status code of the most recently executed program. So, we now add the following to our script:

```
if [ ${?} -eq 0 ]; then
  echo Not a good password
else
  echo Maybe a good password
fi
```

Note. We have a particular usage for this program that we want, and that is that the user should pass exactly one argument. To ensure this use, or rather to provide helpful error messages should it be used incorrectly, we use another special variable, `$#`.

It's common to write a function with usage information as shown here:

```
usage () {
  echo "Usage: $0 password" >&2
  exit 1
}

if [ ${#} -ne 1 ]; then
  usage
fi
```

And the complete script then defines a usage function, checks the number of arguments, runs the dictionary lookup, and finally returns whether or not a password is deemed to be good.

```
#!/bin/bash
# Answers whether a candidate word might be a good password

usage () {
  echo "Usage: $0 password" >&2
  exit 1
}

if [ ${#} -ne 1 ]; then
  usage
fi
egrep "^$1$" /usr/share/dict/words > /dev/null

if [ $? -eq 0 ]; then
  echo Not a good password
else
  echo Maybe a good password
fi
```

Notation 2.2. The general format for if statements is

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

Example 2.11. We'll have a look at an example of loops. This example prints the number 1 to $1. This file is from lectures/shell/scripts/count.

```
#!/bin/bash
# count limit ---counts the numbers from 1 to limit

usage () {
  echo "Usage:  $0 limit" 1>&2
  echo "  where limit is at least 1" 1>&2
  exit 1
}

if [ $# -ne 1 ]; then
  usage
fi

if [ $1 -lt 1 ]; then
  usage
fi

x=1
while [ $x -le $1 ]; do
  echo $x
  x=$((x + 1))
done
```

Note.

- 1>&2 "ties" stdout (1) to stderr (2)
- $((...)) does **arithmetic expansion**

Now suppose we want to loop over a list, this next example portrays this.

Example 2.12. Rename all .C files to .cc. This next script is from lectures/shell/scripts/renameC.

```
#!/bin/bash
# Renames all .C files to .cc

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

Note.

for v in list : sets variable v to each word in list
${name%C} : returns the value of the name variable without the trailing C

Example 2.13. Write a script that returns the date of the last friday of this month.
To accomplish this we use the awk command in conjunction with the cal command. Thus one possible command to accomplish this is. For example,

```
cal January 2014| awk ’print $6’ | grep [0-9] | tail -1
```

The following script is from /shell/scripts/payday. There are two steps, the first is to find the date and the second is to report the answer.

```
#!/bin/bash
# Returns the date of the next payday (last Friday of the month)
# Examples:
# payday (no arguments) -- gives this month’s payday
# payday October 2012 -- gives payday in October 2012

answer () {
  if [ $2 ]; then
     preamble=${2}
  else
     preamble="This month”
  fi
  if [ $1 -eq 31 ]; then
    echo "${preamble}’s payday is on the ${1}st.”
  else
    echo "${preamble}’s payday is on the ${1}th.”
  fi
}

answer ‘cal $1 $2 | awk ’{print $6}’ | grep "[0-9]” | tail -1‘ $1
```

Example 2.14. The following string counts the number of lines that a word appears in a file.

```
#!/bin/bash
# countWords word file
#  Prints the number of times word occurs in file

x=0
for word in ‘cat "$2"‘; do
  if [ $word == $1 ]; then
    x=$((x + 1))
  fi
done
echo $x
```

2.11 Testing

Testing is the essential part of software engineering; it often takes much more time than expected. Additionally, it is ongoing, not just at the end, and tests should be written before the program is written. Test suites are used for streamlining the process of testing. Note that testing is not debugging, and testing must happen first.
Testing can't guarantee your program is correct, it can only prove it is incorrect. Ideally, the developer and tester are different people (but not in this course).

- Human testing - people examine code, looking for flaws. This includes code inspection, and walkthroughs.
- Machine testing - systematically run programs through test input, check the output against specifications. Can't test everything, so choose cases carefully.

There are two main approaches to testing.

- **Black-box Testing** - no knowledge of implementations
- **White-box Testing** - full knowledge of implementation

Gray-box testing could be considered testing done with some knowledge of implementation but not all.
Typically, black-box testing is the first to occur and is then supplemented by white-box testing. We analyze some techniques of both types.

Black-Box

- Think about classes of inputs to avoid redundancy. For example, numeric ranges, positive, negative, etc.
- Boundary cases (edge cases), consider boundaries between classes
- Intuition and experience should be used to guess errors
- Extreme cases, push the limits of the program

White-Box

- Check all logical paths
- Test to make every function run

Strategies

- Single Component:

  - Unit testing: test individual components (e.g., small code, program, class)
- Multiple Component:
  - Integration testing: test interaction between multiple components
- All Components:
  - Functional testing: program works as expected in normal conditions
  - Regression testing: test modified version of a previously validated program.
  - System testing: test the functionality, performance, reliability, and security of the entire system
  - Performance testing: program efficiency
  - Volume testing: ability to handle inputs in different volumes (small and large)
  - Stress testing: ability to handle extreme volume of data in limited amount of time
  - Acceptance testing: Operating the system in the user environment with standard user input scenario

3 C++

3.1 Introduction - Hello World

In C, the Hello World program looks like this:

```
#include <stdio.h>
int main() {
  printf("Hello World\n");
  return 0;
}
```

The C++ version looks like this:

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

Note. In C++, the `main` program must return an `int` (status code). Emitting the return statement will however return 0 anyway. Note that `stdio.h` and `printf` are still available in C++, but the preferred method is as shown above. That is, using the header `<iostream>`, and with output mechanism `std::cout << ... << ... << std::endl` (meaning there can be multiple uses of the << operator).
`std::cout` is the standard output stream, and `std::endl` is the end of line indicator.
`using namespace std;` lets you refer to `std::cout` and `std::endl` without the `std::`.

Most C programs are valid C++ programs, that is C++ is a near-perfect superset of C.

To compile a C++ program: `g++ program.cc` $\underbrace{\texttt{-o program}}_{\text{specifies name of executable}}$. If the name is emitted, then the compiled program is called `a.out`. Then executing is as simple as running `./program`.

3.2 C++ Input & Output

There are 3 I/O Stream objects:

- cin - reading from stdin.
- cout - printing to stdout.
- cerr - printing to stderr.

The I/O operators are:

- << - "put to" (output).
- >> - "get from" (reading).

Example 3.1.

cin >> x means getting data from cin and writing it to x.

cout << x means taking x and writing it to stdout.

cerr << x means writing x to stderr.

Note. cin >> ignores whitespace (tabs, spaces, newlines). That is, cin >> x >> y; looks for two integers seperated by whitespace.

Some scenarios:

(1) What if the input doesn't contain an integer next? Then the statement fails, and the variable is not assigned.

(2) What happens if input is exhausted before we get two integers? Then the statement fails, and the variable is not assigned.

How do we detect if either of these scenarios has occurred? Then,

read failed ⟹ cin.fail() returns true.
end-of-file reached ⟹ cin.eof() **and** cin.fail() will return true, but not until the attempted read fails.

Example 3.2. Read integers from stdin, then echo them one per line to stdout. Stop when the end of the file is reached or the read fails.
Version 1:

```cpp
#include <iostream>
using namespace std;
int main() {
  int i;
  while (true) {
    cin >> i;
    if (cin.fail()) break;
    cout << i << endl;
  }
}
```

Note. cin can be used as a condition in an if statement, since

if (cin) ⟺ if (!cin.fail())

How does this work? The if statement can use booleans, integers, or pointers as conditions. The compiler changed cin to either 0 or the address of cin. This is because cin is a pointer, a reference to an object.
Note that >> is a function under cin. That is,

- cin >> x is equivalent to cin.>>(x)
- >> returns cin itself
- This allows: cin >> x >> y (which can be written as cin.>>(x).>>(y);, which is then cin.>>(y))

Also note that cin is of type istream.

Example 3.3. This example reads all integers from stdin and echos them, one per line, to stdout. It skips non-integer inputs. Note that cin.clear() clears the internal error state and cin.ignore() removes the part in input stream that generated the error.

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

int main () {
  int i;
  while (true) {
    if (!(cin >> i)) {
      if (cin,eof()) break;
      else {
        cin.clear();
        cin.ignore();
      }
    }
    else {
      cout << i << endl;
    }
  }
}
```

3.3 Strings

C-like strings, that is null-terminated character arrays (char[], char*), are well supported in C++. That is a string "Hello" can be defined like char mystr[] = "Hello"; where it is represented like [H][e][l][l][o][null]. C++ has a std::string class that can be used like a basic data type. This reduces the maintenance overhead (e.g., memory management) and is more reliable (it hides the null character). Additionally, they can grow and shrink as needed, and are safer to manipulate.

We can use the following string operators

- == Equality
- != Inequality
- <,>,<=,>= Comparison
- s.length Length
- s[i] Character extraction

- s = s1 + s2 Concatenate (or +=)

This requires adding

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

Creating and initializing a string is done like this:

```
string s1;                         // initialize to empty
string s2("My string in C++");     // initialize by constructor
string s3 = "my value";            // initialize by assignment
string s4 (s2);                    // initialize by another string
s2 = "another string";             // directly assign value, no strcpy
s2 = s1;
```

Strings can also be concatenated using the + operator:

```
string s1 = "Hello";
string s2(" world");
string s3 = s1 + " " + s2 + "!";
cout << s3 << endl;
cout << (s1 + " my" + s2 + ".") << endl;
```

However, strings can still be accessed as though they are characters in an array. So,

```
string city = "Waterloo";
for (int i = 0; i < city.length(); i++) {
  cout << city[i] << " - ";
}
```

would return W - a - t - e - r - l - o - o -.

cin reads strings with the following semantics:

- discard leading spaces
- read non-white characters into string
- stop at next whitespace character

To read the entire line, including the leading spaces, use getline(cin, str);.

Refer to http://www.cplusplus.com/reference/string/string/ for some useful **methods** from the string class.

3.4 I/O Manipulation

The behaviour of cout can be controlled using I/O manipulations. The following header must be included:

```
#include <iomanip>
```

Maniuplators are not variables for input / output but rather control the I/O formatting for all literals / variables after it, continuing to the next I/O expression for a specific stream file. Except for setw, they all apply to all proceeding values.

oct	integral values in octal
dec	integral values in decimal
hex	integral values in hexadecimal
left / right (default)	values with padding after / before values
boolalpha / noboolalpha (default)	bool values as false / true instead of 0/1
showbase / noshowbase (default)	values with / without prefix 0 for octal and 0x for hex
showpoint / noshowpoint (default)	print decimal if no fraction
fixed (default) / scientific	float-point values without / with exponent
setprecision(N)	fraction of float-point values in maximum of N columns
setfill(‘ch‘)	padding character before / after value (default blank)
setw(N)	next value only in minimum of N columns
endl	flush output buffer and start new line (output only)
skipws (default) / noskipws	skip whitespace characters (input only)

Note. Manipulators and Floats

- fixed - fixed decimal place (no exponent)
- scientific - scientific notation (use exponent)

The default is neither fixed not precision, precision meand number of significant digits to display after (not including) decimal point.

Example 3.4. To produce formatted output showing decimal, octal, and hexadecimal values for 16 through 20:

```
cout << showbase
     << setw(6) << "Dec"
     << setw(6) << "Oct"
     << setw(6) << "Hex" << endl;
for (int i = 16; i < 20; ++i) {
  cout << dec << setw(6) << i
       << oct << setw(6) << i
       << hex << setw(6) << i << endl;
}
```

3.5 Working with files

In C we use fopen, then fscanf, then finally fclose. Note that fscanf returns the number of arguments read, and -1 on error. fscanf does not take care of string overflow. The C example:

```
FILE *f = fopen("suite.txt", "r");
char name[80];
while (fscanf(f, "%s", name) == 1) {
  printf("%s\n", name);
}
fclose(f);
```

Now in C++ we can do the following:

```
ifstream file("suite.txt");
string s;
while (file >> s) {
  cout << s << endl;
}
```

ifstream is used for reading a file, and ofstream for writing to a file. The file is automatically closed once out of scope. All operations on cin/cout are valid for ifstream/ofstream.

Example 3.5. Constructing strings from other strings or numbers. See buildString.cc:

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

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

We revisit a past example, of reading and echoing integers, skipping non-integers. See readIntsSS.cc

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

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

3.6 Default Function Parameters

Example 3.6.

```
void printSuiteFile (string name = "suite.txt") {
  ifstream file (name.c_str());
  string s;
  while (file >> s)
    cout << s << endl;
  }
}
```

Note that the file stream initializer must be given a C-style string, use .c_str() to get a C-style string from std::string. Now,

```
print SuiteFile("suite2.txt"); // prints suite2.txt
print SuiteFile(); // prints suite.xt
```

Note that optional parameters must be last.

3.7 Overloading

In C:

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

However in C++ we can do the following:

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

So two functions can have the same name and the function that is called will depend on the type of the argument given, which will be registered by the compiler. This is called **overloading**. The compiler uses the number and types of parameters to choose which function to call. Overloads must differ in number of parameters or type of parameters if they have the same number. Indistinguishable functions will raise an error by the compiler. Additionally, any optional arguments must no cause any ambiguity.

3.8 Declaration Before Use

You cannot use something before it has been declared. So how do we do **mutual recursion**?

```
bool even (unsigned n) {
  if (n == 0) return true;
  else return odd(n-1);
}
bool odd (unsigned n) {
  if (n == 0) return false;
  else return even(n-1);
}
```

This fails because even calls odd before odd has been declared. The solution to this problem is **forward declaration**. We simply declare the existence of odd, we don't write it but we declare it.

```
bool odd (unsigned n); // forward declaration
bool even (...) {
  ...
}
bool odd (...) {
  ...
}
```

There is an important distinction between a **declaration** (which only asserts the existence of the entity) and a **definition** which is the full entity itself, including all details. An entity can be declared several times but defined only once.

3.9 Pointers

We'll show by example.

```
int n = 5;
int *p = &n; // p is a pointer to an int, and p's value is the address of n

cout << p << endl; // displays some hexadecimal number representing the address of n
cout << *p << endl; // 5 (*p is the value at the address of p)

int **pp; // ptr to ptr to int
pp = &p;
cout << **pp << endl; // 5
```

3.10 Arrays

Pointers are very closely related to arrays. This is because an array asignment like `int a[] = {1,2,4,8};` has a being a pointer to the first element of the array, that is, a is equal to the address of the element with value 1. Furthermore

```
*a = a[0] = 1;
p = &a[0];
*(a+1) = a[1] = 2;
```

The notation of writing `[]` after the array name is really just a dereferencing operator with a shift, that is `a[i] = *(a+i)`.

Note. The + operator on strings only works with a character and a string object, thus ”abd” + ”def” is not a valid statement.

3.11 Constants

Constants are useful when you have variables that you don't intend to change. For example, `const int maxGrade = 100`. Constant definitions must be **initialized immediately**. It is a good idea to declare things constant when possible, it is a helpful way to catch errors.

Example 3.7.

```
Node n1 = {5,NULL};
const Node n2 = n1; // creates constant copy of n1
```

Note that `const` can also be used with pointers. For example,

```
const int *A;
int const *A; // same thing
```

In this example, it is equivalent to a variable pointer to a constant integer (value is constant). Also,

```
int *const B; // is a constant pointer (the address is constant, pointing to variable integer)
```

We can combine these two types of situations to create a constant pointer to a constant integer.

```
int const *const C; // constant pointer to constant integer
```

Note that `A` can be reassigned but `*A` cannot. We can't change where `B` points but can change the data `B` points to. We can't change where `C` points to or the value it points to. Basically, "const" applies to whatever is on its immediate left, unless nothing is there in which case it applies to whatever is on its immediate right.

3.12 Dynamic Memory Allocation

Recall from C,

```
int size = ___;
int *p  = malloc(size*sizeof(int)); // allocate memory
...
free(p); // deallocate memory
```

Note that since C++ is a subset of C, both malloc and free are available, however it is not common practice to use them, instead use the type-aware and less error-prone functions new and delete.

```
struct Node { ... };
Node *np = new Node;
delete np;
```

Note that delete np only deletes the node that the pointer is pointing to, and nothing else. In an array, the behaviour of calling delete on it is undefined. To handle this, we call delete [] np. For example,

```
int *a = new int[5];
delete [] np; // deletes all elements
```

3.13 Passing Parameters

Example 3.8.

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

In C++, parameters are passed by value, inc increments a copy of x (not the original). If a functions needs to modify its original argument, pass a pointer.

```
void inc (int *n) { *n = *n + 1; }
int x = 5;
inc(&x);
cout << x << endl; // prints 6
```

Note that x's address is passed by value.
Question. Why do we need to use cin >> x and not cin >> (&x)?
Answer. C++ provides another pointer-like type called the **reference**.

3.14 References

References are like constant pointers with automatic dereferencing.

```
int y = 10;
int &z = y; // z is a reference to int, similar int *const z = &y
z = 12; // now y = 12
int *p = &z; // gives address of y
```

In all cases z behaves exactly like y. That is, z is an alias for y.

There are some things you cannot do with references.

- leave them uninitalized e.g., int &y; (bad)

- create a pointer to a reference e.g., int &*y; (bad) (but you can create a reference to a pointer, e.g., int *&y;)
- create a reference to a reference : e.g., int &&y; (bad)
- create an array of references : e.g., int & r[3] = {n,n,n} (bad)

However there are also some great things you can do!

- pass as function parameters, for example

```
void inc (int &n) { n = n + 1; }
int x = 5;
inc(x); \\ note no &
cout << x << endl; // prints 6
```

So, cin >> x works because it takes x by reference.

```
  istream& operator >> (istream &in, int &n)
```

Recall that pass-by-value copies the argument which implies that if the argument is big, this can be expensive. For example,

```
struct ReallyBig { ... }
int f (ReallyBig rb) { ... } // copies the whole struct
int g (ReallyBig &rb) { ... } // pass an, alias, more efficient but allows changes to propogate to caller
int h (const ReallyBig &rb) { ... } // argument can't be changed
```

It is prefered to pass-by-reference-to-const over pass-by-value for anything larger than int. Note also for

```
int f (int &n) { ... }
int g (const int &n) { ... }
f(5); // can't initialize a reference (n) to a literal value i.e., BAD
g(5); // OK since n can't be changed, allowed by compiler
```

3.15 Operator Overloading

We can give custom meanings for C++ operators for types we construct. For example, Vector.cc.

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

struct Vector {
  int x;
  int y;
};

Vector operator+(const Vector &v1, const Vector &v2) {
  Vector v;
  v.x = v1.x + v2.x;
  v.y = v1.y + v2.y;
  return v;
}

Vector operator*(const Vector &v1, const int k) {
  Vector v;
  v.x = k * v1.x;
  v.y = k * v1.y;
  return v;
}

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

int main () {
  Vector v1 = {1, 2};
  Vector v2 = {3, 4};
  Vector v3 = v1 + v2;
  Vector v4 = 2 * v1;
  Vector v5 = v2 * 3;

  cout << "v3.x = " << v3.x << "   v3.y = " << v3.y << endl;
  cout << "v4.x = " << v4.x << "   v4.y = " << v4.y << endl;
  cout << "v5.x = " << v5.x << "   v5.y = " << v5.y << endl;
}

// Returns
// v3.x = 4   v3.y = 6
// v4.x = 2   v4.y = 4
// v5.x = 9   v5.y = 12
```

also consider grades.cc

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

struct Grade {
  int theGrade;
};

ostream &operator<<(ostream &out, const Grade &g) {
  out << g.theGrade << "%";
  return out;
}

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

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

3.16 The Stack and the Heap

In memory there are typically three layers, the first is the **program**, then somewhere below that is the **heap** and finally at the end is the **stack**. The stack grows towards the heap. (Imagine a tower, at the top floors are the program, somewhere in the middle is the heap, and the bottom floor is the stack, the stack gets taller).

The heap is used for dynamic memory allocation, and data that is allocated to the heap lives on until it is explicitly deallocated (see new and delete). This means that the memory leaks are a danger of using the heap. The stack however is used for local variables within some scope, and may disappear when they go out of scope.

Example 3.9. Note that within some scope, a use of new allocates space on the stack **for the pointer** however the actual object that is created is on the heap. The obvious problem here is that once the pointer goes out of scope, there is still memory on the heap that has no been deallocated.

```
Node n; // on the stack
Node *np = new Node; // np (pointer) on stack, Node object on heap
```

Example 3.10.

```
Node getMeANode () {
  Node n;
  return n;
}
```

This example describes creating an object (note that Node n; does not initialize a null object, there are initial values and so it is an object), then returning that same object. This is expensive because n is copied as a return value, but the memory allocated on the stack for the original n is destroyed. So we've essentially done the same operation of creating n twice, which is a waste. Now consider

```
Node *getMeANode() {
  Node n;
  return &n; // unsafe: returns a pointer to stack-allocated data which is dead on return
}
```

A better way is as follows

```
Node *getMeANode () {
  Node *n = new Node;
  return n;
}
```

This is much better because it returns a pointer to heap allocated data.

Example 3.11. Allocating arrays:

```
cin >> n;
int *a = new int[n]; // a points to memory on the heap, containing n ints
for (int i = 0; i < n; ++i) {
  a[i] = i; // initialize values
}
delete [] a;
```

3.17 The Preprocessor

The preprocessor transforms the program before the compiler sees it. The preprocessor directive is the 'hash/pound' symbol #. Note that we've seen this already, because it is used for include statements, like

```
#include <iostream>
#include "file.h"
```

What this does is that it tells the preprocessor to get the contents of whatever is mentioned and insert the code in that position. Note that `<...>` means to look in the standard include directory (`/usr/include/c++`) and `"..."` means to look inside the current directory. Note also that there is a naming convention for old C headers. For example, instead of `#include <stdio.h>` we use `#include <cstdio>`.

There is also a define directive which looks like

```
#define VAR VALUE
```

this defines a preprocessor variable. All occurrences of `VAR` in the source file are replaced with `VALUE`. It is best applied as inline constants, for example

```
#define MAX 10
int x[MAX]
```

Note that defined constants are useful for **conditional compilation**. For example,

```
#define Unix 1
#define Win 2
#define OS Unix
#if OS == Unix
  int main () {
#elif OS == Win
  int WinMain () {
#endif
```

Note that

```
#if 0 // never true, all innter text is remove dbefore it gets to compiler
 ...
#endif // heavy duty way to comment out code
```

as well, `#if`'s nest. Another fact, you can also define symbols via compiler arguments. For example, in the file `define.cc`,

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

int main () {
  cout << X << endl;
}
```

Just a simple file, but if we compile it with

```
g++ -DX=15 define.cc -o define
```

and then run `./define`, we get the value 15 printed.

In this way we can also do things like

```
#define FLAG // sets variable FLAG (value is empty string)
#ifdef FLAG
#ifndef FLAG
```

where #ifdef and #ifndef are determined by whether or not FLAG has been set. For example, see debug.cc.

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

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

Where we run it with

```
g++ -DDEBUG debug.cc
```

3.18 Seperate Compilation

Split programs into seperate modules, which each prove an

- **interface** - type definitions, function prototypes - .h file
- **implementation** - full definition for every provided function - .cc file

Example 3.12. See seperate/example1

vector.h

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

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

vector.cc

```
#include "vector.h"

Vector operator+(const Vector &v1, const Vector &v2) {
  Vector v;
  v.x = v1.x + v2.x;
  v.y = v1.y + v2.y;
  return v;
}
```

main.cc

```
#include <iostream>
#include "vector.h"
```

```

using namespace std;

int main () {
  Vector v = {1,2};
  v = v + v;
  cout << v.x << " " << v.y << endl;
}
```

These can actually be compiled seperately, such as (in sequence from left to right)

```
g++ -c vector.cc     g++ -c main.cc     g++ vector.o main.o -o main     ./main
```

the third command links object files into an executable. Also note that `-c` means "compile-only" (don't link, don't build executable), it outputs an **object file** (`.o`).

Recall that an entity can be **declared** several times, but **defined** only once. What if we want to put a variable in a `.h` file? For example, abc.h

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

Now every file that includes `abc.h` **defines** a seperate `globalNum`. So the program will not link. The solution to this problem is that we can use the `extern` keyword in the `.h` file and make the definition in the C++ file. So in `abc.h` we have

abc.h

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

abc.cc

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

Example 3.13. Suppose we want to write a Linear Algebra module (see `seperate/example3`). Then within `LinAlg.h` we have include statements for "vector.h" and within `LinAlg.cc` we include "LinAlg.h" and "vector.h". This won't compile since there are 2 copies of `vector.h` thus there are two definitons for `struct Vector`.

We need to prevent the files from being included more than once. The solution is to use **include guards**. Thus in `vector.h` we have

```
#ifndef __VECTOR_H__
  #define __VECTOR_H__
  ...
#endif
```

So the first time `vector.h` is included, we have that `VECTOR H` is not defined, so the file is included, subsequently we have that `VECTOR H` is defined, thus the contents of `vector.h` are suppressed. Always put incldue guards in `.h` files.

Note. **never** put "`using namespace std`" in `.h` files, as it will be forced upon any user who includes the file.

3.19 Classes

The big innovation of Object Oriented Programming is that we can put functions inside structures. For example,

```
struct Student {
  int assn, midterm, final;
  float computeGrade () {
    return assn * 0.4 + midterm * 0.2 + final * 0.4;
  }
}

Student billy = {60,70,80};
cout << billy.computeGrade() << endl; // prints out its grade
```

A **class** is essentially a structure type that can contain functions. (C++ has a class keyword, but we'll start using it later)

An **object** is a particular instance of that class. For example, "Billy" is an object that is an instance of the "Student" class. The function computeGrade is called a **member function** (or a **method**). Also, `assn, midterm, final` are fields of the current object, the object upon which computeGrade is invoked. Formally, there is a hidden extra parameter called **this**, which is a pointer to the current object.

```
// e.g.,
billy.computeGrades(); // *this == billy
float computeGrades () {
  return this->assn * 0.4 + this->midterm *0.2 + this->final * 0.4;
} // recall: this->assn means (*this).assn
```

Initializing Objects

We can initialize objects in this way:

```
Student billy = {60, 70, 80}; // OK, but limited
```

However it is better to initialize using a method: **a constructor**.

```
struct Student {
    int assn, midterm, final;
    float computegrade() { ... }
    Struct (int assn, int midterm, int final) {
        this->assn = assn;
        this->midterm = midterm;
        this->final = final;
    }
}

// Now we can construct students like this:
Student billy(60, 70, 80);
// OR:
Student billy = Student(60, 70, 80);
// Heap allocation:
Student* pBilly = new Student(60, 70, 80);
```

The advantages of constructors are that we can use default parameter values, and overloading.

```
// e.g.
Student(int assn = 0, int midterm = 0, int final = 0) { ... }
Student bob(60, 70); // 60, 70, 0
Student newguy; // 0, 0, 0
// Alternatively: Student newguy = Student();
```

Note that every structure comes with a default constructor (no arguments), which just calls default constructors on any members that have them. For example,

```
Vector v; // default constructor
```

However this goes away as soon as you provide a constructor!

```
struct Vector {
    int x, y;
    Vector (int x, int y) { ... }
};

Vector v; // no longer valid!!
Vector v(1, 2); // okay
```

You also lose C-style structure initialization.

```
Vector v = {1, 2}; // invalid!!
```

What if a struct contains constants or references?

```
struct MyStruct {
    const int myConst = 5; // these need to be initialized "immediately"
    int z;
    int &myRef = z;
}; // won't compile!!
```

Besides, each instance of MyStruct each gets its own myConst and myRef - why should they all be the same? Also, we can't initialize them in the constructor - it's too late at that point.

When an object is created:

- Space for the object is allocated
- Members are initialized to defaults <– needs to put our initializations here
- Constructor is called

Member Initialization List

```
// in struct MyStruct
MyStruct (int c, int &r) : myConst(c), myRef(r) {}
// initializes before calling constructor!

Student(int assn, int midterm, int final): assn(assn), midterm(midterm), final(final) {}
// Can be more efficient than setting fields in body of constructor; otherwise they are ↩
    initialized by their default constructors and then reassigned in the body
```

Note that fields are initialized in the order they were declared, regardless of order in initialization list.
Copy Constructor, for constructing an object as a copy of another.

```
Stuent billy (60, 70, 80);
Student bobby = billy;
```

Every class comes with:

- default constructor (all fields to default), which are lost if you define your own constructor
- copy constructor (blindy copies all fields)
- copy assignment operator
- destructor

Building your own copy constructor:

```
Student (const Student& other) : assn(other.assn), midterm(other.mt), final(other.final){}
// this is equivalent to built-in copy constructor)
```

Consider:

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

// and say we want a node linked list 1 -> 2 -> 3
Node *n = new Node(1, new Node(2, new Node(3, 0)));
Node m = *n; // uses copy constructor
Node *p = new Node (*n); // uses copy constructor
```

Note that the above are only shallow copies - they only copy out the first node! If you need a deep copy (the entire list), write your own copy constructor:

```
Node (const Node &other) : data(other.data),
                           next(other.next ? new Node(*other.next) : 0) {} // recursively copies rest of list
```

When is the copy constructor called?

1. When an object is initialized with a copy of another
2. When an object is passed by value
3. When an object is returned by a function

A deep copy file would look like

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

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

  Node(const Node &n): data(n.data),
                       next(n.next == NULL ? NULL : new Node(*n.next)) {}
};

ostream &operator<<(ostream &out, const Node &n) {
  out << n.data;
  if (n.next) {
    out << ",";
    out << *n.next;
  }
  return out;
}

int main() {
  Node *n = new Node(1, new Node(2, new Node(3, NULL)));

  Node m = *n;
  m.data = 5;

  Node *p = new Node (*n);
  p->data = 6;

  cout << "n: " << *n << endl;
  cout << "m: " << m << endl;
  cout << "p: " << *p << endl;
  cout << endl;

  n->next->next->data = 7;

  cout << "n: " << *n << endl;
  cout << "m: " << m << endl;
  cout << "p: " << *p << endl;
}
// returns
/*
n: 1,2,3
m: 5,2,3
p: 6,2,3

n: 1,2,7
m: 5,2,3
p: 6,2,3
*/
```

Note. Be careful of constructors that take one parameter! For example,

```
  struct Node {
    ...
    Node(int data) : data(data), next(0) {}
  };
```

Single-arg constructors create implicit conversions. For example, Node `n(4);`, but also Node `n = 4;`, there is an implicit conversion from 4 to `Node(4)`, and also `int f(Node n) {...}`, then `f(4);`. That works, the 4 is converted to `Node(4)`. This can be dangerous, for example accidentally pass an int to a function expecting a node, and you get a silent conversion, the compiler does not signal an error, and therefore potential errors are not caught. It is a good idea to disable the implicit conversion. To do this, declare the constructor explicit.

```
  struct Node {
    ...
    explicit Node (int d) : data (d), next(0) { }
  };
```

Destructors

When an object is destroyed (stack-allocated means goes out of scope, heap-allocated means is deleted), a method called the **destructor** runs. Classes come with a destructor (doesn't do much - calls destructor for each field). Wehen do we need to write one? Suppose we have

```
Node *np = new Node (1, new Node(2, new Node(3, 0)));
// np -> 1 -> 2 -> 3 /
```

If np goes out of scope, the pointer np (stack-allocated) is reclaimed, the list is not. If we simply say, `delete np;`, we delete the first node, but the destructor running does not delete the rest of the list so we have a memory leak. We need to write destructor.

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

Even better, since deleting a null does nothing (but works safely), we can just say

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

Assignment Operator
Brad Lushman's story

```
Student tony (60,70,80);
Student liam = tony; // I just cheat on everything
Student devin; // default constructor
devin = tony; // copy assignment operator, not constructor
// assignment operator uses compiler, supplied default
```

We may need to write our own copy assigment operator.

```
struct Node {
  ...
  Node &operator=(const Node &other) {
     data = other.data;
     delete next;
     next = other.next ? new Node (*other.next) : 0;
     return *this;
  }
};
```

But this is very dangerous. Why? Consider

```
Node n(1, new Node(2, new Node(3, 0)));
n = n;
```

So we need to check for this case. In fact, ALWAYS check for this case when writing assignment operators.

```
struct Node {
  ...
  Node &operator=(const Node &other) {
    if (this == &other) return *this;
    data = other.data;
    delete next;
    next = other.next ? new Node (*other.next) : 0;
    return *this;
  }
};
```

Even better than this, we can do

```
  Node &operator=(const Node &other) {
    if (this == &other) return *this;
    data = other.data;
    Node *tmp = next;
    next = other.next ? new Node (*other.next) : 0;
    delete tmp;
    return *this;
  }
```

There is infact, **yet another better way** to accomplish this. It is the alternative copy-and-swap idiom.

```
struct Node {
  ...
  void swap (Node &other) {
    int tdata = data;
    data = other.data;
    other.data = tdata;
    Node *tnext = next;
    next = other.next;
    other.next = tnext;
  }

  Node &operator=(const Node &other) {
    Node tmp = other;
    swap(tmp);
    return *this;
  }
}
```

And, equivalently,

```
Node &operator=(Node other) {
  swap(other);
  return *this;
}
```

It works by using the copy-constructor's functionality to create a local copy of the data, then takes the copied data with a swap function, swapping the old data with the new data. The temporary copy then destructs, taking the old data with it. We are left with a copy of the new data.
In order to use the copy-and-swap idiom, we need three things: a working copy-constructor, a working destructor (both are the basis of any wrapper, so should be complete anyway), and a swap function.

Certain themes have arised from this past discussion, and we call them the Rule of 3.

Definition 3.1 (rule of 3)**.** If you need to write a custom version of any one of

- copy constructor
- operator equals
- destructor

Then you usually need a custom version of all three.

Notice that operator= is a member function, not a standalone function. When an operator is declared as a member function, this plays the role of the LHS assignment. So,

```
struct Vector {
  int x, y;
  Vector operator+(const Vector &v) {
    Vector v2(x + v.x, y + v.y);
    return v2;
  }
  Vector operator*(const int k) {
    return Vector(x*k, y*k);
  }
}
```

Note that the multiplication implements the order v*k. How do we implement k*v? The first argument is not a vector, it can't be a member function, so it must be a standalone.

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

What about I/O operators?

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

However there is a problem with this implementation which is that it makes Vector the LHS operand and not the RHS operand, so we'd need to use v << cout; which is a little confusing. The point is that input and output operators should really be standalones as well. Note that, **certain operator must be members**. That is, you have no choice. They are:

- operator=
- operator[]
- operator->
- operator()
- operatorT()

The reason that operator= must be a member is because it has a default implementation and so what you could end up doing is write your class without operator equal and then write a function using that class that assigns something using operator= before its definition.

3.20 Arrays of Objects

Consider this code.

```
struct Vector {
  int x, y;
  Vector (int x, int y): x(x), y(y) {}
};
Vector *vp = new Vector[10];
Vector moreVectors[15];
```

This does not compile. Why? Because it can't initialize the array elements. You can only create arrays of objects that have a default (i.e., zero-ary) constructor. If you want arrays, provide a default constructor. To fix this, add default values.

```
struct Vector {
  int x, y;
  Vector (int x=0, int y=0): x(x), y(y) {}
};
Vector *vp = new Vector[10];
Vector moreVectors[15];
```

3.21 Seperate Compilation for Classes

Consider **Node.h**

```
#ifndef __NODE_H__
#define __NODE_H__
struct Node {
  int data;
  Node *next;
  Node(int data, Node *next);
  Node(const Node &n);
  explicit Node (int d);
};
#endif
```

Then, in **Node.cc**

```
#include "Node.h"
Node::Node(int data, Node *next): data(data), next(next) {}
Node::Node(const Node &n):data(n.data), next(n.next == NULL ? NULL : new Node(*n.next)) {}
Node::Node(int d): data(d), next(0) {}
```

Let's finally define what the hell this `::` thing is. It is called the **scope resolution operator**. Essentially it means (in this example) that `Node::x` means `x` in the context of the Node class. Usually it is similar to just using ., but the difference is that for `::`, where LHS is a class, not an object. Basically,

- Is LHS an object? Use .
- Is LHS a class? Use ::

3.22 Consts Again

Constants come up much more often in C++. Things likes

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

What is a const object? Essentially it is an object whose fields can't be changed. Can we call methods on const objects? The issue is that the method might modify fields in the process, violating the const.

The answer is a qualified yes, we can call methods that promise not to modify fields.

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

By putting const after the function name, it says "does not modify fields". The compiler checks that const methods don't modify fields. Only const methods can be called on const objects. Now consider, what if we want to collect some usage statistics on Student objects.

```
struct Student {
  int assns, mt, final;
  int numMethodCalls; // increment this counter per method call
  float grade() const {
    ++numMethodCalls; // fails, changing something in a const method
    return ...;
  }
}
```

Since this fails, the result would be that we have to remove several consts from our code just to check statistics. This is called const poisoning. Our problem here is that `numMethodCalls` isn't really a defining characteristic of a `Student`. We want to be able to update `numMethodCalls` even if the object is const. The solution is to declare the field mutable. So,

```
struct Student {
  int assns, mt, final;
  mutable int numMethodCalls; // increment this counter per method call
  float grade() const {
    ++numMethodCalls; // works, mutable
    return ...;
  }
}
```

Mutable fields can be changed even if the object is const.

3.23 SE Topic - Design Patterns

Experience shows that certain programming scenarios arise frequently. If you've got a problem to solve, there's a good chance someone has had the same problem before. The idea is to keep track of solutions to these problems, and use them in similar situations. It is reccomended that you read the book Design Patterns: Elements of Reusable Object-Oriented Software by Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. There are some examples, such as

Definition 3.2 (Singleton Pattern)**.** We have a class C, and we want to ensure that only one instance of C ever gets created, no matter how many times we may attempt to create an instance. An example of this is a database class. If we have a class that is a frontend to a database, and it is used several times, we want to ensure that there is only ever one instance of that database class, so the same database is always being accessed. Similarly for a log file. These are under the assumption that there are only one of these.

Example 3.14. Write a program to track my finances. We will use two classes

- Wallet (this is a singleton - I have only one)
- Expense (several of these - each have access to my wallet (sadly))

For this example we require a new C++ concept, the **static member**.

Definition 3.3 (static member)**. Static members** are associated with the class itself, not with any specific instance of a class.

Example 3.15. For example, how many times as a Student object been created? We'll return to the finance example in a second, first we define a few things.

```
struct Student {
  ...
  static int numInstances; //  shared by all Student instances
  Student (...) : ... { // constructor calls static member
    ++numInstances;
  }
};
```

Appreciate the fact that the **static** keyword within the struct Student means that **every instance of a Student** uses the exact same variable, which means every time a Student is constructed, numInstances is updated.

However we are not done! There is an important line that must be written in the .cc file,

```
int Student::numInstances = 0; // initialization
```

This must be done for every static variable; if not the program won't link. So, if we run

```
int main () {
  Student liam(49,49,49); // almost there
  Student umar(100,100,100); // 100%
  cout << Student::numInstances << endl; // access static using scope operator
}
```

We return 2.

Definition 3.4 (static member function)**.** Next, we look at **static member functions**. These are functions that don't depend on any specific instance (therefore there is no "this" pointer). Therefore within a static member function we can access static member, since they don't depend on any instance (also can call other static methods).

Example 3.16. Consider the structure definition with this new static member function,

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

then if in main we have

```
Student liam(49,49,47);
Student greg(94,94,94);
Student::printNumInstances(); // returns 2
```

Back to the finance example, let's write Wallet.h,

```
struct Wallet {
  static Wallet *instance; // only one instance
  static Wallet *getInstance(); // fetch the instance, initialize if necessary
  Wallet();
  int money;
  void addMoney(int amt);

}
```

Then in Wallet.cc

```
Wallet *Wallet::instance = 0; // first thing, define static member, starts at NULL
Wallet *Wallet::getInstance() {
  if (!instance) { // if instance hasnt been created, then create it
    instance = new Wallet;

  }
  return instance;
}
Wallet::Wallet() : money(0) {}
void Wallet::addMoney(int amt) {
  money += amt;
}
```

Then we have a header file Expense.h, whch each has access to a wallet instance (the same one)

```
struct Expense {
  const std::string desc;
  const int amt;
  Wallet *wallet;
  Expense (std::string desc, int amt);
  void pay();
};
```

and it companion .cc file,

```
Expense::Expense (string desc, int amt) : desc(desc), amt(amt) {
  wallet = Wallet::getInstance();
}
void Expense::pay() {
  cout << "Paying << desc << "("" << amt << ")" << endl;
  wallet->addMoney(-amt);
}
```

Finally, the main C++ file,

```
int main() {
  Expense mortgage("mortgage", 1000);
  Expense car("car", 300);
  Expense ins("insurance", 200);
  Wallet *mywallet = Wallet::getInstance();
  Expense payCheque("paycheque", -2000);
  cout << "Initial Money: " << mywallet->money << endl;
  payCheque.pay();
  mortgage.pay();
  car.pay();
  ins.pay();
}
```

Running this code will get us

```
Initial money: 0
Paying expense: paycheque (-2000)
Paying expense: mortgage (1000)
Paying expense: car (300)
Paying expense: insurance (200)
Final money: 500
```

Great. Now however we have a question, when do we delete the Wallet instance? How can we know when all clients are done with it? Well we can't. Let's write some cleanup function,

.h

```
struct Wallet {
  ...
  static void cleanup(); // must be static
}
```

.cc

```
void Wallet::cleanup() {
  cout << "Cleaning up...” << endl;
  delete instance;
}
```

If we were to run this at the end of main, we'd handle 99% of all cases. However we can be better than that. There is a function called `atexit` (from `<cstdlib>`) that takes a function returning void, and runs it when the program terminates. So in our implementation for `getInstance()`,

```
wallet *Wallet::getInstance() {
  if (!instance) {
    instance = new Wallet;
    atexit(cleanup);
  }
  return instance;
}
```

Okay, so can't we just create our own Wallet instances by calling the constructor? The time has therefore come for us to use the big terms of OOP. For now, that is **encapsulation**. The idea is that we want to control the way our objects are used, we want our clients to treat objects as black boxes (capsules). We want implementation details sealed away, and we want it so you can only interact via provided methods.

```
struct Vector {
  Vector (int x, int y); // public (say nothing implies public)
 private: // can't be accessed outside the struct
  int x, y;
 public: // anyone can access
  Vector operator+(const Vector &v);
};
```

The default visibility in structs is public. In general, you want fields to be private and methods to be public. One might desire for the default to be private, however the problem with this is that all C program will compile (since struct is defaulted to public in C). So instead, we invent something that is not in C, called **class**. So, class vector would look like this:

```
class Vector {
  int x, y; // default private
 public:
  Vector(int x, int y);
  Vector operator+(const Vector &v);
  ...
}
```

The only difference between class and struct is default visibility. It is public in struct, and private in class. Once again, keep fields private. If you have public fields

- you have no way of preventing the user from doing anything they want with them
- you can't maintain class invariants
- can't replace implementations without breaking client code

If you want to provide field access, then provide methods that do just that; that is, write an **accessor method**. For example,

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

If you want to let clients change fields as well, provide what are known as **mutator methods**. For example,

```
class Vector {
  int x, y;
 public:
  ...
  void setX(int newX) { x = newX; }
  void setY(int newY) { y = newY; }
}
```

Returning to our singleton example with the wallet, we can rewrite our Wallet struct as a class like this, which implements the single Wallet instance as private, and has a public instance to get the instance, which means no one can change the single instance after it is initialized the first time.

```
class Wallet {
  static Wallet *instance;

  Wallet();

  int money;
  static void cleanup();

 public:
  static Wallet *getInstance();
  int getMoney() const;
  void addMoney(int amt);

};
```

Now, suppose we don't want to provide accessors and mutators, but we do want to provide operator<<. There is an issue, operator<< needs to get x and y but we don't want to provide general access to everyone. THe solution is to make operator<< a **friend** function, for example

```
class Vector {
  int x, y;
 public:
  ...
  friend std::ostream &operator<<(std::ostream &out, const Vector &v);
}
```

Then in the C++ file,

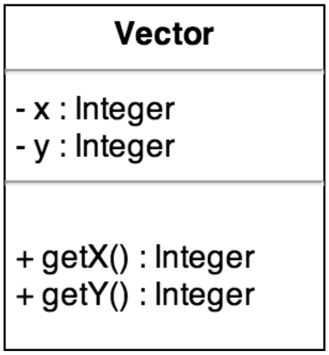
```
...
ostream &operator<<(ostream &out, const Vector &v) {
  return out << v.x << " " << v.y;
}
```

friend functions can see all of a classes members but are not themselved part of the class. Give your class as few friends as possible. When you declare friends, it weakens the encapsulation.

3.24 SE Topic - System Modelling

Building an Object-Oriented system involved identifying the major abstractions (what do I want my classes to be?) and then formalizing the relationships among them. Once I've decided what my classes ought to be, how are they related to each other and how to they interact? It has been proven helpful to map these relationships out, to aid in design and implementation.

A popular standard is called UML (Unified Modelling Language).



Vector

- x : Integer
- y : Integer

+ getX() : Integer
+ getY() : Integer

where '-' represents private, and '+' represents public. In UML, the fields and methods blocks are of course optional but you need the name of the class.

3.25 Composition of Classes

```
class Vector {
  int x, y, z;
 public:
  Vector(int x, int y, int z) : x(x), y(y), z(z) {}
  ...
};
```

Two vectors define a plane.

```
class Plane {
  Vector v1, v2;
```

```
 public:
  ...
};
Plane p;
```

This does not compile however since it cannot initialize v1 and v2. Now, note that, when an object is created:

1. Space is allocated
2. Default constructors / initialization lists for all fields in declaration order
3. Constructor body runs

and when an object is destroyed,

1. Destructor body runs
2. Destructors are invoked for all fields in reverse order
3. Space is deallocated

So for the case of `Plane p;`, the field constructors must be called for `v1`, `v2`, but `Vector` has no default constructor.

Solution 1 : Give Vector a default constructor. What if we don't want to?

Solution 2 : Initialize v1, v2 in Plane's initialization list.

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

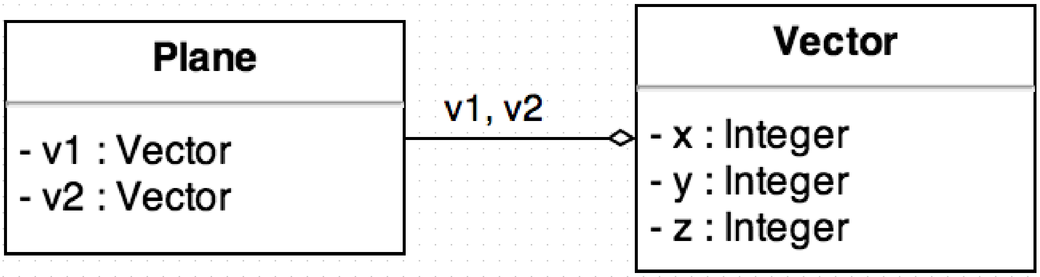
Embedding one object (Vector) inside another (Plane) is called **composition**. Relationships between Plane and Vector is called an "owns-a" relationship. A Plane object owns a vector object (in fact, it owns two of them).

Here are the typical characteristics of an "owns-a" relationship: If A owns a B, then typically

- B has no identity outside A (does not have an independent existence).
- If A is destroyed, then B is destroyed
- If A is copied, then B is copied (deep copy)

Example 3.17. If a car owns four wheels, a wheel is part of a car. Destroy the car implies destroy the wheels. Copy the car implies copy the wheels (don't share wheels).

Modelling composition in UML:



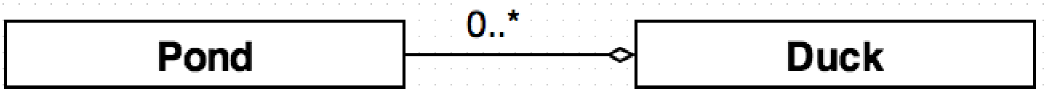
A (diamond) -> B means A owns some part of B. You can annotate with multiplicities and field names. 0..* means any number of B's, 2 = 2Bs, 1...5 = 1-5 Bs.

3.26 Aggregation

Compare car parts in a car ("owns-a") versus car parts in a catalogue. The catalogue contains the parts but the parts have an independent existence. This is a "has-a" relationship ("aggregation"). These typically have

- If A has a B, then B has an existence apart from its association with A
- If A is destroyed then B lives on
- If A is copied, B is not (shallow copy), copies of A share the same B

Example 3.18. Parts in a catalogue is an example. Ducks in a pond is another example, if you're copying a pond you're probably not copying the ducks. In UML this looks like:



Typical implementation looks like

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

3.27 Inheritance

Suppose you want to track your collection of books. Then we have

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

But for CS books, we want to know what programming language it's about.

```
class CSBook {
  string title, author;
  int numPages;
  string languages;
 public:
  CSBook(...)...;
  ...
};
```

For comic books, we want to know who the hero is.

```cpp
class ComicBook {
  string title, author;
  int numPages;
  string hero;
 public:
  Hero(...){...};
  ...
};
```

Creating these individual classes doesn't capture the relationship between books, CS books, and comic books. Additionally how would we create an array with a mix of these classes. Options include

- Use a union
  union bookTypes Book * b, CSBook * csb, ComicBook * cb;
  bookTypes myBooks[20];
- An array of void * that points to Books, CSBooks, and ComicBooks.

There are no good solutions when we are trying to suvert the type system. CS books and comic books are *kinds of books* with particualr additional features. To model this in c++ we use inheritance.

Base Class / Super Class

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

Derived classes or sub classes

```cpp
class CSBook : public Book {
  string language;
 public:
  CSBook(...)...;
  ...
};
```

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

Derived classes inherit fields and methods from the base class. Both CSBook and ComicBook get title, author, numPages fieds. Any mehtod chat can be called on Book can be called on CSBook and ComicBook.
Who can see these members? title, author, and numPages are private in Book, therefore outsiders cannot see them and nor can the subclasses. They do exist in the subclass; however, they cannot be accessed.

How do we initialize a CSBook? We need to initialize the $\overbrace{\texttt{title, author, numPages}}^{\texttt{book part}}$ and language part. Why will the following fail
CSBook(string title, string author, int numPages, string language) : title(title), author(author), numPages(numPages), language(language) {};

1. Title, Author, and numPages are not accessable in CSBook
2. Book has no default constructor to construct superclass part

Remark 3.1. Recall: When an object is constructed

1. Space is allocated
2. Superclass part is constructed
3. Default ctors or init list for fields
4. Ctor body runs

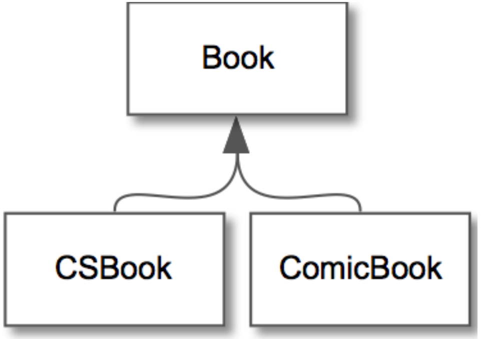
To solve both problems invoke the book constructor in the initialization list.
`CSBook(string title, string author, int numPages, string language) : Book(title, author, numPages), language(language) {};`
If the superclass nas no default constructor you must explicity invoke a non-default constructor for the superclass in the subclass initialization list.
There are good reasons for keeping superclass fields from subclasses. If you want to give a subclass access to certain members use `protected:` visibility. Not a good idea to give subclasses access to fields, it is better to make the fields private but provide protected accesors.

```cpp
class Book {
  string title, author;
  int numPages;
 protected:
  string getTitle() const;
  string getAuthor() const;
  string setAuthor(string name);
 public:
  Book(...)...;
  ...
};
```

The relationship among Book, CSBook, and ComicBook is called a "is-a" relationship.

1. a CSBook *is a* Book
2. a ComicBook *is a* Book

UML:



Now consider a method isItHeavy for books which is true

1. for ordinary books >200 pages
2. for ComicBooks > 30 pages
3. for CSBooks >500 pages

```
class Book {
    ...
  protected:
    ...
  public:
    bool isItHeavy(){return numPages > 200;};
    ...
};

class ComicBook {
    ...
  protected:
    ...
  public:
    bool isItHeavy(){return numPages > 30;};
    ...
};
```

Consider:

```
  Book b("Is small book", "A small man", 50);
  ComicBook cb("A big comic", "A different author", 40, "Superman");
  cout << b.isItHeavy()  // false
       << cb.isItHeavy() // true
```

Because inherriteance is an "is-a" relationship we can do this:
`Book b = ComicBook(... , ... , 40, ...);`

Question: What is the result of `b.isItHeavy`?

Answer: `Book::isItHeavy` executes and returns false.
`Book b = ComicBook(...)` tries to fit a comic book object where there is only space for a book object.

1. ComicBook is sliced (the hero field is cut off)
2. ComicBook is condensed into a book and its `Book::isItHeavy` runs.

When accessing objects through pointers shaving is unnecessary and **does not happen**.

```
  ComicBook cb("A big comic", "A different author", 40, "Superman");
  Book * pb = &cb;
  ComicBook * pcb = &cb
  cout << pcb->isItHeavy()  // true,  because ComicBook::isItHeavy runs and 40 > 30.
       << pb->isItHeavy()   // false, because Book::isItHeavy runs and 40 < 200.
```

The compiler uses the type of the pointer or reference to determine which `isItHeavy` to run and does not consider the actual type. This means that ComicBook is only a ComicBook when a Comic pointer or reference points to it. This is probably not what we want.
So we've seen that ComicBook only behaves as a ComicBook when pointed to by a ComicBook pointer (or reference).
Question: How do we make a ComicBook act like one when pointed to by a Book pointer? (that is, how do we ensure

ComicBook::isItHeavy() runs pB->isItHeavy()?)
Answer: Declare the method **virtual**. Virtual methods choose which class method to run based on actual type of the object at runtime.

```
class Book {
  ...
  virtual bool isItHeavy() { return numPages > 200; }
};
class ComicBook : public Book {
  bool isItHeavy() { return numPages > 30; }
};
```

Then the following is how we can use this result,

```
ComicBook cb (___,__,40,__);
Book *pb = &cb;
Book &rb = cb;
cout << pb->isItHeavy() // true
     << rb.isItHeavy() // true
     << endl;
```

(ComicBook::isItHeavy runs).

Now we can have a mixed book collection:

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

where isItHeavy uses Book::isItHeavy() for Books and ComicBook::isItHeavy() for ComicBooks, etc. This accomodates multiple types under one abstraction, and is known as **polymorphism**.

Note. This is why a function void f(istream &in) can be passed an ifstream; that is, an ifstream "is an" istream.

DANGER! Consider,

```
class One {
  int x;
 public:
  One (int x = 0) : x(x) {}
};
class Two : public One {
  int y;
 public:
  Two(int x = 0, int y = 0) : One(x), y(y) {}
};

One myArray[2];
myArray[0] = Two(1,2);
myArray[1] = One(3);
```

What happens is that this only allocates space for two instances of One, which in memory are two blocks like [ ][ ] and myArray[0] would allocate too much space, because it contains two ints, whereas the base only contains one, and myArray is only allocated with enough space for multiple One objects (just one int). So we end up in memory with [1][3]. The "2" gets overwritten. Moral of the story, **never** use arrays of objects polymorphically.

If you want a polymorphic array, use pointers. Like,

```
One *myArray[2]; // okay :)
```

3.28 Destructor Revisited

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

X *myX = new Y(10, 20);
delete myX; // leaks memory - why?
```

Only x, (not y) is freed. it called ∼`X()`, but not ∼`Y()`. How can we ensure that deletion through a pointer to the superclass won't leak memory? We declare the destructor method **virtual**. Always make the destructor virtual if the class is meant to have a subclass (even if it doesn't do anything). See some examples in `/lectures/c++/inheritance/examples[1-5]`.

3.29 Tools: The make Utility

Recall seperate compilation, for example g++ `book.cc`, g++ `-c csbook.cc`, g++ `-c main.cc`, and finally g++ `book.o csbook.o main.o -o main`. This was our workflow for making builds. We do this because if for example we have a big project and we only make one little change, we don't need to rebuild the entire thing all over again.

How do we keep track of what has changed and what hasn't? We use the make utility. It creates a Makefile that outlines dependencies among components: (note this is a file, but I'm showing it here in a weird format, pointing out some things)

main: $\underbrace{\text{main.o book.o csbook.o}}_{\text{main depends on these}}$

$\underbrace{\qquad}_{\text{tab}}\underbrace{\text{g++ main.o book.o csbook.o -o main}}_{\text{how to build main from these}}$

csbook.o: csbook.cc csbook.h book.h

g++ -c csbook.cc

book.o: book.h book.cc

g++ -c book.cc

main.o: main.cc book.cc csbook.cc book.h csbook.h

g++ -c book.cc

Then from the command line, type in make and it will build the whole project. Now suppose you change `book.cc`. What happens?

```
$ make
  g++ -c book.cc
  g++ main.o book.o csbook.o -o main
```

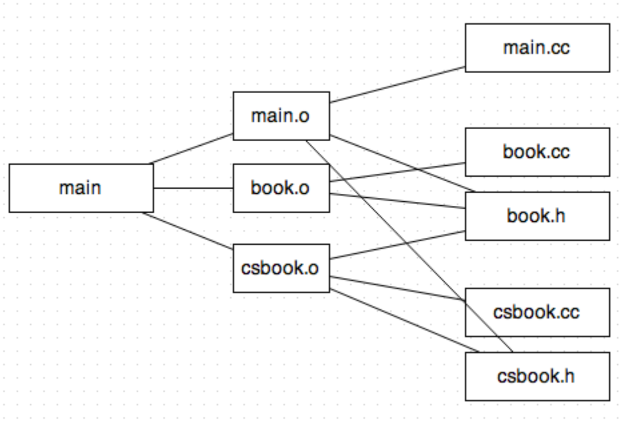
An actual file would look like this

```
main: main.o book.o csbook.o comicbook.o
        g++ main.o book.o csbook.o comicbook.o -o main

main.o: main.cc book.h csbook.h comicbook.h
        g++ -c main.cc

book.o: book.h book.cc
        g++ -c book.cc

csbook.o: csbook.h csbook.cc book.h
        g++ -c csbook.cc

comicbook.o: comicbook.h comicbook.cc book.h
        g++ -c comicbook.cc
```

It only compiles book.cc. In general, make

- builds the first target in Makefile (main)
- recursively builds the things main depends on (book.o, csbook.o, main.o)

(Dependency graph)



So book.cc changes are that book.cc is now newer than book.o (which it tells by a timestampt) and then rebuilds book.o. Book.o is now newer than main, so we rebuild main. We can also rebuild specific targets, like $ make csbook.o. It is common practice to put a target "clean" at the bottom to remove all binaries, which looks like

```
clean:
        rm *.o main
```

Then to do a full rebuild, run, make clean, followed by make. Our new file,

```
A     c++/classes/inheritance
A     c++/classes/inheritance/books.cc
main: main.o book.o csbook.o comicbook.o
        g++ main.o book.o csbook.o comicbook.o -o main

main.o: main.cc book.h csbook.h comicbook.h
        g++ -c main.cc

book.o: book.h book.cc
        g++ -c book.cc
```

```

csbook.o: csbook.h csbook.cc book.h
        g++ -c csbook.cc

comicbook.o: comicbook.h comicbook.cc book.h
        g++ -c comicbook.cc

.PHONY: clean

clean:
        rm *.o main
```

We can generalize this with variables,

CXX = g++ (compiler's name)
CXXFLAGS = -Wall (compilers options) (-Wall turns on all warnings)

For example, we could write

```
book.o : book.h book.cc
         ${CXX} ${CXXFLAGS} -c book.cc
```

Shortcut. For any rule of the form `X.o : x.cc x.h b.h ..`, we can omit the build command and make will assume that it is of the form

```
${CXX} ${CXXFLAGS} -c x.cc -o x.o
```

Example 4 does this, as shown below:

```
CXX = g++
CXXFLAGS = -Wall
EXEC = main
OBJECTS = main.o book.o csbook.o comicbook.o

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

main.o: main.cc book.h csbook.h comicbook.h

book.o: book.h book.cc

csbook.o: csbook.h csbook.cc book.h

comicbook.o: comicbook.h comicbook.cc book.h

.PHONY: clean

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

The biggest challenge with makefiles is tracking and maintaining dependencies. We can get help from g++: g++ `-MMD -c` csbook`.`c will generate csbook`.`d which basically is a file that contains the rule that you would use in your makefile. Now just include this in the Makefile using -include, like in this case:

```
CXX = g++
CXXFLAGS = -Wall -MMD
EXEC = main
```

```
OBJECTS = main.o book.o csbook.o comicbook.o
DEPENDS = ${OBJECTS:.o=.d}

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

-include ${DEPENDS}

.PHONY: clean

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

3.30 Pure Virtual Methods and Abstract Classes

```
class Student {
 protected:
  int num Courses;
 public:
  virtual int fees();
};
```

For example there are 2 kinds of Students, Co-op and Regular.

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

What should we put for Student::fees()? Not sure, every Student should be either regular or co-op. We can explicitly give Student::fees NO implementation:

```
virtual int fees() = 0; // no implementation
```

This is a **pure virtual method**. A class with a pure virtual method cannot be instantiated.

```
Student s; // error
```

called an **abstract class**. The purpose of this is to organise subclasses.

Subclasses of an abstract class are also abstract unless they implement all pure virtual methods. Classes that can be instantiated (that is, no pure virtual methods) are called **concrete classes**.

```
class Regular : public Student {
 public:
  int fees { return 700 * numCourses; }
};
```

In UML, pure virtual methods and abstract classes are identified using italics.

3.31 Inheritance and the Copy Constructor, operator=

```
class Book {
  ...
 public:
  Book(const Book &other) : title(other.title), ... {}
  ...
};
class CSBook : public Book {
  ...
 public:
  // no copy constructor defined
};

CSBook b("Algorithms", "CLRS", 500, "C");
CSBook c = b; // copy constructor, okay b/c blindly copies all fields
```

To write your own copy constructor:

```
CSBook::CSBook(const CSBook &other) : Book(other), language(other language) {}
```

calls the Book copy constructor. The assignment operator works similarly,

```
CSBook c;
c = b;
```

By default, calls Book::operator= and then goes field-for-field for CSBook part. To write your own:

```
CSBook &operator=(const CSBook &other) {
  Book::operator = (other);
  language = other.language;
  return *this;
}
```

Now consider the following situation, we'll create two instances of CSBook,

```
CSBook csb1(...), csb2(...);
Book *pb1 = &csb1;
Book *pb2 = &csb2;
```

What happens if we try to assign the value that the first pointer points to to the value that the second pointer points to? (i.e., *pb1 = *pb2). The answer is that Book::operator= runs. This is a problem because the Book methods assignment operator only knows about itself, so it can only copy the Book part, not any of the CSBook features. So what we get is **partial assigment** (copies only the Book part).

How can we fix this? We can try making the assignment operator virtual.

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

Note. To override the method, parameter types must match. However, different return types allowed here.

But this would allow assignment of a Book object to a CSBook variable.

```
CSBook csb(...);
CSBook *pcsb = &csb;
Book b(...);
Book *pb = &b;
*pcsb = *pb; // BAD (compiles though)
```

Also,

```
ComicBook cb(...);
ComicBook *pcb = &cb;
*pcsb = *pcb; // REALLY BAD
```

Quick recap:

- If operator= is **nonvirtual**, we can get partial assignments.
- If operator= is **virtual**, compiler allows mixed assignments

Reccomendation: make superclasses **abstract**, rewrite the Book hierarchy as follows, AbstractBook as the parent of RegularBook, CSBook, and ComicBook. Then

- make operator= **protected** in AbstractBook to prevent assignment through base class pointers, but make the implementation available to subclasses
- Need at least one pure virtual method, use the destructor

Note. Even though it is purely virtual, the destructor must be implemented (but can be empty)

```
class AbstractBook {
  string title, author;
  int numPages;
 protected:
  AbstractBook &operator=(const AbstractBook &other);
 public:
  AbstractBook(...);
  virtual ~AbstractBook() = 0;
};
AbstractBook::~AbstractBook() {}
```

```
class RegularBook : public AbstractBook {
 public:
  RegularBook(...);
  ~RegularBook();
  RegularBook &operator=(const RegularBook &other) {
    AbstractBook::operator=(other); // since it is protected, not private
    return *this;
  }
};
// etc. This prevents both partial assignments and mixed assignments
```

Note that every class has a destructor, either built-in or user-defined that counts as overriding the pure virtual destructor.

UML Relationships

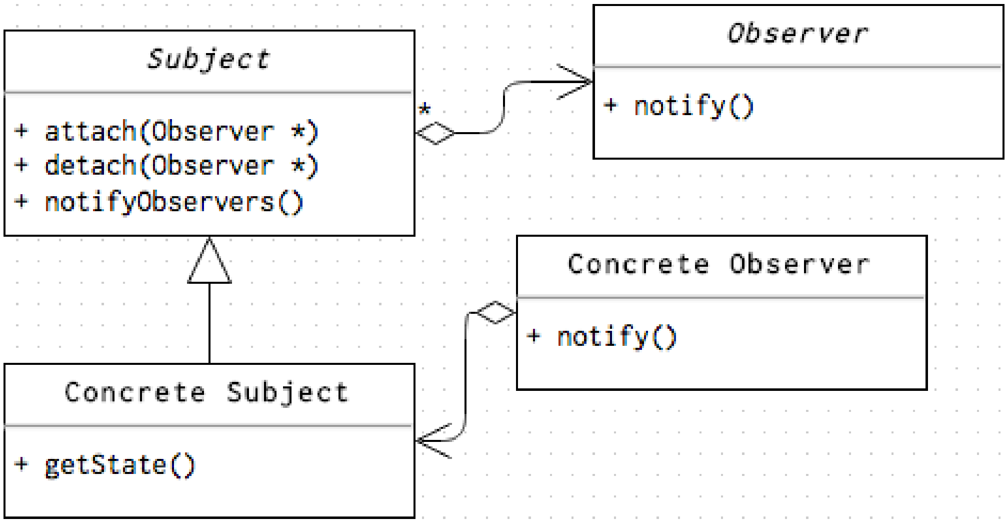
- ⟶ means association
- A line with a hollow diamond on the left and an arrow on the right denotes aggregation. The left side element is the containing class, the right hand side is the contained class.
- A line with a solid diamond on the left and an arrow on the right denotes composition. The left side element is the containing class, the right hand side is the contained class.
- A # represents a protected member.

3.32 Observer Pattern

This is also known as the Public-subscribe model

- In 1 class, there is a subject / publisher idea where a subject is observable and generates data and it is the publisher that does the observing
- In multiple classes, there is an observer / subscriber idea where the observer observes data from the subscriber and responds to it

Example 3.19. For example, a subject could be spreadsheet cells, and the observers could be graphs (based on data in the cells). When cells change, the graphs update. Subject need not know details of the observer.



(sorry again for the terrible quality, if anyone has a better way to draw these on the fly let me know)

Control Flow:

1. Observers are attached to subject (`Subject::attach(Observer *)`)
2. Subjects state is updated
3. Subject notifies each observer (`Subject::notifyObservers()`)
4. Each observer queries the state of the subject and responds (`ConcretSubject::getState()`)

Code:

```
class Subject {
  Observer *observers[maxObservers];
  int numObservers;
 public:
  Subject();
  bool attach(Observer *o) // adds to obeservers (return true if successful)
  bool detach(Observer *o) // remove o from observers
  void notifyObservers() {
    for (int i = 0; i < numObservers; i++)
      observers[i]->notify();
  }
  virtual ~Subject() = 0; // make class abstract
};

Subject::~Subject() {} // destructor must have user-defined implementation (since declared)

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

Example 3.20. Some horse races (see lectures/se/observer). The subject is the race, and it publishes the winners. The observers are the bettors (people making bets), and they declare victory when their horse wins (or cry...).

HorseFace.h

```
class HorseRace: public Subject {
  string lastWinner;
 public:
  HorseRace(string source);
  ~HorseRace();
  bool runRace(); // Returns true if a race was successfully run.
  string getState() { return lastWinner; }
};
```

then

Bettor.h

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

main.cc

```
HorseRace hr;
Bettor Liam(&hr, "Liam", "Secretariat");
// -- some other bettors --
while (hr.runRace())
  hr.notifyObservers();
```

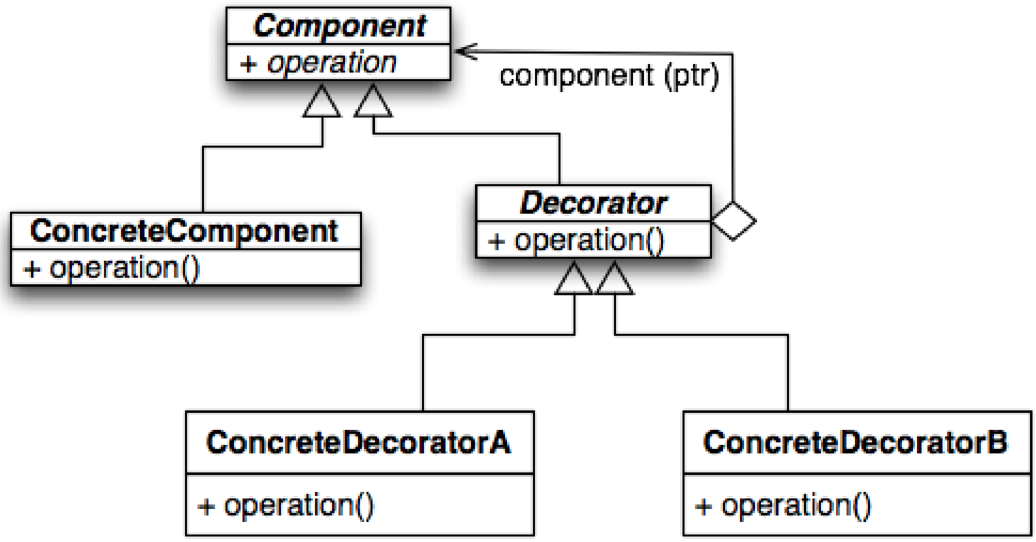
Simplifications:

1. Can merge Subject and ConcreteSubject
2. If just being notified is enough, we don't need getState()
3. If `Subject == Observer` then we can merge these classes. For example, cells in a grid.

3.33 Decorator Pattern

Enhance an object at runtime by decorating it, essentially by adding features or functionality. For example, on a windowing system, we may want to start with a basic window, then add a scroll bar, then add a menu. We want to be able to choose these enhancements at runtime.

Structure:

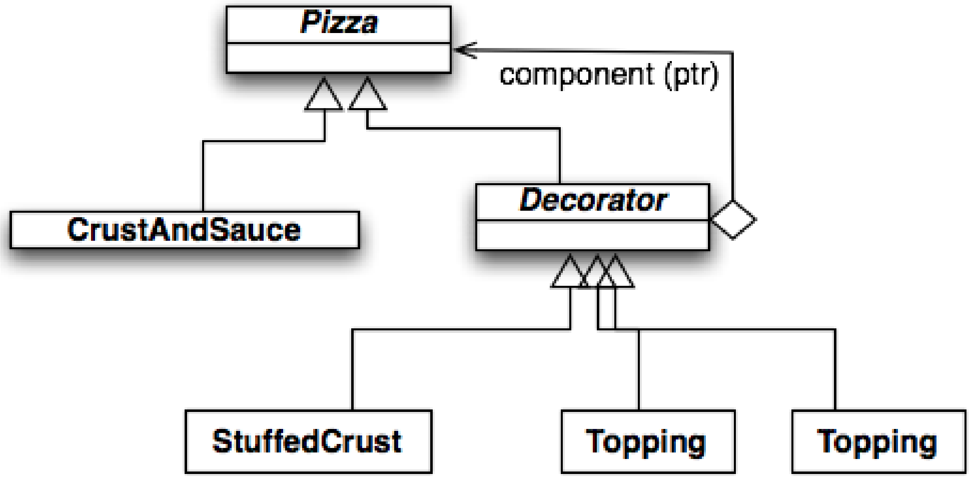


How it works:

- Component - defines the interface (basic operations your objects provide)

- ConcreteComponent - implements the interface
- Decorator - inherits from Componenet and has a pointer to a Component (a Decorator "is-a" component and "has-a" component!). Redirects Component methods to the Component pointer.
- ConcreteDecorators - inherits form the Decorator, overrides any Componenet method(s) you want to "decorate"

All Decorators inherit from abstract Component so component methods can be used polymorphically on all of them. Consider the following class definitions:



Example 3.21.
Code:
Begin with a Component class, in this case it is Pizza. This class is pure virtual.

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

Then we add our concrete class which is essentially a respresentation of the base case. A pizza is at th very least, just some crust and sauce.

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

Then we define the Decorator clsss, which is a subclass of Pizza again, and will be the superclass of multiple optional decorators afterwards like toppings and dipping sauce.

```
class Decorator : public Pizza {
 protected:
  Pizza *component; // has-a relationship
 public:
  Decorator(Pizza *component);
  virtual ~Decorator();
};
```

Then the following are our Decorators. Note that the only reason we have this Decorator superclass above is because we are defining multiple optional decorators. Here are some of our decorations:

```
class DippingSauce: public Decorator {
  std::string flavour;
 public:
  DippingSauce(std::string flavour, Pizza *component);
  float price();
  std::string description();
};
class Topping: public Decorator {
  std::string theTopping;
  const float thePrice;
 public:
  Topping(std::string topping, Pizza *component);
  float price();
  std::string description();
};
class StuffedCrust: public Decorator {
 public:
  StuffedCrust(Pizza *component);
  float price();
  std::string description();
};
```

Note. Decorator is abstract because it doesn't override the pure virtual methods in Pizza.

Consider the following main code:

```
Pizza *p1 = new CrustAndSauce;
p1 = new Topping(”cheese”, p1);
p1 = new Topping(”mushrooms”, p1);
p1 = new StuffedCrust(p1);
cout << p1->desc() << ” ” << p1->price() << endl;
delete p1;
```

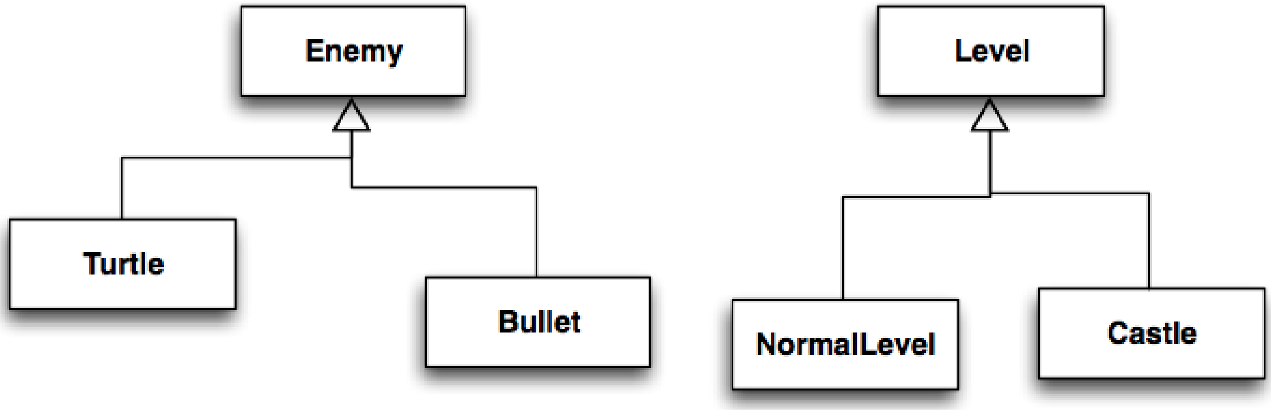
The output would be: `Pizza with cheese with mushrooms with stuffed crust 10.18`
Note that in each case, you are decorating the previous object (p1 changes after each line). You end up with a StuffedCrust which has a pointer to a Topping, which as a pointer to another Topping, which has a pointer to the CrustAndSauce. Calling price() results in a chain of calls that produces the right price: 5.99 + 0.75 + 0.75 + 2.69 (thanks to polymorphism). Also note how the single delete statement leads to a chain of destructors being called, freeing all of the allocated memory. Wikipedia has a nice explanation of the motivation for this design pattern: `http://en.wikipedia.org/wiki/Decorator_pattern#Motivation`.

Note. Make sure you think of it as a chain. The pointer to the object points to the more recently added decoration, and each decoration has a component pointer which points to the decoration added before it. This continues recursively until we hit the base case, which in this example was the CrustAndSauce.

3.34 Factory Method Pattern

Example 3.22. Write a video game with 2 kinds of enemies, turtles and bullets. The system randomly sends turtles and bullets, but bullets become more frequent as you approach the end.

UML:



We don't know exactly which Enemy comes next, so rather than calling Turtle / Bullet constructors directly, we put in a "factory method" in level that creates enemies (Enemys). Consider the following classes for a Level, a NormalLevel, and a Castle.

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

class NormalLevel : public Level {
 public:
  Enemy *createEnemy() {
    // creates mostly Turtles
  }
};

class Castle : public Level {
 public:
  Enemy *createEnemy() {
    // creates mostly Bullets
  }
};

Level *l = new NormalLevel;
Enemy *e = l->createEnemy();
```

Factories integrate well with singletons. Suppose we add another enemy type, Boss, but we only want one Boss, so every time he appears it is the very same one. The solution is to make Boss a singleton (of the Enemy class). Then the UML is updated to include Boss, also deriving from Enemy, taking note of the fact that it is a singleton.

Note. NormalLevel can not generate a Boss, but the Castle can generate a Boss. Meaning, `Castle::createEnemy()` can produce a new Enemy or the singleton Boss instance (difference is transparent to the client).

3.35 Template Method Pattern

What subclasses to override superclasses behaviour, but some aspects must stay the same.

Example 3.23. Suppose we have red turtles and green turtles, then the base clase is Turtle, which looks like

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

where it is up to the subclass to determine how to draw the shell, so we make that a pure virtual function. Then the derived classes:

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

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

Subclasses can't change the way a Turtle is drawn (head, shell, feet), but they change the way the shell is drawn.

3.36 Tools: DGB (GNU Debugger: gdb)

To use this tool you must compile your code with the -g flag to enable debugging information. For example,

```
g++ -g myfile.cc
```

Then, to run the debugger:

```
gdb ./a.out
```

GDB Commands:

- r (run) - runs the program. If the program crashes, tells you which line it crashed on.
- bt (backtrace) - prints the chain of function calls that lead to the crash.
- l (list) - lists source code surrounding the current point of execution
- p (print) - prints the value of a variable

Breakpoints: tell gdb to stop execution at certain points so you can see what's going on.

- b main - break when entering the main function
- b myfile.cc:15 - break at line 15 in myfile.cc

- `n (next)` - run current line of the program without stepping into any functions
- `s (step)` - run one line of program stepping into functions as necessary
- `c (continue)` - continues execution from wherever you are

Check the repository for examples of this tool. (lecturs/se/tools/gdb)

3.37 Relationships Summary

- "has a" (aggregation, uses hollow diamond ♢ on UML)
- "owns a" (composition, uses solid diamond ♦ on UML)
- "is a" (public inheritance, uses triangle △ in UML)

There is another relationship, called "uses for implementation" or "implementation in terms of".

Consider,

```
class LogFile {
  ofstream out;
 public:
  LogFile(string name) : out(name.c_str()) {}
  virtual void log (string s) { out << s << endl; }
};
```

A `loggedWindow` class:

```
class Window {
  LogFile l;
 public:
  Window() : l("window.log") {}
  void drawSquare () {
    l.log("square");
  }
  void drawCircle() {...}
};
```

What if you want to only log the first 100 draws? Then we want to override the virtual log method, but inheritance creates an "is a" relationship. We don't want this though, we want a "uses for implementation" relationship. The solution is to use **private inheritance**.

```
class Window : private LogFile {
  int count;
  void log(string s) {
    if (count > 100) return;
    ++count;
    LogFile::log(s); // call superclass method
  }
 public:
   Window() : LogFile("window.low") {}
   void drawSquare() {
     log("square");
   }
   void drawCircle() {
     log(”circle”);
   }
};
```

To reiterate, private inheritance means that public and protected members of the base class become private in subclasses. Private inheritance does **not** create an "is-a" relationship.

Example 3.24.

```
LogFile l = Window(); // this will not compile
```

(the inheritance relationship is not public)

In general, we prefer fields over private inheritance (meaning its better to include a data member of the superclass instead of using private inheritance). However, use private inheritance if:

- you want to override virtul methods
- you want to access protected members

3.38 Templates

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

What if you want to store something else as data? Do we need a whole new Node class? The alternative is to use a **template**; a class parameterized by type. The code for a more abstract Node template would look like:

```
template <typename T> class Node {
  T data;
  Node<T> *next;
 public:
  Node(T data, Node<T> *next) : data(data), next(next) {}
  T getData() const { return data; }
  Node<T> *getNext() const { return next; }
};

Node<int> intList = new Node<int>(1, new Node<int>(2,0));
Node<char> charList = new Node<char>(’a’, 0);
```

The compilor specializes templates at the source code level, before compilation begins (guarantee this will be a multiple choice question on the exam).

The Standard Template Library (STL)

There are a large number of useful templates, for example dynamic-length arrays: **vectors**. For example,

```
#include <vector>
#indluce <iostream>
using namespace std;
vector<int> v;
v.push_back(1);
v._push_back(2);
...
```

There's a convenient size method for the size of a vector, so we can iterate over vectors:

```
for (int i = 0; i < v.size(); ++i)
  cout << v[i] << endl;
v[i] // accesses i-th element (no subscript checking)
v.pop_back() - remove and return last element
```

This is one way to go throigh a vector, a better way is to use **iterators**.

Iterators

All containers in the standard library implement an `iterator` type, and begin and end methods. An iterator acts like a pointer to the elements.

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

`v.begin()` points to the first element, and `v.end()` points to **one past** the last element.
Some other properties,

```
v.erase(v.begin()); // erases 1st element (i.e., subscript 0)
v.erase(v.begin()+3); // erases 4th element (i.e., subscript 3), using pointer arithmetic
v.erase(v.end()-1); // erase last element
```

Map

Map is for creating association lists. For example an "array" mapping `string`s to `int`s.

```
#include <map>
#include <string>
#include <iostream>
using namespace std;
map<string, int> m; // initilization
m["abc"] = 1; // access element with key "abc" and set value to 1
m["def"] = 4; // access element with ket "def" and set value to 4
cout << m["abc"] << endl; // output
m.erase("abc"); // erase element in the map
if (m.count("def")) // 0 = not found, 1 = found
```

You can iterate over a map:

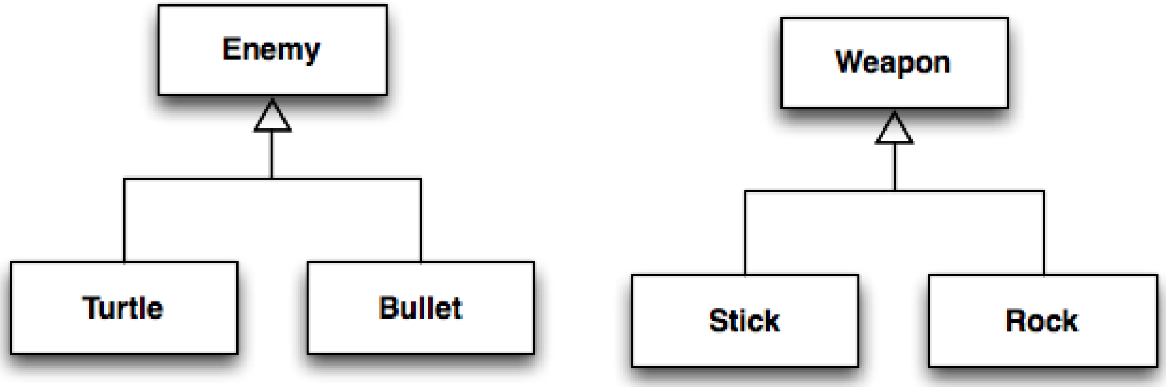
```
map<string, int> m;
for (map<string, int>::iterator i = m.begin(); i != m.end(); i++)
  cout << i->first << " " << i->second << endl;
```

Note that `i` is a pointer to a key-value pair. It uses the sorted key order.

3.39 The Visitor Pattern

Recall that virtual methods are chosen based on runtime of the type of the object on which method is invoked. What if we need to choose the version of a method based on the **two** objects? This is called **double dispatch**.

Example 3.25. Striking enemies with various weapons: Each combination of enemy and weapon needs its own version of `strike()`.



We want something like:

```
virtual void strike(Enemy &e, Weapon &w);
```

If in Enemy:

```
  virtual void strike(Weapon &w);
```

choose based on Enemy but not weapon. However, if in Weapon,

```
  virtual void strike(Enemy &e);
```

choose based on Weapon but not Enemy.

The trick to get dispatched based on both Enemy and Weapon ("double dispatch") is to combine **overloading** with **overriding**:

```
class Enemy {
  virtual void strike(Weapon &w) = 0;
  ...
}

class Weapon { // the "Visitor" class
  virtual void strike(Turtle &t) = 0;
  virtual void strike(Bullet &b) = 0;
  // one overloaded for each kind of Enemy
};

// Then, each kind of Weapon overrides each overload:

class Rock : public Weapon {
  void strike(Turtle &t) {
    //strike Turtle (t) with Rock(*this)
  }
  void strike(Bullet &b) {
    //strike Bullet (b) with Rock(*this)
  }
};

// similarly for class Stick ...
class Stick : public Weapon {
  void strike(Turtle &t) {
    //strike Turtle (t) with Stick(*this)
  }
  void strike(Bullet &b) {
    //strike Bullet (b) with Stick(*this)
  }
};

// Each kind of ENemy delegates to the appropriate overload:

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

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

Then the client using these classes would use:

```
Enemy *e = new Turtle(...);
Weapon *w = new Rock(...);

e->strike(w);
```

It's important to understand exactly what happens when we run `e->strike(w)`, so here it is:

- `Enemy::strike` is (pure) virtual, `*e` is `Turtle`, so `Turtle::strike` runs
- `Weapon::strike` is (pure) virtual, `*this` is `Turtle` so `Weapon::strike(Turtle &t)` is the called overloaded method
- virtual method call resolves to `Rock::strike(Bullet e)` (at runtime, since `*w` is a Rock)

The Visitor pattern can be used to add functionality to existing classes, without having to change or recompile the classes themselves (once the pattern is in place).

Example 3.26. Add a visitor to the Book hierarchy:

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

// same for CSBook and ComicBook

class BookVisitor {
  virtual void visit(Book &b) = 0;
  virtual void visit(CSBook &csb) = 0;
  virtual void visit(ComicBook &cb) = 0;
};
```

Application. Tracking how many of each type of book we have.

Books are organized by author, CSBooks by language, and ComicBooks by hero. We use a mapping map<string, int>. We could add a method to each Book class to update the map, or we could write a visitor.

```
class Catalogue : public BookVisitor {
  map<string, int> cat;
 public:
  map<string, int> getCatalogue()) { return cat; }
  void visit(Book &b) { cat[b.getAuthor()]++; }
  void visit(CSBook &csb) { cat[csb.getLanguage()]++; }
  void visit(ComicBook &cb) { cat[cb.getHero()]++; }
}
```

However this won't compile. Why? Consider a linked list of alternating ints and chars.

alist.h

```
#ifndef __ALIST_H_
#define __ALIST_H_
#include "blist.h"
class AList {
  int data;
  BList *next;
};
#endif
```

blist.h

```
#ifndef __BLIST_H_
#define __BLIST_H_
#include "blist.h"
class BList {
  char data;
  AList *next;
};
#endif
```

So then in our main file we have

main.cc

```
#include "alist.h"
#include "blist.h"
```

at the top, but the problem is that the BList header is included from the AList header but AList is never defined in the BList header so AList is an unknown type when defining BList. The circular include dependencies won't compile.
Question. How much about BList does AList need to know?
Answer. Just that it exists.
So can we declare an AList without having it defined? The answer is to replace "#include blist.h" with class Blist;. This is called a forward declaration. Similarly, for alist.h. This will compile.

3.40 Compilation Dependencies

When does a compilation dependency exist? Consider some class A in a.h, and the following seperate scenarios: (I) first a class that inherits from A,

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

(II) one that has A as a data member,

```
#include "a.h"
class C {
  A myA;
  ...
};
```

(III) one that has a pointer to an A,

```
class A;
class D {
  A *myAp;
  ...
};
```

(IV) one that has a functin that uses A,

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

Like I showed in the code, the first two have a compilation dependency (because you need to know how big A is in order to know how big B is if it inherits from A), and since we have a member variable, we need again to know how big A is in C. For the last two cases, we don't need the know anything about A, since we know the size of a pointer, and the function is just a prototype declaration.

The advice is, if there's no compilation dependency necessitated by the code, don't introduce one with an unnecessary include statement. Now, in the implementations of D and E,

d.cc

```
#include "a.h"
void D::f() {
  myAp->someMethod();
};
// at this point we need to know more about A, there IS a true compilation dependency
```

Do the include in the .cc file instead of the .h file where possible. Now, we can fix the BookVisitor example.

```
#ifndef __BOOKVISITOR_H__
#define __BOOKVISITOR_H__

//#include "book.h"
//#include "csbook.h"
//#include "comicbook.h"

class Book;
class CSBook;
class ComicBook;

class BookVisitor {
  public:
    virtual void visit(Book &b) = 0;
    virtual void visit(CSBook &csb) = 0;
    virtual void visit(ComicBook &cb) = 0;
    virtual ~BookVisitor();
};

#endif
```

Now, let's look at something else; consider the XWindow class:

```
#include <X11/xlib.h>
class XWindow {
  Display *d;
  Window w;
  int s;
  GC gc;
  unsigned long colours[10];
 public:
  ...
};
```

With regards to all of this private data; we can look at it, but do we know or care what it all means? What if I needed to add or change a private member of XWindow? All clients who #included it would need to recompile. It would be better to hide these implementation details away. Enter the "pimpl" idiom (this is the worse name I've ever seen). It means "ptr to implementation". Create a struct to hold the private data.

xwindowimpl.h

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

Now in window.h, theres no need to include xlib.h, just forward declare the implementation struct.

```
struct XWindowImpl;
class XWindow {
  XWindowImpl *pImpl; // no compilation dependency on XWindowImpl.h
 public:              // clients also don't depend on XWindowImpl.h
  ...
}
```

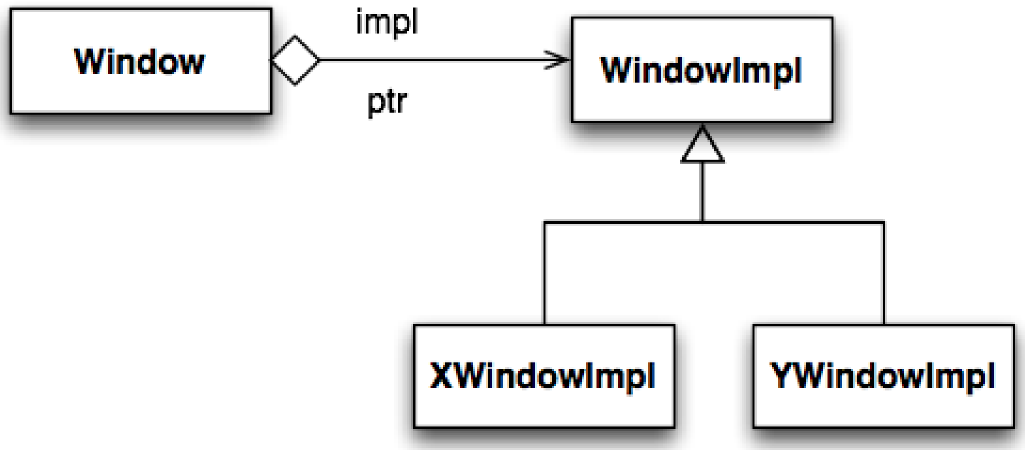
Now, in `window.cc`,

window.h

```
#include "window.h"
#include "XWindowImpl.h"
XWindow::XWindow(...) : pImpl(new XWindowImpl(...)) {}
```

In other methods, replace the fields `d, w, s,` etc with pImpl->d, pImpl->w, etc... Now if you need to change XWindow's implementation only `window.cc` needs to be recompiled.

Generalization. What if there are several possible Window implementations, say XWindow and YWindow? Make the Impl struct a superclass.



This is called the **Bridge Pattern**: pimpl idiom with subclassing to accomodate alternate implementations.

Now, to change implementations, we just need to recompile Windows constructor and then relink.

3.41 Coupling and Cohesion

These are measures of design quality.

Definition 3.5 (coupling)**.** The degree to which distinct program modules depend on or affect each other. For example, interaction via function calls, shared data are considered low coupling. Additionally, suppose modules have access to each other's implementation (friends), this indicates high coupling.

Low coupling usually reflects good structure and design.

Definition 3.6 (cohesion)**.** The degree to which elements of a module belong together. That is, how much they cooperate to achieve a common goal. For example, <algorithm> (the library) has a common theme, but otherwise it's a bunch of unrelated algorithms; this is an example of low cohesion. Another example is <string> which is highly cohesive - elements work together to support C++ strings.

High cohesion usually implies readability, and maintainability.

The goal is to have **low coupling** and **high cohesion**.

3.42 Casting

In C:

```
Node n;
int *i = (int *) &n; // C-style cast
```

It forces C++ to treat a Node* as an int*. Try to avoid using C-style casts in C++.

In C++:

1. **static_cast**

   - use this for "sensible" casts, when you know you have an object of a specific type
   - no runtime checking is performed here
   - for example,

   ```
   double d = 0;
   int i = static_cast<int> (d);
   ```

   another example,

   ```
   Book *b = new CSBook(...);
   CSBook *csb = static_cast<CSBook*>(b);
   ```

   You are taking responsibility that b actually points to a CSBook.

2. **reinterpret_cast**

   - allows you to cast between any arbitrary types
   - disregards type safety (unsafe, avoid or use at own risk)
   - for example,

   ```
   Vector v;
   Student *s = reinterpret<Student *>(&v);
   ```

3. **const_cast**

   - allows you to convert between const and non-const
   - can "cast away const"
   - for example

   ```
   void g(int *p); // doesn't change *p
   void f(const int *p) {
     g(const_cast<int*>(p));
   }
   ```

4. **dynamic_cast**
   Is it safe to convert a Book* to a CSBook*? For example,

```
Book *pb;
...
static_cast<CSBook*>(pb)->getLanguage(); // safe?
```

   - depends on what pb actually points to
   - better to do a tentative cast
   - for example,

   ```
   Book *pb = ...;
   CSBook *pcsb = dynamic_cast<CSBook*>(pb);
   // dynamic_cast checks the dynamic type of the argument
   ```

   If the cast works, (pb points to CSBook, or a subclass of it), pcsb points to the object. If the cast fails, pcsb will be NULL. So we can do checks,

```
if (pcsb) cout << pcsb->getLanguage() << endl;
else cout << "Not a CS Book" << endl;
```

You cna use dynamic casting to make decisions based on an objects runtime type. (RTTI = runtime type information).

```
if (dynamic_cast<ComicBook*>(b)) ...
else if (dynamic_cast<CSBook*>(b)) ...
else ...
```

Code like this is highly coupled to the Book class hierarchy and might indicate poor design. It is better to use **virtual functions** or write a **visitor** (if possible). Dynamic casting aldo workd with references,

```
CSBook csb(...);
Book &b = csb;
CSBook &csb2 = dynamic_cast<CSBook&>(b);
```

if b is a reference to a CSBook then csb2 is a reference to the same CSBook. If not...? That's a problem (no such thing as a null reference).

3.43 Exceptions

What happens when things go wrong in C++?

- dynamic_cast on reference fails
- new fails
- vector::at goes out of bounds (note that at is the checked version of vector::operator[]). For example,

```
vector<int> v;
v.push_back(1);
v.push_back(0);
vout << v.at(2); // out of range
```

What *should* happen? The problem is that

- vector's code can detect the error but doesn't know what to do about it
- client can respond but can't detect the error

The solution is to **raise an exception**.

So what happens? By default, execution stops. But, we can write **handlers** to catch exceptions and deal with them, allowing execution to continue. For example, vector::at throws exception std::out_of_range when it fails. We can handle it as follows:

```
#include <stdexcept>
using namespace std;
...
try {
  cout << v.at(2) << endl; // map statements that might lead to exceptions in the try ↩
      block
} catch (out_of_range) {
  cerr << "range error" << endl;
}
```

Now consider our own example:

```
void f() {
  throw out_of_range("f"); // raise an exception
}
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 throws out_of_range;

- control goes bck through the call chain (unwinds the stack) until a handler is found - in this case, all the way back to main.
- the exception is handled in the catch block
- if no handler in the entire call chain, the program terminates

So what is out_of_range? It's a class.

```
throw out_of_range("f") // invokes its constructor with arg "f" (auxillary info) and throws it
```

To examine that info:

```
catch (out_of_range ex) { // like param declaration
  cout << ex.what() << endl;
}
```

A handler can do part of the recovery and throw another exception for further handling:

```
catch (SomeError s) {
  ...
  throw someOtherError("...");
}
```

or even throw the same exception again:

```
catch (SomeError s) {
  ...
  throw;
}
```

Note. s may be a subtype of SomeError. Then,

- throw; - throws the same exception retaining the **actual** type of s.
- throw s; - throws a new exception of type SomeError

Also, catch exceptions to avoid slicing.

A handler can act as a "catch-all":

```
try {
  ...
} catch (...) { // catches all exceptions (note these literally are ellipses)
  ...
}
```

We can even chain multiple handlers:

```
try {...}
catch (SomeError) {...}
catch (Some OtherError) {...}
catch(...) {...}
```

The first handler (only) accepts the exceptions.

3.44 Exceptions & Polymorphism

Suppose you have a `DerivedExn` class that inherits from a `BaseExn` class. Then,

```
try { throw DerivedExn(); }
catch (BaseExn ex) { // slicing (catching by value)
  throw ex; // slicing (even if caught by reference)
}
```

To avoid slicing, catch by reference and use `throw;` (no arguments) to throw original exception as-is.

```
try { throw DerivedExn(); }
catch (const BaseExn &ex) {
  throw;
}
```

Note.

```
try {
  DerivedExn d;
  BaseExn &b = d;
  throw b;
} catch (DerivedExn&) {...}
catch (BaseExn&) {...}
```

The `BaseExn` handler runs (the static type of the exception thrown determines the handler).

You can throw anything you want! (don't have to throw objects).

Example 3.27. `lectures/c++/exceptions/exfib.cc`

```
void fib(int n) {
  if (n == 0) throw 0;
  if (n == 1) throw 1;
  try {
    fib(n-1);
  }
  catch (int a) {
    try {
      fib(n-2);
    }
    catch (int b) {
      throw (a + b);
    }
  }
}

int main() {
  int n;
  while (cin >> n) {
    try {
      fib(n);
    }
    catch (int m) {
      cout << m << endl;
    }
  }
}
```

This is a recursive Fibonacci by throwing / catching ints. However, regular fibonacci is much faster.

Good Practice. Define exception classes, or use appropriate existing ones, to denote your error cases.

Example 3.28.

```
class BadInput {};
try {
  while (true) {
    int n;
    if (!(cin >> n)) {
      throw BadInput();
      ...
    }
  }
} catch (BadInput &) {
  cerr << "Input not well formed" << endl;
}
```

3.45 Some Standard Exception Classes

- domain_error - for errors in mathematical domains (for eample, sqrt(-1))
- length_error - attempting to resize strings or vectors that are too big
- bad_alloc - trying to allocate more memory than you have (e.g., new)
- bad_cast - invalid cast on reference
- ios_base::failure - I/O errors (e.g., cin.exceptions(istream::failbit) will throw exception on fail())

Example 3.29. Application of exceptions and a good use of dynamic casting: (recall : virtual operator=)

```
CSBook &CSBook::operator=(const Book &other) {
  if (this == &other) return *this;
  CSBook &csbother = dynamic_cast<CSBook &>(other); // throws exn if fails
  title = csbother.title;
  ...
  language = csbother.language();
  return *this;
}
```

3.46 Exception Safety

Consider:

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

What if g raises an exception? During stack-unwinding, all stack-allocated data is cleaned up (destructors run, memory reclaimed). But heap-allocated memory is not destroyed. Therefore if g throws, p is leaked. We could wrap `g()` in a `try-catch` and put a `delete p;` in the handler, but that is tedious and error-prone.

Some languages have a "finally" clause for cleanup code - not in `C++`. When an exception is raised in `C++`, the only guarantee is that destructors for stack-allocated data will run.

Therefore, use stack-allocated data with destructors as much as possible. But, **never** let a destructor throw an exception. If the destructor runs during stack-unwinding while dealing with another exception, you now have **two** active unhandled exceptions, and your program **will** abort.

C++ programming idiom : RATI (Resource Acquisition Is Initialization).

Every resource should be wrapped in a stack-allocated object whose job it is to release it. Resources are acquired during initialization, and released during destruction. For example, in files

```
ifstream f("file"); // acquiring the resource "file" = initializing the object f
```

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

Can this be done with dynamic memory? Yes.

- `class auto_ptr<T>`
  - takes a `T*` in the constructor
  - deletes the pointer in the destructor
  - in between: can dereference just like a regular pointer

  Example 3.30.

  ```
  auto_ptr<MyClass> p (new MyClass);
  ```

  - if exception is raised, then this will be destroyed during stack unwinding.
  - otherwise it will be destroyed when p goes out of scope

auto_ptr copy semantics

- ownership gets passed on to the target (don't want to delete same pointer twice!)
- original owner becomes invalid (`NULL`)

Example 3.31.

```
auto_ptr<MyClass> q = p;
```

- `auto_ptr` was designed to help avoid memory leaks in the presence of exception handling, **not** as a general purppose smart pointer.

3.47 How Virtual Methods Work

First, how are objects laid out in memory?

- data members stored in order of declaration
- methods are stored as ordinary functions, seperate from objects

Example 3.32.

```
Vector v(1,2);
cout << sizeof(v); // 8 (space for 2 ints)
```

but if `Vector` has **virtual** method(s), `sizeof(v)` increases to 16. Why the extra 8 bytes? Recall that

$$\texttt{Book *pb = new} \begin{cases} \texttt{Book} \\ \texttt{CSBook} \\ \texttt{ComicBook} \end{cases}$$

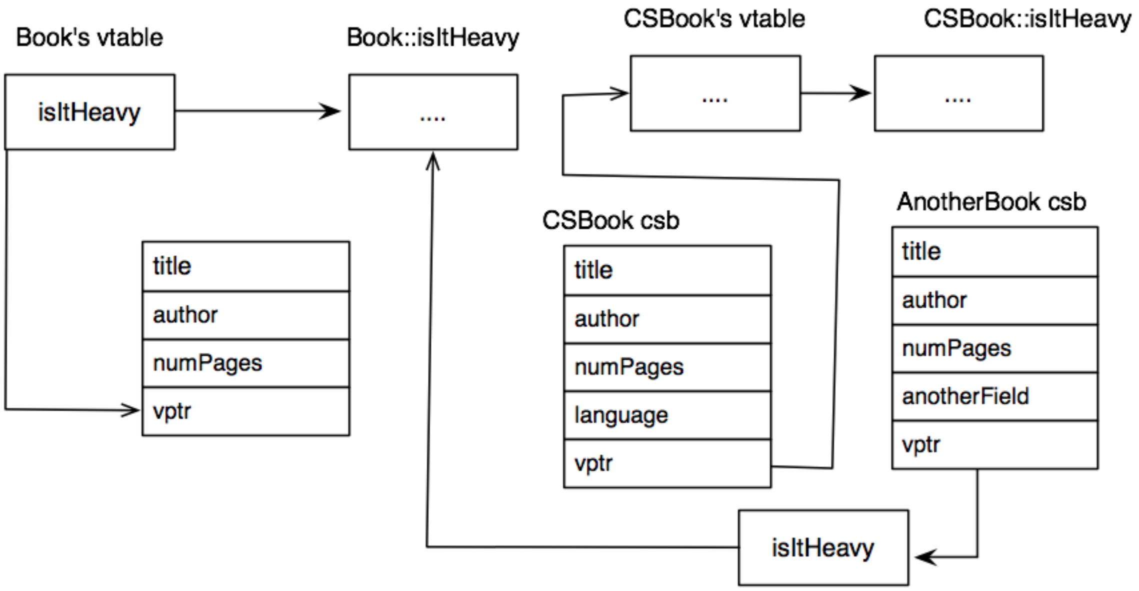
where `isItHeavy()` is a virtual method.

- choice of which implementation to run is based on type of actual object at **runtime** (compiler can't know in advance) How?

For each class with virtual methods, the compiler creates a table of function pointers called the **vtable**. Each instance of the class stores a pointer (`vptr`) to this table.

Example 3.33.

```
class Book {
  string title, author;
  int numPages;
 public:
  string getTitle();
  virtual book isItHeavy();
};
```



For each virtual method, the class' vtable has a pointer to the most derived version of the method accessible to the class. Calling a virtual function (these all happen at runtime):

- follow objects vptr to vtable
- fetch pointer to actual method body
- follow function pointer and call the function

Therefore virtual function calls incur a small overhead cost.

Concretely, how should objects be laid out in memory? This is compiler-dependent.

- g++:

vptr
⋮

  where the dots mean "fields".

  Example 3.34.

```
class A {
  int a, c;
  virtual void f();
};
class B : public A {
  int b, d;
};
```

vptr
a
c

vptr
a
c
b
d

So a B looks like an A if you ignore the last two fields.

3.48 Multiple Inheritance

A class can inherit from more than one class.

```
class A {
  int a;
};

class B {
  int b;
}

class C : public A, public B {
  int c;
  void f() {
    cout << a << b < < c;
  }
};
```

The main challence is explicit repeated inheritance. For example is we have a class D that inherits from other classes B and C which both inherit themselves from A. So,

```
//  A   A
//  ^   ^
//  |   |
//  B   C
//  ^   ^
//   \ /
//    D
class D : public B, public C {
  int d;
  void g() {
    cout << a; // which a? (ambiguous, compiler rejects)
  }
};
```

We need to specify B::a or C::a.

Note. If a were public,

```
D myD;
myD.B::a // or myD.C::a
```

But, if B and C inherit from A, should D have **one** A part ot **two** (default)?

(i.e., should B::a, C::a be the same a, or different a's?)

What if we want to employ **virtual inheritance**?

```
//    A
//    ^
//   / \
//  B   C
//  ^   ^
//   \ /
//    D
class B : virtual public A { ... };
class C : virtual public A { ... };
```

This is sometimes known as the "deadly diamond", and sometimes programmers dress up as it for halloween.

Example 3.35. IO Stream hierarchy: basically `iostream` is like `D` in our example, and `ostream` and `istream` are like `B` and `C`, and `ios` like `A`. (do a Google search)

How would `D` be laid out? Maybe like this?

vptr
A fields
B fields
C fields
D fields

however the top three don't look like a `C` and the top two doesn't look like a `B`. The object should look the same whether pointed to by `A*`, `B*`, `C*`, `D*`. This might work for `B*` but not `C*`. What g++ does:

vptr
B fields
vptr
C fields
D fields
vptr
A fields

then

- `B*` points to the top
- `D*` points to the top
- `C*` points to the second vptr entry
- `A*` points to the third vptr entry

The diagram doesn't look like all of `A, B, C, D` simultaneously, but pieces of it look like `A`'s, `B`'s, `C`'s, `D`'s. `B` needs to be laid out so that we can find its `A` part, but how?

Solution. Store an offset to base class object stored in its part of the vtable. This means **pointer assignment** among `A, B, C, D` could actually **change** the address stored in the pointer.

Example 3.36.

```
D *d;
...
A *a = d; // changes the address
```

Casting under multiple inheritance may also change the value of the pointer, except `reinterpet_cast` will **not**.

See: `lectures/c++/vtable`.

3.49 Return Value Optimization (RVO)

Consider:

```
class C { ... };
c f() { return C(); }
int main () {
  C c = f();
}
```

Theoretically:

1. `f()` calls `C`'s default constructor
2. `C` object is copied to a temporary object in main's stack frame when `f` returns
3. `c` is initialized via copy constructor (with the temporary object)

So, we have 1 default constructor and 2 copy constructors.

In practice: Just the default constructor runs. Why? These two temporaries are only for shuffling data around and are otherwise unobservable.

The compiler has special permission to skip the copy constructors and let `f` write its return value directly into `c`, **even** if the copy constructor has been implemented and even if it has observable side effects - this is called **return value optimization** (RVO) (if you wan't this, compile with the `-fno-elide-constructors` flag).

For assignment operators, there is no such special license. Old data would need to be properly destroyed. So we can't avoid copying:

```
int main() {
  C c = f(); // default constructor (RVO)
  c = f(); // default constructor + operator=
}
```

Now, suppose `C` objects are large:

```
class C {
  int *a; // array of 100's of ints, say
};
```

Returning a `C` object from `f` by value (maybe copies to a tempt, then) copies to the receiving variable - expensive. Can we avoid this expense? In `C++11`, **yes**, with **rvalue refs**.

The idea: rather than copying something that is going to be destroyed, just point to it and don't destroy it. Assignment in `C++` has the left side of the '=' operator as being an lvalue and the right hand side an rvalue.

In `C++03`, a reference (&) must always refer to an lvalue (unless const). In `C+11` an **rvalue reference**(&&) lets you bind a reference to an rvalue - a temp object.

`X&&` is an rvalue reference to type `X`. This provides a way to distinguish between temp and non-temp objects. So you could write

```
C& c::operator=(C&& other) {
  delete [] a; // releases current data
  a = other.a; // point at other’s data rather than deep-copying it
  other.a = NULL; // so that the data won’t be destroyed when other’s destructor runs
  return *this;
}
```

This is called **a move assignment operator** since we are effectively moving the data rather than copying it.

Other C++ goodies:

- fixed nested templated

  Example 3.37. In C++03 we need a space here: `vector<vector<int> > v` otherwise it's interpeted as the bitshift operator.

- additions to `std` library
- automatic type deduction
- anonymous functions ("lambdas")

Everything discussed in class is fair game on the final, emphasis on the second half of the course, focus on fundamentals.