CS 246: Object Orientated Programming

Darrell Aucoin

Winter 2014

Contents

"Programming for Big Kids"
Midterm: Tuesday, March 4 4:30-6:20
Grading:

Assignments: 7, 7, 7, 7, 12% (project)

Midterm: 20%

Final: 40%

You must use linux in this course!

Options:

1. Mac: Command prompt will work most of the time.
2. Install Linux on your computer.
3. Run Linux on a virtual machine.
4. (Windows User) ssh connnection into school linux machine (download putty.exe)
   - WinSCP for file transfer
   - The big advantage to using on the school machine is that it is backed up
5. Cygwin (Unix like environment for windows)

- XWindows server for graphical component (XMing)

4 Modules:

1. Linux Shell (2 Weeks)
2. C++
3. Tools
4. Software Engineering

Homework for next class: Print out linux command reference and bring to every class.

Part I

Module 1: Linux Shell

Chapter 1

Lecture 1

Shell: The interface of the operating system. Gets the OS to do things for us.

- Run programs
- Manipulate files
- etc.

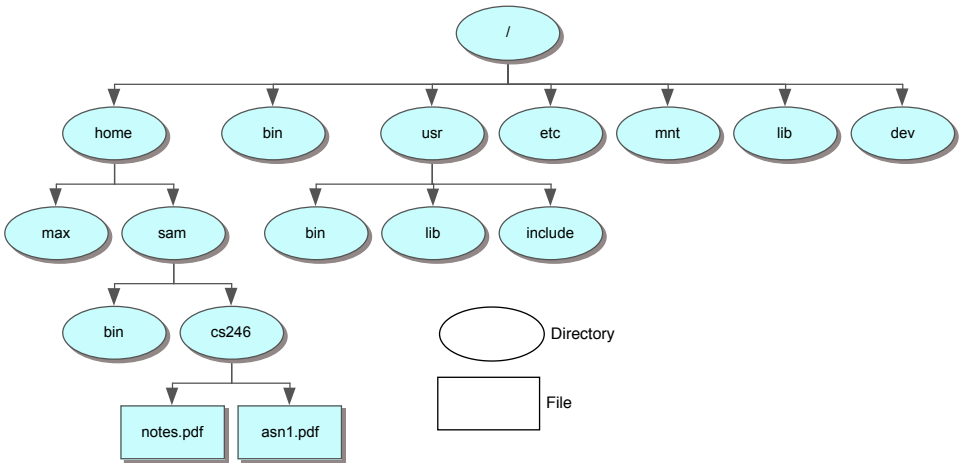
The shell we will use bash (born again shell). Make sure you are running bash.

```
echo $0
```

1.1 Linux File System

- Files (programs, data)
- Directories (contains files, other directories)
  - Technically directories are considered files

Directories have a tree structure.



Specify files by their path from root.
/bin bin director within /
/user/bin bin within user, within /

Current Directory: Where you are "sitting" in the file system right now.

- Initially your home directory

`pwd` Command to see current directory.

Relative Path: Starts from your current directory, instead of /

Example. If current dir is /u/j24smith then the relative path cs246/a1/a1.cc is the same as the absolute path /u/j24smith/cs246/a1/a1.cc

Absolute Paths: Start with /

Relative Paths: Don't start with /

Special Directories:

`.`	Current directory	Relative
`..`	Current directory's parent	
`../..`	(for the grandparent)	
`~`	Home directory	Absolute
`~userid`	userid's home directory	

`ls` command to see what's in a directory.

- ls doesn't show everything: it leaves out files out all files that starts with . (hidden files)
- To see everything:

```
ls -a
```

1.2 Wild Card Matching

What if I just want to see just the text files?

```
ls *.txt
```

`*.txt` is called a globbing pattern. and * means "match anything" ∴ *.txt = "anything ending with .txt"

Q: What happens when I supply a globbing pattern in the command line?

A: The shell substitutes every file in the current directory that matches the pattern onto the command line.

- If no match, original glob is left unaltered on the command line.

Example. if dir contains {1.txt t2.txt index.html}
then `ls *.txt` is transformed by the shell to `ls t1.txt t2.txt` and you get `t1.txt t2.txt`

Works for other commands too: (since wildcard matching is done by the shell)

```
echo hello
```

- prints hello

```
echo *.txt
```

- transformed into `echo t1.txt t2.txt`
- prints `t1.txt t2.txt`

```
rm *.txt
```

- transformed into `rm t1.txt t2.txt`
- removes `t1.txt t2.txt`

Q: What if I really want to print *.txt?

A: Put it in quotes: single or double.

1.3 Files

`cat` Displays the contents of a file.

```
cat /usr/share/dict/words
```

- Big list of words

Command

`^c` to stop the current process

`^d` at the beginning of a line sends an end of file signal to the command. (Allows the program to finish)

Q: What if you just type cat?

A: Prints everything you type in.

Useful? Yes, if you can capture the output in a file.

```
cat > output.txt
```

In general,

```
command arg > file
```

executes command args, and captures the output in file instead of sending to screen.

- Called output redirection.

Chapter 2

Lecture 2

Recall:

```
Cat > output.txt
```

- Output redirection

Can also redirect input

```
cat < input.txt
```

- Takes input from input.txt instead of keyboard
- Displays the contents of input.txt
  - Seems to be equivalent to `cat input.txt`

Q: What is the difference?

A:

- `cat input.txt` passes the name input.txt as an argument to `cat`
  - `cat` opens the file and displays it's contents
- `cat < input.txt` the shell opens the file and passes it's contents to to cat in place of keyboard input.

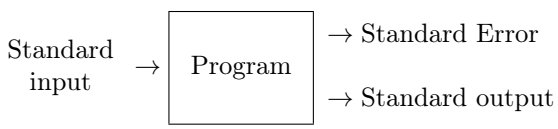
Many (but not all) commands accept both forms.

Can do both input and output redirection:

```
cat < input.txt > output.txt
```

- Takes chars from input.txt and puts them in output.txt
  - A copy

Every process is attached to 3 streams:



By default, stdin is connect to keyboard and stdout is connected to screen.

Redirection:

`>` connects stdout to a file.

`<` connects stdin to a file.

- stderr is a seperate output steam for error messages.
  - So that output and error messages can be sent to different places
  - So that error messages don't clutter output files and corrupt formating
  - Also, stdout may be buffered, thus the system may wait to accumulate output before actually displaying it (flushing) it.
    - Stderr is never buffered, so you get your error messages immediately.
  - Can also redirect stderr:

```
myprogram < in.txt > out.txt 2> errorlog.txt
```

2.1 Pipes

- Lets you use the output of one program as the input as another
- Set the 2nd program's stdin to the 1st program's stdout.

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

```
head -n
```

- Display the first n lines of file

```
wc -w
```

- Counts words

Solution:

```
head -20 sample.txt | wc -w
```

Use the input of first command as the input of the second command
or

```
cat sample.txt | head -20 | wc -w
```

Example. Suppose files words1.txt, words2.txt, contain list of words, one per line.
Print a duplicate free list of all the words that occur in any of words*.txt

`uniq` removes adjacent duplicates

`sort` sorts lines

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

Q: Can we use the output of one program as the parameter of another?

A: Yes, put the command in backquotes (under the esc key)

Example.

```
echo Today is ‘date‘ and I am ‘whoami‘
```

- The shell executes the date command and uses the result as an argument.

or

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

2.2 Pattern Matching in Files

Tool: `grep` ("global regular expression print")

or better:

`egrep` ("extended grep") `grep -E`

```
egrep pattern file
```

- prints every line in file that contains pattern

Example. Prints every line in index.html that contains cs246.

```
egrep cs246 index.html
```

Example. How many lines in index.html contain cs246?

```
egrep cs246 index.html | wc -l
```

Q: What kinds of patterns can we search for?

A: Regular expressions (not the same as globbing)

Example. Seach for cs246 or CS246

```
egrep ‘‘cs246|CS246’’ index.html
egrep ‘‘(cs|CS)246’’ index.html
egrep ‘‘(c|C)(s|S)246’’ index.html
```

- Also matches Cs246, cS246

```
egrep ‘‘[cC][sS]246’’ index.html
```

[...] match any one of the characters between [and]

Example. [abc] = (abc)

[^...] match anything except one of the characters between [and]

Accept an optional space

```
egrep ‘‘[cC][sS] ?246’’ index.html
```

? 0 or 1 of the preceding expression.

Additional syntax

* 0 or more of the preceding expression.

Example. "(cs)246"

- matches 246, cs246, cscs246, ...

. Any single character

So, .* is the equivalent any sequence of characters

Example.

```
egrep ‘‘cs.*246’’ index.html
```

- Matches lines that contain a string that starts with cs and ends with 246

^ begining of line

Example.

```
egrep ‘‘^cs246’’ index.html
```

- Lines that start with cs246

$ end of line

Example.

```
egrep ‘‘cs246$’’ index.html
```

- Lines that end with cs246

Example.

```
egrep ‘‘^cs246$’’ index.html
```

- Lines that contain just cs246

+ matches 1 or more occurances of the preceding pattern

.+ all strings except the empty string

Example. Prints all lines of even length

```
egrep ‘‘^(..)*$’’ index.html
```

Example. Print all files all files in the current directory whose names contain exactly 1 a.

```
ls | egrep ‘‘^[^a]*a[^a]*$’’
```

Chapter 3

Lecture 3

3.1 Output redirection sematntics:

```
program > file
```

- If file doesn't exist, it is created
- If file does exist it is replaced

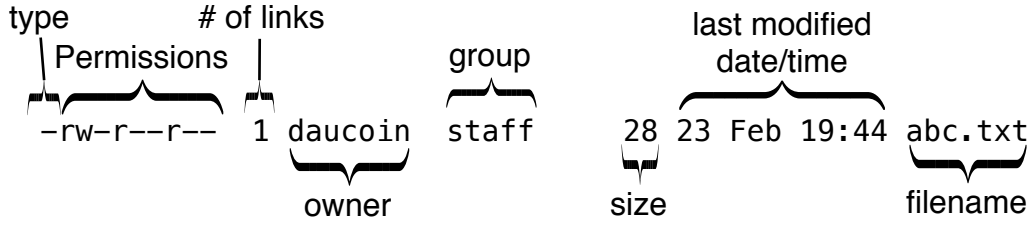
```
program >> file
```

- Output is appended to a file

3.2 Permissions

`ls -l` gives a "long form" directory listing

Example.



Groups

- A user can below to one or more groups
- A file can be associated with one group

Type: - means ordinary file
d means directory

Permissions: 3 groups or 3 bits

Example.

$$\underbrace{rwx}_{\substack{\text{user}\\\text{bits}}}\ \underbrace{r-x}_{\substack{\text{group}\\\text{bits}}}\ \underbrace{r--}_{\substack{\text{other}\\\text{bits}}} \qquad \begin{cases} r & \text{read bit} \\ w & \text{write bit} \\ x & \text{execute bit} \end{cases}$$

User bits: What the file's owner can do with the file

Group bits: What members of the file group (other than the owner) can do

Other bits: What all other users can do.

3.2.1 Each group

What they mean depends on the type of file.

Bit	Ordinary Files	Directories
r	Files contents can be read	Directories contents can be read (ls the directory)
w	Files contents can be modified	Directory contents can be modified (eg. add/remove files)
x	File can be executed as a program	Directory can be navigated (i.e. can cd into it)

Note: If a directory's execute bit is off, you cannot enter that directory, you cannot access any file or subdirectory within it.

- The owner is the only one that change a file's permissions (other than administrator).

3.2.2 Changing Permissions

```
chmod mode file
```

Mode:

Ownership class

u - user
g - group
o - other
a - all

Operator

\+ add permission
\- revoke permission
= set permission exactly

Permissions

r - read
w - write
x - execute

Example. Give others read permissions

```
chmod o+r file
```

Revoke exec perm form group:

```
chmod g-x file
```

Make everyone's perm rx

```
chmod a=rx file
```

Give owner full control

```
chmod u=rwx file
```

3.3 Variables

x=1 (note: NO spaces)

echo $x to get the value of x

```
$ x=1
$ echo $x
1
```

Note: To fetch the value of a variable, precede with $

- When setting a variable, no $
- It is good practice to put the variable name in brace {}, eg ${x}
- All variables have type **string**
- So x above is the **string** 1

Example.

```
dir=~/cs246
echo ${dir}
cd ${dir}
ls ${dir}
```

Some "Global" variables set up for you

- Most important: PATH
  - A list of directories
  - When you type a command, the shell searches these directories in order, for a program of that name.
    - Runs the first one it finds

```
echo ${PATH}
```

- prints the path

```
echo ``${PATH}''
```

- Prints the path again

```
echo '${PATH}'
```

- Prints `${PATH}` literially

Double quotes allow variables and back quotes to expand to their values, single quotes do not.

3.4 Shell Scripts

- Files that contain sequences of linux commands, executed as programs

Example. print the date, current user, currect directory

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

`#!/bin/bash` The "shebang" line

- Tells linux to execute this file as a bash script

- We need to give the file execute permission

```
chmod u+x script
```

To execute, we execute it. If it is in the current directory

```
./script
```

Command-line args: $1, $2, ...

Example. check whether a word is in the dictionary,

```
./isItAWord hello
```

Script

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

- Prints nothing if word not found
- Prints the word if found

Example. A good password is not in the dictionary: answer whether a word is a good password

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

`> /dev/null` suppresses output

Note: Every program returns a status code, typically 0 for sucess, non-zero for failure.

- grep will return 0 if found, 1 if not found.

`$?` status code of the most recently-executed command

Chapter 4

Lecture 4[1]

Recall:

```
#!/bin/bash
egrep ‘‘^$1$’’ /usr/share/dict/words > /dev/null
if [ $? -eq 0 ]; then
    echo Not a good passwrd
else
    echo Maybe a good password
fi
```

[]	is a program, so the spaces need to be there. The contents are it's arguments
	Returns 0 if true
	1 if false

Want to make sure the user passes exactly one argument. Otherwise print a usage message. Common the usage message into a function.

```
#!/bin/bash
usage(){
    echo ‘‘usage: $0[2] password’’
}
if [ $#[3] -ne 1 ]; then
    usage
    exit 1
fi
egrep ‘‘^$1$’’ /usr/share/dict/words > /dev/null
if [ $? -eq 0 ]; then
    echo Not a good passwrd
```

[1] Jan 16, 2014

[2]
- $0 here Name of the program (good Password)
- In case you rename the script

[3]

$# # of args

```
else
    echo Maybe a good password
fi
```

4.1 General Format for condition statement

```
If [ condition ]; then
    ---
    ---
elif [ condition ]; then
    ---
    ---
else
    ---
    ---
fi
```

4.2 Loops

Loops: Print #'s from 1 to $1

```
#!/bin/bash
usage(){
    ...
}
if [ $# -ne 1]; then
    useage
    exit 1
fi
if [ $1 -lt 1]; then
    usage
    exit 2
fi
x=1
while [ $x -le $1 ]; do
    echo $x
    x=$((x+1))[4]
done
```

[4] $((...)) for arithmetic expressions

4.2.1 Looping over a list

Example. Rename all .cpp to .cc

```
#!/bin/bash
for name in *.cpp; do
    mv ${name} ${name%cpp}cc[5]
done
```

How many times does word $1 occur in file $2?

```
#!/bin/bash
x=0
for word in `cat $2`: do
    if [ "$word" = "$1" ]; then[6]
        x=$((x+1))
    fi
done
echo $x
```

Example. Payday is the last friday of the month. When is this months payday?
2 tasks: find the day, and report the answer

```
#!/bin/bash
answer(){
    if [ $1 -eq 31 ];
        echo "on the ${1}st"[7]
    else
        echo "on the ${1}th"
    fi
}
answer `cal | awk '{print $6}' | egrep "[0-9]" | tail -1`
```

Example. Generalize the above to any month

```
cal February 2014
```

- Gives Feb's calender

`./payday February 2014` shold give Feb's payday

```
#!/bin/bash
answer(){
    if [ $1 -eq 31 ];
        echo "on the ${1}st"
    else
        echo "on the ${1}th"
    fi
```

[5] `${name%cpp}` = value of name without the trailing ccp
[6] Generally good to put variables in "" to make it a single thing
[7] in a function, defintion $1, $2, are the arguments of the function

```
}
answer ‘cal $1 $2 | awk ’{print $6}’ | egrep "[0-9]" | tail -1‘
```
[8]

To include the month

```
#!/bin/bash
answer(){
    if [ $2 ]; then # if month is not blank
        preamble=${2}
    else
        preabmle=’’This month’’
    fi
    if [ $1 -eq 31 ];
        echo ‘‘${preamble}’’ ‘‘on the ${1}st’’
    else
        echo ‘‘${preamble}’’ ‘‘on the ${1}th’’
    fi
}
answer ‘cal $1 $2 | awk ’{print $6}’ | egrep "[0-9]" | tail -1‘ $1
```

4.3 Software Enigeering (SE): Testing

Essential part of development:

- On going, not just at the end
- Should start before you start coding
- Not the same as debugging
  - Happens before debugging
- Cannot guarentee correctness by testing
  - Only prove incorrectness
- Ideally, developer and tester should be different people (not in this course)

[8] If $1 $2 are not given, then they are blank if not supplied. Thus, defaults to previous behaviour

Chapter 5

Lecture 5[1]

5.1 Human Vs. Machine Testing

Human testing: Humans read code and look for failures.

- Code inspections

Machine Testing: run the program on selected inputs, check against specs.

- Can't test everything:
  - Need to choose carefully

Black-Box Testing: No knowledge of implementation.

White-Box Testing: Full knowledge of implementation.

Grey-Box Testing: Some knowledge of implementation.

- Start with black-box, supplement with white-box.
- Check various classes of input, eg:
  - Numberic ranges
  - Positive vs negative
  - Boundaries between ranges (edge cases)
    - Multiple simultaneous boundaries (corner cases)
  - Extreme cases
  - Intuitiona and experience

White-box

- Check all logical paths through the program
- Make sure every function runs
- Keep tests small and many of them to give clear paths to errors

[1] Jan 21, 2014

5.1.1 Questions Testing can Answer

- Does my program work (correct)?
  - Functional testing
- Did my change just break something?
  - Regression testing
- Is my program fast enough?
  - Performance testing
- Can I handle large and small inputs?
  - Volume testing

Part II

Module 2: C++

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

Notes: main must return in C++.

- stdio.h, printf still available in C++
  - We can't use them in this course
- Preferred: C++ I/O
  - header `<iosteam>`

```
std::cout << ___ << ____ <<
std::endl // end of line
```

- `using namespace std` lets you refer to `std::cout, std::endl` as just `cout, endl`
- `return` - returns a status to the shell
  - If not specified, main returns 0

5.2 Compiling C++ Programs

in shell:

```
g++ program.cc -o program
```

- program is the name of the executable, if not specified: a.out

To run it:

```
./program
```

Most C programs are valid C++ programs. (Review C as necessary).

5.3 C++ Input/Output

We've seen: `cout << stuff << endl;`
C++ provides 3 I/O steam objects:

1. `cout`: for printing to `stdout`
2. `cin`: for reading from `stdin`
3. `cerr`: for printing to `stderr`

5.3.1 I/O operators:

`<<` "put to" (output)

`>>` "get from" (input)

```
cout << x; // put x to stdout
cerr << x; // put x to stderr
cin >> x; // get x from stdin
```

- The operator points to the direction of information flow

Example. Get two numbers and add them

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

Notes: This ignores whitespace.

`cin >> x >> y;` looks for two int in stdin, ignoring whitespace, newlines, between x and y.

Q: What if input doesn't contian an int next?

- Statement fails, var is unassigned

Q: What if input is exhausted before we get 2 ints?

- Behaviour is the same

Q: How can we tell if a read has failed, or if the input steam is exhausted?

- If the read failed: then `cin.fail()` will be true.
- If EOF: `cin.eof()` and `cin.fail()` will both be true.
  - But not until an attempted read fails

Example. Read all ints from stdin and echo them one per line. Stop if a non-int is encountered or EOF is hit.

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

Note: There is an implicit conversion from cin to void*.

- Lets `cin` be used as a condition.

- ie:
  `if (cin)` $\implies$ true if `!cin.fail()`, false if `cin.fail()`

Example. Read all ints from stdin and echo them one per line. Stop if a non-int is encountered or EOF is hit.

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

Note: operator >> is the C right bitshift operator.

`a >> b` shfit a's bit representation by b slots to the right

Example. 21 = 10101

21 >> 3 = 00010 = 2

Can still do this in C++. But when the LHS is cin, >> is "get from".

Operator >> inputs:

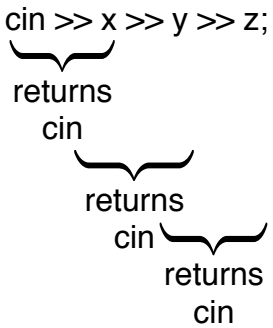
- cin (isteam)
- data (various types)

output?

- returns cin back to you (isteam)

This is why we can write:

```
cin >> x >> y >> z;
```



Example. Read all ints from stdin and echo them one per line. Stop if a non-int is encountered or EOF is hit. Version 3.0

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

Example. Read all ints from stdin and echo them one per line. Stop if a non-int is encountered or EOF is hit. Version 4.0

```
#include <iostream>
using namespace std;
int main(){
        int i;
        while (cin >> i){
                cout << i << endl;
        }
}
```

Chapter 6

Lecture 6[1]

6.1 Reading Ints

Read all ints and echo on stdout until EOF.
Skips non-integer input

```
int main(){
        int i;
        while (true){
                if (!(cin >> i)){
                        if(cin.eof()) break; // EOF
                        else { // bad int
                                cin.clear(); // clear the fail bit
                                cin.ignore(); // ignore current input char
                        }
                }
                else { // No fail
                        cout << i << endl;
                }
        }
}
```

6.2 Reading Strings

C++ provides type std::string

```
#include <iostream>
#include <string>
using namespace std;
int main(){
        string s;
        cin >> s;
        cout << s << endl;
}
```

- Only reads the first word ignoring whitespaces

Semantics: Reads the next character into a string, stopping at the next whitespace, and skipping leading whitespace.

[1] Jan 23, 2014

Q: What if we want the whitespace?

```
getline(cin, s);
```

Reads from current position to next new line into s.

```
int i=95;
cout << i << endl; // Prints 95
```

Q: What if we want to print 95 in hexadecimal?

Solution: Use I/O manipulators.

```
cout << hex << i << endl;
```

- `hex` here is a manipulator
  - Tells cout to print all subsequent values in hexidecimal

To go back to decimal:

```
cout << dec;
```

Others: See notes.

- Some require

```
#include <iomanip>
```

The steam abstraction applies to other sources of data.

Example. Files: read data from a specific file instead of stdin.
Use filesteam objects:

```
#include <fstream>
```

`ifstream` To read files

`ofstream` To write files

In C:

```
#include <stdio.h>
int main(){
        char s[255];
        FILE *f = fopen(suite.txt,r);
        while(true){
                fscanf(f, %s, s);
                if(eof(f) break;
                printf(%s\n, s);
        }
        fclose(f);
}
```

In C++:

```
#include <iostream>
#include <fstream>
using namespace std;
int main(){
        ifstream f("suite.txt");
        string s;
        while(f >> s) {
                cout << s << endl;
        }
}
```

- Declaring the ifstream opens the file.
- File is closed when the ifsteam object goes out of scope.

Imporant: fstreams take in c-style strings (`str.c_str()`)

Note:

- Anything you can do with cin, you can do with an ifsteam.
- Anything you can do with cout, you can do with an ofsteam.

Example. Reading and writing from strings

- Can attach a string to a string and read from/write to it.

```
#include <sstream>
```

`istringstream` for reading from a string

`ostringstream` for writing to a string

```
int lo=___,hi=____;
ostringstream ss;
ss << Enter a number between << lo <<  and  << hi;
string s = ss.str();
cout << s << endl;
```

Example. Converting a string to a number:

```
int n;
while(true){
        cout << Enter a number << endl;
        string s;
        cin >> s; // reading strings can't fail
        istringstream ss(s);
        if (ss >> n) break;
        cout << Not a number << endl;
}
```

6.3 Strings in C++

In C: arrays of chars, terminated by \0 (null)

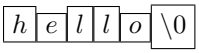
- Need to explicitly manage memory (allocated when strings get bigger, deallocate when done)
- Can overwrite the null terminator \0

In C++: strings

- Grow and shrink as needed (no memory management)
- Safer to manipulate

Example. string s = "Hello";

- "hello" is a c-style string



- Converted to a C++ string on initialization

6.3.1 String Operations

Equality: `s1 == s2`
`s1 != s2`

Comparison: `s1 <= s2`

Length: `s1.length()`

Extract chars: `s1[0], s1[1], s1[2]...`

Concatenation: `string s3 = s1 + s2`

More details: see course notes

Chapter 7

Lecture 7[1]

7.1 Default Arguments

```
void printSuiteFile(string name=suite.txt){
        ifstream f(name.c_str()); //Stream inializer requires a c-style string
        // and string.c_str() produces one
        string s;
        while(f >> s) cout << s << endl;
}
printSuiteFile(suite2.txt); //Prints suite2.txt
printSuiteFile(); //Prints suite.txt
```

Note: Optional parameters must be last.

7.2 Overloading

In C:

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

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

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

This is called overloading.

- Compiler uses the number, types and order of arguments to pick the right "neg" to use.
- Overloads must differ in number or types of arguments
  - It does not matter if they differ on the return type

[1] Jan 28, 2014

- We've seen this already
  - <<, >> are overloaded
  - same thing for +
- Behavior changes based on types of arguments.

7.3 Declaration Before Use

- Cannot use something before it is declared
- Mutual recursion?

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

- Fails: odd is not defined a the point where it is used.

Solution: Introduce a forward declaration

```
bool odd(unsigned int n); //forward declaration
bool even(unsigned int n){
        if(n==0) return true;
        return odd(n-1);
}
bool odd(unsigned int n){
        if(n==0) return false;
        return even(n-1);
}
```

7.3.1 Important distinction between declaration and definition.

Declaration: Only asserts the existence of the entity. e.g bool odd(unsigned int n);

Definition: All details, implementation, including allocation of space (if necessary)

Example.

```
bool odd(unsigned n){
        ...
}
```

Note: Declaration (not definition) before use.

- An entity can be declared several times, but defined only once.

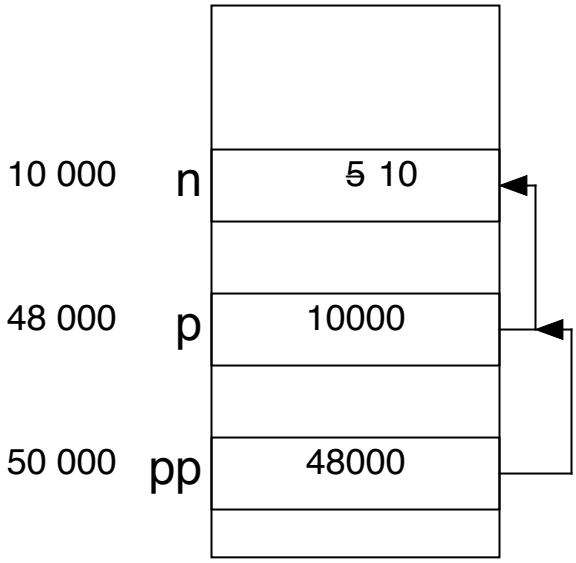
7.4 Pointers

Recall:

```
int n
int *p=&n;
```

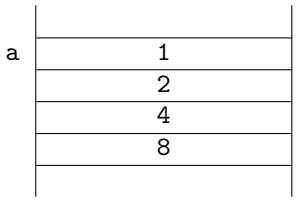
p contains the address of n

```
cout << p << endl; // hex#
cout << *p << endl; // 5
int **pp; // ptr to ptr to int
pp = &p;
**pp = 10; // changes 5 to 10
```



7.5 Arrays

```
int a[] = {1,2,4,8}
```



The name of the array is short hand for the address of the first element (`&a[0]`)

`a==&a[0]`

∴ `*a==a[0] // 1st element`

`*(a+1)==a[1] // 2nd element`

7.6 Structs

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

- Don't forget the ';' after }
  - Artifact from C: can declare variables before ';'

Why is this wrong?

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

- How big should a node structure be?
  - size(node) + size(int) = ∞

7.7 Constants

```
const int maxGrade = 100;
```

- Must be initialized.
- Should make as many things const as you can.

```
const Node n1 = {5, NULL};
```

- Can't change the node

Q: What does this mean?

```
const int *p = &n;
```

This is a ptr to a const int. p can be reassigned to another in but cannot change the value though p.

e.g p=&m
But *p cannot be reassigned though the pointer.
eg. *p=10 is wrong and won't compile.
Compare

```
int * const p = &n;
```

- p is a constant ptr to a (non-const) int
  - Can't change where p points

e.g. p = &x; is wrong

  - Can change the data p points to

Const ptr to const int

```
const int *const p = &n;
```

7.8 Parameter Passing

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

- Parameters in C++ are passed by value
  - inc gets a copy of x, increments the copy
  - Original unchanged

If a function must modify it's argument: pass a pointer

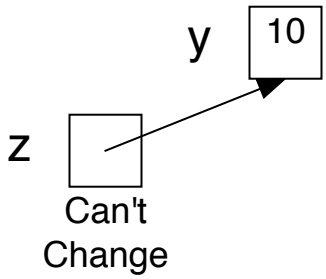
```
void inc(int *n){
        *n+=1;
}
int x = 5;
inc(&x);
cout << x << endl; // 6
```

- x's address is passed by value inc changes data at that adress
  - visible to caller.

Q: Why `cin >> x` and not `cin >> (&x)`?

A: C++ provides another ptr like type: The reference.

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



References are like const ptrs with automatic dereferencing.

```
z=12; // (NOT *z=12;_ y is now 12;
int *p = &z; // gives the address of y.
```

Chapter 8

Lecture 8

Recall: References[1]

```
int y=10;
int &z=y;
z=12; // now y==12
int *p=&z; // gives the address of y
```

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

- reference is like a constant pointer with automatic dereferencing

8.1 Things you can't do with references

1. Leave them uninitialized
   - eg `int &x;` wrong
   - `int &x=5;` wrong, it doesn't have an address
   - `int &x = y + y;` wrong, it doesn't have an address it's an expression

   (a) References must be initialized to something that has an address

2. Create a pointer to a reference
   - `int &*x;` wrong, x is a pointer to a reference to an int
   - But you can create a reference to a pointer:
     - `int *&x = ___;` OK

3. Create a reference to a reference
   - int &&x=____; wrong,

4. Create an array of refrences
   - int &x[3] = {y,y,y}

Q: What can you do with references?

- Use them as function parameters.

[1] Jan 30, 2014

```
void inc(int &n){ //n is a constant pointer to the arg(x)
        n+=1; //no ptr dereference here
}
int x=5;
inc(x); //Note: no &
cout << x << endl; //6
```

- n is a constant pointer to the arg(x), therefore changes to n affect x

Q: So why does `cin >> x` work?

- Because it takes x by reference.

```
isteam &operator>>(isteam &in, int &data);
```

8.2 Pass-by-value

```
(int f(int n){})
```

- Copies the argument
  - $\therefore$ if the argument is big, the copy is expensive

```
Struct ReallyBig{
        ...
};
int f(ReallyBig rb){
        ...
}
```

Pass by reference:

```
int g(ReallyBig &rb){
        ... // passes an alias - efficient
}
```

- Passes an alias
  - Efficient
  - But changes to rb are visible in the caller.

```
int h(const ReallyBig &rb){
        ...
}
```

- Efficient (no copying)
- Parameter cannot be changed (const)

Advice: Prefer pass-by-reference-to-const over pass-by-value for anything larger than an int.

Also:

```
int f(int &n){
        ...
}
int g(const int &n){
        ...
}
f(5); // it fails, because you can't create a refrence of an int
// you can't initialize to something without an address
f(y +y); // wrong
g(5); // This is ok, since n can never be changed, the compiler allows this
g(y+y); // This is also ok
```

8.3 Dynamic Memory Allocation

Recall from C:

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

- malloc/free are available in C++, but we **DON'T USE THEM**.

Instead: use new/delete

- Type aware, less error-prone

Example. Linked list structure from before

```
struct Node{
        int data;
        Node *next;
};
Node *np = new Node;

delete np;
```

8.3.1 Allocating arrays:

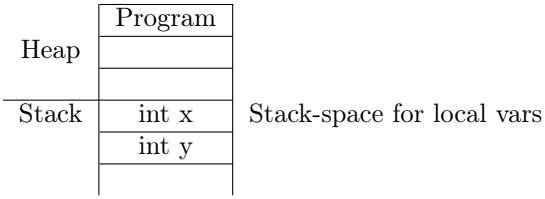
```
cin >> n;
int *arr = new int [n];
for(int i=0, i<n; ++i){
        cin >> arr[i];
}
delete []arr;
...
```

- Use delete [] to delete arrays.
- Use delete to delete non-arrays.

Remark. Difference between i++ and ++i

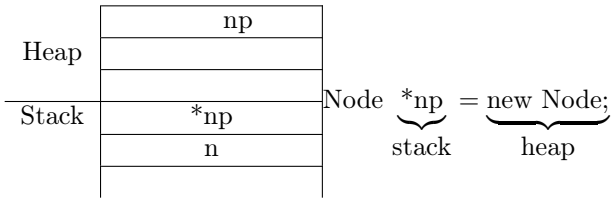
```
i=4
j=++i; // i and j are now 5
```

8.4 Stack vs Heap Allocation



- Vars disappear when they go out of scope (stack is popped)

```
Node n; // on the stack
Node *np = new Node;
```



- Heap-allocated data lives on after the ptr goes out of scope (stack is popped).

Example.

```
Node getMeANode(){
        Node n;
        return n; // n has to be copied on return
}
Node *getMeANode(){
        Node n;
        return &n; // Worse kind of mistake you can make
        // memory is in the stack and can be overwritten
        // returns a pointer to stack-allocated data, which is dead on return
}
```

To get rid of this problem, create the object on the heap

```
Node *getMeANode(){
        Node *np = new Node;
        return np;
        // np is on the stack
        // Node is on the heap
}
```

8.5 Operator Overloading

Example. Want to do linear algebra

```
struct Vec{
        int x,y;
};
Vec operator+(const Vec &v1, const Vec &v2){
        Vec v;
        v.x = v1.x + v2.x;
        v.y = v1.y + v2.y;
```

```
        return v;
}
Vec operator*(const int k, const Vec &v1){
        Vec v;
        v.x = k*v1.x;
        v.y = k*v1.y;
        return v;
}
Vec v={1,2};
Vec v2=v+v;
Vec v3=2*v2;
Vec v4=v3*3; // won't compile because of the order
```

Accounting for this:

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

Chapter 9

Lecture 9[1]

9.1 Operator Overloading

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

9.1.1 Overloading << and >>

Example.

```
struct Grade{
        int theGrade;
};
ostream &operator<<(osteam &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;
}
```

9.2 The Preprocessor

Transforms the program before the compiler sees it.

`#_____` Preprocessor directive

`#include <iostream>` says "insert the contents of file iostream here"

`#include <_>` look in standard include directory

`#include "_"` look in the current directory

[1] Feb 4, 2014

Including Old C headers: New naming convention

Example. Instead of `#include<stdio.h>` prefer `#include<ctdio>`

`#define VAR VALUE` Sets a preprocessor

- Then all occurences of VAR in the sourcefile are replaced with VALUE.
- Application: inline constants

Example.

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

[2]

Use const definitions instead

```
#define ever ;;
for (ever){
        ...
}
```

Special case

```
#define FLAG
```

- Sets the variable FLAG
- Value is the empty string

Defined constants are useful for conditional compilation.

Example.

```
#define Unix 1
#define Windows 2
#define OS Unix (or Windows)
#if OS == Unix
int main(){ //Removed if the OS != Unix
#elif OS == Windows //Removed if the OS != Windows
int winMain(){
#endif
...
}
```

- This is done before the compiler sees it

Special case:

```
#if 0 //Always false: all inner text is removed before it gets to the compiler
...
#endif
```

- Heavy-duty "comment out"
  - (#if's nest)

[2] Transformed to `int x[10];`

```
#ifdef NAME
```

- True if NAME has been defined

```
#ifndef NAME
```

- True if NAME has not been defined

Can also define symbols via compiler arguments

File: define.cc

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

On Command line:

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

- Now compiles

Example. Debugging code using this (debug.cc)

```
#ifdef DEBUG
cout << Setting = 1 << endl;
#endif
int x =1;
++x;
#ifdef DEBUG
cout << x is now << x << endl;
#endif
cout << x << endl;
```

On Command line:

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

- We can also set levels of debugging for testing code (different debug variables)

9.3 Separate Compilation

Split programs into modules, which provide:

Interface: What the client sees

- type defintions, function, headers
- .h file

Implementation: Full definition for all function .cc file.

Algorithm 9.1 vector.h

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

Algorithm 9.2 main.cc

```
#include vector.h
...
int main(){
        Vec v = {1,2};
        v=v+v;
}
```

Algorithm 9.3 vector.cc

```
#include vector.h //So we get the struct definition
Vec operator+(const Vec &v1, const Vec &v2){
        ...
}
```

To compile:

```
g++ *.cc
```

or

```
g++ main.cc vector.cc
```

9.3.1 Compiling separately.

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

`-c` Compile only, don't link, don't build the executable

- Outputs and object file (.o)

to link:

```
g++ vector.o main.o -o main
```

-o detailing you want to name te executable

Q: What if I want a global variable in a .h file?

Example. abc.h

```
int global;
```

Every file that includes abc.h defines a separate "global".

- Program will not link

Chapter 10

Lecture 10[1]

Recall:

Example. abc.h

```
int global;
```

Every file that includes abc.h defines a separate "global".

- Program will not link

Solution: abc.cc

```
int global; // because you never include .cc files
```

abc.h

```
extern int global; // Declaration but not defined
```

Suppose we want to write a linear algebra module.
(~/cs246/1141/lectures/c++/separate/example3)

- It does not compile,

linalg.h

```
#include vector.h
...
```

linalg.cc

```
#include linalg.h
#include vector.h
...
```

- Won't compile:
  - main.cc , linalg.cc incude vector.h, linalg.h, linalg.h includes vector.h
  - $\therefore$ vector.h is included twice
  - $\therefore$ struct Vec is defined twice
- Need to prevent files from being included more than once.

Solution: #include guard

Algorithm 10.1 #include guard: vector.h

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

- First time file vector.h is included: symbol `__VECTOR_H__` is not defined, so file is not included
- Subsequently: `__VECTOR_H__` is defined, so contents of vector.h are suppressed.
- Always put #include guards in .h files.

(~/cs246/1141/lectures/c++/separate/example4)

- **NEVER** put `using namespace std;` in .h files
  - The using declaration is forced on any user who includes the file.
- **ALWAYS** refer to `cin, cout, endl, string,` etc as `std::cin, std::cout, std::endl, std::string`, etc. in .h files.

10.1 Classes

The innovation of OOP:

> You can put function inside of structs.

Example.

```
struct Student{
        int assns, mt, final;
        float grade(){
                return assns * 0.4 + mt * 0.2 + final * 0.4;
        }
};
...
Student billy = {60, 70, 80};
cout << billy.grade() << endl;
```

A class: is a struct type that can contain functions.

C++ has a class keyword

- We will use it later.

An object: is an instance of a class.

Example.
`Student`[2] `billy`[3] `= {60, 70, 80};`

- class object
- Function grade is called a **member function** (or **method**)

[1] Feb 6, 2014
[2] `Class`
[3] `Object`

Q: What do `assn, mt, final` mean inside of grade(){···}?

A: They are the fields of the current object

- The object upon which the method was invoked.

`billy.grade()` Uses billy's assns, mt, final

Formally: Methods take a hidden extra parameter called `this`. `this` is a pointer to the object upon which the method was invoked.

`billy`[4]`.grade();`
Can write:

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

`p->x` ≡ `(*p).x`

10.2 Initializing Objects

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

Better: write a method that does initilization: a constructor.

```
struct Student{
        int assns, mt, final;
        float grade(){}
        student(int assns, int mt, int final){
                this->assns = assns;
                this->mt = mt;
                this->final = final;
                // could also just use a different name for input variables
        }
};
Student billy(60,70,80);
```

or

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

10.2.1 Heap Allocation

```
Student *pbilly = new Student(60,70,80);
...
delete pBilly;
```

10.2.1.1 Advantages of ctors:

- Overloading, default parameters
- Full power of C++ in function body.

[4] `this==&billy`

```
struct Student{
        int assns, mt, final;
        ...
        student(int assns=0, int mt=0, int final=0){
        this->assns = assns;
        this->mt = mt;
        this->final = final;
        // could also just use a different name for input variables
        }
};
Student bobby(70,80); // 70,80,0
Student newGuy; // 0,0,0
```

- Note that the constructor was still called.

Note: You do not say:

```
Student newGuy();
```

- This is a function header, declaring a function newGuy
  - Declares a function header

Note: Every class comes with a built-in default (ie. zero - argument) ctor

- Which just calls default ctors on any members that have them.

Example. Vec v; // default ctor
But the built in default ctor goes away as soon as you provide a ctor.

```
struct Vec{
        int x,y
        Vec(int x, int y){
                this->x=x;
                this->y=y;
        }
};
Vec v; // won't compile, no default ctor anymore
// needs a default value in ctor
Vec v(1,2); // Ok
```

Note: Lose C-style struct initializers when you declare a ctor.

```
Vec v = {1,2}; // won't compile
```

Chapter 11

Lecture 11[1]

RECALL: constructors

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

Q: What if a struct contains constants or referneces?

```
struct MyStruct{
        const int myConst;
        int &myRef;
        // These need to be initialized
};
```

- So lets initialize them:

```
int z;
struct MyStruct{
        const int myConst = 5;
        int &myRef = z;
        // Won't compile, no initializers on fields
};
```

- Besides each instance of MyStruct gets it's own myConst, myRef: why should these all be the same?

Q: Can we initialize them in the ctor body?

A: NO: too late, need to be initialzed before then.

Need a place to do initialization before the constructor body runs.
When an object is created:

1. Space for the object is allocated.
2. Field initialization: default constructors are called on members that have them.

[1] Feb 11, 2014

3. Ctor body runs.

- Need to put our initialization here in 2.

How? The member initialization list.

```
struct MyStruct{
        const in myConst;
        int &myRef;
        MyStruct(int c, int &r): myConst(c), myRef(r){
        } //after the colon: initialization before the ctor body runs
};
```

- Does have to be just constants and references:

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

- Advantages:
  - Can be more efficient than setting fields in the ctor body, in some cases
    - Step 3 isn't called, we save on default ctor time for each field
    - Otherwise: run default field ctor, then reassign in ctor body.
  - Can use consts
  - Can use the names twice.

Note: Fields are initialized **in the order in which they are declared in the struct**. Even if the initializer list orders them differently.

Now consider

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

Q: How does this initialization happen? The Copy Constructor

- The copy constructor is for when constructing one object as a copy of another.

Note: Every class comes with:

- A default ctor (unless you write any ctor)
- A copy ctor: (copies field for field)
- A copy assignment operator.
- A destructor.

11.1 Building your own copy ctor:

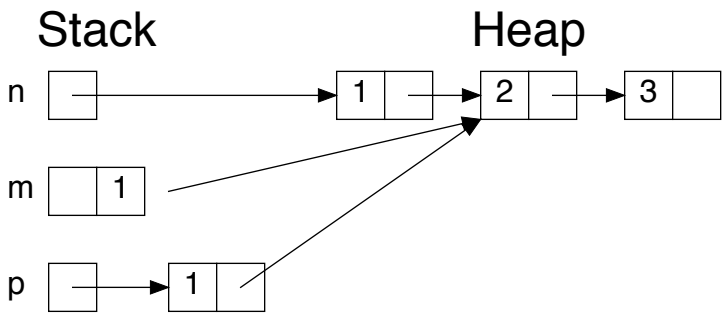
```
struct Student{
        int assns, mt, final;
        Student(__): ___{}
        Student(const Student &other): assns(other.assns), mt(other.mt), final(other.final)
};
```

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

A: When we are using dynamic memory.

```
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){}
};
Node *n = new Node(1, new Node(2, new Node(3,0)));
Node m = *n;
Node *p = new Node(*n);
```

- Both of these are calling the copy ctor



- Simple copy of fields
  - Only the first node is copied
- Called a shallow copy

If you want a deep copy, which copies the whole list, you must write your own copy ctor.

```
struct Node{
        int data;
        Node *next;
        ...
        Node(const Node &other):data(other.data),next(other.next? new Node(*other.next): 0)
        // this calls recursion on the rest of the list
};
```

The copy ctor is called:

1. Whenever an object is initialized with another object
2. Whenever an object is passed by value.

3. When an object is returned from a function.

Be careful with ctors that take a single parameter.

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

- Single-argument ctor create implicit conversions.

Example.

```
Node n(4);
```

- But also

```
Node n = 4;
```

- Implicit conversion, from into Node

```
int f(Node n){}
f(4); // Works: 4 is silently converted to Node(4).
string s = Hello; // Hello is a const char *
```

Can be dangerous

Example. Accidentlly pass an into to a function expecting a Node

- Silent conversion
- No error message

$\Longrightarrow$ potential errors not caught

Good idea: disable the implicit conversion: decalre teh ctor explicit.

```
struct Node{
        ...
        explicit Node (int data): data(data), next(0){}
};
Node n = 4; // Wrong, won't compile
```

Chapter 12

Lecture 12[1]

12.1 Destructors

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

Classes come with a destructor (doesn't do much - calls destructor for all fields)

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

Q: If `np` goes out of scope?

- Memory leak.

The things pointing to were never freed.

- pointer `np` (stack-allocated) is reclaimed
- list is not

If we say `delete np;`

- calls *np's destructor which doesn't do anything.

Write a destructor to ensure the whole list is cleaned up.

```
struct Node {
        ...
        ~Node() {
                ...// delete fields
                delete next; // delete null is safe.
        }
};
```

Now `delete np`, frees the whole list.

[1] Feb 13, 2014

12.2 Separate Compilation for Classes

Algorithm 12.1 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 data);
};
#endif
// explicit is for single argument constructors to disable the implicit conversion
```

Algorithm 12.2 node.cc

```
#include node.h
Node::Node (int data, Node *next):data(data), next(next) {}
Node::Node (const Node &other):data(other.data), next(other.next) {}
Node::Node (int data):data(data), next(0) {}
```

:: is use to say it these belongs to struct node.

:: called the scope resolution operator

`Node::___` means `___` in the context of struct Node :: is kind of like .

12.3 Assignment Operator

Algorithm 12.3 When default ctor, copy ctor, and assignment is called

```
Student billy(60, 70 80);
Student bobby = billy; // uses copy constructor
Student jane; // default ctor - even if its the builtin
jane = billy; // copy all of billys fields into jane - copy but not
    construction - uses assignment operator
```

Algorithm 12.4 node: Assignment under heap

```
struct Node {
        int data;
        Node *next;
        Node(...) {}
        Node(const Node &other) {}
        Node &operator = (const Node &other) {
                data = other.data;
                delete next
                next = other.next ? new Node(*other.next) : 0;
                return *this;
        } // mutates an object that DOES previously exists -
        // careful with setting what it normally existed
};
```

`a=b`

- Both command and expression
- Value of the expr is the value assigned

`a = b = c = 0`

12.3.1 Being Careful of Self-Assignment

- The Assignment Operator is the most Dangerous One.

Q: Why?

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

- What values are copied in this assignment?

When writing `operator =` Check for self assignment.

Algorithm 12.5 Checking for Self-Assignment (incomplete): node

```
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;
}
```

- Consider if `other/this` is another elements in the same list - still a problem
- What if you're short on memory, that new Node might fail?
  - You have a node with a dangling next pointer

Algorithm 12.6 Checking for Self-Assignment (fixed): node

```
Node &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;
}
```

- If new fails, node will still be in a valid state

Algorithm 12.7 Checking for Self-Assignment (Alternative-copy+swap idiom): node

```
struct Node {
        // exchanges this data, with temp data
        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; // copying other
                swap(tmp); //this: copy of other
                // tmp will go out of scope, destructor will be called
                // then my old data gets destroyed
                return *this;
        }
};
```

12.4 Rule of 3

If you need to write a custom version of any of:

- copy ctor
- `operator=`
- dtor

then you usually need a custom version of all three.

Note: `Operator=` is a member function. not a standalone function.

- When an operator is declared as a member function, this plays the role of the LHS arg.

Chapter 13

Lecture 13[1]

When an operator is declared as a member, `this` plays the role of the LHS arg.

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

Q: How do we implement k * v?

- Must do it as a standalone!

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

Q: What about I/O operators?

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

Q: What's wrong with this?

- Makes Vec the LHS arg, not the RHS

- Use as `v << cout`
  - confusing

[1] Feb 25, 2013

So, define `<<, >>` as standalone function.

Certain operators that must be members:

- operator=
- operator[]
- operator->[a]
- operator()[b]
- operator T[c]

[a] if you want your operators to act like pointers
[b] have our object act like a function
[c] T is a type

13.1 Arrays of Objects

```
struct Vec{
        int x,y;
        Vec(int x, int y): x(x), y(y){}
};
Vec vectors[3]; // won't compile
Vec *moreVectors = new Vec[10]; // won't compile
```

Why?
- Can't initialize because no default cont, i.e. zero-arg ctor)

To fix:
On the stack:

```
Vec vectors[3] = {Vec(1,2), Vec(3,4), Vec(5,6)};
```

On the heap: If you want a heap array of objects, you need a default ctor.

Option 1: Provide a default ctor

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

Option 2: Array of pointers

```
Vec ** v = new Vec*[10];
```

13.2 `const` Methods

Q: What is a const object?

- Can't change the fields?

Q: Can we call methods on const object?

Issue: What if the method changes fields?

A: Yes, we can call methods that promise not to change fields.

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

Remark. The method `grade()` doesn't modify fields so declare the method `const`.

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

Now, consider: Want to collect usage statistics on Student objects.

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

Want to be able to change `numMethodCalls`, even if the object is `const`.

- Declare the field `mutable`.
  - Mutable fields can be changed, even if the object is `const`.

```
struct Student{
        int assns, mt, final;
        mutable int numMethodCalls;
        float grade() const {
                ++numMethodCalls;
                return ____;
        }
};
```

13.3 SE Topic: Design Patterns: Singleton Pattern

Known solutions to common problems.

Example. Singleton pattern:
We have a class C, and we want to ensure that only once instance of C is created, no matter how many times we may attempt to create an instance.

Example. Database class, error log class.

- If there is only one of these

13.3.1 Singleton Example

Write a program to track my finances.

2 Classes:

1. Wallet (singleton, I only have one)
2. Expense: Each has access to my wallet :(

13.3.2 C++ Implementation: static members

In C:

1. A static variable: other modules can't see it
2. In a function: a variable has the value that it was when called before

C++:

static members: associated with the class itself, not with any specific instance (object).

Example. Want to track how many Student objects are created.

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

`int Student::`[2]`numInstances=0; // in .cc file`

- Static fields must be defined external to the class, at file level

13.3.3 Static Member Function

Static Member functions: don't depend on the specific instance (no `this` parameter)

- Can only access static fields, and call other static functions

```
struct Student{
        ...
        static int numInstances;
        ...
        static void printNumInstances(){
                cout << numInstances << endl;
        }
};
inst Student::numInstances=0;
...
Student billy(60,70,80);
Student bobby(70,80,90);
Student::printNumInstances(); \\can do the billy.printNumInstances() way
```

Back to Singleton...

[2] `Associated with the class, and not an object`

Algorithm 13.1 wallet.h

```
struct Wallet{
        static Wallet *instance;
        // The pointer is so that we don't need to save the wallet object unless we want to
        // only 1 instance
        static Wallet *getInstance();
        //Fetch the instance initialize if necessary
        Wallet();
        int money;
        void addMoney (int amt);
};
```

Chapter 14

Lecture 14[1]

Algorithm 14.1 Wallet.h

```
struct Wallet{
        static Wallet *instance;
        static Wallet *getInstance();
        Wallet();
        int money;
        void addMoney (int amt);
        // Added for Singleton for deleting
        static void cleanup();
};
```

Algorithm 14.2 Wallet.cc

```
Wallet *Wallet::instance=0; // must define static members
Wallet *Wallet::getInstance(){
        if (!instance){
                instance = new Wallet;
                // Added for Singleton for deleting
                atexit(cleanup);
        }
        return instance;
}
Wallet::Wallet():money(0){}
void Wallet::addMoney(int amt){money+= amt;}
// added
Wallet::cleanup(){
        cout << cleaning up... << endl;
        delete instance;
}
```

Q: When do we delete the wallet instance?

- How do we know when all clients are done with it?

[1] Feb 27, 2014

Algorithm 14.3 Expense.h - Each has access to a Wallet (the same one)

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

Algorithm 14.4 Expense.cc

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

Function `atexit` (`<cstdlib>`) takes a function (taking and returns void) and runs it when the program terminates. (`static void cleanup()` here)

Q: Can't we just create own wallets by called the ctor?

14.1 Encapsulation

Want to control the way objects are used.

- Client should view them as black boxes: capsules
- Seal away implementatoin
- Only interact by provided methods

Example.

```
struct Vec{
        Vec (int x, int y);
private:
        int x,y;
public:
        Vec operator+(___){___}
        ...
}:
```

`private:` can only be accessd inside Vec

`public:` Anyone can see

- By default, visibility in structs = public.

In general: fields shold be private, only methods should be public.

- Better to have the default visiblity to be private.

Algorithm 14.5 main.cc

```
1  int main(){
2          Expense mortage(mortage, 1000);
3          Expense car(car, 300);
4          Expense insurance(ins, 200);
5          Wallet *myWallet = Wallet::getInstance();
6          Expense paycheque(pay, -2000);
7          paycheque.pay();
8          mortgage.pay();
9          car.pay();
10         car.pay();
11         insurance.pay();
12 }
```

- Switch from `struct` to `class`.

Example.

```
struct Vec{
        int x,y;
public:
        Vec (int x, int y);
        Vec operator+(___){___}
        ...
};
```

- Only difference between class and struct is default visibility: public in struct, private in class.
- Keep fields private!

Public fields $\implies$ users can manipulate them any way they want.

- Can't maintain class invariants.
- Can't replace implementatoin without breaking client code.
- If you want to provide field access:
  - Provide accessor methods:

Algorithm 14.6 Accessor Methods

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

If you want to let clients change fields, provide mutator methods:

Algorithm 14.7 Mutator Methods

```
class Vec{
        int x,y;
public:
        ...
        int setX(int v) {x=v;}
        int setY(int v) {y=v;}
};
```

Now suppose we don't want to provide accessors, but we do want to provide `operator<<`

Issue: `operator<<` is not a member, but needs x and y, and can't see them.

Solution: make `operator<<` a friend function

Algorithm 14.8 Friend Function: Vec.h

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

Algorithm 14.9 Friend Function: vec.cc

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

`friend` function can see all of a class members, but are not part of the class.

- Give your classes as few friends as possible.
  - Weakens encapsulation when you declare friends.

14.2 System Modeling

Identify major abstractions in a system and relationships among them.

Popular Standard: UML (Unified Modelling Language)

14.2.1 Modelling a Class

Name	Vec
Fields (optional)	- x: Integer - y: Integer
Methods (optional)	+ getX: Integer + getY: Integer

Visiblity: - means private

\+ means public

Chapter 15

Lecture 15

15.1 Compositon

```
class Vec{
        int x,y,z;
public:
        Vec(int x, int y, int z): x(x), y(y), z(z){}
};
class Plane{
        Vector v1, v2;
public:
        Plane(){}
        ...
}:
Plane p; // can't initialize v1,v2
```

When an object is created:

1. Space is allocated
2. Default ctors/init list for fields in decl order.
3. ctor body

When an object is destroyed:

1. Dtor body runs
2. Dtor runs for fields in reverse declaration order
3. Space is deallocated.

```
class Plane{
        Vector v1, v2;
public:
        Plane(): v1(1,0,0), v2(0,1,0){...}
        ...
}:
Plane p; // can't initialize v1,v2
```

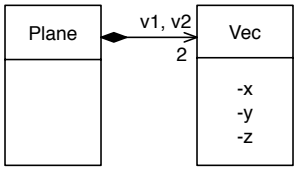
Embeddin one object (Vec) inside another (Plane) is called composition relationship is known as "owns-a" relationship.

A Plane object owns a Vec object (in fact, 2 of them).

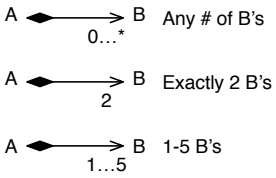
If A owns a B, then typically,

- B has no identify outside of A.
- If A is destroyed , B is destroyed.
- If A is copied, B is copie (deep copies).

In UML:



Multiplicities



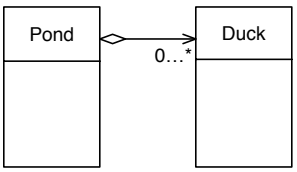
$$\begin{aligned} 0..* &= \text{any } \# \text{ of B's} \\ 2 &= \text{Exactly 2} \\ 1..5 &= 1-5 \text{ B's} \end{aligned}$$

15.2 Aggregation

Compare car parts in a ca("owns-a") vs. car parts in a catalogue.
Catalogue contains the parts, but the parts have their own existence.
Thsi is a "has-a" relationship (aggregation).
If A has a B, then typically:

- If A is destroyed, B lives on
- If A is copied, B is not (shallow copies).

Example. Parts in a catalogue, ducks in a pond.
Aggregation in UML:



15.2.1 Typical Implementation:

Pointer fields:

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

class Catalogue{
        Part *p;
        ...
};
```

15.3 Inheritance

You want to track your collection of books

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

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

```
class CSBook{
        string title, author;
        int numPages;
        string language;
        ...
};
```

For comic books, need the name of the hero

```
class Comic{
        string title, author;
        int numPages;
        string hero;
        ...
};
```

OK, but doesn't capture the relationship among these classes.
How would we create an array (or linked list) that contains a mixture of all 3?

1. Use a union (from C):

   ```
   union Booktypes{Book *b; CSBook *csb, Comic *c};
   BookTypes myBooks[20];
   ```

2. Array of void * (from C):

   - Both of these subvert the type system.

3. Observe: CSBooks, Comics, are kinds of books- books with extra features.

   In C++: Inheritance

```
// Base Class or Super Class
class Book{
        string title, author;
        int numPages;
Public:
        Book(string title, string author, int numPages);
        ...
};

// Derived classes or subclasses

class CSBook: public Book{
        string language;
public:
        ...
};
class CSBook: public book{
        string hero;
public:
        ...
};
```

- Derived classes **inherit** fields and methods from the base class.
  - Any method that can be called on a book, can be called on a CSBook or Comic

Q: Who can see these members?

- Title, author, numPages: private in Book
  - Outsiders can't see them

Q: Can CSBook see them?

A: No, even subclasses can't see them.

Example. CSBook b;
Can't acces b.author (it's private), but can call, b.getauthor() (if it's public).

Q: How do we initialize CSBook if we can't even see Book's fields?

```
class CSBook: public Book{
        string language;
public:
        CSBook(string title, string author, int numPages, string language):
                title(title), author(author), numPages(numPages), language(language){}
                // won't compile
        ...
};
```

Doesn't work for 2 reasons:

1. title, author, numPages are not accessible in CSBook.
   - Once again: when an object is created:
     (a) Space is allocated

(b) Superclass part is constructed.[1]
(c) Field constructors
(d) ctor body

2. And step doesn't work because there is no default constructor for book.

To solve both problems:

- Invoke Book's ctor in CSBook's init list.

```
class CSBook: public Book{
        ...
public:
        CSBook(string title, string author, int numPages, string language):
                Book(title,author,numPages), language(language){}
}; // works now
```

If the superclass has no default ctor, you must invoke a ctor fro the super class explicitly in the subcalss init list.

- If you want subclasses to have acces to superclass members. Use `protected` visibility.

```
class Book{
protected: // accessiable  to book and it's subclasses
        string title, author;
        int numPages;
public:
        ...
};
```

It allows you to do the following:

```
class CSBook: public Book{
        string language;
public:
        void addAuthor(string newAuthor){
                author += newAuthor;
        }
};
```

- Not a good idea to give subclasses unlimited access to fields.
  - Chips away at invariance, same problem as friends

Better: Make fields private, but provide protected accessors.

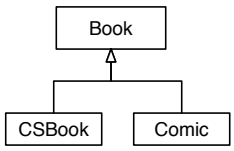
```
class Book{
        string title, author;
        int numPages;
protected:
        string getTitle() const;
        void setAuthor(string a);
public:
        ...
};
```

[1] new

- Relationship among Book, CSBook, Comic is called "is a" relation.

Example. A CSBook is a Book.
A Comic is a Book

In UML:



Now consider method isItHeavy():

Books: >200pages

CSBook: >500 pages

Comic: >30 pages

Chapter 16

Lecture 16[1]

Recall last day:

- When is a book heavy?
  - for ordinary books, >200 pages
  - for CSBooks, >500 pages
  - for ComicBooks, >30 pages

```
class Book{
        ...
public:
        ...
        bool isItHeavy() {return numPages >200;}
};
class CSBook: public Book{
        ...
public:
        ...
        bool isItHeavy() {return numPages >500;}
};
class ComicBook: public Book{
        ...
public:
        ...
        bool isItHeavy() {return numPages >30;}
};
Book b(A Small Book, A Small Man, 50);
ComicBook cb(A Big Comic, A Big Man, 40, Superman);
cout << b.isItHeavy(); // false
cout << cb.isItHeavy(); // true
cout << endl;
```

Now, since we have an "is-a" relationship (inheritance), we can do this:

```
Book b2 = ComicBook(A Big Comic, A Big Man, 40, Superman);
```

Q: What does b2.isItHeavy() return?

[1] Mar 6, 2014

```
b.isItHeavy() // True or False?
```

Which `isItHeavy()` runs? Book's or ComicBook's.

Answer: No, `b` is not heavy. It is `Book::isItHeavy ()`that runs.

Why? Consider

Book	**title**
	author
	numpages

ComicBook	**title**
	author
	numpages
	hero

```
Book b = ComicBook(A Big Comic, A Big Man, 40, Superman);
```

- Tries to fit a ComicBook object where there is only room for a book object.

What happens?

- ComicBook is **sliced**: hero field is chropped off, ComicBook coherrced into a book.
- Book b = ComicBook(____); converts the ComicBook into a Book and Book:isItHeavy runs.

When accessing objects though pointers slicing is unnecessary and doesn't happen:

```
ComicBook cb(____, __, 40, ___);
Book *pB = &cb;
ComicBook *pCB = &cb;
cout << pCB->isItHeavy() // true
cout << pB->isItHeavy() // false
cout << endl;
```

- The same object, cb, calls a method dependent on how it is accessed, through ComicBook ptr ComicBook's `isItHeavy()`, or Book ptr Book's `isItHeavy()`

Compiler uses the type of the pointer (or reference) to decide which class function to run. It does not consider the actual type of the object.

Means: A comic book only acts as a like a comicbook, when a ComicBook ptr (ref) points at it.

- May not be what we want.

Q: How do we make comicbook act like a comicbook, even when pointed to by a book pointer?

A: Delcare the Method **virtual**.

16.1 Virtual Methods

```
class Book{
        ...
public:
        ...
        virtual bool isItHeavy() {return numPages >200;}
        // only needs to be declared virtual in the struct
};
```

CSBook and ComicBook are unchanged.
They don't change, only book does.

```
ComicBook cb(____, __, 40, ___);
Book *pB = &cb;
ComicBook *pCB = &cb;
cout << pCB->isItHeavy() // true
cout << pB->isItHeavy() // true
cout << endl;
```

- Now comicbook:isItHeavy() runs in both cases.

Virtual Method: Choose which class method to run based on actual type of the object at run time.

Remark. Virtual changes based on runtime virtual affects only when we have pointer * or reference &

- If you make everything virtual, your program will slow down dramatically.
- At one point Java was considered slow cause everything is virtual.

Virtual Methods: Choose te method which class method to run based upon actual object type at runtime.

Example. My book Collection

```
Book *collection[20];
        ...
        for (int i=0, i<20, ++i){
                cout << collection[i]->isItHeavy() << endl;
                // uses Book:isItHeavy() for Books
                // uses ComicBook:isItHeavy() for ComicBooks
}
```

- Accommodates multiple types under one abstraction: **Polymorphism** ("many forms").

Note: This is why f(istream &in) will will accept istream and ifstream, becasue istream inherits from ifstream

ifstream "is-a" istream so ifstream inherits from istream.

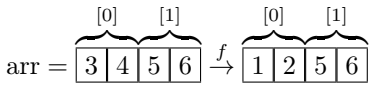
That's how we got away with our operators.

We have the ability to easily shoot ourselves in the foot.

16.2 DANGER

```
class One{
        int x;
public:
        One(int x=0):x(x){}
};
class Two: public One{
        int y;
public:
        Two(int x=0,y=0):One(x), y(y){}
};
void f(A *a){ // an array of A's
        a[0] = One(1);
        a[0] = One(2);
}
Two arr[2];
arr[0] = Two(3,4);
arr[1] = Two(5,6);
f(arr);
```

What happens?



- Data is misaligned
- ∴ Never use an array of objects polymorphically
- If you want a polymophic array: use an array of pointers.

```
B *myArray[2]; //OK
```

16.3 Destructor Revisited

Consider:

```
class X{
        int *x;
public:
        X(int n):x(new int [n]){}
        ~X() {delete [] x;}
};
class Y{
        int *y;
public:
        Y(int m, int n):X(n), y(new int [m]){}
        ~Y() {delete [] y;}
}; // runs with no leaks
X *myX = New Y(10,20);
delete myX; // has leaks! Why?
// Calls ~X() but not ~Y()
```

- So only x, but not y, is freed.

Q: How can we ensure that deletion through a pointer to a superclass will not leak memory?

- Declare the destructor virtual.
  - By making the superclass destructor virtual, we will call the approriate subclass destructor.
  - Calling a subclasses dtor will also call the superclasses dtor.
- This is a huge source of memory leaks in C++ programs.

ALWAYS Make the destructor virtual in classes that are ment to have subclasses. Even if the destructor doesn't do anything.

- We can only have the destructor non-virtual if we NEVER have a subclass.

16.4 Tools Topic: make

Separate compilation:

```
g++ -c book.cc
g++ -c ____
g++ -c ____
g++ -c main.cc
g++ -c book.o csbook.o main.o -o main
```

- Why? So we don't recompile files that haven't changed.

How do you keep track of what has changed?

make Create a Makefile that outlines which files depends on which other files.

```
main: main.o book.o csbook.o comic.o
        main depends on these
    g++ main.o book.o csbook.o comic.o -o main
[tab]       how to Build main from these files
book.o: book.h book.cc
    g++ -c book.cc
csbook.o: csbook.h csbook.cc book.h
    g++ -c csbook.cc
comic.o: comic.h comic.cc book.h
    g++ -c comic.cc
```

Algorithm 16.1 Makefile

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

Note: We need a literal tab for g++ lines, otherwise it won't compile. (We can't use editors that replaces tabs with spaces)

Command: `make`

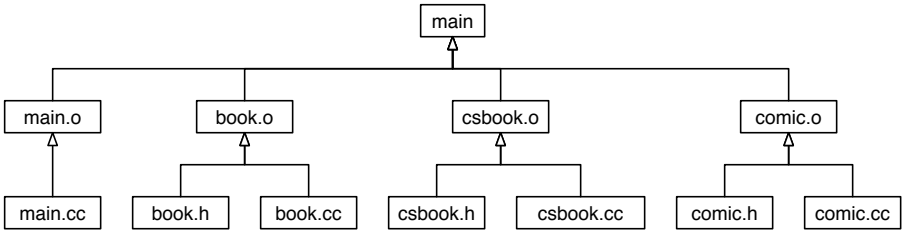
- Builds the whole project
- Only recompiles part of the program

Suppose we just change just book.cc

make

- make will recompile book.o (g++ -c book.cc)
- Relink main (g++ main.o book.o csbook.o comic.o -o main)

Make builds a dependency graph



make: builds the first target (main)

- What does main depend on?
  - book.o comic.o csbook.o main.cc
  - recursively build these
- book.cc changes
  - ∴ book.cc is now newer than book.o ∴ rebuild book.o
  - book.o now newer than main ∴ rebuild main
- Can build specific targets
  - make csbook.o
- Any change will cause rebuilding (a new timestamp)

Common: Put a target clear: at the bottom of the makefile to remove all binaries:

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

- To do a full rebuild
  - make clean
  - make

Chapter 17

Lecture 17[1]

17.1 Makefile Variables

CXX=g++ (Compiler's name)

cxxFLAGS -wall[2] (compiler options)

Example.

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

Shortcut: For any rule of the form:

```
x.o: x.cc a.h b.h ...
```

can leave out the build command

- Make will guess that it is

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

17.1.1 Biggest problem with writing Makefiles-

- Tracking dependencies and maintaining them if they change.

Can get help from g++:

`g++ -MMD -c` creates comic.o and comic.d

comic.d

```
comic.o: comic.cc comic.h book.h
```

- Now include this in the Makefile

Remmember:

...

```
OBJECTS=main.o book.o csbook.o comic.o
DEPENDS=${OBJECTS:.o=.d}
...
-include ${DEPENDS}
```

[1] Mar 11, 2014

[2] turns on all compiler warnings

17.2 Pure Virtual Methods and Abstract Classes

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

2 kinds of student: Regular and Co-op

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

class Coop: public Student{
        ...
public:
        int fees(){...}
};
```

Q: What should Student::fees do?

A: Not sure, every student should be regular or co-op.

- Can explicitly give Student::fees NO implimentation:

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

- Called a pure virtual method

A class with a pure virtual method cannot be instantiated.

```
Student s; // won't compile
```

- Called an **abstract class**
  - Purpose is to organize subclasses
- Subclasses of an abstract class are also abstract unless all pure virtual methods are implemented.

Classes that can be instantiated (no pure virtual methods) called **concrete classes**.

17.2.1 In UML:

Virtual/Pure Virtual methods: *italics*

Abstract classes: class name in *italics*

static: underline

protected: #

17.3 Inheritance and the Copy/ctor/operator=

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

class CSBook: public Book{
        ...
public:
        // No copy ctor
        ..
};

CSBook b("Algorithms", "CLRS", 500, "C");
CSBook c = b;
// Built in copy ctor for CSBook calls Book's copty ctor
// then goes field-by-field for the CSBook fields.
```

If you wrote it yourself:

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

Operator= is similar

```
CSBook c;
c=b;
// Assume operator = defined in Book, but not in CSBook.
```

Default operator= for CSBook calls Book's operator= and then goes field-by-field for the CSBook Fields.
If you wrote it yourself

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

- Since there is no dynamic memory, self-assignment check is not necessary

Now consider

```
CSBook b1(____), b2(___);
Book *pb1 = &b1;
Book *pb2 = &b2;
```

What if we do

```
*pb1 = *pb2; // like b1=b2 through pointers
```

- Partial assignment: copied only the book part
  - Book's operator= ran

Q: How can we fix this?

- Try making opertor= virtual

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

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

- Override needs the methods to be the same (same type as parameter)

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

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

Note: Different return types, but parameter types must match exactly, or the subclass version will not override the superclass version.

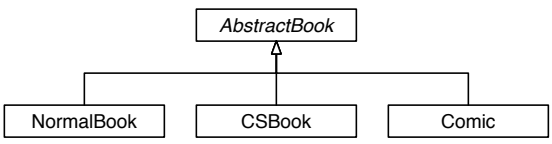
$\therefore$ Assignment of a Book obj to a CSBook var. would be allowed

```
CSBook c;
c= Book(_____); // What's the language?

c = Comic(___); // REALLY BAD and it compiles
```

- If operator= is non-virtual, partial assignment.
- If it's virtual: mixed assignments.

Recommend: All superclasses should be abstract.

Rewrite Book hierarchy



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

- Need a pure virtual method, use dtor if nothing else;

abstractbook.cc:

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

- Virtual dtor must be defined, even if it's pure virtual.

```
class NormalBook: public AbstractBook{
public:
        ...
        NormalBook &operator=(const NormalBook &other){
                AbstractBook::operator=(other);
                return *this;
        }
};
```

- No mixed assignments because opertor= is not virtual
- No partial assignments because *pb1 = *pb2 won't compile anymore (operator= on AbstractBook is protected)

Chapter 18

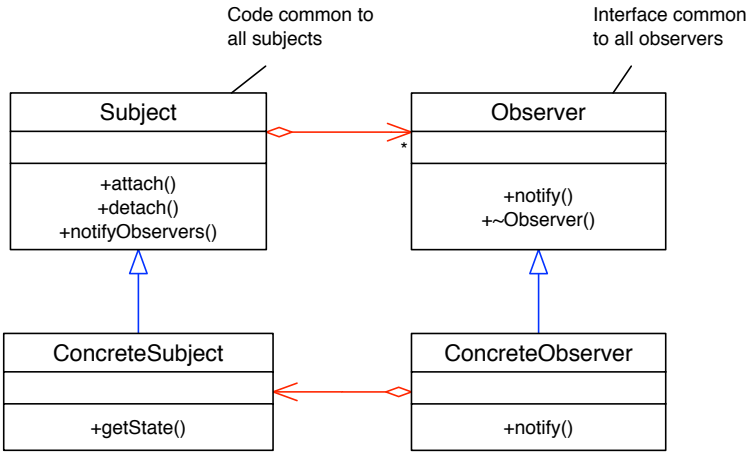
Lecture 18[1]

18.1 Observer Pattern

Publish-subscribe model

- One class is the publisher/Subject: generates data.
  - Spreadsheet cells
- One or more subscriber/observer classes
  - Recieve data and react to it
  - A graph on cells for said spreadsheet
- Can have many different kinds of observer objects:
  - Subject should not need to know all of the details

18.1.1 Observer Pattern: UML



[1]Mar 13, 2014

18.1.2 Sequence of calls:

1. Subject's state Changes
2. Subject::notifyObservers()
   - Calls each observer's notify
3. Each observer calls ConcreteSubject::getState() and reacts to the new data.

Example. Horse races
Subject: publishes winners
Observers: betters

- Declare victory when their horse wins.

```
class Subject{
        Ovserver *observers[maxObservers];
        int numObservers;
public:
        Subject(): numObservers(0){}
        bool attach(Observer *o); // add to observers
        // returns true if successful
        bool detach(Observer *o); //returns true if sucessful
        notifyObservers(){
                for (int i =0; i < numObservers; ++i){
                        observers[i]->notify();
                }
        }
        virtual ~Subject()=0;
};
```

in .cc file

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

The horse Race:

```
class HorseRace: public Subject{
        ifstream in; // source of data
        string lastwinner;
public:
        HourseRace(string source); //connected to ifstream
        ~horseRace();
        bool runRace(); //true if successful
        string getState() {return lastWinner;}
};
```

The better:

```
class Better: public Observer{
        HorseRace *subject;
        string name, myHorse;
public:
```

```
        Better(HorseRace *hr, string name, string horse): Subject(hr), name(
            name), myHorse(horse){
                subject->attach(this);
        }
        void notify(){
                string winner = subject->getState();
                if (winner == myHorse){
                        cout << "Yppee!" << endl;
                }
                else{
                        cout << "______" << endl;
                }
        }
};
```

main.cc

```
int main(){
        HorseRace hr;
        Better Larry(&hr, "Larry", "RunsLikeACow");
        ...(other betters)
        while(hr.runRace()){
                hr.notifyObservers();
        }
}
```

18.1.3 Simplificantions

1. If only one subject class, could merge subject and concrete subject
2. If the state is trivial(e.g. just a notification is all you need), don't need getState()
3. If subject==observer(e.g. cells in in a grid), you can merge these classes.

18.2 Decorator Pattern

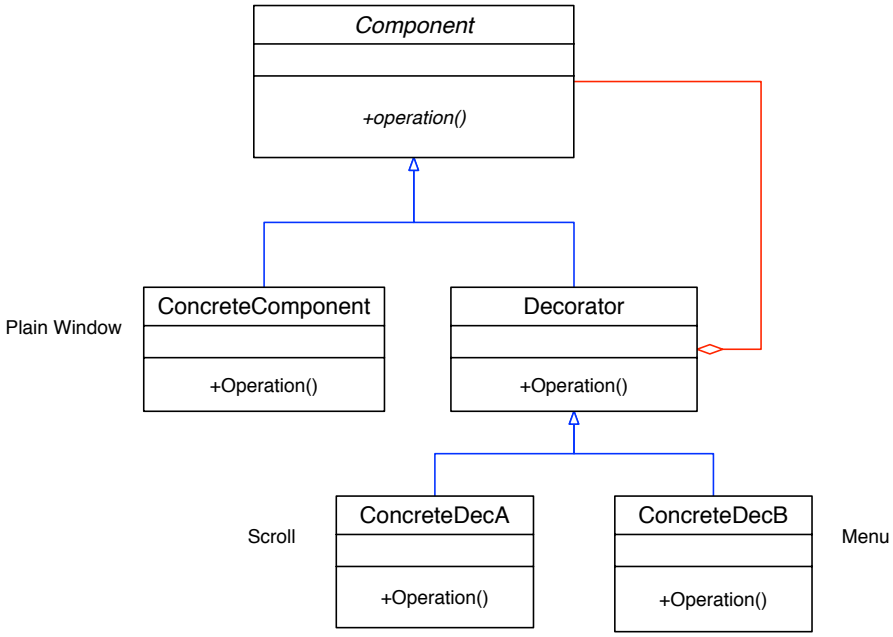
Want to add features to an object at run-time.

Example. Windowing system:

- Basic window
- Add menu
- Add scroll bar

Happens at run-time

18.2.1 Decorator Pattern: UML



Class Component: Interface: operations your objects will support.

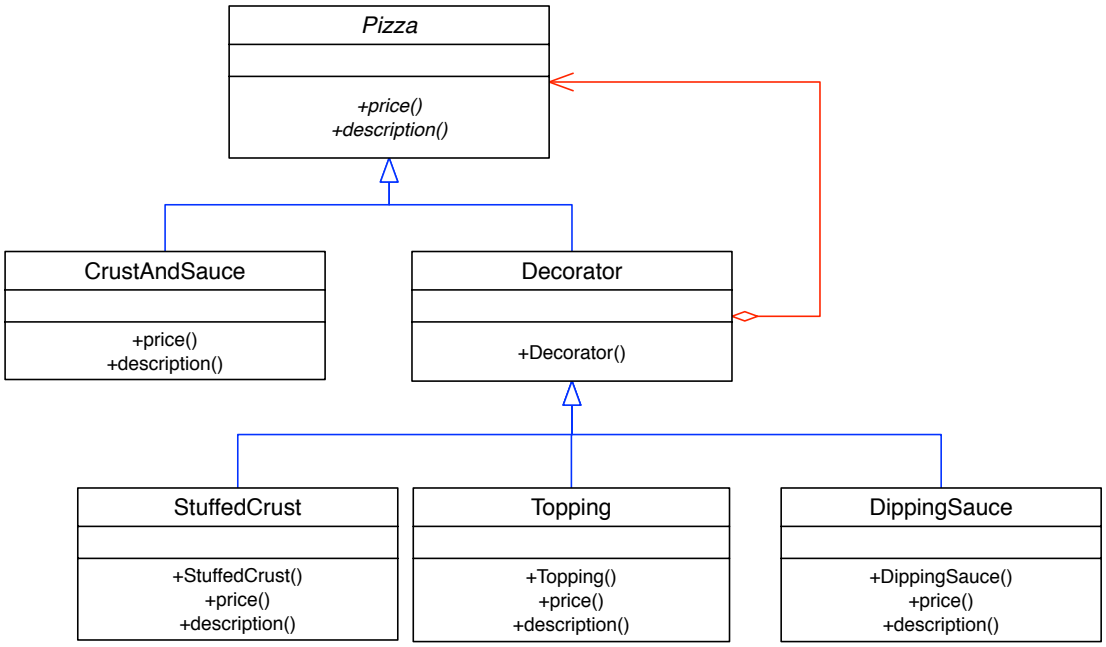
Concrete Component: Implements the interface.

Decorators: All inherit from decorator, which inherits from component.

- Every decorator is a component and HAS A component.

- A window with scrollbar is a window and has a pointer to the underlying plain window.
- A window with scroll and menu is a window and has a pointer to windows with/scroll, which has a ptr to plain window.

18.2.2 Pizza Example




```
class Pizza{
public:
        virtual float price()=0;
        virtual string desc()=0;
        virtual ~Pizza(){}
};
class CrustAndSauce: public Pizza{
public:
        float price() {return 5.99}
        string desc() {return "Pizza";}
};

class Decorator: public Pizza{
protected:
        Pizza *component;
public:
        Decorator(Pizza *p): component(p){}
        virtual ~Decorator() {delete component;}
        // may not want to delete component, situation dependent
};

class Topping: public Decorator{
        stirng theTopping;
public:
        Topping(string topping, Pizza *p): Decorator(p), theTopping(topping){}
        float price() {return component->price() + 0.75}
        string desc(){return component->desc() + " with " + theTopping;}
};
```

Use

```
Pizza *p = new CrustAndSauce;
```

```
p=new topping("Cheese", p)
p= new StuffedCrust(p);
cout << p->price() << endl;
```

18.3 Tools: Debugger

GNU debugger gdb

To Use: compile with -g

(embeds debugging info into object file)

```
g++ -g source.cc
gdb ./a.out
```

r (run) runs the program

- If the pgroam crasses, tells you which line

l (list) print source surrounding pto of execution

p (print) prints the value of a variable.

Chapter 19

Lecture 19[1]

Recall: GDB

g++ -g cfile.cc

`r` = run

`l` = list

`p` = print

`bt` = backtrace

- Print the chain of function calls that lead to current pointer

Breakpoints: Tell GDB to stop the program at some point so you can see what's going on.

break f break when entering function f

break myfile.cc:15 break at myfile.cc, line 15

step: run one line of the program

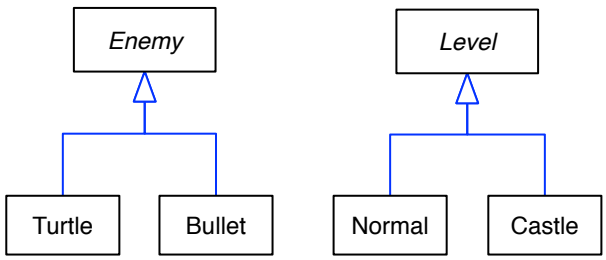
continue: go to the next breakpoint

19.1 Design Pattern: Factory Method Pattern

Writing a video game with 2 kinds of enemies:

1. Turtles
2. Bullets

System randomly sends enemies, with bullets more frequent at the end.



[1] Mar 18, 2014

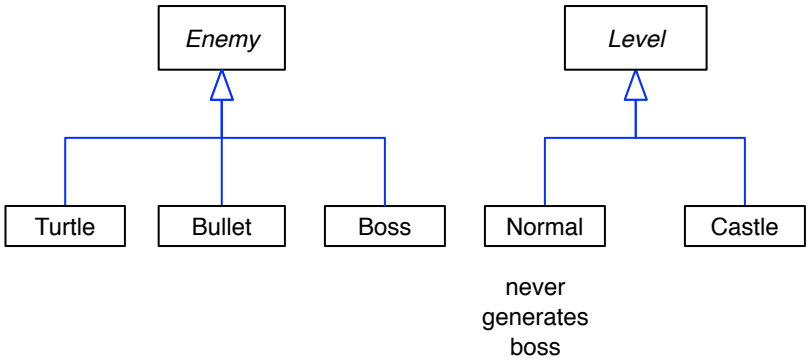
- Never know which enemy is coming next, so can't call ctor directly.
- Instead, put a factory method inside level that creates enemies.

```
class Level{
public:
        virtual Enemy*createEnemy()=0; //factory method
        // creates enemies
        ...
};
class Normal: public Level{
public:
        enemy *createEnemy(){
                // mostly turtles
        }
};

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

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

- Factories integrate well with Singletons.

Suppose we add another enemy type: Boss

- Only one
  - Everytime he appears, it is the same one
  - Singleton



- createEnemy can produce a new enemy or the singleton boss
  - Makes no difference to the client

19.2 Design Patterns: Template Method Pattern

Want subclasses to override superclass behaviour, but some aspects stay the same.

Example. red turtles, and green turtles

```
class Turtle: public Enemy{
public:
        void Draw(){
                drawHead();
                drawShell();
                drawTail();
        }
private:
        void drawHead(){...}
        virtual void drawShell()=0;
        void drawTail(){...}
};
class RedTurtle: public Turtle{
        void drawShell(){\\ draws red shell}
};
```

- Subclasses can't change the way a turtle is drawn (head, shell, tail), but they can change the one way a shell is drawn.

19.3 Templates

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

Q: What if you want to store something else? Whole new class?

A: Use a template: a class parameterized by a type.

```
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', new Node<char>('b',0)
```

Compiler specializes templates at the source code level before compilation begins.

Q: What about a linked list of linked lists?

```
Node<Node<int>> *listoflist; \\won't compile because of >> (input operator)

Node<Node<int> > *listoflist; \\ will now compile
```

19.4 The Standard Template Library (STL)

- Large number of useful templates

Example. Dynamic-length arrays: vectors

```
#include <vector>
using namespace std;
vector<int> v;
v.push_back(1);
v.push_back(2);
for (int i=0; i<v.size(); ++i){
        cout << v[i] << endl; // [i] accesses the ith element with no range checking
        // v.at(i)  same, but does range checking
}
```

`v.pop_back()` Removes the last element.

19.4.1 Iterators

In CS 136:

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

in C++:

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

Note: v.end() points one back the end.

- Iterator is an abstraction of a ptr.

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

Q: Why use this instead of the usual?

A: It is useful as more types can use the iterator: linked list etc.

Can use iterators to remove items from inside a vector.

```
v.erase(v.begin()); \\erases element 0
v.erase(v.begin()+3); \\ erases element 3
v.erase(v.end() -1); \\ erases last element
```

Chapter 20

Lecture 20[1]

20.1 Maps: for creating Association Lists (Dictionaries)

Example. "arrays" that map strings to int

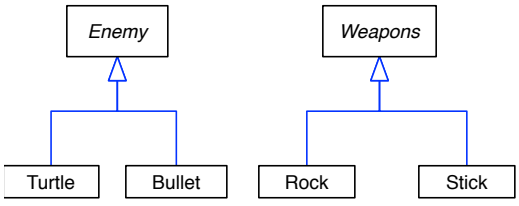
```
#include <map>
using namespace std;
map<string,int> m;
m["abc"] = 1;
m["def"] = 4;
cout << m["abc"] << endl; // prints 1
m.erase("abc");
if (m.count("def")){...} // 1 = found, 0 = not found
```

Iterating over a map $\Longrightarrow$ sorted key order

20.2 Design Pattern: Visitor Pattern

For implementing double dispatch. Recall virtual methods: pick the version based on the run-time type fo the object it goes with.

Q: What if we want to choose based on two objects?



Strike an enemy: effect depends on both Enemy type and weapon type.

Q: Which class should have the strike method?

If

```
void Enemy::strike(Weapon &w);
```

- Pick based on Enemy, but not on weapon

[1] Mar 20, 2014

```
void Weapon::strike(Enemy &e);
```

- Choose based on weapon, but not on enemy.

Trick to get dispatch based on both

- Combine overriding with overloading

```
class Enemy{
public:
        virtual void strike(Weapon &w)=0;
};
class Turtle: public Enemy{
public:
        void strike(Weapon &w){
                w.useOn(*this);
        }
};
class Bullet: public Enemy{
public:
        void strike(Weapon &w){
                w.useOn(*this);
        }
};
```

Weapon

```
class Weapon{
public:
        virtual void useOn(Turtle &t)=0;// Overloading
        virtual void useOn(Bullet &b)=0;
};
class Stick: public Weapon{
public:
        void useOn(Turtle &t){//strike Turtle with stick}
        void useOn(Bullet &b){//strike Bullet with stick}
};
class Rock: public Weapon{
public:
        void useOn(Turtle &t){//strike Turtle with Rock}
        void useOn(Bullet &b){//strike Bullet with Rock}
};
```

```
Enemy *e = new Bullet;
Weapon *w = new Stick;
e->strike(*w);
```

- strike is virtual ∴ Bullet::strike(*w) runs
- w->useon(Bullet) is called.
- useOn is virtual ∴ Stick::useOn(Bullet) runs

Visitor pattern can be used to add functionality to existing classes without changing or recompiling them.

Example. Configure Book hierachy to accept visitors

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

class CSBook: public Book{
public:
        virtual void accept(BookVisitor &v){
                v.vist(*this); //*this is a CSBook here
        }
};
class Comic: public Book{
public:
        virtual void accept(BookVisitor &v){
                v.vist(*this); //*this is a comic here
        }
};

class BookVistor{
public:
        virtual void visit(Book &b)=0;
        virtual void visit(CSBook &csb)=0;
        virtual void visit(Comic &c)=0;
};
```

Application: Track how many of each type of book we have:

Books: by author

CSBook: by language

Comic: by hero

Use a map<string, int>
Could add

```
virtual void updateMap(___);
```

to each class;
Or write a visitor:

```
class catalogue: public Book Visitor{
        map<string,int> theCat;
public:
        map<string,int> getCatalogue(){return theCat;}
        void visit(Book &b){ theCat[b.getAuthor()]++; }
        void visit(CSBook &csb){ theCat[b.getLanguage()]++; }
        void visit(Comic &csb){ theCat[b.getHero()]++; }
};
```

- The program in the repository won't compile

Q: Why didn't it compile?

Consider a linked list of alternating int/char

```
#ifndef _____
#define _____
#include "blist.h"
class Alist{
        int data;
        Blist *next;
};
#endif

#ifndef _____
#define _____
#include "alist.h"
class Blist{
        char data;
        Alist *next;
};
#endif
```

Included once: but one will be first: won't know about the other one.

Q: How mcuh about Blist does Alist need to know?

A: Just that it exists.

Q: Can we declare a class without defining it?

A: Yes

```
#ifndef _____
#define _____
class Blist;
class Alist{
        int data;
        Blist *next;
};
#endif

#ifndef _____
#define _____
class Alist;
class Blist{
        char data;
        Alist *next;
};
#endif
```

This will compile.

20.3 Compilation Dependencies

- When do you need to include.

```
class A {...};

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

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

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

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

- Need to know how big A is to konw how big B and C are.

Chapter 21

Lecture 21[1]

In the implementation of D,E

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

Now consider the XWindow class:

```
class XWindow{
        Display *d;
        Window w;
        int s;
        GC gc;
        unsigned long colours[10];
        // This is private data.
        // Yet we can look at it. We shouldn't care about it.
public:
        ...
};
```

Q: What if I add or change a private member?

A: All clients need to recompile.

Would be better to hide these details away

Solution: Use the pimpl idiom.

("pointer to implementation")
Create a second class XWindowImpl:

- No need to include Xlib.h
- We forward declare Impl class
- Now there is no compilation dependency on XWindowImpl

Algorithm 21.1 d.cc

```
#include "a.h"
void D::f(){
        myAp ->someMethod(); // need to know that class A has this method
}
```

Algorithm 21.2 XWindowImpl.h

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

Algorithm 21.3 Window.h

```
class XWindowImpl;
class XWindow{
        XWindowImpl *pImpl;
public:
        ...
};
```

Algorithm 21.4 XWindow.cc

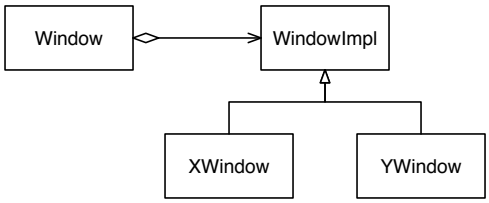
```
#include "xwindow.h"
#include "XWindowImpl.h"

XWindow::XWindow(): pImpl(new XWindowImpl){...}

//Other Methods: replace fields d, w, s, gc, colours
// with pImpl->d, pImpl->w, etc.
```

If you keep all private fields in XWindow Impl, then only windows .cc needs to recompile if you change XWindow's Implementation.

21.1 Generalization

Q: What if there are several possible window implementations, say XWindows and YWindows?

- Make the Impl struct a superclass



- pimpl idiom with subclassing to accommodate alternative implementations
  - Called the **Bridge Pattern**

21.2 Measures of Design Quality

Coupling and Cohesion:

Coupling: The degreee to which program modules depend on each other

Low: Modules communicate via function calls with basic parameters/results
- Modules pass arrays/structs back and forth
- Modules affect each other's control flow (e.g. flag parameters)
- Modules shared global data.

High: Modules have access to each other implementation.

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

Low: Arbitrary grouping of unrelated elements
- Elements share a common theme, but are otherwise unrelated, perhaps share the same base code (e.g. <algorithim>)
- Elements manipulate state over the lifetime of an object (e.g. open/read/close files)
- Elements pass data to each other

High: Elements cooperate on exactly one task.

Goal: Low coupling, High cohesion.

21.3 Casting

In C:

```
Node n;
int *ip = (int *)&n;
//forces c/C++ to treat the Node* as an int*
```

C-style casts should be avoided in C++.
If you must do it, use a C++ style cast.

[1]Mar 25, 2014

21.3.1 4 Kinds of Casts

21.3.1.1 static_cast

static_cast "sensible casts"

Example.

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

superclass ptr->subclass ptr

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

- You are taking responsibility that b actually points to a CSBook. "Trust me!"

21.3.1.2 Reinterpret Cast

reinterpret_cast Unsafe, implementation-specific, "weird" casts

```
Vec v;
Student *s = reinterpret_cast<Student *>(&v);
```

21.3.1.3 const_cast

const_cast the only C++ cast that can "cast away const"

```
void g(int *p);
void f(const int *p){
        g(const_cast<int *>(p));
}
```

Chapter 22

Lecture 22[1]

Recall: reinterpret_cast
const_cast

```
Book *b = new comic(____);
Comic *c = static_cast<Comic *>(b);
```

Q: What about this?

```
Book *b2 = myBooks[i];// Not sure if it's a comic
```

- Don't know if it's safe to treat the Book ptr as a Comic ptr.

22.1 dynamic_cast

- tentative cast: try it and see if it suceeds

```
Comic *c2 = dynamic_cast<Comic *>(b2);
```

- If the cast works (b2 actually points at a comic), c2 points to the object. If the cast fails, c2 will be NULL.

```
if (c2) cout << c2->getHero();
else cout << "Not a comic." << endl;
```

- For dynamic cast to work, we need the base class to have virtual functions.

Can use dynamic casting to make decision based on an object's run-time type (RTTI = run-time type information)

```
string whatisit(Book *b){
        if (dynamic_cast<Comic *>(b)) return "comic";
        if (dynamic_cast<CSBook *>(b)) return "CSBook";
        return "Book;
}
```

Code like this is highly coupled to the book heirarchy and may indicate bad design.

Better: Use virtual methods or write a visitor

[1] Mar 27, 2014

Dynamic Casting also works with references.

```
Comic c(____);
Book &b = c;
Comic &c2 = dynamic_cast<comic &>(b);
```

If b "points to" a comic, then c2 is a reference ot the same comic.
If not...????

22.2 Error Handling

Q: What happens in C++, if things go wrong?

- Dynamic cast on refs fails
- new fails
- vector::at(range checked) fails

Problem: 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.

C Solution: function returna status code or set the global variable errrno.

- Awkward
- Encourages programmers to ignore error checks.

C++ Solution: When an error condition arises, the function raises an exception.

Q: What happens-by default, execution stops.

We can write handlers to catch exceptions

```
vector::at raise out_of_range
```

Handle as follows:

```
#include <stdexcept>
...
try{
        cout << v.at(3) << endl;
}
catch (out_of_range){
        cerr << "Range error" << endl;
}
```

Consider

```
void f(){throw out_of_range("f");}
void g(){f();}
void h(){g();}
int main(){
        try {h();}
        catch (out_of_range) {...}
}
```

What Happens:

- main calls h

- h calls g
- g call f
- f raises

g has no handler for out_of_range.

- Control back though the call chain (unwinds the stack) until a handler is found.
  - Control goes all the way to main and main handle the exception.
- If no handler in the entire call chain

out_of_range is a class

```
throw out_of_range("f");
```

invokes a ctor.

A handler can do part of the recovery job, execute some correct code and then throw another exception

```
try{...}
catch(SomeErrorType s){
        ...
        throw SomeOtherExn("...");
}
```

Or throw the same exception:

```
try{...}
catch(SomeErrorType s){
        ...
        throw;
}
```

missing the last image

throw; actual type of s si retained

throw s; rethrows a new exception of type SomeErrorType

s could be a subtype of SomeErrorType

A handler cna act as a catch_all

```
try{...}
catch(...){ // the ... catches all classes
        \\__ catches all exceptions
}
```

You can throw anything you want dont have to throw objects.

Good Practice: define exception classes (or use appropriate existing ones) for your error cases

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

Advice: Catch exceptions by reference

- Reduces copying
- If a subclass exception is caught by value, it is sliced
- catch byref $\Longrightarrow$ no slicing

22.3 Standard Exceptions

bad_cast dynamic ref cast fails

bad_alloc new fails

Chapter 23

Lecture 23[1]

Recall:

```
Book *p = new Comic(_____);
Book *q = new Comic(___);
*p = *q;
```

- if Book::operator= non-virtual: partial assignment (slicing)
- If Book::operator= is virtual, then Comic::operator= must accept any Book object.

```
Comic &Comic::operator=(const Book &other){
        Comic &cother = dynamic_cast<comic &>(other);
        // throws if other is not a comic
        title=cother.title;
        ...
        hero=cother.hero;
        return *this;
}

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

- No leaks: what if g throws

23.1 Exception Safety

Q: What's guaranteed?

- During stack-unwinding all stack allocated data is cleaned up: dtors run, memory reclaimed.
- Heap-allocated memory is not destroyed.

$\therefore$ If g throws, p is leaked, but q is not.

[1] April 1, 2014

```
void f(){
        Myclass *p = new Myclass;
        Myclass q;
        try { g(); }
        catch(...){
                delete p;
                throw;
        }
        delete p;
}
```

- Tedious and error-prone

Q: How can we guarantee that something (delete p) happens no matter how we exit f (normally or exceptionally)?

A: In some languages: "finally" clauses guarantee certain actions at the end of a function.

- Not in C++

Only guarantee in C++: dtors for stack allocated data will run. ∴ Use stack allocated data as much as possible.

BUT NEVER let a dtor throw an exception.

- If the dtor was executed during stack-unwinding while looking for a handler for another exception, you now have two active unhandled exception, and your program will abort.

Key is to use the guarantee that stack-allocated objects are destroyed to our advantage.

C++ idiom: RAII: Resources Acquistion is Initialization.

R	Resouce
A	Acquisition
I	Is
I	Initialization

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

Example. files

```
{
        ifstream f("file"); //Acquireing the resource ("file")
        // = initialization an object
        ...
}
```

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

This can be done with dynamic memory.

class auto_ptr<T>: Takes a T* in the ctor.

- dtor deletes the ptr
- In between-derefence just like a ptr

```
void f(){
        auto_ptr<Myclass> p(new Myclass);
        Myclass q;
        g();
}
```

Diffculty:

```
class C{};
auto_ptr<C> p(new C);
auto_ptr<C> q=p;
```

Q: What happens when an auto_ptr is copied?

- Don't want to delete the same ptr twice!

Q: What happens when p is copied to q?

- p loses possession of the ptr (p is set to NULL).

23.2 3 Levels of Exception safety

1. **Basic guarantee**: If an exception occurs, the program will be in a valid state, including no leaked resources.
2. **Strong guarantee**: If an exception occurs, while running a function f, then the state of the program is as if f had not been run (f is atomic).
3. **No throw guarantee**: f will never throw: always accomplishes it's purpose.

Example.

```
class A{...}
class B{...}
class C{
        A a;
        B b;
public:
        void f(){
                a.g();//may throw (strong guarantee)
                b.h(); //May throw (strong guarantee)
        }
};
```

Q: Is f exception safe?

- If g throws: nothing has happened yet
  - Ok
- If h throws: effects of g must be undone to offer the strong guarantee
  - Very hard if g has non-local side effects

∴ No, probably not strongly exception safe.
Assume g, h do not have non-local side-effects.

- Use Copy-and-Swap

Example.

```
class A{...}
class B{...}
class C{
        A a;
        B b;
public:
        void f(){
                A atemp = a;
                B btemp - b;
                atemp.g();
                atemp.h();
                a = atemp;
                b = btemp; // what if these throw?
                // not haveing the strong guarantee if b throws
        }
};
```

Better if the swap was nothrow.

Copy ptrs will never throw. Use pimpl.

```
struct CImpl{A a; B b;}
class C{
        auto_ptr<cImpl> pImpl;
        ...
public:
        void f(){
                auto_ptr<CImpl> temp(new CImpl(* pImpl));
                temp->a.g();
                temp->b.h();
                std::swap(pImpl, temp); // no throw
        }
};
```

If A::g and B::h offer no exception safety guarantees, then neither can f.

Chapter 24

Lecture 24[1]

```
class Vec{
        int x,y;
        int f();
};
class Vec2{
        int x,y;
        virtual int f();
};
Vec v;
Vec2 w;
cout << sizeof(v) << sizeof(w) << endl;
```

Q: Do v and w look the same in memory?

- v is size 8 and w is size 16

Consider class Vec
int has a size of 4
With our current knowledge we would expect a size of 8 (the size of 2 ints).
These two objects do not look the same in memory.

Q: Where does the other 8 for w come from?

Fact. *No space is allocated in an objects for methods, methods are not sitting inside an object. The compiler turns methods into ordinary functions.*

- *This is a good thing, if there was a method inside objects, every object would hae a copy of the same method and that would be a waste in memory as they are all the same.*

To explain this go back to book hierarchy:
Recall:

```
Book *b = new Comic;
b->isitHeavy();
```

- Remember that `isitHeavy()` method is a virtual method.
- The choice of which `isItHeavy()` to run depends on the actual type of the object
  - Key observation is that the compiler doesn't know what that is

[1] Apr 3, 2014

What if the initialization of this pointer apphended in if[2] and depending on the if[3] it may be comic or some other kind. Possible that we may not know what kind of book we have.

Q: So if compiler doesn't know, then how is it going to run the right one?

A: That decision will have to be made at runtime.

Q: How?

- For each class with virtual methods, the compiler creates a table of function pointers (the vtable)
  - v as in virtual

```
class C{
        int x,y;
        virtual int f();
        virutal int g();
        int h();
        virtual ~C();
};
```

- Seeing that C has virtual methods compiler creates a vtable

vtable

"C"

f→ f's code
g→ g's code
~C→ ~C's code

C objects have an extra pointer (the vptr) that pts to C's vtable
the C object in memory.

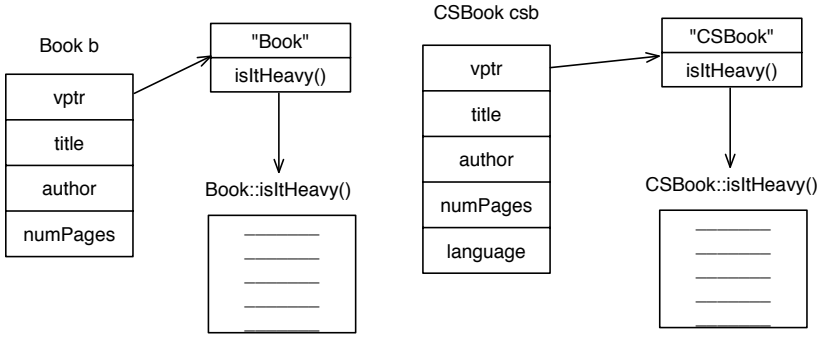
c

x
y
vptr

C objects have an extra pointer (the vptr) that pts to C's vtable.
Recall:

```
Book *pb = _____
pb-> isItHeavy();
```

Example. Book, CSBook



[2]What is this sentance saying???
[3]What is this sentance saying???

If `isItHeavy()` is virtual_choice of which version of `isItHeavy()` to run is based on the actual type of the object which the compiler can't know in advance.

Critically imporant to remember that we are not creating one table for every object. No, we create only one for the class.

Calling a virtual function: (at run-time)

- Follow vptr to vtable
- fetch ptr to actual method form vtable
- follow the ptr and call the function

$\therefore$ Virtual function calls incur a small overhead cost. Calls are slower than actual method calls because you have a couple of ptr chases.

Also: delaring at least one virtual function adds a vptr to object. $\therefore$ classes with no virtual functions produce smaller objects than if at least one function are virtual.

Ptrs are 8 bytes, so the extra 8 bytes come from the vptr.

virtual methods cost you in space and time.

Also, declaring at least one virtual function adds a vptr to the object, making the object larger.[4]

Vec2 had extra ptr, so each Vec2 object had 16 bytes instead of 8, mading it twice as big. So in any given memory you can store half as much Vec2's as Vec. So what? Remember things about memory from assignment? So you'll have cache issue, so it can have difference. But what if the object was even smaller? 1 field char = 1 byte. So that 1 byte became 9. Sometimes compiler leave a little bit of space cause pointers are better, so the compiler might add itself another 7 bytes, so that it becomes large and nice and 16, so ur 1 byte became 16.

What's next? A little time about what these things actually look like in memory. This was a conceptual model...

Q: How should an object be laid out?

A: Compiler-dependent: answer to how exactly it is load out depends on the compiler, it can do whatever it wants but certain patterns are common and let me show you g++

24.1 How g++ Arranges Objects in Memory

Q: Should I put the vptr first last or in the middle?

```
class A{
        int a,c;
        virtual int f();
};
```

vptr
a
c

```
class B: public A{
        int b,d;
};
```

vptr
a
c
b
d

[4] huge deal if programs huge maybe?

An object B "is-an" A, supposed to be a kind of A. Sure would be nice if a B ojbect looked like an A object. So by covering the b and d variabes, it looks like an A object. If ptr at bottom, I can append it and it will not look like it. That's a very good reason to the ptr first.

So a ptr to B looks like a ptr to A, if we ignore the B part. Very easy to treat B as A cause you just ignore the two fields.

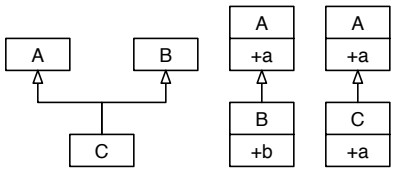
24.2 Multiple Inheritance

A class can inherit from more than one class.

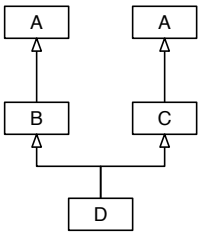
```
class A{
public:
        int a;
};
class B{
public:
        int b;
};
class C: public A, public B{
public:
        void f(){cout << a <<    << b << endl;}
};
```

Challenges: Suppose B inherits from A

Example.



Also:




```
class D: public B, public C{
public:
        int d;
};
D dObj:
dOjbj.a; // which a is this?
```

We now have 2 a's.

Q: Which a is this?

A: We have to tell it, we ahve to be more specific and tell it which a we mean. The syntax is a little weird but quite logical.

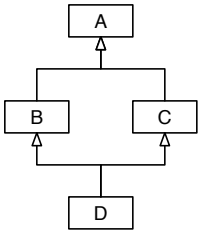
Need to specify:

```
dObj.B::a;
dObj.C::a;
```

But if B and C inherit from A, should D have two A parts or one? (two is the default)
In a lot of cases the fact that you get two copies of a is not the most sensible thing, only sometimes.

Q: If B and C inherit from A, should there be one A part of D or two?

- Should B::a and C::a be the same or different?
- What if we want is just one A ("deadly diamond")
  - deadly if you need to write a compiler, one more reason not to do it



- Make a virtual base class

24.2.1 Virtual Inheritance

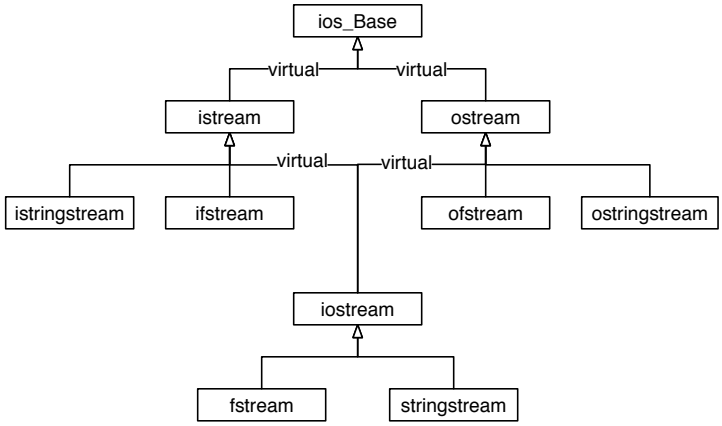
```
class B: public virtual A{};
class C: public virtual A{};
```

- virtual here means I only ever want to be one copy of A when I inherit from this

If we do this, then, I will take the ambiguity away. Saying that when they inherit from A there should be only one A part.

We've been using it all term. We've een interacting with virtual base classes all term.
Reintroduce you to the very first thing you learned in C

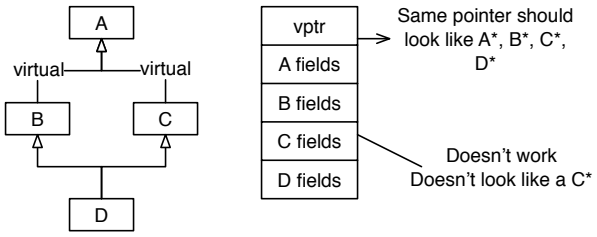
Example. Io: steam hierarchy:



istream inherits fail bit from iostream and ostream does similarly, so if I am an iostream I need only one fail bit, don't want it twice. So it doesn't make sense to inherit one copy though the isteam and one from the osteam. By ensuring that ios is virtual there is only one fail bit.

Q: How will this be laid out?

A: vptr should be valid as all four things A*, B*, C*, or D*



- It looks like A and B, and like D but not like C* becasue C doesn't know that it has a B somewhere else.

Q: It doesn't look like a C*, so what does g++ actually do?

I'm taking the D object and treating it like an array of ints with reinterpret cast.
ptrs have to ints because that are 8 bytes.
This is contortions that the compiler has to go through to make multiple inheritance work.
If it looked like B since B is inheriting from A and A is down

- Be needs to be laid out so we can find it's a part, but the distance is unknown

Solution: Location of the base class object is stored in the vtbl.

- Doesn't look like A, B, C, D simultaneously, but slices of it do.

When you assign to pointers, it doesn't mean that they get the same address.
What does g++ do?

vptr	← ptr to B	- BUT NOT REALLY: B needs to be laid out so we can find it's A part, but the distance to A is unknown
B fields		
vptr	← ptr to C, D?	
C fields		
D fields		
vptr	← ptr to A	
A fields		

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

ptr assignment among A,B,C,D changes the address stored in ptr.
An change of the class of the pointer shift the address of the pointer so that it points to the right thing in the object.

- static_cast, dynamic_cast, c-style case under multiple in heritiance will also change the pointer. reinterpret_cast will not.

24.3 Return-Value Optimization

```
class C{
public:
        C(){cout << Default ctor << endl;}
        C(const C &c){cout << Copy ctor << endl;}
        c & operator=(C &c){cout << Assignment << endl;}
```

```
};
C f(){
        cout f << endl;
        return C();
}
        int main(){
        C c= f();
}
```

Output?

- Conceptually: f calls default constructor
- Copies to temporary object in main stack frame when f returns
- then copy constructor runs to initialize C

$\therefore$ 2 copy constructors + 1 default constructor.

- When we run it, just the default constructor runs

  Q: Why? these temporaries are only for shuffling data around and are otherwise unobservable

So the compiler is allowed to skip the copy constructors and allow f to write it's value directly into c.

- Even if the copy constructor has observable side-effects.
  - Called the Return Value Optimization (RVD)
  - If you don't want it: compile with

```
-f no-elide-constructors
```

  - For assignment operators, this can't be done because we can't write new data into an existing object because the old data needs to be destroyed!

```
C c = f(); //RVO ok
c = f(); // default ctr + assignment
```

If C objects are big

```
class C{
        int *a;  // 10,000 entries
        ...
};
```

Returning a C object from f by value will be expensive.
End of Final Material

Can we get around this expense?
In C++11- Yes.
With move constructors and rvalue refs

$$\underbrace{x}_{\text{lvalue}} = \underbrace{f()}_{\text{rvalue}};$$

C++ refs must be lvalues.
A rvalue ref pts at a temporary value created by the compiler.

X && rvalue ref to x

So you could write

```
C& operator=(C && other){
        // instead of copy, swap the ptrs
        std::swap(a, other.a);
        return *this;
}
```

- copy and swap without the copy!