Strings

- Array of chars(char * or char[])
- Easy to corrupt (overwrite '\0')
- must explicitly allocate memory

In C++

```
"import <string>;"
  std::string -> type
  - it grows as needed
  - safer to manipulate

// C++
  string s = "Hello";
  s = "Hi";
```

Strings Operations

```
- Equality (s1 == s2, s1 != s2)
- Comparison: s1 <= s2
- Length (s.length()) //O(1)
- individual chars (s[0], s[1])
- concat: s3 = s1 + s2
          s3 += s4
- cin >> s; //!!! read a word
              - ignores leading white space
              - stops at next white space
- getline(cin(stream), s(string))  // read in "a line"
```

Example in VSCODE

Streams are **abstraction**

Abstraction:

- Wrap an interface of getting and putting items to keyboard and screen

Files (file streams is an abstraction)

- Read/write from/to a file instead cin/cout

```
std::ifstream //a file stream for reading
```

```
std::ofstream // a file stream for writing

Ex: in C
#include <stdio.h>
int main () {
  char s[256]; // Hoping no words is larger than 255 chars
  FILE *f = fopen("file.txt", "r");
  while (1) {
    fscanf(f, "%255s", s);
  }
}

in C++:
import <iostream>;
import <fstream>;
using namespace std;

int main () {
  ifstream f{"file.txt"}; // Declaring and initializing opens the file
  string s; // declare string variable; no concerns about size of the words
  while (f >> s) { //f getting from s
    cout << s << endl;
  }
} // not closing the file because of "f". file is closed when it is out of scope
```

Command Line Arguments:

```
./program abc 123

//To access these args:
int main(int argc, char* argv[]) {
  // argc: # of arrays given to the program - including program name
  // argb -> array of char *s, i.e - C style Strings
  // argv[c] - name of program
  // argv[1] - 1st arg
  // argv[2] - 2nd arg
  ..
  // argv[argc] - Null Pointer
}
```

Stack

argv: _ _ _ _

Index 0: ./program\0

Index 1: a b c /0

Index 2: 1 2 3 /0

Index 3: NULL

!!! **WRONG**

```
if (argv[1] == "abc") // Comparing pointer locations instead of String contents!!!!
```

Advice: Convert the args to **std::String**

Before use:

```
int main(int argc, char **argv) { // char**argv Equivalent to *argv
  for (int i = 0; i < argc; ++i) {
    String arg = argv[c];
    if (arg == "abc") {} //WORKS!
  }
}
```

Ex: Add all integers given as args on the cmd Line:

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

Default Function Parameters

```
void printSuiteFile (string fileName = "suite.txt") {
  ifstream file {fileName};
  string line;
  while (getline(file, line)) cat << line << endl;
}

// printSuiteFile("abc.txt") -> Read from abc txt
// printSuiteFile() -> Read from file - suite.txt
// Notes: optional parameters must be last
```

Who's responsibility is it to provide the default argument? Who writes suite.txt into the stack frame of arguments?

- The caller sets up the argument, not the function itself
- printSuiteFile() is replaced with printSuiteFile("suite.txt") where it is found

void f(int x, string s = "Hello")

Int main() { f(5); }

stack: _ _ main_ (see WeChat visualization)

The defaulted function (f)

- does not know what will be passed in
- It **cannot** detect the difference between uninitialized memory and other strings
- Caller of f must provide **all arguments** to f in advance

Note: Default parameters are part of the **declaration**, not necessary definition

```
in Header (interface) file:
void f(int x; string s = "Hello");

// in .c file
void f(int x, string s) {}
```

Default parameters are given in the **interface (header file in this case)**, **not** the implementation (.c / .cc file)

Overloading

- if I want a function to negate ints and bool
    - in C:

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

- in C++:

```
int neg(int x) {return -x;}
bool neg(bool b) {return !b;}
// neg(5) -> -5
// neg(true) -> false
```

- Compiler chooses which **overload** of the function to use based on the **#** and the **types of the args** at compile line.
  - seen this in the case of operators: ==, <<, >>, +
  - We cannot overload based on the **return type alone**
  - can't have same args in overloading functions. s.t. f(int x); f(int x)

Structs

- Structs also exist in C++, just as they existed in
  - Ex: Linked list node:

```
struct Node {
  int data;
  Node* next;
}; // dont forget ;
```

In C: this would be Struct Node*

in C++: **omit** the struct

Recall: why is this invalid?

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

- What is the size of the struct?
  - we need to solve the following:
    - **sizeof(Node) = sizeof(int) + sizeof(Node)** X=4+X

- The previous (*Node) works because ptrs are if fixed size;
  - 8 bytes no matter what you point at

Constant

```
const int max_grade = 100;
// Any attempt to modify this variable wont compile

const Node n {5, nullptr};
// n's data cannot be changed and n's next field cannot be changed
```

In C:

```
// The null ptr is inficated via NULL or 0
// wont do this in C++!!!
```

- in C++:
  - Typically in C - there is the following line imported via a library
    - #defines NULL 0 -> replaces all instances of NULL with 0
    - consider following scenario:

```
void f(int x);
void f(Node* p);

1. calls f(NULL) - calls int version of f instead of pounter version


// nullptr is a special type that can be automatically converted to any other ptr
type
2. f(nullptr) -> calls ptr version of f
```

Parameter Passings:

```
void inc(int x) {++x;}

int main() {
  int x = 5;
  inc(x);
  cout << x <<endl; // output 5 because we made a copy of 5 and call the function (AKA
pass by value semantics)
}
```

Pass-by-value semantics: Copy arguments into the function instead of using orginal.

```
void inc(int *p) {++*p;}

int main() {
  int x=5;
  inc(&x); // pass by reference
  cout << x << endl;
}
```

- If we want to mutate an argument and change the value outside the function
    - pass a pointer

C++ has another pointer-like type: **Reference**

References:

```
int x = 5;
int& z=x; // any changes to z will be reflected in x (lvalue)
// z is an "alias" to the value of x
// anytime you see z, simply imagine replacing it with x

++z;
cout << z <<" "<<x<<endl;
// z=x=6
```

- References are somewhat like **const ptrs** with automatic dereference.
- We cannot change what a reference is aliasing
- When does & mean reference vs when it is address-of?
    - & in a **type** means references, **in an expression it means address-of**
    - f(int& x) // int&: reference

    - Int* y = &x; // expression: address-of
- Things you **cannot** do with references:
    - Leave them uninitized, must alias a variable at all times. (**MUST BE INITIALIZED**)

```
int &z; //INVALID: WONT COMPILE
```

- Lvalue references must always be initialized with lvalues
- Lvalue: a value you may take the address-of
    - Rvalue below:

```
int& z = 5; //cannot take the address of 5, it is not an lvalue - it is an rvalue
```

    - lvalues have a "name" - which tells you where in memory the variable is stored!
        - rvalue again:

```
int& z = x + y; // rvalue - tempaorary variable
int& z = f(); // wont compile
```

- We cannot **create a pointer** to a **reference**:
    - read from right to left
        - int&*x; // x *wont* compile, ptr to a reference
        - Int*&x; // reference to a ptr - this is *allowed*
- Cannot create a **refernce** to a **reference**

```
int&& x; //This is not a reference to a reference;
         // this is a rvalue reference
```

- Cannot create an array of references

```
int&arr[3] = {x,x,x}; // X wont compile
// Reference Assignment: Once a reference is initialized, it cannot be reassigned
to refer to a different object. If you had an array of references, you wouldn't
be able to change what they refer to after initialization.
```

    - what are references useful for?
        - Most commonly: Use as **args** to a function

```
// we saw
inc(int x) and inc(int* p) {++*p;}
```

- with references:

```
void inc(int& x) {++x;} // automatically dereference
x = 5;
inc(x);
cout<<x<<endl; //6
```

- How does int x; cin>>x work?

```
cin >> x // actually just a fn call
// how it is implemented in stardard library
istream& operator>>(istream& in, int& x) {
  // read from in into X
  return in;
}
```

- Is there a good reason why we take in and return istream& instrad of an istream?
  - **pass by value** can be an expensive operation
  - copying memory from caller's stack frame into called stack frame
  - ex:

```
struct ReallyBig {...};
void f(ReallyBig rb) {...};// no changes reflected, pass by copy (copy all
mempory in rb -> expensive)
void g(ReallyBig& rb) {...} // Changes are reflected, not expensive, this is
fast- 8 bytes copied
void h(const ReallyBig& rb) {...} //FAST; rb wont be changed; should be a good
choice
```

- When designing a function with argiments
  - **pass-by-const-reference** should be first choice
  - use pass-by-ref only if you **need** to changes reflected outside function call

```
pass-by-value: 2 scenarios;
1) Arg is small-pointer size of smaller (< 8bytes)
2) void a(const ReallyBig& rb) {
  ReallyBig temp {rb}; // just use pass-by-value instead, copies for you ???
}
```

```
in istream& operator>>(istream& in, int&x);
// pass cin via reference to avoid expenses of copying
// (streams have copying disabled - copying does not compile)
void f(int&x); // can only pass by lvalue
void g(const int&y); // can pass both lvalue and rvalue: g(z), g(5) workss
```

- **const** - lvalue references like in g(ex above) can bind to rvalues (tempraries) because we know they won't be changed
    - Compiler create a tempory location and store the temporary (ex: g(5) where 5 is the temporary memory) inside

Dynamic Memory

- In C:

```
int *p = malloc(n * sizeof(int)); //dynamic array of int
... //maybe resize...
free(p); // give memory back to OS
```

- In C++: use new/delete

```
Node* np = new Node {5, nullptr};
...
delete np; // give back memory to OS; follows np's memory address and heap and delete
it, but in stack, it changes nothing
```

- Important!
    - All local variables rewrite space on the stack
    - use new to acquire memory from the heap
    - calling delete returns memory to OS to be reused,
    - does not affect value of pointer at all
    - Stack allocated vars; // automatically deallocated when they leav scope
    - Heap allocated memory exists until delete is called
    - A memory leak occurs when memory that is allocated with new is new deleted.
- C++: dynamic arrays;

```
int* arr = new int[10]; // array of 10 ints on heap;
                        // arr points to first element in the arr
...
delete[] arr; // frees a dynamically allocated array
```

- Always pair **new + delete** new[] with delete[]
- Do not mix and match or else undefined behaviour

Lecture 5

We have seen for args: pass-by-value, pass-by-ptr, pass-by-ref

Which is the best option for returns? //

```
Node getMeANode(int n) {
  return Node {n, nullptr}; // return-by-value
}
```

This works - make a copy from getMeANode's stack frame into the caller's frame: Too slow?

not working version:

```
Node* getNodeptr(int n) {
  Node x {n, nullptr};
  return &x; // wrong: it returns the pointer into the stack frame; returing a dangling
pointer;
             // x will be cleaned up when fn returns, pointing at memory we no longer own.
}
```

working version:

```
Node* getNodeptr(int n) {
  Node* p = new Node {n, nullptr};
  return p;
}
```

This is fast and works **BUT** user must remeber to **delete** heap allocated memory, or else **memory leak**

Return by Ref:

```
Node& getNodeptr(int n) {
  Node x{n, nullptr};
  return x; // still not working because acess to the alias of the node, we still lose x
            // dangling ref - same problem as ptr example
}
```

- Returning by reference is rare.
- Returning a reference to stack allocated variables will cause undefined behaviour.
- Only use return-by-ref if returning a ref to a non-local variables. Eg:

```
istream& operator>>(istream& in, int& x) {
  ...
  return in; // works because in is not local variable
}
```

- Advice: Usually, **return-by-value**: easy to use, not as slow as it may seem (more semantics)

Operator Overloading

- Defined the meaning of operators acting on our own types

```
struct vec {
  int x, y;
};

vec operator+(const vec& v1, const vec& v2) {// const reference of v1; aliases v1
  vec r{v1.x+v2.x, v1.y+v2.y};
  return r;
}

vec operator*(int k, const vec& v) {
  return {k*v.x, k*v.y}; // vec could go here to explicityly construct a vec, obied,
but assumed by return type
}

vec z = w * 2; //This doesnt mach our first overload - order of args is wrong

vec operator*(const vec& x, int k) {
  return k*v; // int * vec from the previous function. calls the other operator*
}
```

Example: Overloading << and >>

```
struct grade { // support: grade g; cin>>g; cout<<g;
  int n;
};

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

ifstream file {"file.txt"};
grade g;
file>>g; // works

ostream& operator<<(ostream& out, const grade& g) {
  return out<<g.n<<'%';
} // generally, no need for operator<<, it would force users to print newlines
```

Separate Compilation

Point: Speed up compilation times, by only recompiling what we need to

Recall:

- Interface files (think .h files): provide declarations
- Inplementation files (think .c): provide definitions

```
// interface (vec.cc)
export module vec; // indicates an interface file
export struct vec z { // Anything marked export clients may use
  int x,y;
}
export vec operator+(const vec& v1, const vec& v2);

// Client code (main.cc)
import vec;
int main() {
  vec v{1,2};
  vec w = v+v;
}
```

```
// Implementation file: (vec-implementation.cc)
module vec;
vec operator+(const vec& v1, const vec& v2) {
  return {v1.x+v2.x, v1.y+v2.y};
}
```

Recall:

- we can repeatedly declare structs/functions, but we may only define once.
- Interface files: export module x;
- implementation files: module x;

To create an object file (.o file):

```
g++20m vec.cc -c // -c means compile only
                 // creates vec.o
```

with c++20 modules, object files must be created in **dependency** order

- First compile **interface files**, then **implmentation** / client files!!! (ORDER BELOW)
  - vec.cc
  - vec-impl.cc
  - main.cc

```
g++20m vec.o vec.impl.o main.o -o program //links object files
```

Building dependencies with Make in progress.

```
g++20m *.cc -c // fail on some which are out of order
// can write g++20m *.cc without -c
g++20 *.cc -c // create remaining .o files
g++20m *.o -o program // linking
```

CLASSES!!

- we can put functinns inside our structs
  - // Student.cc

```
export module Student;
export Struct Student {
  int assns, mt, final;

  float grade();
}
```

- // Student-impl.cc

```
float Student::grade() { // Student:: means "scope resolution operator"
  return 0.4*assns + 0.2*mt + 0.4*final;
}

// C::f - inside of the Class C - define, member function f
```

```
Student s{60, 70, 80}; // s - is an object, instantiation of a class
s.grade(); // returns 60*0.4+70*0.2+80*0.4

// Student - this is a class - struct with functions defined inside it.
// grade() function calls member functions / method
```

- Inside Student::grade we use variables assns, mt, final
- These variables are bound to the object the method is called on

```
s.grade() // assns is s.assns
          // mt is s.mt


// When we call s.grade(), there is a hidden parameter that is used to access
assns, mt, final
this -> a pointer to the object the methods was called.

s.grade() // inside grade, this == &s



// we could write:
export struct Student {
  int assns, mt, final;
  // method
  float grade() {
```

```
// provide mthod defitions inside of class - but generally best to separate into
interface/implementation file
    return this->assns * 0.4+ this->mt * 0.2 + this->final * 0.4;
  }
};
```

Initializing objects

```
Student S {60, 70, 80};
- To better control initialization, write a constructor (ctor) will get called whenever a
Student object is created.

Struct Student {
  Student(int assns, int mt, int final) {
    // name of the method is the name of the class
    this->assns = assns;
    this->mt = mt;
    this->final = final;
  }
};

// using this-> to differ between arg and field

Student s{60, 70, 80};
Student *p = new Student{60, 70, 80};
//Invokes a ctor call - passed as args to ctor method.
// If we dont write a ctor - we get C-style field-be-field initialization

Student S = Student{60, 70, 80};
Same as Student S{60, 70, 80};
// Benefit of ctors: write our own logic - use them as functions. eg: default params:
Sturct Student { // default constuctor
  Student(int assns=0, int mt=0, int final=0) {
    this->assns = assns;
    this->mt = mt;
    this->final = final;
  }
};

Student s(60, 70, 80);
Student newkid; // newkid sets to 0
Student toby{40, 50}; // final=0 for default
```

```
Student newKid; // calls our 0 -arg ctor
A ctor that may be called with 0 args;
This is a """default ctor""". If you do not write a default ctor - compiler provides one.
// As soon as you write a ctor - compiler procided default ctor goes away
```

Compiler provides default ctor:

- Primitive types - leave them uninitialized //原始type
- Classes/ Structs - calls their default ctor.

1. **Primitive Types**:
   - Examples of primitive types include `int`, `float`, `double`, `char`, etc.
   - If you create a variable of a primitive type without initializing it, the value of that variable is left uninitialized in the case of local (stack) variables. This means the value is essentially whatever garbage data was in that memory location. For global or static variables, they are zero-initialized by default.

     ```
     cppCopy code
     int a;    // uninitialized value for local variable
     static int b; // zero-initialized
     ```

2. **Classes/Structs**:
   - If you define a class or struct without a default constructor, the compiler will implicitly define a default constructor for you, as long as there isn't any user-defined constructor present.
   - The compiler-provided default constructor will do the following:
     - It will call the default constructors for any member objects.
     - It will leave primitive type members uninitialized (just like the rule for standalone primitive types).

     For instance, consider the following example:

```
cppCopy code
struct Point {
    int x, y;
};

class Circle {
    Point center;
    double radius;
};
```

The default constructor provided by the compiler will be called. This constructor will call the default constructor of `Point` (which in this case does nothing and leaves `x` and `y` uninitialized) and leaves `radius` uninitialized as it is a primitive type.

```
Struct vec {
  vec(int x, int y)  {
    this->x = x;
    this->y = y;
  }
};
// vec v; // should call default ctor but none exist - doesnot compile.

Struct Basis {
  Vec v1, v2;
};
// Basis b; what happened?
// vec is the default ctor. (calls default constructor)
// wont compile, we cannot default construct v1 and v2 since it needs two parameters

Struct Basis {
  vec v1, v2;
  Basis() { // does not compile
    v1 = vec{0, 1}; // if write sth like this, they should have been initialized, but they
are not
    v2 = vec(1, 0);
  }
}; // doesnt work because v1 and v2 must have values initialized by the time the function
starts to run;
//v1 and v2 are being assigned values inside the constructor body. This means they are
default-constructed first and then assigned new values inside the constructor. If the vec
type doesn't have a default constructor, this would result in a compilation error.
```

```
Instead, you should use an initializer list to directly initialize v1 and v2:

Struct Basis {
  vec v1, v2;
  Basis() : v1{0, 1}, v2(1, 0) {
  }
};
// Now, v1 and v2 are directly initialized with the provided values when an instance of Basis is created. This is more efficient and avoids potential issues if vec does not have a default constructor.
```

Steps of object creation

1. Space is allocated
2. save for later
3. Fields are initialized
4. Ctor body runs

MIL - Member Initialization List

- Let us provide values for fields in str

```
Student::Student(int assns, int mt, int final):
  assns{assns}, mt{mt}, final{final} {...}
//Field names{values} -> step 3    {ctor body} -> step 4

//Back to the pre example
Struct Basis {
  vec v1, v2;
  Basis(): v1{0, 1}, v2{1,0} {}
};
```

- More generally, Overload

```
Struct Basis {
  Vec v1, v2;
  Basis(): v1{1,0}, v2{0,1} { }
  Basis(const Vec& v1, const Vec& v2): v1{v1}, v2{v2} {} //copy ctor
}
However, you are copying their values into the v1 and v2 member variables of the
Basis object. Once those values are copied, you can modify the member variables, but
not the original objects passed to the constructor.

// Provide defaults for MIL
Struct Basis {
  Vec v1{0, 1}, v2{1,0}; // default values, used if nothing is specified in MIL
  Basis() { }
  Basis(const Vec& v1, const Vec& v2): v1{v1}, v2{v2} {} //copy ctor
}
```

- Fields in the MIL will always be **initialized** **in the order** they are declared in the class.
- MIL can also be **more efficient** than using ctor body

```
Struct Student {
  int assns, mt, final;
  string name;
  Student(int assns, int mt, int final, const string& name) {//name set to empty
string
    this->assns = assns;
    ...
    this->name = name; // overwriting ""(empty string) with name
  }
};

Use MIL:
Student(int assns, int mt, int final, const string& name):
  assns{assns}, mt{mt}, final{final}, name{name} {...} //directly set name to name.
more efficient!!

// MIL is required: Object fields want default ctors
                //  const fields and reference fields

//1. Object fields want default actors: It's often the case that object fields (i.e.,
fields that are instances of classes) have default constructors that allocate
resources or set initial states. By using the MIL, you can avoid calling the default
constructor only to immediately overwrite the value inside the constructor body. This
can be more efficient.
```

```
//2. const fields: In C++, if a member variable is declared const, it can only be initialized and cannot be assigned a value afterward. Therefore, for const member variables, you must use a MIL. Otherwise, you won't be able to set their value.

//3. Reference fields: Like const fields, reference fields in C++ must be initialized when an instance is constructed and cannot be reassigned later. Again, you must use a MIL to set their value.


// using MIL as much as possible!
```

- Copy ctor is running
  - Purpose: create 1 object as the object of another
  - Ex: Student s {60,70,80}
    - Student r{s}; same as Student r = s; // copy ctor is running

- Compi;er gives you the following if you dont write them;
  - DEFAULT CTOR, DESTEUCTOR, COPY CTOR,
  - COPY assignment operator, move ctor, move assivnment operator

Copy Ctor for Student

```
struct Student {
  int assns, mt, final;
  Student(const Student& other): //const - dont make a copy inside of function, not need to change "other"
    assns{other.assns}, mt{other.mt}, final{other.final} {}
};

This is a copy ctor - replicates compiler provided copy ctr which inst sets this fields to other;s fields
What is a case where compiler provided copy ctor doesnt work?

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

```
Node* n = new Node {1, new Node{2, new Node {3, nullptr}}};
Node p {*n};
Node* q = new Node {*n};

// Stack:              Heap:
n ----------------> "1"->"2"->"3""X"     "X" means nullptr // n is pointer
p                   "1"->⬆️           // p is not a ptr, it is a Node; copy ctor
q ----------------> "1"->⬆️           //is a pointer on the stack that points to the
heap; is dereference of n
// they are sharing info (pointer)

This is an example of compiler provided copy ctor being wrong: Performs a shallow copy,
data is shared amomg lists. We want a deep copy: 3 independent lists with the same info.

Here how we do:
struct Node {
  int data;
  Node* next;
  Node(const Node& other):data{other.data},
    next{other.next? new Node {*other.next}:nullptr} {} // Recursively calls copy ctor.
                                                        // other.next is a pointer
                                                        // if other.next is a nullptr:
return false
};

// Stack:              Heap:
n ----------------> "1"->"2"->"3""X"     "X" means nullptr // n is pointer
p "1"->"2"->"3""X"
q ----------------> "1"->"2"->"3""X"
```

When is copy ctor called:

- Initialize an object from another of same type
- Pass by value
- Return by value

```
struct Node {
  Node(int data, Node* next=nullptr): data{data}, next{next} {}
};
Node n {5, nullptr};
Node m{6};
Node p = 100; // implicit conversion; invodes the single arg ctor for Node

string s = "Hello"; // single arg ctor for string that takes in a const char*
//stad::String  const char*

void f(Node n);
f(Node {5});
f(5) // arg 5 is converted into a Node
```

Why might you want to disable this?

- Compi;er does conversion implicitly
- No warning or indication the conversion will occur
- Potential to miss errors

Makde the ctor explicit: (to make is disable single arg ctor)

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

Node p = 4; //DOES NOT COMPILE
f(4); // DOES NOT COMPILE
//only
Node p{4};
f(Node {4});
```

Destructors (dtors)

- Runs whenever an object is destroyed
- when an object goes out of scope (stack allocated) or when delete is called on a ptr to that object (heap allocated)
- Compiler provided dtor: calls dtor on all object fields
- STeps

- Dtor body runs
- object fields have their dtors called in *reverse* declaration order
- *Later*
- Space is deallocated

```
Nodes: should write our own dtor
Node *p = new Node{1, new Node{2, new Node{3, nullptr}}}; // p is in the stack pointing to
"1" Heap 1->2->3X
delete p; // still memory leak
          // only delete "1", not NOdes 2 and 3
// we need to write dtor - handle cleaning up rest of linked list focus

struct Node {
  ...
  ~Node() {
    delete next; // delete nullptr does nothing
                 // recursively deletes rest of the list, base case is delete nullptr
  }
};

Node p {4, new Node {5, nullptr}};
// Entire linked list is freed when p goes out of scope.
```

Copy Assignment Operator

```
Student s {60, 70, 80};
Student r = s; // copy ctor
Student t{0, 0, 100};
r = t; // r exists alrdy, and needs a
       // copy assignment operator
```

- **Assign** one object to another of the same type
- Compiler provided copy assignment operator; Does **field-by-field assignemnt**

```
Node n{1, new Node{3, new Node{3, nullptr}}};
Node p{4, new Node {5, nullptr}};
p = n; //leads to copy assignemnt operator will run

   Stack           Heap
n   "1"   ------>  "2" -> "3X"
p   "1"   --------^ "5X" // "5X not freed - memory leak"
```

```
struct Node {
  Node& operator = (const Node& other) {
    delete next; // purpose: delete "5X" above. delete an object will recursively called until delete nullptr
    data = other.data;
    next=other.next?new Node{*other.next}:nullptr;
    return *this;
  }
};
// support chained assignment: return type of operator "=" is Node&
int a, b, c, d;
a = b = c = d = 100; // d=100 returns d     d would be this, 100 would be other
                     // c=d returns c       c would be this, d would be other
                     // a=b returns b

n = n; // this will lead an issue cuz we delete all our own nodes by (delete next;) and try to copy them
       // self-assignment

// so, we need self-assignment check:
struct Node {
  Node& operator = (const Node& other) {
    if (this==&other) return *this;
    delete next; // purpose: delete "5X" above. delete an object will recursively called until delete nullptr
    data = other.data;
    next=other.next?new Node{*other.next}:nullptr;
    return *this;
  }
};

?? if this still work?  MIDTERM QUESTION NEEDED TO THINK ABOUT: WORKS
if (*this==&other) return *this;

!!!
// one more improvement: if we request memory with new, there is a chance it may fail. Better strategy, request memory before deleting Nodes.
struct Node {
  Node& operator = (const Node& other) {
    if (this==&other) return *this;
    Node* temp = other.next?new Node{*other.next}:nullptr; // better cuz we dont change the original code and will fail if temp not work
```

```
    delete next; // purpose: delete "5X" above. delete an object will recursively called
until delete nullptr
    data = other.data;
    next = temp;
    return *this;
  }
};
```

```
// Copy and Swap Idiom: Another way of writing copy assignment operator
std::swap(a.b) <- <utility> // a gets b's value; b gets a's value
struct Node {
  void swap(Node& other) {
    std::swap(data, other.data);
    std::swap(next, other.next);
  }
};

// deep copy
Node& operator = (const Node& other) {
  Node* temp{other};
  swap{temp};
  return *this;
}
```

Lec 9 - Move ctor/assignment, elision, operator overloading (revisited)

Move ctor - Rvalues

Recall: lvalues - long lasting, memory address that we can get with & (address-of)

```
Node oddsOrEvens() {
  Node o {1, new Node{3, new Node{5,..., new Node{99, nullptr}}}};
  Node e {2, new Node{4, new Node{6,..., new Node{100, nullptr}}}};
  char c;
  cin>>c;
  return (c=='o')?o:e;
}
Node l = oddsOrEvens(); //returns by value (COPY CTOR)

  stack       Heap
o   1-------->3->5->7->...->99X
l   1-------->3->5->7->...->99X

// Here, o will gone soon when it is out of scope, and nobody changes value in o, copy
ctor takes time
```

```
//    we copy 50 Nodes to create l. Then, o goes out of scope, and we delete those 50
NOdes we just copied from.
```

Why copy Nodes just to delete roginals immediately after?

- We can steal data from another object. So long as we know that object will no longer be used.
- We need a way to determine if a paramter is an
    - lvalue (long-lasting, must copy), OR
    - an rvalue (temporary), then we can steal data.
- MOVE CTOR !

```
Node&& - this is an rvalue reference - binds to temporaries.
We can overload ctor
Node(const Node&other) - copy ctor
Node(Node&&other) - move ctor

struct Node {
  Node(Node&& other):data{other.data}, next{other.next} {
    other.next = nullptr;
  }
}

memory diagram for the above code:
o: 1X        3->5->7->...->99X  (1X is because we let other.next = nullptr;)
l: 1--------⬆️
```

Move Assignment Operator:

```
Node n{1, new Node{2, new Node{3, nullptr}}};
n = oddsOrEvens(); // Rvalue move assignment operator

struct Node {
  Node& operator=(Node&& other) {
    swap(other);
    return *this;
  }
};
memory diagram for the above code:
o: 1->2->3X
e: 2->4->6->8->...->100X

After swap
o: 2--⬇️ 2->3X
e: 1-----⬆️
```

```
4->6->8->...->100X
```

If you write move ctor/move assignment operator - these are used instead of **copies** where possible. If you write copy ctor/assignment op, but no move version - copying version always used.

Rule of Big 5

- If you write one of: dtor, copy ctor, copy assignment op, move ctor, move assignment op, you should probably write all 5
- Only write them if compiler provided versions are incorrect. Often write the big 5 for classes with ownership - classes that manage a resource (like memory)

Elision:

```
vec getvec() {return {0,0}}
vec v=getvec(); what happens? Only Basic ctor, no move/copy
```

- In certain cases, compiler can perform something called more/copy elision rather than constructing vec in getVec and moving it, simply write values into main stack frame. Can happen even if it changes output of program!

```
other Example:
void useVec(vec v) {...}
use vec({1,2}); // elision happens, one ctor is run, basic ctor
```

Not expected to know exactly all the places where e;ision may occur, must that it is possible

Member Operators

Note: Previously we wrote operator overloads outside structs. Operator=, we wrote as a method

If we write operator overloads as methods, this becomes left-hand side parameter

```
struct vec{
  int x,y;
  vec operator+(const vec& other) {
    return {x+other.x, y+other.y};
  }
  vec operator*(int k) {
    return {x*k, y*k}; // this only supports v*k
  } // to support k*v - must write a standalone function as before
};
```

Advice: For arithmetic assignment, implement one using the other

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

vec& operator+(const vec& v1, const vec& v2) {
  vec tmp{v1};
  return tmp+=v2;
}

or

vec& operator+(vec& v1, const vec& v2) {
  return v1+=v2;
}
```

I/O Operators:

```
struct vec {
  ostream& operator<<(ostream&out) {
    return cout<<x<<" ";
  }
}; //wrong
// this means vec<<ostream
```

I/O operators must be defined as standlone functions. Operator overloads that must be methods

- Operator overloads that must be methods:
  - operator=
  - Operator[]

- operator->

CS 246 - Lec 10

Last time: Elision, member operators

This time: Arrays of objects, const objects, static, 3-way comparison

Arrays of Objects

```
struct vec {
  int x, y;
  vec(int x, int y):x{x}, y{y} {}
};
vec* vp = new vec[15]; //wont compile
vec array[10]; //wont compile
```

When we create an array of objects, default ctor will run for each object in the array. No default ctor -> no compilation

Solutions:

1. Add a **default constructor** back, so it can be called for each element in the array. If no default ctor makes sense, then what?
2.

```
// stack allocated arrays; give initial vectors
vec array[3] = {{1,2}, {3,4}, {5,6}};
```

3. Use an array's of pointers

```
vec** vp = new vec*[15]; //Heap
for (int i = 0; i < 15; ++i) {
    vp[i] = new vec(0, 0);  // Now each pointer in the array points to a vec object on the
heap
}
// vp[1] = *vp[i+1]

vec * array[10]; // stack
// compiler is allowed
array[0] = new vec(1, 2);  // Points to a heap-allocated vec
vec v(3, 4);
array[1] = &v;             // Points to a stack-allocated vec
```

Const Objects

- Objects can be const. Used most commonly in the form of const lvalue references

```
const Student S{60, 70, 80};
S.assns = 80; // will not compile
cout<<S.grade()<<endl; // Does not compile
// Because the compiler does not know whether or not S.grade() will modify the fields.
Calling S.grade() with a const S is not allowed.
Struct Student {
  int assns, mt, final;
  float grade() const; // This indicates that grade() will not change any fields.
}

float Student::grade() const {
  return assns*0.4 + mt*0.2 + final*0.4;
}
constness is a part of the methods' signature - must appear in both interface and
implementation


const Student S{60, 70, 80}; // Now will Compile
cout<<S.grade()<<endl;
// Now compiles, only must methods can be called on const objects.
```

- Imagine collecting stats on our program

```
Struct Student {
  int assns, mt, final, calls;
  float grade() const {
```

```
    ++calls; // wont compile since it is const function
    return assns*0.4+mt*0.2+final*0.4;
  }
};
// Issue: Physical vs. logical constness
Physical constness: Making sure that the bits that make up the object do not change.
Logical constness: Making sure the "essence" or the fields and memory we care about do not
change.

// Solution: Decalre calls as Mutable
Struct Student {
  int assns, mt, final;
  mutable int calls; // This field may be changed, even for a const object or const
method.
};
Advice: Use mutable sparingly, decreases usefulness of const
```

Static fields/Static Methods

Issue: calls is tracked separately for each student

What if I want it shared among all students?

Declare a **static field** - Shared among all instances of a class

```
Struct Student {
  inline static int numInstances = 0; // at the beginning of the program, keeps the
variable to zero
  Student(int assns, int mt, int final): assns{assns}, mt{mt}, final{final} {
    ++numInstances;
  }
};
// The inline keyword, in this context, tells the compiler that the variable can be
defined in the header file and the definition can be shared across multiple translation
units without violating the One Definition Rule (ODR).
Before g++20:
// In the header file (.h or .hpp)
struct Student {
    static int numInstances;
};

// In the source file (.cpp)
int Student::numInstances = 0;
```

```
Student s{1, 2, 3};
Student t{4,5,6};
cout << Student::numInstances << endl; // prints out 2
```

- Static methods: called on the class itself, rather than any particular object - can only access static fields

```
Struct Student {
  static void howMany() {
    cout << numInstances << endl;
  }
};
Student s{60,70,80};
Student::howMany();

This is a static member function of the Student struct called howMany(). Being static
means this function can be called on the struct itself (i.e., Student::howMany()) without
needing to instantiate an object of Student. This function, when called, prints the value
of numInstances to the console.
```

3-Way COmparison

- Recall: Comparing string in C;
    - Strcmp(s4, s2);
        - < 0 if s1<s2
        - = 0 if s1 = s2
        - greater than 0 if s1>s2

In C++:

```
// 2 comparions instead of 1
// Slower than the C version
if (s1<s2) {

} else if (s1 == s2) {

} else {

}
```

Can we achieve this in C++? YES

3-Way comparison operator: <=> // Spaceship operator

```
import <compare>
String s1 = " ", s2 = " ";
std::strong_ordering n = (s1<=>s2);
if (n < 0) {} // s1 < s2
else if (n == 0){} // s1 == s2
else if (n > 0){} // s1 > s2

The class type std::strong_ordering is the result type of a three-way comparison that:
Admits all six relational operators (==, !=, <, <=, >, >=).
Implies substitutability: if a is equivalent to b, f(a) is also equivalent to f(b), where
f denotes a function that reads only comparison-salient state that is accessible via the
argument's public const members. In other words, equivalent values are indistinguishable.

std::strong_ordering - this is a long type name.
To avoid writing it out, use automatic type deduction,
auto n = (s1<=>s2); //n gets the type of the expression on the RHS.
```

Ex:

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

- Having written <=> for Vecs, I can new do:
  - v1<=>v2, v1!=v2, v1==v2, v1<v2. v1>v2, v1<=v2, v1>=v2;
  - All of these are given for free!

CS 246 - Lec 11

Last time: const objects, static, <=>

This Time: <=>, encapsulation, Classes, Iterator

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

If all your operator <=> does is compare fields in declaration order - we can "default" this:

```
struct Vec {
  int x, y;
  auto operator<=>(const vec&other) const = default;
};
// This line declares the spaceship operator for the Vec struct and defaults its
implementation. There's a typo: vec should be Vec.
By using = default, you're telling the compiler to generate a default implementation of
the spaceship operator. The default implementation will perform a member-wise comparison
in the order that the members are declared in the struct.

For the Vec struct, this means:

It will first compare the x values of the two Vec objects.
If the x values are the same, it will then compare the y values.
The default implementation is a concise and expressive way to define comparison operations
for simple structs/classes, especially when the desired behavior is just to compare the
members in the order they're declared.

Using = default for the spaceship operator is particularly useful for structs like Vec
where the desired behavior is straightforward member-wise comparison. It eliminates
boilerplate and reduces the potential for errors in custom implementations.
```

Consider a case there = default is not the correct approach: comparing Nodes (Linked Lists) Default would **compare memory addresses** - we want to compare list contents.

```
struct Node {
  auto operator<=>(const Node& other) const {
    auto n = data<=>other.data;  // Compare the data members of the nodes
    if (n!=0 || (!next&&!other.next)) return n; // If data is different or both nodes
don't have a next node, return the comparison result
    if (!next && other.next) return std::strong_ordering::less; // If current node doesn't
have next but other does, current is less
    if (next && !other.next) return std::strong_ordering::greater; // If current node has
next but other doesn't, current is greater
    return *next<=>*(other.next);  // If both nodes have a next node, compare the next
nodes
  } // operator <=>
}; //Node
// <compare>
std::strong_ordering::less: Represents that the left-hand operand is less than the right-
hand operand.
std::strong_ordering::equal: Represents that the left-hand operand is equal to the right-
hand operand.
std::strong_ordering::greater: Represents that the left-hand operand is greater than the
right-hand operand.
```

Encapsulation:

```
struct Node {
  int data;
  Node* next;
  Node(int data, Node* next):data{data}, next{next} {}
  // big 5;
};
Node n {1,nullptr};
Node m {2, &n};
Node o {3, &n};

// Destructor Runs:
When n is freed, no issue;
// m's dtor runs, delete next = &n;
// Deleting stack allocated memory, crash.

// o's dtor runs -> delete next. Double delete; and deleting stack allocated memory.
```

- Assumptions we made when writing BIG 5:
  - next is always heap allocated, or nullptr.

    - No nodes share between linked lists.

  These are **invariants** - this property you expect to hold true, whenever using an object of a particular type

- Issue: Invariants can be easily broken by the user.
- Introduce: **Encapsulation** - Classes should be treated as a capsule, or a "black box". Clients should not have access to underlying data, should interact solely via your methods.
- Access specifiers allow us to control how a user accesses data from our classes.

```
struct Vec {
  private: // private fields and methods only be accessed / called from within Vec methods
    int x, y;
  public: // can be accessed / called from anywhere.
    Vec(int z, int y);
    int getX() const {return x;}
};
```

- Default specifier for structs is public
    - Preferred: keep data private by default - **use class keyword**

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

- Only difference b/w classes and structs:
    - Structs are public by default, classes are private by default.

Revisit Linked List, ensure invariants are always enforced.

```
// List.cc
export module List;
export class List {
  Struct Node; // private nested class, to be defined in impl file. declaration
  Node* head = nullptr;
  public:
    ~List();
    int ith(int i) const;
    void addToFront(int n);
};

// List-impl.cc
```

```
Struct List::Node {
  int data;
  Node* next;
  ..
  ~Node() {delete next;}
};

void List::addToFront(int n) {
  this->head = new Node{n, head};
}
List::~List() {delete head;}

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

Problem: Now takes *O(n^2)* time to loop through our List - not good!!!

- How do we maintain encapsulation while also getting fast iteration?
  - **Iterator pattern**
  - Iterator is an example of a design pattern -
    - effective solution to a common problem
  - create a class that acts like an abstraction of a ptr:
    - allows us to walk the linked list, while maintaining encapsulation.

```
class List {
  struct Node;
  Node* head = nullptr;
  public:
    class Iterator {
      Node* cur;
      public:
        Iterator(Node* cur): cur{cur} {}
        Iterator& operator++() {
          cur = cur->next;
          return *this;
        }

        bool operator!=(const Iterator&other) const {
          return cur != other.cur;
        }
        int operator*() const {
```

```
          return cur->data;
        }
    }//ends Iterator
  Iterator begin() const {
    return Iterator {head};
  }
  Iterator end() const {
    return Iterator{nullptr};
  }
};
```

```
int main() {
  List l;
  l.addToFront(1);
  l.addToFront(2);
  l.addToFront(3);
  for (List::Iterator it=l.begin(); it!=l.end(); ++it) {
    cout << *it << endl;
  }
}
```

If we have the following:

- A class with methods begin and end which returns some "Iterator type"
- This type supports ++, != and *
- Then we get range-based for loop syntax

  ```
  for (int n:l) { // This copies each int in the list
    cout << n << endl;
  }
  ```

- If we want modification:

  ```
  for (int& n:l) {
    n*= 2;
    cout << n << endl;
  }
  ```

Slight encapsulation problem:

```
List::Iterator it {nullptr};
// User should only create Iterators via begin and end
```

```
// Solution: make Iterators ctor private. Then, user cannot create Iterators, but also
List cannot create iterators. Give List privileged access to call the provate constructor.
Class List {
  Struct Node;
  Node* head;
  Public:
    Class Iterator {
      Node *cur;
      Iterator(Node* cur): cur{cur} {}
    Public:
      ..
      friend class List; // can be placed anywhere in Iterator
// friend class List;: This line is critical. It declares the outer List class as a friend
of the Iterator class.
    };
    Iterator begin() {
      return Iterator {head};
    }
    Iterator end() {
      return Iterator {nullptr};
    }
};
// Because Iterator declared List as a friend, List may access Iterator's private fields
and methods
```

- A `friend` class has access to the private (and protected) members of the class in which it's declared as a friend. This essentially breaks the usual encapsulation rules of C++.
- The line `friend class List;` inside the `Iterator` class declares the outer `List` class as a `friend` of the `Iterator`. This means that the `List` class can access the private (and protected) members of the `Iterator` class.
- The comment provided in the code indicates the design intention. The designer of these classes wants the user to create instances of `Iterator` only via the `begin` and `end` methods of the `List` class and not directly (to perhaps prevent misuse or ensure the iterator is always in a valid state). To achieve this, the constructor of `Iterator` is made private. Normally, this would mean that no one, not even the `List` class, could create instances of `Iterator`. But with the `friend` declaration, the `List` class can.
- Rather than making friends - use **access/mutator** methods; (getters/setters)

```
Class Vec {
  int x, y;
public:
  Vec(int x, int y):x{x}, y{y} {}
  int getX() const {return x;}
  void setX(int a) {x=a;}
};
```

What about operator<<? Standalone function, but still want access to private fields/

- Use getX, getY if defined

```
std::ostream& operator<<(std::ostream& os, const Vec& v) {
    os << "(" << v.getX() << ", " << v.getY() << ")";
    return os;
}
```

- Declare a friend function

```
Class Vec {
  int x, y;
Public:
  friend ostream& operator<<(ostream&, const Vec&); // declare a friend function.
This fn can access private dield methods
};
```

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

Equality Revisited:

List class encapsulation Node

We can now keep track of extra info about our List, e.g. length.

Strategies to implment a length method:

1. Iterate through Nodes, count number - return O(n)
2. O(1) time: keep a length field, incrementation addToFront is called, return on length() called.

Now: can optimize spaceship comparison:

- If we write (l1 <=> l2) == 0, this checks if two lists are equal to each other.
    - Takes O(n) time
    - Shortcut for == comparison, two Lists of different length cannot be equal

```
Class List {
  Node* head;
  int length;
Public:
  auto operator<=>(const List& other) {
    if (!head && !other.head) return std::strong_ordering::equal;
    if (!head && other.head) return std::strong_ordering::less;
    if (head && !other.head) return std::strong_ordering::greater;
    return *head <=> *other.head;
  }
  bool operator==(const List&)const {
    if (length!=other.length) return false;
    return (*this<=>other) == 0;
  }
};
```

Now:

- If we write l1 == l2: we use == operator, more efficient - checks to see if lengths are equal.
- Otherwise, !=, <=, >=, >, < still use spaceship operator. Allows us to optimize equality checks.

System Modelling:

Defn: View classes relationships between them in a graphical way so as to communicate about large programs

```
UML diagrams, (Unified Modelling Language)

       Vec                    <- Name
- x: Integer            <- Fields(optional): - means private; + means public
- y: Integer
_____________________________________________________

+getX: Integer     <- Methods (optional)
+getY: Integer
```

Relationships between Classes

- Composition: Nesting one object with another.

```
Class Basis {
  Vec v1, v2; // Vec is composed with Basis
};
```

- Generally if class B is composed with class A:
  - If A dies, then B dies // B is Vec, A is Basis
  - If A is copied so is B (deep copy)
  - B has no independent existence in the program
  - Also called an "owns-a" relationship
    - Basis "owns" 2 vectors
  - Implementation: Typically done via object fields, although not necessarily

Let 13

Composition: If A "owns-a" B, then

- A dies -> B dies
- A copied -> B copied (deep copy)
- B has no independent existence apart from A

Aggregation: If class A "has-a" B, then

- A dies -> B keeps living
- A is copied -> B is not copied (shallow copy)
- B may have independent existence outside A

```
class Student {
  int id;
  University* myUni;
public:
  Student() {...}
}
```

```
int main() {
  Unievrsity uw{..};
  Student s{1, &uw};
  Student t{2, &uw};
}
```

Consider Linked List, implemented via ptrs, but is a composition relationship. Implementing big usually means

owernship and composition

Specialization:

Imagine a program to manage and catalogue library books. We will manage Books, Texts, and Comics

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

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

Comic (UML form)
-----------------
- title: string
- author: string
- length: Integer
- hero: string

Question: What is I want an array of all types of Books
  - Should store Texts, Books, and Comics
```

1. Use void*s, which can point to anything
    - Not type safe
2. Use a union type

```
union BookType {
  Book* b; // stores one of these types
  Text* t; // stores one of these types
  Comic* c; // stores one of these types
};
BookType u;
u.b = new Book{...};
cout << u.t->type << endl; // u.t is not safe; undefined
```

Issue is that the compiler is not aware of the relationship; A comic is a type of Book.

A text "is-a" Book; Specialization relationship or "is-a". Implemented in C++ via public inheritance.

```
class Book { // Basic or Superclass
  string titile, author;
  int length;
public:
  Book(string title, string author, int length): title{title}, author{author},
length{length} {}
}

class Text:public Book { // means Text is a type of book
                         // derived subclasses
  string topic;
public:
  Text(...) {..}

};

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

- Text and Comic inherit from Book, this means that they have titile, author, and length fields
- A subclass inherits **all fields+methods** from its superclass. title, author, length are private in Book
- title, author, length are only accessible in Book, not subclasses of Book

```
class Text:public Book {
  string topic;
public:
  Text(string title, string author, int length, string topic): // DOES NOT WORK
  title{title}, author{author}, length{length}, topic{topic} {}
};
```

Why not work?

- Titile, author, length are private in Book, cannot be changed in Text. AND MIL can only be used for your class's fields
- Object creation Sequence
    - Space is allocated

- Superclass ctor runs
- Fields are initialized via MIL
- ctor body runs

In step 2, superclass ctor is called. If you do not specify the args for superclass ctor, default ctor will be used. If that does not exist -> wont compile

```
class Text:public Book {
  string topic;
public:
  Text(string title, string author, int length, string topic):
  Book{title, author, length}, // step 2 - superclass
  topic{topic} {} // step3
};
```

Generally, it is a good idea to keep superclass fields, private to subclasses.

If we do want to give just subclasses access; use protected=>accessible in class and subclasses, nowhere else

Consider adding multple authors to Texts:

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

class Text:public Book {
  string topic;
public:
  Text() {}
  void addAuthor(string a) {author+=a;} // protected can be accessed
};
```

Preferred: use protected mutators

```
class Book {
  string title, author;
  int length;
protected:
```

```
  void setAuthor(string a) {author=a;} // may only be called from subclasses
};

Specialization "is a" in UML
  Book
   ^
   |     // Speicialization: public inheritance in C++
 -- --
 |    |
Text  Comic
```

CS246 - Lec 14

Recall: we want a bookcase, an array of Books that may behave different depending on if it is a Book/ Text/ Comic

```
Book: heavy if > 200 pages

Text: heavy > 500 pages

Comic: heavy > 30 pages

class Book {
protected:
  string title, author;
  int length;
public:
  Book(..) {..}
  bool isHeavy() const {
    return length > 200;
  }
};

class Text: public Book {
  string topic;
public:
  Text(...) {..}
  bool isHeavy() const {
    return length > 500;
  }
};

Book b{"..", "..", 300};
Text t{"..", "..", 300, "topic"};
b.isHeavy(); //return true
t.isHeavy(); //return False
```

```
Book b = Text{"..", "..", 300, "topic"};
// This is allowed due to public inheritance, a Text is a type of Book
****b.isHeavy??? return True
- After assigning b to the Text, we lost information that b was a Text, b is from that point treated as a Book.
- This is running compiler provided book move ctor, which only moves title, author, length. (Same thing happens w/copies)


Book b        Text
title         title
-----         -----
author        author
-----         -----
length        length
              -----
              topic

object slicing - ctor runs for a superclass and chops off subclass fields
```

- What if we use ptrs instead?

```
Text t{"..", "..", 300, "topic"};
Book* bp = &t;
bp->isHeavy();
return True;
```

- To get isHeavy to use the Text definition even it pointed to by a Book - must use a virtual method.

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

class Text:public Book {
  ...
public:
  ...
  bool isHeavy() const override {return length > 500;}
};
```

- Definition:

    - static type of a ptr - given an LHS in the type declaration: Book* pb = &t; (underline is static type)
    - dynamic type: what is the type of the "underlying" object.

1. If not using via ptr or reference: we always call the **method definition** based on the **static type**
2. Using via ptr or ref?
    1. Non-virtual method -> use the static type
    2. virtual method -> use dynamic type

```
Text t{"..", "..", 300, ".."};
Book b{"..", "..", 350};
Text* pt = &t;
Book* pb = &t;

t.isHeavy(); // False - static type
b.isHeavy(); // True - static type
pt->isHeavy(); // False
pb->isHeavy(); // False - using via ptr -> using virtual method -> using dynamic type
```

1. `pt->isHeavy();` - This uses the pointer `pt` which is of type `Text*`. When we use a pointer to call a virtual function, C++ uses dynamic dispatch to determine the actual type of the object. In this case, `pt` points to a `Text` object, so the overridden `isHeavy` in `Text` is called. This will again check if 300 is greater than 500. It's not, so it returns `False`.
2. `pb->isHeavy();` - This is the most interesting case. The pointer `pb` is of type `Book*`, but it's actually pointing to a `Text` object (`t`). When calling a virtual function using a pointer, C++ determines the method to call based on the actual object the pointer points to, not the type of the pointer. Here, since `pb` is pointing to a `Text` object, the overridden `isHeavy` in `Text` is called. This again checks if 300 is greater than 500. It's not, so it returns `False`.

- Bookcase example:

```
Book* myBooks[20];
myBooks[0] = new Text{..};
myBooks[1] = new Comic{..};
...
for (int i = 0; i < 20; ++i) {
  cout << myBooks[i]->isHeavy() << endl; // use dynamic type
}
```

- What does override do in Text::isHeavy?

Nothing in terms of program execution; Checks at compile time that superclass method is actually virtual

This is an example of polymorphism - "many forms" - superclass being able to represent multiple subclass types. We have seen this; ifstream is-a istream;

Destructions revisited

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

class Y:public X{
  int* m;
public:
  Y(int size1, int size2):X{size1}, m{new int[size2]} {}
  ~Y() {delete[] m;}
};
X xobj {5};
Y yobj {5,10};
X* xp = new X{5};
Y* yp = new Y{5,10};

X* xptoy = new Y{5,10};
delete xp;
delete yp; // non-virtual; static type
delete xptoy; Leaks!! we want to run all dtors, not only superclass' dtor
```

Object destruction Sequence

1. Dtor body runs
2. Object fields have dtors called in reverse declaration order
3. superclass dtor runs (*new*)
4. space reclaimed

- Why does delete xptoy leak memory?
  - calls dtor on underlying object --- that is a method!
  - dtor is non-virtual -> use static type
  - we call x's dtor instead of Y's for xptoy, leaking memory.
- Solution: Make x's dtor virtual

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

- If you knew a "Class may be subclassed", **ALWAYS** make its dtor virtual
- If you know a class will **NEVER** be subclassed, make compiler enforce it via final

```
class C final {} // wont compile if C is subclassed
```

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

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

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

In my program, I will create regular and coop student objects. Student feees is declared, how should we implment it. Ideally, no implementation, best to have the compiler enforce that only regular and coop students are created.

Solution:
Define Student::fees as pure virtual:

```
class Student {
  virtual int fees() const = 0; // pure virtual
}

// pure virtual is organized subclass and dont need to implement this function
```

- Pure virtual methods are allowed to not have an implementation. Any class that defines a pure virtual method cannot be instantiated.

Student S; // wont be compile

- Student is now an "abstract class",
- Subclasses of an abtract class are also abstracted unless they provide an implementation for all inherited pure virtual methods. (Regular and Coop subclasses need to provide implementation for fees() function), then they are concrete.
- The point of abstract classes is to **organize** subclasses.

```
class Student {// abstract class
protected:
  int numCourses;
public:
  virtual int fees() const = 0;
};

class Regular:public Student { // concrete now -> it has implementation of fees() function
public:
  int fees() const override {return 600* numcourses};
};
```

- In UML, represent abstract classes and pure virtual methods via italics.(or use ** instead)

Ex:

```
*Student*
----------
+*fees(); Integer*
```

- Revisiting Big 5 with inheritance

```
class Book {
  string title, author;
  int length;
public:
  big 5
};

class Text{
  string topic;
public:
  // use compiler provided big 5;
};
Text t1{...}, t2{...};
```

```
t1 = t2; //compiler provided works here, t1's fields get the values of t2's fields;


// Lets see compiler provided big 5
// Copy ctor
Text::Text(const Text& other):Book{other}, topic{other.topic} {} // take other as a Text
reference "other" seen as a Book reference and do the copy ctor

// move ctor
Text::Text(Text&& other): Book{std::move(other)}, topic{std::move(other.topic)} {}

Above: If we were to call Book{other}, this would do copy title, author, length, rather
than move then, why??
- Other although an rvalue reference, but in the function, it is an lvalue because wont
die within the line, only disappear when curly brace seen;
- lvalues invoke copy sementics, we use std::move to force something to be treated as an
rvalue


// copy assignment operator
Text& Text::operator=(const Text& other) {
  Book::operator=(other); // this will take text reference treated as a book reference and
call Book's operator=
                          // if not write Book::, it will get recurrsion to call the
function
  topic = other.topic;
  return *this;
}

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

Now consider:

```
Text t1{"Alias", "CLRs", "1000", "CS"};
Text t2{"Shakespeare", "Mr. English", 2000, "English"};

Book& r1 = t1;
Book& r2 = t2;
r1 = r2; // what happens?
         // reference cannot be assigned necessary
// Operator equal is non-virtual, we use static type, will call book assignment operator

t1 now contains:
"Shakespeare", "Mr. English", 2000, "CS" // because in Book has no "topic" field.
"English" not changing to "CS"
```

- The above is **partial assignment**, cause not all the fields in Book superclass
  - this is because we use static type. we call Book::operator=
  - we need to use dynamic type, using Text::operator=
  - So, we need to make Book::operator= virtual

```
class Book {
  string titile, author;
  int length;
public:
  virtual Book& operator=(const Book& other);
};

class Text:public Book {
  string topic;
public:
  Text& operator=(const Text& other) override; // make sure the method overrides that in
the superclass
  // However, this is not a valid override. argument is (const Text& other), in
superclass, it is not
  // we are allowed to override the subclass pointing to a reference/ptr
  // issue is argument type!!!!!!!!!!
  // args must match eaxtcly superclass.
  need to change to:
  Text& operator=(const Book& other) override; // OK!
};

// This allows:
Text t1{...};
t1 = Book{...};
t1 = Comic{...};
// this compiles, bad. we should  not assign a comic to a book.
```

```
// This is mixed assignemnt
```

- Conclusion:
  - If operator= is non-virtual: **partial assignment**
  - If its virtual, we get **mixed assignment**
  - Solution: restructure class hierarchy
  - Advice: **Make superclasses abstract**

```
          *AbstractBook*
               ^
               |
         ----------------
         |     |        |
     regular  Text    comic
```

```
class AbstractBook {
  string title, author;
  int length;
protected:
  AbstractBook& operator=(const AbstractBook& other) = default; // will set title{title}, author{author}, length{length}
public:
  AbstractBook() {...}
  virtual ~AbstractBook()=0; // use dtor as pure virtual to make a class abstract if no other method makes sense
};

class Regular:public AbstractBook {
public:
  Regular& operator=(const Regular&other) {
    AbstractBook::operator=(other);
    return *this;
  }
};
... same for text and comic
```

Methods assignment - makes sense. Text can only

Partial assignment:

```
Text t1{}, t2{}
AbstractBook& r1 = t1;
AbstractBook& r2 = t2;
//what if
r1 = r2; // no longer compile because we try to call AbstractBook::operator= outside the class and it is protected
```

CS246 - Lec16

```
class  AbstractBook {
  ...
public:
  virtual ~AbstructBook() = 0;
}
```

- In some cases, we must provide an implementation for pure virtual methods. Namely, if we call pure virtual method, and, Text, Comic, Real book all call superclass dtor.
- Implement AbstractBook;s dtor.

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

- **Templates**: Consider List class again:

```
class List {
  struct Node {
    int data;
    Node* next;
  }
  Node* head;
public:
  class Iterator {
    ...
    int& operator*() const;
  }
  int ith(int i) const;
  void addToFront(int n);
};
```

- What if we want a List of strings, or Students. We could copy/paste contents - this is not an effective solution. Instead - use a **template**

```
template<typename T> class List {
  struct Node {
```

```
    T data;
    Node* next;
  };
  Node* head;
public:
  class Iterator {
    ...
    T& operator*() const;
  };
  T ith(int i) const;
  void addToFront(const T& n);
};

List<int>l;
l.addToFront(2);
l.addToFront(3);

List<string>ls;
ls.addToFront("hello");

for (List<int>::Iterator it = l.begin(); it != l.end(); ++it) {
  cout << *it << endl;
}
List<List<int>>l3;
l3.addToFront(l)
```

- **Templates** are just as fast at runtime as writing custom versions of stringList, intList, StudentList, etc.
- At compile-time, copies of the class are created for each type T it is used with, then compile as normal.

Standard Template Library

- Collection of useful templated classes.

```
std::vector, found in <vector>, resizable array
Vector<int> v{4,5}; //array containing 4,5
v.emplace_back(6); // contains 4,5,6
v.emplace_back(7) // contains 4,5,6,7; auto allocate memory
```

  - **Note**:

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

// type deduction exists
vector w {1,2,3}; // int is inferred from list
```

```
// Loop
for (int i = 0; i < v.size();++i) { //.size() returns sized vector
  cour << v[i] << endl; //gets ith element
}

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

for (int n:v) cout<<n<<endl;

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

v.pop-back() // removes final element
v.erase(it) // erases element pointed to by iterator it;

!!!careful using v.erase() in a loop
  for (auto it = v.begin(); it != v.end(); ++it) {
    if (*it == 5) v.erase(it); // DOESNOT WORK!!
  }
// ex: 4 5 5 6 // the second 5 is not erase

!!!Correct way!!!
  for (auto it = v.begin(); it!=v.end();) {
    if (*it == 5) v.erase(it);
    else ++it;
  }

!!! Recommended to use vectors instead of new[] and delete[] - safer!!!
!!! Vectors are guaranteed to be implemented using arrays!!!
```

Design Patterns

- In general, we would like to program to interfaces instead of implementations.
- We use abstract classes to provide methods which subclasses may customuze behavior of

```
class AbstractIterator {
public:
  virtual int& operator*() const = 0; // sets to pure virtual
  virtual AbstractIterator& operator++() = 0;
```

```
  virtual bool operator!=(const AbstractIterator& other) const = 0;
  virtual ~AbstractIterator() {}
}

class List {
  ...
public:
  class Iterator:public AbstractIterator { // inherit from AbstractIterator
    ...
  }
};

class Tree {
  ...
public:
  class Iterator:public AbstractIterator {
    ...
  }
};

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

- Foreach is programmed using AbstractIterator's interface. The behavior is customized by subclasses. Writing new subclasses does not require any changes to foreach

Decorator Pattern

Problem: Add/remove functionality at runtime;

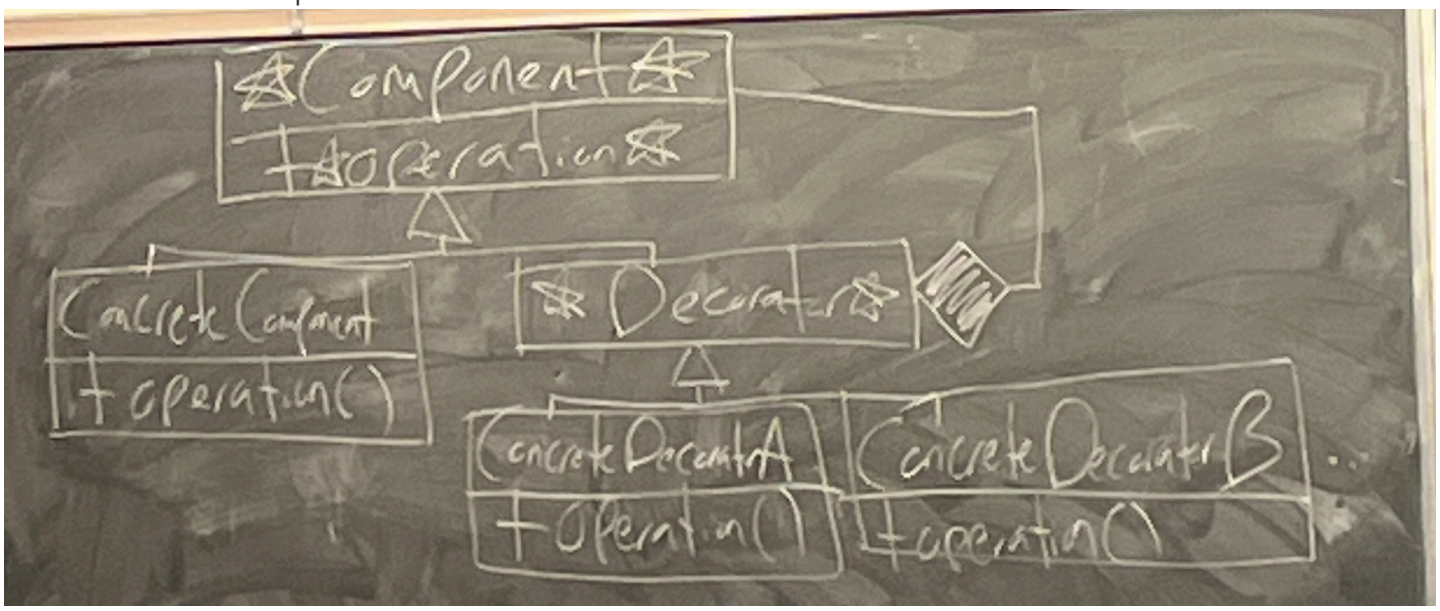
Example: Consider GUI - windowing system

WE want

- Basic window
- Tabs (on/off)
- Scrollbar (on/off)
- Bookmarks (on/off)
- If we create a window superclass and subclass for different window types.

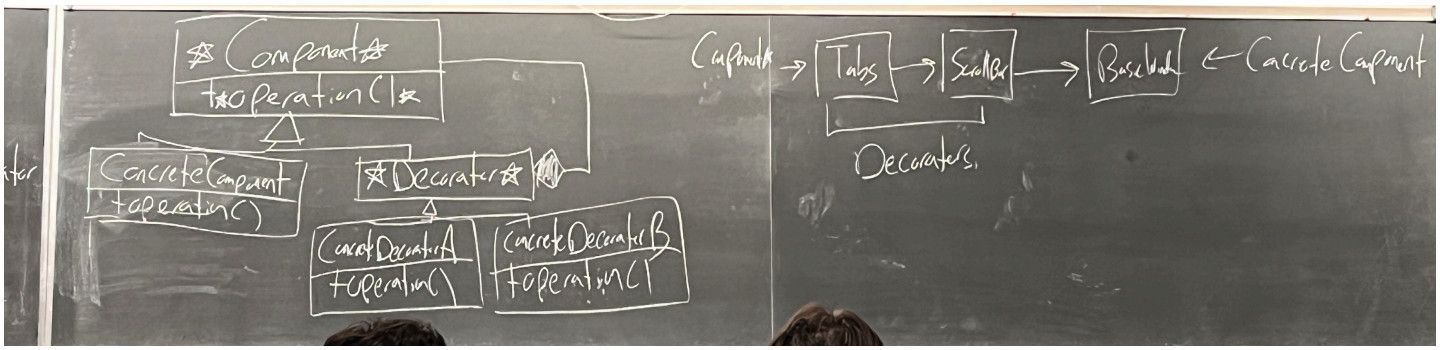
2^n subclasses: n is the number of teggleable features

Instead use decorator pattern

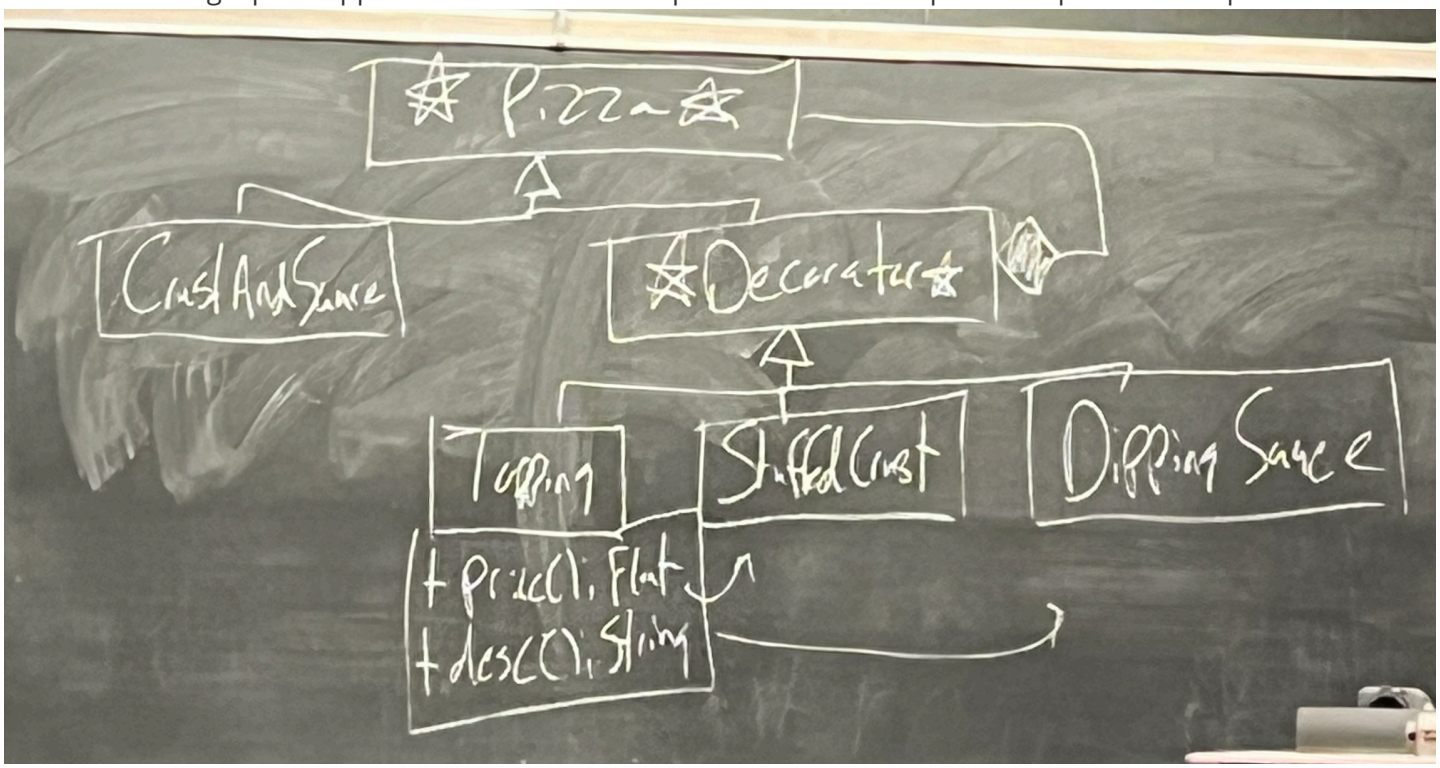



```
//This acts like a Linked List of functionality contrtecomponent describes default behaviour eg. BasicWindow concrete decorators augment functionality, using rest of the list
Components* -> Basic Window

//Add tabs:
Components* -> Tabs -> Basic Window
```

CS246 - Lec 17



Consider making a pizza application. Use decorator pattern to calculate price and print a description.




```
                        *Pizza*
                        --------
                        +*Price(); float*
                        +*desc(); string*
                              |
          ------------------------------------------------------
          |                                                    |
CrustAndSauce (Concrete)                                  *Decorator*
--------------                                                |
+price(): float                                  ----------------------------
+desc(): string                                  |            |              |
                                          Topping     Stuffedcrust    Dipping Sauce
                                          +price(); float
                                          +desc(); string
```

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

class CrustAndSauce:public Pizza {
public:
```

```
  float price() const override {return 7.99;}
  string desc() const override {return "pizza";}
};

class Decorator:public Pizza { // Decorator still abstract!: cuz it did not implment price
or desc methods
protected:
  Pizza* next;
public:
  Decorator(Pizza* p): next{p} {}
  ~Decorator() {delete next;}
};

class Topping:public Decorator {
  string the-topping;
public:
  Topping(string t, Pizza* p): Decorator{p}, the_topping{t} {}
  float price() const override{return 0.99+next->price();}
  string desc() const override{return next->desc()+"with"+the-topping;}
};

class Stuffedcrust:public Decorator {
public:
  Stuffedcrust(Pizza* p): Decorator{p} {}
  float price() const override{return 1.5+next->price();}
  string desc() const override{return next->desc()+"with Stuffedcrust";}
}

// acts like linked list
Pizza* myPizza = new CrustAndSauce{};
myPizza = new Topping{"pepperoni", myPizza};
myPizza = new Stuffedcrust{myPizza};
delete myPizza; // using dynamic type

delete myPizza; // Points to Stuffedcrust
    -> ~Stuffedcrust() (implicit, since not defined)
        -> ~Decorator()
            -> delete next; // Points to Topping
                -> ~Topping() (implicit, since not defined)
                    -> ~Decorator()
                        -> delete next; // Points to CrustAndSauce
                            -> ~CrustAndSauce() (implicit, since not defined)
                                -> ~Pizza() // The Pizza base class destructor is called
last
```

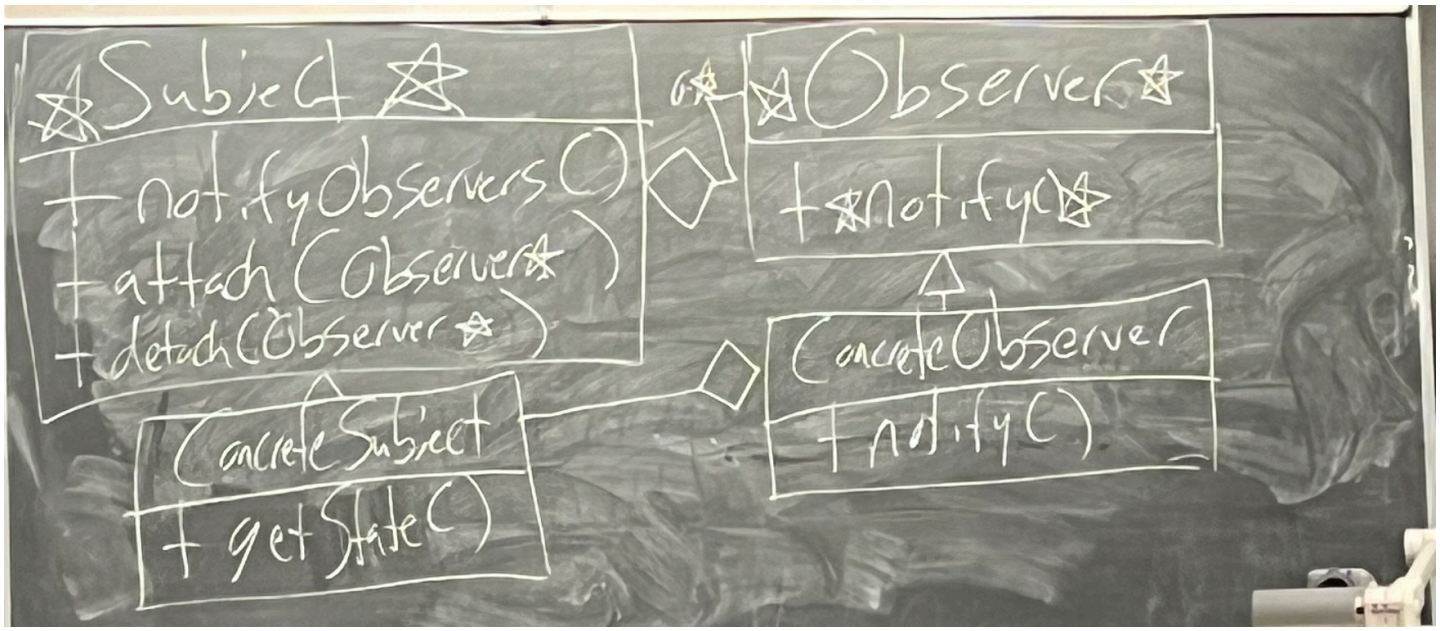
Oberserver Pattern

Problem: I want some number of classes to react to updates from data sources, perhaps in different ways.

consider a spreadsheet application. Some number of cells, some number of charts. Change a cell, all charts that depend on that cell update.

Different types of charts redisplay differently.

```
*Subject*                                   Observer
-------------------
+notifyobservers()
+attach(observer*)
+detach(observer*)
```



1. Concrete Subject has its data updated
2. Notifyobservers is called (inside or outside subject class)
3. Notify() is called for each observer in the subjects list
4. Resolves to calling notify for each concreteObserver
5. Observer calls getState to receive new data and do its job

Consider a simplified form of Twitter;

ConcreteSubjects: Tweeters, make a post, followers should be notified

ConcreteObserver: Followers, react on a tweetbeing made.

```
class subject {
  vector<observer*> observers;
public:
  void notifyobservers() const {
    for (auto p:observers) p->notify();
  }
  void attach(observer* o) {
    observers.emplace-back(o);
  }
  void detach(observer* o) {...o from vector}
  virtual ~subject() = 0;
};

subject::~subject() {}

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

class Tweeter:public subject {
  string lastTweet;
  ifstring file;
public:
  Tweeter(const string& fileName):file{fileName} {} // no need to call subject{}, since it
is a default ctor
  bool tweet() {
    file >> lastTweet;
    return file.good();
  }
  string getState() const {return lastTweet;}
};

class Follower:public Observer {
  string name;
  Tweeter* iFollow;
public:
  Follower(string name, Tweeter* t): name{name}, iFollow{t} {
    iFollow->attach(this);
  }
  void notify() override {
    string tweet = iFollow->getState();
```

```
    if (tweet.find(name) != string::npos) {
      cout << "Yay" << endl;
    } else {
      cout << "Boo" << endl;
    }
  }
}
int main() {
  Tweeter elon{"elon.txt"};
  Follower we{"we", &elon};
  Follower mary{"mary", &elon};
  while (elon.tweet()) {
    elon.notifyobservers();
  }
}
```

Lec 18

Last time: Observer, Decorator

This time: Exceptions, fatory Method Pattern

Revisit Vectors

```
v[c] - gets ith element of the vector
// if i is outside bounds - undefined behavior
v.at(i) - gets ith element of vector - if i is outside bounds, an error is signalled
rather than crashing
```

- How does error handling work in C?
  - Reserve a value like -1, INT_MIN, etc. Shrinks our return space, what should it be for other types, like Student.
  - Struct with an error field
    - Waste space for most returns
    - Easy to giniore error field
  - Use global variable, like error integer; Easy to ignore, may be written if multiple errors
- How to handle errors in C++? - **Exceptions**
  - to signal an error, an exception (exn) can be **raised** or **thrown**
  - Program then goes to a **handler** or **catch back** to deal with exn.
    - No handler found => then program crashes (**std::terminate**)
- If **v.at(i)** has i out of bounds, a **std::out_of_range** exn is thrown Found in

- Ex:

```
vector<int> v{1,2,3};
try { // this is used when u think it has some errors
  int n = v.at(100); // this throws an exception and immediately jumping to catch
  cout << n << endl; // so never get excecute in this case
} catch (out_of_range r) { // what types of exception you want to handle (r); catch must pair with try block
  cout << "Error" << r.what() << endl; // what is return string decribing what the error occur
}

out_of_range is just an object,
.what() is a method that returns a string describes the error that occurred.
```

```
void h() {g();}
void g() {h();}
void f() {throw out_of_range{"f throw"};} // creating an object; Return by what() - run ctor
// control flow transfers from f directly to the catch block. In doing so, we perform stack unwinding

int main() {
  try {
    h();
  } catch (out_of_range r) {
    cout << r.what() << endl;
  }
}
```

  - control flow transfers from f directly to the catch block. In doing so, we perform **stack unwinding**
    - In doing so, local variables in f, g and h are destroyed.

- Sometimes, we want to catch handler to perform some work, and then raise a new exn

```
try {....}
catch (out_of_range r) {
  // Do work, change vector, etc.
  throw invalid_argument{"Reprompt input"};
}

we can reraise an exn as well;
try {...}
catch (out_of_range r) {
  // Do some work;
  throw;
}
```

- Why **throw;** instead of **throw r;**?
  - **Inheritance**

    ```
    Error Situation
          ^
          |
    Special Error
    ```

```
try {....}
catch (errorSituation& e) {
  throw e; // If e is an ErrorSituation, this is fine.
} // If e is a specialError, then because throw e; performs a copy, object slicing occurs
```

- Whereas throw; does not copy -> No object slicing.
- In C++ exception handling, especially in the context of object slicing.

1. `throw e;` **- Copy and Potential Object Slicing:** In the code snippet you provided, if `e` is an instance of a derived class (like `specialError` which is derived from `errorSituation`), using `throw e;` will indeed cause object slicing. This happens because `throw e;` creates a copy of `e`, but only as its base class type (`errorSituation`). As a result, any additional information or properties that were part of the derived class (`specialError`) will be lost. This can lead to loss of information and potentially incorrect exception handling further up the call stack.
2. `throw;` **- No Copy, Preserving the Exact Exception:** On the other hand, using `throw;` in the catch block will rethrow the original exception object that was caught. This does not involve creating a new object or copying; it simply propagates the existing exception object. Therefore, it preserves the exact type of the exception, including any additional information if it's a derived class instance. This is crucial for accurate and effective exception handling, especially in complex systems where exceptions can be of various derived types with additional context or data.

- Can we write a genral handler?
  - Yes, using **catch-all syntax**

```
try{...}
catch(...) {...} // dots means catch all
```

- Although we usually throw objects, you can throw anything, including **primitives** like **int**, and your own types.
- You may also create your own exn types:

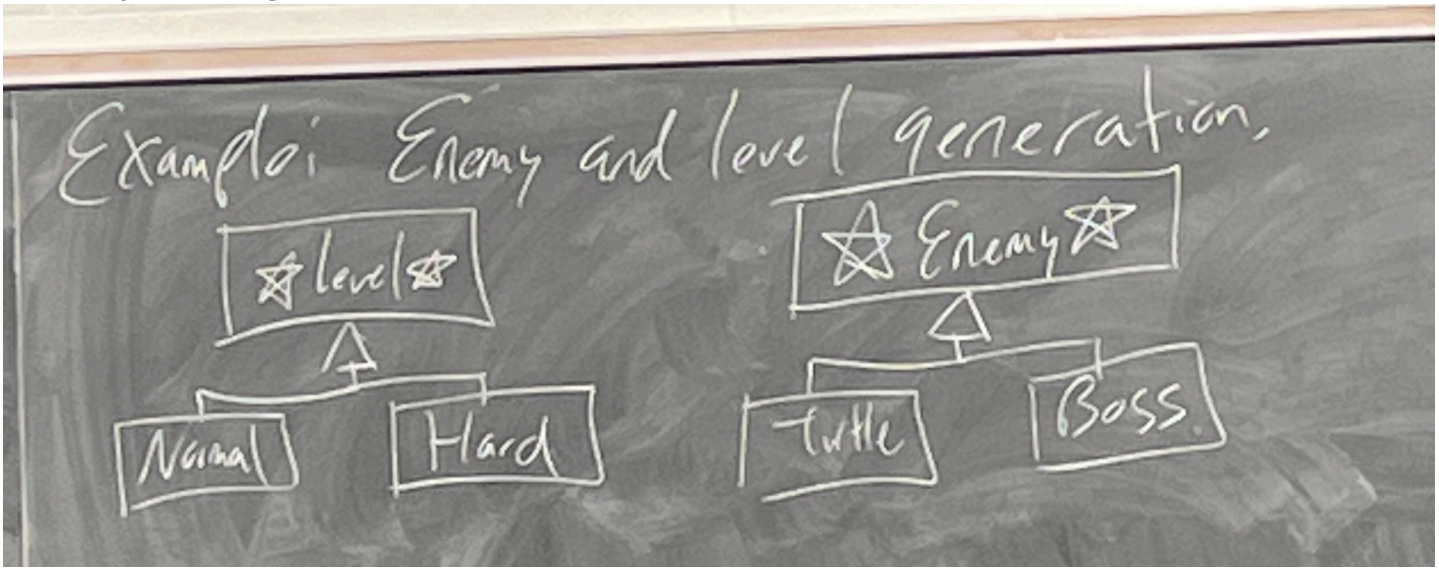
```
class BadInput{};
try {throw BadInput{};}
catch (BadInput & b) {...}
```

- Advice: throw by value, catch by reference, prevents object slicing.
- If we call new and OS rejects request for more memory -> **std::bad_alloc** is thrown
- Advice: **Do not let dtors throw an exception!**
  - **Dtors** have an implicit tag called noexcept -> if you throw, program immediately crashes! (**std::terminate**)
- If we tag our dtor as noexcept(false), it wont immediately crash.
  - However, can cause an issue with **stack unwinding**
  - **RULE**: At any time, we may only have one active exn;
- During stack unwinding, dtors run for stack allocated objects. This can cause two exns:
  1. The original exn that started unwinding
  2. Exn thrown from dtor.

Factory Method Pattern

Problem: Want to create different version of objects based on policies that should be easily customizable.

Ex: Enemy and level generation



Normal levels generate mostly Turtles, a feew Bosses

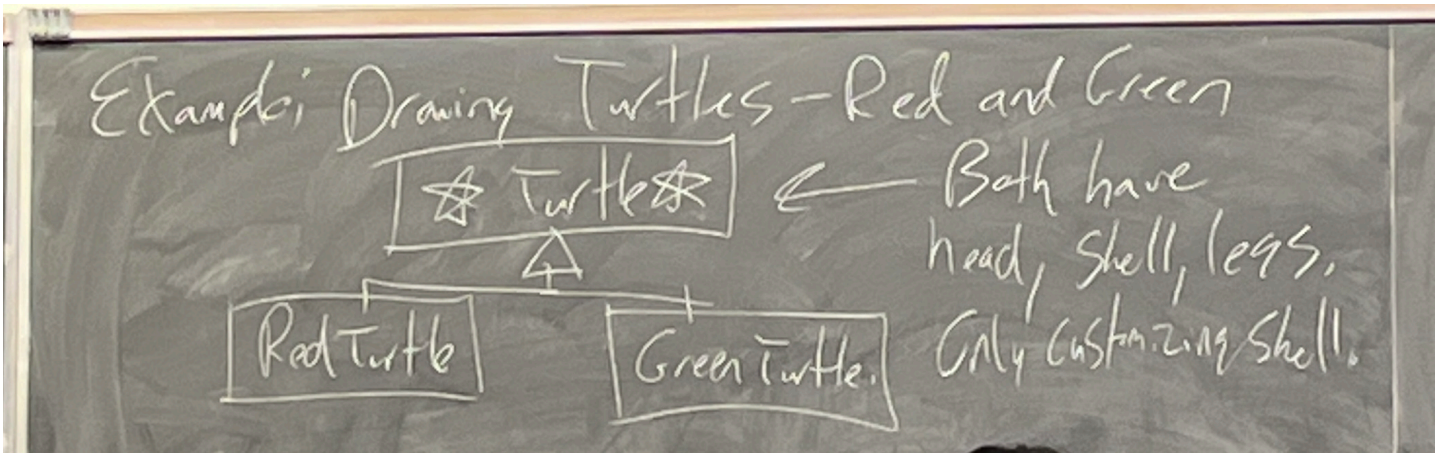
Hard levels: mostly bosses, some Turtles

```
class Level {
public:
  virtual Enemy* getEnemy() = 0;
  virtual ~Level() {}
};

class Normal:public Level {
public:
  Enemy* getEnemy() override {
    // mostly turtles, some bosses
  }
};

class Hard:public Level {
 public:
  Enemy* getEnemy() override {
    // mostly bosses, some turtles
  }
};

Level* l = ...;
Enemy* e = l->getEnemy();
// Use e and l, use their public methods.
```

```
// Easy to add newLevels, new Enemies, and new policies for making enemies
// Sometimes called virtual ctor pattern
```

Template Method Pattern

- Not related to C++ templates, just the name
- Problem: what if i only want **some behavior** that is customizable in the **subclasses**, **not all**
- Ex. Draing Turtles - red and green




```
class Turtle {
  void drawHead() {}
  void drawLeg() {}
  virtual void drawShell() = 0; // private pure virtual method
}

public:
  void draw() {
    drawHead();
    drawShell();
    drawLegs();
  }
};

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

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

// Ifwe call draw a Turtle*;
```

```
call Turtle::drawHead,
  (Red/ Green) Turtle::drawShell
  Turtle::drawLegs()
```

- What is the purpose of public methods?
    - Provide an interface to clients - with invariants, pre/post conditions, and a description of what the method does.
- Purpose of virtual methods - "an interface" for subclasses to override and change behaviour.
- What about **public virtual methods**? - sort of **contradictory** - they promise behavior to clients, while giving subclasses power to change behavior.
- No guarantee subclasses will respect invariants when overriding.
- **Template Method Pattren** may be generalized into non-virtual Idian (NVI)

NVI States

1. Public methods should be non-virtual
2. Virtual methods should be private or protected
3. Exception: dtor should be public virtual

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

class DigitalMedia { (with NVI)
  virtual void doPlay() = 0;
public:
  void play() {doPlay();}
};
```

- Benefit: **Flexibility**
    - We can add code that will always run by putting it before/after doPlay() - eg. Copyright, PlayCount, etc.
    - Can also add more "hooks" for custoization by adding private virtual methods like showCoverArt().
- All without **changing public interface**!
- And nust as fast as before - compiler will optimize out extra function call.
- Easier to use UVI from the start rather than adding it later.

STL Maps (Dictionary)

Array that can be indexed with different types than integer. Found in

```
map<string, int> m;
m["abc"] = 2;
m["def"] = 3;
cout << m["abc"] << endl; // giving 2
cout << m["ghi"] << endl; // giving 0
// if a key is not found, the value is default ctor for objects, or zero-initialized
(primitives)

if (m.count("abc")) {...}
return 0 if key not found, 1 if found.

m.erase("abc");

Iteration:
for (auto& p:m) {
  cout << "Key" << p.first << "value" << p.second << endl;
}

Type of p -> std::pair<string, int>&
```

- Pair is just a struct, first stores a string, second is an int. Used a struct, because it is just a callection of data, no invariants to preserve.

Or use a **structured binding**:

```
for (auto& [key, value]:m) {
  cout << key << " " << value;
}
```

- Used when structs have **public** fields

```
vec v{1,2};
auto [localX, localY] = v; // localX = 1; localY = 2;
// This is creating a copy of v
if:
auto& [localX, localY] = v;
// This could change the fields in v
```

- May also use for stack arrays if size is known;

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

What should go in a Module? / How can i tell if code is well-structured?

So fat => one class per module; Larger programs contain multiple classes and functions per module

Use **compling + cohesion**

Compling: To what **extent** do modules depend on each other.

- Low coupling: Simple communication via parameters/ Results
  - Communication via arrays or structures
  - Modules affect each other;s control flow;
  - Modules share global data
- High coupling: modules have access to each other's implementation (**friend**)
- Desire: Low coupling, makes program easier to change.
- Cohesion: How much do the parts of module relates to each other
  - Low Cohesion: Module parts are unrelated.
    - Module parts share a common theme -
    - Parts of module cooperate to manage lifetime state (road/ open/ write files)
  - High Cohesion: Parts of a module cooperate to perform one task
- **Dires: High cohesion**
  - Strive for low coupling and high cohesion
  - What about 2 classes that depend on one another.?

```
Does not work::
class A {
  int x;
  B y;
};

class B {
  int x;
  A y;
};

Because we cannot determine size of A or B
```

- Break chain of dependencies via a ptr:

```
class B;
class A {
  int x; // 4bytes for an int
  B* y; // 8bytes for a pointer
}
class B {
  int x;
  A y;
};
```

- We cannot forward declare from another module, so A and B must be in the same module
- Forward declaration is not allowed for object fields or for inheritance

Consider applying couping + cohesion to TikTacToe

```
class Board {
public:
  void play() {...cout <<"YourMove" << endl;...}
};

Book is coupled with cout - cannot reuse Board without getting print statement
- What if I want to print to a file? or not print at all?
```

```
class Board {
  istream&in;
  ostream&out;
public:
  Board(istream&in, ostream& out):in{in}, out{out} {}
  void play() {...out<<"Your Move" << endl...}
};
```

- Still coupled with streams - What if I want a graphical display? or a web API?
- Board should not do communication with user
- **Single Responsibility Principle:** Each class should have exactly one reason to change
- If changes to 2 different parts of the spec requries changes to the same class, SRP violated
- Board state + communication these are different things.
- Where to do communication> in main? **NO**
- Main function is not reusable like other classes are

CS 246 - Lecture 20

Last time: Maps, Couping, Cohesion

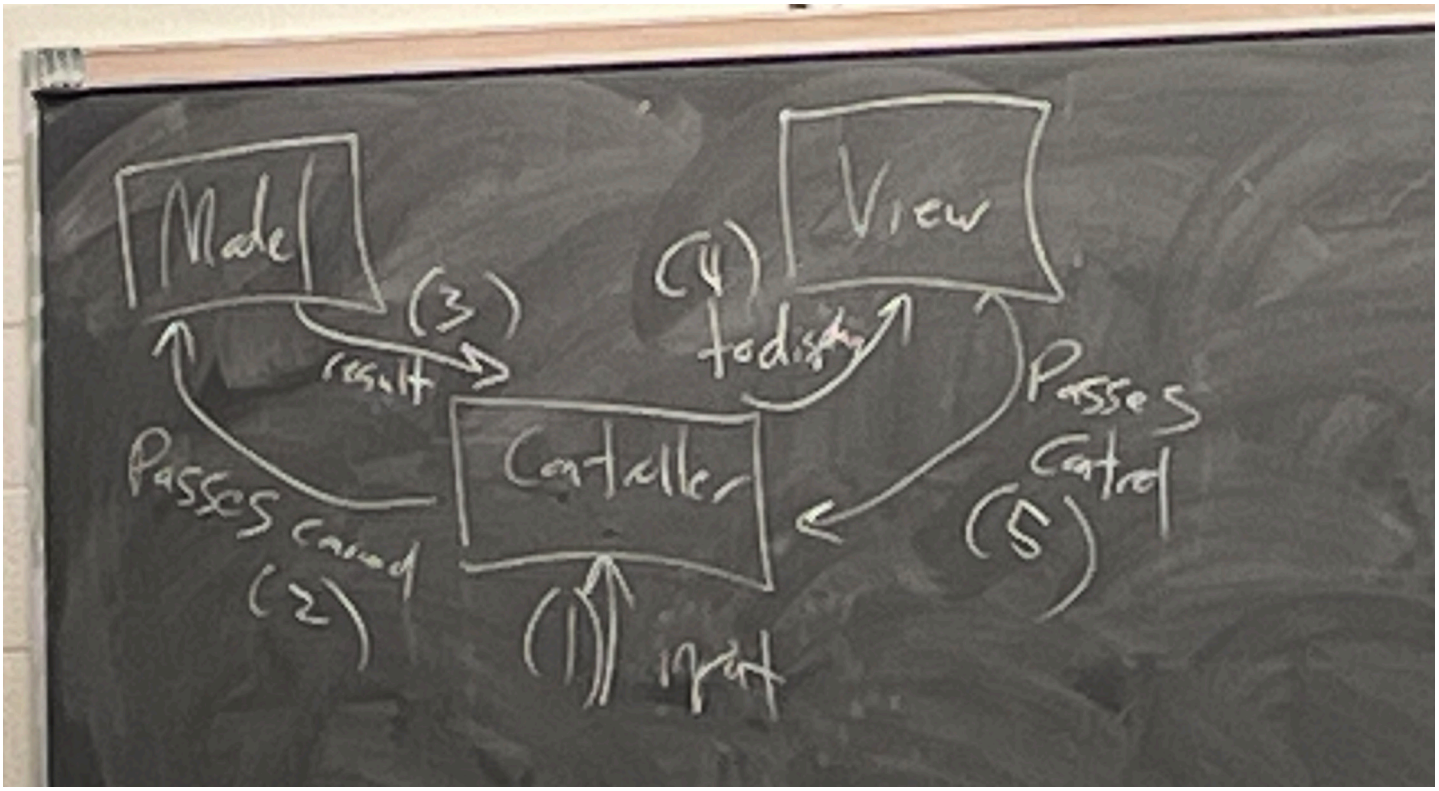
This time: MVC, Exception Safety, RAII,

```
Tic Tac Toe: class Board {
  ...
    cout << "Your Move" << endl;
};
```

If board should not perform communication, then who? Main? Generally no, main is not reusable

Alternative: Use **MVC** architecture - Model-view-controller

- Model - keeps track of data and logic surrounding data
- View - Output, communication with a user
- Controller - manages input, and control flow between model and view



Model - Keep track of data

- Communicates with controller via parameters / results
- Sometimes observer relationship between model and views
- Controller - May encapsulate logic for rules / ..
- Sometimes input is taken from view if it is a Window
- MVC promotes reuse of code

Exception Safety

```
// Assume C is a Class
void f() {
  C myC{};
  C* myCptr = new C{};
  g();
  delete myCptr;
}

Does f leak?
// Depends on g
```

- If g does not throw -> no leak

- If g throws -> control is immediately transferred to catch, stack unwinding runs my C dtor, but heap memory is leaked, delete was never called.

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

- Works, but it is clunky - duplicated code for deleting - have to repeat for any function that may throw - only gets more complex with more ptrs and more function calls
- We wanted to make sure **delete runs** if we leave f normally. or via exn. Other languages have finally. which always runs. C++ doesnot
- Guarantee: Stack allocated object dtors will run dwing stack unwinding.

Solution

- Wrap dynamically allocated memory in a stack allocated object
- And in general, prefer to use stack allocated objects

RAII - Resource acquisition is initialization

Ctor - acquires some resource

Dtor - releases the resource

```
{
  ifstream f{"file.txt"}; // file is accquired in ctor
} // file closed in dtor

std::unique-ptr<T> found in <memory> is an RAII class that manages dynamic memory
  Ctor - takes in a T*
  Dtor - deletes it for you

void f() {
    C myC{};
    unique_ptr<C> myCptr{new C{}};
    g(); // No leaks, whether we leave normally or via exn
  }
```

OR:

Use make-unique

```
unique_ptr<C> myCptr = std::make_unique<C>(...);
make_unique creates a C object in the heap for you using new, and the args you give to
make_unique
```

What happens if we copy?

```
auto p = make_unique<C>(...);
unique_ptr<C> q = p;  // Cpy Ctor for unique_ptrs
```

- Copy Ctor and Copy assignment - disabled for unique_ptr - code wont compile if we invoke them.
- If we pointed them at same location (shallow copy) -> double delete
- If deep copy, then pointer is meaningless

```
template <typename T> class unique_ptr {
  T* ptr;
public:
  explicit unique_ptr(T* ptr):ptr{ptr} {}
  ~unque_ptr() {delete ptr;}
  unique_ptr(const unique_ptr<T>& other) = delete;
  unique_ptr<T>& operator=(const unique_ptr<T>& other) = delete;
  unique_ptr(unique_ptr<T>&& other):ptr{other.ptr} {
    other.ptr = nullptr;
  }
  unique_ptr<T>& operator=(unique_ptr<T>&& other) {
    delete pre;
    ptr = other.ptr;
    other.ptr = nullptr;
    return *this;
  }
  T& operator*() {return *ptr;}
};
```

- If I need to copy a ptr: what should I do?
  - First answer: who is in charge of ownership?
  - Whoever owns the memory gets the unique_ptr
  - Everyone who just uses memory, access via **raw ptrs** - can get those via p.get()
- New understanding of ownership - can be signalled via type
- Unique_ptr - represents ownership, associated memory is deleted when it goes out of scope
- Raw ptrs - represent non-ownership, default is they do not free memory when they go out of scope
- **Moving** unique_ptr - **Transfer** of **ownership**

```
//Parameters
void f(unique_ptr<C> p) - f takes ownership of the unique_ptr - deleted when f
finishes
void g(*p) - ownership is not transferred from caller to g - g should just use p, not
delete it
```

- Results:

```
unique_ptr<C> f() - returns always move - ptrs is now owned by the caller
C* g() - caller should not delete returned value - stack memory, or owned by someone
else.
```

- Rarely, we may need true shared ownership - consider a graph data structure
- In such a case, we can use shared ptrs

```
{
  auto p = make_shared<C> {...}
  if (...) {
    auto q = p;
  } // q goes out of scope, does not delete p exists
} // p goes out of scope, nobody points at C object anymore, deleted.
```

- Shared_ptrs work by maintaing a reference count on creation, incremented. On deletion, decremented. When it reaches O, underlying object.

CS 246 - Lec 21

Last time: MVC, Exn, Safety. RAII, Smart ptrs

This time: Exn. Safety, vectors, casting

What is **exn safety**

- It is not that a function never throws, or that it handles all exes internally
- It is about guarantees we are given if we call a function and it throws or exn at us

1. Basic Guarantee - if an exn is thrown from a function f - then program is in a valid but unspecified state class invariants maintained, no leaks, no data corruption, but thats it

2. Strong guarantee - if an exn is thrown from f - then program state is reverted to **before function call.**

   i.e. - it is as if you never called f in the first place
3. Nothrow guarantee - if we call f, it will never throw and it will always do its job.

```
class C {
  A a;
  B b;
public:
  void f() {
    a.g();
    b.h();
  }
};
```

- Lets assume both A::g and B::h both provide **strong guarantee**.
- If **a.g() throws**
  - it undoes any side effects - so it is as if f was never called.
- If **b.h() throws**
  - it undo any of its own side effects, but it cannot undo a.g()'s work => **basic guarantee**

Since C::f has basic guarantee, can we change it to provide **strong guarantee**?

- Idea: Use temporary values, that way if we throw, a and b aren't changed. Must assume a.g() and b.h() have only local side effects for this to work.

```
class C {
  A a;
  B b;
public:
  void f() {
    A atemp = a; B btemp = b;
    atemp.g(); btemp.h(); // if either throws, a and b never change
    a = atemp;
    b = btemp;
  }
}
```

- This leaks like it provides strong guarantee, but if b = bTemp throws, we are modified a already - just basic guarantee.
- We need the ability to set a and b w/out throwing

Use pimples or (pointer to implmenetation) idiom. Then, swapping pointers - guaranteed to be nothrow.

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

class C{
  unique_ptr<cImpl> pImpl;
public:
  void f(){
    auto temp = make_unique<cImpl>(*pImpl);
    temp->a.g(); // strong guarantee
    temp->b.h(); // strong guarantee
    std::swap(pImpl, temp); // no throw
  }
};
// if either A::g, B::h offer no exception safety, then in general, C::f can't
// Exception Safety: The use of make_unique to create a temporary cImpl object ensures
that if either temp->a.g() or temp->b.h() throws an exception, the original state of C
(held by pImpl) remains untouched. Furthermore, std::swap for unique pointers is
guaranteed to be no-throw, so the final step of swapping pImpl with temp doesn't risk
throwing an exception. This provides a strong exception safety guarantee.
```

Vectors - RAII, Exn. safety

Vector v - represents ownership of C objects - when dtor runs for v when it goes out of scope - C objects are deleted, No Polymorphism!

Subclasses inserted into V will be sliced:

vector<C*> w - repressents non-ownership, when w goes out of scope - dtor will not delete what is pointed to by the C *"s. Supports polyphirsm - we can point subclass ptrs in the vector

```
vector<unique_ptr<C>> u - when u goes out of scope, dtor runs, frees associated memory -
ownership, and allows polymorphism.
```

- Vectors and exn. safety

  ```
  vector<T>: emplace_back - supports strong guarantee. If it throws, dynamically
  allocated array inside the vector is unchanged.
  ```

  - What happens when size == capacity>
    - Allocate new array W/ 2X capacity
    - copy each object from old array to new array if one throws, delete new array and rethrow

        - Point our old array ptr at the new array
- This is a little inefficient - why not move instead of copy?
    - Allocate new array
    - use move ctor to move each object to new array
    - Reassign array ptrs
- Does not provide strong guarantee - if a move ctor throws, our original array has been modified, because we stole data
- If move ctor will not throw will not throw - compiler will move elements in the vector rather than copy
- if move ctor may throw - compiler will resize vectors via copying
- At a minimum, moves and swaps should provide nothrow guarantee - tag them noexcept so compiler can facilitate optimizations:

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

Casting:

```
Node n;
int* ip = (int*) &n; // forces Node* to be treated as an int*
// This is C-style casting, not recommended in C++ dangerous, not type-safe
// C++ instead provides 4 casting functions for different situations.
```

1. Static_cast - for "sensible casts" with well-defined semantics. Float->int

```
float f;
void g(int x);
void g(float f);
g(static_cast<int>(f)); - calls int version of g
```

Static_cast allows us to downcast -

```
superclass* -> subclass*
Book* pb = ...;
Text* pt = static_cast<Text*>(pb);
cout << pt->topic << endl;
```

CS 246 - Lec 22

1. Static-cast - sensible, well-defined casts
2. Const-cast - Allows you to remove const from a type

```
void f(int*p) // Assume f does not modify the int p points to
void g(const int* p) {
  f(p); // wont compile
  f(const_cast<int*>(p)); // will compile
}
// If f changes p-undefined behavior
```

3. Reinterpret_cast_cast - take memory and reinterpret the bits stored there as a different type

```
Turtle t;
student* s = reinterpret_cast<student*>(&t); // Will point to the same memory location
// Behaviour depends on compiler and object layouts
// Gnerally unsafe
```

4. Dynamic_cast - Let us check which subclass I am pointing to

- Only works if you have at **least one virtual** method

```
Book* bp = ..; // super class
Text* tp = dynamic_cast<Text*>(bp);
// If bp points at a Text, tp will point at that same Text object
Otherwise - tp is set to nullptr

if (tp) cout << "Text" << endl;
else cout << "Not a Text" << endl;
```

5. We can also cast from shared_ptrs to other

```
 share_ptrs. In <memory>
static_pointer_cast, dynamic_pointer_cast, const_pointer_cast, reinterpret_pointer_cast
```

6. Dynamic cast references:

```
Book& br = ..;
Text& tr = dynamic_cast<Text&>(br); // since there is null reference, so instead, the
std::bad_cast exception is thrown
```

Recall polymorphic assignment problem:

- If operator= is non-virtual: partial assignment
- If operator= is virtual: mixed assignment
- Recall: signiture of virtual operator=

```
Text t1{...}, t2{...};
Book& r1 = t1;
Book& r2 = t2;
r1 = r2;

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

7. is dynamic_cast good style?

With dynamic_cast, we can make decisions based on the runtime type information of an object (RTTI)

- Tightly capled to book hierarchy - If I add a subclass, must add another else-if. This must be done everywhere I do this pattern
- If I miss one, it is a bug
- If dynamic_cast a bad style?
    - Depends on use
    - Use in Text::operator= does not need changing if we add more subclass types

```
void whatIsIt(shared_ptr<Book>b) {
  if (dynamic_pointer_cast<Text>(b)) cout << "Text";
  else if (dynamic_pointer_const<comic>(b)) cout << "Comic";
  else cout << "regular book";
}
```

- Fix whatIsIt via virtual method

```
class Book {
public:
  virtual void identify() {cout<<"BOOK"<<endl;}
};

class Text:public Book {
public:
  void identify() override {cout << "Text" <<endl;}
};

void whatIsIt(book* b) {
  if (b) b->identify();
  else cout << "Nothing";
}
```

- We were able to use this solution of identify
  - There are a high number of possible Book subtypes and we want to add new ones with ease
  - Books can have a uniform interface, subclass behaviour does not deviate too much
- What about opposite scenario
  - Consider we know subclass types in advance and we are Ok that adding new subclasses will require extensive code changes
  - Subclasses may not conform a uniform interface - each way have significantly differenting behavior

```
class Turtle:public Enemy {
  void stealshell();
};

class Boss:public Enemy {
  void epicBossBattle();
};
```

- Adding new enemies will requrie large changes as each may have unique behavior - so maybe dynamic_casting is not so bad.
  - another option is using std::variant
- Variant
  - found in acts as a type-safe union;

```
using Enemy = variant<Turtle, Boss>; // type alias - Enemy means variant<Turtle, Boss>;
Enemy e {Turtle{...}}
// or
Enemy e {Boss{...}}
```

```
if(holds_alternative<Boss>(e)) {
  // it is a boss
} else {...}
// Access value
try {
  Turtle t = get<Turtle>(e);
} catch(std::bad_variant_access&  ) {..}

// If variant is left uninitialized - eg Enemy e;
default construct first type in variant list.
Wont compile if first type is not default constructable
```

- IF first type does not have a default ctor:
  1. Add a default ctor
  2. Reorder types so first has a default ctor
  3. Use std::monostate - represents empty

  e.g.

  ```
  using Enemy = variant<monostate, Turtle, Boss>;
  ```

  ```
  std::optional<T> = variant<monostate, T>
  ```

How do virtual methods actually work?

```
struct vec {
  int x, y;
  void f();
};

struct vec2 {
  int x,y;
  virtual void f();
};

// Are the following in different memory?
vec v;
vec2 w;

cout << sizeof(int) << sizeof(v) <<sizeof(w); // 4 8 16
```