CS 246 - Spring 2023 - Course Notes

Author: Brandon Zhou

Instructor: Mark Petrick

Room: MC 2034

Days & Times: TTh 1:00PM - 2:20PM

Section: 003

Date Created: May 6, 2023

Final Exam Date: August 16, 2023

Disclamer: Please note that portions of these notes have been sourced from ChatGPT, an OpenAI product. While I've made every effort to ensure the accuracy of the content, there's potential for errors or outdated information. It's important to approach these notes as a supplementary reference and not as a primary source. Should you come across any uncertainties or ambiguities in the material, I strongly recommend consulting with your course instructors or the course staff for a clearer understanding. I apologize in advance for any potential discrepancies or oversights. Additionally, any changes made to this notebook after its initial creation are not endorsed or recognized by me. Always cross-reference with trusted resources when in doubt.

Table of Contents

Lecture 1

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

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

Language of delivery: C++20

Intro to C++

- Hello world in C:

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

- Hello world in C++:

```
import <iostream>;
using namespace std;
int main() {
        cout << "Hello world!" << endl;
        return 0;
}
```

Note:

- `main` MUST return type `int`
- `void main(...)` is not legal in C++
- return statement - returns status code to OS
- can be omitted from `main` (0 assumed)
- `stdio`, `printf` are still available in C++.
- preferred C++ I/O: header

```
std::cout << ___ << ___ << ___;
```

- where ___ is data. `std:endl` = end of file + flush output buffer
- `using namespace std;` lets you omit the `std::` prefix
- Most C programs work in C++ (or require minimal changes)

Input/Output

- There are 3 I/O streams:
    - `cout/cerr` for printing to stdout/stderr
    - `cin` for reading from stdin
- There are 2 I/O operators:
    - << "put to" (output)
    - >> "get from" (input)

```
cerr << x
cin >> x
```

operator "points" in the direction of information flow

- E.g. Add 2 #'s

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

- the first two lines will omit these from now on for brevity

Note:

- By default, `cin >>` ignores leading whitespace/space/tab/newline
- What if bad things happen?
    - input does not contain an integer next
    - input too large/small to fit in the var
    - run out of input before we get 2 ints (EOF)
- If the read failed: `cin.fail()` will be true. If EOF, `cin.fail()` and `cin.eof()` will both be true. But not until the attempted read fails.

E.g. Read all ints from stdin of echo item, 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

- Let `cin` be used as a condition
- `cin` converts to true, unless the stream has a read failure
- E.g. V2.0

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

Note:

- >> is C's right bitshift operator and C++'s
- From ChatGPT: you cannot copy objects of types derived from `std::istream` (including `std::istream` itself). This is because the copy constructor and copy assignment operator for `std::istream` (and `std::ostream`) are deleted functions, meaning they are explicitly disabled. This is done because copying such objects doesn't make sense conceptually and could lead to various problems.
    - Thus, you cannot declare a field as `std::istream` in a struct (or class) because `std::istream` is an abstract base class. It has at least one pure virtual function, which means you cannot create an instance of it.
    - If you need a stream as a member of your struct or class, you could use a pointer or a reference to `std::istream`. This way, you could point or refer to an instance of a concrete subclass such as `std::istringstream` or `std::ifstream`.
    - In C++, `std::cin` is an object of type `std::istream` that represents the standard input stream. It's a global object, and it's an lvalue. When you pass `std::cin` to the `MyStruct` constructor below, you're passing it as an lvalue reference, not as an rvalue.

```
import <iostream>;

struct MyStruct {
    std::istream& in;  // reference to a std::istream
    MyStruct(std::istream& input) : in(input) {}
};

int main() {
    MyStruct s(std::cin);  // Now MyStruct::in refers to std::cin
    return 0;
}
```

Lecture 2

- ReadEx5: read all ints, skip non-integer inputs

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

- There are global flags that store var, cns states such as state of a stream fail/okay
- `cout << 95 << endl;` by default, it prints as a decimal. What if you want it printed as hex?
    - use a manipulator `cout << hex << 95 << endl;`, `cout << dec << 95 << endl;`
- Advice: don't forget to change the stream back to how you found it.

Strings

- C: array of char ending with a null terminator `\o`, must explicitly manage memory
    - Allocate more as string gets longer
    - easy to overwrite `\o` so corrupt memory
- C++: `import <string>;`
    - type: std::string
    - ignore as needed (no need to manage memory)
    - safer to manager
    - `string s = "Hello"`, C++ string created from C string on initialization
- String operations:
    - length `s.length(); // O(1)`
    - individual character `s[0], s[1]`

    - equality/inequality `s1==s2, s1!=s2`
    - comparison `s1<=s2` (lexicographic)
    - concat `s3=s1+s2, s3+=s4`
- From ChatGPT about the operator + for `std::string`:
    - You cannot directly use the + operator on two C-style string literals (i.e., pointers to characters) because the + operation is not defined for pointer addition in this context.
    - You can use the + operator to concatenate two `std::string` objects.
    - You can also use the + operator to concatenate an `std::string` object with a C-style string literal, regardless of the order.
    - The primary point is that at least one of the operands must be of type `std::string` for the + operator to work as string concatenation. This is because the + operator is overloaded for the `std::string` type to perform string concatenation.

```
string s;
cin >> s; // read a string; skip leading whitespace; stop at whitespace 'read one word'
cout << s << endl;
```

- want to include white space? Use getline(cin, s)
- Streams are abstraction
- Files: read/write from/to a file instead of stdin/stdout
    - `std::ifstream`
    - `std::ofstream`
- File access in C:

```
#include <stdio.h>
int main() {
        char s[256];
        FILE *f = fopen("File.txt", "r")
        while (1) {
                fscanf("%255s", s); // you only have 256 char available, one for null terminator
                if (feof(f)) break;
                printf(""%s\n)
        }
        fclose(f)
}
```

- File access in C++:

```
import <iostream>;
import <string>;
import <fstream>
using namespace std;
int main() {
        ifstream f{"file.txt"}; // ("file.txt"); initialization of the ifstream use an initialzation list
        string s;
        while (f >> s) { // no concern on word length
                cout << s << endl;
        }
}
```

- Strings
    - import ;
    - extract data from chars in a string `std::istringstream`
    - send data to a string as chracrters `std::ostringstream`

```
string IntToString(int n) {
        ostringstream oss;
        oss << n;
        return oss.str();
}
```

- convert string to #

```
int n;
while (true) {
        cout << "Enter a number" << endl;
        string s;
        cin >> s;
        if (istringstream iss{s}; iss >> n) break; // first part is a variable declaration in if statement C++17;
{s} is initialization list C++11
        cout << "I said, " << n << endl;
}
```

- Ex. echo all ints, skip non-# s

```
int main() {
        string s;
```

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

Lecture 3

- If we read integers using a string type and convert it into `istringstream`, then it will use the same memory address

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

- If the input is 3.14, will print 3

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

- With above codes, if reading 'abc123', will print 123(`cin.clear()` is called to clear the failbit, and `cin.ignore()` is called to ignore the next character in the input stream. This moves the stream position to the next character. So 'a' will be ignored then read 'bc123', then ignore 'b' again then read 'c123'...untill it successfully reads an integer). However, if using the first code block above, will print nothing.
- To accept command line arguments in C or C++, always give `main` the following parameters: `int main(int argc, char *char[])`. `char *` is a pointer, eveything in the array is stored in C-style string.
- `int argc` is the number of command line arguments, always greater or equal to 1.
- `char *char[]` is an array of C-style strings. `argv[0] = program name`, `argv[1] = 1st arg`, `argv[2] = 2nd arg`.
- If C++ does not allow default arguments (here I think it means no default constructor which accpets no arguments), then **# of default arguments + 1** functions would be needed to provide equivalent funtionality for a function with default arguments. E.g. `void func(int num, char ch = 'a', int num2 = 2);` it needs 2+1=3 functions to achieve this, which are `func(int);`, `func(int, char)`, `func(int, char, int)`.
- Advice: convert into C++ strings to manipulate

```
int main(int argc, char *char[]) {
        for (int i = 0; i < argc; ++c) {
                string arg = argv[i];
                // can convert to an int, etc
        }
}
```

- `./a.out < test.txt`, `./a.out` is not a command line argument, but does get stored in the array, the program is directed to standard input. `< test.txt` is NOT command line arguments, it will not show up in the array. The number of command line arguments is 0, but it (the program name) does count in argc. `./a.out` is just the program you are executing, not considered as a command line argument, but still saved, which means argc = 1 + # of command line arguments.
- `head -n 20 < text.txt`, two command line arguments.
- *Default Function Parameters*

```
void printSuiteFile(string name = "suite.txt") {
  ifstream file{name.c_str()};
  for (string s; file >> s;) cout << s << endl;
}
```

- You can call the above function using `printSuiteFile(); // prints from suite.txt` or `printSuiteFile("my_file.txt");`

```
int f(int x = 2, int y = 3, int z = 4) {}
```

- By calling `f(7)`, it will map to the first one, which means `(f, 7, 3, 4)`.
- Default value is an interface detail
    - something the users should know

    - default parameters must be last
- What happens when you call a function:
    - allocate memory in the stack: stack frame

Stack Frame
parameters
local variables
return address

- How are the default values going to be placed in the stack frame?
    - the caller must place the default values into the stack frame
    - the function implementation assumes the first part of the stack frame are the arguments
    - if one (or more) are missing, it takes the next piece of data (local variables, etc) and interpret them as the parameter arguments.
    - caller needs to fill in any default values
- Follow these 3 steps
    - You write codes
    - Compiler will change your function call to add any data default params. `printSuiteFile();` => `printSuiteFile("suite.txt;")`.
    - During runtime, a function call can place all arguments in the stackframe.
- Overloading:
    - C: `int negInt(int n) {return -n;}` or `bool negBool(bool b) {return !b};`
    - C++: functions can reuse the same name
        - must have different parameter lists
        - the compiler choose the correct version of `neg` for each function call based on the type of arguments
    - So in C++, we can do `int neg(int n) {return -n}` and `bool neg(bool b) {return !b}`.
    - Is it enough to simply have diferrent return type? NO!!!
- So overloads must differ in # or type or args
- Structs - similar to C

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

- BE CAREFUL about the semicolon in the end of the struct definition
- Constants `const int maxGrade=100;`, it must be initialized upon creation
- Declared as many things as const as you can
    - help prevent/track errors
- `Node n {n, nullptr}` DON'T use NULL or 0
- `const Node n2 = n;`, the copy `n2` is constant, we created an unmutable copy of `n`
- Parameter Passing
    - Pass by value
- Valid Function Overloading:
    - Different Number of Parameters
    - Different Types of Parameters
    - Different Order of Parameters of Different Types
    - Different Reference Qualifiers (lvalue reference vs rvalue reference)
    - Different Qualifiers (e.g. `const int` and `int`)
- Invalid Function Overloading
    - Different Return Type Only
    - Different Order of Parameters of the Same Type
    - Different Parameter Names
- Special Cases and Considerations
    - Ambiguity due to Implicit Conversions
    - Explicit Casting or Different Literals to Resolve Ambiguity

```
void foo(int a, double b);
void foo(double a, int b);

foo(1, 1); // This is ambiguous!


foo(1, 1.0); // Not ambiguous, 1.0 is a double literal
foo(static_cast<double>(1), 1); // Not ambiguous, explicit cast
```

Lecture 4

- Pass-by-value: makes a copy of the argument and passes it into the function parameter

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

```
void inc(int n) {
  n = n + 1;
}

int main () {
  int x {5};
  inc(x);
  cout << x << endl;
}
```

- To fix this:

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

void inc(int *n) {
  *n = *n + 1;
}

int main () {
  int x {5};
  inc(&x);
  cout << x << endl;
}
```

- The above codes pass by value (C only passes by value)
- You need () when saying `(*n)++`, but not for `++*n`.
- Pass by reference

```
int y = 10;
int &z = y; // z is an lvalue reference to an int
```

- lvalue

```
int x = 10;
x++++; // this doesn't work, x is an lvalue, x++ is not an lvalue, but ++x returns itself
```

- In `if (++x)`, `++x` returns 11.
- **lvalue** is something that has a *memory address*, which means locator value.
- `x+++++y` is not valid but `x+++--y` is valid.
- Similar to `int * const z = &y`
  - this means `z` always points to `y`.
  - `z` can be used to change the value of `y`
- For `const int * z = &y`
  - `z` points to `y`, but `z` can point to something else
  - `z` cannot change the value of the thing it's pointing at (i.e. the value storing in `y`)
- *in all cases, `z` behaves exactly like `y`*
  - alias. `z` is an alias ("another name") for `y`

```
z = 12; // Not *z = 12
int *p = &z; // it stores the address of y, since z is an alias to y
```

- Why `int x; cin >> x`? why not `cin >> &x`? (In C, `scanf("%dd", &x)`)
  - `cin >> x`, `x` is an lvalue reference
- How do we know `&` means reference vs address?
  - if it's being used in the type definition, `int &p = ...` => reference.
  - when `&` occurs in an expression, it means address-of or bitwise-AND
- things you can't do with an lvalue reference
  - initialization: must have one. `int &x;` is not valid.
  - must be initialized with something that has a memory address(since refs are pointers)

```
int &x = 3; // not valid
int &x = y+z; // not valid
int &x = y; // valid
```

- CANNOT DO
  - create a pointer to reference, that is, `int &*z`
  - But you can make a reference to a pointer, the syntax is `int *&z = ...`.
  - create a reference to a reference, if the syntax exists, it would be `int &&z = ...`. This means something else but is valid, which will be rvalue reference.
  - create an array of references such that `int &r[3] = {n,n,n}`. E.g.: `r[1]` is equivalent to `&(r+1)`, but it doesn't have access to address.
- What CAN you do with reference?
  - Pass as f'n parameters

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

int main() {
        int x = 5;
        inc(x);
}
```

- Why does `cin >> x` works?
  - it takes `x` as a reference
  - `std::istream & operator >> (std:: istream &in, int &n)`, where `& operator` is return type, `istream &in` is parameter, `int &n` is a reference to int.
  - also passing `cin` into operator >>, returning as a reference.
- Why? cost at passing by value

```
int f(int n) {...}
int f(Huge h) {...}
// copy: potentially slow
int g(Huge &h) {...}
// alias: fast
// may change the one the caller has

int j(const Huge &h) {...}
// Fast, no copy, won't change caller's structure
```

- What if a function wants to change `h` locally only? so caller won't see the change.

```
int k(const Huge &h) {
        struct Huge h2 = h; //h2 is mutable
}
```

- If you need to make a copy anyway, it's better to use pass-by-value and have the compiler make the copy for you. Maybe it can be optimized.
- 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
- Consider the following two

```
int f(int &n);
f(5); // not valid, since 5 is not an lvalue


int g(const int &n);
g(5); // this is valid. Since 5 is a constant and the compiler will create a temporary memory location for 5,
then treated as an lvalue you can't change it anyway.
```

- `cin >> n` pass return `cin` as a reference, saves us from copying, not allowed to copy it

Lecture 5

- Dynamic Memory Allocation
  - CS 136: C: `int *np = malloc()` ... `free(np)`
  - Don't use malloc/free in C++(this course)
  - Use `new` and `delete`
  - type aware -> less error prone

```
struct Node {...};

int main() {
        Node *np = new Node; // np is in the stack, but Node np points to is on the heap
        delete np;
}
```

- Recall
  - all local variables reside on the stack
  - variables are deallocated when they go out of scope(stack frame is popped)
  - Dynamically allocated memory resides on heap
  - remains allocated until delete is called
  - if you don't delete all allocated memory -> memory leak
  - if your program has a memory leak, we consider it incorrect
- Arrays

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

- Memory aloocated with `new`(`new []`) should be deallocated with `delete`(`delete []`)
- Returning by value/ptr/ref

```
Node getANewNode() {
        Node n;
        return n;
}
// ok but inefficient
// main: Node n = getANewNode();

Node *getANewNode() {
        Node n;
        return &n;
}
// Bad, returning the address to a local variable, when getANewArray completes, local variable is out of scope,
but you have a pointer to it.

Node &getANewNode() {
        Node n;
        return n;
}
// Bad, similar to previous

Node *getANewNode() {
        Node *np = new Node;
        return np; // np stores the memory address, not &np
        // don't forget to delete later, return a memory address instead of the structure itself
}
```

- Why is it okay to return a reference number?
    - Because the reference is not a local variable
    - it's okay to the return reference is the same reference that was passed as the parameter in, so refers to something accessible to the caller

```
istream &operator>>(istream &in, int &n);
// it is actually doing
main:
int n;
cin >> n;
```

- Operator Overloading
    - give meaning to C++ operators for types we create

```
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;
}

int main() {
        Vec v1, v2;
        Vec v = v1 + v2;
}
```

- `const int *p`: `p` can point to different thing, but cannot change the thing `p` points to
- `int * const p`: reference acts like this; `p` points to excact same thing but you can change the thing `p` points to.

```
Vec operator*(const int k, const Vec &v) {
        return {k*v.x, k*v.y}; //k*v; -> ok/However, v*k is not okay since the type doesnt match
        // ok in C++, compiler can figure out this is a Vec from return type
}

Vec operator*(const Vec &v, const int k) {
        return k*v; // use one previously defined
}
```

- Special cases overloading >> and <<

```
struct Grade {
        int Grade;
}

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

- ChatGPT: One purpose of returning reference to the stream is to support chaining. By returning the stream, you can perform consecutive operations on the same stream without interruption. This is a common pattern in C++ I/O operations and is utilized extensively in code to read or write multiple items consecutively.

```
int x, y, z;
std::cin >> x >> y >> z;
```

- Separate compilation
    - split programs into module
    - each module provides
        - interface - type definitions, prototypes for functions
        - implementation - full details, allocate space for vars/fns; define all functions provided
- Declarations vs Definition
    - Declaration - stating existence
    - Definition - full details, allocate space

```
// interface (vec.cc) .h file in C, but .cc in C++
export module vec; // Indicates this is a module interface
export struct vec {
        int x,y;
}
export Vec operator+(const Vec &v1, const Vec &v2);

// Client(main.cc)
import vec;
int main() {
        vec v1, v2;
        vec v = v1+v2;
}

// Implementation (vec-impl.cc)
module vec; // This file is part of the vec module; Implicits imports the interface

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

- Interface files always start with: `export module __;`
- Implementation always start with: `module __;`
- You can only define once, but declare multiple times.

Lecture 6

```
// interface vec.cc
export module vec;

// implementation vec-impl.cc
module vec; // Implicitly import interface

// client main.cc
import vec;
int main() {
        vec v{1, 2};
}
```

- Dependency Order:
    - interface must be compiled first before implementation, client
- Compiling separately: `g++ -c ___.cc`
    - `-c` compile but don't link

```
g++ -c vec.cc   -> creates object files .o
g++ -c vec-impl.cc
g++ -c main.cc
g++ vec.o vec-impl.o main.o -o main
```

- short cut `g++ -c *.cc`, attempts to compile each cc file; fails if one that is dependent on has not been compiled yet; `g++ -c *.cc // call again, usually enough to compile all files`; `g++ *.o -o main`

Classes

- can put functions into structures

```
// Ex1
// student.cc
// ...
export struct student {
        int assigns, mid, final;
        float grade();
};


// student-impl.cc
float student::grade() {
        return assigns * 0.4 + mid * 0.2 + final * 0.4;
}


// client
        student s{60, 70, 80};
        cout << s.grade() << endl; // '.' is dot operator
```

- Class
    - Essentially a structure type that can contain functions
    - `class` keyword later
- object - an instance of a class

```
student s{60, 70, 80}; // studnet is a class, where s is an object
```

- function `grade()` is called a member function or method
- `::` called scope resolution operator
    - `c::f` means `f` in the context of `c`, similar to the dot operator `s.mid`; but left hand side is a class (or namespace)
- what do `assigns, mid, final` mean inside `student::grade`?
    - they are fields of the receiver object
    - from the object upon which garde was called

```
        s.grade(); // method called; use s's assigns, mid, final data
```

- Formally, methods take a hidden extra parameter called `this` - *points* to the object on which the method was called `s.grade();` what is inside `()` is `this`.
- could write

```
struct student {
        float grade {
                return this->assgns * 0.4 + this->mid * 0.2 + this->final * 0.4;
        } // can put implementations inside structure definitions
}
```

- WARNING: above code block, we do it for convenience for class and so in this note; you should define separately in an implementation file
- Initializing objects

```
student s{60, 70, 80}; // ok but limited
```

- Better write a constructor
    - a method that initializes

```
struct student {
        int assigns, mid, final;
        float grade();
        student(int assigns, int mid, int final);
};

// implementation file
student::student(int assigns, int mid, int final) {
        this->assigns = assigns;
        this->mid = mid;
        this->final = final;
}

// execution
student s{60, 70, 80};
```

- if constructor has bee defined, `60, 70, 80` are passed as arguments to the constructor
- if no constructor has been defined,
    - C-type structure initialization. field-by-field initialization

- C-style struct initialization is only available if you have not defined a constructor
- Alternate syntax: `student s = {60, 70, 80}` syntatically the same
- Heap allocation: `student *p = new student{60, 70, 80}`
- Advantage: they are functions!
    - you can write customized code to initialize the object
    - default params, overloading, sanity check

```
struct Student {
        Student(int assigns = 0; int mid = 0; int final = 0) {
                this->assigns = assigns;
                ...
        }
}


Student s{70, 80}; // 70, 80, 0
Student s2; // 0, 0, 0; same as Student s2{};
```

- whenever an object is created, a constructor is always called
- what if you didn't write one?
    - every class comes with a default (zero-argument) constructor which simply default-construct all fileds that are objects

```
struct Vec {
        int x, j;
};

Vec v; // default-constructor does nothing, the value in the fields may be garbage values;
```

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

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


Vec v; // ERROR!!! you have a construtor but no default values
Vec v{1, 5}; // OK
```

- How about

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

Basis b; // ERROR; won't compile
```

- Default-constructor for `Basis` attempts to default-constrcut all fileds that are objects
    - `v1,v2` are objects
    - But they have no default constructor
    - so `Basis` does not have a default constructor built-in

Lecture 7

- Definition: default-constructor
    - constructor with no params
    - or has params but all parameters have default values
    - built-in - supplied by compiler

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

struct Basis{
        Vec v1, v2;
        Basis () {
                v1 = Vec{1, 0};
                v2 = Vec{0, 1};
        } // IT WON'T WORK; This is step 3 based on the steps below; Body of constructor can contain arbitrary
```

```
code so fields of the class are expected to be constrcuted and ready to go before step 3; However, if the Vec
definition has default values, then it WILL work;
}

// Initialization of v1 and v2 must happen in Step 2 not Step 3

Basis b; // FAIL
```

- Object creation steps
  - when an object is created
    1. space is allocated
    2. Fields constructed in declaration order i.e. constructor runs for fields that are objects
    3. constructor body runs
- Member Initializations List (MIL)

```
Basis::Basis(): v1{1, 0}, v2{0, 1} // MIL, step 2
{
// step 3
}
```

- Note: any field can be initialized this way not only object fields

```
// Default values with MIL
struct Basis {
        Vec v1{1, 0}, v2{0, 1}; // If an MIL does not mention a field these values are used

        Basis () {}; // uses default values above
        Basis (const Vec & v1, const &v2): v1{v1}, v2{v2} {} // what constructor is this?
}
```

- Note: fileds are initialized in the order they were declared in the class, regardless of order in MIL
- Consider

```
struct Student {
        int assigns ...; // not objects
        string name; // object
        student (int assigns, ..., const string &name) {
                this->assigns = assigns; //this is an assignment statement not initialization
                ...
                this->name = name;
// in step 2, string is allocated and assigned an empty string
// in step 3, the string is reassigned to name
        }
// VS
struct student (int assigns, ... , const string &name): assigns{assigns}, ..., name{name} {}
}
// in step 2: name is allocated and initialized with name fields
```

- Sometimes MIL is more efficient than setting fields in the constructor body
- MIL must be used for
  - fields that are objects with no default constructor
  - fields that are const or reference
- MIL should be used as much as possible!
- Destructors - a method
  - when an object is destroyed
    - stack-allocated goes out of scope
    - heap-allocated deleted
- Classes come with destructor
  - simply invokes destructor for all fields that are objects
- When an object is destroyed
  1. destructor body runs
  2. (object) fileds destructors invoked in **reverse** declaration order
  3. space deallocated
- When do we need to write our own?

```
struct Node {
        int data;
        Node *next;
}
Node *np = new Node{1, new Node{2, new Node{3, nullptr}}}; // np is on the stack
```

- If `np` goes out of scope(`np` is not an object, it's just a pointer), the pointer is reclaimed (stack-allocated) -> memory leak - whole list is lost
- If we say `delate np;`, first node is reclaimed, `*np`'s destructor is called but does nothing because fields are not objects, nodes 2 and 3 are leaked!!! So we need to write our own.

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

        ~Node() {delete next;}
};
```

- what does `delete nullptr;` do?
    - SAFE, it does nothing
    - stops the recursion
- Now `delete np;` frees whole list
- Recall

```
Basis::Basis(const Vec &1, const Vec &2): v1{v1}, v2{v2} {}
// what ('v1{v1}, v2{v2}') constructor is this? copy constructor
```

- Also

```
student s{60, 70, 80};
student s2 = s;
/*
initialization uses the copy constructor for constructing one object as a copy of another, so the type has to be
the same
*/
```

- Note: every class comes with
    - default constructor - default constructs all fields that are objects
        - goes away if you define your own constructor
    - copy constructor (simply copies all fields)
    - destructor
    - move constructor
    - move assignment operator
- Can write your own

```
struct Student {
        int assigns, mid, final;
        ...
        Student(const Student &other): assigns{other.assigns}, mid{other.mid}, final{other.final} {}
};
```

- When is the built-in copy constructor not correct?

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

Node *n = new Node{1, new Node{2, new Node{3, nullptr}}}
};
Node m = *n; // copy constructor
Node *p = new Node{*n}; // copy constructor;
```

- simply copy the filelds only first node is copied: Shallow Copy
- If you want a deep copy(copy a whole list), you must write your own copy constructor.

Lecture 8

```
struct Node {
        int data;
        Node *Next;
};
Node *n = new Node{1, new Node{2, new Node{3, nullptr}}}
Node m = *n; // copy constructor
Node *p = new Node{*n}; // copy constructor     "shallow copy", not a "deep copy"
// Note that m and p share the same heap memory with n on heap for Node 2 and 3
```

- What if you want a deep copy(whole list) instead?

```
Node(const Node &other): data{other.data}, next{other.next ? new Node {*other.next} : nullptr} {}
// recursively copies the list
```

- The copy constructor is called
    - When an object is initialized with an object of the same type
    - When an object is passed by value
    - When an object is returned by value

    - for now ... more truths later

```
struct Node {
        Node (Node other): ... {} // the problem for defining a copy constructor but need it to pass by value;
you call constructor here since you pass by value
}
```

- Note: careful with constructor that can take one argument

```
Node (int data): data{data}, next{nullptr} {}
Node n{4};
Node n = 4; // implicit conversion of 4 into Node

int f(Node n) {...}
f(n);
f(4); // This works, because 4 is implicitly converted to Node
```

- Danger: accidentally passing an `int` to a function expecting a Node
    - silent conversion
    - cimpiler does not sign any errors
    - hard to debug
- Disable the implicit conversion
    - make the constructor explicit

```
explicit Node (int data, Node *n = nullptr): int{data}, next{n} {}
Node n{4}; // works
Node n = 4; // won't work
f(4); // won't work
f(Node{4}); // works


Student s1{60, 70, 80};
Student s2 = s1; // copy constructor
Student s3; // default constructor(which means your constructor has no arguments, not neccesarily the built-in
one)
s3 = s1; // Assignment statement
         // copy object of same type but not constructor
                 // copy assignment operator
                 // uses compiler-supplied default
                 // may want to write your own


struct Node {
// Below, &, so cascading works, expected, like a = b = c;
Node &operator=(const Node &other) { // '&' in 'Node &other' is reference
                if (this == &other) return *this;
                data = other.data;
                delete next; // must delete before overwritre, otherwise, memory leak
                next = other.next ? new Node(*other.next) : nullptr;
                return *this;
        }
}
```

- *I've got an interview in 30 minutes ... How's that gonna be??? who knows for now...*
- Dangerous?

```
Node n{1, new Node{2, nullptr}};
n = n;
```

- deletes list(`next`) then tries to make a copy of it; undefined behaviour
- When writing operator =, ALWAYS make sure it behaves well in the case of self assignment
- if `(*this == other)` okay? Answers from ChatGPT
    - No, replacing `(this == &other)` with `(*this == other)` would not work as intended in the assignment operator function.
    - The comparison `this == &other` checks whether the address of the current object (`this`) is equal to the address of the `other` object. It is used to handle self-assignment and ensure that unnecessary work is avoided when assigning an object to itself.
    - On the other hand, `(*this == other)` compares the values of the current object `(*this)` and the other object using the equality operator (==). This comparison is typically used in the equality operator overload (`operator==`), not in the assignment operator.
    - To handle self-assignment correctly in the assignment operator function, you should use `(this == &other)` as originally suggested. This ensures that the assignment is only performed if the objects are distinct and avoids unnecessary work in the case of self-assignment.
- When does this really happen? `n = n`?
    - `*p = *q;` where `p,q` point to same location
    - `a[i] = a[j];` where `i` is equal to `j` in a for loop
- Better implementation

```
struct Node{
        Node & operator = (const Node &other) {
                if (this == &other) return *this;
```

```
                Node *tmp = next;
                next = other.next ? new Node{*other.next} : nullptr;
                // if new fails, Node will still be in a valid state
                // when new fails ... happens ... details later
                data = other.data;
                delete tmp;
                return *this;
        }
}
```

- Alt Approach copy & swap

```
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; // copy constructor
                swap(tmp);
                return *this;
        }
}
```

- Destructors are called when one of the following events occurs:
  - A local (automatic) object with block scope goes out of scope.
  - An object allocated using the `new` operator is explicitly deallocated using `delete`.
- Should the copy constructor take a reference to the object being copied or should it take the object by value? Answers from
  - ChatGPT: The **copy constructor** should always take a reference to the object being copied. This is because passing by value would require a copy to be made, and to make a copy, you'd need to call the copy constructor, which would in turn need to make a copy, and so on, leading to an infinite loop. This would not even compile, as it would violate the language's completeness requirements.

Lecture 9

```
struct Node {
        Node &operator=(const Node &other) { // copy assignment operator implementation #4
                Node temp = other; // copy constructor; most of the work is done here
                swap(temp);
                return *this;
        }
} // tmp goes out of scope and destroys all of our old nodes with it
```

Rvalue and rvalue references

- Recall
  - an lvalue is anything with an address
  - an lvalue reference (&) is like a
    - const pointer with auto-dereference
    - always initialized with an lvalue
- Consider

```
Node oddsOrEvens() {
        Node odds{1, new Node{3, new Node{5, nullptr}}};
        Node evens{2, new Node{4, new Node{6, nullptr}}};
        char c;
        cin >> c;
        return (c == 0) ? evens : odds;
}

Node n = oddsOrEvens(); // what constructor is used? copy constructor? But what is the other parameter??
```

- The compiler createds a temporary object to hold the result of `oddsOrEvens`
  - `other` is a reference to this temporary object, where `other` is the parameter con copy constructor
- But
  - temporary object is simply going to be discarded anyway, when `Node n = oddOrEvens()` is done
  - wasteful to handle deep-copy data from the temporary object - why not steal its data
    - save the cost of a copy
- Need to be able to tell whether `other` is
  - a reference to a temporary object: okay to steal from
  - or a stand alone object: must make a copy
- In C++, an rvalue reference of type `Node &&` is a reference to a temporary object (or rvalue) of type `Node`
- Version of constructor that takes `Node &&`

```
struct Node {
        // called a move constructor
        Node (Node &&other) : data{other.data}, next{other.next} // here stealing other's data
        {
                other.next = nullptr; // stops the rest of the list from being reclaimed when the temporary
object is destroyed
        }
}
```

- Similarly

```
Node m;
m = oddsOrEven(); // move assignment operator
// steals nodes'from temporary object, must take care of any nodes currently associated with m
struct Node {
        Node &operator=(Node &&other) {
                using std::swap;
                swap(data, other.data);
                swap(next, other.next);
                return *this;
        } // stealing other's data, giving other the old data, easy, fast swap without copy.
          // temporary gets destroyed, takes our nodes with it
};
```

- If you don't define a move constructor/move assignment operator, the copy constructor/copy assignment operator is used
- If the move constructor/move assignment operator are defined, they will replace calls to the copy constrcutor/copy assignment operator when the argument is a temporary object (rvalue).
- Copy/Move Elision

```
Vec makeAVec() {
        return {0, 0}; // c++ allows just return {0, 0}, in C it must return a Vec
}
// use a basic constructor that returns {0, 0};

Vec v = makeAVec(); // what constructor is used? copy constructor? move constructor?
// TRY IT!!!
```

- In g++, just the basic constructor runs - no copy constructor, no move constructor
- In certain cases, the compiler is required to skip calling copy/move constructor
- In this example, `makeVec` writes its result(`{0, 0}`) directly into the space occupied by `v` in the caller rather than copy it later. (more efficient)
- Eg

```
void doSomething(Vec v) { // pass by value
        ...
} // copy/move constructor

doSomething(makeAVec()); // result of makeAVec written directly into parameter no copy or move constructor
```

- this happens even if dropping constructor calls would change the behavious of the program (this sentence might be a bit confusing, ChatGPT interpreted it as: "This means that the compiler is allowed to perform this optimization even if the copy or move constructor has observable side effects. In other words, the compiler can elide (omit) the constructor call even if doing so changes how the program behaves." Suupose we want to print out a message in the copy constructor, but the compiler can still elide the constructor call)
    - eg. cout in copy/move constructors
- You are not expected to know eaxctly when Elision happens - only that it does(this is an optimization in C++20)
- Summary: Rule of Five (Big 5)
    - If you need to write one of
        1. copy constructor
        2. copy assignment operator
        3. destructor
        4. move constructor
        5. move assignment constructor
    - Then typically, you need to write all 5
    - But, many classes don't need to write any
        - built-in implementations are enough
    - Characterization classes when I need to write them
        - ownerships: classes usually taked with managing something, often memory, could also be other resources
- Member Operators
    - Notice `operator=` was a member function, part of a class, not a stand alone
    - `Node &operator=(const Node &other) {...}` when an operator is declared as a member function, `this` plays the role of the *first* operand

```
x = y; // x is the this, y is the other


struct Vec {
        int x, y;
        Vec operator+(const Vec &other) {
```

```
                return {x+other.x, y+other.y};
        } // x+y, x is this, y is other
        Vec operator*(const int k) {
                return {x*k, y*k};
        }
};
// How about k*v; k is an int not a Vec, so Vec not 1st operand, must be stand alone
Vec operator*(const int k, const Vec &other) {
        return v*k;
}
```

Lecture 10

- Advice: If overloading arithmetic operations overload the assignment versions
    - implement the former in terms of the latter

```
Vec &operator+=(Vec &1, const Vec &2) {
        v1.x += v2.x;
        v1.y += v2.y;
}

Vec operator+(const Vec &v1, const Vec &v2) {
        Vec temp{v1};
        return temp += v2; // use += to implement
        // return its value
}
```

- I/O operators

```
struct Vec {
        ...
        operator &operator<<(ostream &out) {
                return out << x << ' ' << y;
        }
        // what's wrong with this?
        // v << out; // it tries to implement like this, this is confusing
}
```

- So >> and << should he standalone, not member functions
- Some operations must be members
    - operator =
    - operator `[]` - index for arrays
    - operator ->
    - operator `()` - function call operator
    - operator `T` - where `T` is a type/the operator acts as a conversion
- Arrays of objects

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

Vec *vp = new Vec[10]; // ERROR
Vec moreVecs[15]; // ERROR
```

- Problems above:
    - Need a default constructor
        - or at least that's why we have an error
- Fix
    - Define a default constructor if it makes sense for the object
    - If on stack `Vec moreVecs[3] = {{0,0}, {1,3}, {2,4}}`
    - If on the heap, create an array of pointers

```
Vec **vp = new Vec*[10];
vp[0] = new Vec{0,0};
vp[1] = new Vec{1,3};

for (int i = 0; i < 10; i++) {
        delete vp[i];
}

delete [] vp;
```

- const objects

```
int f(const Node &n) {...}
// fields of object are protected, can't mutate
// often use const for parameters
```

- Can we call member functions/methods on a const object?
    - concern: method may change fields => violating const
    - A: yes but only if we promise the function won't change the data

```
struct Student {
        int assigns, mid, final;
        float grade() const; // const is part of function signature, compiler will verify this
                                                    // must be in both interface and implementation
};

float grade() const {
        return assigns*0.4 + mid*0.2 + final*0.4;
}
```

- An object declared as `const` cannot be modified and hence, can invoke only const member functions as these functions ensure not to modify the object.
- Profiler
    - suppose we want some usage stats

```
Student {
        int numMethodCalls = 0;
        float grade() const {
                ++numMethodCalls; // ERROR
                return ...
        }
};
```

- Two types of constness
    - Physical constness - whether or not the bits themselves have changed 0 -> 1
    - Logical constness - whether the updated object should logically be regarded as different after the update
- Want to be able to update `numMethodCalls` even if the object is const => declare field `mutable`

```
Student {
        mutable int numMethodCalls = 0;
        float grade() const {
                ++numMethodCalls; // no longer an error
                return ...
        }
};
```

- Use mutable fields to initalize a field does not contribute to the logical constness of the object
- `numMethodCalls` tracks the number of times `grade` is called for that particular object
- What if we want calls over all objects of this type?
    - `static` variables
- Eg Track # student objects created
- `static` members - associated with the class itself not any particular object of that type

```
struct Student {
        inline static int numInstances = 0; // inline typically means replace function calls with the body of
function instead
        static void howMany() {
                cout << numInstances << endl;
        }
}
```

- comparing objects
    - c-style strings in C: `strcmp(s1, s2); // s1: char *`
    - strings c++: `s1 < s2, s1 == s2, s1 > s2`

```
string s1 = ..., s2 = ...,
if (s1 < s2) {...} // 2 comparisons
else if (s1 == s2) ...
else // s1 > s2
```

- C

```
int n = strcmp(s1, s2) // 1 comparison
if (n < 0) ...
else if (n == 0) ...
else ...
// want an operator that works like this but for c++
```

- Threeway comparison operator <=>
    - aka spaceship operator

```
import <compare>;
std::strong-ordering result = s1 <=> s2; //std::strong-ordering: a lot to type, not function to remember
if (result < 0) cout << "less";
else if (result == 0) cout << "equal";
else cout << "greater";
```

- shortcut: `auto result = s1 <=> s2`

```
auto x = expr; // declares x to have the type matching value of expr
```

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

Lecture 11

- `static` - associated with the class, not a particular object
- `static` data members
    - cannot be mutable
    - `inline` -> can be specified with an initalization
        - does not need an "out-of-box" definition (i.e. implementation) from C++17
        - ChatGPT: In traditional C++, if you have a `static` data member in a class (without `inline`), you must provide a definition for it outside of the class.
- static functions
    - no implicit `this`
- Comparing objects

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

- When you define <=>, you get the 6 relationships for free `< , <=, ==, !=, >, >=`, sometimes <=> is also free

```
struct Vec {
        int x, y;
        auto operator <=> (const Vec &other) const = default;
};   // lexigraphic on fields
```

- When is default not what we want?
    - When do we write our own

```
struct Node {
        int data;
        Node *next; // stores memory address
                                // after comparing data, it compares memory address in next pointer - meaningless
}

// so
struct Node {
        auto operator<=>(const Node &other) const {
                auto n = data <=> other.data;
                if (n != 0) return n; // from here, n = 0
                if (!next && !other.next) {
                        return n;
                }
                if (!next) return std::strong-ordering::less;
                if (!other.next) return std::strong-ordering::greater;
                return *next <=> *other.next;
        }
}
```

- consider

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

Node n1{1, new Node{2, nullptr}}
Node n2{2, nullptr};
Node n3{3, &n2};
```

- What happens when these go out of scope?
    - `n1` ~ works okay, frees memory on heap
    - `n2` ~ `delete nullptr` okay, does nothing
    - `n3` ~ `delete &n2` - trying to delete a stack address - wasn't new, wasn't allocated on the heap
- Assumption: `next` is either nullptr or address for a node allocated on heap
- Invariant - a statement that is true
- Can we generate the Invariant?
    - A: No, can't trust users
- Eg. stack Invariant: LIFO, FILO
    - if user has access to change the ordering in the stack we can't guarantee the Invariant

Invariants and Encapsulation

- To enforce invariants, we introduce Encapsulation
    - we want clients to treat our objects as "black boxes", "capsules"
    - abstract away implementation details
        - sealed away from client
    - clients can only manipulate the object using provided method
- Encapsulation is the binding of data together with the methods that operate on the data. Also, how we limit access to the data through the provided methods.
- Want clients to treat objects as *capsules* - similar to black boxes.
- Want to avoid: clients writing code dependent on an implementation (that could change), using code in a way that violates how it was intended to be used (violating invariants), etc.
- Properties of encapsulation:
    - hide implementation details
    - design to prevent client misuse
- Eg

```
struct Vec { // by default, members of a struct are public
        Vec(int x, int y);
        private: // what follows is private, cannot be accessed outside struct Vec
                int x, y;
        public: // accessible by anyone
                Vec operator+(const Vec &other);
};
// In general, we want members to be private then specify which ones are public

Vec v1{1, 2};
v1.x = 3; // ERROR


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

- Encapsulation for Linked Lists
    - create a wrapper class List that has exclusive access to the underlying Node objects

```
// list.cc Interface
export class List {
        struct Node; // Private nested class
        Node *theList = nullptr;
        Public:
                void addToFront(int n);
                int &ith(int i);
                ~List();
}

// list-impl.cc
struct List::Node { // Nested class
        int data;
        Node *next;
        ...
        ~Node() {delete.next;}
};
```

```
List::~List() {delete next};
void List::addToFront(int n) {
        theList = new Node{n, theList};
}
int &ith(int n) {
        Node *cur = theList;
        for (int j = 0; j < i; ++j, cur = cur->next);
        return cur->data;
}
// only List can manipulate Node objects -> Invariant guaranteed
// How long does it take a client to traverse a linked list?
ith(0);
ith(1);
ith(2);
...
ith(n);
// the time complexity if O(n^2)
// We want traverse list to be O(n) but also don't want the client to have access to the nodes for node address
```

- SE topic: Design Pattern
  - certain pragramming challenges arise often => remember goto solution and apply/adopt solution when this problem arises
- Essence of a design pattern: if you have "this" problem, then "this" programming techinique may solve it
- Iterator Pattern
  - create a class that manages access to nodes
    - abstraction of a pointer
    - lets us walk the list without exposing the actual pointers

```
class List {
        struct Node;
        Node * theList;
        public:
                class iterator {
                        Node *p;
                        public:
                                explicit Iterator(Node *p): p{p} ()
                                int &operator*() {return p->data,}
                                Iterator &operator++() {
                                        p = p->next;
                                        return *this;
                                }
                                bool operator!= () {};
                                Iterator begin() {}
                                // complete definition see next lecture
                }
}
```

- Binary Operators (`operator+`, `operator-` etc.) as Member Functions:
  - Invoked on the left-hand side object.
  - The right-hand side object is passed as an argument.
  - Example: `a + b` is equivalent to `a.operator+(b)`.
- Unary Prefix and Postfix Increment/Decrement Operators (`operator++`, `operator--`) as Member Functions:
  - Prefix (e.g., `++a`): Increments/Decrements the object and returns the new value. Defined as `Type& operator++()`.
  - Postfix (e.g., `a++`): Increments/Decrements the object but returns the original value. Defined as `Type operator++(int)`.
  - The `int` parameter in the postfix version is not used; it's only for distinguishing between the two versions.
- You don't need to provide a default value for the int parameter in the postfix version of the increment/decrement operators. The int parameter is just a placeholder to distinguish the postfix version from the prefix version, and the compiler automatically understands that if the increment or decrement operator is used with the syntax `variable++` or `variable--`, it should call the postfix version with the `int` parameter. The actual value passed for this parameter is irrelevant and typically not used within the implementation.

Lecture 12

- Complement course slides Encapsulation.pdf
- Encapsulation, similar to modularization
  - information hiding
- `class` - by default, visibility is `private`
- Iterator Pattern! What do we want to be able to do?
  - Move from one item in the List to another ("increment the pointer").
  - Access the data at the current location ("dereference the pointer").
  - Have a starting point: `begin()`.
  - Have a finishing point: `end()`.
  - Be able to check if we are at the end of not: `operator!=`

```
class List {
   struct Node;
   Node *theList;
 public:
```

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

// client usage
int main() {
  List lst;
  lst.addToFront(1);
  lst.addToFront(2);
  lst.addToFront(3);
  //  What type is auto here? Iterator
  for (auto it = lst.begin(); it != lst.end(); ++it) {
    cout << *it << endl;
  }
}
```

- Built-in Support for the Iterator Pattern
  - Class Requirements:
    - Methods `begin` and `end` that return `Iterators`
  - Iterator Requirements:
    - Must support prefix `operator++`, `operator!=` and unary `operator*`
  - Range-based for loop (C++11)

```
// access by value (makes a copy) of variable n, of type int
for (auto n : lst) {
    cout << n << endl;  // implicit: n = *it
}
```

- Recall: `int &operator*() { return p->data; }` gives access to mutate.

```
// access by reference to be able to mutate
for (auto &n : lst) {
        n=...; //e.g.++n
}
```

- Equality revisited
  - Suppose we want to add `length()` to `List`
    1. Traverse `List` to count: O(n)
    2. Add field to store length
       - keep it up to date, `addToFront()` ~ increment length, minimal change O(1)
       - `length()`~O(1)
       - Preferred!!!
- Consider implications on spaceship operator <=>
  - special case: if lengths of the two lists we are comparing are different then the lists `!=` => O(1)
  - original: Node by Node comparison O(n) where n is the length of the shorter list

```
class List {
        Node *theList;
        public:
        auto operator<=>(const &List other) const {
                if (!theList && !other.theList) return std::strong-ordering::equal;
                if (!theList) return ...;
                if (!other.list) return ...;
                return *theList <=> *other.theList; // Note: The dot . has higher precedence than the dereference
*, so the expression *other.theList is equivalent to *(other.theList).
        }
        bool operator==(const &List other) const {
                if (length != other.length) return false;
                return (*this <=> other) == 0;
        }
        // if you write operator== separately, compiler will use it for both == and !=
};
```

- For C++ Operator Precedence, see here

- Friendships weaken encapsulation - classes should have as few friends as possible.

Lecture 13

- Complement course slides UML.pdf
- `operator<=>` provides automatic implementation of 6 relation operators
- if you implement `operator==` compiler will use it for ==, `!=`
- if you only implement `operator!=`, you will not get `operator==` automatically
- car owns an engine, 'owns a'
    - destroy car, destroy engine
    - copy car, copy engine(deep copy)
- catelog of car parts
    - car parts are in a warehouse
- 'has a' destroy catelog, parts live on
    - copy catelog, share parts
- 'Ducks'(B) and 'Ponds'(A)
- Book collection

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

- 'Text book' also stores topic

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

- Comic also want hero

```
class Comic {
        string author, title;
        int length;
        string hero;
        public:
                Comic(...)...
};
```

- Organization standpoint OK but it really doesn't represent how these things are related.
- How would I make an array of these objects?
    - `void *` but can't dereference
    - union `BookTypes{Book *b, Text *t, Comic *c};`
        - `BookTypes myBooks[20];`
    - need an extra array to keep track of type of each index
- Neither are great.....
- **C++ Inheritance**

```
// SuperClass, Base Class, Parent
class Book {
                string author, title;
                int length;
        public:
                Book(...) ...
};


// Subclass, Derived Class, Child
class Text : public Book {
                string topic;
        public:
                Text(...)...
};


// Subclass, Derived Class, Child
class Comic : public Book {
                string hero;
        public:
```

```
                Comic(...)...
};
```

- Derived classes inherit fields or methods from the Base class
- `Text, Comic` inherit `author, title, length` from `Book`
- Any method that can be called on `Book` can be called on `Text` and `Comic`
- Who can see the members' of `Book`?
    - private in `Book` means no one outside can see them
- Can `Text, Comic` see them? NO! even subclasses can't see them
- How do we initialize `Text`?
    - need `author, title, length` (`Book` part)
    - need `topic`

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

- 2 things WRONG above
    - Text does not have access to `Book` fields: `author, title, length`
    - MIL only lets you mention your own fields
- Object Construction
    1. Allocate space
    2. The superclass part is constrcuted
    3. fields are constrcuted
    4. body of constructor runs

```
class text : public Book {
        ...
        public:
                Text(string author, string title, int length, string topic):
                Book{author, title, length}(step 2), topic{topic}(step 3) {(step 4)}
}
```

- What if a `Book` default constructor doesn't exist
    - need to explicitly call a constructor
- May want to give subclasses access to certain members.
    - `Visibility:protected`

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

- Should we allow unrestcited access (to a superclass) by a subclass?
    - No, if this is allowed, a child may break an invariant => break encapsulation

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

// isHeavy()
// - Book > 200 pages
// - Text > 300
// - Comic > 30
isHeavy()
comic c{};
Book *pb = &c;
// call isHeavy()?
```

- In case the concept of encapsulation is still confusing, here is the explaination from ChatGPT (yes, ChatGPT again): Encapsulation is one of the fundamental concepts in object-oriented programming (OOP). It refers to the bundling of data (attributes) and methods (functions) that operate on the data into a single unit or class, and restricting the direct access to some of the object's components.
    - **Information Hiding**: Not all the details of an object are to be exposed to the outside world. Encapsulation ensures that the internal representation of an object is hidden from the outside. This is usually achieved using access specifiers like `private`, `protected`, and `public` in languages like C++ and Java.

- **Control**: By encapsulating the details, you can control activities like reading or modifying the data of an object. For example, you can use getters (accessor methods) and setters (mutator methods) to control how attributes of an object are accessed or modified.
- **Flexibility** and Maintenance: Since the internal details are hidden, changes can be made to the internal representation of the class without affecting external components that use the class. This makes the system more flexible and easier to maintain.
- **Increased Security**: By hiding the attributes and making them private, you can prevent unauthorized operations on the data.

Lecture 14

- We will use `#include` for A3 and A4 instead of `import`
- May want to give subclasses access to certain members

```
Visibility::protected
class Book {
        protected:
                string title, author; // Books can access, subclassess can acces
                int length;
        public:
                Book(...)
}
```

- Better: keep the fields private and provide protected accessors/mutators

```
class Book {
                string title, autor;
                int length;
        protected:
                void setAuthor(const string &newAuthor);
        public:
                Book(...);
                bool isHeavy() const;
}
```

- `isHeavy()`
    - `Book`: > 200 pages
    - `Text`: > 400 pages
    - `Comic`: > 30 pages

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

class Comic : public Book {
        ...
        bool isHeavy() const {return length > 30;}
}

Book b{"...", "...", 100};
Comic c{"...", "...", 50, "..."};

cout << b.isHeavy() // False
         << c.isHeavy(); // True
```

- Since a `Comic` is a `Book` (Inheritance), can use

```
Book b = Comic{"...", "...", 50, "..."}; // Allocate space for a Book
b.isHeavy(); // Book::isHeavy()? or Comic.isHeavy()?
```

- `Comic` is sliced - here field is chopped off.
    - `Comic` object is coerced into a `Book`
    - convert `Comic` into a `Book` => `Book::isHeavy()` is executed.

```
Comic c{"...", "...", 50, "..."};
Book *bp = &c;
cout << c.isHeavy()  // call Comic::isHeavy() True
         << bp->isHeavy(); // ?? Book::isHeavy() is called even though object is Comic
```

- Compiler uses the type of pointer (or reference)
    - does not consider actual type of object
- Behaviour of the object is based on the type of pointer(or reference) you access it through
- How do we make `Comic` act like a `Comic` even when pointed at a `Book` ptr?
    - `virtual` - make method virtual

```
class Book {
        ...
```

```
        virtual bool isHeavy() const {length > 200;}
};

class Comic : public Book {
        bool isHeavy() const {length > 30;}
}

Comic c{"...", "...", 40, "..."};
Comic *pC = &c;
Book *pB = &c;
Book &rB = c;

cout << c.isHeavy()
         << pC->isHeavy()
         << pB->isHeavy()
         << rB.isHeavy();
// True for all of the above, all use Comic::isHeavy()
```

- `virtual` method - choose which class's method to use based on actual object type at runtime.
- Ex: My Book Collection

```
Book *myBooks[20]; // array of pointers not Book
// Careful: pass a Comic array to a Book array parameter, the memory will not matched when dereference.
// Think 2D array
for (int c = 0; c < 20; ++c) {
        cout << myBooks[c]->isHeavy() << endl; // isHeavy() uses version associated with object type
}
```

- From ChatGPT: When dealing with polymorphism and inheritance in C++, pointers (or references) are used to access the derived class's methods and avoid object slicing. By using pointers or references, you maintain the ability to call the appropriate derived class methods on an object, thanks to the virtual dispatch mechanism.
    - If you try to "pass a Comic array to a Book array parameter," you're effectively trying to assign each `Comic` object to a `Book` object. This will result in object slicing for each element, and the specific `Comic` attributes will be lost.
    - Now, if you try to dereference a pointer or access the objects in the `Book` array, you're only accessing the `Book` part of each original `Comic` object, because the `Comic`-specific data has been sliced off.
    - The memory mismatch or the behavior you mentioned arises because you might be expecting the full `Comic` object features, but since object slicing has occurred, you're left with only the `Book` features. This is one reason why, in C++, polymorphism is achieved using *pointers* or *references*, which avoids object slicing.
- Accomodating multiple types under one abstraction
    - **Polymorphism** ~ Poly = "many", morph = "form"
- Destructor revisted. Steps for Destroying a Subclass object:
    1. Subclass destruction
    2. Subclass fields that are objects are destroyed
    3. Superclass part is destroyed(destructor called)
    4. Space is reclaimed

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

class Y : public X {
        int *y;
        public:
                Y(int m, int n): X{n}, y{new int[m]} {}
                ~Y() {delete [] y;} // as part of step 3, destructor for X is called, don't do it explicitly
}

X *myX = new Y{10, 20};
delete myX;
// X's destructor is called, memory leak!!!
```

- How do we make it so the object's destructor is used rather than the one associated with pointer type => make superclass destructor `virtual`

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

- If a class might have a subclass, declare destructor `virtual` even if it's empty
- If a class is not meant to have a subclass, declare it `final`

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

- Pure `virtual` methods and Abstract classes

```
class Student {
        protected:
                int numCourses;
        public:
                virtual int fees() const = 0; // pure vitual methods
};
// 2 types of students: Regular and Coop
// Only these 2, what do we write for fees()?
// Don't want to write it for Student
// No implementation

class Regular : public Student {
        public:
                int fees() const override; // overiding Student::fees(); Okay because it's virtual; there will be
an error if there is no virtual in the SuperClass
                // readability - lets the client know
                // compiler will check params match
};

class Coop public Student {
        public:
                int fees() const override;
};


Student s; // ERROR
```

- A class with a pure virtual method cannot be instantiated => called an Abstract class => purpose: organization
- subclasses of an abstract class are also abstract unless they implement all pure virtual methods
- Non-abstract classes are called concrete
- UML: represent virtual and pure virtual methods using italics
    - Representing classes by italicizing the class name

Lecture 15

```
void f(Book books[]) {
        books[1] = Book{"book", "bauth", 100};
};

Comic c[2] = {{"c1", "a1", 10, "h1"}, { "c2", "a2", 20, "h2"}};

f(c);
```

- NEVER use arrays of objects polymorphically => use array of pointers

Inheritance with copy/move

```
class Book {
        public:
                // Defines copy/move constructor & assigns
}

class Text : public Book {
        string topic;
        public:
                // Does not define copy/move constructor and assigns
}

Text t{"Algorithm", "CLR", 1200, "CS"}; // CS 341

Text t2 = t; // copy constructor but not defined for Text
```

- This copy initialization calls `Book`'s copy constructor and then goes by field (default behaviour) for the `Text` part
- Same is true for other compiler-provided ops
- Write your own:

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

Text &Text::operator=(const Text &other) {
        Book::operator=(other);
        topic = other.topic;
        return *this;
}
```

```
Text::Text(Text &&other) : Book{std::move(other)}, topic{std::move(other.topic)} {} // std::move(other) tells
Okay to steal from

Text &Text::operator=(Text &&other) {
        Book::operator=(std::move(other));
        topic = std::move(other.topic);
        return *this;
}
```

- `std::move` is a standard utility in C++ that casts its argument to an rvalue reference, effectively making it eligible for move operations (like move constructors or move assignment operators)
- Note: even though other "points" at an rvalue, `other` itself is an lvalue (as is `other.topic`). `std::move(x)` forces an lvalue `x` to be treated as an rvalue so that "move" versions of the operators run.
- The operation we define here, mimic default behaviour
    - specialize them as needed, for `Nodes`, etc ...
- Consider

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

*pb1 = *pb2; // will treat them as Books; Book::operator=(const Book &other) runs => partial assignment-only Book
part is copied
```

- Fix? How do we make it use ops based on obj type rather than pointer type? => `virtual`

```
class Book {
        public:
                virtual Book &operator=(const Book &other) {} // must be virtual
}

class Text : public Book {
        public:
                Text &operator=(const Book &other) override {...}
}
```

- Note: different return type is okay but must(won't compile if not) have the same parameters or it's not an override
- By the "is-a" principle, if a `Book` can be assigned from another book => a `Text` can be assigned from another book

```
text t{};
t = Book{...}; // overwrite the Book part; BAD but it compiles
t = Comic{...}; // VERY BAD but is allowed
```

- Summary
    - if `operator=` is non-virtual then we get partial assignment when using assign through base class pointer
    - if virtual, then the compiler allows mixed assignment
- Recommendation: all superclass should be abstract
- Modify Hierarchy

```
 class AbstractBook {
        string title, author;
        int length;
        protected:
                AbstractBook &operator=(const AbstractBook &other); // prevents assignment through base class
pointers; But implementation is still availble to subclasses
        public:
                AbstractBook(...);
                virtual ~AbstractBook() = 0; // pure virtual method => Abstract class
                // Need at least one pure virtual method if don't have, use the destructor
 };

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

- What the protected assignment operator in the base class prevents?
    - When the assignment operator is protected in the base class, you cannot perform the following:

```
Base b1, b2;
Derived d;
b1 = b2; // Error! Cannot use the base class's assignment operator
b1 = d;  // Error! Cannot use the base class's assignment operator
```

- other classes are similar
- prevents partial and mixed assignments
- Note: virtual destructor must be implemented, even though it is pure virtual, otherwise won't compile
- `Abstract::~Abstract() {}` By step 3 (step to destroy a subclass;) `~AbstractBook` superclass destructor will be called by the subclass, so it must exist.
- Here let's clarify what's Abstract class by ChatGPT.
    - An abstract class in C++ is a class that has at least one pure virtual function. You cannot instantiate objects of an abstract class directly because it's considered incomplete by the compiler; it's missing the implementation of those pure virtual functions.
    - However, you can have pointers (or references) to an abstract class. These pointers can point to instances of derived classes that provide concrete implementations for all of the abstract class's pure virtual functions. This is fundamental to how *polymorphism* works in C++.

Lecture 16

- Vectors
    - Dynamic length arrays
    - guaranteed to be implemented internally as arrays; should use vectors instead of creating your own dynamic length array
    - if calling `new`/`delete []` => probably use vectors instead

```
#include <vector>
using namespace std;
vector<int> v{4,5}; // 4, 5
// if you say v(4, 5), it would be 5, 5, 5, 5
v.emplace_back(6); // 4, 5, 6
v.emplace_back(7); // 4, 5, 6, 7
v.pop_back(); // 4, 5, 6; remove last element
```

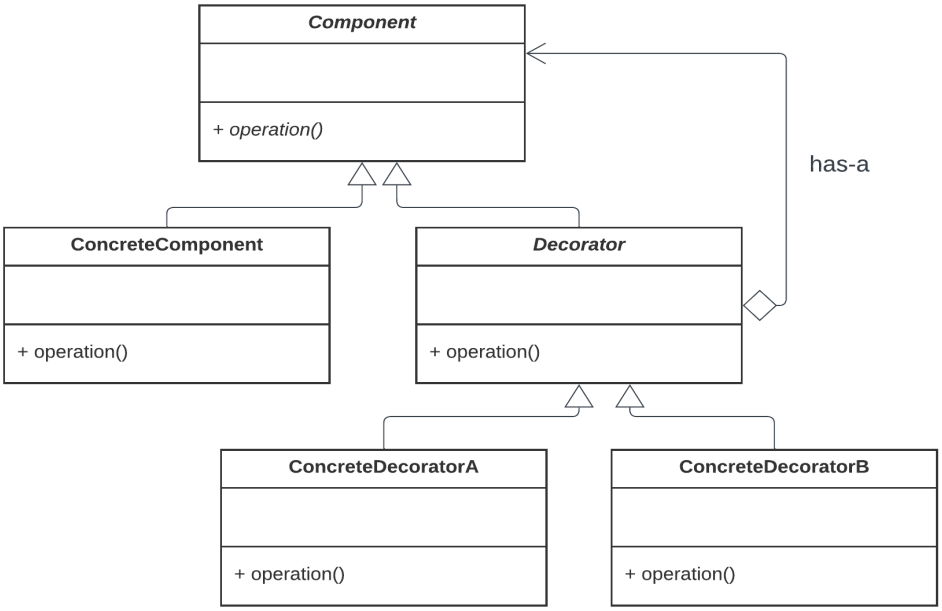
- looping

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

for (vector<int>::iterator it = v.begin(); it != v.end(); ++it) { // you could also use `vector<auto>::iterator`
here instead of `vector<int>::iterator`
        cout << *it << endl;
}

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

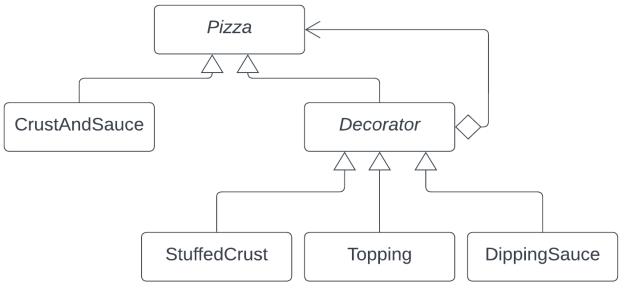
- iterate in reverse

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

- More Design Patterns
- Guiding Principle: "Program to an interface, rather an implementation"
    - Interface: set of operations that a class supports
    - Implementation: how the class supports the operations
- Abstract base class to define the interface
    - work with base class pointer and call their methods
    - concrete subclasses can be supplied in and out
        - provides abstraction over a variety of behaviours
- Decorator Pattern
- Want to enhance an object at runtime
    - add funtionality/features
- Ex. Windowing System
    - start with a basic window
    - add a scroll bar
    - add a menu
- Want the enhancements at runtime
- * means italics
    - Abstract class
    - virtual function



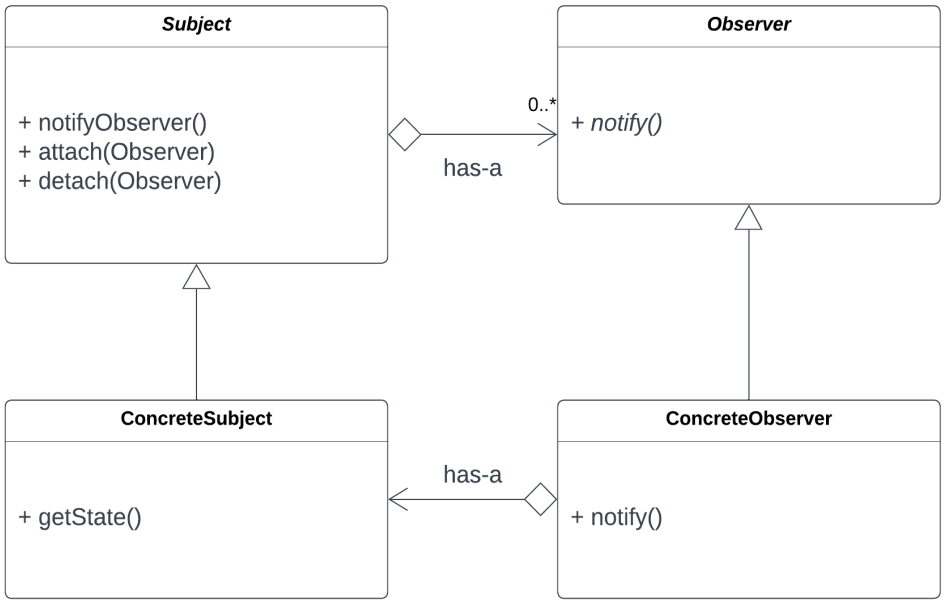
- Component - defines the interface
  - operations your objects will provide
- ConcreteComponent - implements the interface
- Decorators - all inherit from Decorator which inherit from Component
  - Every Decorator is - a Component
  - Every Decorator has - a Component
- Eg. window with srollbar is a kind of window
  - has a pointer to the underlying normal(base) window
- A (window with scrollbar) with menu is a window
  - has a pointer to window with scrollbar(which has a pointer to the underlying normal(base) window)
- All inherit from Abstract Window so window methods can all be used polymorphically on all of them
- Eg. Pizza




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

class CurstAndSauce : 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 StuffedCrust";
                }
};

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;
                }
};

// Usage
Pizza *p1 = new CrustAndSauce{};
p1 = new StuffedCrust(p1);
p1 = new Topping("pineapple", p1);
cout << p1->desc() << " costs $" << p1->price() << endl;
delete p1;
```

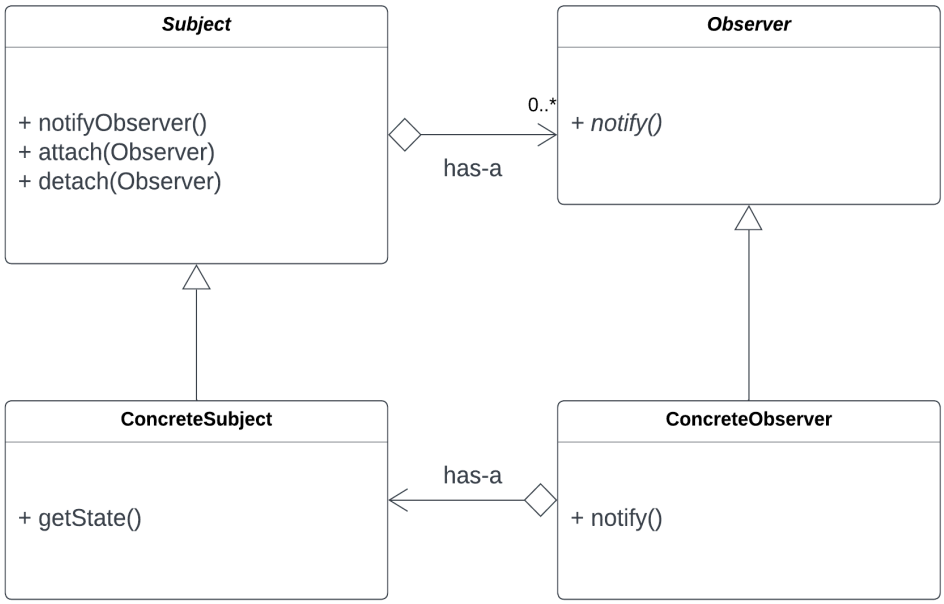
- Obeserver Pattern
    - Publish - Subscribe model
- One class publisher/subject - generate data
- One or more Observers/Subscribers classes
    - receive data and react to it



- Ex. Publisher: Spreadsheet cells; Observers: graphs
- There can be many different kinds of observers
    - subject should not need to know all of the details about them

Lecture 17

- The abstract class `Subject` contains code common to all subjects
- The abstract class `Observer` contains code interface common to all observers



- Sequence of method calls
    1. Subject's stat is updated
    2. `Subject::notifyObserver()` calls each observer's `notify()` notify observer's could be called by `Subject` or an external controller
    3. Each observer calls `ConcreteSubject::getState()` to query the state (what changed) and react
- Ex. Horse Race
    - Subject: Horse Race that states a winner after each race
    - Obeservers: Bettors that bet on horses - watching the race results to see if their horse wins or loses

```
class Subject {
                vector<Observer *> observers;
        public:
                Subject();
                void attach(Observer *ob) {observers.emplace_back(ob);}
                void detach(Observer *ob) {...}
                void notifyObservers() {
                        for (auto &ob: observers) ob->notify();
                }
                virtual ~Subject() = 0;
};

Subject::~Subject() {}

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

class HorseRace : public Subject {
                ifstream in;
                string lastWinner;
        public:
                HorseRace(const string &source): in{source} {}
                ~HorseRace() {}
                bool runRace() {return in >> lastWinner;}
                string getState() const {return lastWinner;}
};

class Bettor : public Observer {
                HorseRace *subject;
                string name, myHorse;
        puclic:
                Bettor(string name, HorseRace *subject, string myHorse): name{name}, subject{subject},
myHorse{myHorse} {
                        subject->attach(this);
                }
                ~Bettor() {
                        subject->detach(this);
                }
                void notify() {
```

```
                        if (subject->getState() == myHorse) {
                                cout << "Woohoo, winner!" << endl;
                        } else {
                                cout << "Boohoo" << endl;
                        }
                }
};

int main() {
        HorseRace hr{"source.txt}
        Bettor Mark{&hr, "Mark", "MyLittlePony"};
        // more bettor
        while (hr.runRace()) {
                hr.notifyObservers();
        }
}
```

- Template
  - A template class is a class parameterized by a type
- List class as template, `vector<int>`

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

List<int> l1;
List<List<int>> ls;
l1.addToFront(3);
ls.addToFront(l1);

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

- How do templates work?
  - Roughly:
    - compiler specialize template into actual code
    - as a source-level transformation then compiles resulting code as uaual
- `vector<int> v = {...}`, use an iterator to remove all S's

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

- There is an issue above!
- Rule: after an insertion or an erase, all iterators pointing after the point of insertion(or erasure) are considered invalid and must be refreshed

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

- `v[i]` - unchecked - doesn't check bounds of the array
- `v.at(i)` - checked. What happens if it's out of bounds?
- Complement course slides Exceptions.pdf

Lecture 18

- Recall: `catch (SomeErrorType s) { ... }`
- Suppose exception `s` is actually a type that is a subclass of `SomeErrorType`, rather than `SomeErrorType` itself.
- Can use `...` as a catch-all for exceptions.

```
try {
        ...
}
catch (...) {  // literally use ... here
        ...
}
```

- Many existing exceptions, but you can also define your own exception classes for errors:

```
class BadInput {};
try {
  if (int n; !(cin >> n)) { throw BadInput{}; }
  catch (BadInput &) {
    cerr << "Input not well-formed\n";
  }
}
```

- Note: exception caught by reference which prevents the exception from being sliced (if it's from a subclass of `BadInput`). Instead it will be treated like the kind of object that it actually is.
- **Catching exceptions by reference is usually the right thing to do.**
- Maxim in C++: Throw by value, catch by reference.
- NEVER let a dtor throw an exception!
    - By default, the program will terminate immediately by calling `std::terminate`. Also, if a dtor is being executed during stack unwinding, while dealing with another exception, and it throws an exception, there will be 2 active, unhandled exceptions and the program will about immediately.
- Design Patterns

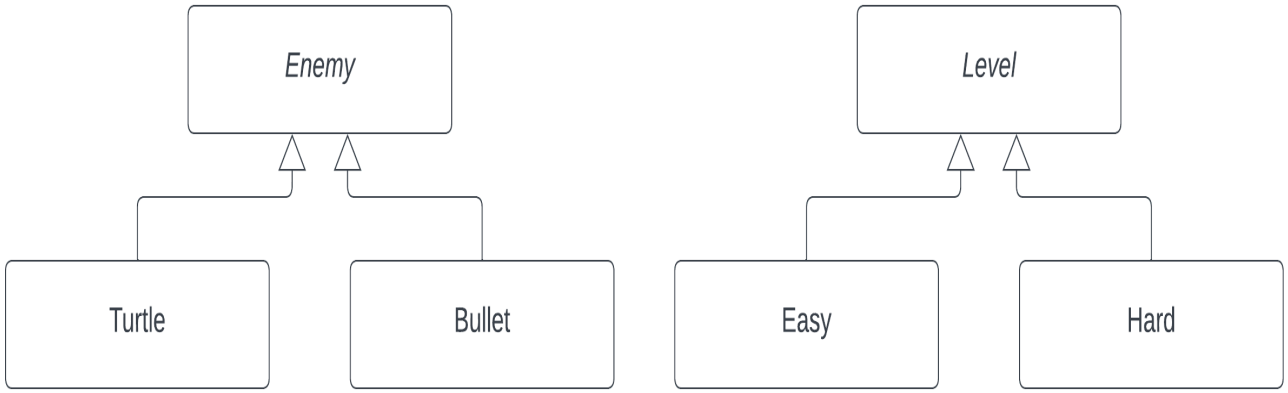
```
class AbstractIterator {
        public:
                virtual int &operator*() const = 0;
                virtual AbstractIterator &operator++() = 0;
                virtual bool operator!=(const AbstractIterator &other) const = 0;
                virtual ~AbstractIterator() {}
};

class List {
                struct Node;
        public:
                class Iterator : public AbstractIterator {
                        ...
                };
};

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

- write functions that operate on iterators derived from `AbstractIterator`
    - same function to iterate over many different data structures
        - don't need to know underlying data structures

```
void foreach(AbstractIterator &start, AbstractIterator &end, void (*f)(int)) {
        while (start != end) {
                f(*start);
                ++start;
        }
}
// works with both List and Set, etc.
```

- Factory Method Pattern
- Problem: write a video game with two kinds of enemies: turtles and bullets but bullets become more frequent in later levels.
- From ChatGPT: The Factory Method Design Pattern is a creational pattern that provides an interface for creating objects in a super class, but allows subclasses to alter the type of objects that will be created. Instead of calling a constructor directly, a factory method is used to create the object. This pattern is particularly useful when the exact type of the object to be created isn't known until runtime, or when the creation process is complex or involves business logic.

- The idea is to create a factory class with a single responsibility to create objects, hiding the details of class modules from the user.



- Never know exactly which enemy will be next so can't tell turtle/bullet constructors directly.
- Don't want to hardcode the policy that decide what's next - it should be customizable.
- Instead, put a factory method in `Level` that creates them.

```
class Level {
        public:
                virtual Enemy *createEnemy() = 0; // 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 *e = l->createEnemy();

// A.k.a Virtual Constructor Pattern
```

- Template Method Pattern
- Want subclass to override some aspects of superclass behaviour but other aspects must stay the same.
- ChatGPT: Template Method Design Pattern defines the structure of an algorithm, allowing the steps of the algorithm to be implemented by derived classes. The base class provides a template (sequence) of steps, and some of those steps can be "hooked" or overridden by derived classes.
- Eg. There are 2 kinds of turtles - green or red. (shellcolour's)

```
class Turtle {
        public:
                void draw() {
                        drawHead();
                        drawShell();
                        drawFeet();
                }
        private:
                void drawHead() {...}
                virtual void drawShell() = 0; // virtual functions can be private
                void drawFeet() {...}
}

class RedTurtle : public Turtle {
        void drawShell() override {... // draw red shell}
}

class GreenTurtle : public Turtle {
        void drawShell() override {... // draw green shell}
}
```

- subclass can only change how the shell is drawn

- Extension: Non-virtual Interface (NVI) Idiom
- A public virtual method is 2 things:
    - `public` an interface for client
        - promises certain behaviour with pre/post conditions
        - promises class invariants
    - `virtual` an interface to subclass
        - behaviour can be replaced with anything the subclass wants
        - conflicting ideas
    - making promises that are designed to be broken
- NVI:
    - all public methods should be non-virtual
    - all virtual methods should be private or at least protected
    - except destructor
- From ChatGPT: NVI aims to keep virtual functions private, exposing only non-virtual public member functions as the class's interface. The public non-virtual member functions can then call the private virtual ones, allowing derived classes to override behavior while still maintaining controlled access.
- Ex. Digital Media Player

```
class DigitalMedia {
        public:
                virtual play() = 0;
};

// Translate to NVI

class DigitalMedia {
        public:
                void play() {
                        // can add code before and after
                        doPlay();
                        // can add code before and after
                }
        private:
                virtual void doPlay() = 0;
};
```

- Now if we need to exert control over `play` we can
    - we can later decide to add before/after code around `doPlay()` that can't be changed.
        - Eg. check copyright before, update play count after, etc.
    - we can add more "hooks" by calling additional virtual methods in play
        - `ShowConvertArt`
    - All of this without changing the public *interface*
- It is much easier to take this kind of control over our virtual methods from the beginning than to try to take back control over them later.
- NVI extends Template Design Pattern by putting every virtual method inside a non-virtual wrapper.
- ChatGPT: The public non-virtual interface methods serve as gateways. They can be thought of as wrappers around the virtual methods. This design allows the base class to enforce invariants, preconditions, or postconditions around the virtual calls. This also provides flexibility in evolving the base class without affecting derived classes. For instance, if there's a need to add logging, validation, or other behaviors around the virtual call in the future, it can be done in the public method without changing the interface or affecting the derived classes.
- essentially no disadvantge - a good compiler can optimize away extra function calls.

Lecture 19

- STL(Standard Template Library) Maps: for creating dictionaties
    - Eg. "array" that map strings to int

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

- if key is not found, it is inserted into the dictionary and the value is default constructed; eg. `int` is 0

```
m.erase("abc");
if (m.count("def")) { // count returns 1 if key is in map, 0 otherwise
// which means, not found = 0, found = 1
        cout << "def is in the map" << endl;
}

// Iterate over a map => sorted order of keys
for (auto &p: m) {
        cout << p.first << " " << p.second << endl;
        // first/second are fields not methods
}
```

- p's type is `std::pair<std::string, int>&`

    - `<pair>` is in `<utility>`
- 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 data/values. no invariants, all fields are values.
- Alt

```
for (auto &[key, value]: p) { // structured binding
        cout << key << " " << value << endl;
}
```

- structure bindings can be used on any structure (class) where all fields are public.

```
Vec v{1, 2}; // assume fields are public
auto [x, y] = v;
```

- or on 'stack' array of fixed size

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

- Exception Safety
    - Consider

```
void f() {
        C mc;
        C *p = new C;
        g(); // what if g throws an exception?
        delete p;
}
```

- What is guaranteed to happen?
    - During stack unwinding, all stack-allocated data is cleanup. Destructor runs, memory reclaimed.
    - BUT heap allocated memory is not reclaimed.
    - if `g` throws, `mc` is not leaked but what `p` points to is leaked.

```
void f() {
        C mc;
        C *p = new C;
        try {
                g();
        }
        catch (...) {
                delete p;
                ... // if you throw another error, you cannot exclude delete p as the subsequent codes will not
be executed. Resource management! That's why we need unique_ptr
        }
        delete p;
}
```

- duplicated code for `delete p;`
- `finally` clause guarantees certain final actions - not in C++, in Java
- Only things you count on in C++ is:
    - destructor for stack-allocated data will run
    - use stack-allocated data as much as possible
- C++ Idiom: RAII - Resource Acquisition Is Initialization
- Ex. files

```
{
        istream f{"file"};
}
```

- Every resource should be wrapped in a stack-allocated object, whose job it is to delete .
- Acquiring the resource (file) happens by initializing the object `f`.
- The file is guranteed to be released (closed) when `f` is popped from the stack - `f`'s destructor runs.
- This can also be done with dynamic memory

```
class std::unique_ptr<T>; // import <memory>
```

- Takes `T*` in the constructor
- The destructor will delete the pointer
- In between, you can dereference the object just like a regular/raw pointer

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

```
        g(); // no leaks guranteed
}
```

- Alt

```
void f() {
        C c;
        auto p = std::make_unique<C>();
        g();
}
```

- constructor args(if any) go between ()
- allocates a `C` object on the heap and puts pointer to it into a `unique_ptr` object
- p's type is `std::unique_ptr<C>`
- Difficult

```
unique_ptr<C> p{new C{...}};
unique_ptr<C> q = p; //ERROR
```

- WE THINK: simply copy the memory address
- What happens when a `unique_ptr` is copied?
    - Don't want to delete the same pointer twice.
    - So can't simply copy the pointer.
- Instead - copying is disabled for `unique_ptr`
    - they can only be moved

```
template <typename T> class unique_ptr {
                T *ptr;
        public:
                explicit unique_ptr(T *p): ptr{p} {} // explicit: no implicit type conversion
                ~unique_ptr() {delete ptr;}
                unique_ptr(const unique_ptr &) = delete; // disable copy
                unique_ptr &operator=(const unique_ptr &) = delete;
                // copy constructor and copy assignment operator are disabled.
                unique_ptr(unique_ptr &&other): ptr{other.ptr} {other.ptr = nullptr;} // move constructor
                unique_ptr &operator=(unique_ptr &&other) { // move assignment operator
                        std::swap(ptr, other.ptr);
                        return *this;
                }
                T &operator*() const {return *ptr;}
};
```

- If you need to able to copy pointers first answer the question of ownership.
    - Who will own the resource?
    - Who will be responsible to free it?
    - If you need to share ownership, use `std::shared_ptr`
    - If you need to transfer ownership, use `std::unique_ptr`
- That pointer should be the `unique_ptr`; All other pointers should be raw pointers; Can fetch the underlying raw pointer using `p.get()`
- New understanding of pointers

Lecture 20

- Smart pointers
    - `unique_ptr` - owns the object it points to
    - `shared_ptr` - shared ownership
- New undertsanding of pointers
- `unique_ptr` - indicates owndership
    - Delete will happen automatically when the `unique_ptr` goes out of scope
- raw pointer indicates non-ownership
    - since a raw pointer is considered not to own the resource it points at => you should not delete it
- moving a `unique_ptr` = transfer of ownership
    - `unique_ptr` can only be moved, not copied
- pointers as parameters
- `void f(unique_ptr<C> p);`
    - `f` will take ownership of the object pointed to by `p`
    - caller loses custody of the object
- `void g(C *p);`
    - `g` does not own what `p` points to => g should not delete `p`
- callers ownership does not change
    - Note: caller may not even own the object
- pointers as results
    - `unique_ptr<C> f();`
        - return by value is always a move
        - `f` is transferring ownership of the `C` object to the caller
    - `C *g();`

        - the pointer returned by `g()` is understood not to be deleted by the caller => it might represent a pointer to non-heap data or to heap data that someone else already owns
        - `g` is not transferring ownership
- Rarely, a situation does arise where you do want joint/shared ownership
    - any of several pointers might end up freeing the resources => `std::shared_ptr`

```
{
        auto p1 = std::make_shared<C>();

        if (...) {
                auto p2 = p1;
        } // p2 goes out of scope
} // p1 goes out of scope
```

- shared pointers maintain a reference count
    - count of all shared pointers pointing at same object
- Memory is freed when count reaches 0
- Use the type of pointer that accurately reflects the ownership role you want
    - Dramatically fewer opportunities for memory leaks
    - No need to remember to delete
- What is *exception safety*?
- It is not
    - exceptions will never happen
    - all exceptions get handled
- It is: after exception has been handled, the program is not left in a broken state or unstable state
- Specifically, 3 levels of exception safety for a function `f`:
    1. Basic Guarantee: if an exception occurs, then program will be in some valid but unspecified state
        - Nothing leaked
        - No corruption of data structures
        - All class invariants are maintained
    2. Strong Guarantee: if `f` throws or propagates the state of the program will be as if `f` has not been called
    3. No-throw Guarantee: `f` will never throw or propagate an exception and will always accomplish its task
- Ex

```
        class A{...}
        class B{...}
        class C {
                A a;
                B b;
        }

public:
        void f() {
                a.g(); // strong guarantee
                b.h(); // strong guarantee
        }
```

- Is `f` exception safe?
    1. if `a.g()` throws, nothing has happened yet, so OK
    2. if `b.h()` throws, the effects of `a.g()` must be undone to offer the strong guarantee
        - very hard or impossible if `a.g()` has non-local side effects
        - so, no probably not exception safe
        - If `A::g` and `B::h` do not have non-local side effects, can use copy-and-swap

```
class C {
        void f() {
                A atemp = a;
                B btemp = b;
                atemp.g(); // if these throw, the original a and b are intact
                btemp.h();
                a = atemp; // if copy assignment operator throws, what happens?
                b = btemp; // in particular, what if b = btemp throws?
        }
}
```

- Better is the "swap" is No-throw
    - assigning *pointers* cannot throw
- Solution: access C's internal state through pointers
    - called the pImpl idiom (pointer to implementation)
    - ChatGPT said: If you have a class with a lot of internal details that are not part of the public interface and are likely to change often, then the Pimpl idiom can be beneficial. It hides these details from users of the class.
        - One of the primary benefits of the Pimpl idiom is that it can drastically reduce compile-time dependencies. If you change the implementation details, the header doesn't change, which means less code needs to be recompiled.
        - It enables forward declaration of classes that are incomplete types in the public header, which can again reduce compile-time dependencies.
        - If a class is simple, stable (unlikely to change often), and doesn't have many implementation details that need to be hidden, then the Pimpl idiom might be overkill.

        - How Pimpl can help in achieving a strong exception guarantee (but not directly by itself. It's more about how you implement your operations with the Pimpl in place): Make a local copy of the "Impl" object, then operate on this copy. If an exception is thrown, the original state (inside the original "Impl" object) remains unchanged. If the operation succeeds, you can swap the original "Impl" with the modified copy, effectively committing the changes. You should know that swap pointers should never fail or throw! See below example for illustration (from lecture).

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

class C {
                unique_ptr<CImpl> pImpl;
        public:
                void f() {
                        auto temp = make_unique<CImpl>(*pImpl);
                        temp->a.g();
                        temp->b.h();
                        std::swap(pImpl, temp);
                }
}
```

- If either `A::g` or `B::h` offer no exception guarantee, then in general, neither can `C::f`
- Exception safety with STL vectors
- vectors
    - encapsulate a heap-allocated array
    - follow RAII - when a stack-allocated vector goes out of scope, then internal heap-allocated array is freed

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

void f() {
        vector<C *> v;
        ...
} // v goes out of scope, array is freed, pointers don't have destructor, so only array is freed.
// * any objects pointed to by the pointers are not freed;
// this vector is not responsible to free them, no ownership - raw pointer

void h() {
        vector <unique_ptr<C>> v;
        ...
} // array is freed, unique pointer destructor runs => so the objects are deleted => don't need any explicit
deallocation
```

- Summary
    - `Vector<C>` owns the objects
    - `Vector<C *>` does not own the objects
    - `Vector<unique_ptr<C>>` owns the objects

Lecture 21

- `vector<C>` - owns the objects
- `vector<C *>` - does not own the objects
- `vector<unique_ptr<C>>` - owns the objects
- Next, consider method `vector<T>::emplace_back` offers a strong guarantee
    - if the array is full `size == cap`, allocate a new array, larger(double?) copy the objects over - copy constructor
        - if we throw an exception
            - destroy new array
            - old array intact
        - strong guarantee, delete old array
- But we would prefer to move array items instead of copy - since they will be deleted anyway
- But if we start to move some items, then throw an exception, the old array is no longer intact => no strong guarantee
- `emplace_back` has a strong guarantee => if the move constructor offers a no-throw guarantee, it will use no-throw. Otherwise it will use the copy constructor which may be slower => your move constructor, move assignment operator, swap should have a no-throw guarantee if possible
    - facilitates optimization

```
class C {
        public:
                C(C &&) noexcept {...}; // move constructor
                C &operator=(C &&) noexcept {...}; // move assignment operator
}
```

- What should go in a Module?
- Two measures of design quality
    - cohesion

    - coupling
- Coupling: how many distinct modules depend on each other
- Low: modules communicate via function calls with basic parameters/results
    - module pass array/structs back and forth
    - module affects each other's control flow
    - module share global data
- High: modules have access to each other's implementation (`friends`)
- High coupling => change to one module require greater changes to other modules
    - harder to reuse and maintain
- Cohesion: how closely elements of a module are related to each other
- Low: arbitrary grouping of unrelated elements; share a common theme, otherwise unrelated elements manipulate state over the lifetime of an object; elements pass data to each other
- High: elements cooperate to perform exactly one task
- ChatGPT: Aim for designs that have **low coupling and high cohesion** for modular, maintainable, and understandable software.
- Special cases

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

class B {
char x;
A y;
}

// ERROR: need to know size
```

- Forward declaration - tell us the type exists but not the size => but can make a pointer to it, forward declare `B`

```
class B;  // Forward declaration of B

class A {
        int x;
        B *y;
}

class B {
        char x;
        A *y;
}
```

- Sometimes one class must come before another

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

- Question: how should `A` and `B` be placed into modules?
    - Modules must be compiled in dependency order
    - One module cannot forward delcare another
    - Therefore `A` and `B` must be in the same module
- Decoupling the Interface (MVC)
    - your primary program class should not be printing things
- Ex

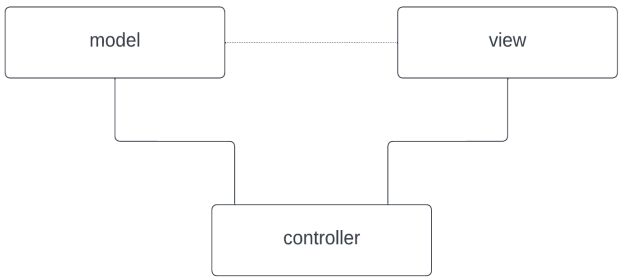
```
class ChessBoard {
        ...
        cout << "Your move">>; // hardcoded std::cout
        ...
}
```

- One solution: have stream fields

```
class ChessBoard {
        istream &in; // set these in constructor
        ostream &out;
        ...
        out << "Your move";
}
```

- What if you don't want streams at all? difficult to reuse either of these module implementations
- **Single Responsibility Principle**
    - a class should have only one reason to change
    - if two or more distinct parts of the problem specification affect the same class then it is doing too much

        - a class should have only one responsibility
        - each class should only have 1 job
            - state of game and communication are 2 jobs
- Better: communication with the chessboard via parameters/results/exceptions; confine user interaction to outside the game class => freedom to change how the game interacts with outside
    - Question: should the interaction code simply in `main`?
    - Answer: NO! It should be in its own module that can be reused
- **Architecture: Model-View-Controller (MVC)**
- Separate:
    - Model: the distinct notions of the data
    - View: the presentation of the data
    - Controller: control and manipulate the data



- Observer pattern; possibly =
- The model
    - can have multiple views: text, graphical
    - doesn't need to know about their details
    - classic observer pattern
- The controller
    - mediates control flow between model and view
    - may encapsulate turn-taking, game rules
    - may communicate with the user input
        - could be view
- MVC promotes reuse

Lecture 22

- Complement course slides Casting.pdf
- In general, casts should be avoided.
- `static_cast` - "sensible cast" with well-defined semantics.

```
// double to int:
double d;
void f(int x);
void f(double x);
f(static_cast<int>(d)); // calls the int version of f
```

- Note: decimal gets truncated. What if we want rounded instead?

```
// Superclass ptr to subclass ptr
Book *b = new Text{ ... };
Text *t = static_cast<Text *>(b);
```

- You are taking responsibility that `b` actually points to a `Text`.
- `reinterpret_cast` - Generally unsafe, implementation dependent, "weird" conversions. Most uses result in undefined behaviour.

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

- For when you want a `Student` to be treated as a `Turtle` - when is that?? Weird!
- `const_cast` - For converting between const and non-const. This is the only C++ cast that can cast away const-ness.

```
void g(int *p); // Knowing g won't actually modify *p
void f(const int *p) {
   ...
   g(const_cast<int *>(p));
   ...
}
```

- `dynamic_cast` - Is it safe to convert a `Book *` to a `Text *`? Is this safe?

```
Book *pb = ...;
static_cast<Text *>(pb)->getTopic();
```

- Depends on what `pb` actually points at! Better to try the cast first and see if it succeeds or not (a tentative cast).

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

- If the cast works (`*pb` really is a Text or a subclass of `Text`), conversion is successful: `pt` will point at the object.
- If object is not the desired type, `pt` will be `nullptr` - you can then test for this.

```
if (pt) cout << pt->getTopic();
else cout << "Not a Text";
```

- Previous examples used raw pointers but we can also cast smart pointers (`unique_ptr`, `shared_ptr`)
    - `static_pointer_cast`
    - `const_pointer_cast`
    - `dynamic_pointer_cast`
    - `reinterpret_pointer_cast`
- Stay within the type: cast `shared_ptrs` to `shared_ptrs`.
- Dynamic Casting with References: Yes you can do this too!

```
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, then `t2` is nullptr?
    - No! There is no such thing as a *null reference*.
    - Raises an exception: `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 &textother = dynamic_cast<const Text&>(other);
   // If other is not a Text then it throws
   if (this == &textother) return *this;
   Book::operator=(other);
   topic = textother.topic;
   return *this;
}
```

- You can use dynamic casting to make decisions based on an object's **R**un**T**ime **T**ype **I**nformation (RTTI).

```
void whatIsIt(shared_ptr<Book> b) {
   if (dynamic_pointer_cast<Comic>(b))
      cout << "Comic";
   else if (dynamic_pointer_cast<Text>(b))
      cout << "Text";
   else if (b)
      cout << "Ordinary Book";
   else
      cout << "Nothing";
}
```

- What would we say about the coupling of this with the Book hierarchy?
    - Highly coupled ⇒ might indicate a bad design.
- Suppose you want to create a new type of Book, what changes would you need to make?
    - must update `whatIsIt` to add a new clause.
    - must find and fix all uses of dynamic casting before your code will work properly
    - easy to misuse, error prone
    - Better to use virtual methods!
- Are all uses of dynamic casting indicative of bad design?
    - No. `Text::operator=` (previous) does not require updates, etc - only needs to compare with its own type (not everything in the hierarchy). Why?


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

- Fixing `whatIsIt`
    - Try to create an interface function that is uniform across all Book types.

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

- What if the interface isn't uniform across all types in the hierarchy?
- Inheritance and virtual methods are well-suited when
    - there is an unlimited number of specializations of a basic abstraction
    - each follow the same interface
- Adding a new subclass, for a new specialization, is easy.
- BUT, what if we have the opposite case:
    - there is a small number of specializations, all are known in advance and they are unlikely to change
    - the different specializations may have very different interfaces
- What do we need to do to add a new, unexpected, subclass?
    - Must rework existing code to accommodate new interface.
        - you weren't expecting to add a new subclass so you should expect to put in extra effort.

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

- Interfaces are not uniform - a new enemy means a new interface
    - unavoidable work.
- We could regard the set of enemy classes as fixed and maybe dynamic casting on enemies is justified.
- BUT, in this case, maybe inheritance isn't the correct abstraction mechanism to use.
- **Variant** If you know that an `Enemy` will only be a `Turtle` or a `Bullet` and you accept that adding new `Enemy` types will require widespread changes anyway, then consider:

```
import <variant>;
// An Enemy is either a Turtle or a Bullet
// old-style: typedef variant<...> Enemy;
using Enemy = variant<Turtle, Bullet>;
// Check what type e is:
if (holds_alternative<Turtle>(e) {
   cout << "Turtle";  // True if e is a Turtle
}
else ...


// Extracting the value:
try {
   Turtle t = get<Turtle>(e);
   // C++17 throws on error: bad_variant_access
   //use t ...
}
catch (std::bad_variant_access &) {
   // It wasn't a Turtle
}
```

- A variant is like a union but it's *type-safe*.
    - attempting to store as one type and fetch as another will throw an exception
- If a variant is left uninitialized, what happens? `std::variant<Turtle, Bullet> e;`
- The first option of the variant is default-constructed to initialize the variant.
- What if the first option does not have a default constructor? Compile error! (as we would expect)
- Options:
    1. Make the first option a type that has a default ctor.
    2. Don't define uninitialized variants.
    3. Use `std::monostate` as the first option. This creates a "dummy" type that can be used as a default;
        - i.e. can be used to create an "optional" type.
        - `variant<monostate, T> // = "T or nothing"`
- Also, `std::optional<T>` which contains a value or does not contain a value. Can convert to a Boolean T/F.
- From ChatGPT: Here's a quick overview of `std::optional<T>`:
    - Usage: `std::optional<T>` can either contain a value of type `T` or it might be empty (i.e., not contain a value).
    - Safe Access: Before accessing the value inside an `std::optional`, you can (and should) check if it actually contains a value using its `has_value()` member function or by using it in a boolean context (e.g., `if(opt)`).

- Value Retrieval: If the `std::optional` contains a value, you can retrieve it using the `value()` member function. Alternatively, the `*` operator and `->` operator can be used as if the `optional` object is a pointer.
- Error Handling: `std::optional` is especially useful for functions that might fail to produce a result. Instead of returning a special error value or throwing an exception, a function can return an empty `std::optional` to signify the absence of a result.

```
#include <iostream>
#include <optional>

std::optional<int> divide(int numerator, int denominator) {
    if (denominator == 0) {
        return {};  // return an empty optional
    }
    return numerator / denominator;  // return a valid result wrapped in an optional
}

int main() {
    auto result = divide(10, 2);
    if (result) {
        std::cout << "Result: " << *result << std::endl;
    } else {
        std::cout << "Division by zero!" << std::endl;
    }

    result = divide(10, 0);
    if (result) {
        std::cout << "Result: " << *result << std::endl;
    } else {
        std::cout << "Division by zero!" << std::endl;
    }

    return 0;
}
```

Lecture 23

- Reminder: 1 more class - Tuesday Aug 1
- Template Functions

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

int f() {
        int x = 1, y = 2;
        int z = min(x, y); // T = int
}
```

- C++ can infer the type from `x, y` or you can also explicitly state it.
    - must need to if the compiler can't figure it out, eg. function without args

```
z = min<int>(x, y);
min('a', 'c'); // T = char
min(3.0, 1.0); // T = double
```

- What type can `T` be? (for `min`)
- To compile, needs operator < for `T`
- Any types that implements operator <
- Recall

```
void foreach(AbstractIterator &start, AbstractIterator &finish, int (*f)(int)) {
        while (start != finish) {
                f(*start);
                ++start;
        }
}
```

- What do we need for this to work? `!=, *, ++`
- We can generalize to a template function as long as specified type has `!=, *, ++` and `f` can be called as a function.

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

- Now `Iter` can be any class that has `!=, *, ++` such as raw pointers, etc.

```
void f(int n) {cout << n << endl;}
int a[] = {1,2,3,4,5}
foreach(a, a + 5, f); // print out items of array
```

- The `algorithm` library (STL)
  - A suite of template functions
  - `import <algorithm>`
  - many over iterators
  - Ex. `for_each`

```
template<typename Iter, typename T>
Iter find(Iter first, Iter last, const T &val) {
        // returns the iterator to first element matching val or last if not found
        while (first != last && *first != val) ++first;
        return first;
}
```

- `count` ~ like `find` but returns # of occurrences of val

```
template <typename InIter, typename OutIter>
OutIter copy(InIter first, InIter last, OutIter result) {
        // copy elements from [first, last) to [result, result + (last - first))
        // starting at result
        while (first != last) {
                *result = *first;
                ++first;
                ++result;
        }
        return result;
}
```

- Note: does not allocate new memory, so output container must have space available
- Eg.

```
vector v{1,2,3,4,5,6,7,8};
vector w(4); // != w(4,0) != w{4}
copy (v.begin() + 1, v.begin() + 5, w.begin());
```

- w is now `{2,3,4,5}`

```
template<typename InIter, typename OutIter, typename Fn>
OutIter transform(InIter first, InIter last, OutIter result, Fn f) {
        while (first != last) {
                *result = f(*first);
                ++first;
                ++result;
        }
        return result;
}
```

- Eg.

```
int add1(int n) {return n+1;}
...
vector v{2,3,5,7,11};
vector w(v.size());
transform(v.begin(), v.end(), w.begin(), add1);
```

- w is now `{3,4,6,8,12}`
- How virtual methods work?
- Consider 2 classes

```
class Vec {
                int x, y;
        public:
                void f(){...};
};

class Vec2 {
                int x, y;
        public:
                virtual void f(){...};
};
```

```
Vec v{1,2};
Vec2 v2{1,2};

// sizeof(int) => 4 bytes
// sizeof(v) => 8 bytes: 2 * sizeof(int) - no space for a method f; compiler turns method into an ordinary
function and store them seprately from objects
// sizeof(v2) => 16 bytes: pointer?


Book *pb = new Book/Text/Comic;
// or
auto pb = make_unique<Book/Text/Comic>();
pb->isHeavy();
```

- If `isHeavy()` is virtual - chance of which version of `isHeavy()` to run is based on the type of the actual object, which we won't know at compile time. The correct `isHeavy()` must be chosen at runtime.
- How? For each class with at least one virtual method, the compiler creates a table of function pointers (the vtable usually) and instances of the class have an extra pointer (the vptr) that points to the class vtable

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

C c,d;
```

- table inserted for above example
- table inserted for Book, Text
- Calling a virtual method
    - follow the vptr to the vtable
    - fetch the ptr for the actual method from vtable
    - follow the function ptr and call the function
    - this happens at runtime.
    - virtual function calls incur a small amount of overhead
    - classes with at least one virtual function adds a vptr to the object
- Concretely how is an object laid off?
    - compiler dependent

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

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

Lecture 24

- Compiler dependent layout

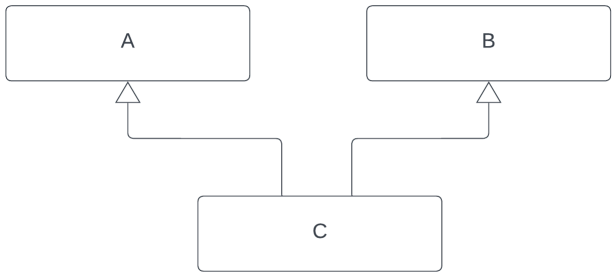
```
class A {
        int a,c;
        virtual void f();
};

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

A
vptr
a
c
B

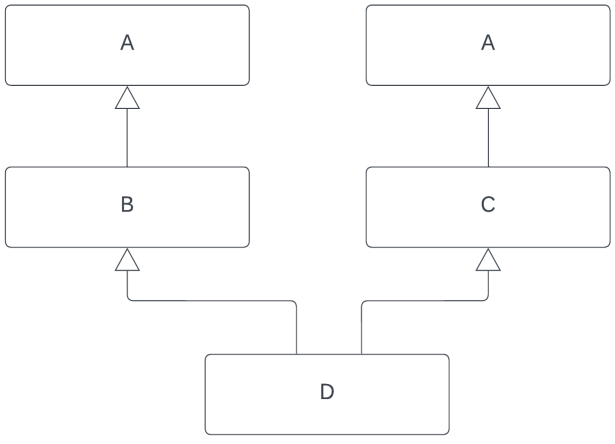
B
vptr
a
c
b
d

- Multiple Inheritance
  - A class can inherit from more than one class

```
class A {
        int a;
        ...
};

class B {
        int b;
};

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



- Challenges: suppose `B` and `C` both inherit from `A`



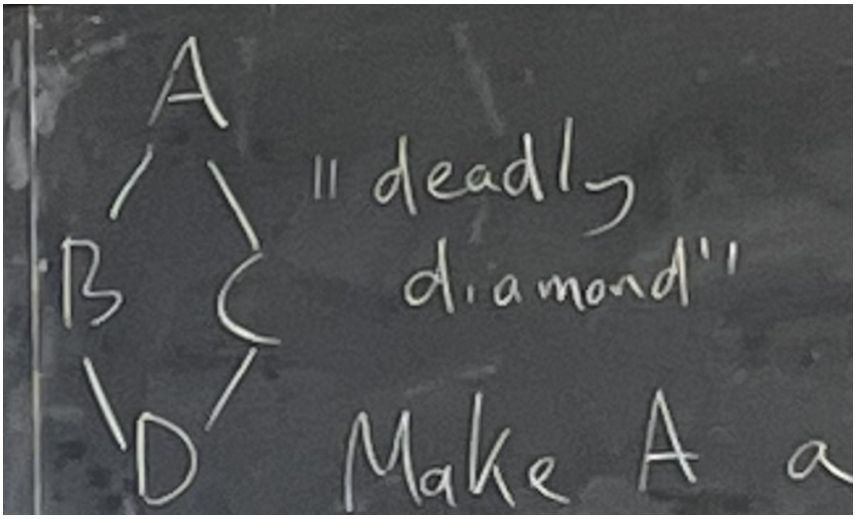

```
class D : public B, public C {
        public:
                int d;
};
```

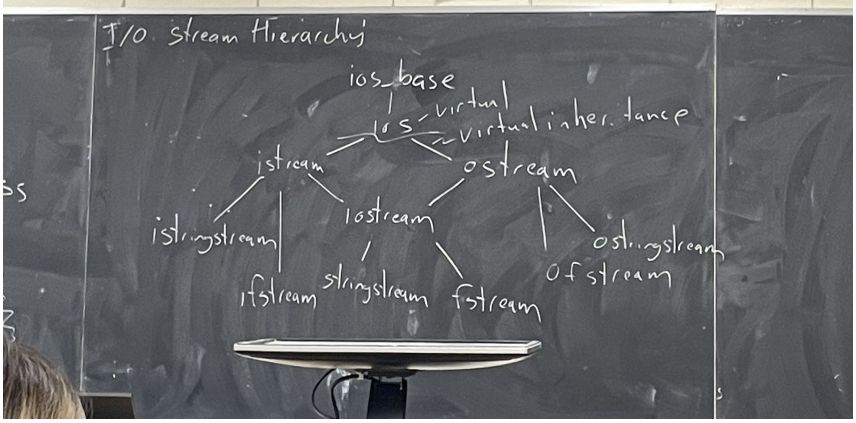
```
D dObj;
dObj.a = 2; // which 'a' is this?
```

- It's ambiguous
- We need to specify `dObj.B::a` or `dObj.C::a`
- But if `B` and `C` both inherit from `A` should there be one `A` part in `D` over two?
    - Should `B::a` be the same as `C::a` only one `a`
- "deadly diamond" Make `A` a virtual base class (only one copy) and empty virtual inheritance

```
  A
/   \
B   C
 \ /
  D
```

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





- What would the layout look like? What do we want?
- Same pointer should look like an `A*, B*, C*, D*`

D's Fields
vptr

D's Fields
A's fields
B's fields
C's fields
D's fields

- If we have a ptr to `A`, `B`, `D` these look okay. But a ptr to `C` does not look like a `C`
- But first four rows does not look like a `C*`

C's Fields
vptr
A's fields
C's fields

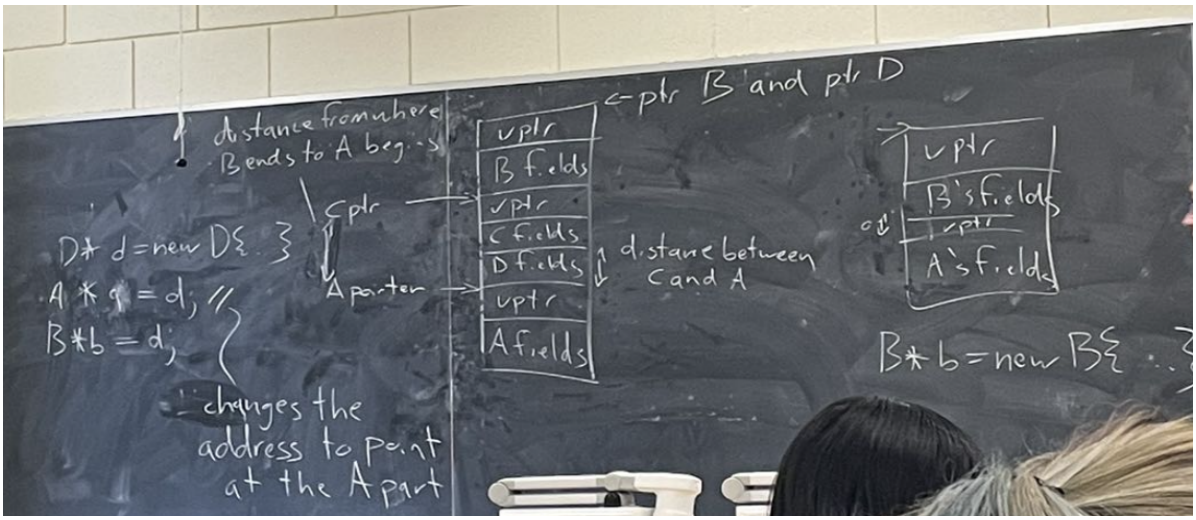
- The actual `D`'s fields should be

D's Fields
vptr (ptr B or ptr D)
B fields
vptr (C ptr)
C fields
D fields
vptr (A ptr)
A fields

- [C ptr, A ptr) is Distance from `B` to `A`; [D ptr, A ptr) is Distance from `C` to `A`
- Consider `B b;`
    - using previous, what do we expect `B` to look like?
        - `b` has no `C` part or `D` part
        - class `B` does not even known about clss `C` or class `D`

B's Fields
vptr
B's fields
vptr
A's fields

- distance between `B` part and `A` part is 0
- Distance from a class to its parent is not constant
    - varies with the runtime type of the object
    - location of the base class part of the object is stored in the vtable (this is where virtual inheritance gets its name)
- Also the diagram doesn't look like all of `A, B, C, D` simultaneously but has slices that resememble `A, B, C, D`



- so ptr assignment among `A, B, C, D` pointers changes the address stored in the ptr

```
D *d = new D {...}
A *a = d // changes the address to point at the A part; only points to a different place in memory but the base structure does not change
```

- static-cast and dynamic-cast
  - will change the memory address (only multiple inheritance)
- reinterpret-cast
  - will not change the memory address

Final Review

- Exercise: write a program that uses stringstream to convert between integers and strings

```
string s = "5";
stringstream ss{s};
int i;
while (ss >> i) {
        ...
}
```

- Pass-by-reference: creates an alias to the parameter.
- A literal cannot be passed by reference since it is not an lvalue (something that has an address).
- A constructor with all parameters defaulted is still a default constructor.
- If we define our own constructor(s), we lose both the implicitly-declared (i.e. compiler- provided) default constructor and list initialization (for aggregates).
- A default destructor is provided for us by the compiler. This destructor calls the destructors for all data fields that are objects. Note that it will not call delete on data fields that are pointers, because pointers are not objects.
- **Destructors do not have a return type.**
- If a copy constructor (i.e. constructor with argument of const lvalue reference) and a move constructor are both present in a class definition, the compiler will decide which constructor to call based on the value category.
- If a move constructor is not available, the copy constructor will always be called regardless of what kind of value you pass into the constructor call.
- A non-const static variable must be initialize outside of a class.
- Since static members are not associated with any object instances of the class, it doesn't come with a this pointer, and thus it cannot access any non-static fields.
- Writing `= default` after the declaration will allow the compiler to generate a comparison operator that compares all the fields in *lexicographical* order.
- If a class A has a pointer to an instance of class B, you **cannot** know if the relationship is composition or aggregation without looking at documentation or code.
- Cannot Call via Base Pointer: If you have a method that's unique to `C` and you have a pointer of type `A*` pointing to a `C` object (`C` is a derived class of `A`), you cannot directly call that method through the `A*` pointer, since the method is not part of `A`'s interface.
- Declaring a method virtual means if we override it in a subclass, the subclass version of the method will be called through polymorphic pointers or reference. If we do not override the method, the definition in the *closest related ancestor* will be used.
- Note: dynamic casting only works on classes with at least one virtual method.
- If you create a constant instance of an object, then it can only call its const methods.
- **The following contents are from ChatGPT about some questions I have during the review, please check if they are truly accurate (at leats it makes sense to me):**
- If the Derived object is on the stack (i.e., it's an automatic variable, not dynamically allocated), the derived class's destructor will always be called, followed by the base class's destructor, regardless of whether the base class destructor is virtual or not. This is because the compiler knows the exact type of the object (it's not being accessed polymorphically via a pointer or reference to the base class).
- If you destroy a derived object through a pointer (or reference) to the base class, and the base class destructor is not virtual, then only the base class's destructor will be called. The derived class's destructor will be skipped, leading to potential resource leaks and undefined behavior. This is why it's crucial to declare base class destructors as virtual when working with inheritance.
  - If the base class destructor is virtual, and you destroy a derived object through a pointer (or reference) to the base class, both the derived class's destructor and the base class's destructor will be called, in that order.
- A private virtual method in a base class can be overridden in a derived class. However, there are some nuances to be aware of:

    - *Overriding*: The derived class can provide its own implementation of the private virtual method from the base class. This is valid because the virtual mechanism works based on the method's signature, not its access specifier.
    - *Direct Access*: Even though the derived class can override the private virtual method, it cannot directly call it using the method's name because it's private in the base class. However, if the base class provides a public or protected interface (like another method) that internally calls this private virtual method, the overridden version in the derived class will be invoked.
    - *Accessibility*: You don't need to maintain the same access specifier in the derived class when overriding. For instance, a private virtual method in the base class can be overridden as public or protected in the derived class, but this doesn't change the fact that the original method in the base class remains inaccessible directly.
- To access a method in the derived class through a base class pointer, the method must be declared in the base class. The method must be public in the base class to be directly accessible through a pointer or reference of the base class type from outside the class.
    - However, if you're accessing the method from within another method of the base class (or a derived class), then it doesn't necessarily have to be `public`. It could be `protected` or even `private`, depending on the context.
- When you have a base class and one or more derived classes:
    - Deleting through a Derived Class Pointer:
        - If you have a pointer of the derived class type pointing to an object of the same derived class type and delete the object through this pointer, the destructor of the derived class will be called. Subsequently, the destructor of the base class will also be automatically called. No matter if the destructor in base class in virtual or not.
    - Deleting through a Base Class Pointer:
        - If the base class's destructor is not virtual: Deleting an object of a derived class through a pointer of the base class type will only call the base class's destructor. The derived class's destructor will not be invoked, which can lead to resource leaks or other issues.
        - If the base class's destructor is virtual: Deleting an object of a derived class through a pointer of the base class type will call both the derived class's destructor and the base class's destructor in that order.