University Of Waterloo

CS-246

Object-Oriented Programming

Edited By Yiming, Dai © at Github

Last Revision: December 24, 2019

Table of Contents

Object-Oriented Programming **Fall 2019**

Chapter 01 - Linux Shell

Professor: Nomair *Author: Yiming Dai*

> Shell is program to interface with an OS

> Graphical Shells

+ intuitive/easy to learn
- inflexible

> Command Line Shell

+ modules/flexible
- you get a command prompt where you type commands
- steep learning curve

> Linux File System

– files that can contain after files are called directories;
– files are organized in a tree structure;

> Path is the location of a file in the file system;

> **ls** - listing of non-hidden files with formatted display (i.e. more spaces between names);

> Hidden files in Linux - name begins with a **.**

> **pwd** - present working directory;

> Special Directory

– **.** => current directory
– **..** => parent directory
– **../..** => grandparent directory
– **~** => home directory
– **-** => back to last directory
– **~userid** => user's home directory

> **cat**(concatenated) => display a text file

> **[Ctrl + C]** => kill the program

> **[Ctrl + D]** => send **E**nd **O**f **F**ile signal

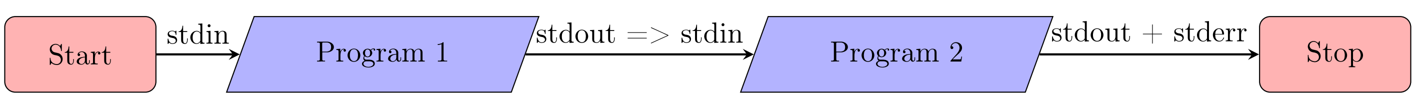
> **man** => show the manual of the command

IO redirection

- Output Redirection
  - **cat > output.txt** => Output redirection to a text file [May overwrite]
  - **cat >> output.txt** => Output redirection to **append** words to a text file
- Input Redirection
  - **cat input.txt** => cat Open file => Read => Display
  - **cat < input.txt** => **Shell** not cat to Open file => Read => Send content directly to cat => Display
- **cat -n < sample.txt > tmp.txt** => Input/Output redirection
- **myprog < in.txt > out.txt 2> err.log** => Order does not matter
- **myprog > out.txt 2>&1** => Connects stdout with stderr to be out.txt
- **&1** => First output stream
- **myprog 2>&1 > out.txt**
- Every program has 3 streams
  - **Standard Input = stdin** => default: keyboard | Use < to change to file
  - **Standard Output = stdout** => default: screen | Use > to change to file
  - **Standard Error = stderr** => default: screen | Use 2> to change to file

Wildcard Matching

- **prog1 | prog2** => prog1 > temp and prog2 < temp
  - **ls *.txt** => match any file end with **.txt** [**Shell** intercepts the Globbing Pattern, and substitutes the globbing pattern with filename that matches]
- Linux pipe | connects stdout of prog1 to stdin of prog2



pipe

- **prog1 2> err.log | prog2 2>> err.log** => Input redirection errors of two progs

  Example.

  Q: Count number of words in the first 20 lines in sample.txt
  A: **head -20 sample.txt | wc -w**

  Example.

  Q: Suppose files words1.txt words2.txt etc. contain lists of words, 1 per line. Print a duplicate free list of words in words*.txt
  A: **cat words*.txt | sort | uniq**

Output as arguments

> Use **$(⋯) to embed a command within another command**

> **echo "Today is $(data) and I am $(whoami)"** => Single argument (MUST double quote)

> **echo Today is $(date) and I am $(whoami)** => Multiple arguments

> Use double quptes to indicate an argument that contains space

> Single quotes will suppress embed commands

egrep

> **grep -E = egrep** => egrep pattern files

> **egrep** goes **line by line** to match pattern;

> **egrep "cs246 | CS246" = egrep "(cs|CS)246" ≠ egrep "(c|C)(s|S)246 = egrep [cC][sS]246**

> Use \ to escape any characters with special meaning;

> **[abc]** ≡ **(a|b|c)** => choose a single charfrom set

> **[^abc]** => choose one char **NOT** in set

> **?** => 0 or 1 occurence of preceeding subexpression

> ***** => 0 or more occurence of preceeding subexpression

> + => 1 or more occurence of preceeding subexpression

> **.** => any one character

> **.*** => 0 or more characters

> **cs.*246** or **.*cs.*246.*** => output lines containing a substring cs followed by 0 or more chars followed by 246

> **^cs.*246** => force the line to start with **c**

> **cs.*246$** => force the line to end with **6**

> **egrep "^(..)*$"**

> **ls -a | egrep "^[^a]*a[^a]*$** => list all files in current directory whose name contains exactly one **a**

File Permission

> When use command **ls -l**

- the first position is **ordinary file** => **d** if directory; **-** if file
- followed by three groups (**u**ser, **g**roup, **o**ther, **a**ll)
- operate with +|−|= => notice that = may have implicit operation

Shell Variables

> **x=1** => all shell variables are strings (no spaces between)

> **echo $x** => echo $x

> **$PATH** => a list of directory paths system used to search commands separated by :

> **PATH=$PATH:/bin** => Add **/bin** to $PATH (String Concatenation)

Shell scripts

> Text files that contain bash commands executed as a program

> File: newscript

```
vi newscript # New script
#!/bin/bash
ls
whoami
date
pwd
# Exit vim
# Notice that default permission is r + w
chmod u+x newscript
# But current directory may not in $PATH, i.e. OS cannot find command newscript
./newscript # Run commands inside the file
```

> Command Line Arguments => ./newscript as $0 | arg1 as $1

> File: isItAWord

```
vi isItAWord
#!/bin/bash
egrep "^$1$" /usr/share/sict/words
# Exit vim
./isItAWord (word)
```

> Single quotes suppress variables

> File goodPassword

```
vi goodPassword
#!/bin/bash
# Answer whether a word is in the dictionary => not a good password

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

# Method 1
if [ $? -eq 0 ]; then
    echo Not a good password
else
    echo Maybe a good password
fi

```

```
# Method 2
if [ $? -eq 0 ]; then
    echo Not a good password
elif [ $? -ne 0 ]; then
    echo Maybe a good password
```

> Linux process sets/returns a status code of last executed command (hidden as $?)

> **echo $?** => Status code 0 if success

> **test 1 -eq 2** or **[1 -eq 2]** => test if 1 equals to 2, answer saved in $?

> **Spaces** need to be set between commands and arguments

> File: goodPassword (Upgraded)

```
vi goodPassword
#!/bin/bash
# Answer whether a word is in the dictionary => not a good password

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

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

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

# Method 1
if [ $? -eq 0 ]; then
    echo Not a good password
else
    echo Maybe a good password
fi
```

> File: Count

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

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

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

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

```
fi

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

> **$((x + 1))** => treat x as arithmetic argument and do arithmetic operation instead of string concatenation

> File: group rename files

```
#!/bin/bash
# rename all .c files to .cc

# Examples of "for"
# for filename in $(ls); do echo $filename; done => print all files in current directory
# for filename in add*; do echo $filename; done => print all files start with add in current directory

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

> Double quotes make a content to a single argumetn like a single stirng

Object-Oriented Programming **Fall 2019**

Chapter 02 - C++

Professor: Nomair *Author: Yiming Dai*

I/O

> When we include **iostream** we can produce output;

```
std::cout << data1 << data2;
std::cout << std::endl
```

> When we use 'using namespace std', we cannot use our own defined 'cout'

> Compile using **g++ -std=c++14 hello.cc -o myprog** (**-o** means executable)

> When we include iostream we are able to access 3 global variables

i. **std::cout** (struct ostream) => connects to stdout with output operators **cout** $\ll \cdots$

ii. **std::cerr** (struct ostream) => connects to stderr with output operators **cerr** $\ll \cdots$

iii. **std::cin** (struct istream) => connects to stdin with input operators **cerr** $\gg \cdots$

> File: Plus.cc => Reads 2 ints and prints the result of addition

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

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

> **cin** will read from the first non-whitespace char until it hits a whitespace

> When reading from input, it's a good idea to check if a **read** successful

> If a read fails, the expression **cin.fail()** is going to be true

> If a read fails due to **EOF**, cin.fail() and **cin.eof()** are to be true

> File: io/readInts.cc => Read all ints from stdin and echo one per line to stdout. Stop if a read fails

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

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

> C++ defines an automatic conversion from istream (type for cin) to bool;

> cin is treated as true if the last read succeeded, false otherwise;

> File: io/readInts_final.cc

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

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

> If a read fails, all subsequent reads fail * (unless you ackowledge the failure)

> File: io/readInts_err_check.cc

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

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

> **cin.ignore()** => ignores one character at a time

String

> In **C**, we use characters arrays (null terminated) to represent strings;

> In **C++**, we use header <**string**>; Not a array;

	C	C++
Comparison	strcmp	==, !=, <, >
Length	strlen	str.length();
Concatenation	strcat(s1, s2)	s1 = s1 + s2;
Character Access	s[i]	s[i]

> C++ use I/O manipulators as placeholders in C

```cpp
int x = 95;
cout << x;          // prints x in decimal
cout << hex << x; // prints x in hexadecimal
cout << dec;        // back to default print option
```

> In A2, use iostream, iomanip, showpoint, setprecision(3), boolalpha

> EVerything we learnt about reading/writing from/to stdin/stdout applies to other sources of data.

> Anything you can do with istream variables (e.g. cin) you can do with an ifstream variables;

> Anything you can do with ostream variables (e.g. cout) you can do with an ofstream variables;

> Use <**fstream**> to read and write files

- **ifstream** => input file stream
- **ofstream** => output file stream

> The file is closed using ifstream when the variable goes out of scope (e.g. stack pop up)

> We can connect streams to strings;

> Use <**sstream**> to read in and write to a string by **i/ostringstream**

> Default Arguments

```cpp
void printFile(string file = "t.txt") {
// code
}
printFile("a.txt");
printFile(); // default t.txt
# Default arguments must come last;
# If we omit arguments, we must omit the last one
```

Function Overloading

> In C++ we write functions of same name as long as the numbers or types of arguments have a difference

```cpp
# Legal in C++ but not in C
int neg(int x) { return -x; }
bool neg(bool b) { return !b; }
# cin >> a is equivalent to operator >> (cin, a) [a is pass by reference]
```

> If multiple functions with the same name appears, 'func()' will not be compiled;

> 'const int *p = &n' means p is a point to int which is a constant

New Syntax

```
struct Node {
int data;
struct Node *next;
};
Node n = {5, NULL};

# Constants
const int *p = &n;
p = &x; [Legal]
*p = 5; [Illegal]

int *q = &n;
*q = 2; [Legal]

int y = 42;
p = &y;
*p = 5; [Illegal]

int * const r = &n; # r is a constant pointer to integer
r = &y; [Illegal]
*r = 1; [Legal]

int const * p = &n; # p is a pointer to a constant integer
```

Reference

```
int y = 10;
int &z = y;
# z is an lvalue reference to y; it acts like a const. ptr. to y;
# A lvalue reference is a const. ptr. with [automatic dereference];
# which means no need to write '*z=15' but rather 'z=15'

int *q = &z;
# above create a ptr. to the integer
# z acts like y (a alias to y)
# thus size(z) = size(y)

# Cannot create a reference to a reference
# Cannot create a ptr. to a reference but can create a reference to a ptr.
# Cannot create an array of reference
# References must be initialized;
# References must be initialized to something with an address
int &z = 5; [Illegal]

# Lvalue is (1) a storage location and (2) something with an address
# Lvalue => anything that can be written on the LHS of an assignment
# Rvalue => anything not an lvalue | temp/computed value
void inc(int &x) {
x = x + 1;
}
int y = 5;
inc(y);
cout << y; => print 6

```

```
std::istream &operator >> (std::istream &in, int &x);
# return type is std::istream since may need consecutive reading
# cin >> x >> y
$ streams cannot be copied

struct ReallyBig{};
void f(ReallyBig rb);
ReallyBig b = ___;
f(b) => requires create huge stack frame or pass by value of b

# C++ could pass by ptr. to avoid copy and pass by reference by below
void g(ReallyBig &rb);
g(b); => within g, rb is another name for b
# changes made by g will be cisible outside g

void h(const ReallyBig &rb); => no copy | h cannot modify the original
# Pass by reference to const whenever possible for anything bigger than an Int
```

Dynamic Memory Allocation

```
int * ptr = (int *) malloc (10 * sizeof(struct Node));
free(ptr);
# C only works (void *) pointers

# C++ version => new, delete, adn nullptr
int * ptr = new int;
*ptr = 5;
int * p2 = new int{13};
delete ptr;
delete p2;
# If delete a nullptr, does nothing

# Two dimensional array
int ** arr;
arr = new int * [10];
for (int i = 0; i < 10; ++i) {
  arr[i] = new int[5];
}
for (itn i = 0. i < 10; ++i) {
  delete []arr[i]; # Empty square bracket to delete all the elements but not only the first element in the address
}
delete []arr;
# If allocated using new ...[...], delete using []
# Do not delete the same memory address twice
# Do not refer to already freed/out of scope addresses
# If you think you might be doing that, set it to nullptr to have a better chance of catching the problem
```

Return by value, pointer, or reference

```
# Let's have a function to build nodes
Node getNode() { # returned value is an "rvalue"
```

```
  Node n;         # being "copied" into "m", but as of C++11, not actually this inefficient
  return n;
}
Node m = getNode();
# See more about move semantics versus copy semantics

# Better method below
Node * getNode () {
  return new Node; # Have to remember and free it
}
Node * ptr = getNode();
delete ptr;
# If used (Node &) same problem; i.e. bound to variable that no longer exists

# Best way
Node & getNode () {
  Node * ptr = new Node;
  return * ptr;
}
Node & n = getNode();
delete & n;

# Q: which to use?
# A: return by value; if can, since nto actually as expensive as it looks (at least in C++11 and up)
```

Operator Overloading

```
# I/O operators we have already seen use pass by ref. and return by ref.
cin.operator >> (int)
std::istream & operator >> (std::istream & in, int & i);
std::ostream & operator << (std::ostream & out, const int & i);

std::ostream & operator << (std::ostream & out, const Node * n) {
  if (n != nullptr){
    out << n->value;
  }
  return out;
}
3 If not modifying the parameters, often want to return by value;
```

Preprocessor

```
#define MAX 10 // Not type safe, use global constant instead;
```

Tutorial

> ‘std::string str1;‘ is equivalent to ‘std::string str1 = ”;‘

> ‘using std::string;‘ => only use string part, better to debug;

Separate Compilation

> Interface/headers (.h)
  - type definitions
  - function declarations

> Implementation (.cc)
  - function defintions

> *main.cc* and *vec.cc* all include *vec.h*

> Never compiled *.h* files

> Never include a *.cc* file

> By default, the compiler compiles links and produces an executable file;

> To just compile **g++ -c** file.cc produces *.o* (object) files Object files contain
  - compiled binary
  - a list of things that were **defined**
  - a list of things that were **required**

> Use a linker to merge object files **g++ main.o a.o -o a.out**

> Separate compilation allows us to only recompile files that have changed and then link everything

> Tools => /lectures/tools/make

> **extern int global** in *abc.h* and **int global** in *abc.cc*

Include Guard

> Prevents a header file from being included multiple times

> Every header file should have an include guard

C++ Classes

```
// student.h
#ifndef STUDENT_H
#define STUDENT_H
struct Student {
  int assns, mt, final;
  float grade();
}
#endif

// student.cc
#include "student.h"
float Student::grade() {    // scope resolution => "In the scope of"
```

```
  return assns * 0.4 + mt * 0.2 + final * 0.4;
}

Student billy {80, 50, 70};  // must be constants
cout << billy.grade() << endl;
```

- A class is any struct that can have functions;
- In C++ all structs are allowed to have functions, so they are all classes;
- An instance/value of a class is called an object;
- A function written inside a struct is called a members function => Method
- Calling a method requires an object of that class => within the method we have access to the fields of the object on which we called the method
- Every method has a hidden parameter named **this**
  i. *this* == &*billy*
  ii. **this* == *billy*
- C++ allows to write methods to construct objects => **constructors (ctors)**

```
struct Student {
  Student(int assns, int mt, int final); // no return value
  // constructors have the same name as the class
}

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

Student billy{80, 50, 70};

Student *pBilly = new Student{80, 50, 70};
delete pBilly;
```

- Advantages of ctors
  - sanity check
  - can overload
  - default value for parameters

```
struct Student {
  Student(int assns = 0, int mt = 0, int final = 0);
}

Student::Student(int assns, int mt, int final) {
  this->assns = assns < 0 ? 0 : assns;
  this->mt = mt < 0 ? 0 : mt;
  this->final = final < 0 ? 0 < final;
```

```
}

Student s1{80, 45}; // final = 0;
Student s2{}; // all are 0
```

Default/Zero Parameter Ctors

> Every class comes with a default ctor

```
struct Mystud {
  int a;
  int *b; // not initialized by default ctor
  Student c; // default ctor of Student is called by the default ctor of Mystud
}
```

> Rule 1: If we write our own ctor, we lose the default ctor

```
struct Vec{
  int x;
  int y;
  Vec(int, int); // two integer value default to 0
}
// Valid => Vec v{1, 2};
// Invalid => Vec v; // no default ctor
```

> Initialize fields that are const/reference

```
int x;
struct MyStruct{
  const int y = x;
  int &z = x;
}

struct Student{
  const int id = 20233118;
}
```

> Steps for Object construction

i. Space is allocated
ii. Field initialization: call default ctors for fields that are objects
iii. Ctors body runs

> Let's biject step 2 using **Members initialization List** (MIL)

- syntax available only in class
- can be used to initializing all/some fields
- MIL always initializes fields in declaration orders
- In some cases, using an MIL to initialize a field is more efficient than assigning it in the class body
- MIL has priority over in-class initialization

```
Student::Student(int id, int assns, int md, int final):
    id{id}, assns{assns}, md[md], final{final}{}

struct Vec{
  int x = 0; // never used
  int y = 0; // never used
  Vec(int x, int y);
};

Vec::Vec(int x, int y): x{x}, y{y}{}

struct Bla{
  Student s;
  Vec v;
};

// Copying Ctors below

Student billy{80, 50, 70}
Student bobby{billy}
```

> Construct an object as a copy of an existing object

- We got one for free
- Copies fields of existing object into the new object

> Every class comes with

- default ctor (0 parameters)
- copy ctor
- destructor (dtor)
- copy assignment operator
- move ctor
- move assignment operator

> A copy ctor takes a reference of the existing object

```
Student::Student(const Student &others):
    assns{others.assns},md{others.md},final{others.final}
```

> The free copy ctor might not always do the **correct** thing

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

Node::Node(int data, Node *Next): data{data}, next{next}{}
Node::;Node(const Node& others): data{others.data}, next{others.next}{}
Node *np = new Node{1, new Node{2, new Node{3, nullptr}}};
```

```cpp
Node n1{*np}; // calls copy ctor
Node *n2 = new Node{*np}; // copy ctor

// Above only copies the address => wrong
// Below is correct

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

> A copy tor is called

i. constructing obj. as a copy of an existing obj.
ii. passing an object by value
iii. returning an object by value
iv. The second is the reason why parameter to a copy ctor is a reference

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

// If use 'explicit'
Node n{4};
void foo(Node);
foo(n);

// else
Node n = 4;
foo(10)
// single param ctor create implicit/automatic conversions
```

Big 5 cont.

> Destructors

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

> every class comes with a dtor

- dtor calls dtors on fields that are objects

```cpp
delete np; // calls dtor on Node 1, 2, ... until nullptr

Node::~Node(){
  delete next; // deleting nullptr is safe
}
```

> Copy Assignment Operator

```cpp
Student billy{80, 50, 70};
Student bobby{billy}; // copy ctor
Student jane{};
jane = billy; // copy assignment operator
jane.operator = (billy) // operator is a method
```

> Sometimes the one we get free does not do the correct thing

```
Node n1{}, n2{}, n3{};
n2 = n2; //n2.operator = (n1);

Node &Node::operator=(const Node &others){
  if (this == &others) return *this; // 1.
  data = others.data;
  delete next;                       // 2.
  next = others.next ? new Node{*others.next} : nullptr;
  return *this; // return a reference
}

// 1. if new fails to allocate memory, next is not assigned;
      make next a dangling ptr;
      Self Assignment Check;
      Betles to delay deleting next until we know that new will not fail;

// 2. next might already pointing to heap memory;
n1 = n1; n1.operator = (n1);
//    next and others.next are the same => access a dangling ptr;

// Optimized version below
Node &Node::operator=(const Node &others){
  if (this == &others) return *this; // 1.
  Node *temp = next;
  next = others.next ? new Node{*others.next} : nullptr;
  delete next;                       // 2.
  data = others.data;
  return *this; // return a reference
}
```

> Copy & Swap Idiom

```
// node.h
struct Node{
    //
    //
    void swap(Node &);
    Node &operator=(const Node &);
}

// node.cc
#include <utility>
void Node::swap (Node &others) {
  using std::swap;
  swap(data, others.data);
  swap(next, others.next);
}

Node &Node::operator=(const Node &others) {
  Node tmp{others}; // copy ctor (deep)
  swap(tmp);
  return *this;
}
```

```

// rvalue/node.c
// 2 calls to Basic ctor
// 2 calls to copy ctor(to copy argument)
Node n2{return value};
// 2 more calls of copy ctor (construct n2 as copy of return value)
```

> Move ctor

```
// C++11 introduced rvalue reference &&
Node::Node(Node &&other):
    data{other.data}, next{other.next}{}
// A rvalue reference is a reference to a temporary(a rvalue)
// If a move ctor is available, it is called whenever we are constructing an object from a rvalue
// If we implement any of the Big 5 (copy ctor, copy operator=, dtor, move operator=), you lose move ctor

// Move assignment operator (move operator=)
Node &Node::operator=(Node &&other){
  swap(other);
  return *this;
}
```

> Rule of 5: If you need to write a custom copy ctor, or copy operator=, or move ctor, or move operator=, then you usually need write all;

> Copy/Move Elision => Under certian conditions, the compiler is allowed (not required) to avoid some copy/move ctor calls(even if these have side effect)

```
Vec v = makeVec()
// This would cause move/copy ctor to be called
// The compiler may directly construct the Vec in the space for Vec v

Void foo(Vec v){  };
foo(makeVec());
// fno-elide-constructors
```

> Operator Overloading: Methods or standalone functions

```
n1 = n2; // n1.operator=(n2);
Vec v1{1, 2}
Vec v2{3, 4}
Vec v = v1 + v2; // v1.operator+(v2);
v = v1 * 5; // v1.operator*(5);
v = 5 * v1; // 5.operator*(v1);

Vec Vec::operator+(const Vec &other) {
  return Vec{x + other.x, y + oter.y}; // implicitly this->x
}
Vec Vec::operator*(const int k) {
  return Vec{x * k, y * k};
}
Vec Vec::operator*(const int, const Vec &);
```

```
ostream & Vec::operator<<(ostream &out) {
  out << x << " " << y;
  return out;
}
v2 << (v1 << cout); // v1.operator<<(cout);
// cout << a << b; I/O operators are implemented as standalone functions
```

Arrays / const methods / invariants / encapsulation

```
struct Vec{
  int x, y;
  Vec(int, int);
}
// Want compile as default ctor gone
```

> Options:

```
// implement a default (0 param ctor)
// stack arrays: use C's array initializing syntax
Vec arr[2] = {Vec{1,2}, Vec{3,4}};
// Instead of array of objects, create array of pointers to object
Vec * arr[3];
Vec ** arr = new Vec*[3];
for (    ) {
  delete arr[i];
}
delete[] arr;
```

> Const Methods

```
struct Student{
  int assn, mt, final;
  float grade() const {
    return 0.4 * assn + 0.2 * mt + 0.4 * final;
  }
}

const Student billy{70, 65, 75};
```

> const object can only call methods that are const

> Invariants => some assumption/statement that must not be violated for program to function corectly

> We need to hide information/implementation to protect invariants

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

```

```
Node n2{3, nullptr};
// When n2 is out of scope, dtor will run automatically
// Cannot call delete below
Node n1{4, &n2};
```

> Node() assumed next is either nullptr or a ptr to heap

> Encapsulation

i. treat objects as capsules (black box)
ii. interact through provided methods (interface)
iii. public/private labels (control content below label)

```
struct Vec{
  // default public visibility
    Vec(int, int);
  private:
    int x, y;
  public:
    Vec operator+(const Vec &);
}
```

> Try make every fields in struct to be private

> Class => default visibility is private (Struct is public)

```
// Rewrite Vec above
Class Cec {
  int x, y;
  public:
    Vec(int, int);
    Vec operator+(const Vec &);
}
```

> Node Invariant => wrap Node within a List class

```
// list.h
Class List {
  struct Node; // declare Node as private
  Node *thelist = nullptr;
  public:
    void addToFront(int n);
    int ith(int i);
    ~List();
}

// list.cc
struct List::Node{
  // Nobody have access to here since struct Node is already a private struct in List class, so we do not need to set fields to be private
  int data;
  Node *next;
  Node (int, Node *);
  ~Node() {
```

```
    delete next;
  }
}

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

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

int List::ith(int i) {
  Node *curr = thelist
  for (int j = 0; j < i; ++j, curr = curr->next);
  return curr->data;
}
```

Iterator Design Pattern

> Good solution to common design problems

> Keep track of how much of teh list we have traversed

```
for (int *p = arr; p != arr + size; ++p) {
  ... *p ...
}
```

> We will create a new (Iterator) class that acts as a ptr into a list

> Iterator have access to Node

> To Do List

- A way to create a fresh Iterator to the start of list
- A way to represent on Iterator to the end of list
- Overload !=, ++(unitary), *(unitary)

```
Class List {
  Struct Node {
    ...
  };
  Node *thelist = nullptr;
  public:
  Class Iterator {
    Node *curr;
    public:
    explicit Iterator (Node *n) : curr{n} {}
    itn & operator*() {
      return curr->data;
    }
    Iterator & operator++() {
      curr = curr->next;
```

```
      return *this;
    }
    bool operator!=(const Iterator &other) {
      return curr != other.curr;
    }
  };
  Iterator begin() {
    return Iterator{thelist};
  }
  Iterator end() {
    return Iterator{nullptr};x
  }
  // prev. List Methods
};

// Client Code
itn main(void) {
  List l;
  l.addToFront(3);
  l.addToFront(2);
  l.addToFront(1);
  for (List::Iterator it = l.begin(); it != l.end(); ++it) {
    cout << *it << endl; // print 1,2,3
    *it += 1; // change to 2,3,4
  }
}
```

> C++ has built-in support/syntax for the Iterator Design Pattern

```
auto x = y; // define x to be the same type as y

for (auto i: l) {
cout << i << endl; // i is int (by value)
}
for (auto &i : l) {
i += 1; // i is int & (ref. to data fields)
}
```

Range-Based For Loops

> We can us e a range-based for loop for a class MyClass if

- MyClass has methods begin/end which return objects of a class, say Iter.
- Iter. must overload !=, *(unitary), ++(unitary prefix)
- To force clients to use begin/end, make Iterator ctor provide
- if we make it private, List::begin/List::end will lose access (private really implies no access)
- the Iterator class can declare List to be a friend

```
Class List {
  ...
  public:
  Class Iterator{
```

```
    Node *curr;
    explicit Iterator (Node *n) : curr{n} {}
    friend class List;
  };
};
```

– Friendship break encapsulation, so make as fewer friends as possible!

Finish Accessors/Mutators System Modeling

> Keep fields private and provide accessors(getters) and mutators(setters) public

> Suppose a class has private fields and no accessors/mutators - want to implement the I/O operators => as standalone functions

```
Class Vec {
  int x, y;
  ...
  friend ostream & operator<<(ostream &, const Vec &);
}
```

Object-Oriented Programming **Fall 2019**

Chapter 03 - System Modeling

Professor: Nomair *Author: Yiming Dai*

> Identify the abstraction/entities/classes

> Interaction/relationship between classes

UML(Unified Modelling Language)

Inheritance, Virtual

> A desired class inherits all members from the base class

> A desired class **cannot** access private inherited members

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

Class Comic : public Book{
  string hero;
  public:
}

Class Text : public Book{
  string topic;
  public:
}
```

```
Text::Text(string t, string a, int n, string topic) :
  title{t}, author{a}, numPages{n}, topic{topic} {}
```

> Above won't work

- inherited fields were private
- MIL can only use fields declared by the class
- No default ctor for Book

> 4 steps to object construction

- space is allocated
- **super**class part is constructed
- **sub**class fields constructed
- ctor body runs

```
Text::Text(string t, string a, int n, string topic) :
    Book{t, a, n}, topic{topic} {}
```

> **protected** => if a member is "protected" it is accessible by the class and its subclasses

```
Class Book{
  protected:
    string title, author;
    int numPages;
  public:
    Book(string, string int);
}

Class Comic : public Book{
  string hero;
  public:
}

Class Text : public Book{
  string topic;
  public:
}

void Text::addAuthor(string a) {
  author += a; // OK as author is protected
}

int main() {
  Text t = ...
  t.author = X // no access!
}
```

> Advice

- Keep fields private
- provide protected accessors/mutators

Method Overriding

```
Book b = Comic{  ,  , 40, "batman"}; // Cause slicing
b.isHeavy(); // Call Book's copy ctor
// Book::isHeavy() => false // Call this!
// Comic::isHeavy() =>true  // Not call this!
```

> Using superclass pbs, to point to subclass objects will not cause slicing

> By default, compiler looks at the declared type of the ptr. to choose which method to run (static dispatch)

> We typically want the method to be chosen based on the type of **object** => use **virtual** keyword

> Virtual Method => the choice of method is based on the runtime type of the object (Dynamic Dispatch) but more costly than static dispatch

> Polymorphism => the ability to accommodate multiple types within one abstraction

Destructor

> 4 steps to destroying objects

- subclass dtor runs
- dtor for subclass fields that are objects
- superclass dtor runs
- space is reclaimed

Virtual

> Superclass do not know the existence of subclasses, so

i. If a class might have subclasses, make the dtor virtual in the base class
ii. If a class will never have a subclass, declare it as final

```
class X {
  virtual ~X(){...}
}
class Y : public X {
  ~Y(){...}
}
class Z final : public X {

}
```

> *Student::fees()* has no implementation since

i. We don't want to implemented it
ii. make it **Pure Virtual (P.V.)**

```
class Student{
  public:
    virtual int fees() = 0; // Pure Virtual Method => No implementation
}
```

> **Virtual**: subclasses **may** override behaviour

> **P.V.**: subclasses must implement mrthod to be **CONCRETE**

> *Student* has a P.V. method

i. it is incomplete
ii. it is an **ABSTRACT** class

> A class is abstract if

i. it declares a P.V. method
ii. it inherits P.V. methods that it does not implement

> A class is concrete if it is not abstract

- Cannot create objects of an abstract class
- Abstract Classes
  1. organize subclasses
  2. members (fields/methods) that are common across all subclasses can be placed in the base class
  3. act as an interface
  4. Polymorphism
- Virtual/P.V. - italics
- abstract class - class name in italics

C++ — Template

- C++ template class is a class parameterized on type

```
template <typename T>
class Stack{
  int len;
  int cap;
  T *contents;
  public:
    void push(T);
    void pop();
    T top();
    ~Stack();
};
Stack<int> s1;
s1.push(1);
Stack<string> s2;
s2.push("hello");

// Template list class
template <typename T>
class List{
  struct Node{
    T data;
    Node *next;
  };
  Node *thelist = nullptr;
  public:
    class Iterators{
      Node *curr;
      explicit Iterator(...);
      public:
        T &operator*() ...
        bool operator=(...)
        Iterator operator++() ...
        friend class List<T>;
    };
    T &ith(int i);
    void addToFront(T &t);
```

```
    ~List() ...
};
List<int> l1;
l1.addToFront(1);
List<List<int>> l2;
l2.addToFront(l1);
```

- Standard Template Library (STL) (std::vector - #include <vector>)
  - template class
  - dynamic length arrays
    - heap allocated
    - automatically resize as needed

```
vector<int> v{3, 4};
v.emplace_back(5);
v.pop_back();
for (int i = 0; i < v.size(); ++i) {
  cout << v[i] << endl;
}
for (vector<int>::iterator it : v.begin(); it != v.end(); ++it) {
  cout << *it <<endl;
}
for (auto n : v) {
  cout << n << endl;
}
for (vector<int>::reverse_iterator it = v.rbegin(); it != v.rend(); ++it) {
  cout << *it << endl;
}
v.erase(v.begin());
v.erase(v.end() - 1);

v[i] // unchecked access
v.at(i) // checked access
// if i is not in range, out_of_range exception is thrown
// After an exception is caught, program continues after the catch slack
```

- Stack unwinding => when an exception occurs, the call stack is repeatedly popped until an appropriate catch slack is found

Exception

- Exception recoveries can be done in stages

```
try{...}
catch (Some Exc e) {
  // do part recovery
  throw otherExcep{...}
}
```

  i. throw some other exception

ii. throw e; => throw the caught exception but might have sliced the original exception
iii. throw; throws original exception

> All C++ library exceptions inherit from **std::exception**

> C++ does not require exceptions to inherit from std::exception

> In C++ you can throw anything

> Good Pratice

i. throw objects of existing exceptions or create your own exception classes
ii. catch by reference $out_of_range|bad_alloc|length_error$

Observer Pattern

> Publish/subscribe model

> Publish - Subject generates data

> Subscribe - Observe wants to be notified of new data

Design Pattern

> Design Philosophy

i. Use abstract base classes to provide interface
ii. USe ptrs to base class that call these interface methods
iii. behaviors changes based on the runtime type of objects

Iterator Pattern

> operator*, operator++, operator !=

```
class AbsIter{
  public:
    virtual int &operator*() const = 0;
    virtual AbsIter &operator++() = 0;
    virtual bool operator!=(const AbsIter &) const = 0;
    virtual ~AbsIter(){}
};

class List{
  public:
  class Iterator : public AbsIter{
    // List::Iterator IS A AbsIter
  };
};

class Set{
  public:
    class Iterator : public AbsIter {
```

```

    }; // Set::Iterator IS A AbsIter
}
```

> We can now implement code that operates using AbsIter and not being tied to a specific data structure

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

List e ...
List::Iterator i = e.begin();
foreach(i, e.end(), addTen(i));
```

Factory Pattern

> The factory method pattern is called the virtual constructor pattern

```
class Level{
  public:
  virtual Enemy *createEnemy() = 0;
  };
class Normal : public Level {
  public:
  Enemy *createEnemy() override { //more turtles }
  };
class Castle : public Level {
  Enemy *createEnemy() override { // more bullets }
  };

Player *p = ...
Level *l = ...
Enemy *e = ...
while (p->notDead()) {
  // generate enemy should depend on level
  e=l->createEnemy(); // factory of enemy
}
```

Template Method Pattern

> Base class implements the template/skeleton and the subclass fills the blanks

> Base class allows overriding some virtual method but other non-virtual methods must remain unchanged

> Some methods in base class are virtual and some are non-virtual

```
class Turtle{
  public:
    void draw() {
```

```
      drawHead();
      drawShell();
      drawFeet();
    }
  private:
    void drawFeet(){}
    void drawHead(){}
    virtual void drawShell() = 0;
};

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

> Non-virtual Interface(NVI) idiom

Public Virtual Method

> Public(interface) - provide a contract

> Virtual - invitation to subclasses to change behaviour

> Contradictory!

> In a NUI

i. All public methods are non-virtual (expect dtor)

ii. All virtual methods are private/protected

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

Visitor Design Pattern

> In C++, dynamic dispatch does not account for runtime type of parameters

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

```cpp
  public:
     void strike (Weapon &w) override {w.useOn(*this);} // Compile time this becomes Bullet
};
class Weapon {
  virtual void useOn (Turtle &) = 0;
  virtual void useOn (Bullet &) = 0;
};
```

> Double Dispatch => using a combination of overriding and overloading

Book Hierarchy

> want to add functionality that denepnds on runtiem type of objects

> What is we do not have the source code?

> What if the behaviour does not really nelong to the classes?

> We can do this if the Book hierarchy will **accept** BookVisitors

```cpp
class Book {
  public:
    virtual void accept(BookVisitors &v) {
      v.visit(*this); // *this must be a Book
    }
};
class Text : public Book {
  public:
  void accept(BookVisitors &v) override {
    v.visit(*this); // *this must be a Text
  }
};
class Comic : public Book {
  public:
  void accept(BookVisitors &v) override {
    v.visit(*this); // *this must be a Comic
  }
};
class BookVisitors {
  public:
    virtual void visit(Book &) = 0;
    virtual void visit(Text &) = 0;
    virtual void visit(Comic &) = 0;
    ~BookVisitors();
};
```

STL std::map

> Template class, generalized arrays

> parameterized on 2 types key/value

> key type must support operator<

```
std::map<string, int> m;
m["abc"] = 5;
m.erase("abc);
m.count("abc");
for (auto p : m) { // p => std::pair<string, int>
    cout << p.first << p.second;
} // iteration is sorted key orders
```

> Cataloging Book

i. author -> # of Book

ii. topic -> # of Text

iii. hero -> # of Comic

iv. string -> int

```
class Catalog : public BookVisitors {
  map<string, int> cat;
  public:
    void visirt(Book &b) override {++cat[b.getAuthor();]}
    void visirt(Comic &c) override {++cat[c.getHero();]}
}
```

Compilation Dependencies

> Fix cycle of includes by forward decreasing classes whenever possible

> an include creates a dependency

> best to **reduce** dependencies

i. avoid cycles

ii. fewer recompilation

iii. faster compile time

> Usage

- include a file at Inheritance

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

- include a file when constructing another class since need to know its size

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

- Forward declaration with pointer but include a file in .cc file

```
// .h
class A;
class D {
  A *pA;
};

// .cc
#include "a.h"
pa->method();
```

– Forward declaration with constructing methods but include a file in .cc file

```
// .h
class A;
class E {
  A foo(A a);
};

// .cc
#include "a.h"
A E::foo(A a){}
```

Reduce Dependencies

```
// File "window.h:"
#include <xlib/xlib.h>
class XWindow {
  Display *d;
  Window w;
  GC gc;
  public:
    draw();
}
```