Object-Oriented Software Development

CS246

Jaiden Ratti

Prof. Brad Lushman

1239

Lecture 1

Intro to C++

In this course, we discuss the paradigm of object-oriented programming from 3 perspectives:

1. The programmer's perspective - how to structure programs correctly, and how to lower the risk of bugs
2. The compiler's perspective - what do our constructions actually mean, and what must the compiler do to support them?
3. The designer's perspective - how can we use the tools that OOP provides to build systems? Basic SE

Assume knowledge of C from CS136

C:

```c
#include <stdio.h>

int main()
{
    printf("Hello world!\n");
    return 0;
}

```

C++

```cpp
import <iostream>;
using namespace std;

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

Notes:

- main MUST return int - void main() illegal
- return stmt - returns the status code to OS
  - can be omitted from main (0 is assumed)
- stdio, printf still available in C++
- preferred I/O header <iostream>
  - std::cout ≪ _≪ _≪_
- std::endl = end of line & flush output buffer
- using namespace std - lets your omit the std:: prefix

To compile use: g++20h iostream (creates gcm.cache directory)

Then can compile program with: g++20m file name

./a.out will display the program

Input/Output

3 I/O streams

- cout/cerr — for printing to stdout/stderr
- cin — reading from stdin

I/O operators

- $\ll$ "put to" (output)
- $\gg$ "get from" (input)
- cerr $\ll$ x (produce value of x to output, from x to screen)
- cin $\gg$ x (grab value of x from input, from screen to x)

E.g. add 2 numbers

```
import <iostream>;
using namespace std;

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

In terminal: g++20m plus.cc -o plus, ./plus, (enter two numbers)

By default, cin skips leading whitespace (space/tab/newline)

What if bad things hapen, eg

- input doesn't contain an integer next
- input too large/small to fit in a variable
- out of input (EOF)

Statement fails

If the read failed: cin.fail() will be true

If EOF: cin.fail(), cin.eof() both true

But not until the attempted read fails!

Ex - read all ints from stdin & echo them, one per line, to stdout. Stop on bad input or eof

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

Note:

- There is an implicit conversion from cin's type (istream) to bool
- lets you use cin as a condition
- cin converts to true, unless the stream has had a failure

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

```

Note:

- $\gg$ is C's (and C++'s) right bitshift operator
- If a & b are ints, a$\gg$b shifts a's bits to right by b spots
- e.g 21 $\gg$ 3 21 = 10101 21 $\gg$ 3 = 2
- But when the LHS is an istream (i.e cin), » is the "get from" operator
- First example of overloading (same function has multiple implementations)

Recall:

- 21 $\gg$ 3 - bitshift
- cin $\gg$ x - input

First example of overloading (same function/operator with multiple implementations & the compiler chooses the correct implementation (at compile time)), based on the types of arguments.

Lecture 2

Operator $\gg$

- inputs: cin (istream): placeholder for data (several possible types)
- output?: returns cin (istream)

This is why we can write cin $\gg$ x $\gg$ y $\gg$ z

Stepper (goes from left to right), where cin carries through the left (cin»x = cin, $\implies$ cin»y etc.)

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

```

Successful read, cin evaluates as true, evaluates as false if otherwise.

Final Example

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

Ex: Read all ints & echo on stdout until EOF and skip non-integer input.

```
int main()
{
    int i;
    while(true) {
        if (!(cin>>i)) {
            if (cin.eof()) break; // done - eof
            cin.clear(); // Reset the streams failure flag
            cin.ignore(); // offending char is still in istream, remove
        }
        else cout<<i<<endl;
    }
}
```

The stream will not function after failure until you do this (clear).

```
cout << 95 << endl; // Prints 95
```

What if we want to print a # in hexadecimal?

```
cout << hex << 95 << endl; //Prints 5f
```

Hex is an I/O manipulator — puts stream into "hex mode" All subsequent ints printed in hex.

To go back to decimal mode: cout ≪ dec;

Note that manipulators like hex and dec set flags in the standard stream variables cout, etc. These are effectively global variables. I.e. changes you make to these flags affect the whole program

Good Practice: If you change a stream's settings, change them back when you are done

Strings

In C: array & char (char* or char[]) terminated by \0.

- Must explicitly manage memory — allocate more memory as strings get longer.
- Easy to overwrite the \0 & corrupt memory.

C++ strings: import <string>, type std::string

- Grow as needed (no need to manage the memory)
- Safer to manipulate

e.g

```
string s = "Hello";
```

"Hello" — C style string (character array ["H","e","l","l",o",\0]

s — C++ string created from the C string on initialization

String Operations

Equality/Inequality: $s1 == s2, s1! = s2$

Comparisons: $s1 <= s2$ etc. (lexicographic)

Length: s.length() (it is $O(1)$)

Individual Characters: s[0], s[1], s[2], etc.

Concat: $s3 = s1 + s2, s3+ = s4$

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

Skips leading white space & stops at white space (i.e. read a word). i.e. If given "hello world" to the above program, only returns "hello".

What if we want the white space? getline(cin,s)

- reads from the current position to the next new line into s

Streams are an abstraction — they wrap an interface of "getting and putting items" around the keyboard and screen.

Are there other kinds of "things" that could support this same "getting & putting" interface?

1. File
   (a) Read/write to/from a file instead of stdin/stdout
   (b) std::ifstream — a file stream for reading
   (c) std::ofstream — a file stream for writing

File access in C:

```
#include <stdio.h>
int main(void) {
    char s[256] // Hoping no word is longer than 255 chars
    FILE *f = fopen("file.txt", "r");
    while (true) {
        fscanf(f, "%255s",s);
        if (feof(f)) break;
        printf("%s\n",s);
    }
    fclose(f);
}

```

C++

```
import <iostream>;
import <fstream>;
import <string>;
using namespace std;

int main(){
    ifstream f {"file.txt"};
    string s;
    while (f >> s) { // using f the same way cin was
        cout << s << endl;
    }
}
```

f: Declaring & initializing the ifstream var opens the file.

Important: The file (file.txt) is closed when f goes out of scope.

Anything you can do with cin/cout, you can do with ifstream/ofstream.

Lecture 3

Recall: Other applications of the stream abstraction

1. Files
2. Strings
    (a) Extract data from chars in a string
        i. std::istringstream
    (a) Send data to a string as chars:
        i. std::ostringstream

```
import <sstream>;
string intToString(int n) {
    ostringstream sock;
    sock << n;
    return sock.str(); // extract the string
}
```

Convert string to #:

```
int n;
while (true){
    cout << "Enter a number"<<endl;
    string s;
    cin >> s;
    if (istringstream sock{s};sock>>n) break;
    // sock has that string (s) in it
    // taking n out of it
    // if successful, ends the loop, otherwise repeats
    cout<<"I said,";
}
cout<<"You entered"<<n<<endl;
```

Example Revisited: Echo all #'s, skip non-#s

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

This program picks up 123abc (as 123), but def456 fails to read any number.

Application: Consider processing the command line.

To accept command line args in C or C++, always give main the following params:

```
int main (int argc, char *argv[]) {..}
// int argc (count) is # of cmd line args
// >= 1 (includes the program name itself)

// int argv (vector) of c-style strings
// argv[0] = program name
// argv[1] = arg1, argv[2] = arg2 ...
// argv[argc] = null
```

eg.

```
./myprogram abc 123

//argc = 3
// argv = [[./myprogram\0],[abc\0],[123\0]]
```

Note: The args are stored as C-style strings. Recommendation: Convert to C++ strings for processing

eg.

```
int main(int argc,char *argv[]) {
    for (int i = 1; i<argc;++i){
        string arg = argv[i];
    }
}
```

eg: Print the sum of all numeric args on the cmd line

```
int main(argc,char*argv[]) {
    int sum = 0;
    for (int i = 1; i < argc; ++i){
        string arg = argv[i];
        int n;
        if(istringstream sock{arg}; sock >> n) sum += n;
    }
    cout << sum << endl;
}
```

Default Function Parameters

```
void printWordsInFile(string name = "words.txt"){
    ifstream file{name};
    for (string s; file >> s;) cout<<s<<endl;
}
printWordsInFile("othername.txt");
printWordsInFile(); // prints from words.txt
```

Note: Optional parameters must be **last**.

Also note: The missing parameter is supplied by the caller, not by the function

Why? The caller passes parameters by pushing them on the stack. The function fetches parameters by reading them off the stack. If a parameter is missing, the function as no way of knowing that. Would interpret whatever is in that part of the stack as the argument.

So instead, the caller must supply the extra parameter if it is missing. ∴ when you write printWordsInFile(); The compiler replaces this with printWordsInFile("words.txt")

For this reason, default arguments are part of a function's interface, rather than its implementation.

∴ defaults go in the interface file, not the implementation file.

Overloading

C:

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

C++: Functions with different parameter lists can share the same name

```
int neg(int n) {return -n;}
bool neg(bool b) {return !b;}
// referred to as overloading
```

Correct version of neg, for each function call, is chosen by the compiler (i.e at compile-time) based on the # and type of arguments in the function call.

neg(4) $= -4$, neg($true$) $= false$

∴ overloads must differ in #/types of arguments — just differing in the return type is not enough.

We've seen this already: == (ints/strings) $\gg$ (shift/I/O)

Structures

```
struct Node {
    int data;
    Node *next;
    // no longer need struct Node *next from C
}; // don't forget the semicolon.
```

Constants

```
const int maxGrade = 100; // must be initialized
```

Declare as many things constant as you can — helps catch errors

```
Node n {5,nullptr};

//syntax for null pointers in C++
// Do not say NULL in this course

const Node n2 = n; // immutable copy of n

// cannot mutate the fields
```

Lecture 4

Parameter Passing

Recall

```
void inc(int n) { ++n; }
...
int x = 5;
inc(x);
cout << x;; // returns 5
```

Pass-by-value — inc gets a copy of x, mutates the copy. Thus the original is unchanged

Solution: If a function needs to mutate an argument, you pass a pointer.

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

X's address is being passed by value. Increment changes the data address. Now visible to the caller.

Q:

```
scanf("%d",&x);
```

Why cin ≫ x and not cin ≫ &x?

A: C++ has another pointer-like type, reference.

References (**Important**)

```
int y = 10;
int &z = y; //z is an lvalue reference to y (int)
// like a constant pointer - similar to saying int *const z = &y;
```

References are like constant pointers with automatic dereferencing.

y [10]

z [pointer to y (arrow to 10)]

z [] can't change. Once pointing to y, it can't change.

If the const was on the other side of the =, then y would be constant.

z = 12; (NOT *z=12)

z = 12 **changes** the value associated with y (y[10]→ y[12]) now y == 12

int *p = &z;

Address of z gives the address of y.

In all cases, z acts exactly like y. Whatever changes happen to z are going to happen to y.

Z is an alias ("another name") for y.

Q: How can we tell when & means "reference" and when it means "address of".

A:

Whenever & occurs as part of a type (eg int &z). It always means reference.

When & occurs in an expression, it means "address of". (or bitwise -and).

lvalue, rvalue x = y; Interested in the value of y, and the address/location of x (not it's value).

Things you can't do with lvalue refs:

- Leave them uninitialized e.g. int &x;
  - Must be initialized with something that has an address (an lvalue).
    - int &x = 3; is illegal. (3 doesn't have a location)
    - int &x = y + z; is illegal (value of y + z has no location)
    - int &x = y; is legal
- Create a pointer to a reference
  - int &*p; (start at the variable work backwards)
    - int *&p = ; is legal
- Create a reference to a reference
  - int &&r = ; is illegal
    - denotes something different (later)
- Create an array of references
  - int &r[3] = {, , ,}; is illegal

What can you do?

- pass as function parameters

```cpp
void inc (int &n) { ++n; }

int x = 5;
inc(x); //don't do anything special to x
cout << x << endl; // 6

//no pointer dereference
//constant pointer to the argument x
//IT IS X
//changes affect x
```

Why does cin ≫ x work? Takes x as a reference.

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

Q: Why is the stream being taken & returned as a reference? And what does returning by reference mean?

A: We need a better understanding of pass-by-value

Pass-by-Value

Pass-by-value, e.g. int f (int n) { ... } copies the argument.

If the parameter is big, the copy is expensive

```
struct ReallyBig{...};

void f(ReallyBig rb); // copies rb, slow
void g(ReallyBig &rb); // alist-fast, but g may be change rb
void h(const ReallyBig &rb); //fast, no copy, parameter cannot be changed.
```

But what if a function does want to make changes to rb locally, but doesn't want these changes to be visible to the caller?

Then the function must make a copy of rb.

```
void k(const ReallyBig &rb) {
    ReallyBig rb2 = rb;
    // mutate rb2
}
```

But if you have to make a copy anyway, it is better to just use pass-by-value & have the compiler make it for you. It might be able to optimize something.

Advice: Prefer pass-by-const-ref over pass-by-value for anything larger than a pointer. Unless the function needs to make a copy anyway, then use pass-by-value.

Also:

```
void f(int &n);
void g(const int &n);

f(5); // illegal
//can't initialize an lvalue reference (n) to a literal value (non lvalue)
//if n changes, can't change the literal 5
g(5); // OK - since n can never be changed. The compiler allows this
//5 currently stored in a temporary location on the stack.
//So n can point there.
```

So in the case of

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

the istream is being passed (and returned) by reference to save copying.

This is important because stream variables are not allowed to be copied.

Dynamic Memory Allocation

C:

```
int *p = malloc(x*sizeofint);
...
free(p);
// DON"T USE THESE IN C++!
```

Instead: new/delete. Type-aware, less error prone.

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

Stack includes np Heap includes Node np points to Node.

Lecture 5

Recall:

```
Node *np = new Node;
...
delete np;
// np is in stack, pointing to..
// Node is in heap
```

- All local variables reside on the stack.
  - Variables are deallocated when they go out of scope (stack is popped)
- Allocated memory resides in the heap
  - Remains allocated until delete is called
- If you don't delete all allocated memory — memory leak
  - Program will eventually fail — we regard this as an incorrect program

Arrays

```
Node *nodeArray = new Node[10];
...
delete [] nodeArray;
```

Memory allocated with new must be deallocated with delete.

Memory allocated with new [...] must be deallocated with delete [].

Missing these = Undefined Behaviour (**UB**)

Returning by Value/Ptr/Ref

```
Node getMeANode() {
    Node n;
    return n;
}
// Expensive, n is copied to the caller's
// stack frame on return
// Return a pointer (or ref) instead?
```

```
Node *getMeANode() {
    Node n;
    return &n;
}
// BAD (one of worst things can do)
// Returns a pointer to stack allocated memory
// Which is dead on return. (UB)
```

```
Node &getMeANode() { //Also bad - same reason
    Node n;
    return n;
}
```

Q: Why was it OK for operator ≫ to return an istream reference

A: Because the reference is not to a local variable. The returned reference is the same reference that was passed as the parameter "in", so it refers to something accessible to the caller.

```
Node *getMeANode() {
    Node *np = new Node;
    return np;
}
// OK O returns a pointer to heap data (still alive)
// But don't forget to delete it when done with it
```

Which should you pick? Return by value. Often not as expensive as it looks (we will see why later).

Operator Overloading

Can give meanings to C++ operators for types we create

E.g.

```
struct Vec {
    int x,y;
}
Vec operator+(const Vec &v1, const Vec &v2) {
    Vec v{v1.x+v2.x, v1.y+v2.y};
    return v;
}
Vec operator*(const int k, const Vec &v){
    return {k*v.x,k*v,y};
    // OK because the compiler knows it's a vector, based on the
    // return type
}
Vec operator*(const Vec &v, const in k) {
    return k*v;
    // calling the above function
}

// now will work when Vec v4 {2 * v1} AND Vec v5 {v1 * 2}
```

Special Case: Overloading ≫ and ≪

```
struct Grade {
    int theGrade;
};
ostream &operator<<(ostream &out, const Grade &g){
    out << g.theGrade << '%';
    return out;
}

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

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

Separate Compilation

Split programs into modules, which each provide

- interface — type definitions, function headers
- implementation — full definition for every provided function

Recall: declaration — asserts existence definition — full details — allocates space (variables / functions)

E.g: Interface (vec.cc)

```
export module vec; // Indicates that this is the module interface file.

export struct Vec {
    int x,y;
};

// Anything marked "export" is available for the client to use

export Vec operator+(const Vec &v1, const Vec &v2);
```

main.cc

```
import vec;
int main() {
    Vec v{1,2,3};
    v = v+v;
    ...
}
```

Implementation: vec-impl.cc (see line 9 in interface block)

```
module vec;
// Thie file is part of module vec
// Implicitly importrs the interface
Vec operator+(const vec &v1, const vec &v2) {
    return {v1.x+v2.x, v1.y+v2.y}
}
```

Recall: An entity can be declared many times, but defined only once.

Interface files: start with export module ....; Implementation files: start with module ....;

Compiling separately: g++ -c ...cc

Above says, compile only, do not link, do not build exec.

Produces an object file (.o)

g++20m -c vec.cc

g++20m -c vec-impl.cc

g++20m -c main.cc

g++20m vec.o vec.-impl.o main.o -o main

./main Must be in dependency order

Dependency order: interface must be compiled before implementation, client.

Build tool support for compiling in dependency order (e.g. make) is still a work in progress.

Lecture 6

Classes

Can put functions inside of structs.

e.g: student.cc

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

student-impl.cc

```
float Student::grade () {
    return assns*0.4+mt*0.2+final*0.4;
}
```

client

```cpp
Student s {60,70,80};
cout << s.grade() << endl;
```

A class is essentially a struct type that can contain function. C++ does have a specific class keyword (later).

An object is an instance of a class.

```cpp
Student s {60,70,80};
// Student is the class, s is the object
```

The function grade is a member function or (method)

:: is called the scope resolution operator.

C::f means f in the context of class C.

:: like . where LHS is a class (or namespace), not an obj.

What do assns, mt, final mean inside of Student::grade?

- They are fields of the receiver objects — the object upon which the method was called
- e.g s.grade() is a method call that uses s's assns, int, final

Formally: methods take a hidden extra parameter called this — pointer to the receiver object.

e.g s.grade(); within grade(), $this == \&s$

Can write

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

(Methods can be written in the class. We will often do this for brevity. You should put impls in a separate file).

Initialization Objects

```cpp
Student s{60,70,80}; // OK, but limited
```

Better — write a method that initializes a constructor (ctor)

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

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

```

Student s{60,70,80}; // better
```

If a constructor has been defined, these are passed as arguments to the constructor. If no constructor has been defined, it is C-style field-by-field initialization. C-style is only available if you have not written a constructor.

Alternative syntax: Student s = Student{60,70,80};

Looks like construction of an anonymous student obj (i.e Student{60,70,80}) which is then copied to initialize s.

But it is not — semantically identical to Student s {60,70,80}; More on this later

Advantages of ctors: they're functions!

- Can write arbitrarily complex initialization code
- Default parameters, overloading, sanity checks

e.g

```
struct Student{
    ...
    Student (int assns=0, int mt=0, int final=0) {
        this->assns=assns;
        ...
    }
};
```

```
Student s2{70,80}; // 70,80,0
Student newKid; // 0,0,0

// Student newKid{}; and Student newKid; are identical
```

It may look like Student newKid; calls a ctor, and Student newKid; does not. That is not correct. Whenever an object is created, a ctor is always called.

Q: What if you didn't write one? e.g Vec v;

A: Every class comes with a default (i.e zero-arg) ctor (which just default-constructs all fields that are objects).

e.g

```
Vec v; // default ctor (does nothing in this case)
```

But the built-in default constructor goes away if you write any constructor.

e.g

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

```

Vec v{1,2} // OK
Vec v; // Erorr, doesn't compile
```

Continue with this definition for now.

Now consider:

```
struct Basis {
    Vec v1,v2;
};
Basis b; // Won't compile
```

The built-in default ctor for Basis wants to default-construct all fields that are objects. v1, v2 are objects, but they have no default ctor. So basis cannot have a built-in default ctor.

Could we write our own?

```
struct Basis {
    Vec v1, v2;
    Basis () {
        v1 = Vec{1,0};
        v2 = Vec{0,1};
    }
};
```

This also does not work. Why? Too late. The body of the ctor can contain arbitrary code, so the fields of the class are expected to be constructed and ready to use before the constructor body runs.

Object Creation Steps

When an object is created: 3 steps

1. Space is allocated
2. Fields are constructed in declaration order (i.e ctors run for fields that are objects)
3. Ctor body runs

Initialization (i.e. construction) of v1, v2 must happen in step 2, not step 3. How can we accomplish this?

Member Initialization List (MIL)

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

    // step 2                           // step 3
    // outside of {x} needs to be fields, inside are all parameters
```

Lecture 7

Recall: Member Initialization List (MIL)

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

    // step 2                       // step 3
    // outside of {x} needs to be fields, inside are all parameters
```

Note: can initialize any field this way, not just object fields.

```
struct Basis {
    Vec v1, v2;
    Basis(): v1{1,0}, v2{0,1}{}
// Step 1: v1{1,0}, v2{0,1}
// Step 2: {}

    Basis(const Vec &v1, const Vec &v2): v1{v1},v2{v2}{}
// What ctor for Vec is being called here?
};
```

Default values for the MIL

```
struct Basis {
    Vec v1{1,0},v2{0,1}; // If an MIL does not mention a field,
    // these values will be used
    Basis(){} // uses default values
    Basis(const Vec &v1, const Vec &v2): v1{v1},v2{v2}{}
}
```

Note:

Fields are initialized in the order in which they were declared in the class, even if the MIL orders them differently.

MIL: sometimes more efficient than setting fields in a constructor body

Consider:

```
struct Student{
    int assns,mt,final; // not objects
    string name; // object
    student(int assns,int mt, int final, string name){
        this->assns = assns;
        this->mt = mt;
        this->final = final;
        this->name = name;
    }
}
```

Name default-constructed (to the empty string) in step 2 and reassigned it.

Versus

```
Student (int assns, int mt, int final, string name):
assns{assns}, mt{mt}, final{final}, name{name}{}
```

Name is initialized to the correct value from the beginning in step 2, and there is no reassignment in step 3.

More efficient.

MIL must be used:

- For fields that are objects with no default ctor
- For fields that are constant or references

MIL should be used as much as possible. Embrace MIL.

Recall once again:

```
Basis::Basis (const Vec &v1, const Vec &v2): v1{v1},v2{v2}{}

// Consider
Student s {60,70,80};
Student s2 = s;
```

How does this initialization happen?

- The copy constructor
- For constructing one object as a copy of another

Note: Every class comes with

- default ctor (default-constructs all fields that are objects)
  - lost if you write any ctor
- copy ctor (just copies all fields)
- copy assignment operator
- destructor
- move ctor
- move assignment operator

Building your own copy ctor:

```
struct Student{
    int assns, mt, final;
    Student (const Student &other):assns{other.assns},mt{other.mt},
    final{other.final}{}
}; // equivalent to built-in
```

When is the built-in copy ctor not correct?

Consider:

```
struct Node{
    int data,
    Node *next;
};
Node *n = new Node{1,newNode{2,newNode{},{3,nullptr}}};
Node m = *n;
Node *p = new Node{*n}; // copy ctor
```

p points to 1 on the heap which points to 2 also on the heap

Simple copy of fields → only the first node is actually copied. (shallow copy).

If you want a deep copy (copies the whole list), must write your own copy ctor:

```cpp
struct Node{
    int data;
    Node *next;
    Node(const Node &other): data{other.data},
    next{other.next ? new Node {*other.next} : nullptr} {}
};
// new Node {*other.next} recursively copies the rest of the list
```

The copy ctor is called:

1. When an object is initialized by another object of the same type
2. When an object is passed by value
3. When an object is returned by value

The truth is more nuanced, as we will see.

Q: Why is this wrong:

```cpp
struct Node{
    int data,
    Node *next;
    Node(node other):___{}
};
```

A: Taking 'other' by value means 'other' is being copied, so the copy ctor must be called before we can begin executing the copy ctor (∞ recursion).

Note: Careful with ctors that can take one argument:

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

Single-arg ctors create implicit conversions:

E.g Node n4;

but also Node n = 4; implicit conversion from int to Node.

Seen this before with: string s "Hello"; Implicit conversion through single-arg ctor.

What's the problem?

```cpp
void f(Node n);
f(4); // works - 4 implicitly converted to Node.
```

Danger

- Accidentally passing an int to a function expecting a Node.
- Silent conversion
- Compiler does not signal an error
- Potential errors not caught

Don't do things that limit the compiler's ability to help you! Disable the implicit conversion:

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

```
Node n{4}; // OK
Node n = 4; // X
f(4); // X
f(Node{4}); // OK
```

Destructors

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

Lecture 8

Destructors

When an object is destroyed (stack-allocated: goes out of scope, heap-allocated: is deleted),

Method called destructor runs (dtor).

Classes come with a dtor (just calls dtors for all fields that are objects).

When an object is destroyed:

1. Dtor body runs
2. Fields' dtors are invoked in reverse delaration order (for fields that are objects)
3. Space is deallocated

When do we need to write a dtor?

```
Node *np = new Node {1, newNode {2, newNode {3, nullptr}}};
```

If np goes out of scope

- The pointer (np) is reclaimed (stack-allocated).
- The list is leaked

If we say delete np; calls *np's dtor, which doesn't do anything.

np [] → [1][-]→[2][-]→[3][/]

1 is freed, 2,3 are leaked

Write a dtor to ensure the whole list is freed:

```
struct Node {
    ~Node(){delete next;}
}
// recursively calls next's dtor
// whole list is deallocated
```

Now — delete np; frees the whole list

What happens when you reach the null pointer at the end of the list?

Deleting a null pointer is guaranteed to be safe (and to do nothing). The recursion stops.

Copy Assignment Operator

```
Student s1 {60,70,80};
Student s2 = s1; // copy ctor

Student s3; // default ctor
s3=s1; // copy, but not a construction
//^ copy assignment operator - uses compiler - supplied default
```

May need to write your own.

```
struct Node {
    ...
    Node &operator=(const Node &other){
        // Node & so that cascading works
        data = other.data;

        next = other.next ? newNode{*other.next}:nullptr;
        return *this;
    } // DANGEROUS
};
```

Why?

```
Node n {1,newNode{2,newNode{3,nullptr}}];
n=n; // deletes n.next and tries to copy n.next to n.next.
// UB
```

When writing operator=, ALWAYS make sure it behaves well in the case of self-assignment.

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

Q: How big of a deal is self-assignment? How likely am I to write $n = n$?

A: Not that likely. But consider $*p = *q$ if $p + q$ point at the same location.

Or a[i] = a[j] if $i + j$ happen to be equal (say, in a loop). Because of aliasing, it is a big deal!

Q: What's wrong with if $(*this == other)$ as a check for self-assignment

A: Exercise

An even better implementation.

```
Node &Node::operator=(const Node &other) {
    if (this == &other) return *this;
    Node *tmp = next;
    next = other.next ? new Node{*other.next} : nullptr;
    data = other.data;
    delete tmp;
    return *this;
} // if new fails, still have the old value of Node
```

Alternative: Copy + swap idiom

```
import <utility>;
struct Node {
    ...
    void swap(Node &other) {
        std::swap(data, other.data);
        std::swap(next, other.next);
    }
    Node &operator=(const Node &other) {
        Node tmp = other;
        swap(tmp) // I am a deep copy of other, tmp is old me
        return *this; // tmp is stopped, dtor runs, destroys my old data
    }
    ...
};
```

RValues & RValue References

Recall:

- An lvalue is anything with an address
- An l value reference (&) is like a constant ptr with auto-dereferencing. Always initialized to an lvalue

Now consider:

```
Node oddsOrEvens() {
    Node odds {1,newNode{3,newNode{5,nullptr}}};
    Null evens {2, newNode{4,newNode{6,newNode}}};

    char c;
    cin >> c;
    if (c == '0') return evens;
    else return odds;

    Node n = oddsOrEvens(); // copy ctor:

}
```

Lecture 9

RValues, RValue Refs

```
Node oddsOrEvens() {
    Node odds {1,new Node{3,newNode{5,nullptr}}};
    Node evens {2,new Node {4, newNode {6, nullptr}}};
    char c;
    cin >> c;
    if (c=='0') return evens;
    else return odds;
}
Node n{oddsorEvens()}; // copy ctor called #node times
// what is "other" here?
// reference to what?
```

- Compiler creates temporary object to hold the result of oddsorEvens
- Other is a reference to this temporary
  - Copy ctor deep-copies the data from this temporary

But

- The temporary is just going to be discarded anyway, as soon as the start Node n oddsOrEvens(); is done.
- Wasteful to have to copy from the temp
  - Why not just steal it instead? — save the cost of a copy
- Need to be able to tell whether other is a reference to a temporary object (where stealing would work) or a standalone object (where we would have to copy).

C++ - rvalue reference Node && is a reference to a temporary object (rvalue) of type Node.

Version of the ctor that takes a Node &&

```
struct Node {
    ...
    Node(Node &&other): // called move ctor
        data{other.data}; // steals other's data
        next{other.next} {
            other.next=nullptr;
        }
}
```

Similarly:

```
Node m;
...
m - oddsOrEvens(); //assignmnent from temporary
```

Move assignment operator:

```
struct Node {
    ...
    Node &&operator=(Node &&other) {
    // steal other's data
    // destroy my old data
    // Easy: swap without copy
    std::swap(data, other.data);
    std::swap(next, other.next);
    return *this;
    // the temp will be destroyed & take
    // my old data with it
    }
}
```

If you don't define move operations, copying versions of them will be used instead.

Copy/Move Elision

```
vec makeAVec() {
    return{0,0}; // invokes a basic Vec ctor
}

Vec v = makeAVec(); // what runs? Move ctor? Copy ctor?
```

Answer: Just the basic ctor. No copy ctor, no move ctor.

In some cases, the compiler is required to skip calling copy/move ctors.

In this example, makeAVec writes its result (0,0) directly into the space occupied by v in the caller, rather than copy it later.

Eg

```
void doSomething(Vec v); // pass-by-value - copy/move ctor
doSomething(makeAVec()); //result of makeAVec written directly into the param
// no copy or move
```

This happens, even if dropping ctor calls would change the behaviour of your program (e.g. if ctors print something).

You are not expected to know exactly when elision happens — just that it does happen.

In summary: Rule of 5 (Big 5)

If you need to write any one of

1. copy ctor
2. copy assignment operator
3. dtor
4. make ctor
5. move assignment operator

Then you usually need to write all 5.

But note that many classes don't need any of these. The default implementations are fine.

What characterizes classes that need the big 5, typically? ownership

- these classes are usually tasked with managing something (often memory), but there are other things that need managing (resources).

Notice:

Operator= is a member function. Previous operators we've written have been standalone functions.

When an operator is declared as a member function, this plays role of the first operand.

```
struct Vec {
    int x,y;
    ...
    Vec operator+(const Vec &other) {
        return {x+other.x, y+other.y};
    }
    Vec operator*(const int k) {
        return {x*k, y*k};
        // implements 50% of the time
        // implements v * k
    }

}
```

How do we implement k*v? Can't be a member function — first arg not a Vec! Must be external:

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

Advice: If you overload arithmetic operators, overload the assignment versions of these as well, and implement, e.g. + in terms of +=.

E.g

```
Vec &operator+= (Vec &v1, const Vec &v2) {
    v1.x += v2.x;
    v1.y += v2.y;
    return v1;
}
Vec &operator+(const Vec &v1, const Vec &v2){
    Vec temp{v1};
    return tmp += v2; // uses += to implement +

}
```

I/O Operators:

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

What's wrong with this? Makes vec the first operand, not the second.

→ use as v ≪ cout | w ≪ (v ≪ cout)

So define operator ≪,≫ as standalone. Would have to put the stream on the right

Certain operators must be members:

- operator=
- operator[]
- operator→
- operator()
- operator T (where T is a type)

Lecture 10

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

Vec *p = new Vec[10];
Vec moreVecs[10];
// these want to call the default ctor on each item.
// If no default ctor, can't initialize items (error)
```

Options:

1. Provide a default ctor
   (a) This is not a good idea unless it makes sense for the class to have a default ctor
2. For stack arrays:
   (a) Vec moreVecs[3] = 0,0,1,1,2,4;
3. For heap arrays — create an array of pointers

```cpp
Vec **vp = new Vec*[5];
vp[0] = new Vec{0,0};
vp[1] = new Vec{1,1};
..
for (int i = 0: i < 5; ++i) delete vp[i];
delete [] vp;
```

Const Objects

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

Const objects arise often, especially as parameters.

What is a const object?

- Fields cannot be mutated

Can we call methods on a const obj?

Issue: The method may modify fields, violate const.

A: Yes — we can call methods that promise not to modify fields.

Eg:

```cpp
struct Student {
    int assns, mt, final;
    float grade() const;
    // doesn't modify fields, so declare it const
}
```

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

Now consider: want to collect usage stats on Student objs:

```cpp
struct Student {
    int numMethodCalls = 0;
    float grade() const{
        ++numMethodCalls;
        return ___;
    }
};
```

- Now can't call grade on const students — it can't be a const method.
- But mutating numMethodCalls affects only the physical constness of student objects, not the logical constness.
- Physical constness — whether actual bits that make up the object have changed at all.
- Logical constness — whether the object should be regarded as different after the update.

Want to be able to update numMethod calls, even if the object is const:

```cpp
struct Student {
    ...
    mutable int numMethodCalls=0;
    float grade() const {
        ++numMethodCalls;
        return ___;
    }
};
```

Mutable fields can be changed, even if the object is const. Use mutable to indicate that the field does not contribute to the logical constness of the object.

Static Fields & Methods

numMethodCalls tracked the # of times a method was called on a particular object. What if we want the # of times a method is called over all student objects.

Eg: What if we want to track how many students are created?

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

```cpp
struct Student {
    ...
    inline static int numInstances = 0;
    Student(___); ____ {
        ++numInstances;
    }
    // only one, not one per student
};
```

Static member functions — don't depend on the specific instance (no this parameter). Can only access static fields & other static functions.

```
struct Student {
    ...
    inline static int numInstances = 0;
    ...
    static void howMany() {
        cout << numInstances << endl;
    }
};

Student s1{60,70,80}, s2{70,80,90};

Student::howMany(); // 2

```

Comparing Objects

Recall: string comparison in C.

```
strcmp(s1,s2) =
// < 0 if s1<s2
// 0 if s1=s2
// > 0 if s1>s2
// done lexicographically
```

Linear scan, char-by-char comparison.

Compare to string comparison in C++:

```
s1<s2, s1==s2, s1>s2, etc.
```

C++ version is easier to read. But one drawback.

Consider:

```
string s1 = ___, s2 = _____;
if (s1< s2) {...} // compare s1 & s2
else if (s1 == s2) {...} // compare s1 & s2 again
else {...}
```

Two comparisons! Versus with strcmp

```
int n = strcmp(s1, s2); // char *s1, *s2;
if (n < 0) {...}
else if (n == 0) {...}
else {..}
// only one comparison
```

Can we achieve the same using C++ strings, i.e have one comparison that answers whether s1 >, =, < s2?

Introducing the 3-way comparison operator <=>

```
import <compare>;
string s1 = ____,s2=____;

std::strong-ordering result = s1<=>s2;
// makes one comparison
if (result <0) cout << "less";
else if (result == 0) cout << "equal";
else cout << "greater";
```

Side note:

```
//std::strong-ordering is a lot to type & difficult to remember
// shortcut
auto result = s1 <=> s2;
// automatic type deduction
auto x = expr;
// declares x to have a type meatching that of the value of expr.
```

Lecture 11

Recall:

```
auto result = s1 <=> s2;
// 3-way comparison
// automatic type detection
```

How can we support <=> in our own classes?

```
struct Vec {
    int x,y;
    auto† operator <=> (const Vec &other) const {
        auto n = x <=> other.x;
        return (n==0)? y <=> other.y: n;
    }
}
```

Now we can say

```
Vec v1{1,2},v2{1,3};
v1 <=> v2;
```

But we can also say v1 <= v2, etc. The 6 relational operators <, <=, ==, !=, >, >= automatically rewritten in terms of <=>.

Eg v1 <= v2 → (v1 <=> v2) <= 0

6 operators for free! But you can also sometimes get operator <=> for free!

```
struct Vec {
    int x,y;
    auto operator <=> (const Vec &other) const = default;
    // does lexicographical ordering on fields of Vec.
    // Equivalent to what we wrote before
}
```

When might the default behaviour not be correct.

```
Struct Node {
    int data;
    Node *next;
    // lex order on these fields would compare ptr values - not useful
}
```

Starship operator for Node

```
struct Node {
    int data;
    Node *next;
    auto operator <=> (const Node &other) const {
        // Note: works for non-empty lists
        auto n = data <=> other.data;
        if (n!=0) return n;
        if (!next && !other.next) return n; // n already equal
        if (!next) return std::strong.ordering::less;
        if (!other.next) return std::strong.ordering::greater;
        return *next <=> *other.next;
    }
};
```

Invariants & Encapsulation

Consider:

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

Node n {2,nullptr};
Node m {3,&n};
```

What happens when these go out of scope?

m's dtor tries to delete n, but n is on the stack, not on the heap! UB!

Class Node relies on an assumption for its proper operation: that next is either nullptr or was allocated by new.

This is an example of an invariant. Statement that must hold true, upon which Node relies.

We can't guarantee this invariant — can't trust the user to use Node properly.

Eg Stack — invariant — last item pushed is the first item popped. But not if the client can rearrange the underlying data.

Hard to reason about programs if you can't rely on invariants.

To enforce invariants, we introduce encapsulation.

- Want clients to treat objects as black boxes — capsules
- Creates an abstraction — seal away details
  - Only interact via provided methods

Eg

```cpp
struct Vec {
    Vec(int x, int y); // also public
    // default visibility is public
    private:
        int x,y;
    public:
        Vec operator+(const Vec &other);
        ...
};
// Private, can't be accessed outside of struct Vec
// Public, anyone can access
```

In general: want private fields; only methods should be public.

Better to have default visibility = private.

Switch from struct to class.

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

Difference between struct & class — default visibility. Struct is public, class is private.

Let's fix our linked list class.

list.cc

```cpp
// externally this is struct list::Node
export class list {
    struct Node; //private nested class- only accessible within class list
    Node *theList = nullptr;
public:
    void addToFront(int n);
    &ith(int i); // means we can do lst.ith(4) = 7;
    ~List();
}
```

list-impl.cc

```
struct list::Node { // nested class
    int data;
    Node *next;
    ...
    ~Node() {delete next;}
};
void List::addToFront(int n) {
    theList = new Node{n, theList};
}
int &list::ith(int i) {
    Node *cur = theList;
    for (int j = 0; j < i; ++i) cur = cur->next;
    return cur->data;
}
List::~List(){delete theList;}
```

Only List can create/manipulate Node obs now.

∴ can guarantee the invariant that next is always either nullptr or allocated by new.

Iterator Pattern

- Now we can't traverse node to node as we would a linked list.
- Repeatedly calling ith = $O(n^2)$ time
- But we can't expose the nodes or we lose encapsulation.

SE Topic: Design Patterns

- Certain programming challenges arise often. Keep track of good solutions. Reuse & adapt them.

Solution — Iterator Pattern

- Create a class that manages access to nodes
- Will be an abstraction of a pointer — walk the list without exposing nodes.

Recall (c):

```
for (int *p = arr; p != arr+size; ++p) {
    printf("%d", *p);
}
```

```
class List {
    struct Node;
    Node *theList = nullptr;
public:
    class Iterator {
        Node *p;
    public:
        Iterator(Node *p):p{p}{}
        Iterator &operator++() { p = p->next; return *this;}
        bool operator != (const Iterator other) const { return p!= other.p;}
        int &operator*() {return p->data;}
```

```cpp
    };
    Iterator begin() {return Iterator{theList};}
    Iterator end() { return Iterator{nullptr};}
};
```

Lecture 12

Recall: Iterator Pattern

```cpp
class List {
    struct Node;
    Node *theList = nullptr;
public:
    class Iterator {
        Node *p;
    public:
        Iterator(Node *p):p{p}{}
        Iterator &operator++() { p = p->next; return *this;}
        bool operator != (const Iterator other) const { return p!= other.p;}
        int &operator*() {return p->data;}
    };
    Iterator begin() {return Iterator{theList};}
    Iterator end() { return Iterator{nullptr};}
};
```

Client code

```cpp
List l;
l.addToFront(1);
l.addToFront(2);
l.addToFront(3);
for (List::Iterator it = l.begin(); it != l.end(); ++it) {
    cout << *it << endl;
}
// can replace List::Iterator with auto
```

This now runs in linear time, and we are not exposing the pointers.

Midterm Cutoff Here

Shortcut: range-based for loop

```cpp
for (auto n:l) {
    cout << n << endl;
}
// n stands for item in list here
// changes made on n here will not actually modify what is on the list
// n is a copy
```

This is available for any class with

- methods begin & end that produce iterators

- the iterator must support ! =, unary *, prefix++

If you want to modify list items (or save copying):

```
for (auto &n: l) {
    ++n;
}
```

Encapsulation (continued)

List client can create iterators directly:

```
auto it = List::Iterator{nullptr};
// functions as end operation but did not call end
```

Violates encapsulation because the client should be using begin/end. Don't want client calling iterators directly.

We could — make Iterator's constructor private → then client can't call List::Iterator. But, then neither can list.

Solution

Give List privileged access to Iterator. Make it a friend.

```
class List {
    ...
    public:
        class Iterator {
            Node *p;
            Iterator (Node *p);
        public:
            ...
            ...
            friend class List;
        };
};
// List has access to ALL members of Iterator
// Note: does not matter where in class Iterator you put this
```

Now List can still create Iterators, but client can only create Iterators by calling begin/end.

Give your classes as few friends as possible — weakens encapsulation.

Providing access to private fields. Accessor / mutator methods.

```
class Vec {
    int x,y;
public:
    ...
    int getX() const { return x;} //accessor
    void setY(int z) { y = z;} // mutator
};
// Since we make setY, we can limit certain values
```

What about operator≪

- needs x, y, but can't be a member
- if getX, getY defined — OK
- if you don't want to provide getX, getY — make operator $\ll$ a friend f'n.

Friendship does not go both ways.

```cpp
class Vec {
    ...
public:
    friend ostream &operator<<(ostream &out, const Vec &v);
    // non member function
};

ostream &operator<<(ostream &out, const Vec &v)
    return out << v.x << ' ' << v.y;
    // Note: has access to Vec fields.
}
```

Equality Revisited

Suppose we want to add a length() method to List: How should we implement it?

Options:

1. Loop through the nodes & count them. $O(n^2)$
2. Store the length as a field & keep it up to date. $O(1)$ length with negligible additional cost to addToFront.

Option 2 is generally preferred.

But consider again the spaceship operator <=> in the special case of equality checking:

$l1 == l2$ translates to $(l1 <=> l2) == 0$

What is the cost of <=> on 2 lists? O(length of shorter list).

But for equality checking, we missed a shortcut: Lists whose lengths are different cannot be equal.

In this case, we could answer "not equal" in $O(1)$ time.

```cpp
class List {
    ...
    Node *theList;
public:
    auto operator<=>(const List &other) const {
        if (!theList && !other.theList){
            return std::strong-ordering::equal;
        }
        if (!theList) return std::strong-ordering::less;
        if (!other.theList) return std::strong-ordering::greater;
        // both non empty
        return *theList <=> *other.theList;
        // comparing pointers so we need to dereference
    }
    bool operator==(const List &other) const {
        if (length != other.length) return false; // O(1)
        return (*this <=> other) == 0;
    }
};
```

Operator <=> gives automatic impl's to all 6 additional operators, but if you write operator== separately, the compiler will use that for both == and != instead of <=>. Lets you optimize your equality checks, if possible.

System Modelling

Visualize the structure of the system (abstractions & relationships among them) to aid design, implementation, communication.

Popular standard: UML (Unified Modelling Language)

Modelling class

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

Access:

- - private
- + public

Relationship: Composition of Classes

Recall:

```
class Basis {
    Vec v1,v2;
}
```

Embedding one object (Vec) inside another (Basis) is called composition.

A basis is composed of 2 Vecs. They are part of a Basis, and that is the only purpose of those Vecs.

Relationship: a Basis "owns a" Vec (in fact, it owns 2 of them).

If A "owns a" B, then typically

- B has no identity outside A (no independent existence).
- If A is destroyed, B is destroyed.
- If A is copied, B is copied (deep copy)

Eg A car owns its engine — the engine is part of the car.

- Destroy the car → destroy the engine
- copy the car → copy the engine

Implementation: Usually as composition of classes.

Modelling (*see image*) A (diamond arrow) B A owns some # of B's

Can annotate with multiplicities field names.

Aggregation Compare car parts in a car ("owns a") vs. car parts in a catalogue. The catalogue contains the parts, but the parts exist on their own.

"Has a" relationship (aggregation).

If A "has a" B, then typically

- B exists apart from its association with A
- If A is destroyed, B lives on

Node
- data: Integer

Table 1: filled in diamond on Integer with arrow to Node. 0..1 (next)

- If A is copied, B is not (shallow copy) — copies of A share the same B.

Lecture 13

Recall: Aggregation.

eg: parts in a catalogue, ducks in a pond.

UML: [pond][diamond]→ 0...*[duck]

Typical Implementation: pointer fields

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

Case Study: Does a pointer field always mean non-ownership?

No! Let's take a close look at lists & nodes.

[see table at top of page] fig 1

A Node owns the Nodes that follow it (Recall: implementation of Big 5 is a good sign of ownership).

[fig 2]

But these ownerships are implemented with pointers.

Another view of Lists & Nodes.

[fig 3]

We could view the List object as owning all of the Nodes within it.

What might this suggest about the implementation of Lists & Nodes in this case?

Likely — List is taking responsibility copying and construction/destruction of all Nodes, rather than Node.

Possible iterative (i.e loop-based) management of pointers vs. recursive routines when Nodes managed other Nodes.

Inheritance (Specialization)

Suppose you want to track a collection of Books.

```cpp
class Book {
    string title, author;
    int length; // pages
 public:
    Book(___);
    ...
};
```

For Textbooks — also a topic:

```
class Text {
    string title, author;
    int length;
    string topic;
 public:
    Text(___);
    ...
};
```

For comic books, want the name of the hero:

[fig 4]

This is OK — but doesn't capture the relationships among Book, Text, Comic.

And how do we create an array (or list) that contains a mixture of these?

Could

1. Use a union

```
union BookTypes{Book *b; Text *t; Comic *c;};
BookTypes myBooks[20];
```

2. Array of void * — pointer to anything

Not good solutions — break the type system.

Rather, observe: Texts and comics are kind of Books — Books with extra features.

To model that in C++ — inheritance

```
class Book {
    string title, author;
    int length;
 public:
    Book(---);
    ...
};
// Base class (or super class)

class Text: public Book {
    string topic;
 public:
    Text(---);
    ...
};

class Comic: public Book {
    string hero;
 public:
    Comic(---);
    ...
};
```

```

// Derived classes (or subclasses)
```

Derived classes inherit fields & methods from the base class.

So Text, Comic, get title, author, length fields.

Any method that can be called on Book can be called on Text, Comic.

Who can see these members?

title, author, length — private in Book — outsiders can't see them.

Can Text, Comic see them? No — even subclasses can't see them.

How do we initialize Text? Need title, author, length, topic. First three initialize the Book part.

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

// Does not compile
```

Wrong for two reasons.

1. Title, etc. are not accessible in Text (and even if they were, the MIL only lets you mention your own fields).
2. Once again, when an object is created:
   (a) Space is allocated
   (b) Superclass part is constructed *new*
   (c) Fields are constructed in declaration order
   (d) Constructor body runs

So a constructor for Book must run before the fields of Text can be initialized. If Book has no default constructor, a constructor for Book must be invoked explicitly.

```
class Text: public Book {
    ...
 public:
    Text(string title, string author, int length, string topic):
    Book{title,author,length},topic{topic} {}
    // step 2                  step 3       step 4
    ...
};
// this is how to intialize when don't have access
```

Good reasons to keep superclass fields inaccessible to subclasses.

If you want to give subclasses access to certain members: protected access:

```
class Book {
    protected:
        string title,author;
        int length;
        //accessible to Book and its subclasses
        // but no one else
    public:
        Book(---);
        ...
};

class Text: public Book {
    ...
    public:
        ...
        void addAuthor(string newAuthor) {
            author += newAuthor;
            // OK
        }
}
```

Not a good idea to give subclasses unlimited access to fields.

Better: keep fields private, provide protected accessors/mutators

```
class Book {
    string title, author;
    int length; // pages
 protected:
    string getTitle() const;
    void setAuthor(string newAuthor);
    // subclasses can call these
 public:
    Book(---);
    bool isHeavy() const;
```

Relationship among Text, Comic, Book — called "is—a"

- A Text is a Book
- A Comic is a Book

[fig 5]

Now consider the method isHeavy — when is a Book heavy?

- for ordinary books — > 200 pages
- for Texts — > 500 pages
- for Comics — > 30 pages

```
class Book {
    ...
    protected:
        int length;
    public:
        bool isHeavy() const { return length > 200;}
};

class Comic: public Book {
    ...
    public:
        ...
        book isHeavy() const { return > 30;}
        ...
};
etc.
```

Lecture 14

Recall: isHeavy

bool isHeavy() const:

- for Books: > 200 pages
- for Texts: > 500 pages
- for Comics > 30 pages

```
Book b {"A small book", ___, 50};
Comic c{"A big comic", ___, 40, "Hero"};

cout << b.isHeavy(); // false
cout << c.isHeavy(); // true

```

Now since public inheritance means "is a", we can do this.

```
Book b = Comic{"A big comic",___,40,___};
```

Q: Is b heavy $\iff$ I.e. b.isHeavy() — true or false? $\iff$ Which isHeavy runs? Book::isHeavy or Comic::isHeavy?

A: No b is not heavy, Book::isHeavy runs.

Why? Book b [title, author, length space on stack] = Comic [title, author, length, hero]. Tries to fit a comic object where there is only space for a Book object. What happens? Comic is sliced — **hero field chopped off**.

- Comic coerced into a Book

So Book b = Comic{...}; creates a Book and Book::isHeavy runs.

Note: slicing takes place even if the two object types are the same size. Having the behaviour of isHeavy depend on whether Book & Comic have the same size would not be good.

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

```
Comic {___,___,40,___};
Book *pb = &c // wont slice
c.isHeavy(); // true
pb->isHeavy(); // still False!
// ... and still Book::isHeavy runs when we access pb->isHeavy()
```

Compiler uses the type of the pointer (or ref) to decide which isHeavy to run — does not consider the actual type of the object (uses the declared type).

Behaviour of the object depends on what type of pointer or reference you access it through.

How can we make Comic act like a Comic, even when pointed to by a Book pointer, i.e. How can ewe get Comic::isHeavy to run?

Declare the method virtual.

```
class Book {
    ...
 public:
    virtual bool isHeavy() const {
        return length > 200;}
    };
class Comic:public Book {
    ...
 public:
    bool isHeavy() const override {return length >30;}
    // override = make sure that this overrides the virtual
};
```

```
Comic c {__,__,40,__};
Comic *pc = &c;
Book *pb = &c;
Book &rb = c;
pc->isHeavy(); // true
pb->isHeavy(); // true
rb.isHeavy(); // true

//Comic::isHeavy runs in all 3 cases
```

Virtual methods — choose which class' method to run based on the actual type of the object at run time.

E.g. My book collection

```
Book *myBooks[20];
...
for (int i = 0; i < 20; ++i) {
    cout << myBooks[i]->isHeavy() << endl;
}
// isHeavy() above uses Book::isHeavy for Books
//                      Text::isHeavy for Texts
//                      Comic::isHeavy for Comics
// automatically
```

Accommodating multiple types under one abstraction: polymorphism. ("many forms").

Note: this is why a function void f(istream) can be passed an ifstream — ifstream is a subclass of istream.

Danger!: What if we had written Book myBooks[20], and tried to use that polymorphically.

Consider:

```
void f(Book books[]) {
    books[1] = Book{"book", "book author", ___};
}
Comic c[2] = {{"comic 1", "artist 1", 10, "hero1"},
                {"comic 2", "artist 2", 20, "hero2}};

f(c); // legal - a Comic * is a Book*
```

What might c be now? (UB).

```
{{"comic 1", "artist 1", 10, "book"},
{"book author", ???,...}};
```

Never use arrays of objects polymorphically. Use array of pointers.

Destructor Revisited

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

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

Is it wrong that Y doesn't delete x?

No, not wrong:

1. x is private. Can't do it anyway
2. When an object is destructed:
   (a) Destructor body runs
   (b) Fields are destructed in reverse declaration order
   (c) Superclass part is destructed ($\tilde{\text{Y}}$ implicitly calls $\tilde{\text{X}}$)
   (d) Space is deallocated

```
X *myX = new Y{10,20};
delete myX;
// leaks! Why? -calls ~X but not ~Y
// only x, but not y, is freed.
```

How can we ensure that deletion through a pointer to the superclass will call the subclass destructor? Make the destructor virtual!

```
class X{
    ...
 public:
    ...
    virutal ~X() {delete []x;}
};
```

ALWAYS — make the destructor virtual in classes that are meant to have subclasses. Even if the virtual destructor doesn't do anything.

If a class is not meant to have subclasses, declare it final.

```
class Y final:public X{...};
```

Pure Virtual Methods & Abstract Classes

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

2 kinds of student: regular & co-op.

```
class regular:public Student {
    public:
        int fees() const override;
        // computes students' fees
}
class CoOp: public Student{
    public:
        int fees() const override;
        // computes students' fees
}
```

What should we put for Student::fees?

Not sure — every student should be regular or co-op.

Can explicitly give Student::fees no implementation.

```
class Student {
    ...
 public:
    virtual int fees() const = 0;
    // method has no (*) implementation
    // called a pure virtual method
    // subclass must override this because we don't have impl
};
```

A class with a pure virtual method cannot be instantiated.

Student s; (will not compile) (needs to be either regular or co-op).

Called an abstract class. Its purpose is to organize subclasses.

Lecture 15

Abstract class

```cpp
class Student {
    ...
    public:
    virtual int fees() const = 0;
    // pure virtual method
};
student s; // cannot be instantiated
```

Subclasses of an abstract class are also abstract, unless they implement all pure virtual methods.

Non-abstract classes are called concrete.

```cpp
class Regular: public Student { // concrete
    public:
        int fees() const override {...}
};
```

UML:

- virtual/pure virtual methods — italics
- abstract classes — class name in italics
- # protected : static

Inheritance and Copy/Move

```cpp
class Book {
    public:
    // defines Big 5
}

class Text: public Book {
    string topic;
 public:
    // does not define the big 5
};

Text t {"Algorithms","CLRS",500000,"cs"};
Text t2 = t; // No copy ctor in Text - what happens?
```

- It calls Book's copy ctor
- Then goes field-by-field (i.e default behaviour) for the Text part
- Same for the other operations

To write your own operations:

```

//copy constructor
Text::Text(const Text &other): Book{other},topic{other.topic} {}

// copy assignment operator
Text &Text::operator=(const Text &other) {
    Book::operator=(other);
    // then assign topic yourself (the field belonging to text)
    topic = other.topic;
    return *this;
}

// move constructor (WRONG)
Text::Text(Text &&other): Book{other}, topic{other.topic} {}
// other is an lvalue
// these are both copy constructors, not move constructors
// won't work

Text::Text(Text &&other): Book {std::move(other)}, topic{std::move(other.topic)} {}
// std::move is a function that treats an lvalue as an rvalue
// now acts as move instead of copy

Text &Text::operator=(Text &&other) {
    Book::operator=(std::move(other));
    // topic = other.topic would be a copy
    topic = std::move(topic.other);
    // strings move assignment operator
    return *this;
}
```

Note: even though ‘other’ points at an rvalue, other itself is an lvalue (so is other.topic).

std::move(x) forces an lvalue x to be treated as an rvalue, so that the "move" versions of the operations run.

Operations above are equivalent to the default — specialize as needed for Nodes, etc.

Now consider:

```
Text t1{,,,}, t2{,,,};
Book *pb1 = &t1, *pb2 = &t2;

// What if we do *pb1 = *pb2?
// Book::operator= runs
```

Partial assignment — copies only the Book part.

How can we prevent this? Try making operator= virtual.

```
class Book {
    ...
    public:
        virtual Book &operator=(const Book &other) {...}
};
class Text {
    ...
    public:
        Text &operator=(const Book &other) override {...}
        // not Text &operator=(const Text &other) override {.}
}
```

Note: Different return types are fine, but parameter types must be the same, or it's not an override (and won't compile). Violates "is a" if they don't match.

Assignment of a Book object into a Text object would compile:

```
Text t{...};
t = Book{...};
// using a Book to assign a Text
// BAD (but it compiles)

//Also
t = Comic{...}; -- REALLY BAD
```

If operator= is non-virtual — partial assignment through base class pointers.

If virtual — allows mixed assignment — also **bad**.

Recommendation: all superclasses should be abstract.

Rewrite Book hierarchy [see figure 6]

```
class AbstractBook {
    string title, author;
    int length;
 protected:
    AbstractBook &operator*(const AbstractBook &other);
    // prevents assignment through base class ptrs from compiling,
    // but implementation still available for subclasses.
 public:
    AbstractBook(---);
    virtual ~AbstractBook() = 0;
    // Need at least one pure virtual method If you don't have one,
    // use the destructor (to make the class abstract).
};
```

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

Other classes — similar. Prevents partial & mixed assignment.

Note: virtual dtor MUST be implemented, even though it is pure virtual.

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

Because subclass destructors WILL call it as part of Step 3.

Templates

Huge topic — just the highlights

```
class List {
    struct Node {
        int data;
        Node *next;
        ...
    };
    ...
};
```

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

OR a template — class parameterized by a type.

```
template<typename T> class Stack {
    int size,cap;
    T *contents;
 public:
    ...
    void push(T x) {...}
    T top() {...}
    void pop() {...}
};

template<typename T> class List {
    struct Node {
        T data;
        Node *next;
    };
    Node *theList;
 public:
    class Iterator {
        Node *p;
        ...
    public:
        ...
        T&operator*(){...};
    };
    T &ith(int i) {...}
    void addToFront(T n);
};
```

Lecture 16

Recall:

```
template <typename T> class List {
    struct Node {
        T data;
        Node *next;
    };
    public:
    ....
}
```

client

```
List <int> l1;
// Can also do
List <List<int>> l2;
l1.addtoFront(3);
l2.addtoFront(l1);

for (List<int>::iterator it = l1.begin(); it != l1.end(); ++it) {
    cout << *it << endl;
}
```

or indeed

```
for (auto n: l1) {
    cout << n << endl;
}
```

Compiler specializes templates at the source code level, and then compiles the specializations.

The Standard Template Library (STL)

Large # of useful templates

Eg dynamic-length arrays: vectors

```
import <vector>;
std::vector<int> v{4,5}; // 4 5
v.emplace_back(6); // 4 5 6
v.emplace_back(7); // 4 5 6 7
```

But also:

```
std::vector w{4,5,6,7}; // Note: no int
```

If the type argument of a template can be deduced from its initialization, you can leave it out. <int> is deduced here.

To get an array of 4 5's, we need to use round brackets.

```
vector <int> v(4,5); // 5 5 5 5
vector <int> v{4,5}; // 4 5
```

Looping over vectors

```
for (int i = 0; i < v.size(); ++i) {
    cout << v[i] << endl;
}
//or
for (vec<int>::iterator it = v.begin(); it != v.end; ++it) {
    cout << v[i] << endl;
}
// vec<int>::iterator is auto

for (auto n : v) {
    cout << n << endl;
}
// To iterate in reverse:

for (vector<int>::reverse_iterator it = v.rbegin(); it != v.rend(); ++it) {
    ...
}
// vector<int>::reverse_iterator is auto

// Cannot use range based loop because implicitly calls normal begins

// rbegin points to last item (!= end)
// rend points before the first item (!= begin)
```

```
v.pop-back() --- remove last element
```

Use iterators to remove items from inside a vector: Ex: Remove all 5's from the vector v.

Attempt #1:

```
for (auto it = v.begin(); it != v.end(); ++i) {
    if (*it == 5) v.erase(it);
}
```

Why is this wrong? Consider

```
// vector of 1,5,5,2
v.erase(it)
// it is still pointing to second slot, and now second 5 is in that slot
// incrementing to the next (++it)
// missed the second 5
```

Note: After erase, it points at a different item. The rule is: after an insertion or erase, all iterators pointing after the point of insertion/erasure are considered invalid and must be refreshed.

Correct:

```
for (auto it = v.begin(); it != v.end();) {
    if (*it == 5) it = v.erase(it);
    // returns iterator to the point of erasure
    else ++it;
}
```

Design Patterns Continued

Guiding principle: program to the interface, not the implementation.

- Abstract base classes define the interface
  - work with base class pointers & call their methods
- Concrete subclasses can be swapped in & out
  - abstraction over a variety of behaviours

Eg Iterator pattern.

```
class List {
    ...
    public:
     class Iterator: public AbstractIterator {
     ...
     };

};

class AbstractIterator {
    public:
        virtual int &operator*() = 0;
        virtual AbstractIterator &operator++() = 0;
        virtual operator != (const AbstractIterator &other) = 0;
};


class Set {
    ...
    public:
        class Iterator: public AbstractIterator {
            ...
        };
};
```

Then you can write code that operates over iterators.

```
void for_each(AbstractIterator &start, AbstractIterator &finish, void (*f)(int)) {
while(start != finish) {
    f(*start);
    ++start;
}
// to f to all the things
}
```

Works over lists and sets.

Observer Pattern

Publish — subscribe model

One class: publisher/subject — generates data. One or more subscriber/observer classes — receive data and react to it.

Eg subject = spreadsheet cells, observers = graphs

- when cells change, graphs update

Can be many different kinds of observer objects — subject should not have to know all the details.

Observer Pattern

[figure 7]

Sequence of method calls:

1. Subject's state is updated.
2. Subject::notifyObservers() — calls each observer's notify
3. Each observer calls ConcreteSubject::getState to query the state & reacts accordingly

Example: Horse races

Subject — publishes winners

Observers — individual bettors — declare victory when their horse wins.

Subject class

```cpp
class Subject {
    vector<Observer*> observers;
    public:
        void attatch(Observer *ob) {
            observers.emplace_back(ob);
        }
        void detach(Observer *ob) {
            // remove from observers
            // exercise (in lec folder)
        }
        void notifyObservers() {
            for (auto ob: observers) {
                ob->notify();
            }
        }
        virtual ~Subject() = 0;
};

Subject::~Subject() {}
```

Observer class

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

```cpp
class HorseRace: public Subject {
    ifstream in; // source of data
    string lastWinner; // state
 public:
    HorseRace(string fname): in{fname} {}
    bool runRace()  {
        in >> lastWinner;
        return !in.fail();
    }
    string getState() {
        return lastWinner;
    }
    // concrete subject (bottom left on fig 7)
}

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

main:

```cpp
HorseRace hr {"name.txt"};
Bettor Larry (&hr, "Larry", "RunsLikeACow");
// ... (other bettors)
while(hr.runRace()) {
    hr.notifyObservers();
}
```

Lecture 17

Decorator Pattern

Want to enhance an object at runtime — add functionality/features.

Eg Windowing system

- start with a basic window
- add scrollbar
- add menu

Want to choose these enhancements at runtime.

[figure 8]

Component — defines the interface — operations your objects will provide

Concrete Component — implements the interface

Decorators — all inherit from Decorator, which inherits from Component.

Therefore every Decorator is a Component and every Decorator HAS a Component.

Eg Window w/scrollbar is a kind of window and has a pointer to the underlying plain window.

Window w/scrollbar and menu is a window, has a pointer to the window w/scrollbar, which has a pointer to window. (similar to linked list).

All inherit from Abstract Window, so Window methods can be use polymorphically on all of them.

Eg Pizza.

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

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

```
class Decorator: public Pizza {
 protected:
    Pizza *component;
 public:
    Decorator(Pizza *p): component {p}{}
    virtual ~Decorator() {delete component;}
};
```

```
class StuffedCrust: public Decorator {
 public:
    StuffedCrust (Pizza *p): Decorator{p} {}
    float price() const override {
        return component->price() + 2.69;
    }
    string desc() const override {
        return component->desc() += " with stuffed crust";
    }
};

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

Client Code.

User:

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

```
v[i]
// ith element of v
// unchecked -- if you go out of bounds, UB
v.at(i)
// checked version of v[i]
```

What happens when you go out of bounds? What should happen?

Problem Vector's code can detect the error, but doesn't know what to do about it. Client can respond, but can't detect the error.

C Solution: Functions return a status code or sets the global variable errno. Leads to awkward programming. Encourages programmers to ignore error-checks.

Exceptions

C++ — when an error condition arises, the function raises an exception. What happens? By default, execution stops.

But we can write handlers to catch errors & deal with them.

vector<T>::at throws an exception of type std::out.of.range when it fails. We can handle it as follows.

```
import <stdexcept>;
...
try {
    cout << v.at(10000); // stmts that may throw go in a try block
}
catch (out.of.range r) {
cerr << "Range error." << r.what() << endl;
}
```

Now consider:

```
void f() {
    throw out.of.range{"f"};
    // what .what() will say.
}
void g() {f();}
void h() {g();}
int main () {
    try{
        h();
    }
    catch(out_of_range) {
        ...
    }
}
```

What happens? Main calls h, h calls g, g calls f, f throws out-of-range.

Control goes back through the call chain (unwinding the stack) until a handler is found.

All the way back to main, main handles the exception.

No matching handler in the entire call chain → program terminates.

A handler might do part of the recovery job, i.e execute some corrective code.

Lecture 18

Recall: An exception can do part of the recovery job, throw another exception.

```
try {---}
catch(SomeError s) {
    ...
    throw SomeOtherError {};
}
```

Or rethrow the same exception.

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

```
// throw; vs throw s;

throw s;
// s may be a subtype fo SomeError
// throws a new exception of type SomeError
//[SomeError]
//↑
//[SpecialError]
throw;
// actual type of s is retained.
```

A handler can act as a catch-all.

```
try{---}
catch(...) {  // actually ...
// don't worry about the type of exception
---

}
```

You can throw anything you want — don't have to be objects.

When new fails: throws std::bad_alloc. Never let a destructor throw or propagate an exception.

- Program will abort immediately
- If you want a throwing destructor, you can tag it with noexcept(false).

But — if a destructor throws during stack unwinding while dealing with another exception, you know have two active, unhandled exceptions, & the program will abort immediately.

Much more to come.

Factory Method Pattern

Write a video game with 2 kinds of enemies: turtles & bullets.

- System randomly sends turtles & bullets. Bullets more common in harder levels.

(factory.jpeg here)

- Never know exactly which enemy comes next, so can't call turtle/bullet directly.
- Instead, put a factory method in Level that creates enemies.
  - Method that "creates things"

```
class Level {
 public:
    virtual Enemy *createEnemy() = 0;
    // this is a factory method
}

class Easy: public Level {
 public:
    Enemy *createEnemy() override {
    // create mostly turtles
    }
}

class Hard: public Level {
 public:
    Enemy *createEnemy() override {
    // create mostly bullets
    }
}

Level *l = new Easy;
Enemy *c = l->createEnemy();
```

Template Method Pattern

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

Eg

There are red turtles & green turtles.

```
class Turtle {
 public:
    void draw() {
        drawHead();
        drawShell();
        drawFeet();
    }
 private:
    void drawHead() {...}
    void drawFeet(){...}
    virtual void drawShell() = 0;
    // drawShell is virtual
    // it is only one that can be overriden
}
class RedTurtle: public Turtle {
    void drawShell() override {
        // draw red shell
    }
}
class GreenTurtle: public Turtle {
    void drawShell() override {
        // draw green shell
    }
}
```

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

Generalization: the Non-Virtual Interface (NVI) idiom.

- A public virtual method is really two things:
  - public:
    - an interface to the client

    * promises certain behaviour with pre/post conditions.
  - virtual:
    * an interface to subclasses
    * behaviour can be replaced with anything the subclass wants

Public & virtual → making promises you can't keep!

NVI says:

- All public methods should be non-virtual.
- All virtual methods should be private or protected
- except the destructor

```
class DigitalMedia {
 public:
    virtual void play()=0;
};
// public virtual method (no NVI)
class DigitalMedia {
 public:
    void play() {
    // can insert code here that checks first
    // i.e checkCopyright()
    // doPlay could be put in an if statement
        doPlay();
    // can also make things happen after
    // i.e update playcount
    }
 private:
    virtual void doPlay() = 0;
}
// public method that calls private virtual method
```

Generalizes Template Method

- every virtual method should be called from within a template method.

STL Maps — for creating dictionaries

Eg "arrays" that map strings to integers

```
import <map>
std::map<std::string,int> m;
m["abc"] = 2;
// maps abc to 2
m["def"] = 3;
cout << m["ghi"] << m["def"];

// 0 , 3
// m[ghi] gets inserted (because not present)
// value gets default constructed
// for ints that is 0
```

Lecture 19

Recall:

```cpp
map<string,int>m;
m["abc"]=2;
m["def"]=3;
cout << m["ghi"]; // 0
cout << m["def"]; // 3
m.erase("abc");
if (m.count("def")) // 0 = not found, 1 = found
```

Iterating over a map → sorted key order.

```cpp
for (auto &p : m) {
    cout << p.first << ' ' << p.second << endl;
    //      key                value
    //first and second are fields
}
```

p's type is std::pair<string,int> (<utility>)

std::pair is implemented as a struct, not as a class. Fields are public.

In general: using 'class' implies you are making an abstraction, with invariants that must be maintained.

Struct signals that this is purely a conglomeration of values, no invariants, all field values valid.

Alternatively:

```cpp
for (auto &[key,value]:m) {
    //^ called a structured binding
    cout << key << ' ' << value << endl;
}
```

Structured bindings can be used on any structure(class) type with all fields public:

Eg

```cpp
Vec v {1,2};
auto [x,y]=v; // x = 1, y = 2
```

Or on a stack array of known size:

```cpp
int a[] = {1,2,3};
auto [x,y,z] = a; // x=1, y=2,z=3
```

What should go into a module?

So far — each class gets its own module.

But a module can contain any # of classes & functions.

When should classes/functions be grouped together in a module and when should they be in separate modules?

Two measures of design quality — coupling & cohesion.

Coupling — how much distinct program modules depend on each other.

- low:
  - modules communicate via function calls with basic parameters and results
  - modules pass arrays/structs back & forth
  - modules affect each other's control flow
  - modules share global data
- high:
  - modules have access to each other's implementation (friends)

High coupling → changes to one module require greater changes to other modules. Harder to reuse individual modules.

Cohesion — how closely elements of a module are related to each other.

- low:
  - arbitrary grouping of unrelated elements (e.g. <utility>)
  - elements share a common theme, otherwise unrelated, maybe some common base code (e.g. <algorithm>)
  - elements manipulate state over the lifetime of an object (e.g. open/read/close files)
  - elements pass data to each other
- high:
  - elements cooperate to perform exactly one task

Low cohesion → poorly organized code — can't reuse one part without getting other stuff bundled with it — hard to understand, maintain.

Goal: low coupling, high cohesion.

Special case: What if 2 classes depend on each other?

```
class A {
    int x;
    B y;
};
class B {
    char x;
    A u;
};
```

Impossible. How big would A & B be?

But

```
class B; // forward-declare B
class A {
    int x;
    B *y;
    // compiler doesn't know what B is yet
};
class B {
    char x;
    A xy;
};
```

Sometimes one class must come before the other.

Eg

```
class C {...};
class D: public C {...};
class E {C a};
```

Need to know the size of C to construct D & E. C must come first.

Q: How should A & B be placed into modules?

A: Modules must be compiled in dependency order. One module can't forward declare another module, nor any item within that module. Therefore, A & B must be in the same module.

(Makes sense, since A & B are obviously tightly coupled).

Decoupling the Interface (MVC)

Your primary program classes should not be printing things.

E.g

```
class ChessBoard {
    ...
    cout << "Your move"
    ...
}
```

Bad design — inhibits code reuse.

What if you want to reuse ChessBoard, but not have it communicate via cout?

One solution: parameterize the class by a stream:

```
class Chessboard {
    istream &in;
    ostream &out;
    public:
        ChessBoard(istream &in, ostream &out): in{in}, out{out} {}
        out << "Your Move";
}
```

Still suffers from the same problem. Better — but what if we don't want to use streams at all?

Your chessboard class should not be communicating with users at all.

Single Responsibility Principle: "A class should have only one reason to change"

I.e if ≥ 2 distinct prats of the problem specification affect the same class , then the class is doing too much.

Each class should do only one job — game state & communication are two jobs.

Better:

- Communicate with ChessBoard via params/results/exns.
- Confine user communication to outside the game class

Q: Should main do the talking?

A: No. Hard to reuse or replace code if it is in main

Should have a class to manage communication, that is separate from the game state class.

Architecture: Model-View-Controller (MVC)

Separate the distinct notions of the data (or state — "model") the presentation of data ("view") and control or manipulation of the data ("controller").

MVC image here

Model:

- Can have multiple views (e.g. Text & graphics)
- Doesn't need to know their details
- Classic Observer pattern (or could communicate through the controller)

Lecture 20

Recall: MVC

Controller:

- Mediates control flow through model & view
- May encapsulate turn-taking, or full game rules
- May communicate with the user for input (or this could be the view)

Exception Safety

Consider:

```
void f() {
    C c;
    C *p = new C;
    g();
    delete p;
}
```

No leaks — but what if g throws?

What is guaranteed?

- During stack-unwinding all stack-allocated data is cleaned up — destructors run, memory is reclaimed
- Heap-allocated memory is not reclaimed

Therefore, if g throws, C is not leaked, but *p is.

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

Error-Prone duplication of code. How else can we guarantee that something (eg delete p) will happen, no matter how we exit f? (normal or by exn)?

In some languages — "finally" clauses guarantee certain final actions — not in C++. Only thing you can count in in C++ — destructors for stack-allocated data will run. Therefore use stack-allocated data with destructors as much as possible. Use the guarantee to your advantage.

C++ Idiom: RAII — Resource Acquisition Is Initialization

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

Eg Files

```
{
ifstream f{"file"};
}
```

Acquiring the resource ("file") = initializing the object(f)

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

This can be done with dynamic memory.

```
//(import <memory>)
class std::unique-pointer_ptr<T>
```

- Takes a T* in the constructor
- The destructor will delete the pointer
- In between — can dereference, just like a pointer

```
void f() {
    C c;
    std::unique_ptr<c> p {new C};
    g();
}
```

No leaks guaranteed

Alternative

```
void f() {
    C c;
    auto p = std::make_unique<c>();
    //                              ctor args go here, if any
    g();
}
```

Allocates a C object on the heap, and puts a pointer to it inside a unique pointer to object.

Difficulty:

```cpp
unique_ptr<c> p {new C};
unique_ptr<c>q = p;
```

What would happen if a unique_ptr were copied?

Don't want to delete the same pointer twice.

Instead — copying is disabled for unique_ptrs. They can only be moved.

```cpp
template<typename T> class unique_ptr {
    T *ptr;
 public:
    exlpicit unique_ptr(T *p): ptr{p} {}
    ~unique_ptr() {delete ptr};
    unique_ptr(const unique_ptr &other) = delete;
    unique_ptr<T> &operator(const unique_ptr &) = delete;
    unique_ptr(unique_ptr &&other): ptr {other.ptr} {
        other.ptr = nullptr;
    }
    unique_ptr<T> &operator=(unique_ptr &&other) {
        if (this == &other) return *this;
        delete ptr;
        ptr=other.ptr;
        other.ptr = nullptr;
        return *this;
    }
    T &operator*() {return *ptr;}
    T *get() {return ptr;}
};
```

If you need to be able to copy pointers — first answer the question of ownership. Who will own the resource? who will have responsibility for freeing it?

- That pointer should be a unique_ptr. All other pointers should be raw pointers (can fetch the underlying raw pointer with p.get()).

New understanding of pointers:

- unique_ptr — indicates ownership — delete will happen automatically when the unique_pointers go out of scope.
- raw pointer — indicates non-ownership. Since a raw pointer is considered not to own the resource it points at and you should not delete it.

Moving a unique_ptr = transfer of ownership.

Pointers as parameters

```cpp
void f(unique_ptr<c> p);
// f will take ownership of the object pointered to by p
// caller loses custody of the object
void g(C *p)
//g will not take over ownership of the object pointed to by p
// caller's ownership of the object does not change

```

```
//Note that the caller might not own the object
```

Pointers as results:

```
unique_ptr<c> f();
```

Return by value is always a move, so f is handing over ownership of the C object to the caller.

```
C *g();
```

The pointer returned by g is understood not to be deleted by the caller, so it might represent a pointer to non-heap data, or to heap data that someone else already owns.

Rarely, a situation may arise that calls for true shared ownership, i.e. any of several pointers might need to free the resource.

- use std::shared_ptr

```
{
auto p = std::make_shared<c> ();
if (---) {
    auto q =p;
} // q popped, pointer not deleted

} // p is popped, pointer is deleted
```

Shared pointers maintain a reference-count of all shared_ptrs pointing at the same object.

Memory is freed when the # of shared_ptrs pointing to it will reach 0.

Recall (Racket)

```
(define l1 (cons 1 (cons 2 (cons 3 empty))))
(define l2 (cons 4 (rest l1)))
```

rest of l2 points to second element of l1

Lecture 21

Exception safety... What is exception safety?

It is not

- exceptions never happen
- all exceptions get handled

It is

- after an exception has been handled, the program is not left in a broken or unusable state

Specifically, 3 levels of exception safety for a function f:

1. Basic guarantee — if an exception occurs, the program will be in some valid state. Nothing is leaked no corrupted data structures, all class invariants maintained.
2. Strong guarantee — if an exception is raised while executing f, the state of the program will be as if f had not been called.
3. No-throw guarantee — f will never throw or propagate an exception and will always accomplish its task.

Eg

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

Is C::f exception safe?

- If a.g() throws — nothing has happened yet. OK.
- If b.h() throws — effects of g would have to be undone to offer the strong guarantee
    - very hard or impossible if g has non-local side-effects

No, probably not exception safe.

If A::g and B::h do not have non-local side effects, can use copy & swap.

```
class C {
    ...
    void f() {
        A atemp = a;
        B btemp = b;
        atemp.g();
        btemp.h();
        //if any of the above throw, the original a and b are still intact
        a = atemp;
        b = btemp;
        //What if copy assignment throws?
        //In particular, what if a=atemp succeeds
        //but b = btemp fails.
    }
}
```

Better if swap was no-throw. Recall : copying pointers can't throw.

Solution: Access C's internal state through a pointer (called the pimpl idiom).

```
struct CImpl {
    A a;
    B b;
};

class C {
    unique_ptr<CImpl> pImpl;
    void f() {
        auto tmp = make_unique<CImpl>(*pImpl);
        tmp->a.g();
        tmp->b.h();
        std::swap(pImpl, tmp); // no-throw
    }
};
```

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

Exception Safety & the STL — Vectors

Vectors — encapsulate a heap-allocated array

- follow RAII — when a stack-allocated vector goes out of scope, the internal heap array is freed.

```
void f() {
    vector<c> v;
    ...
} // V goes out of scope, array is freed, C dtor runs
// on all objects in the vector
```

But

```
void g() {
    vector<c*> v;
    ...
} // array is freed, ptrs don't have dtors, and objects
// ptd to by the ptrs are not deleted.
// v is not considered to own these objects.
```

But

```
void h() {
    vector<unique_ptr<c>>v;
    ...
} // array is freed, unique_ptr dtors run
// the objects are deleted
// no specific deallocation
```

```
vector<c> // owns the object
vector<c*> // does not own the object
vector<unique_ptr<c>> // owns the object
```

```
vector<T>::emplace_back // offers the strong guarantee
```

- If the array is full (i.e size == cap)
  - allocate new array
  - copy objects over (copy ctor)
    - if a copy ctor throws*
      - destroy the new array

        · old array still intact
        · strong guarantee
    – delete old array (no throw)

*But — copying is expensive & the old array will be thrown away. Wouldn't moving the objects be more efficient?

- allocate the new array
- move the objects over (move ctor)
    - if move constructor throws
        * original is no longer intact
        * can't offer the strong guarantee
- delete the old array (no-throw)

If the move constructor offers the no-throw guarantee, emplace_back will use the move constructor. Otherwise it will use the copy constructor, which may be slower.

So your move operations should offer the no-throw guarantee, and you should indicate that they do:

```
class C {
 public:
    C (C &&other) noexcept {...}
    C &operator=(C &&other) noexcept{...}
};
```

If you know a function will never throw or propagate an exception, declare it noexcept. Facilitates optimization.

At minimum: moves & swaps should be noexcept.

Casting

In C:

```
Node n;
int *ip = (int *)&n; //cast - forces C++ to treat
// a Node * as an int*
```

C-style casts should be avoided in C++.

If you must cast, use a C++ style cast:

4 kinds

1. Static_cast — "sensible casts" — casts with a well-defined semantics.

Eg double → int

```
double d;
void f(int x);
void f(double d);
f(static_cast<int>(d));
```

superclass ptr → subclass ptr

```cpp
Book *b = new Text {...};
Text *t = static_cast<Text*>(b);
```

Lecture 22

Recall: Superclass → subclass pointer

```cpp
Book *b = new Text{...};
...
Text *t = static_cast<Text*>(b);
```

Taking responsibility that b actually points to a Text. "Trust me".

1. Static_cast — "sensible casts" — casts with a well-defined semantics.
2. Reinterpret_cast — unsafe, implementation-specific, "weird" casts

   ```cpp
   Student s;
   Turtle *t = reinterpret_cast<Turtle*>(&s);
   ```

3. Const_cast — for converting between const & non-const — the only C++ cast that can "cast away const".

```cpp
void g(int *p);
// suppose we know that under the circumstances in which f operates
// g won't modify *p

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

1. Static_cast — "sensible casts" — casts with a well-defined semantics.
2. Reinterpret_cast — unsafe, implementation-specific, "weird" casts.
3. Const_cast — for converting between const & non-const — the only C++ cast that can "cast away const".
4. Dynamic_cast — Is it safe to convert a Book * to a Text *?

```cpp
Book *pb ___;
static_cast<Text *>(pb) // safe?
```

Depends on what pb actually points to. Better to do a tentative cast — try it & see if it succeeds.

```cpp
Text *pt = dynamic_cast<Text*>(pb);
```

If the cast works (*pb really is a Text, or a subclass of Text), pt points to the object.

If not — pt will be nullptr.

```cpp
if (pt) cout << pt->getTopic();
else cout << "Not a text.";
```

These are options on raw pointers. Can we do them with smart pointers?

Yes — static_pointer_cast, etc. Cast shared_ptrs to shared_ptrs.

Dynamic_casting also works with refs:

```
Text t {___};
Book &b = t;
Text &t2 = dynamic_cast<Text &>(b);
```

If b "points to" a Text, then t2 is a reference to the same Text.

If not ... ? (No such thing as a null reference). Throws std::bad_cast.

Note: Dynamic casting only works on classes with at least one virtual method.

Dynamic reference casting offers a possible solution to the polymorphic assignment problem:

```
Text &Text::operator=(const Book &other) { // virtual
    const Text &tother = dynamic_cast<const Text&>(other);
    // throws if other is not a Text
    Book::operator=(other);
    topic = tother->topic;
    return *this;
}
```

Is dynamic casting good style?

Can use dynamic casting to make decisions based on an object's runtime type information (RTTI)

```
void whatIsIt(Book *b) {
if (dynamic_cast<Comic*>(b)) cout << "Comic";
else if (dynamic_cast<Text*>(b)) cout << "Text";
else if (b) cout << "Book";
else cout << "Nothing";
}
```

Code like this is tightly coupled to the Book hierarchy, and may indicate bad design.

Why? What if you create a new kind of Book?

- WhatIsIt doesn't work anymore until you add a clause for the book type.
- Must do this wherever you are dynamic casting

Better: use virtual methods

Note: Text::operator= does not have this problem (only need to compare with your own type, not all the types in the hierarchy).

So dynamic casting isn't always bad design.

How can we fix whatIsIt?

```
class Book {
    ...
    virtual void identify() { cout << "Book";}
};
void whatIsIt(Book *b) {
    if (b) b->identify();
    else cout << "nothing";
}
```

Works by having an interface function that is uniform across all Book types. What if the interface isn't uniform across the hierarchy?

Inheritance & virtual methods work well when

- There is an unlimited number of potential specializations of a basic abstraction.
- Each following the same interface

But what about the opposite case

- Small number of specializations, all known in advance, unlikely to change
- With different interfaces

In the first case — new subclass → no effort at all

In the second case — new subclass → rework existing code to accommodate the new interface, but that's fine because you are not expecting to add new subclasses . . . or you are expecting to put in that effort.

Eg

```
class Turtle: publicEnemy {
    void stealShell();
};
class Bullet publicEnemy {
    void deflect();
}
```

Interfaces not uniform. A new enemy type is going to mean a new interface & unavoidable work. So we could regard the set of Enemy classes as fixed, and maybe dynamic casting is justified.

But: in this case, maybe inheritance is the wrong tool. If you know that the enemy will only be a Turtle or Bullet, and you accept the work that comes with adding new Enemy types, then consider:

```
import <variant>;
typedef variant <Turtle,Bullet> Enemy;
// equiv:
using Enemy = variant <Turtle,Bullet>
// An Enemy is a Turtle or a Bullet. Period.
Enemy e {Turtle{}}; // or bullet
if (holds.alternative<Turtle>(e) {
    cout << "Turtle";
} else ...
```

Lecture 23

Recall:

```
using Enemy = variant<Turtle,Bullet>;
Enemy e {Turtle{}};
// Discriminating the value
if (holds_alternative<Turtle>(e)) {
    cout << "Turtle";
} else ...
```

```
// Executing the value:
try {
    Turtle t = get<Turtle>(e);
    //use t...
}
catch (bad_variant_access f) {
    // it's not a Turtle...
}
```

A variant is like a union but type-safe. Attempting to store as one type & fetch as another will throw.

If a variant is left uninitialized, the first option in the variant is default-constructed to initialize the variant.

Compiler error if first option is not default-constructible.

Options:

1. Make the first option a type that has a default ctor.
2. Don't leave your variant uninitialized.
3. Use std::monostate as the first option. "Dummy" type that can be used as default.
    (a) Can be used to create an "optional" type: e.g. variant<monostate,T> = "T or nothing". (Also: std::optional<T>).

How Virtual Methods Work

```
class Vec {
    int x,y;
    void f() {...}
}
class Vec2 {
    int x,y;
    virtual void f() {...}
}
Vec v{1,2};
Vec2 w{1,2}; // Do these two look the same in memory

cout << sizeof(v) << ' ' << sizeof(w);
// 8  16 ???
```

First note:

- 8 is space for 2 integers. No space for f method
- Compiler turns methods into ordinary functions & separates them.

Recall:

```
Book *pb = new{Comic}; // among book, text, comic
auto pb = make_unique<Comic>(...) // per above choice
```

isHeavy is virtual → choice of which version to run is based on the type of the actual object — which the compiler won't know in advance.

∴ choice must be made at runtime. How?

For each class with virtual methods, the compiler creates a table of function pointers (the vtable).

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

[figure 9]

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

[figure 10]

[figure 11]

Calling a virtual method:

- follow vptr to vtable
- fetch ptr to actual method from table
- follow the function pointer & call the function
- (all of the above happens at run-time)

∴ virtual function calls incur a small overhead cost in time.

Also: Having >= 1 function adds a vptr to the object. ∴ classes with no virtual functions produce smaller obs than if some were virtual — space cost.

Concretely, how is an object laid out? Compiler-dependent. Why did we put the vptr first in the object and not somewhere else (e.g last)?

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

[figure 12]

But...

Multiple Inheritance

A class can inherit from more than one class.

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

[figure 13]

Challenges: Suppose — [figure 14]

B & C inherit from A.

```
class D: public B, public C {
    public:
        int d;
};
D dobj;
dobj.a // which a is this? Amibiguous
// compiler error
```

Need to specify dobj.B::a or dobj.C::a.

But if B & C inherit from A, should there by one A part of D or two? (Two is the default).

Should B::a, C::a be the same or different?

What if we want [figure 15]

("deadly diamond")

Make A a virtual base class — virtual

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

E.g. IO stream hierarchy See visual 16.

How would this be laid out?

[Figure 17] Clear that [figure 17] is wrong.

[Figure 18] — distance from class to parent is not constant. It depends on the runtime type of object.

Solution: Distance to the parent object is stored in the vtable.

Diagram still doesn't look like all of A,B,C,D simultaneously. But slices of it do look like A,B,C,D.

∴ pointer assignment among A,B,C,D changes the address stored in the pointer.

```
D *d = new D;
A *a = d;
// this changes the address (adds the distance)
```

Static/dynamic cast will also do this, reinterpret_cast will not.

Lecture 24

```cpp
template <typename T> T min(Tx, Ty) {
    return x<y? x : y;
}

int f() {
    int x = 1, y = 2;
    int z = min(x,y); // C++ infers T = int from types
    // of x and y
}

```

If C++ can't determine T, you can tell it.

```cpp
int z = min<int>(x,y);

min('a','c') // T = char
min(1.0, 3,0); // T = double
```

For what types T can min be used? For what types T does the body compile?

Any type for which operator < is defined.

Cutoff for Exam (I stopped paying attention here, something about algorithms)