

CS 246 - Object-Oriented Software Development

Cameron Roopnarine

Last updated: February 13, 2020

Contents

1	2020-01-07	3
1.1	Linux Shell	3
1.2	Linux File System	3
1.3	Commands	3
2	2020-01-09	4
2.1	Linux Streams	4
2.2	Wildcard Matching	5
2.3	Linux Pipe	5
3	2020-01-14	5
3.1	Searching Text	5
3.2	File Permissions	7
3.3	Shell Variables	8
3.4	Shell Scripts	8
3.5	Summary of Files	8
4	2020-01-16	8
4.1	Shell Scripts	8
4.2	Summary of Files	10
5	2020-01-21	10
5.1	Testing	10
5.2	C++ Introduction	11
5.3	iostream header	12
5.4	Compile C++	12
5.5	Run C++	12
5.6	C++ I/O	12
5.7	Summary of Files	13
6	2020-01-23	13
6.1	C++ I/O	13
6.2	C++ Strings	14
6.3	Summary of Files	15
7	2020-01-28	15
7.1	Short C++ Topics	15
7.2	Default Arguments	16
7.3	Structs	17
7.4	Constants	17
7.5	Parameter Passing	18

7.6	Lvalue References	18
8	2020-01-30	19
8.1	More Short C++ Topics	19
8.2	Dynamic Memory Allocation	20
8.3	Operator Overloading	20
9	2020-02-04	21
9.1	Preprocessor, Separate Compilation	21
9.2	Conditional Compilation	22
9.3	Separate Compilation	22
9.4	Build Tools	23
10	2020-02-06	23
10.1	Preprocessor, Include Guards, C++ Classes	23
10.2	C++ Classes	24
10.3	Initializing Objects	24
11	2020-02-11	26
11.1	C++ Classes (Cont.)	26
11.2	Destructor	29
11.3	Copy Assignment Operator	29
12	2020-02-13	29
12.1	Big 5 (Cont.)	29
12.2	Copy and Swap Idiom	31

1 2020-01-07

1.1 Linux Shell

Shell: interface to an OS

Graphical shell: click/touch, intuitive

Command line shell: type commands, not intuitive, more powerful

Stephen Bourne (70s): original UNIX shell

History: C shell (`cs`h) → Turbo C shell (`tc`sh) → KornShell (`k`sh) → Bourne Again Shell (`ba`sh)

Check what command line shell: `echo $0`

Go into bash: `ba`sh

1.2 Linux File System

Directories: files that contain files (called folders in Windows), e.g. `usr`, `share`, `dict` are all directories

Root (literally a backslash) `/`: top directory

Path: location of a file in the file system, e.g. `/usr/share/dict/words`

Absolute path: path that starts at the root directory

Relative path: path relative to a directory

The path `dict/words` relative to `/usr/share` is `/usr/share/dict/words`

1.3 Commands

- NAME: `ls` → list directory contents.

SYNOPSIS: `ls [OPTION]... [FILE]`

DESCRIPTION: List information about the non-hidden `FILE`s (current directory by default). Hidden files start with a `.`

`ls -a` or `ls -all` do not ignore entries starting with a `.`; the `-a` is an argument

- NAME: `pwd` → print name of current/working directory.

SYNOPSIS: `pwd [OPTION]...`

DESCRIPTION: Print the full filename of the current working directory.

- NAME: `cd` → change the shell working directory.

`..` → parent directory

`.` → current directory

`-` → previous directory

`~` → home directory

`~userid` → `userid`'s directory

2 2020-01-09

It's strongly recommend that you **do not** memorize these commands presented, you should try them out on your own to see what the output is.

- CTRL + C → send kill signal
- CTRL + D → send EOF (end-of-file)
- NAME: `cat` → concatenate files and print on the standard output

SYNOPSIS: `cat [OPTION]... [FILE]`

DESCRIPTION: Concatenate FILE(s) to standard output. With no FILE, or when FILE is -, read standard input.

- > → output redirection, overwrites files

`cat > out.txt` → redirects output produced by `cat` to the file `out.txt`

`cat t1.txt > t2.txt` → redirects all text from `t1.txt` to `t2.txt`

- >> → appends at the end of the file instead of overwriting like >
- < → input redirection

`cat < sample.txt` → input redirection, the shell handles this

`cat sample.txt` → `cat` handles this

`cat -n < in > out` → -n numbers all output lines. Input redirect from file `in` to `cat`, then output redirect with numbered lines to file `out`.

2.1 Linux Streams

- 1. Standard input (stdin)

keyboard

use < to change to file

- 2. Standard output (stdout)

terminal

use 1> to change to file; the 1 before the > is optional

buffered

- 3. Standard error (stderr)

terminal

use 2> to change to file

non-buffered

We use the non-buffered stream when we immediately want to output an error so that it does not take extra CPU cycles (extra material).

Within the stream,

`stdin` → program → 1. `stdout` and 2. `stderr`

`prog arg1 < in > out 2>&1` → &1 is the location of `stdout`, so any errors will be redirected to `stdout`.

2.2 Wildcard Matching

`ls *.txt` → match anything that ends with `.txt`. The shell performs this operation.
globbing pattern

Using single/double quotes will suppress globbing patterns.

`\` is the escape character

Example

Count the number of words in the first 15 lines of `sample.txt`.

Solution.

- `wc -w` → print number of words in entire text
- `head -15 sample.txt` → get only the first 15 lines of `sample.txt`
- `head -15 sample.txt > temp.txt wc -w temp.txt` → doing both, with output in a `temp.txt` file.

What if we didn't want `temp.txt` to be produced? We use Linux pipes.

2.3 Linux Pipe

Connect `stdout` of `prog1` to `stdin` of `prog2`.

`head -15 sample.txt | wc -w` → the first program, `head` runs with `sample.txt`, then the output is fed into the second program, `wc`.

Example

Suppose `words*.txt` contains one word per line. Produce a list of words sorted, with no duplicates from `words*.txt`.

Solution.

`cat words*.txt | sort -u` OR `cat words*.txt | sort | uniq` → `sort -u` will sort and remove any duplicate words. `uniq` removes duplicates.

`echo Today is $(date)` → `$(date)` is embedding a command date

Double quotes: does not suppresses embedded commands

Single quotes: suppresses embedded commands

3 2020-01-14


3.1 Searching Text

- NAME: `grep`

SYNOPSIS: `grep [OPTIONS] PATTERN [FILE...]`

DESCRIPTION: `grep` searches for `PATTERN` in each `FILE`. A `FILE` of `-` stands for standard input. If no `FILE` is given, recursive searches examine the working directory, and non-recursive searches read standard input. By default, `grep` prints the matching lines. In addition, the variant programs `egrep`, `fgrep` and `rgrep` are the same as `grep -E`, `grep -F`, and `grep -r`, respectively. These variants are deprecated, but are provided for backward compatibility.

PATTERN examples:

- outputs on `stdout` lines that contain a match for the pattern
- case sensitive
-  \rightarrow OR
`"cs246|CS246"` \rightarrow `cs246` or `CS246` or possibly both
- `\` \rightarrow "escape" special characters
- factor stuff
`"cs246|CS246"` \iff `"(cs|CS)246"`
- `"a|b|c|d"` \iff `"[abcd]"` \rightarrow choose 1 character from this set
- `^` \rightarrow negation
`"[^abcd]"` \rightarrow 1 character *not* from this set.
`"CS24[^6]"` \rightarrow anything character except the `6` after `CS24`
- within square brackets, characters don't have their typical meanings
- `?` \rightarrow 0 or 1 occurrences of the proceeding subexpression
`"CS ?246"` \rightarrow `CS246` or `CS 246`
`"(CS)?246"` \rightarrow `CS` is optional
- `*` \rightarrow 0 or more of the proceeding subexpression
`"CS *246"` \rightarrow `CS246`, $\underbrace{\text{CS } 246}_n, n \geq 0$
- `+` \rightarrow 1 or more occurrences
`"(CS)+246"` \rightarrow $\underbrace{\text{CS } 246}_n, n \geq 1$
- `.` \rightarrow any 1 character
- `.*` \rightarrow any number of any character
`"CS.*246"` \rightarrow lines that contain substrings that contain `CS` and end with `246`
- `^` \rightarrow match beginning of line
`"^CS246"` lines that start with `CS246`
- `$` \rightarrow match ending of line
`CS246$` \rightarrow lines that end with `CS246`
`^CS246$` \rightarrow lines that *only* contain `CS246`
- words in `dict` that begin with `e` and have length `5`
`egrep "^e(.){4}" /usr/share/dict/words`
- words in `dict` that have even length
`egrep "^(..)*$" /usr/share/dict/words`

- files in current directory that have exactly one `a` in their name

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

3.2 File Permissions

- `ls -l` → long listing
- `ls -la` → long listing with hidden files

When above commands are run, in the first column there will be a sequence of 10 characters.

d	rwX	r-X	r-
---	-----	-----	----

- `d` → directory
- `r` → read
- `w` → write
- `x` → execute
- Box 2: `usr` bits, owner permissions
- Box 3: group bits
- Box 4: other bits

The owner can change perms with `chmod`.

```
chmod MODE FILEs
```

`MODE` has three subcategories:

1. ownership

```
u → usr
g → group
o → other
a → all
```

2. operator

```
+ → add permission(s)
= → set exact permission(s)
- → remove permission(s)
```

3. permissions

```
r → read
w → write
x → execute
```

Examples of `chmod`:

- `chmod g-x 1201`
- `chmod a=rx file` → set all read, execute access, take away write; there is a implicit `-w` here

- `chmod u+x shellscript`

shortcut: `chmod 744`, in binary they are corresponding to the box[2,4] above: `111 100 100`

`umask` → default permissions of a file

3.3 Shell Variables

`x=5` → sets variable `x` to `5`; can't have spaces

`echo ${x}` → prints out value of `x`; curly braces are good

Shell variables hold strings.

`dir=$(pwd)` → `dir` holds `pwd`'s value now

`$PATH` → special variable; to append stuff to `PATH` we can do `PATH=newpath:$PATH`

3.4 Shell Scripts

Text file containing Linux commands executed as a program. See `1201/lectures/shell/scripts` for some examples of shell scripts.

File: `basic`

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

- `#!` → Shebang
- `chmod a+x basic` → gives permission to execute `basic`
- `./basic` → executes `basic`

3.5 Summary of Files

Files covered in this lecture found in `1201/lectures/shell/scripts`:

- `basic`

4 2020-01-16

4.1 Shell Scripts

Shell scripts: text files containing Linux commands executed as a program. Information to a program: **arguments**, `stdin`

We can provide arguments to a script. Arguments are available in special variables named `$1`, `$2`, ...

File: `isItAWord`

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

- `./isItAWord hello` → finds `hello` in `/usr/share/dict/words`

Every process sets a status code: 0 a success, non-zero for failure. `$?` → last status code

Run: `[1 -eq 2] echo $?` → returns 0 because $1 \neq 2$

File: `goodPassword`

```
#!/bin/bash
# Answers whether a word is in the dictionary (and therefore not a good
# password)

egrep "^$1$" /usr/share/dict/words > /dev/null

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

- `/dev/null` → equivalent to discarding output

`if` statement:

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

`while` loop:

```
while [ ]; do
    ...
done
```

File: `goodPasswordCheck` → same as `goodPassword`, but checks for the correct number of arguments by adding the following (exits with a non-zero code if incorrect number of arguments are supplied):

```
if [ ${#} -ne 1 ]; then
    echo "Usage: _$0_password" >&2
    exit 1
fi
```

- `${#}` → number of arguments to the script

File: `count`

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

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

- `./count 10` → prints out numbers 1 to 10, each on a new line
- `$((x+1))` is proper addition for `int` data type

Run: `x=1`

`echo $((x+1))` → outputs `2`

`echo $x+1` → outputs `1+1`

File: `renameC`

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

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

- given a file, `mv ${file} ${file%C}cc` → renames `file.C` to `file.cc` → removes `C`, adds `cc`; that is, anything after `%` is removed

Files: `countWords`, `payday`

4.2 Summary of Files

Files covered in this lecture found in `1201/lectures/shell/scripts`:

- `basic`
- `isItAWord`
- `goodPassword`
- `goodPasswordCheck`
- `renameC`
- `count`
- `countWords`
- `payday`

5 2020-01-21

5.1 Testing

A pizza shop allows users to order pizza online and earn 10 points for each pizza ordered.

Ordering: A user types `0` followed by a number `N` to order `N` pizzas. e.g. `0 2` orders 2 pizzas

The system allows ordering between 1 to 10 pizzas. If `N` is outside this range, the system prints “Illegal order”.

On a successful order, the system display “2 pizzas ordered” followed by the total number of points they have.

Redeeming: At any time, users can type `R` to try to redeem free pizza. If the user has enough points(50), “Free Pizza!” is printed. If the user does not have enough points, “No pizza for you!” is printed followed by the number of points the user has.

Points: At any time, users can type `P` to print their current points balance.

Write exhaustive tests for this system.

Solution.

- `0 1` → 1 pizza ordered
- `0 10` → 10 pizzas ordered
- `0 0` → Illegal order
- `0 11` → Illegal order
- `1` → X
- `0 1 1` → X
- `0 5` → 5 pizzas ordered
 - `P` → 50
 - `R` → Free pizza!
 - `P` → 0
- `0 7` → 7 pizzas ordered
 - `P` → 70
 - `R` → Free pizza!
 - `R` → No free pizza for you! 20

5.2 C++ Introduction

Simula 64 → first OO language (has classes)

C with classes → C++

History: C++99 → C++03 → C++11 → C++14

In C,

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

File: `hello.cc`

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

int main() {
    cout << "Hello world" << endl;
    return 0;
}
```

5.3 iostream header

`stdio.h`, `printf`, `scanf`, `read` → not allowed in C++ (although valid)

Instead use, `std::cout << data1 << data2;`

By placing `using namespace std;`, we can say

- `cout` instead of `std::cout`
- `endl` instead of `std::endl`

5.4 Compile C++

Since we created an alias for `g++` in assignment 0, we can instead compile with simply `g++14 hello.cc`. To rename the output file we can specify the `-o` parameter.

- `g++ -std=c++14 hello.cc` → creates `a.out`
- `g++14 hello.cc -o prog.out` → creates `prog.out`

5.5 Run C++

- `./a.out` → runs `a.out`

5.6 C++ I/O

```
cout << "Hello" << "World" << endl;
```

When we create `iostream`, we get access to three stream variables.

1. `stdin`

```
std::in
type: istream
e.g. cin << x;
```

2. `stdout`

```
std::out
type: ostream
e.g. cout << "Hello";
```

3. `stderr`

```
std::err
e.g. cerr << "Error";
```

File: `add.cc`

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

int main() {
    int x, y;
    cin >> x >> y;
    cout << x + y << endl;
```

}

- If a read fails, the expression `cin.fail()` is true
- If a read fails due to EOF, then expressions `cin.fail()` and `cin.eof()` are both true

File: `readInts.cc`

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

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

- Read as many `int` s from `stdin` and output to `stdout`
- Stop if a read fails
- C++: an automatic conversion from `istream` variables to `bool`.
- `cin` is true if `cin.fail()` is false
- `cin` is false if `cin.fail()` is true

5.7 Summary of Files

Files covered in this lecture found in `1201/lectures/c++` :

- `hello.cc`
- `add.cc`
- `readInts.cc`

6 2020-01-23

6.1 C++ I/O

```
cin >> x >> y; → cin >> y; → cin;
```

If a read fails, all subsequent attempts to read fail, unless you acknowledge that failure.

Read and print all ints from `stdin`. Terminate on EOF. Ignore “bad input” (non-int).File: `readInts5.cc`

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

int main () {
    int i;
```

```

while (true) {
    if (!(cin >> i)) {
        if (cin.eof()) {
            break; // nothing left to read
        } else { // read failed, infinite loop without below
            cin.clear(); // set flag to false
            cin.ignore(); // ignore next character
        } else {
            cout << i << endl;
        }
    }
}
}
}

```

6.2 C++ Strings

In C, we used null-terminated character arrays. In C++, we have a string type, header: `<string>`

`string str = "hello";` → creates a null terminated string in both C and C++

C++ strings automatically resize.

	C	C++	When we do <code>s1 + s2</code> , we create a new string. If we wanted to append s1 to s2, we can do <code>s1 = s1 + s2</code> .
comparisons	<code>strcmp(s1,s2)</code>	<code>s1 == s2</code>	
length	<code>strlen(s)</code>	<code>s.length</code>	
concatenation	<code>strcat(s1,s2)</code>	<code>s1 + s2</code>	

File: `readStrings.cc`

- reads until first whitespace (ignore all whitespace until first character)

`getline(cin,s)` → reads until a new line

In C, we used `%d`, `%x`, etc. for `printf`. In C++, we can use the `<iomanip>` header as follows:

```

int x = 24;
cout << x; // prints 24 in decimal
cout << hex; // switch cout to hexadecimal
cout << x; // prints 24 in hexadecimal
cout << dec; // switch cout to decimal

```

`<fstream>` → input file stream

`<ofstream>` → output file stream

File: `fileInput.cc`

```

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

int main () {
    ifstream file{"suite.txt"};
    string s;

```

```

    while (file >> s) {
        cout << s << endl;
    }
}

```

`ifstream file {"suite.txt"};` → uniform initialization syntax, only in $\geq C++11$
type variable initialization of variable

Other examples of uniform initialization syntax:

- `int x{5};`
- `string s{"hello"};`

`<sstream>`

- `istringstream`
- `ostringstream`

File: `buildString.cc`, `getNum.cc`

6.3 Summary of Files

Files covered in this lecture found in `1201/lectures/c++/2-io`:

- `readInts5.cc`
- `readStrings.cc`
- `fileInput.cc`
- `buildString.cc`
- `getNum.cc`

7 2020-01-28

7.1 Short C++ Topics

File: `readInts5.cc`

- Read as many inputs from stdin, int output, bad int ignore
- Terminate when done receiving input

File: `readIntsSS.cc`

```

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

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

```

```

    }
}

```

- Difference: ignores entire string if read fails, e.g. `hello` would ignore `hello`, but `readIntsSS.cc` discards entire string

7.2 Default Arguments

```

void print (string name = "suite.txt" ) { // set default value
    string s;
    ifstream f{ name };
    while (file >> s) {
        cout << s << endl;
    }
    print("filename");
    print(); // default value
}

```

- Default arguments must come last, e.g. `void test(int x = 0, string str);` → illegal because `string` has no default value but comes last
- `void test(int x = 0, string str = "hello");` → legal, can be called in three different ways as denoted “legal” below.
- When calling the function, we can use default values for the last N parameters, e.g. →

```
test(); → legal;
```

```
test(s); → legal;
```

```
test(s, "bla"); → legal;
```

```
test( , "bla") → illegal;
```

```
test("bla"); → illegal;
```

- We cannot implement `void test();`, or `void test(int);`, or `void test(int, string)`
- In C, function names must be unique. In C++, we can have functions with the same name, but they must differ in the number of types of parameters, called **function overloading**.
- **Signature**: name of function, types and number of parameters.
- The return type of a function is not part of the signature.
- A new function: `void test(int, int);` → legal, does not conflict with `test(int, string);`
- A new function: `void test(string);` → legal

```

int a = 21; // 10101
int b = 3;
a = a << b; // 10101000

```

- left shift operator (bit shift): multiply by 2^b ;
- right shift operator: divide by 2^b

- `<<` is a overloaded operator. Overloading is a type of function Overloading.
- C++ allows us to define the meaning of operators for user-define types (all types not built-in); `istream` → `iostream` header.

```
int x;
cout << x;
string s{"hello"};
cout << s;
```

- these operators are **overloaded**, because `<<` can work differently for `string` and `int`
- example of function overloading since `int x` and `string s` call different functions

```
int a, b;
a + b;
string c, d;
c + d;
```

- operators are overloaded as `+` differs depending on the type

7.3 Structs

In C,

```
struct Node {
    int data;
    struct Node *next;
};
struct Node n = {3, NULL};
```

- In C++, you don't have to write `struct` before `Node` after defining `Node` as a struct
- Can remove the `=` before `{}` for uniform initialization
- Discouraged to use `NULL` constant, use `nullptr` instead

7.4 Constants

```
const int MAX = 10;
```

- Constants must always be initialized
- `const Node n{3, nullptr}` → 3 data, `nullptr` next
- `n.data` or `n.next` → illegal

```
int n = 5;
const int *p = &n;
```

- `p` is a pointer to an `int` which is `const`; pointer is not `constant`

```
p = &m; → legal
```

```
*p = 10; → illegal, but can still do n = 10;
```

- `int *const q = &n;` → `q` is a `const` ptr to an `int`
`q = &m;` → illegal
`*q = 10;` → legal
- `const int *const r = &n;` → `r` is a `const int` to a ptr that is `const`
- `const` applies to the “thing” on the left, unless there is nothing to the left, in which case it applies to the right

7.5 Parameter Passing

```
void inc(int x) { // copied passed by value
    x = x + 1;
}
int x = 5;
inc(x); // does not actually modify x
cout << x; // prints 5
```

Passing pointers:

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

<code>scanf("%d", &x);</code>	<code>cin >> x;</code>
	operator >> (cin, x);
	pass by reference

7.6 Lvalue References

Informally, an **lvalue** is anything that can appear on the left hand side of an assignment.

- `x = 5;` → `x` is an lvalue
- `5 = 7` → not an lvalue
- `x + y = 5` → not an lvalue
- `str[i] = '3';` → `str[i]` lvalue

Formally, an **lvalue** is a storage location, something whose addresses we can obtain.

```
int y = 10;
int &z = y;
```

- `z` is an lvalue reference to `y`
- `z` acts as a constant pointer to `y` with **automatic dereferencing**
- `z = 20;` → automatically dereferencing, don't need `*z` (actually a compile error); changes `y` to 20

- `z` becomes an alias to `y`
- `int *p = &z` → gets `y`'s address
- `z` behaves like `y`
- `int &` is a type, not an address

References must be initialized to lvalues.

- `int &z;` → illegal
- `int &z = 3;` → illegal
- `int &z = a + b;` → illegal
- Cannot create a pointer to a reference.
- Cannot create a pointer to an array of reference.
- Cannot create a pointer to a reference to a reference.

```
void inc(int &n) { // n is an alias for x
    n = n + 1; // n acts like a constant pointer to x with automatic dereferencing
}
int x = 5;
inc(x); // x is passed by reference
cout << x; // prints 6
```

`cin >> x` → `x` is passed by reference

8 2020-01-30

8.1 More Short C++ Topics

Why does `cin >> x` work?

`operator >> (cin, x);` → `x` is passed by reference

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

We take the stream by reference since

1. we want changes to the stream be changes to `cin`.
2. streams cannot be copied.

Pass by value vs pass by reference

```
struct ReallyBig {...};
void f(ReallyBig rb); // pass by value, costly
void g(ReallyBig &rb); // pass by reference: avoid copy
void h(const ReallyBig &rb); // avoid copy, h cannot change what rb refers to
```

Advice: prefer to pass arguments as reference to `const` for anything bigger than an `int`.

```
void f(int &n);
f(5); // 5 is not an lvalue; illegal
f(x + y); // illegal
```

```

int temp = x + y;
f(temp); // legal
// pass by reference to constant
void g(const int &n);
g(5); // legal
g(x + y); // legal

```

8.2 Dynamic Memory Allocation

- In C, `int *p = malloc (sizeof(int) * length);`, `free(p);`.
 - In C++, `malloc`, `free`, `realloc` → banned
-

```

// {data, next};
Node n{5, nullptr}; // n is on stack
// np is on stack, pointing to the heap
Node *np = new Node {2, nullptr}; // on heap, new figures out how much memory is
    needed
n.next = np;
delete np;

```

Stack allocated nodes:

```

Node myNodes[10];
Node *np = new Node[10];
delete np; // undefined behaviour
delete []np; // syntax to deallocate an array

```

```

// return by value, copy is made
Node getNode() {
    Node n;
    ...
    return n;
};
// compiles, "Dangling Pointer", but incorrect
// to correct: heap allocate + return pointer to heap node
Node *getNode() {
    Node n;
    ...
    return &n;
}

```

8.3 Operator Overloading

Define the meaning of C++ operators for user-defined types.

```

struct Vec {
    int x;
    int y;
};
Vec v1{1, 2};

```

```

Vec v2{1, 2};
Vec v3 = add(v1, v2); // In C, we would do this

// Overload the + operator such that 'v1 + v2' works
Vec operator+(const Vec &vec1, const Vec &vec2) {
    Vec toRet{vec1.x + vec2.x, vec1.y + vec2.y};
    return toRet;
}

Vec v4 = v1 + v2 + v3; // works, wouldn't work without const

// Overload the * operator such that 'c * v' works
Vec operator*(int c, const Vec &vec) {
    return {c * vec.x, c * vec.y};
}

Vec v = 3 * v2; // works
Vec v = v2 * 3; // doesn't work

Vec operator*(const Vec &vec, int c) {
    return c * vec;
}

```

```

struct Student {
    int grade;
};
Struct s{15};
cout << s.grade << "%";

// Overload the << operator such that 'cout << s' works
ostream operator<<(ostream &out, const Student &s) {
    out << s.grade << "%";
    return out;
}

// Overload the >> such that 'cin >> s' works
istream &operator>>(istream &in, Student &s) {
    int n;
    in >> n;
    s.grade = (n < 0) ? 0 : n; // short hand if statement
    s.grade = (s.grade < 100) ? 100 : s.grade;
    return in;
}

```

9 2020-02-04

9.1 Preprocessor, Separate Compilation

Source code → Preprocessor → Compiler → `a.out` executable

`#include` is a direct copy/paste for the preprocessor directive

`#include` with quotes → look in current directory, e.g. `#include "file"`

`g++ -E FILE` shows what the `include` copy/pastes

1201/lectures/c++/4-preprocess

File: `hello.cc`

- `#define VAR VALUE` → searches and replaces `VAR` with `VALUE`
`#define MAX 10` → replaces `MAX` with `10`
 Obsolete because now we have `const`

9.2 Conditional Compilation

File: `course.cc`

`g++14 -DVAR=VALUE FILE` → changes type in command-line for `course.cc`

Preprocessor comment; nests perfectly:

```
#if 0
...
#endif
```

Block comments (less powerful):

```
/*
...
*/
```

```
#define VAR // VAR gets empty string
#ifdef VAR // if VAR is defined, true
#ifndef VAR // if VAR is not defined, true
```

File: `debug.cc`

`g++14 -DDEBUG debug.cc` → defines `DEBUG`, so one can see full verbose due to the `#ifdef` s in `debug.cc`

Note: `-D` can be used to define multiple variables

9.3 Separate Compilation

- Header files (`.h`): Declarations of functions, Global Variables, and Type Definitions
- Implementation files (`.cc`): Definitions of functions

1201/lectures/c++/5-separate

File: `example1`

Compile the files with either:

- `g++14 main.cc vec.cc`
- `g++14 *.cc`

Notes:

- Implementation files (`.cc`) are never `include` d

- Implementation files (`.cc`) are compiled
- Header (`.h`) files are **never compiled**, they are `include d`

Want to compile each file separately to produce a position of the executable, then finally merge these positions.

By default, `g++` will compile and link to produce the executable.

`g++14 -c vec.cc , g++14 -c main.cc` → compiles each separately, without the executable (`.o` (Object) files are produced).

`g++14 main.o vec.o -o myprog` → merges them into the `myprog` executable

9.4 Build Tools

Don't memorize any of this section.

make: Automatically use "last modified timestamp" (uses `ls -l`)

Specify dependencies in a `Makefile` .

`1201/lectures/tools/1-make`

File: `example1`

- `.PHONY` → specifies that `clean` is not a file, but a command
- `clean` → `make clean` will delete any `.o` files in the current directory

File: `example2`

- Uses variables for compilation
- `-Wall` → warn all, compiler will give errors for warnings

File: `example3`

- `main.o` , `vec.o` , `myprog` within the `Makefile` is all one needs to change for A2Q5

10 2020-02-06

10.1 Preprocessor, Include Guards, C++ Classes

File: `example3`

- won't compile as `vec.h` gets included twice
- use an include guard to prevent multiple includes

File: `example4` → fixes the issue above

In `vec.h` ,

```
#ifndef VEC_H // true
#define VEC_H
struct Vec {
    ...
}
#endif
```

Never put `using namespace std;` in a header file since it forces others to use the `namespace`.

10.2 C++ Classes

A C++ class is a `struct` that may contain functions.

The big innovation of OOP: `struct`s can have functions.

File: `student.h`

```
struct Student {
    int assign, mt, final;
    // since grade is only relevant for this function, we declare it here
    float grade(); // good style to have declaration only, and not the entire thing
};
```

File: `student.cc`

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

`std::ostream` → In the scope of the standard namespace, there is an `ostream`. Above, the same thing is happening.

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

```
// Bobby is an object.
Student Bobby{75, 50, 65};
// Let's compute Bobby's grade.
cout << Bobby.grade();
```

- A function within a class is called a **member function** or **method**.
- You can only call methods using objects of the class.
- All methods have a hidden parameter named `this` → a pointer to the object used to call the method.

```
this == &bobby
```

`ptr -> field` is the same as `(*ptr).field`. Equivalently as in `student.cc`.

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

10.3 Initializing Objects

C style initialization:

`Student Bobby = {75, 50, 65}` → not going to use this syntax, but it's allowed.

In C++:

Special methods to construct objects are called **constructors**, they do not need a return type.

Header file (`.h`):

```
// same name as class
struct Student {
    ...
    Student (int assign, int mt, int final); // declaration, no return type
};
```

Implementation file (.cc):

```
Student::Student (int assign, int mt, int final) {
    // cannot do assign = assign;
    this->assign = assign < 0 ? 0 : assign; // short hand if statement
    this->mt = mt;
    this->final = final;
}
Student s1{70, 60, 75};
Student s2{70, 60}; // final = 0
Student s3{70}; // mt = final = 0
Student s4{}; // assign = mt = final = 0
Student s5; // equivalent to Student s4
```

Older initialization:

```
Student Bobby = Student(75, 50, 65);
```

Heap allocated Student :

```
// round or curly braces acceptable, but curly braces is good style
Student *p = new Student{75, 50, 65};
...
delete p;
```

Default constructor: A zero parameter constructor. Alternatively, it is a constructor where all parameters have default values.

Every class comes with a built-in/free default constructor. It calls default constructors on any fields that are objects.

```
struct MyClass {
    int x;
    Student s;
    Vec *p;
};
Myclass a;
```

For `a`, the default constructor:

- initializes `s` as it is an object `Student`.
- **does not** initialize `p` as is not an object, but a **pointer to an object** `Vec`.
- **does not** initialize `x`.

As soon as you write any constructor, you lose the built-in default constructor and C style initialization, for example:

```

struct Vec {
    int x, y;
    Vec (int x, int y) { // bad style
        this->x = x;
        this->y = y;
    }
};
Vec v; // does not compile

```

Initializing constant fields:

```

int m;
// probably not what you want, all objects here have a constant id of 10
struct MyClass {
    const int id = 10; // in class initialization
    int &n = m;
};

```

```

struct Student {
    const int id; // want to figure out how to do this, next class
};

```

11 2020-02-11

11.1 C++ Classes (Cont.)

Last time: Initializing objects

```

struct Student {
    const int id;
};

```

How to initialize fields that are constants or lvalue references?

Fields must always already be initialized before constructor body runs; i.e. step 2 occurs before step 3 below

Steps for Object Construction

- (1) allocate space: could be on stack or could be on heap if new
- (2) field initialization: only fields that are **objects** are initialized
their default constructor gets called
- (3) constructor body runs

‘Hijack’ (2): Use **Member Initialization List** (MIL)

Example (4 fields, 4 parameter constructor):

```

Student::Student(int id, int assigns, int mt, int final)
    // pairs separated by commas
    // before curly brace are field names

```

```
// initialize id to id, assigns to assigns, ...
: id{id}, assigns{assigns}, mt{mt}, final{final} {
    ... // constructor body
}
```

MIL should use field declaration order; will compile with a warning if you do not do this, but it will initialize according to field order.

Not that we did not need to use `this` to disambiguate field/parameter.

MIL is necessary to initialize constants and references that can be used for all fields.

Using the MIL can be more efficient than using the constructor body.

Special constructor that takes a single parameter: Constructing objects as copies of other objects

```
Student billy{75, 50, 65}; // three parameter constructor {assigns, mt, final}
Student bobby{billy}; // Uses copy constructor
```

Constructing an object as a copy of another called the **copy constructor**. There is always a built-in constructor for any class.

(.cc) (You get this for free already)

```
// crucial to be by reference; will not compile if done by value
Student::Student(const Student &other)
: assigns{others.assigns}, mt{others.mt}, final{others.final} {}
```

When is a copy constructor called?

- (1) Explicitly constructing an object as a copy of another
- (2) Pass by value
- (3) Return by value

Every class comes with (gets for free):

- (I) default constructor
- (II) copy constructor
- (III) copy assignment operator
- (IV) destructor
- (V) move constructor
- (VI) move assignment operator

The **Big 5**: (II)-(VI).

```
struct Node {
    int data;
    Node *next;
    // two parameter constructor
    Node(int data, Node *next);
    // free copy constructor
    Node(const Node &other);
};
Node::Node(int data, Node *next) : data{data}, next{next} {}
```

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

Shallow copy:

```
Node *n = new Node{1, new Node{2, new Node{}, nullptr{}}};
// Three heads with shared tails, this is bad
// stack allocated node
// m on stack with head same as n's head, but tail points to n's tail (on heap)
Node m{*n}; // make a copy of *n and make it m
// heap allocate the copy
// p on stack, same as m, but with first node in heap
Node *p = new Node{*n};
```

Sometimes we want a **deep copy** (need to write your own copy constructor):

```
Node::Node(const Node &other) {
    data = others.data;
    // INCORRECT, segmentation fault if next is nullptr (dereferencing nullptr)
    next = new Node{*other.next};
    // CORRECT
    if (others.next) {
        next = new Node{*other.next};
    } else {
        next = nullptr;
    }
}
```

More compactly,

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

One parameter constructors create “implicit” conversions.

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

```
void foo(Node n) {
    ...
}
Node n{4};
Node n = 4;
// Both possible, implicit conversion
foo(n);
foo(5);
```

We can declare a constructor “explicit” to disable implicit conversions:

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

11.2 Destructor

Destructor: method that gets called when objects are destroyed.

- Stack: when stack is popped
- Heap: when delete is called on a pointer to a heap allocated object

A class can only have one destructor, they cannot be overloaded.

Steps for Object Deconstruction

- (1) Destructor body runs
- (2) Fields that are objects are destroyed (reverse declaration order)
- (3) Space is reclaimed

Free destructor has an empty destructor body.

```
// Allocate everything on heap
Node *p = new Node{1, new Node{2, new Node{ }, nullptr}};
delete p; // leaks Node 2 and Node 3
```

(.h)

```
struct Node {
    ~Node();
};
```

(.c)

```
Node::~~Node() {
    delete next; // recursively deletes next (including nullptr which is safe)
}
```

11.3 Copy Assignment Operator

```
Student billy{..., ..., ...};
// Bobby was born a cheater
Student bobby{billy}; // done already
// Jane existed, then started cheating off Billy
Student jane; // default constructor
jane = billy;
jane.operator=(billy); // equivalent to line above (don't do this)
```

The **Copy Assignment Operator** is used to assign to existing objects.

12 2020-02-13

12.1 Big 5 (Cont.)

```

a = b = c = 0; // right associative (first: c = 0)
cout << a << b; // left associative (first: cout << a)
// n1, n2, n3 are nodes
n1 = n2 = n3; // n2 = n3 is done first
// n1.operator=(n2.operator=(n3))

```

`.h`

```

struct Node {
    ...
    Node &operator=(const Node &);
};

```

`.cc`

```

// Node& Node and Node &Node are equivalent
Node &Node::operator=(const Node &other) {
    if (this == &other) {
        return *this;
    }
    data = other.data;
    // next might already point to heap nodes; must deallocate those
    delete next;
    // *other.next -> copy constructor
    next = other.next ? new Node{*other.next} : nullptr;
    return *this;
}

```

Case (self assignment):

```

Node n{...};
Node &m = n; // self assignment
n = m; // breaks w/o first if-statement of above

```

If new fails (no more heap memory), next becomes a dangling pointer.

`.cc`

```

Node &Node:: operator=(const Node &other) {
    if (this == &other) {
        return *this;
    }
    Node *temp = next;
    // Exception safety
    next = other.next ? new Node{*other.next} : nullptr;
    data = other.data;
    delete temp;
    return *this;
}

```

12.2 Copy and Swap Idiom

node.h

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

.cc

```
#include <utility>

// not const, hint to update other
void Node::swap(Node &other) {
    using std::swap;
    swap(data, other.data);
    swap(next, other.next);
}

Node &Node::operator=(const Node &other) {
    Node temp{other}; // calls copy constructor
    swap(temp);
    // temp is destroyed automatically b/c stack allocated (calls destructor)
    return *this;
}
```

1201/lectures/c++/6-classes/rvalue

node.cc

Trace of node.cc

Node n...;

(1) Create linked list with two nodes. The basic constructor is called twice as there are two nodes.

Node n2plusOne(n);

(1) Evaluate plusOne(n), which takes n by value; that is the copy constructor is called twice as n is passed by value.

(2) plusOne(n) is returned n2 is constructed, the copy constructor is called twice again.

(3) Node n2 calls the copy constructor twice for plusOne(n)'s value.

Total: 6 calls to copy constructor, 2 calls to basic constructor.

The returned value from plusOne(n) is temporary only alive until n2 has been constructed; an rvalue. For a simpler example,

```
int x = ...;
int y = ...;
int z = x + y; // x + y is an rvalue which is temporary
```

Copy constructor: copies from objects that will continue to live

Move constructor: steals from objects that are about to die (temporaries, rvalues)

In C++, we can use rvalue references to refer to temporaries. `Node&` lvalue reference, `Node&&` is a rvalue reference

```
Node::Node(Node &&other)
// n2 and temp 'share' next, which is bad because when temp dies => dangling pointer
: data{other.data}, next{other.next} {
    other.next = nullptr;
}
```

```
// MIL only available to constructors, can't use it here
Node &Node::operator=(Node &&other) {
    swap(other);
    return *this;
}
```

Compiler Optimization: Copy/Move Elision, which can be turned off with `-fno-elide-constructors`.

The compiler is allowed to avoid copy/move constructor calls even if this changes program behaviour.