CS 246

A **shell** is an interface to a computer.

A graphical shell usually includes mouse inputs and fancy graphical interfaces.

A command shell is one where commands are entered to execute programs.

Linux shell

Linux files are either directories or regular files. Windows only has folders and files.

Backticks execute commands within double quotes (not single)

Eg: `echo "Today is \date“‘“`

Eg: `echo "Today is $(date)"`

`egrep` is the extended version of `grep.` Equivalent to `grep -e`

`[abc]` match a single character within box

`[^abc]` not match any of the characters

`?` matches 0 or 1 of preceding expression. Optional argument

`*` is 0 or more of preceding

`+` is one or more of preceding

`.` is any one character

`^` is start of line

`$` is end of line

[0-9], [a-z] etc

Special symbols lose meaning in square brackets. No need to escape

chmod

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

Example:

Only read write permission for everyone

```
chmod a=rw file
```

Remove execute permission from everyone

```
chmod a-x file
```

Variables

`x=1 echo ${x}`

Double quotes allow expansion variables while single quotes do not

Shell script

A script is a text file that contains a sequence of Linux commands, executed as a program.

Start with `#!/bin/bash`

Execute with `./file`

Don't forget to chmod execute permissions.

Arguments to a script

Access arguments within a script with `$1,$2,$3` etc.

If an argument is not provided, it will be the empty string

```
#!/bin/bash
egrep "^${1}" /usr/share/dict/words > /dev/null # This throws away the output
usage(){
    echo "Usage $0 password"
    exit 1
}
if [ ${#} -ne 1 ]; then
    usage
fi
if [ $? -eq 0 ]; then
    echo 'yes'
else
    echo 'no'
fi
```

In Linux 0 represents success, anything else represents failure.

Status code can be obtained by `$?`

The square command is used for comparison.

`[ 1 -eq 1 ]` This doesn't provide any output `echo $?` Must use status code to determine truth value.

`$#` gives the number of arguments provided to the script.

`$0` is the name of a script.

`$@` lists all the arguments

`exit 1` terminates the program and returns a status code of 1 (failure).

`lt` is less than. `le` is less than equal.

By default, if no status code is returned, it is assumed to be 0 (success).

Print numbers from 1 to `$1`

```
#!/bin/bash
x=1

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

Mathematical operations are done with `$((1+1))`, operation surrounded by two brackets. Else it will just concatenate two strings.

For loops

Rename all .c files to .cc.

```
filename=hello.C
mv hello.c ${filename%C}.cc
```

Another way is

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

`{filename#abc}`

Example: How many times does the word $1 appear in the file $2.

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

To get the last Friday of the month, where `$1` and `$2` represent month and year.

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

C++

C++ was originally called C with classes.

Hello World!!!

```
# include <iostream>
using namespace std; // Don't need to write std::cout
int main() {
    cout << "Hello World" << endl;
    return 0;
}
```

```
g++ hello.cc -o executable.out
```

C++ has three stream variables.

`cout` to output to standard output `cin` to get input from standard input `cerr` to output to standard error

Input Operator: `>>`

Output Operator: `<<`

```
int x,y;
cin >> x >> y; //Reads in two variables
```

Whitespace is ignored when reading.

sends an EOF signal, the read fails, program continues.

Once a read fails (bad input), all subsequent reads fail

If a read fails, `cin.fail()` is true.

If a read fails due to EOF, `cin.fail()` and `cin.eof()` are both true

If you add an int to a double, int is converted to double

There is an implicit conversion from `cin` to `void *`

`cin` is automatically true if `!cin.fail()`

`a >> b` is a binary shift operation

Example: `21 >> 3` is `10101` right shifted 3 times, resulting in `10`

Operating overloading: `21 >> 3` and `cin >> a`

Same operator has different meaning depending on the types of the operands

The input operator evaluates to an istream (cin)

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

Cascading

`cout << i << endl;`

`cout << endl;`

`cout;`

Dealing with bad input

`cin.clean()` acknowledges that a read failed. It lowers the fail flag.

`cin.ignore()` discards whatever is at the front of the input stream. Discards everything until it hits whitespace

`cin.ignore()` discards only one character actually. TODO!

Strings

```
#include <String>

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

`getline(cin,s)` will start reading from first non whitespace character until newline character

IO Manipulators are used to format output

`cout << hex << x` changes all subsequent couts to hexadecimal. To revert back to decimal, simply use `cout << dec`

`cout << boolalpha` changes boolean numbers to true and false

The stream abstraction can be used with other data sources. To read/write to files, include `<fstream>`, which is split into `ifstream` input file stream and `ofstream` output file stream

```
# include <iostream>
# include <fstream>

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

`file` is a stack-allocated variable. Memory is reclaimed when stack is popped.

We can treat a string as a stream. `# include <ssstream>`

istringstream to read from a string and ostringstream to write to a string.

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

Note: To read integer input, first read as string, and using istringstream as input and read into integer. It avoids cleaning and ignoring input resulting in much cleaner code.

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

Beautiful code XD

Strings

In C, strings are an array of characters with a null terminator. C++ has a string type. Automatic resizing is much safer because no change of overwriting null terminator. `string str = "hello"` is represented as an array of characters.

Conversions from C style strings to C++ strings.

C: `strcmp`, C++: `s1 == s2`, `s1 != s2`, `>`, `<`

C: `strlen`, C++: `str.length()`

C: `strcat(s1,s2)`, C++: `s1+s2`

In C++, can use `s[1]` to access characters in a string.

Default Arguments

Aside:

```
void print(string filename) {
    string s;
    ifstream file(filename); // Assumes filename is a C style string.
    while (file >> s) {
        ... // Doesn't work
    }
}
```

Working version

```
void print(string filename) {
    string s;
    ifstream file(filename.c_str()); // Works!
    while (file >> s) {
        ...
    }
}
```

A parameter that has a default value must be followed only by parameters which have default values.

In C, there is no function overloading. Valid in C++ though. (must still be different number of or type of arguments)

Two functions cannot just differ on their return type

Cannot do `void test()` and `void test(int i=0)` for method overloading.

Operator Overloading

Operators are implemented as functions.

`-21 >> 3` is equivalent to `operator>>(21,3)`

Declaration Before Use

Must declare something before using.

Problem: Mutual recursion.

Solution: Forward declaration. Declare first, then define later. In declaration, don't need to declare variable name. Just type is enough.

Array

Built in construct to store collections. Located in a continuous block of memory.

The name of the array is the same as the address of the first element of the array.
`a == &a[0]`

Structures

C:

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

struct Node n1 = {3,0};
```

C++

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

Node n2 = {3,0};
```

Constants

NOTE: A constant definition must be initialized.

```
int n = 5;
const int * p = &n;
p = &m; //valid
*p = 10; // INVALID, the integer is a constant
n = 19; // Valid
```

Pass by Reference

`cin >> x`

In C++, `x` is pass by reference, which is why it can be modified by the operator function.

```
int y = 10;
int &z = y; // z is a reference to y (constant pointer)

// Similar to
int * const z = &y;
```

A reference acts as a constant pointer with automatic dereferencing.

z is both another name and an alias for y.

`int * p = & z` creates a pointer to y.

A reference must be initialized with a value that has an address.

```
int &x = 5; // Invalid
int &x = y + y; // Invalid
```

An lvalue is anything that is a storage location

Other fun stuff

- Cannot create pointer or reference to another reference.
- Cannot create array of references
- Can create a reference to a pointer
- Can use references as parameters!

```
void inc(int &n) { // Changes to pass by reference
    n+=1;
}

int main() {
    int x = 5;
    inc(x);
    cout << x; // Incremented because of pass by reference instead of value
}
```

To prevent a function from changing a variable, we can use a const reference. Cannot use the reference to change the original variable. :D

```
void bar (const ReallyBig & rb) {
    //...
}

void f(int &n) {}
void g(const int *n) {}
f(5) // Illegal
f(y+y) // Illegal

g(5) // Legal
g(y+y) // Yes
```

This is because g does not have permission to change n, so compiler will allow you to pass a reference to an unknown.

Try to always consider passing by cost reference for anything larger than an int (4 bytes).

Dynamic Memory

Always use the stack unless

- Value must persist beyond variable scope
- Size of collection is unknown or subject to resizing
- Large memory allocation

In C++, we use new and delete for memory allocation. (Type aware, safer to use)

```
Node n; // Stack
Node * p = new Node; // Heap

delete p; // p must be a heap allocated pointer
```

No need to tell `new` how much space is needed

Calling delete on null is fine.

Arrays

```
int * p = new int[num];

delete [] p; // Don't forget the [] to delete the array
```

A **dangling pointer** is a pointer to memory that is not in use.

Operator Overloading

In C, when we wanted to add structs, we had to define our own function to do so. For example `Vec v3 = add(Vec v1, Vec v2)`

What if we want to do `Vec v3 = v1 + v2`? The `+` operator may be overloaded.

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

Need const for cascading. The return value is not an lvalue, but it will still work if const is specified.

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

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

Example:

```
struct Grade {
    string name;
    int grade;
};
```

What if we want to do `cout << grade`?

```
ostream & operator<<(ostream &out, const Grade &g) {
    out << "Student: " << g.name << endl;
    out << "Grade: " << g.grae << "%";
    return out;
}
```

Another Example:

```
istream & operator>>(istream &in, Grade &g) {
    in >> g.name;
    in >> g.grade;
    if (g.grade > 100) g.grade = 100;
    return in;
}
```

Preprocessor

`# include` is a preprocessor directive. Copies the actual file.

`g++ -E file.cc` stop at preprocessor output

`g++ -E -P file.cc` stop at preprocessor output and omits line numbers

`# define VAR VALUE` creates a search and replace for VAR and replaces with VALUE. (Previously used for constants).

Preprocessor does not need to know C++

Conditional Computation

Native windows applications needed `int winMain()` instead of `int main()`.

Can use a preprocessor if statement to check and follow operating system guidelines

```
#define Unix 1
#define Windows 2
```

```
#define OS Unix

#if OS == Unix
int main(){
#elif OS == Windows
int winMain() {
#endif
    //code
}
```

Must manually change defines. Alternative: Use `g++` with arguments

`g++ -DOS=Unix`

In this case, don't need to define OS anymore. Everything else remains the same.

`#ifdef VAR` is true if var is defined

`#ifndef VAR` is true if var is not defined

Don't forget `#endif`

```
int main(){
#ifdef DEBUG
    cout << some_debug_var << endl;
#endif
    // other stuff
}
```

Separate Compilation

Files are split into two categories: Interface files, implementation file

Compiling multiple files

`g++ file.cc` (requires that all cc file are part of the program)

NOTE: Never compile a header file.

Separate compilation involves compiling individual cc files separating and then merging them to create the executable.

`g++` compiles, links, and produces an executable. Cannot compile individual files because linker will fail

To just compile, use `g++ -c` to avoid linker problems. This compiles into a `.o` file

Object file contains compiled code, what implementations are needed and defined.

To link, `g++ main.o vector.o`

Dealing with Global Variables

Define the global variable in a header fie. Define it in a .cc file. Or else there will be two of the same variable and compiler will cause error.

`extern int global`

Avoiding multiple definitions of types (multiple include statements on the same file):

```
# ifndef HEAD_H // common convention is FILENAME_H
# define HEAD_H

struct blah {
    // definition
}
# endif
```

This is known as an include guard.

NOTE: Never `# include` .cc files

NOTE: Do not use `using namespace std` in header files

C++ Classes

Classes are structures with functions.

Instance of a class is an object.

A function inside a class is called a **member function** or **method**.

Methods have access to the fields of the parent object.

Method has a hidden parameter called `this` which refers to a pointer to the parent class.

Initializing Object

Constructors

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

```
    // Constructor (no return type)
    Student (int assns, int mt, int final) {
        this.assns = assns;
        this.mt = mt;
        this.final = final;
    }
}

Student person(60,70,80); //stack
Student otherguy = Student(60,70,80); //stack
Student jared = new Student(60,70,80); // heap
```

Constructors can be overloaded and initialized with default values.

`Student newguy()`. Compiler gives warning because it might be a function definition.

Use `Student newguy = Student()` instead to specify 0 parameters.

Every class comes with a default constructor that initializes all fields of the class.

`Student newguy` uses the default constructor.

NOTE: Primitive types and pointers are not objects.

Default constructor goes away if any constructors are written. C style initializations are also lost.

Cannot use C style field initializers in C++.

NOTE: constants and references must be initialized before the body runs.

Steps for creating object

- Space allocated
- Field initialization, constructors for any fields that are objects are called.
- Constructor body runs

Member Initialization List

- Only available in constructors

```
struct myStruct {
    const int myConst;
    int & myRef;
    myStruct (int c, int & r) : myConst(c), myref(r) { // runs in conjunction with step 2 f
        // blah
    }
}
```

Using a MIL is the only way to initialize constants and references.

If MIL does not initialize a field that is an object, its default constructor is still automatically called.

Fields are initialized in declaration order irrespective of the order they occur in the MIL.

Copy Constructor

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

A copy constructor takes one parameter: a reference to const of the type of the class. Does a field for field copy.

Class Properties

- Default constructor
- Copy constructor
- Destructor
- Copy assignment operator

```
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) {}
}
```

Linked List Example

```
Node * np = new Node(1, new Node(2, new Node(3, NULL)));
Node m = * np; // Only copies first element
Node * npCopy = new Node(*np);
```

This does shallow copy. How about deep copy?

```
struct Node {
    Node(const Node & other) { // Copy constructor
        data = other.data;
        next(other.next ? new Node(* other.next) : NULL);
    }
}
```

Use cases for Copy Constructor

- Creating a copy object
- Object is passed by value (this is why parameters are passed by reference)
- Returning object by value

Single parameter constructors

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

Node n(4);
Node m = 4;

struct Node {
    int data;
    Node * next;
    explicit Node(int data): data(data), next(Node) {} // Disables implicit conversion
};
```

Single parameter constructors create an implicit conversion.

Destructors

Run when object is destroyed.

Stack: Object goes out of scope.

Heap Allocated: Explicitly deleted

Also calls the destructors on any field of the object.

A class only has **ONE** destructor. No parameters. No return type. Has name of class prefix with ~.

```
Node * np = new Node(1, new Node(2, new Node(3, NULL)));
delete np; // Only deletes first node

struct Node {
    ~Node() {
        delete next; // If next is null, delete on null is fine
    }
}
```

```
// header file
~Node()

// cc file
Node::~Node(){
    // blah.
    // Satisfies both belonging to a class and telling header file that function is declare
}
```

Copy Assignment Operator

NOTE: This is a method

```
Student bobby = billy; // Copy constructor
Student jane; // zero parameter constructor
jane = billy; // Copy assignment operator
```

When dynamic memory is involved, we want a deep assignment.

Default Copy Assignment

```
Node & Node::operator=(const Node & other) {
    data = other.data;
    next = other.next;
    return * this;
}
```

Deep Assignment

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

Wrong implementation. `this->next` could have been pointing to dynamic memory. Should delete it first to avoid leaking memory.

```
Node & Node::operator=(const Node & other) {
    data = other.data;
    delete next;
    next(other.next ? new Node(*other.next) : NULL)); // If other.next and this.next are po
    return * this;
}
```

Self Assignment

```
Node * np = // blah
Node *p = np;
* np = * p; // Self Assignment
```

Deep assignment must guard against self assignment.

```
Node & Node::operator=(const Node & other) {
    if (this == &other) return * this; // Guard
    data = other.data;
    delete next;
    next(other.next ? new Node(*other.next) : NULL)); // If new Node fails, next becomes a
    return * this;
}
```

Exception Safe Version of Assignment Operator

```
Node & Node::operator=(const Node & other) {
    if (this == &other) return * this;
    data = other.data;
    Node * temp = next;
    next(other.next ? new Node(*other.next) : NULL));
    delete temp; // Next will not become dangling because if previous line fails to execute
    return * this;
}
```

Copy and Swap Idiom

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

Node & operator=(const Node & other) {
    Node temp = other; // Copy constructor. Don't need to delete because stack allocated
```

```
    swap(temp);
    return * this;
}
```

Rule of 3

If you need to write a copy constructor, destructor, or operator=, you usually need to write all three.

`operator=` is always implemented as a method.

When an operator is implemented as a method, the this pointer represents the left hand side.

Can also implement `operator+` as a method.

```
Vec & Vec::operator+(const Vec & other) {
    Vec v(x + other.x, y + other.y);
    return v;
}

Vec & Vec::operator*(int k) {
    Vec v(x * k, y * k);
    return v;
}
```

`Vec v5 = 5 * v3`. Cannot do `5.operator*(v3)` because `5` is not an object. Must implement as a function in this case.

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

But can only call this with `Vec << cout`. Do not implement input output operators as methods. Always use functions

Operators that must be implemented as methods: - `operator=` - `operator[]` - `operator->` - `operator()` - `operatorT()`

Arrays of Objects

```
struct Vec {
    int x,y;
    Vec(int x, int y): x(x), y(y) {}
}
Vec vectors[3];
Vec * myVecs = new Vec[10]; // Won't compile because objects within array must be initializ
```

Option 1: Provide a 0 parameter constructor

Option 2: Stack allocated, use array initialization.

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

Option 3: Heap allocated. Create an array of pointers to objects instead of the objects.

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

const Methods

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

const Student bully(60,70,80);
billy.grade(); // Must use const method in order for the function call to work (even though
```

A const method does not change the fields of an object. Can only call const methods on const objects.

Mutable Fields

A **mutable field** can be changed even for const objects. A const methods can still change mutable fields.

Static Keyword

- Static fields associates with the class and not individual objects.

```
struct Student {
    static int numObjects; // Record number of student objects
    Student(...) {
        ++numObjects;
    }
}
```

Put the definition of a static field external to the type definition. Typically in the .cc file.

```
int Student::numObjects = 0
cout << Student:numObjects << endl;
```

Static Member Functions

- Do not need an object to call a static member function. (Eg: Just use `Student::printObjectsCreated()`)
- Do not have the `this` parameter
- Can only call other static member functions and access static fields

```
static void Student::printObjectsCreated() {
    cout << numObjects << endl;
}
```

NOTE: Can still use object to call a static member function, but bad practice.

Design Patterns

Singleton Pattern

Class C: Only want one object of C to be created ever.

- Logging
- Database Connection

Aside: In `<cstdlib>`, function `atexit`

- Takes a pointer to a function with no parameters and void return type.
- Will call the function passed in at exit.
- Can register multiple functions LIFO

The singleton pattern for Wallet breaks since constructor is available to everyone.

Encapsulation: Hide implementation

Let outsider access functionality through an exposed interface.

```
struct Vec {
    Vec(int x, int y);
    private: // Outsiders can no longer access these variables without using the interface
    int x,y;
    public:
    Vec operator+(const Vec & v) {
        Vec v(x + v1.x, y + v1.y); // Still works because still within class (not outsider)
        return v;
    }
}
```

NOTE: Always keep fields private.

Default visibility for struct is public.

Default visibility for class is private.

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

Fields should be private:

- Maintain class invariants.(Example: `x+y` are positive)
- Flexibility to change implementation
- If needed, we can provide controlled access to read/write fields

Scenario: `Vec` class:

- Fields private
- Don't want provide accessors

How do we implement the output operator? - Solution is the use the friend keyword + This provides real write access - However, friendship breaks encapsulation and class invariance

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

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

System Modelling

- Figure out the abstraction/entities
- Determine the relationship between classes

UML: Unified Modelling Language

- Class Name
- Fields
- Methods

```
----------------
Vec
----------------
- x: Integer
- y: Integer
----------------
+ getX() Integer
----------------
```

Relationships between Classes

Composition: Embedding an object inside another object.

```
struct Grid {
    Vec v1, v2;
    Grid(): v1(1,2), v2(0,0) {}
}
```

Replaced the default initialization with calls to the 2 parameter constructor for `Vec`.

Embedding an object inside another creates an "owns a" relationship

Example: Grid owns Vec v1 and v2 (Vec does not exist on its own)

To illustrate this relationship in UML, draw a diamond (shaded in) on the Grid class and point it to Vec.

Aggregation: A "has a" B: B has an existence on its own. Copying A does not copy B (shallow copy). Deleting A does not delete B.

Use pointer instead of structure declaration for this.

To illustrate this relationship in UML, draw a diamond (not shaded) on the Grid class and point it to Vec.

```
class Pond {
    Duck * ducks[MAXDUCKS];
}

class Catalog {
    Part * parts;
}
```

Inheritance

To illustrate inheritance, instead of drawing a diamond, just draw an arrow.

Classes that inherit properties are called Subclass/derived class/child class

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

class Textbook() : public Book {
    // stuff unique to textbook here
}

Textbook tb;
tb.getAuthor(); // Valid because of inheritance
tb.author = "Normair"; // NOT VALID: Inherits fields too, but it is private.
```

When an object is created:

- Space is allocated
- Superclass part of the object is created
- Field initialization/MIL
- Constructor Body

When an object is destroyed

- Destructor body runs
- Destructor for fields run (Reverse declaration order)
- Superclass object is destroyed
- Space is deallocated

Preempt the call to the 0 parameter constructor in step 2 of the object creation by yourself calling the 3 parameter book constructor.

```
TextBook::Textbook(string t, string a, int n, string topic) : Book(t,a,n), topic(topic) {};
```

If you want direct access to fields of the base class, use protected in the base class.

Protected means private except all subclasses will have access to the method. In UML, use `#` to indicate protected.

```
class Book {
    protected:
    string title, author;
    int numPages;
    public:
    // ..
}

class Textbook : public Book {
    String topic;
    public: void addAuthor(string auth) {
        author += auth; // Can access protected fields
    }
}
```

NOTE: Keep fields private and provide protected accessors and mutators

Method Overriding

Assume Book already has a method `isItHeavy()`. This method can be overridden in subclasses.

```
class Textbook : public Book {
    // ..
    public:
    bool isItHeavy() {
        return getNumPages() > 300;
    }
}
```

Object Slicing/Coercion

```
Book b2 = Comic();
cout << b2.isItHeavy() << endl; // Runs Book's isitheavy method
```

Placing a subclass object into a superclass variable can cause object slicing.

```
Comic = Comic();
Comic * cp = &c;
Book * bp = & c;
cout << cp->isItHeavy() << endl; // Comic's isItHeavy called
cout << bp->isItHeavy() << endl; // Book's isItHeavy called
```

Dynamic Dispatch

Compiler looks at the declared type of a pointer to decide (at compile time) which method will be called.

To change C++'s default behaviour, prefix the method with the `virtual` keyword. (In the superclass `Book`)

Compiler will delay decision to runtime

At runtime, the type of the object being pointed to will be determined to decide which method to call. (Runtime type information)

This has a slight performance penalty

Array of Books

```
Book * collection[100];

for (int i = 0; i < 100; i++) {
    cout << collection[i]->getTitle() << collection[i]->isItHeavy() << endl; // Calling eac
}
```

When you call a subclass destructor, the superclass destructor is called automatically.

NOTE: In an inheritance hierarchy, make all destructors virtual

Pure Virtual/Abstract/Concrete

Concrete class: Contains no pure virtual methods. Can only instantiate concrete classes

Abstract class: Contains pure virtual methods

Cannot create instance of abstract class

```
class Student {
    public:
    virtual int fees() = 0; // Pure virtual method
}
```

Fees is a pure virtual method: Method that has no implementation

Benefits:

- Organize types (place common fields in abstract classes)
- Polymorphism

Subclasses inherit all pure virtual methods. Must implement all inherited virtual methods to avoid being abstract.

In UML, abstract is also indicated with italics.

Static methods are underlined

Observer Pattern

- Subject/Publisher - generates data
- Observer/Subscriber - waiting for this data

Steps: (C represents concrete)

- `CObserver` calls attach on `CSubject.`
- Data us generated by `CSubject`
- Calls its `notifyObservers` -> `CObserver::notify`
- `CObserver` calls `Csubject::getState`

If you need a class to be abstract and have no pure virtual methods, make a pure virtual destructor.

Pure virtual methods are methods which subclasses must implement to be concrete.

Decorator Pattern

- Fill in

Factory Method Pattern

Game has enemies and levels.

Enemies: Turtles & Bullets

Levels: Normal, Castle

```
Enemy *e;
Level *l;
Player 8p;

while (p->notDead()) {
    // create enemy
    e = l.createEnemy();
    // Attack player
}
// rip player

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

};

class Normal: public Level {
    public:
    Enemy * createEnemy() {
        // turtles
    }
}

class Castle: public Level {
    public:
    Enemy * createEnemy() {
```

```
        if (somelogic) {
            return Boss::getInstance();
        } else {
            // more bullets less turtles
        }
    }
}
```

G++ Flags

- `Wall`, `Weverything`, `pedantic`, `Werror`, `MMD`

Template Method Pattern

```
class Sort {
    input();
    output();
    virtual doSort() = 0; // Subclasses implement patterns that are different, main class ha
    public:
    sort();
};

class MergeSort : public Sort {
    doSort();
};

void Sort::sort() {
    input();
    doSorT();
    output();
}

Sort * s = new MergeSort();
s->sort();
```

C++ Templates

```
class Node {
    int data;
    Node * next;
    public:
    Node();
    // getters, setters, remove, add
};
```

Generalizing a class by parameterizing on one or more types

```
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;
    }

    setNext(Node<T> * _next) {
        next = _next;
    }
};

Node<int> * intList = new Node<int>(2, new Node<int> (3, NULL));

Node<char> * charList = new Node<char>('a', new Node<char> ('b', NULL));

// Specializing a template

Node<Node<int> > * listOfLists;
```

Standard Template Library (STL)

Dynamic Length Arrays

Vector template parameterized on one type

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

int main() {
    Vector<int> v;
    v.push_back(1);
    for (int i = 0; i < v.size(); i++) {
        cout << v[i] << endl;
    }
}
```

Iterators

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

```
}

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

Iterator is an abstraction of a pointer

```
v.erase(v.begin());
v.erase(v.rbegin());
```

Map

```
std::map // parameterized on two types

# include <map>
int main() {
    map<string, int> myMap;
    myMap["abc"] = 24;
    myMap["abc"] = 42; // overwrite
    cout << myMap["abc"];
    myMap.erase("abc");
    if (myMap.count("abc")) { // found or not found
        // blah
    }
    for (map<string,int> iterator it = myMap.begin(); it != myMap.end(); ++it) {
        cout << (*it).first << endl;
        cout << (*it).second << endl;
    }
}
```

Visitor Design Pattern

- Double dispatch: Choosing which method to call based on runtime type information of two pointers.

```
Player * p = // ..
while (p->notDead()) {
    Enemy * e = // ...
    Weapon * w = p->chooseWeapon();
    // strike enemy with weapon
    e->strike(&w); // Can either dynamically dispatch on enemy or weapon. Unable to do both
}
```

```
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);
    }
};

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

class Stick : public Weapon {
    public:
    void useOn(Turtle &t) {
        // use stick on turtle
    }

    void useOn(Bullet & b) {
        // use stick on bullet
    }
};

class Rock : public Weapon {
    public:
    void useOn(Turtle &t) {
        // use rock on turtle
    }

    void useOn(Bullet & b) {
        // use rock on bullet
    }
};
```

Book Example

```
map<string, int> catalog;
Book * collection[100];
for (blah) {
    collections[i]->update();
}

// catalog should be updated now
// Bad design, need access to source code, clutters code (separations of concerns)
```

Prepping Book hierarchy to accept visitors

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

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

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

class BookVisitor() {
    public:
    virtual void visit(Book & b) = 0;
    virtual void visit(TextBook & b) = 0;
    virtual void visit(Comic & b) = 0;
};

class Catalog() : public BookVisitor() {
    std::map<std::string, int> theCatalogue;
    virtual void visit(Book & b) {
        theCatalogue[b.getAuthor()]++;
```

```
    }
}
```

Use forward declaration to deal with circular include

When to Forward Declare

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

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

}

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

//d.h
class A;

class D {
    A * myA;
}

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

STL

`std::accumulate` behaves the same way as foldl.

Pointers to Implementation

Changes to private members of the header file require recompilation of the client code.

window.h

```
class XWindow {
    Display * d;
    Window w;
    GC gc;
}
```

windowImpl.h

```
struct XWindow {
    Display * d;
    Window w;
    GC gc;
}
```

window.h

```
class Window {
    XWindow * pImpl;
    public:
        drawRectangle();
        // ..
}
```

window.cc

```
# include "window.h"
# include "windowImpl.h"

XWindow::drawRectangle() {
    pImpl->d->update();
    // blah
}
```

This is called the `pImpl` idiom.

The `pImpl` idiom generalized to accommodate multiple implementations is called the bridge pattern.

Coupling & Cohesion

- Low coupling:
  - Code to a public interface
  - Not access public fields
  - Minimize friends
- High Cohesion
  - Classes working together to achieve a given task
  - MVC pattern

The Big Three

- Destructor: Make the base class destructor virtual

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

class TextBook : public Book {
    string topic;
    public:
    // ..
}

Textbook b1 = Textbook("Algorithms", "CLR", 500, "C++");
TextBook b2 = b1;
```

Default copy constructor and assignment operator operates on superclass first, then subclass.

```
TextBook::Textbook(const TextBook & other) : Book(other), topic(other.topic) {}

Book::Book(const Book & other) : title(other.title), author(other.author), numPages (other.

b1 = b2;

TextBook & TextBook::operator=(const TextBook & other) : {
    Book::operator=(other); // has access to this->
    topic = other.topic;
    return * this;
}
```

```
Book * pb1 = new TextBook("CS246", "Normair", 200, "C++");
Book * pb2 = new TextBook("CS136","Adam", 100, "C");

*pb1 = *pb2; // Calls assignment operator

cout << pb1 << endl; // Prints CS136, Adam, 100, C++
// Partial assignment. Only assignment for book gets called
// operator= by default is not virtual
```

Solution: Make `operator=` virtual.

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

class TextBook : public Book {
    public:
    virtual TextBook & operator=(const TextBook &); // Invalid, virtual signatures must be s
}

class TextBook : public Book {
    public:
    virtual TextBook & operator=(const Book &); // Invalid. Mixed assignment problem
}
```

Advice: Keep base class abstract

Casting

C Style cast

```
Node n;
int * p = (int *) & n;
```

In C++, there is a different syntax which stands out. There are four different kinds with varying power.

Static Cast

```
int m = 9;
int n = 2;
cout << m/n << endl; // prints 4
```

```
cout << static_cast<double>(m)/n << endl; // prints 4.5

Book * bp = new TextBook();
TextBook * tbp = static_cast<Textbook *>(bp);
```

Requirement: There needs to be an "is a" relationship.

If `bp` does not point to a `TextBook`, the cast fails at runtime, and leads to undefined behaviour.

Const cast

Used to add or remove the const property.

```
void g(int * p) {}
void f(const int * q) {
    g(q); // invalid signature
    g(const_cast<int *>(q));
    // cannot use q to change the value it is pointing to
}
```

- If g does not use p to change the value pointed to, this code still has defined behaviour.
- Since g didn't explicitly state that it took a const int pointer, it could change the value pointed by p, there could be undefined behaviour.

Reinterpret Cast

```
Vec v;
Student * sp = reinterpret_cast<Student *> (&v);
sp->grade();
```

Relies on low level compiler decisions on how objects are laid out in memory. Aka yolo cast.

Dynamic Cast

```
Book * collection[10];
for (...) {
    Book * bp = colection[i];
    TextBook * tbp = dynamic_cast<Textbook *>();
    if (tbp != NULL) {
        tbp->getTopic();
```

```
    } else {
        cout << "Not a textbook" << endl;
    }

}
```

To use the dynamic_cast hierarchy, needs at least one virtual method.

```
void whatIsIt(Book * bp) {
    if (dynamic_cast<Textbook *>(bp)) {
        cout << "Textbook" << endl;
    } else if (dynamic_cast<Comic *>(bp)) {
        cout << "Comic" << endl;
    } else {
        cout << "Book" << endl;
    }
}
```

- Brittle: Subject to undefined behaviour if hierarchy changes.
- Better to use virtual methods

```
Comic c(...);
Book & b = c;
Comic & cr = dynamic_cast<Comic &>(b); // What if c is not a comic? Cast fails, but can't b
```

Error Handling

- In C, a function encountering an error should set a value to global variable `errno`
- Require diligent monitoring/checking of `errno`

Examples:

- Dynamic cast to a reference fails
- New fails (no more memory)
- Vector's at method (out of range)

When an error occurs, C++ will terminate the program. If an exception is not caught, the program will terminate.

```
try {
    cout << v.at(3) << endl;
} catch (out_of_range r) { // catch exceptions to prevent termination
    cerr << "Bad range " << r.what() << endl;
}
```

Catch can catch exceptions and subclasses of the exception.

Throwing Exceptions

Must throw an exception if there is no try catch block in the immediate function. (Stack frame)

```
void f () {
  cout << "Start f" << endl;
  throw (out_of_range("f")); // must throw since exception is not handled in f
  cout << "Finish f" << endl;
}

void h() {
  cout << "Start h" << endl;    f ();    cout << "Finish h" << endl;
}

int main () {
  cout << "Start main" << endl;
  try {
    h();
    cout << "Done with h" <<endl;
  }
  catch (out_of_range) { cerr << "Range error" << endl; }
  cout << "Finish main" << endl;
}
```

Stack Unwinding

Keep popping stack frames until a handler for an exception is found. If a handler is not found, the program terminates.

Exception handling can be done in stages.

```
try {}
catch (Exception e) {
    // partial recovery
    // 1. Throw some other exception
    // 2. Throw; Throw the same exception
    // 3. Throw e; Throws a copy of the caught object
}
```

Advice: Use throw instead of throw e. Catch exceptions by reference.

All C++ library exceptions inherit from the exception class.

User defined exceptions do not need to inherit from the exception class.

```
try {}
catch(...) { // Use 3 dots to catch all exceptions
    //  handle error
}
```

In C++, any value can be thrown.

Advice: Create own exception types.

Common Exceptions

- `bad_cast` : When dynamic cast to reference fails
- `out_of_range` : (vector::at)
- `bad_alloc` : new fails (no more heap space)

```
class BaseExn{};
class DrivedExn : public Exn {};

try {
    DerivedExn // ...
    BaseExn & b = r;
    throw b;
} catch (DerivedExn &) {
} catch (BaseExn &) { // This will run. Only declared type is checked
} catch (...) {
}
```

Assignment Operator

```
Book * b1 = new TextBook;
Book * b2 = new TextBook;
*b1 = *b2; // Partial assignment
```

Make Book's operator= virtual and override it in textbook.

```
TextBook & Textbook::operator=(const Book & other) {
    Textbook & tother = dynamic_cast <TextBook *>(other);
    Book::operator=(other);
    topic = tother.topic;
    return *this;
}
```

Exception Safe Code

```
void foo () {
    MyClass * p = new MyClass;
    MyClass q;
    g(); // If g throws an exception, p will not be deleted
    delete p;
}
```

Exceptions must be recovered from.

C++ Guarantee: IF an exception occurs, and the call stack is being popped, any stack allocated memory for the function will be reclaimed.

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

Never let the destructors throw an exception. If there are two simultaneously live exceptions, program will terminate.

RAII: Resource Acquisition is initialization

- Every resource should be wrapped in a stack allocated variable/type whose job is to release the resource. Construct an object on the stack that points to dynamic allocated memory.

C++03 provides a template class `auto_ptr`

The constructor for `auto_ptr` accepts a pointer to dynamic memory. The destructor for `auto_ptr` causes delete on the pointer set during initialization.

```
class c {};

int main () {
  auto_ptr<c> p(new c(5));
```

```
  auto_ptr<c> q = p; // copy constructor and assignment operator steals the pointer
                     // p is then pointing to NULL

  // Still able to access fields through auto pointer
}
```

`auto_ptr` is deprecated in C++11. Use `unique_ptr`

`tr1::shared_ptr` uses reference count. IT prevents double free by only calling delete when there are no more pointers to the heap allocated memory.

Levels of Exception Safety

- Basic guarantee: An exception will leave the program in a valid state: No memory leaks or dangling pointers
- Strong guarantee: If an exception occurs, then the state of the program is as if the function was never called.
- No throw guarantee: A function always succeeds (achieves its task).

```
class A {};
class B {};
class C {
    A a;
    B b;
    void f() {
        a.g(); // g gives a strong guarantee;
        b.h(); // h gives a strong guarantee
    } // f does not provide strong guarantee: g succeeds, h fails
};
```

Assume that g and h have only local side effects. (Local to the object)

```
void C::foo() {
    A tempa = a;
    B tempb = b;
    tempa.g();
    tempb.h();
    a = tempa;
    b = tempb; // What if exceptions occur here?
}
```

Pointer assignment will never throw an exception.

```
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.b();
        std::swap(pImpl, temp);
    }
}
```

A virtual method increases the size of a structure/class.

Function Call:

- Load the address of the function
- Jump to the address

Non-virtual Methods

- Treated the same way as a function call, as multiple objects should not need to store another copy of the method.

Virtual Methods

- A VTable is created. Only one VTable is created per class. (not per object). The VTable stores the address of each function as well as the type of the original class.
- A pointer to the VTable is stored in the class with a virtual method.
- Load the VTable
- Locate the pointer for called method
- Jump to the address

Dynamic casts require at least one virtual method as they require access to the VTable.

Putting the VTable pointer at the top off an object allows you to treat a subclass object as a base class object as a portion of the subclass would be sliced off to become a base object.

Return Value Optimization

```
g++ -f no-elide-constructors
```