CS 246: Object-Oriented Software Development

Jason Hon

Spring 2024, University of Waterloo

Caroline Kierstead

Contents

1 Lecture 1

1.1 I/O

Definition 1.1: Streams

`cin` — standard input
`cerr` — standard error

Consider an example where we read in two numbers and a string.

```
1  import <iostream>;
2
3  using namespace std;
4
5  int main() {
6    int x, y;
7    cin >> x >> y; // Read x then read y
8    if (!cin.fail()) // Error Checking
9    string s;
10   cin >> s; // Read until whitespace
11 }
```

To read in numbers and outputs, consider the following code

```
1  import <iostream>;
2
3  using namespace std;
4
5  int main() {
6    int i = 0;
7    while (true) {
8      cin >> i;
9      if (cin.fail()) break;
10     cout << i << endl;
11   }
12 }
```

Definition 1.2: Arrows

<< Put to operator
>> Get from operator

2 Lecture 2

2.1 Function Overloading

Definition 2.1: Function Overloading

The compiler determines which version of the function to call based on the number and/or types of parameters but **not** return types.

Careful! 2.1

Must be unambiguous

```
1    // Multiple types of a function
2    int operator>>(int, int); // Function prototype
3    std::istream operator>>(std::istream in, char c);
4    std::istream operator>>(std::istream in, int i);
```

2.2 Error Handling

Question: 2.1

If reading in 2 ints and "fails", what happens?

```
1    int main() {
2        int x, y;
3        cin >> x >> y;
4        return 0;
5    }
```

Possible errors:

- Could run out of input (EOF) for either x or y
- Reading value smaller than INT_MAX or smaller than INT_MIN; (<climits>)
- Reading in not an integer

Question: 2.2

How can we detect failure?

std::cin is an instance of std::istream
has state bits that tell us the condition of cin → read first then check

```
    cin.good() // returns true if succeeded
    cin.eof() // * reaches the end of the file (not a char, its a state)
    cin.fail() // * true if either EOF or didn't get int
    cin.bad() // unrecoverable error (not commonly used)
```

Consider the following code of reading an integer

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

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

To check errors, we can use this shortcut

```
// std::istream has a function that can convert an istream to a bool
// i.e. std::istream::operator bool()) =? !cin.fail()
// e.g. if (cin) ...
while (cin >> i) { // only true if successfully reading an int
    cout << i << endl;
}
// std::istream can call ! on cin => bool std::istream::operator!()
// that returns std::istream::fail()
// e.g.
while (true) {
    cin >> value;
    if (!cin) { // equivalent to cin.fail()
        break;
    }
}
```

Question: 2.3

How do we revise the program to "throw-away" non-ints, output ints and stop on EOF?

```
while (true) {
    cin >> value;
    if (!cin) {
        if (cin.eof()) break;
        cin.clear(); // reset all state bits to original
        cin.ignore(); // removes 1 char (by default) [take it out of the input
        ↪ stream]
    } else {
        cout << value << endl;
    }
}
```

2.3 Formatting I/O

C++ has "manipulators" to format I/O

```
import <iomanip>;
/*
  can be used to :
  - print things as hexa/octal decimal
  - boolalpha, setw, setfill, skipws [on by default], noskipws
  (wont be used in this course)
*/
cout << hex << value // all subsequent #s are in hexadecimal i.e. strictly cout <<
↪ dec;
```

Remark.

Best practice: undo any sticky changes

2.4 Strings

Definition 2.2: Strings

C uses `const char *` and `char *` where every string needs a null terminator.

- length via dynamic memory allocator requires +1
- append requires realloc, need to worry about memory leaks
- In C++, we use `<string>`, is part of the std namespace

Careful! 2.2

With G++11, always import and compile string last

```
import <string>;
int main() {
    std:string s1; // empty string, s1.size() == 0
    std::string s2 = "LOL";
    std::cin >> s1; // reads white-spaced eliminated word
    std:: cout << s2 << s1 << endl;
    s1 += s2;
    std::string s3 = s2 + s2;
    std::string line;
    getline(std::cin, line); // reads entire line including whitespace up till \n
}
```

3 Lecture 3

3.1 Strings

```
1  void f(string s) {
2      cout << s << endl;
3  }
4
5  int main() {
6      string s1 = "Stuart";
7      string s2{" is a good cat"};
8      s2[11] = 'b';
9      if (s1 != s2) ... // lexicographic comparison (s1 < s2)
10
11     f(s1);
12     f(string{"stuart"});
13 }
```

3.2 Streams

Stream is an abstraction wrapped around input and output (e.g. keyboard, screen, files etc.) We already know

- std::cin (std::istream)
- std::cout, std:cerr (std::ostream)

```
1      std::istream *ip = &std::cin; // a1 q3
```

3.3 Files

Files are another form of stream
Reading in file in C:

```
1  #include <stdio.h>
2
3  int main() {
4      char s[256];
5      FILE *f = fopen("file.txt", "r");
6      while (true) {
7          fscanf(f, "%255s", s);
8          if (feof(f)) break;
9          print("%s\n", s);
10         fclose(f);
11     }
```

```
12 }
```

In C++:

```
1  import <iostream>;
2  import <fstream>;
3  import <string>; // import and compile string last!!
4  using namespace std;
5
6  int main() {
7      string s;
8      ifstream in {"file.txt"}; // open in read mode
9      while (in >> s) { // in.fail() true if cannot read
10         cout << s << endl;
11     }
12 } // out of scope file closed
```

3.4 File Stream

An std::ifstream can do everything an std::istream can (will revisit when discussing inheritance)

```
1      std::istream *ip2 = &in; // legal!
2      void bar(std::istream *in) { ... }
3      bar(&cin);
4      bar(&in);
5      bar(std::ifstream{"in.txt"});
```

3.5 String Streams

- library: sstream => combination of string + stream
- std:: ostringstream (used to convert and possibly concatenate data that can be then retrieved as a C-style string)

```
1  std::string intToString(int i) {
2      std::ostringstream oss;
3      oss << i;
4      return oss.str();
5  }
```

Until we cover exception std::istream is the only way to convert a string to an int

```cpp
int StringToInt(std::string s) {
    std::istringstream iss{s};
    int i = 0;
    iss >> i; // iss.fail() is true if it could not read an int;
    return i;
}
```

```cpp
// stops when EOF or invalid int
int i;
while (true) {
    cout << "Enter a number: ";
    string s;
    cin >> s;
    if (cin.eof()) break;
    if (istringstream iss {s}; iss >> i) break; // have an int
    cout << "I said, ";
}

// repeats until EOF, outputting ints
string s;
while (cin >> s) {
    int n;
    if (istringstream is {s}; is >> n) {
        cout << n << endl;
    }
}
```

3.6 Command line arguments

```
    ./pgm abc 123 < file.in 2 > err.txt 1 > out.out
              ^   | Bash redirection, not cmd line |
         Args to program
    |  C/C++ args |
```

```
int main(int argc, char *argv);

| "./pgm\0" | "abc\0" | "123\0" | null ptr |
  ^    0          1         2        3 = argc
argv
```

To read in from arguments, consider from the following code

```
import <iostream>;
import <sstream>;
import <string>;

using namespace std;

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

4 Lecture 4

4.1 Overloading

Definition 4.1: Overloading

The same function name is used but the parameters list must differ in numbers and/or parameter types.

Remark.

Return values are not part of consideration! Calls must be unambiguous at compile time!

```
1  bool negate(bool b) {
2      return !b;
3  }
4
5  int negate(int i) {
6      return -i;
7  }
8
9  int main() {
10     negate(true); // valid!
11     negate(5); // valid!
12     return 0;
13 }
```

4.2 Print Suite

A function that either prints the "stem", one per line for the specified file name or the file "suite.txt"

```
1 printSuiteFile(); // outputs suite.txt
2 printSuiteFile("stuart.txt"); // outputs "stuart.txt"
```

Implementation:

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

void printSuiteFile(string fname) {
    ifstream in{fname};
    for (string s; in >> s) {
        cout << s << endl;
    }
}
```

You can give a parameter a default value. If fname is not specified, compiler puts in "suite.txt".

```
// Header
void printSuiteFile(string fname="suite.txt");
```

The only time we know the actual number of parameters is at the call site when we compile. The compiler generates any missing information using the specified default value ("suite.txt"). This allows the function to retrieve the full parameters list of values off of the runtime stack. Otherwise, could try to access missing info from where it shouldn't.

Careful! 4.1

All default values must be placed at the end of the parameter list!

```
bar(int i, j, k, l) { ... }
bar(int i, j, int k = 1, int l = 3) { ... }
```

The values **must** be given in the declaration if separate compilation

4.3 Structures

C++ is backwards compatible with C. Code fragment in C:

```
struct node {
    int value;
    struct node *next;
}

typedef struct node Node;

// Implementing node
Node n; n.value = -1; n.next = NULL;
```

In C++, we do not need typedef

```
struct Node { // types start with a capital letter
    int value;
    Node *next;
}
// Implementing node
Node n; n.value = -1; n.next = NULL;
```

Question: 4.1

Why doesn't this node definition work?

```
struct Node {
    int value;
    Node next; // compile error, missing a '*'
}
```

While defining the Node type, the size (amount of memory to allocate) is **unknown**. "next" is a Node, but don't know the size so cannot define it.

4.4 Constants

Old C way uses `#define` but it isn't type safe. (may substitute it in illegal locations)

```
#define MAX_GRADE 100
```

C++ uses "const" keyword (do not use magic number)

```
const int MAX_GRADE = 100;
```

It cannot change a constant value, make const on what should not change.

```
Node n1 {5, nullptr}; // aggregate initialization: value: 5, next: nullptr;

const Node n2;
n2.value = 5; // INVALID
const Node n3 {n1}; // copies n1 into n3, immutable, constant
const Node n4 {-1, &n1};
n4.next.value += 1;
```

4.5 Parameter Passing

Definition 4.2: Parameter Passing in C

In C, there is pass by value and pass-by-reference (pointer)

```
1  void inc(int i) {
2      ++i;
3  }
4  void inc2(int *i) {
5      *i += 1;
6  }
7
8  int i = 4; inc(i);
9  cout << i; // outputs 4 still
10 inc2(&i);
11 cout << i; // outputs 5;
```

If I have:

```
1 int i;
2 cin >> i; // reads 4
3 cout << i; // outputs 4;
```

Question: 4.2

Why no & anywhere?

Definition 4.3: Parameter passing in C++

C++ Introduces "references" as a third parameter passing mechanism. Reference is a constant pointer that automatically dereferenced

```
1 std::istream & operator>> std::istream & in;
2 int & value;
```

5 Lecture 5

5.1 Left Value and Right Value

Definition 5.1: lvalue (left value)

What can appear on the LHS of an assignment.

- It has a name / pointer / reference that allows access

Definition 5.2: rvalue (right value)

Opposite of an lvalue. e.g. Anonymous object, the result of an expression
`int x = 5;`
where x is the lvalue, and 5 is the rvalue.
`Node n = Node{5, nullptr}`
where n is the lvalue, and `Node{5, nullptr}` is the rvalue

Definition 5.3: Reference

A reference is a constant pointer that is automatically dereferenced

```
1 int i = 5;
2 int & ref = i; // ref pointing to i
3 ref += 2; // i becomes 7
```

Remark.

> `int &` is a lvalue reference
> If `'&'` appears as part of an expression, probably taking an address of a bitwise and; if immediately follows a type, it's a reference

5.2 Lvalue Referencing

```
1 int main() {
2     int i = 5; // &i = 0x00004
3     int &j = i; // j = 0x00004
4     int &k = j; // k = 0x00004
5 }
```

Careful! 5.1: Common Traps

- Since reference are constant, it must be bound to an lvalue upon declaration e.g.
  `int & j1; // wrong;`
  `int & j2 = i // ok`
- Cannot bind an lvalue reference to an rvalue. e.g.
  `int & j = 5; // wrong`
- Cannot have a pointer to a reference (it won't compile). e.g.
  `int &* irp = &j; // wrong`
  However, a reference to a pointer is fine!
  `int *p;`
  `int *&q = p;`
  `q` is a reference to a pointer to an integer.
- Cannot have an array (or STL containers e.g. `std::vector`) of references.

Question: 5.1

When do we use references?

It is mostly used for passing parameters. It may sometimes be returned by reference.

```
1 void inc (int & n) { // lvalue reference, bound to that memory location
2     ++n;
3 }
4
5 void main() {
6     int n = 5;
7     inc(n);
8     cout << n << endl; // print 6
9 }
```

5.3 Choice of Mechanism

Motivating Example:

```
1 int main() {
2     int i;
3     cin >> i >> j;
4 }
5
6 std::istream & operator>>(std::istream & in, int & i);
```

Question: 5.2

What mechanism should we use when passing information?

Consider a really big struct object. e.g.

```
1 struct ReallyBig { ... };
2 ReallyBig obj;
```

1. Pass by value: `void f(ReallyBig o);// makes a copy`
   Subtleties (discussed later) leads to time cost
2. Pass by pointer: `void f(ReallyBig *ptr); // address is copied`
   contents can be changed, it is cheap in cost (8 bytes) but pointers (have to dereference)!
3. Pass by reference: `void f(ReallyBig &rbref); // cost of passing lvalue ref`
   A constant pointer is passed but the original (obj) can be changed
4. Pass by const reference: `void f(const ReallyBig *crbref);`
   It has all the benefits of pass by reference but the original is immutable

Question: 5.3

What if need function to make changes as part of the algorithm but don't want to change the original?

Either make a local copy and change that or pass by value. (Done if only absolutely required as the cost is heavy)

Question: 5.4

If we do not need to make a copy, what choices do we have?

- If the value doesn't have to be changed and the size is an integer or less, prefer pass by value.
- If the value doesn't have to be changed but is larger than an int, prefer pass it by a const reference.
- If it should be changed, pass by reference (avoid pass by pointer)

Question: 5.5

What about where we want to pass an int by reference but could be called as either:

```
int main() {
    int i = 5;
    f(5);
    f(i); // for both to be valid?
}
```

It must be `void f(const int &i);`. It allows `f(5);`

5.4 Dynamic Memory Allocation

- C uses alloc / calloc / malloc / realloc and `void *` so it is not type safe
- C++ has no realloc but is type safe (it uses `nullptr` and not `NULL`)

```
int *p1 = nullptr;
*p2 = new int {5};
*p2 = 13;
delete p1; // ok!
delete p2; // ok!

p2 = new int;
*p2 = -1;
delete p2; // also ok
delete p2; // illegal! double free

// may show up on the exam
int i = 0;
p2 = &i;
delete p2; // illegal! freeing stack memory
```

```
int size;
int *arr = new int [size];
arr[0] = -1;
delete [] arr;

Node **ar2 = new Node* [10] {nullptr}; // initializes everything to nullptr
ar2[5] = new Node {6, nullptr};
...
delete ar2[5]; // must do this before deleting array
...
delete [] ar2; // does not delete the heap-allocated nodes
```

Careful! 5.2

DO NOT mix the forms of new and delete. If allocated with `new xxx [x]`, free with `delete [] xxx` and vice versa.

use valgrind for all programs with dynamic memory allocation

6 Lecture 6

6.1 Dynamic Memory Allocation

```
1 Node *np = new Node {5, nullptr};
2 ...
3 delete np;
```

- np is on the stack
- freed when goes out of scope (i.e. local)
- If it points to heap-allocated memory, it must be freed

```
1  // allocating on heap
2  int **arr = new int* [5];
3  for (int i = 0; i < 5; ++i) {
4      arr[i] = new int [10] {0}; // set all as 0
5  }
6  ...
7  arr[0][0] = -1;
8  ...
9  // freeing
10 for (int i = 0; i < 4; ++i) delete [] arr[i];
11 delete[]arr;
```

Careful! 6.1

It is undefined behaviour to mix the 2 forms of new / delete!

Remark.

Remember, a memory leak is a failure to free heap-allocated memory

- leads to eventual failure
- In CS246, invalidates program.

6.2 Returning Information

6.2.1 Return by value

`ReallyBig f(); // makes a copy (elision??)` Potentially expensive

6.2.2 Return by pointer

```
ReallyBig *f() {
    ReallyBig rb;
    ...
    return &rb; // returns the address of a local variable
}
```

This is a dangling pointer, potentially dangerous. Revised version:

```
ReallyBig *f2() {
    ReallyBig *rb = new ReallyBig;
    ...
    return rb; // rb is on heap, it still persists
}

int main() {
    ReallyBig *p = f2();
    ...
    delete p;
    return 0;
}
```

6.2.3 Return by reference

```
// Wrong
ReallyBig &f() {
    ReallyBig rb;
    ...
    return rb; // reference to no longer existing local variable
}

// Correct
ReallyBig &f2() {
    ReallyBig *p = new ReallyBig;
    ...
    return *p;
}  // have to know that the reference is heap-allocated

int main() {
    ReallyBig &ref = f2();
    ...
    delete &ref; // gets the heap allocated address
}
```

Question: 6.1

`operator>>` and `operator<<` for I/O returns the stream by reference and it works?

Remember `cin >> i` is using a **global** variable so `operator>>` is returning the parameter it received by reference.

```
1 std::istream &operator>>(std::istream &in, int &i);
```

Question: 6.2

Which returning technique should we use?

Most of the time, return by value since turns out not to be that expensive in practice due to its elision (see this later).

6.3 Operator overloading

Can overload all operators except for:

1. `::` (Scope resolution operator)
2. `.` (Member access)
3. `*(Node.next)` (Dereference + member access)
4. `?` (ternary operator)
5. `**, <>, &|` (New operators [not covered yet])

We overload these to help provide an intuitive understanding of what they do. A potential use case is extending functionality:

```
1  struct Vec {
2      int x, y;
3  };
4
5  Vec operator+(const Vec &lhs, const Vec &rhs) {
6      return Vec {lhs.x + rhs.x, lhs,y + rhs, y};
7  }
8
9  std::ostream& operator<<(std::ostream& out, const Vec& v) {
10     out << '(' << v.x << ", " << v.y < ")";
11     return out;
12 }
13 ...
```

```
Vec v1 {0,1}, v2 {1. 0};
cout << v1 + v2 << endl; // Outputs "(1, 1)";
```

Question: 6.3

What if I want to do a scalar multiplication where I can do `v2 * 5; 3 * v2;` (both ways)

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

Vec operator*(const Vec& v, int s) return (s * v); // calls prev func
```

```
struct Grade {
    int g;
};

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

ostream& operator<<(ostream& out, const Grade& g) return out << g.g << "&";
```

6.4 Separate Compilation

Recall the following definitions from CS 136

Definition 6.1: Declaration

Statement of existence

Definition 6.2: Definition

Full description of type or function.

- Allocates space for functions and variables
- How big is struct/class (later: what methods exist)

Definition 6.3: Interface

Collection of declarations and class definitions

- Multiple declarations are ok, but can only have 1 definition.

Definition 6.4: Implementation

Where function / constants / globals are defined.

C++20 modules guarantee single definitions

- It can be made up of multiple interface and implementation files.
- When writing a module for `Vec` convention in CS246:
  - Interface file called `vec.cc`
  - Implementation file called `vec-impl.cc`

7 Lecture 7

7.1 Seperate Compilation (cont')

Sample Interface file for Vec

```
1  // c++20 module
2  // interface: vec.cc
3  // DO NOT PUT namespace std in the interface file!!!!!
4  export module vec; // first line!
5  import <iostream>; // after the module export
6  export struct Vec {
7      int x, y;
8  };
9
10 export Vec operator+(const Vec& v1, const Vec& v2);
11
12 export std::ostream& operator<<(std::ostream& out, const Vec &v);
```

When using a preprocessor, do not need exports

```
1  // c++20 preprocessor
2  // interface: vec.h
3  #ifndef VEC_H // not always needed but best practice
4  #define VEC_H
5  #include <iostream>
6  struct Vec {int x, y};
7
8  Vec operator+(const Vec& v1, const Vec& v2);
9
10 std::ostream& operator<<(std::ostream& out, const Vec &v);
11 #endif
```

Implementation File for Vec (module version)

```cpp
// c++20 modules
// file: vec-impl.cc
module vec; // must be 1st line, reads in interface file.

Vec operator+(const Vec &lhs, const Vec &rhs) {
    return Vec {lhs.x + rhs.x, lhs,y + rhs, y};
}

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

Implementation File for Vec (preprocessor version)

```cpp
// c++20 preprocessor
// file: vec-impl.cc
#include "vec.h" // the only difference!

Vec operator+(const Vec &lhs, const Vec &rhs) {
    return Vec {lhs.x + rhs.x, lhs,y + rhs, y};
}

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

To use the module file in the main (module):

```cpp
// c++20 module
// file: main.cc
import vec; // no square brackets since not a system file
int main() { ... }
```

For the preprocessor version:

```cpp
// c++20 preprocessor
// file: main.cc
#include "vec.h"
int main() { ... }
```

7.2 Module Compilation

1. Compile system headers using g++20h
2. **Must** compile in dependency order!
   (a) `g++20m -c vec.cc`
   (b) `g++20m -c vec-impl.cc`
   (c) `g++20m -c main.cc`

   This creates corresponding `.o` files
3. `g++20m vec.o vec-impl.o main.o` (link to create `a.out`) [order doesn't matter]

Careful! 7.1: Cheap Hack

1. Run: `g++20m -c *.cc` twice
2. Link: `g++20m *.o -o name`

Repo has compile "bash script": `./compile files.txt a.out`
file.txt:
`vec.cc`
`vec-impl.cc`
`main.cc`

Careful! 7.2: Preprocessor Compilation

DO NOT compile `.h` files!
`g++20i -c *.cc // creates *.o`
g++20i *.o -o name // links
Can use `tools/Makefile`

7.3 Classes

Definition 7.1: Class

A "class" (usually) contains data (i.e. data fields) and functions, also known as "methods" or "member functions

```cpp
// header file
struct Student {
    int assns, mt, final;
    float avg(); // do not write the code inline
};
```

```cpp
 // implementation file
float Student::avg() return 0.4 * assns + 0.2 * mt + 0.4 * final;
```

Student is the class name, and `::` is the scope resolution operator.

Definition 7.2: Instance

An instance variable of the class (Type) is called an object

```cpp
Student s; // data fields aren't automatically set to 0;
s.assns = 60;
Student stuart {70, 69, 85}; // Aggregate Initialization since fields are public
cout << stuart.avg() << endl;
```

`Stuart` is a receiver / recipient of the method call, `avg()` is the method call. It has implicit hidden first parameters `this`. `this = &stuart`

- `this->assns, this->mt, this->final`

Definition 7.3: Constructors

- Methods used to initialize objects
- Benefits:
  - Can be overloaded
  - Can perform more complicated code
  - Sanity checks
  - Can have default parameters
- Lose aggregate initialization if provided any constructor and/or the data fields aren't public
- A "default constructor" is a constructor with 0 parameters
  - Compiler gives you a default constructor for free [subject to certain limitations] if you don't write any constructors
  - Calls default constructor for all data fields that are objects

```
struct Student {
    int assns, mt, final;

    Student() { // put this in the implementation file (don't do it inline)
        this->assns = 0;
        this->mt = 0;
        this->final = 0;
    }

    Student(int assns=0, int m=0, int f=0) {
        this->assns = a; // this->assns specifies
        this->mt = m;
        this->final = f;
    }
};
```

Using the class:

```
Student s1; // {0, 0, 0}
Student s2{}; // {0, 0, 0}, both semantically same
Student jim{45}, ellen{80, 90}, fred{60, 30, 20};
/*
jim: 45, 0, 0
ellen: 80, 90, 0
fred: 60, 30, 20
*/
Student s3 = Student{60, 70, 80};
// before c++17, the rvalue Student copied into s3
// c++17 and up, standard requires "elision" so  equivalent to:
Student s3{60, 70, 80};
```

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

Vec v1; // illegal, needs 2 params
Vec v2{0, 1}; // ok!
```

8 Lecture 8

8.1 Member Initialization List (MIL)

Recall `struct Vec` from the last class.

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

Basis b; // illegal since Vec no longer has default constructors
```

Question: 8.1

Can we initialize `v1` and `v2` in Basis' default constructor?

```
Struct Basis {
Vec v1, v2;
Basis() { v1{0, 1}; v2{1, 0};} // making the ctor call for data fields in the body
  ↪ of the ctor is too late!
}
```

Consider the steps for object creation (for now):

1. Allocate enough space for all the object fields (allocates in declaration order)
2. for all data fields that are objects, all their default constructors
3. Run the constructor body

If we use the member initialization list (MIL) syntax, we can call a constructor other than the default in step 2.

Remark.

It must also be used to initialize references and constants

Pattern: `<ctor-name>(<param-list>)` : $f_1\{p_1\}, f_2\{p_2\}$, { body of ctor }

```
struct Vec {
    int x, y;
    Vec(int x, int y) : x{x}, y{y} {}
};
struct Basis {
    Vec v1, v2;
    Basis() : v1{0, 1}, v2{1, 0} {}
};
Basis b; // legal!
```

In C++20,

```
struct Vec1 {
    int x = 0, y = 0;
    Vec() {}
};

Vec1 v1; // v1.x = 0, v1.y = 0

struct Vec2 {
    int x = 0, y = 0;
    Vec(int x): x{x} {}
};

Vec v2{2}; // x = 2, y = 0
```

Careful! 8.1

The MIL takes precedence, only uses `= ?` if data field is not listed in MIL!

```
struct T {
    ReallyBig rb;
    ...
    T(int i) : rb{i} {}
    T() : {rb = init(i);}
};

struct Student {
    int assns, mt, final;
    string name;
    Student(string n) : assns{0}, mt{0}, final{0} { name = n; }
}; // name is "" by the end of MIL, then copies n
```

Remark.

> Using the MIL can provide some efficiency (we'll see better use cases later)
> Preference is to use it as much as possible.

8.2 Copy Constructor

```
struct Basis {
    ...
    Basis(const Vec& v1, const Vec& v2) : v1{v1.x, v1.y} v2{v2.x, v2.y} {}
};
```

Question: 8.2

But what if I don't want to manually extract every field from my parameter object?

Use an alternate constructor form, the "copy constructor"

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

For some class "c", the pattern for the copy constructor.

```
// header file
struct c {
    ...
    c(const c& otherc); // copy ctor
    ...
};

// implementation file
c::c(const c& otherc) ...
```

Fact 8.1

Compiler provides 5 ("The big 5") operations for you (unless you write your own):

- Default constructor (ctor) (lost as soon as you write any constructor). It is called the default constructor for that class for all object.datafields
- Destructor (dtor)
- Copy Constructor
- Move Constructor
- Copy Assignment
- Move Assignment

```
Student s1{60, 70, 80};
Student s2 = s1; // without elision, this is a copy ctor call!
Student s3{s1}; // also copy ctor
```

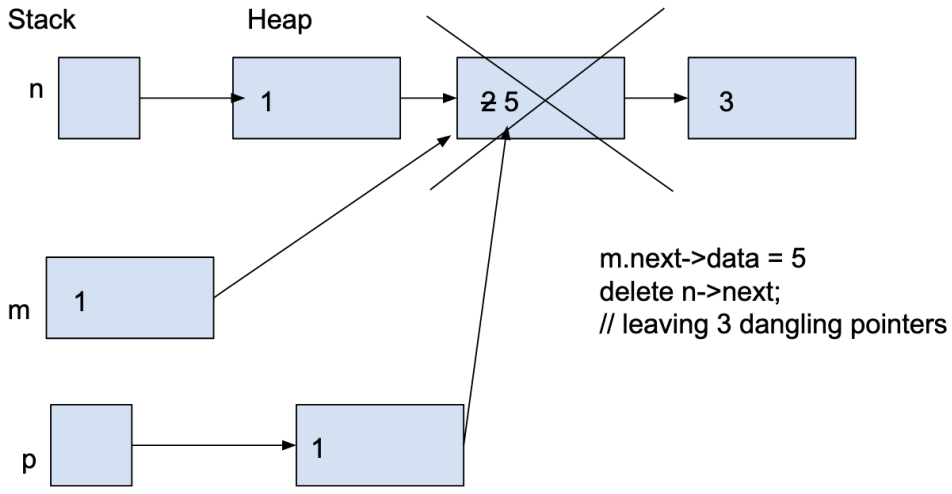
Fact 8.2

If all of your data fields are "primitive" types (int, char, float ...) then the compiler-provided versions are "good enough".

```
struct Student {
    int a, m, f;
    string n;
    ...
    Student(const Student& o): a{o.a}, m{o.m}, f{o.f}, n{o.n} {}
    // o.n is a string copy constructor! (compiler provided)
}; //compiler-provided version does exactly that, and can be omitted
```

Consider the following:

```
truct Node {
    int data;
    Node *next;
    Node (int d, Node *next = nullptr;
};

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



Fact 8.3

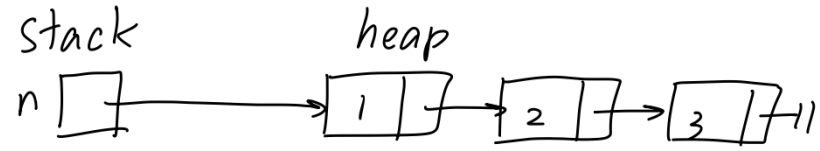
The default behaviour of compiler-provided copy constructor is to make "shallow" (rather than "deep") copies since it is copying the memory address (just like an int).

Before we fix copy constructor, lets take a quick look at destructors

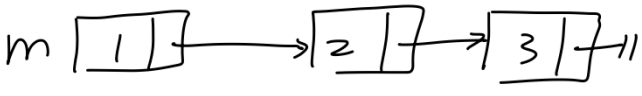
```
1 struct Node {
2     ...
3     ~ Node() { delete next; }
4 }
```

- Destructors have no return type, never take parameters
- It **cannot** be overloaded

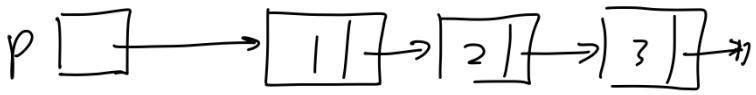
```
1 Node::Node(const Node &n) :
2     data{n.data},
3     next{n.next == nullptr ? nullptr :
4         new Node{*n.next}} // cpy ctor call, does deep cpy
5     {} // recurses till the whole list is copied over
```



Node m = *n;



Node *p = new Node{*n};



delete n; delete p; // no leaks bcuz of dtor

Question: 8.3

What happens if Node has a constructor with a single parameter, an int?

```
1 struct Node {
2     ...
3     Node(int i);
4     ...
5 };
6
7 void foo(const Node& n);
8
9 foo(4); // silent, implicit type comparison from int to Node;
```

9 Lecture 9

9.1 Copy Constructor (cont')

Question: 9.1

What are the 3 ways that a copy constructor is invoked? (Assuming no elision)

- **Part of an object declaration**
  `Student s { ... }; Student s1 = s; Student s2{s};` They both takes on lvalue
- **Pass an object by value**
  `void f(Student s) { ... }`
  `...`
  `f(s1);` It copies 's1' into the parameter 's' through a copy constructor
  It can be seen if compile with `-std=c++14` and `-fno-elide-constructors`
- **Return by value**
  `Student g() {`
  `Student s {30, 40, 50}; return s;`
  `}`
  `...`
  `cout << g() << endl;` Copy constructor on s to runtime stack, so can output value
  `Student joe = g();` copy constructor at least until we see "move semantics"

Question: 9.2

Why is this copy constructor implementation wrong?

```
1  // Wrong Implementation:
2  struct Node {
3      ...
4      Node(Node n) { ... } // "Node n" is a copy constructor call, and Node(...) is a
       ↪  copy constructor definition. It causes infinite recursion.
5      ...
6  };
7
8  // The fix:
9  struct Node {
10     ...
11     Node(const Node& n) { ... }
12     ...
13 };
```

9.2 Destructors Addendum

Fact 9.1

Steps of object destruction (without inheritance)

1. Run destructor body: Close resources, free heap memory etc.
2. Destroy data fields in reverse order of creation, runs destructors on any objects
3. Space for object is freed

Careful! 9.1: Use of Exit

C `exit()` function doesn't know about OOP. On execution, **no objects destructors** will be run! If the object was supposed to be free heap memory in destructor, it will not and will cause a memory leak!

9.3 Single Parameter Constructors

```
1  struct Node {
2      ...
3      Node(int i) : data{i}, next{nullptr} {}
4      ...
5  };
6
7  void f(const Node& n) { ... }
8  void f(Node n) { ... } // changed to "g"
9
10 f(4); // call to f ambiguous
11 g(4); // if we change one of the functions to g, it will work
```

Careful! 9.2

Silent conversion is dangerous. You don't realize where the mistake is coming. We can force the client to explicitly make the choice by making the constructor "explicit" keyword.

```
struct Node {
...
explicit Node(int i) ...
...
}
f(Node{4});  // legal! 4 is an rvalue, ctor call
```

9.4 Copy Assignment operator

By definition, the left hand side object already exists.

```
1  Node n1{1, new Node{2, new Node{3, nullptr}}};
2  Node n2{4, new Node{5, nullptr}};
3  ...
4  n2 = n1; // copy assignment
5
6  // this == &n2
7  Node& Node::operator=(const Node& rhs) {
8      data = rhs.data; // "this" not needed here
9      delete next; // calls dtor to delete all the nodes after
10     next = (rhs.next == nullptr ? nullptr : new Node{*(rhs.next)} /* copy ctor call
       ↪  */);
11     return *this;
12 }
```

Question: 9.3

What happens if assignment was `n1 = n1;`? (Self Assignment)

Problem is that freed "next" then tried to deep copy [error!!] (could lead to segment fault)
Not too likely to write but far more possible if using `Node *p, *q; *p = *q;`. Therefore, we must provide a self-assignment check.
The compiler doesn't provide `==` or `!=` for free.

```
1  Node& Node::operator=(const Node& rhs) {
2      if (this == &rhs) return this; // self assignment check
3      /*
4      same as before
5      */
6  }
```

Question: 9.4

What happens if new failed (heap ran out of memory)?

Truth: exception is raised
But for now: `operator=` returns immediately the value of "next" is **not** changed but is a dangling pointer!

Fact 9.2

Best option: do a deep copy (after self-assignment check) but before everything else. Worst case, copy fails and lhs is untouched.

```
Node& Node::operator=(const Node& rhs) {
    if (this == &rhs) return *this;
    Node *tmp = (rhs.next == nullptr? nullptr : new Node{*(rhs.next)};
    delete next;
    data = rhs.data;
    next = tmp;
    return *this;
}
```

9.5 Copy and Swap Idiom

Definition 9.3: Idiom

Language-level solution to a common problem

```
import <utility>; // std::swap
Node& Node::operator=(const Node& rhs) {
    Node tmp{rhs}; // copy ctor to stack => deep copy!
    std::swap{data, tmp.data};
    std::swap(next, tmp.next);
    return *this; // tmp goes out of scope, dtor gets ran
}

// If we change it to
Node& Node::operator=(Node tmp) { ... }
// We can get rid of the Node tmp{rhs} line (deep copies it)
```

10 Lecture 10

10.1 Move Semantics

Remember that an lvalue either has a name, a reference or a pointer to it. An rvalue is anything not an lvalue (e.g. anonymous object, something returned by value)
Consider `10-rvalue/node.cc` (A node class with a copy constructor)

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

```
    char c;
    cin >> c;
    if (r == '0') return evens;
    else return odds; // compiler has to copy whichever list is returned onto the
    ↪  runtime stack
}
...
Node n = oddsOrEvents();
// should be a copy constructor to be copy return value into n (as of c++2011,
↪  elision is required)
```

Goal: Recognize when we have an rvalue

- "steal" the contents. (only works for dynamic memory)
- Parameter type for "move" is an "rvalue reference" is `(Node &&)`

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

- Without elision: 1 move constructor to "move" local list onto run-time stack plus 1 move constructor for n to "steal" rvalue's content (With elision in c++20, only get the first move constructor call)

Question: 10.1

What happens with "move assignment"?

```
1 Node n{1, nullptr};
2 ...
3 n = oddsOrEvens();
```

If self-assignment (probably through `std::move`) is possible, then put in a check,

```
1 import <utility>; // std::swap
2
3 Node& Node::operator=(Node&& other) {
4     if (this == &other) return *this; // self assn check
5     std::swap(data, other.data);
6     std::swap(next, other.next);
7     return *this;
8 }
```

Summary:

- If have only defined copy operators, they get used for both copy and move
- If defined both copy and move operations:
  - Copy for lvalues
  - Move of rvalues

Question: 10.2

When do we implement the "Big 5"?

If write 1 of them consider whether or not should write all 5, need to answer first question of whether or not we own the "resource". If we do, and can't share (need deep copies, need all 5)

Fact 10.1

Elision

- As of C++11, the compiler is required to "elide" move and/or copy constructor calls in favour of byte-wise copy, even if the control flow is different
- You aren't required to know the rules for when it does (or does not) occur, you just have to know that it could (and probably will) occur

10.2 Member Operators

- There are 5 operators that must be methods of the class
  - `operator= // assignment`
  - `operator[] // index`
  - `operator() // function call)`
  - `operator-> // dereference and call access`
  - `T operator // type conversion`

Question: 10.3

When can it be part of a class?

When first operand of operator is an instance of the class

```
struct Vec {
    int x, y;
    Vec operator+(Vec rhs);
    Vec operator*(int i); // v * i
    ...
};

Vec operator+(int i, Vec v); // i * v
```

Note: if define +, should also define += etc.

```
struct Vec {
    ...
    std::istream& operator>>(std::istream& in); // do not make methods of the
    ↪ class!!!!
    std::ostream& operator<<(std::ostream& out);
}
...
Vec v{2, 3};
v >> cin; // syntatically legal but counter intuitive!
v << cout;

int i;
(v >> cin) >> i; // gets ugly do not do this
v >> (cin >> i);
```

11 Lecture 11

11.1 Object Arrays

Question: 11.1

If we have the following, is `Vec arr[10]` legal?

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

Class must have a default constructor (same for STL `std:vector`). Options:

1. Add a default constructor (may not be reasonable)
2. Have a stack-allocated array and call constructor on each object.
   `Vec arr[3] = { {0, 0}, {1, 1}, {2, 2}};`
   Doesn't scale up, not a feasible solution
3. Array of (`Vec *`) and call constructor when (heap) allocate each
   ```
   Vec **ap;
   ap = new Vec*[10];
   for (int i = 0; i < 10; ++i) ap = new Veci, i
   ...
   for (int i = 0; i < 10; ++i) delete ap[i];
   delete [] ap;;
   ```
   This is the most common solution!
4. More approaches but outside of the syllabus

11.2 Constant Objects

Consider the student class from before (`lectures/13-const/studentBad.cc`)

Question: 11.2

In the following code, there is a compilation error since `student::grade` doesn't promise it won't change "this". How do we fix it?

```
ostream& operator<<(ostream& out, const Student& s) {
    out << '[' << s.assns << ", " << s.mt << ", " << s.final << " = " << s.grade();
    return out;
}
```

The fix: We must declare the method as `const`!

```
// student.h
...
struct Student {
    int assns, mt, final;
    float grade() const;
}

// student.cc
...
float Student::grade() const {
    return ...
}
```

Remark.

> Presence / Absence allows method overloading so "const" must be on both declaration and definition.

Fact 11.1

We can only call `const` methods on `const` objects. If object isn't `const`, can either call `const` or non-`const` methods.
Best Practice: always declare methods `const` if it is not changing anything, compiler will help you catch errors.

Consider a case where we modify `Student` to add a counter `numCalls` (for system profiling)

```
struct Student {
    ...
    unsigned int numCalls = 0;
    ...
    float grade() const {
        ++numCalls; // trying to change data even though its const!
        return ...
    }
};
```

Need to differentiate between:

- Physical `const`ness (change any bit and the object is considered change)
- Logical `const`ness (only considered what "makes up" the logical concept to have changed) e.g. change in assignment grade, final grade etc. (`numCalls` isn't part of that!)

The fix: We can declare it `mutable`

```
struct Student {
    ...
    mutable unsigned int numCalls = 0;
    ...
    float grade() const {
        ++numCalls; // legal!
        return ...
    }
};
```

11.3 Comparing Objects

Motivation:
String comparison, char by char lexicographical comparison

- **In C:**
  ```
  const char *s1 = "...", *s2 = "...";
  int result = strcmp(s1, s2); // 1 comparison
  if(result == 0) {...}
  if(result < 0) {...}
  ```
- **In C++:**
  ```
  std::string s1 = "...", s2 = "...";
  if (s1 == s2) {...}
  if (s1 < s2) {...} // That will be 2 sets of comparison!
  ```

- More efficient approach introduced in C++20 is the **spaceship** operator, `<=>`

```
import <compare>; // std::strong_ordering::{less, greater, equal, equivalent}

std::strong_ordering result = s1 <=> s2;
if (result < 0) {
    ...
} else if (result == 0) {
    ...
} else {
    ...
}
```

Fact 11.2

To save time, we can:

- `auto result = s1 <=> s2 // compiler chooses the type`
- `using std::strong_ordering;`
  `if (result == greater) ...`
  `instead of std::strong_ordering::greater`

Remark.

> Defining `operator<=>` gives you all 6 relational ops (`<, <=, >, >=, == !=`) since:
> `s1 <=> s2 < 0 // has to compare result to 0 (s1 < s2)`

```
1  struct Vec {
2      int x, y;
3      Vec(int x, int y);
4      auto operator<=>(const Vec& v) const;
5      ...
6  };
7
8  auto Vec::operator<=>(const Vec& v) const {
9      auto result = x <=> v.x;
10     return (result == 0 ? y <=> v.y : result);
11 } // actually the compiler-provided version of the spaceship op by default. Field by
   ↪  field, lexicographical comparison
```

We can tell the compiler to provide it by

```
1  struct Vec {
2      ...
3      auto operator<=>(const Vec &) const = default;
4      ...
5  };
```

Question: 11.3

When does the default `<=>` not make sense?

```
struct Node {
    int data;
    Node *next; // comparing pointers are not useful
    ...
};

auto Node::operator<=>(const Node& n) const {
    auto result = data <=> n.data;
    if (result != 0) return result;
    if (!next && !n.next) return result; // == 0
    if (!next) return std::strong_ordering::less;
    if (!n.next) return std::string_ordering::greater;
    return *next <=> *n.next;
}
```

11.4 Invariants and encapsulation

Definition 11.3: Invariant

Statements that must be true at all points of an object's lifetime.

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

We have an implicit invariant in Node that next is either a `nullptr` or a valid heap address. It can be fixed by encapsulating data (and helper methods)

- i.e. a "capsule" being an opaque container
- In code, only access "public" info

```
struct Node {
   private:
       int data;
       Node *next;
   public:
       ~Node();
       Node(...);
       ...
};

Node n;
n.data = 1; // compiler error (since it's private)
```

Default accessibility/visibility with `struct` is public. Use `class` instead since it defaults to private.

12 Lecture 12

12.1 Encapsulation (cont')

```
class Vec {
    int x, y; // defaults to private accessibility
  public:
    Vec (int x, int y);
    Vec operator*(const Vec& o) const;
    int getX() const; // accessors (gettors)
    int getY() const;
    void setX(int nx); // mutators (setters)
    void setY(int ny);
};
```

Consider the following `/lectures/16-encapsulation/list.cc`

```
// list.cc
...
export class List {
    class Node; // forward declaration
    Node* theList = nullptr;
  public:
    ~List();
    void addToFront(int i);
    int ith(int i) const;
};
```

Fact 12.1

If we forward declare a class or struct, we can define it as either a struct or class.

```
// list-impl.cc
List::~List() { delete theList; }
struct List::Node { // we chose struct for simplicity
    int data;
    List::Node *next;
    ~Node();
};
List::Node::~Node() { delete next; } // class -> class -> func
void List::addToFront(int i) {
    theList = new Node{i, theList};
}
int& List::ith(int i) const {
```

```
    Node *ptr = theList;
    for (int ctr = 0; ctr < i: ++ctr, ptr = ptr->next); // assumes it does not
    ↪ dereference nullptr
    return ptr->data;
}
```

Encapsulation maintains class invariant, pointer either being nullptr or heap address. However, traversal of N-element list is now $O(n^2)$.

12.2 Iterators

Definition 12.2: Design Patterns

Common solutions to common problems but the focus is on classes and their relationships.

- Allows for code reuse, perhaps with slight tweaks

Iterator (design pattern):

- Given some sort of ADT / structure, want a pointer abstraction that enables "traversal"

```
class List {
    class Node;
    Node* theList = nullptr;
  public:
    class Iterator { // iterator NEVER has destructor
        Node *p;
      public:
        explicit Iterator(Node *p) : p {p} {}
        int& operator*() const { return p->data; }
        Iterator& operator++() {
            p = p->next;
            return *this;
        }
        bool operator==(const Iterator& o) const { // operator != can be generated
        ↪ by compiler
            eturn p == o.p;
        }
    };
    ...
    Iterator begin() const { return Iterator{theList}; }
    Iterator end() const { return Iterator{nullptr}; }
};
```

To traverse through the list with an iterator, consider the following code

```
List myList;
...
for (Iterator it = myList.begin(); it != myList.end(); ++it) { // O(n)
    std::cout << *it << std::endl;
}
```

Fact 12.3

So long as any class provides the iterator operators, it can be used as an iterator. As long as the ADT class has `begin()` and `end()` methods that returns an iterator, that code will work!

- C++ provides a **range-based** for loop as a shortcut since this is so common.

```
List myList;
...
for (auto i : myList) cout << i << endl; // range based for loop
// i is a dereferenced iterator, copied by value
...
for (auto& i : myList) i += 1; // i is now a reference, can be mutated
```

12.3 Friend

Question: 12.1

Anybody can call the Iterator constructor. How do we make the constructor private?

```
class List {
        ...
    public:
    class Iterator {
        Node *p;
        explicit Iterator(Node *p) : p{p} {}
      public:
        ...
        friend class List; // this can go anywhere in the itereator class
    };
};
```

A "friend" can access everything. We can narrow friendship down to functions / methods. A nested class can access static methods / data in outer class. Given an object/pointer/reference to the outer class, it can access even private data / methods.

13 Lecture 13

13.1 Encapsulation and Friendship (cont')

Question: 13.1

What do we do if a class doesn't provide accessors and mutators? For example `Vec` only used in arithmetic, but still need I/O functions, which can't be methods of `Vec`.

Declare I/O functions to be `friend`s in `Vec` (anywhere in the class)

```
1  class Vec {
2      friend std::ostream& operator<<(std::ostream&, const Vec&);
3      friend std::istream& operator>>(std::istream&, const Vec&);
4      int x, y;
5      public:
6      ...
7  };
8  std::ostream& operator<<(std::ostream&, const Vec&);
9
10 std::ostream& operator<<(std::ostream& out, const Vec& v) {
11     out << '(' << v.x << ", " << v.y << ')';
12     return out;
13 }
```

Question: 13.2

How to add `List::size()` so can safely use `List::ith()`?

One possible is to count the number of nodes for every call to size but that is $O(n)$! A better approach would be to add a counter.

- `List::add()` increments the counter
- Constructor initializes it to 0
- `size()` is now O(1) [Small space and time inc; latter amortized]

In general, cost of `<=>` on two `List` objects. When checking for equality, worse case is $O(n)$ on length of shorter list. However, for `operator=` we can immediatley judge not equal if the 2 lengths of differ $O(1)$!

Fact 13.1

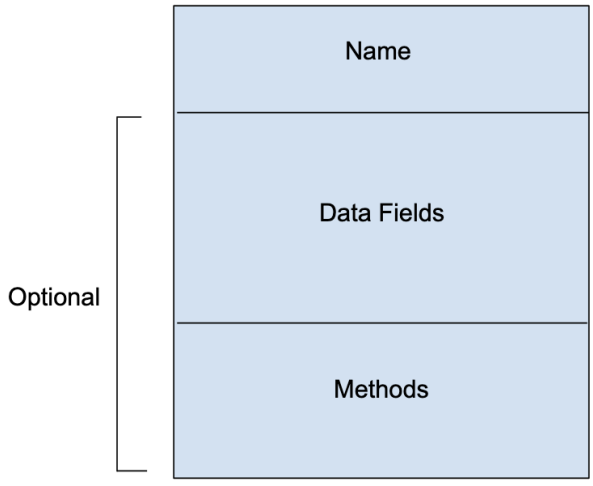
If you write `operator==` the compier will generate `operator!=` but the reverse **is not** true.

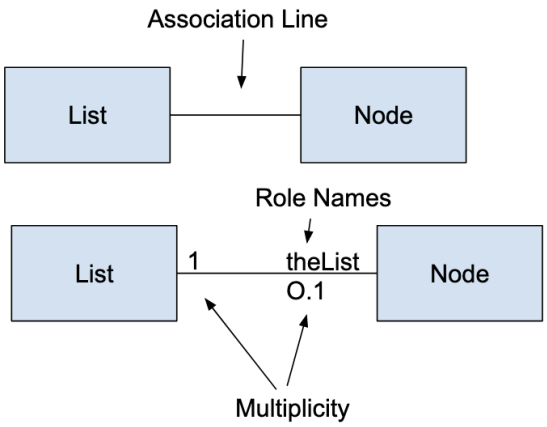
```cpp
bool List::operator==(const List& rhs) const {
    if (length != rhs.length) return false;
    return (*this <=> rhs) == 0;
}

auto List::operator<=>(const List& rhs) const {
    if (!theList && !rhs.theList) return std::strong_ordering::equal;
    if (!theList) return std::strong_ordering::less;
    if (!rhs.theList) return std::strong_ordering::greater;
    return (*theList <=> *rhs.theList);
}
...
List m1, m2;
...
if (m1 <=> m2 < 0) ... // invokes it
List *p = new List;
...
if (*p <=> m2 < 0) ... // needs to dereference if its a pointer
```

13.2 System Modeling

Definition 13.2: System Modeling

A visual representation of the abstractions and their relationships. The Unified Modeling Language (UML) is a common standard. (We will be using "class models", somewhat simplified to represent our classes.) Model is supposed to be "language agnostic" (doesn't reference any particular implementation language)

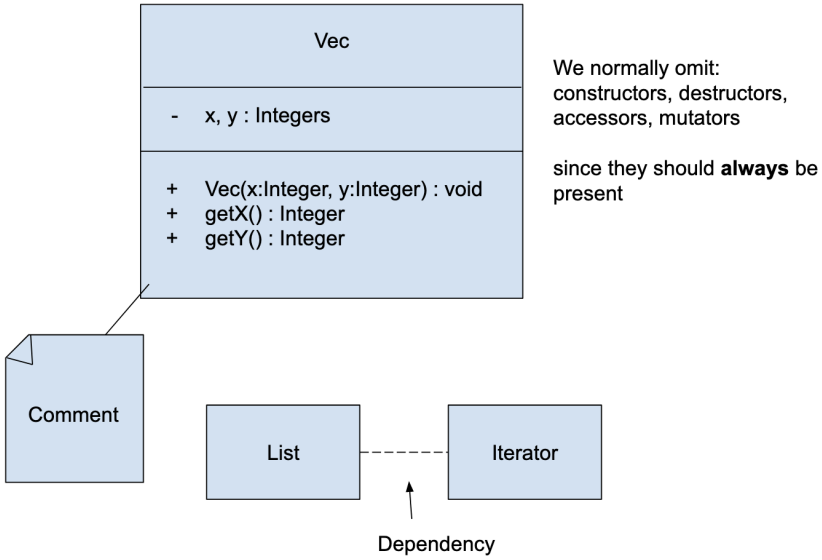




Definition 13.3: UML Symbols

+ means public

- means private

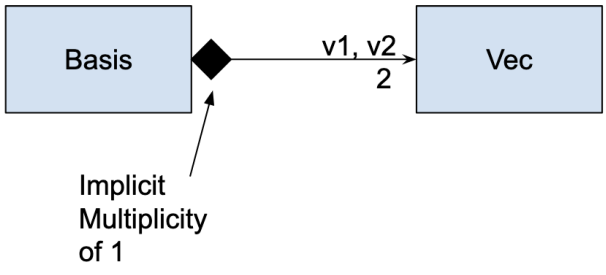


Definition 13.4: Composition Relationship

It is also called "owns-a". For example, a `Basis` object is "composed" (built out of) 2 `Vec` objects

Properties:

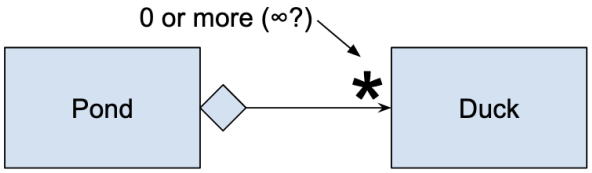
- If A is composed of B
- If A is destroyed, B is destroyed
- No sharing
- Deep copying required



Definition 13.5: Aggregate Relationship

It is called "has a" relationship.

- If A "has a" B
- Sharing is possible
- A and B exists independently of each other
- If A is destroyed, B is not
- It creates a shallow copy
- Implemented with Pointers (maybe references)



It can be a dynamically allocated array, `std::vector` but memory is finite

```
1 class Pond {
2     Duck *duck[MAX_NUM_DUCKS];
3     ...
4 }
```

Question: 13.3

To think about: Does the use of a pointer in implementation always implfy aggregation?

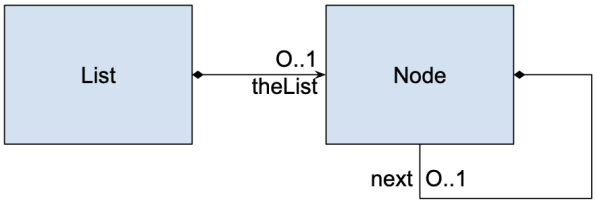
14 Lecture 14

14.1 System Modeling (Cont')

Question: 14.1

Does the presence of a pointer (in a class) always imply non-ownership of what is being pointed to?

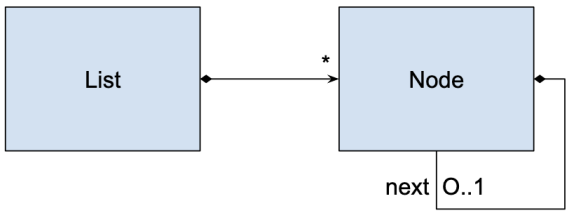
Not ownership if destructor does not free it. (insufficient information). For example,



Ownership since destructor free pointers

Question: 14.2

What is implied if we change the UML class model to the following

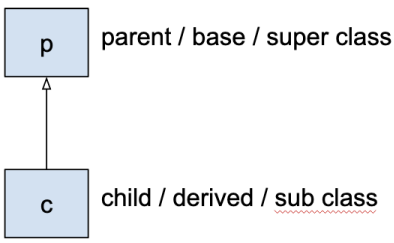


Implies `List` loops to free each `Node` rather than recursive `Node` deletion.

14.2 Inheritance

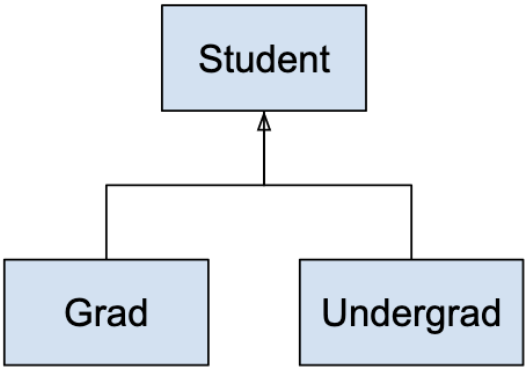
Definition 14.1: Inheritance

`c` "is-a" `p` (No multiplicities one either end of the association line)

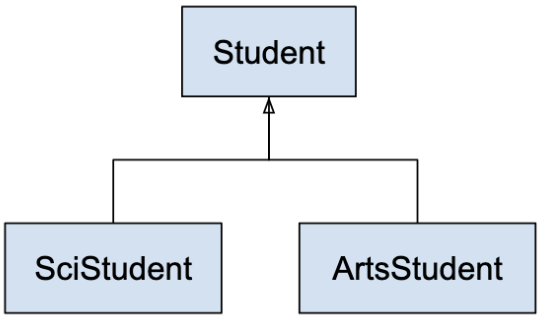


Alternate Names:

1. Specialization (e.g. start with `Student`)



2. Generalization (e.g. start with `SciStudent, ArtsStudent`)



Recognizes common data / methods and promote to parent i.e. inherit commonalities

Motivating Example:
- Collection of Book, Comics and Text objects. Consider the following UML,

```
//         Book         |         Text         |         Comic        |
// ====================| ====================| ====================|
// - author : String    | - author : String    | - author : String    |
// - title : String     | - title : String     | - title : String     |
// - numPages : Integer| - numPages : Integer| - numPages : Integer|
//                      | - topic : String     | - hero : String      |
// ====================|=====================|=====================|
```

Question: 14.3

How to store these objects?

- **"tagged union"**

  `union BookTypes { Book *b; Text* t; Comic *c; };`

`Book Types books [20];`

This needs a paarallell array to remember which fields was used in `books`

- **Void pointers**

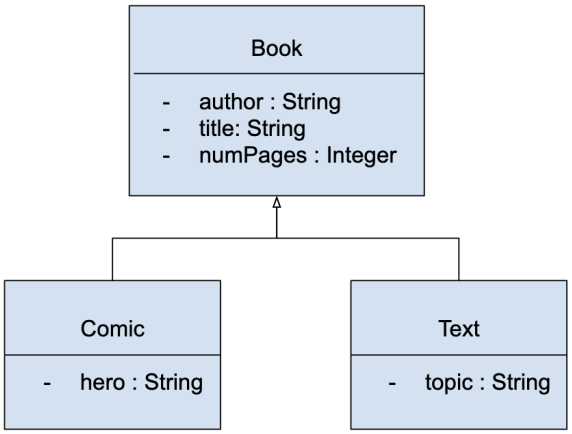
  `void * books [20];`

  Not type safe!!!, still need a parallel array to remember what was stored and context

- **Recognize "is-a" relationship**

  `Text` is a `Book`, `Comic` is a `Book`

  We can generalize this by promoting `Book` to be the parent class.




```
1  class Book {
2      ...
3    public:
4      Book ( ... ); // don't want default ctor
5  };
6
7  class Text:public Book {
8      // don't repeat parent's data fields
9      // since "shadows" / hid ethem
10     std::string topic;
11   public:
12     Text ( ... );
13 };
14
15 class Comic:public Book {
16     std::string hero;
17   public:
18     Comic ( ... );
19 };
```

Consider the following `Text` constructor implementation

```
1 Text::Text(const string &a, const string &t, int p, const string & top) :
2     author{a}, title{t}, numPages{p}, topic{top} {}
```

This implementation is wrong as the child class `Text` does not have access to the parent class's, `Book`, information as those fields are private. Neither the "outsiders" nor the "children" can access those fields.
- Other problem: `Book` does not have a default constructor and also can't call the constructor in `Text` constructor since it is too late.
- Object creation steps are now slightly different due to inheritance.

1. Allocate space for entire Object
2. Call parent's default constructor
3. Call the default constructor for any object data fields (Execute MIL)
4. Run the constructor body

MIL can be used to call parent's constructor. Consider the following changes to the `Text` constructor implementation

```
1 Text::Text(const string& a, const string& t, int p, const string &top) :
2     Book{a, t, p}, topic{top} {}
```

Definition 14.2: Protected keyword

Can weaken encapsulation to allow children but nobody else to access data fields.

```
1  class Book {
2    protected:
3      string author, title;
4      int numPages;
5    public:
6      Book ( ... );
7      ...
8  };
9  class Text:public Book {
10     ...
11   public:
12     void addAuthor(const string& a) {
13         author += a; // append the string in Book (legal!)
14     }
15 };
```

A better alternative is to leave data fields private and add protected `addAuthor` to `Book` that `Text` can call. This maintains invariants.
Consider `isHeavy()` to class hierachy

- `Book` returns true if the number of pages is>200
- `Text` returns true if the number of pages is >500
- `Comic` returns true if the number of pages is >30

```
Book b{"Alex D", "The Court of Monte Cristo", 199};
Comic c{"Tom King", "A comic book", 35, "Superman"};
b.isHeavy(); // false
c.isHeavy(); // true
...
b = c; // Object slicing, only the inherited Book portion of c is copied onto B
// => still treated as a Book::isHeavy() call

Book *bptr = &b;
bptr = &c; // legal! no object-slicing
bptr->isHeavy(); // invokes Book::isHeavy() not Comic::isHeavy()
// => set statically at compile time!

Comic* cptr = &c;
cptr->isHeavy(); // Comic::isHeavy()
Book& bref = c;
bref.isHeavy(); // statically set to Book::isHeavy()
```

To make this work, we will need to deeclare `Book`'s `isHeavy()` to be virtual and overriding it in the children. Virtualness is inherited, so virtual in child even if virtual keyword is not repeated

```
class Book {
    ...
  public:
    ...
    virtual bool isHeavy() const;
    ...
};
class Comic:public Book {
    ...
  public:
    ...
    virtual bool isHeavy() const override;
    ...
};
```

The `virtual` and `override` keyword is optional. The compiler checks that parent has a `virtual` methodwith same signature excluding return type. It is best practice!

```
1 Book *bptr = &c;
2 bptr->isHeavy() // invokes Comic::isHeavy()
3 bptr = &b;
4 bptr->isHeavy() // invokes Book::isHeavy()
```

The change allows the compiler to check the object type at runtime and call the appropriate `isHeavy()` method.

Question: 14.4

What happens if we have the following:

```
1 Book* bptr;
2 ...
3 bptr = new Comic { ... };
4 ...
5 delete bptr;
```

This code will call the `Book` destructor instead of `Comic` one, causing a memory leak.
To fix this, we can set the destructor of `Book` as `virutal`.

Careful! 14.1

If there is an inhertance hierachy, always set the destructor as `virtual`.

15 Lecture 15

15.1 Polymorphism

Definition 15.1: Inheritance (cont')

Polymorphism is Greek for "Many Forms"

- The mechanism of which of runtime, the correct class method is determine and called
- Implementation, requires a reference/pointer of the parent class type that can be set to an instance of either the parent class or one of its children classes
- Parent has at least 1 `virtual` method that the child can choose to "override" (or not)

Fact 15.2

If we want to ensure that we cannot inherit from a class (enforced by compiler) by declaring it `final`

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

15.2 Abstract Base Classes (ABC)

To learn more, `https://isocpp.org/wiki/faq/abcs`

- Some base classes shouldn't be instantiated (abstract)
- Some methods may not have an implementation in the abstract e.g. `Student::calcFees()` requires know if is a regular or co-op student

```
1 class Student {
2   protected:
3     unsigned int numCourses;
4   public:
5     ...
6     virtual int calcFees() const = 0; // = 0 makes it a "pure" virtual method, i.e.
      ↪  cant create a Student
7     ...
8 }
```

Even though pure `virtual`, can still provide an Implementation! (has to go in the "implementation" file)

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

We can now implementation `RegularStudent::calcFees()` in Implementation file, and instantiate `RegularStudent` objects. `Student` is called an Abstract Class, `RegularStudent` is called a "concrete" class.

- If no good choice as to which method to make pure `virtual`, can always use the destructor **but must still implemented in the implementation file**
- UML notation: ABC name must be italicized. *Student* or `Student{abstract}`
- The UML standard only italicizes pure virtual methods but in this course, we'll italicize all forms of virtual methods. (If it matters, put `= 0` at end of a pure `virtual` method signature.)

15.3 Templates

Remember our `List` class from before

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

Question: 15.1

If we want a `List` that can store other types, need a new class would duplicate everything and just replace data types. Are there any alternatives?

We can use templates (Inheritance would not work the desired way)

```
template <typename T> class List { // typename can be replaced with class
    struct Node {
        T data;
        Node *next;
        ...
    };
    Node *theList = nullptr;
  public:
    ~List() { delete theList; }
    void addtoFront(T value) {
        theList = new Node{value, theList};
    }
    T& ith(int idx) const;
    class Iterator {
        friend class List;
        Iterator( ... ) { ... }
      public:
        T operator*() const;
        Iterator& operator++();
        bool operator!=(const Iterator &) const;
    };
};
```

Since we need the entire class definition and implementation at the point of instantiation, we can't seperate code into interface and implementation file. However, we can still implement in the interface/header file.

```
List<int> myList1;
List<List<int>> myList2;
List<string> myList3;

myList1.addToFront(100);
myList2.addToFront(myList1); // valid

for (auto el : myList1) {
    cout << el << endl;
}
```

STL provides a variety of containers. We are going to start with `std::vector`, which is dynamically resizable, but does all of the allocation/copying for you. Consider the following use cases of `std::vector`,

```cpp
import <vector>;
using std::vector;

vector<int> v1; // empty i.e. v1.size() => 0
vector<int> v2{4, 5}; // only contains: 4, 5
vector<int> v3(4, 5); // contains: 5, 5, 5, 5 (4 x 5)

if (v3[1] == 6) ... // [] has no bounds checking, must be 0 <= x < size() - 1
v1.push_back(1); // optimized to insert at end
v1.push_back(2);

vector<Student> vo;
vo.emplace_back(30, 40 ,50); // emplace_back for object ctor call

vector v4{4, 5, 6, 7}; // compiler deduces types from values

for (std::vector<int>::iterator it = v1.begin(); it != v1.end(); ++it) {
    cout << *it << endl;
}

for (auto elem : v3) cout << elem << endl;

for (std::vector<int>::reverse_iterator it = v3.rbegin();
     it != v3.rend(); ++it) { ... }

v3.pop_back(); // removes last item
v3.clear(); // empties entire vector, has no notion of ownership
            // so if its an arr of pointers, must free yourself
```

Question: 15.2

Remove all 5's from `v {1, 2, 5, 5, 6, 7, 5}`, how do we do that?

We can use erase in combination with iterator e.g. `v.erase(v.begin() + 3)`

```cpp
// 1st attempt
for (auto it = v.begin(); it != v.end(); ++it) {
    if (*it == 5) v.erase(it);
}
```

This shifts everything by 1, skipping the next element in sequence "5". So if we erase, we don't want to preincrement it since we might skip over next match. Iterator will no longer be valid after erase (might have made array smaller and points to old array)

```
for (auto it = v.begin(); it != v.end(); /* no ++ */) {
    it (*it == 5) it = v.erase(it);
    else ++it;
}
```

16 Lecture 16

16.1 Templates (cont')

Polymorphism ensures the correct behaviour at runtime

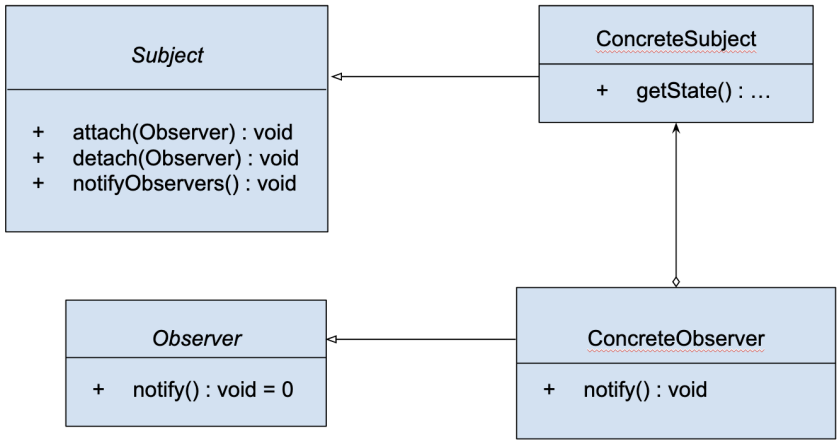
```
class AbstractIterator {
  public:
    virtual ~AbstractIterator();
    bool operator!=(const AbstractIterator &) const = 0;
    AbstractIterator* operator++() = 0;
    int operator*() const = 0;
};

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

We can now do the following

```
void forEach(AbstractIterator & st, const AbstractIterator & en, void (*f) (int)) {
    while (st != en) {
        f(*st);
        ++st;
    }
}
```

16.2 Observer Design Pattern

Consider the code from `lectures/24-observer`. It is also known as "Public Subscribe".



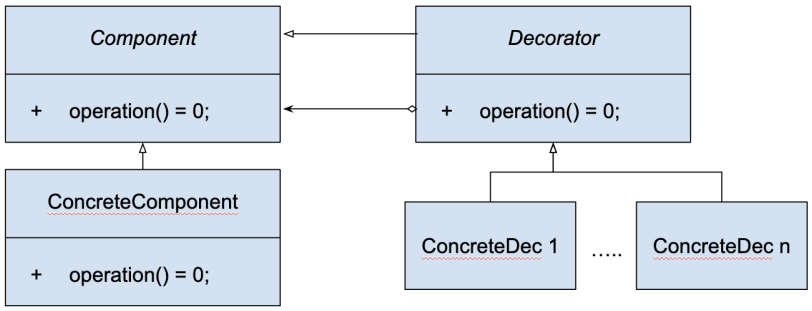

```
1  class Observer;
2
3  class Subject {
4      std::vector<Observer*> obs;
5    public:
6      virtual ~Subject() = 0;
7      void attach(Observer *);
8      void detach(Observer *);
9      void notifyObservers();
10 };
11
12 Subject::~Subject() {}
13 void Subject::attach(Observer *o) {
14     obs.push_back(o);
15 }
16 void Subject::detach(Observer *o) {
17     ... if (*it == 0) {
18         obs.erase(it);
19         break;
20     } ...
21 }
22 void Subject::notifyObservers() {
23     for (auto o : obs) o->notify();
24 }
```

```
class Observer {
  public:
    virtual void notify() = 0;
    virtual ~Observer() {}
};
...
class HorseRace:public Subject {
    ifstream in;
    std::string lastWinner;
  public:
    HorseRace(string fname) : in{fname} {}
    bool runRace() {
        in >> lastWinner;
        return in.good();
    }
    String winner() const {
        return lastWinner;
    }
}

class Bettor:public Observer {
    HorseRace *hr;
    string name, hname;
  public:
    Bettor(HorseRace *h, string n, string hn) : hr{h}, name{n}, hname{hn} {
        hr->attach(this);
    }
    ~Bettor() { hr->detach(this); }
    void notify() {
        if (hr->winner() == hname) {
            cout << "Yahoo" << endl;;
        } else {
            cout << "Boohoo << endl;
        }
    }
}

int main() {
    HorseRace hr { "races.txt };
    Bettor bob {&hr, "Bob", "Horse 1"};
    ... // create more bettors
    while (hr::runRace()) {
        hr.notifyObservers(); // general may be triggered from outside or by
        ↪ subject
    }
}
```

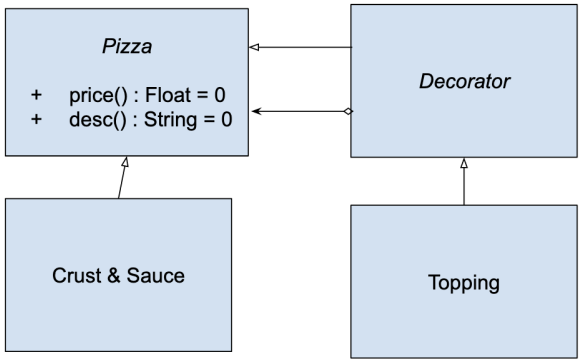
16.3 Decorator Design Pattern

Dynamically change appearance and/or capabilities

- Let me combine in any desired fashion without an exponential number of classes being needed



25-decorator




```
1  Pizza *p = new CrustAndSauce;
2  p = new Topping {"bacon", 1, p};
3  p = new Topping{"Cheese", 0.25 p};
4  ...
5  cout << p->desc << " " << p->price << endl;
6
7  class Pizza {
8    public:
9      virtual ~Pizza() {}
10     virtual string des() const = 0;
11     virtual float price() const = 0;
12 }
13
14 class CrustAndSauce:public Pizza {
15   public:
16     string desc() const { return "crust and sauce"; }
17     float price() const { return 8.50; }
18 };
```

```
19
20 class Decorator:public Pizza {
21   protected:
22     Pizza *next;
23   public:
24     Decorator(Pizza *p) : next {p} {}
25     virtual Decorator() { delete next; }
26 };
27
28 class Topping:public Decorator {
29     string topping;
30     float cost;
31   public:
32     Topping(Pizza *p, string t, float c) :
33         Decorator{p}, top{t}, cost{c} {}
34     string desc() const { return top + " " + next->desc(); }
35     float price() const { return cost + next->price(); }
36 };
```

17 Lecture 17

17.1 Modularization

Question: 17.1

What goes into a module?

So far we've had 1 class per module; but a module can contain multiple functions and/or classes (can also be split across multiple files.)

Question: 17.2

How do we choose a module's content

There are 2 main metrics for judging software design (coupling and cohesion are used to help decide upon the contents of your module (or class))

1. Coupling (How much do 2 distinct modules depend upon each other)
   Low → High:
   (a) Communicate by passing around primitive types
   (b) Pass around arrays and Structures
   (c) Modules Affect each other's control flow
   (d) Share global dataknows and has access to implementation details (e.g. "friends")

   If we have high / tight coupling,
   - Changes in 1 causes changes in other
   - Makes code reuse harder since we cannt just pick up and use something else without changes

2. Cohesion (How closely related are the eleements of a module (or class))
   Low → High:
   (a) Unrelated/arbitrary grouping of elements (e.g. `<utility>:std::swap, std::pair ...`)
   (b) Possibly still unrelated, but common theme or code base (e.g. `<algorithm>` has templated function that use iterators)
   (c) Elements manipulate state over a lifetime (e.g. openfile, read, close file)
   (d) Pass data to each other
   (e) Cooperate to perform a single task

Low Cohesion: Hard to understand and maintain (if reuse, get other unwanted elements as well)

Fact 17.1

We should aim to have high cohesion and low coupling.

Question: 17.3

What happens if you have 2 classes that depend upon each other? Consider the following code.

```
1 class A {
2     int x;
3     B y;
4 };
5
6 class B {
7     char x;
8     A y;
9 };
```

C++ does not allow this.

Question: 17.4

Does forward declaration fix this?

```
1 class B;
2 class A { int x; B y; };
3 class B { char x; A y; };
```

Still need change. We need to change either `A y` or `B y` (or both) into `A *y; B *y` to get this to work.
Cases where we must still declare classes in order

```
1 class C { ... }; // parent
2 class D:public C { ... }; // child
3 class E { C a; ... };
```

Remark.

Remember that `C++20` modules requires compilation independancy order (`C` before `D` or `E`.)

17.2 Decoupling Interfaces (MVC)

Consider a chess program:

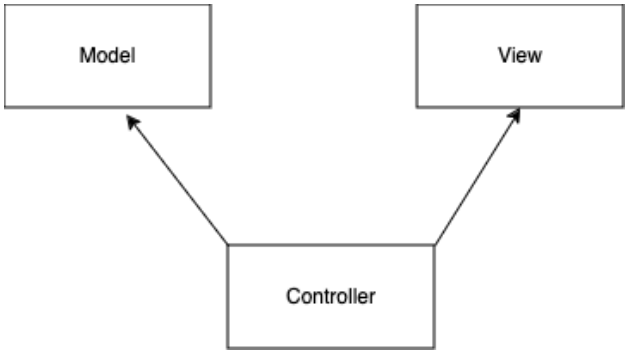
```
1 class ChessBoard { ... cout << "Your move" ... };
```

Have to change code if you want I/O to use files.

Question: 17.5

What if we want to add a graphical user interface (GUI)?

Problem is that the chess board class is both the data model and interacts with the user. That violates the "Single Responsibility Principle" (SRP) i.e. each class should only have one reason to change. This leads us to the architectual pattern, Model-View-Controller (MVC)



Model: Data Model
View: User interface/display (text, graphics, etc.)
Controller: Mediates between view and model (Grey area for rule, enforcementment-model? Controller?)
Can also add an observer design pattern on it by linking model and view.

17.3 Exception

Should be used in "exceptional" circumstances (can't easily locally reecover from an error). Though it has a performance penalty for deciding "who" handles this. `std::vector::at(i)` verifies `0` $\leq$ `i < size()` if outside of valid range, `at()` raises/throws an exception. Generated code looks for a handler for `std::out_of_range` (exception library). (C++ lets you throw anything as an exception). If you can't find a matching handler, program terminates with an unhandled exception error.

```
1 void h() { throw 1; }
2 void g() { ... h(); ... }
3 void f() { ... g(); ... }
4 int main() { ... f(); ... }
```

"Stack unwinding" is the process of finding a matching handler. If frame has no handler, immediately disqualified; else look through them. When frame is removed, all local object destructors are run.

```
1  void h() { throw 1; }
2  void g() {
3      try {
4          h(); cout << 1;
5      } catch ( ... ) {
6          cout << 2; throw 'x';
7      }
8  }
9  void f() { ... g(); ... }
10 int main() {
11     try {
12         f(); cout << 3;
13     } catch ( ... ) {
14         cout << 4; cout << 5;
15     }
16 } // outputs 2 4 5
```

18 Lecture 18

18.1 Exception (cont')

Fact 18.1

Modules cannot forward declare another module or elements of another module!

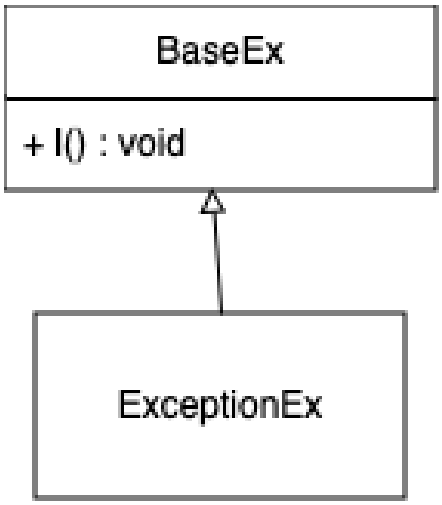
Question: 18.1

Why do we resolve exception if handling them is slow?

```
1  int& 2int::get(unsigned int idx) {
2      return v.at(idx); // raise std::out_of_range because at() uses exception
3  }
4  ...
5  int& set = mylist.get(1000); // after exception is raised no code is processed
   ↪  after
6
7  try {
8      int& set = mylist.get(1000);
9  } catch (std::out_of_range e) {
10     cerr << e.what();
11 }
```

- – `std::vector::at()` recognizes out of bounds index but doesn't know how to respond (e.g. ask another time for idx)
  – Client deals with error but doesn't detect it
  – Isolate error handling from "Regular exception code"
- We also cannot ignore exception because it stops the program.
  Alternative: C mechanism of checking error is return vals as a global `errno` val. (Passive, checking cannot be forced.)
- We can also throw whatever object you want for your exception. (Use object inheritance for exception object)
- Exception lib is an inheritance hierachy

Consider the following example inheritance + exception



1. Throw Base Ex, 2. Throw ExceptionEx

(Cause object slicing but legal)

```
try {
    ...
} catch (BaseEx l) { b.l(); }

try { // We should always catch by reference
    ...
} catch (BaseEx &b) { // will catch ExceptionEx because it is an inheritance of
↪ BaseEx
    ... b.l() ...
} catch (ExceptionEx &e) {
    .. e.l() ...
}
```

Fact 18.2

We should always catch in order of most to least specific

```
ExceptionEx e;
BaseEx &b = e;
throw b; // raises base exception, looks at type of b, doesn't consider what b is
↪ pointed to
```

Fact 18.3

We should always Raise by value

```
ExceptionEx e;
throw e;

try {
    ... // call that will throw e
} catch (BaseEx &b) {
    ... l.b(); ... // method l has to be virtual base exception or override for
    ↪  specific exceptions
    ... throw b; // b is type BaseEx& raises base exception
    throw; // checks for underlying object type throws ExceptionEx
}
```

Now consider destructors + exceptions

- Remember that "stack unwinding" guarantees that destructors for all local objects are executed as the function / method frame activation is removed from the runtime stack in the search for a matching handler

Question: 18.2

What happens if stack is being unwound to find handlers, + destructors trigger another exception

As long as it doesn't allow destructors to raise exceptions.
Way around destruction is to tag it with `noexcept`(false)

18.2 Smart Pointers

Consider the following code

```
void l() {
    C mc;
    C* pc = new C;
    g(); // raises exception, causes memory leak since pc not deleted
    delete pc;
}
```

A potential solution to this problem is the following,

```
void l() {
    C mc;
    C* pc = new C;
    try {
        g();
    } catch (...) {
        delete pc;
        throw; // reraise exception, prevent double free
    }
    delete pc;
}
```

2 Frees are not ideal, it could get complex quickly. Consider another solution with a helper.

- Some languages like Java have a "finally" clause (`finally`) code in block is always executed even if exception is raised.
- "Resource Acquisition is Initialization" idiom (RAII)
  - Applied to any owned resource (e.g heap memory)
  - `ifstream file{"in.txt"};` outside of scope, file closed, buffer cleared
  - `std::unique_ptr` from `<memory>` library stores any pointer type

```
void f() {
    C mc;
    std::unique_ptr<C> p{new C}; // this is an object itself
    g(); // if g raises exception, destructor for p is called no issues
}
```

Copy is disabled, we must use move. Other ways of creating unique pointers:

```
void l() {
    C mc;
    auto p = std::make_unique<C>(ctor args); // calls new for us
    g();
}
```

Consider the following code,

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

Careful! 18.1: Common Mistakes

This will cause an error, implicit copy

To fix this, we must pass underlying address without changing ownership

- Owner holds unique pointers
- Owner passes underlying raw heap address to/from methods and functions unless transferring ownership.

Consider `lectures/26-unique_ptr`

```
template<typename T> class unique_ptr {
    T* ptr;
    explicit unique_ptr(T* p) : ptr{p} {}
    ~unique_ptr() { delete ptr; } // not virtual, ho inheritance in stdlib
    unique_ptr(const unique_ptr &) = delete; // can't copy
    unique_ptr<T>& operator=(const unique_ptr &) = delete;
    unique_ptr(unique_ptr<T> && o) : ptr{o.ptr} { o.ptr = nullptr; }
};
```

19 Lecture 19

19.1 Smart Pointers (cont')

Fact 19.1

There is no copy for unique pointers, only move!!

```
unique_ptr<T>& operator=(unique_ptr<T> && other) {
    delete ptr;
    ptr = other.ptr;
    other.ptr = nullptr;
    return *this;
}

T* get() const { return ptr; } // extracts raw pointer

T& operator*() { return *ptr; } // dereference operator
```

To determine which type of pointers to use,

- Check ownership (i.e. do we allow copies?) If sole owner with no sharing, we should use `std::unique_ptr`. We can use `get()` to obtain raw pointer and pass that.

Passing unique pointers:

```
void f(std::unique_ptr<c> p); // f takes on ownership of p => caller loses
↪ ownership
void g(c* p); // no ownership transfer, caller may not even point to heap memory! g
↪ should NOT delete it
```

Returning unique pointers:

```
std::unique_ptr<c> f(); // move operation, transfers ownership from f to reciepient
C* g(); // no ownership transfer, caller may not be final owner (do not delete)
```

In certain rare circumstances, ownership is truly shared It is usually implemented by a "reference counter". The last one pointing to it frees it upon going out of scope.

```
int main() {
    auto p1 = std::make_shared<C>(); // ref cnt == 1

    if ( ... ) {
        auto p2 = p1; // ref cnt == 2
    } // ref cnt == 1 after it goes out of scope
} p1 goes out of scope so frees ((x))
```

Best practice is to choose the appropriate smart pointer type based upon ownership. It drastically reduces the number of memory problems.

19.2 Map

`STL map`

- `include / import <map>; std::map<k, v>`. This uses `std::pair` from `<utility>`
  `std::pair<S, T> => S first; T second;` // <= public data fields
- Used for dictionaries since it consists of a key value pair. Its keys **must be unique**, it either provides `operator<` or client defines a "comparator function" It is usually implentered
- Itereation is in key order

```
std;:map<std::string, int> m;
m["abc"] = 1;
m["def"] = 3;

cout << m["abc"] // outputs 1
cout << m["ghi"] // inserts previously non existent key and default

if (m.count("xyz") == 1) { // it will either return 0 or 1
    m.erase("def");
    // removes key value pair when key == "def"
}

for (auto& p : m) { // p is a std::pair
    cout << '(' << p.first << ", " << p.second << ')' << endl;;
}
```

C++20 also allows "structured binding'

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

Structured bindings can only be used in

1. Structs where all fields are public. (e.g. `Vec v1, 2; // Assume struct with only public fields`
   `auto [x, y] = v;`)
2. Stack-allocated Array of known size (e.g. `int a[] = {1, 2, 3}; auto [x, y, z] = a;`)

19.3 Inheritance and the Big 5

Assume that the `Book` class is defined as before, including hte big 5 i.e. all the move, copy constructors and assignment operators

```
class Book {
    std::string author, title;
    int numPages;
  public:
    // ctor, accessors, mutators, big 5
};

class Text: public Book {
    std::string topic;
  public:
    // ctor, accessors, mutators, no big 5
};

Text t1 { ... };
Text t2 = t1;
```

Compiler uses Book copy constructor to copy Book portion of t1 into t2. It will then proceed, field-by,field, copying (`std::string` has copy ops). We could declare `Text`'s copy constructor to be `default`.

```
class Text: public Book {
    ...
  public:
    Text(const Text &) = default;
    ...
};
```

If we wanted to instead implement it:

```
Text::Text(const Text *t) : Book{t}, topic{t.topic} {}
```

Question: 19.1

How about the move constructor?

We need to apply `std::move` to parameter since it is an lvalue (despite it being an rvalue reference)

```
Text::Text(Text && t) : Book{std::move(t)}, topic{std::move(t.topic)} {}
```

`Text && t` is an lvalue and `Book{std::move(t)}` invokes Book move constructor

```
1  Text& Text::operator=(const Text& other) {
2      if (this == &other) return *this;
3      Book::operator=(other); // copy inherited book information
4      topic = other.topic;
5      return *this;
6  }
7
8  Text& Text::operator=(Text && other) {
9      if (this == &other) return *this;
10     Book::operator=(std::move(other));
11     topic = std::move(other.topic);
12     return *this;
13 }
```

All of the code above is default behaviour.
Consider the following

```
1 Text t1{...}, t2{...};
2 Book *pb1 = &t1, *pb2 = &t2;
3 ...
4 *pb1 = *pb2; // t1 = t2?
```

Question: 19.2

Whose copy assignment operator gets invoked?

`Book`'s copy assignment operator gets invoked as it is not a virtual method. We statically hard-code at compile time the address of `Book`'s copy assignment. Therefore, the `Book` portion of `t2` is copied into `t1`. This is called **partial assignment**.

Question: 19.3

To think about: Can we fix this by making move and copy assignments `Virtual`?

Hint: no it makes things worse.

20 Lecture 20

20.1 Inheritance and the Big 5 (cont')

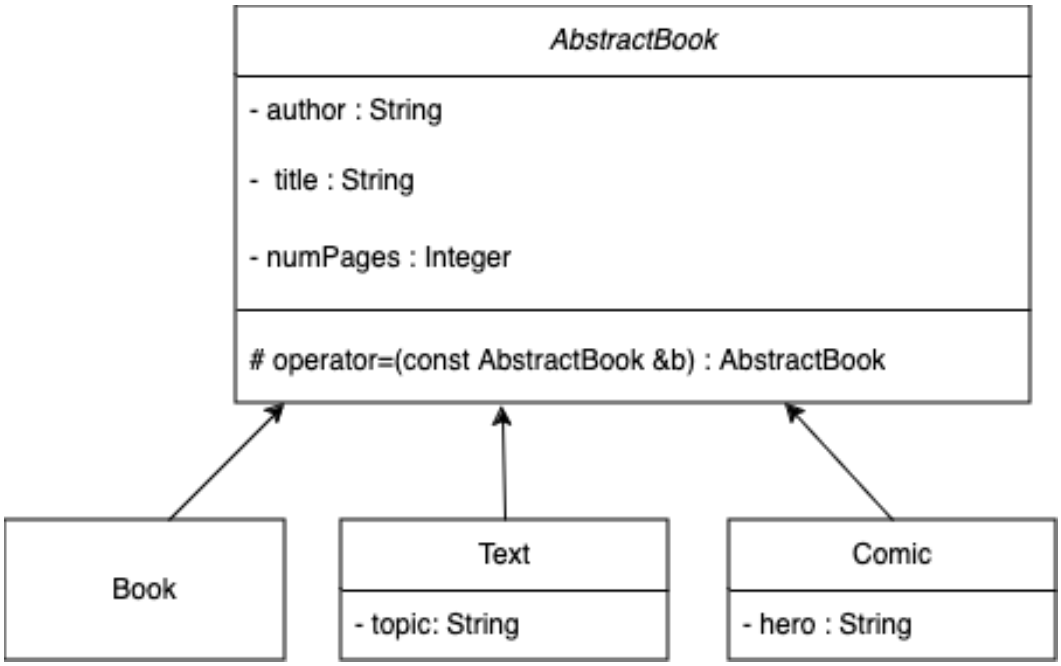
Question: 20.1

Would making copy or move assignment `virtual` fix this?

Consider the following code

```
1  class Book {
2      ...
3    public:
4      ...
5      virtual Book& operator=(const Book &);
6      ...
7  };
8
9  class Text: public Book {
10     ...
11   public:
12     ...
13     virtual text& operator=(const Book &) override; // const Text & is NOT an
       ↪  override
14 };
15
16 Text t { ... };
17 Comic c { ... };
18 Book b { ... }, *b1p = &t, *bp2 = &c;
19
20 t = b; // legal! but still partial assignment
21 *bp1 = b; // still ^^
22 *bp1 = *bp2; // legal but assigning siblings across the hierachy is called "mixed"
   ↪  assignment
```

Version 1 of fixing this prperly prevents both partial and mixed assignment but at the cost of disallowing assignment through base class. (Solution is to introduce an abstract base class whose copy / move assignment operation is `protected`)




```cpp
class AbstractBook {
    std::string author, title;
    int numPages;
  protected:
    AbstractBook& operator=(const AbstractBook & o) {
        if (this == &o) return *this;
        author = o.author; title = o.title;
        numPages = o.numPages;
        return *this;
    }
  public:
    virtual ~AbstractBook() = 0;
    // ctors, accessors, mutators
};

AbstractBook::~AbstractBook() {} // must be implemented and not inline

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

- Nobody from outside the hierachy can assign using `(AbstractBook *)` or `(AbstractBook &)` since it is `protected`. (This prevents partial and mixed assignment)
- It can only assign "like" objects i.e. `Text = Text`

```
Comic& Comic::operator=(const Comic & o) {
    AbstractBook::operator=(o);
    hero = o.hero;
    return *this;
}
```

20.2 Casting

Remember that `C` casts as `float f = (float) 4;` (Cast may cause "widening" or "narrowing" of the data). This method of casting is not typesafe which is forbidden in `CS246` .`C++` provides four different ways to cast information

1. **Static Cast** (*): you are guaranteeing that the compiler that the cast is safe and will work; otherwise, unspecified behaviour.

```
int pi = std::static_cast<int>(3.1415);
Book *bptr = new Text { ... };
Text *tptr = std::static_cast<Text *>(bptr);
Text &tref = std::static_Cast<Text &>(*bptr);
Text t = std::static_cast<Text>(*bptr);
```

2. **Reinterpret Cast**: Unsafe, implementation-dependent for "weird" casts where behaviour may be unspecified. Consider the following examples in repo,

   - Makes a `private` datafield into `public`
   - Converts a 1-D array (on heap) to a 2-D array

   We will later explore object layout to see how compiler implements `virtual` methods

```
Text t { ... };
Comic c = std::reinterpret_cast<Comic>(t);
```

3. **Const Cast**: Useful but use it with caution. It is used to add or remove the `const`ness from something

```
void f(Student & s);
void g(const Student & s) {
    f(std::const_cast<Student &>(s)); // f COULD change s
}
```

4. **Dynamic Cast** (**): It is used when conversion might fail

   - If we are trying to convert from 1 pointer type to another and the conversion fails, it returns a nullptr

```
Book *bptr = ...;
...
auto p = std::dynamic_cast<Text *>(bptr);
if (p != nullptr) { ... p->xxx ... }
```

   - If casting to an object or reference and cast fails, the exception `std::bad_cast` is raised

```
try {
    Text &tref = std::dynamic_cast<Text &>(bref);
    cout << tref.getTopic();
} catch (std::bad_cast &e) { ... }
```

N.B.: only works for inheritance hierachy with at least 1 `virtual` method due to need to check runtime type information (RTTI)
Aside: 4 types of casting for smart pointers but **only** works on `std::shared_ptr`

```
static_pointer_cast, const_pointer_cast, dynamic_pointer_cast,
↪ reinterpret_pointer_cast
```

Given that we now have `std::dynamic_cast`, we can fix our version of assignment that was `virtual`

```
class Book {
    ...
  public:
    virtual Book& operator=(const Book &);
    ...
};
Text& Text::operator=(const Book & o) {
    if (this == &o) return *this;
    Text *tptr = std::dynamic_cast<Text *>(o);
    if (tptr == nullptr) {
        cerr << "RHS not a Text"
        return *this;
    }
    Book::operator=(o);
    topic = tptr->topic;
    return *this;
};
```

21 Lecture 21

21.1 Static Fields and Methods

Definition 21.1: Static

It can be used outside of an object instance. It exists outside of th eobject memory, it is associated with the class.

```
class Student {
  public:
    inline static int NUM_INSTANCES = 0;
    static int getNumInstances() { return NUM_INSTANCES; }

    Student(int as, int md, int fi) : as{as}, md{md}, fi{fi} {
        ++NUM_INSTANCES;
    }
};
```

Definition 21.2: Inline Keyword

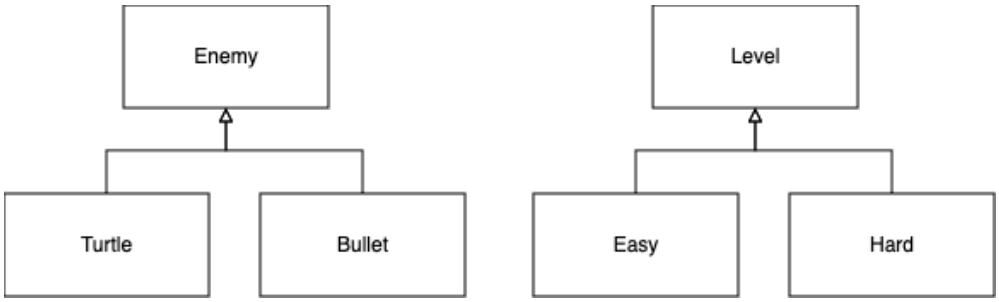
Lets you initialize variables in the `.h` class definition

```
Student s1 = { ... }
cout << Student::NUM_INSTANCES << endl; // 1
Student s2 = { ... }
cout << Student::NUM_INSTANCES << endl; // 2
cout << Student::getNumInstances() << endl; // 2
```

21.2 Factory Method

Consider the scenario where we want to write a video game with two kinds of enemies,

- Turtles and Bullets
- Turtle flies bullets
- Bullets kills us
- We need to shoot down turtles so they don't kill us
- We also have two levels, easy and hard. More bullets in hard.




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

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

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

int main() {
    Player *p = ...
    Level *ez;
    Level *op;
    Level *currentLevel = ez;
    while (p->notDead()) {
        // generates enemy based on level
        e = currentLevel->createEnemy(); // factory of enemy
    }
}
```

21.3 Template Method

Definition 21.3: Template Method

- Base class implements the template or skeleton and the subclasses fills the blank
- Base class allows overriding some `virtual` method but other non-virtual methods must remain unchanged
- Some methods in base class are `virtual` and some are not

```
1  class Turtle {
2      void drawFeet() { ... }
3      void drawHead() { ... }
4      virtual void drawShell() = 0; // private func can also be virtual
5    public:
6      void draw() {
7          drawHead();
8          drawShell();
9          drawFeet();
10     }
11 };
12
13 class RedTurtle : public Turtle {
14     void drawShell() override {
15         // draw red shell
16     }
17 };
18
19 class GreenTurtle : public Turtle {
20     void drawShell() override {
21         // draw green shell
22     }
23 };
```

In the example, the subclass can only change the way the shell is drawn since we want to have a standard way to draw head and feet.

Remark.

Virtual methods:

- public: interface for client, promise certain behaviour - promise pre/post conditions and class invariants
- private: an interface to subclasses, behaviour can be replaced by anything the subclass wants

Definition 21.4: NVI Idiom

All `public` methods should be made non-`virtual` and all `virtual` metods should be private or at least protected (except destructors).

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

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

If we need to exert extra control over `play()` we can do it. We can later decide to add extra code before or after `doPlay()`. We can also add more hooks by calling `virtual` methods from `play()` (e.g. `showCoverArt()`). It is much easier to take this kind of control over our virtual methods from the beginning that to try to take back control over them later.

Remark.

> The NVI Idiom extends to the Template Method Pattern by putting every virtual method inside a non-`virtual` wrapper. We aren't losing efficiency because a good compiler can optimize away the extra function calls.

22 Lecture 22

22.1 Exception Safety

Exception safety does not imply that exceptions we never raised, nor that all exceptions are handled. Consider the execution of some function f, what levels of "exception safety" can f guarantee? There are 3 levels:

1. **Basic**: If an exception occurs, then the program must be in some valid but unspecified state. (No data is corrupted, class invariants still hold, no memory leaks)

2. **Strong**: Provides everything Basic that the basic exception safety level guaranteed does, but in an addition restores state to be as if `f()` was never called (If `f()` has non-local side-effects (changes global or static data, I/O etc.), we may not undo changes in which case can't provide a strong exception safety level guarantee)

3. **No Throw**: Does not let exceptions propogate and guarantees that `f()` succeeds (extremely hard)

Consider the following code

```
1  class A { ... public: void g(); ... };
2  class B { ... public: void k(); ... };
3  class C {
4      A a;
5      B b;
6    public:
7      void f() {
8          a.g();
9          b.k();
10     }
11 };
```

If `A::g` and `B::k` provides no exception safety level guarantees, `C::f` may not be able to provide any exception safety level guarantees.
Let's assume that both `A::g` and `B::k` provide strong exception safety level guarantee

```
1  void C::f() {
2      a.g(); // 1
3      b.k(); // 2
4  }
```

1. If `a.g()` throws an exception, we know `a` is back in original state due to strong exception safety level guarnatee. (No handler, so stack unwinds, remove `f`'s frame and as if `c::f()` never executed)

2. If `b.h()` throws an exception but `a.g()` does not, we will need to undo changes of `a.g()` if `c::f` is going to meet strong exception safety level guarantee

Since the compiler provides copy and move, we can do the following (copy and swap variant)

```
1 void C::f() {
2     A tmpA{a}; // copies
3     B tmpB{b}; // copies
4     tmpA.g();
5     tmpB.k();
6     // if either raises an exception originals are untouched
7     a = tmpA;
8     b = tmpB;
9 }
```

If copy assignment cannot fail, it meets strong exception safety level guarantees.

Question: 22.1

What happens if the copy assignment fails (e.g. deep copy or heap ran out of space?)

We will use the Pointer to Implementation (PImpl) idiom

```
1  struct CImpl { A a; B b; };
2  class C{
3      std::unique_ptr<CImpl> pImpl; // assume initialized in C constructor
4    public:
5      void f() {
6          auto tmp = std::make_unique<CImpl>(*pImpl);
7          tmp->a.g(); // changes the copy
8          tmp->b.k();
9          std::swap(pImpl, tmp); // no throw
10     }
11 }
```

22.2 Exception Safety and std::vector

- `std::vector` is implemented using a dynamic array (uses RAII)
- If type is some (class) C:

```
1 vector<C> v1; // ownership i.e. destroys all C INSTANCES
2 vector<unique_ptr<C>> v2; // ^
3 vector<C*> v3; // not owner "has a"
```

`std::vector` requires that `C` provides some form of copy assignment

std::vector::emplace_back offers a strong exception safety level guarantee
However, copying is slower and more expensive than move

Question: 22.2

Can `emplace_back` take advantage of move semantics?

If move assignment fails partially through moving data from the old array to the new array, our original array state changed!

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

Fact 22.1

Best Practice: mark move, swap operations as `noexcept` so that compiler can optimize

22.3 Template Functions

Consider the function,

```
int min(int a, int b) {
    return a < b ? a : b;
}
```

Turning `min()` into a template function requires that the `type T` with which its instantiated provides `operator<`

```
template<typename T>
    T min(T a, T b) {
        return A < b ? a : b;
    }

cout << min<int>(3, 5);
cout << min(3, 5); // <int> type deduction exists!
cout << min('x', 'y'); // <char>
cout << min(1.2, 6.9); // <float>
```

Fact 22.2

So as long as the parameter type is not ambiguous, compiler can deduce it.

Remember the following

```
1 void foreach(AbstractIterator & start, AbstractIterator & stop, int (f*)(int)) {
2     while(start != stop) {
3         f(*start);
4         ++start;
5     }
6 }
```

Can generalize this further with templates (`std::for_each` from `STL <algorithms>`)

```
1 template<typename It, typename Fn>
2     void for_each(It & start, It & stop, Fn f) { // same as before }
```

This will work on arrays and pointers to!

23 Lecture 23

23.1 STL Algorithms (cont')

Remeber the `std::for_each` from last lectures

```
void f(int n) { std::cout << n << std::endl; }
int a[] = {1, 2, 3, 4, 5};
for_each(a, a + 5, f); // prints array contents
```

Let's look at some others,

```
template <typename Iter, typename T>
Iter find(Iter & start, Iter & stop, T & val);
// searches [start, stop) for "val" => T::operator==
// If found, returns iterator pointing to match
// Otherwise, returns "stop"
```

There is also a similar function `std::count` that returns the number of matches

```
template <typename InIter, typename OutIter>
OutIter std::copy(InInter & start, const InIter & stop, OutIter & out);
// Copies from [start, stop) into out and returns where outer stopped

std::vector v{1, 2, 3, 4, 5, 6};
std::vector<int>w(4); // reserves space for 4 ints
std::copy(v.begin() + 1, v.begin() + 5, w.begin()); // copies 2 3 4 5 into w
```

Remark.

> Current form of iterator won't increase vector capacity under `operator=` if we run out of space. We must ensure output container has enough space.

Consider `std::transform` i.e. works just like `std::copy` but adds a function to change each element at the end

```
template <typename InIter, typename OutIter, class fn>
OutIter std::transform(InIter & start, const InIter & stop, OutIter & out, Fn f);
// applies f to each element copied

std::vector v{1, 2, 3, 4, 5};
std::vector(v.size());
transform(v.begin(), v.end(), w.begin(), add1);
// w contains {2, 3, 4, 5, 6}
```

Let's look at generalizing how we cann `std::transform`, starting with the function (pointer)

[Look at function objects and lambdas]

Definition 23.1: Function object

Object that behaves like a function (Class has `operator()`) as a method

It start with replacing the function `add1`

```
class Plus1 {
  public:
    int operator()(int i) {
        return ++i;
    }
};
Plus1 p;
cout << p(4); // 5
transform(v.begin(), v.end(), w.begin(), Plus1{});
```

Question: 23.1

What if we want to add a user-specified amount to every int?

We can initialize with amount and use in `operator()`

```
class Plus {
    int m;
  public:
    Plus(int m) : m{m} {}
    int operator()(int i) { return i + m; }
}
transform(..., Plus{5}); // adds 5 to each element
```

Key benefit to a function object its its ability to remember state.

```
class IncPlus {
    int m = 0;
  public:
    int operator()(int n) { return n + m++; }
};
```

23.2 Lambdas

Question: 23.2

What if the function is only ever used in one place in the code?

We can replace a lambda declaration in the code

```
even(int n) {
    return n % 2 == 0;
}
vector<int> v;
...
int numEven = std::count_if(v.begin(), v.end(), even);
int numEven = std::count_if(v.begin().v.end(), [](int n){ return n%2 == 0; });
```

23.3 Iterator Library

- `include / import <iterator>`
- Can apply iterators to Streams

```
vector v{1, 2, 3, 4, 5};
std:ostream_iterator<int> out{cout, ", "}; // {"write to", "put after"}
copy(v.begin(), v.end(), out); // outputs to cout "1, 2, 3, 4, 5, "
```

Question: 23.3

Can we now fix things such that `operator=` can increase the vector capacity?

Every container with some form of `push_back` (i.e. vector, deque, list) can use the "back inserter" iterator

```
vector v = { ... };
vector<int> w;
copy(v.begin(), v.end(), back_inserter(w.begin()));
```

23.4 Casting

Question: 23.4

Is `std::dynamic_cast` good or bad?

In the case of copy / move assignment under polymorphism in an inheritance hierachy was fine. It can s till easily extend hierachy without affecting assignment. Consider the following

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

This is tightly highly coupled to the `Book hierachy`, so it must be changed if hierachy changes. We have to change all similar code structures, may miss some. A better approach would be to use `virtual` methods i.e. delegate!

```
class Book {
    ...
  public:
    virtual void print() const { cout << "Book"; }
    ...
};
class Text : public Book {
    ...
  public:
    virtual void print() const override {cout << "Text"; }
};

void whatIsIt(shared_ptr<Book> b) {
    if (b) b->print();
    else cout << "Nothing";
}
```

Inheritance (and polymorphism) works well when:

1. Have infinitely extendable hierachy
2. Public interfaces are the same

Doesn't when have only a few classes (known from the start) that will (almost) never change (willing to absorb cost of change if do add a class) and interfaces are very different.

```
1 class Turtle {
2   public:
3     void stealShell();
4 };
5
6 class Bullet {
7   public:
8     void deflect();
9 };
```

24 Lecture 24

24.1 Variant

Question: 24.1

What do we do when we want the equivalent of a polymorphic container [holds (possibly abstract) base class pointers] but inheritance isn't appropriate?

`import / include <variant>`

- Equivalent to a "tagged union" but type-safe
- If attempt to retrieve a type it's not holding, raises `std::bad_variant_access`

```
1 std::variant<Turtle, Bullet> enemy; // can hold a turtle or a bullet
2 // could use "typedef" but we prefer "using"
3 using Enemy = std::variant<Turtle, Bullet>;
4 Enemy e1{Turtle{}};
5 Enemy e2{Bullet{}};
6 Enemy e3; // what happens?
```

Since `Turtle` is listed first and has a default constructor, that's what `e3` holds. We must have the following creation rules:

- Since first-type must have a default constructor, either add a default constructor or reorder classes so one with default constructor is first
- use `std::monostate` as first type (equivalent to a "null object")

```
1  if (std::holds_alternative<Turtle>(e1)) {
2      cout << "Turtle";
3  } else ...
4
5  try {
6      Turtle t = std::get<Turtle>(e2);
7      t.stealShell();
8      ...
9  } catch (std::bad_variant_accessor & e) {
10     ...
11 }
```

Aside: `<optional>` contains `std::optional<T>`. It is used for mostly return types from functions that may fail (i.e. returns T or "nothing")

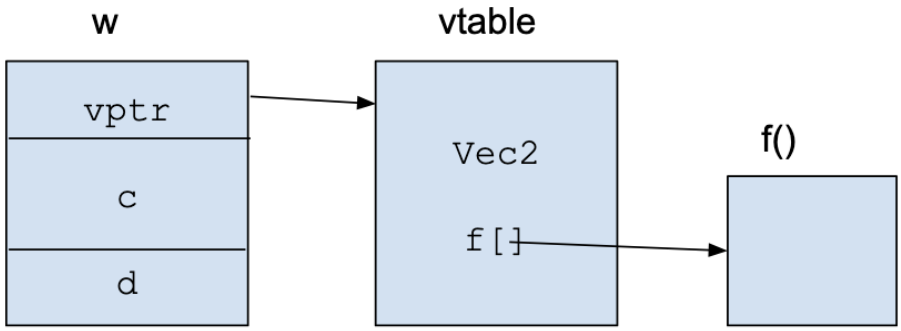
24.2 Compiler Level Virtual Methods

Consider the following code:

```
1  class Vec1 {
2      int a, b;
3    public:
4      void g() { ++a; }
5  };
6
7  class Vec2 {
8      int c, d;
9    public:
10     virtual void f() { ++c; }
11 };
12
13 int main() {
14     Vec1 v;
15     Vec2 w;
16     cout << sizeof(int) // 4
17          << sizeof(v)   // 8 (g turned into a function and stored with others)
18          << sizeof(w)   // 16 (storing an additional pointer)
19 }
```

`w` contains a `vptr` ("virtual pointer") + c + d. All `Vec2` objects have the same address for their vptr. It points to a "virtual table" (`vtable`) that contains runtime type info (RTTI) i.e. "What are you?" and the names and addresses of all `virtual` methods.

```
1  w.f();
```

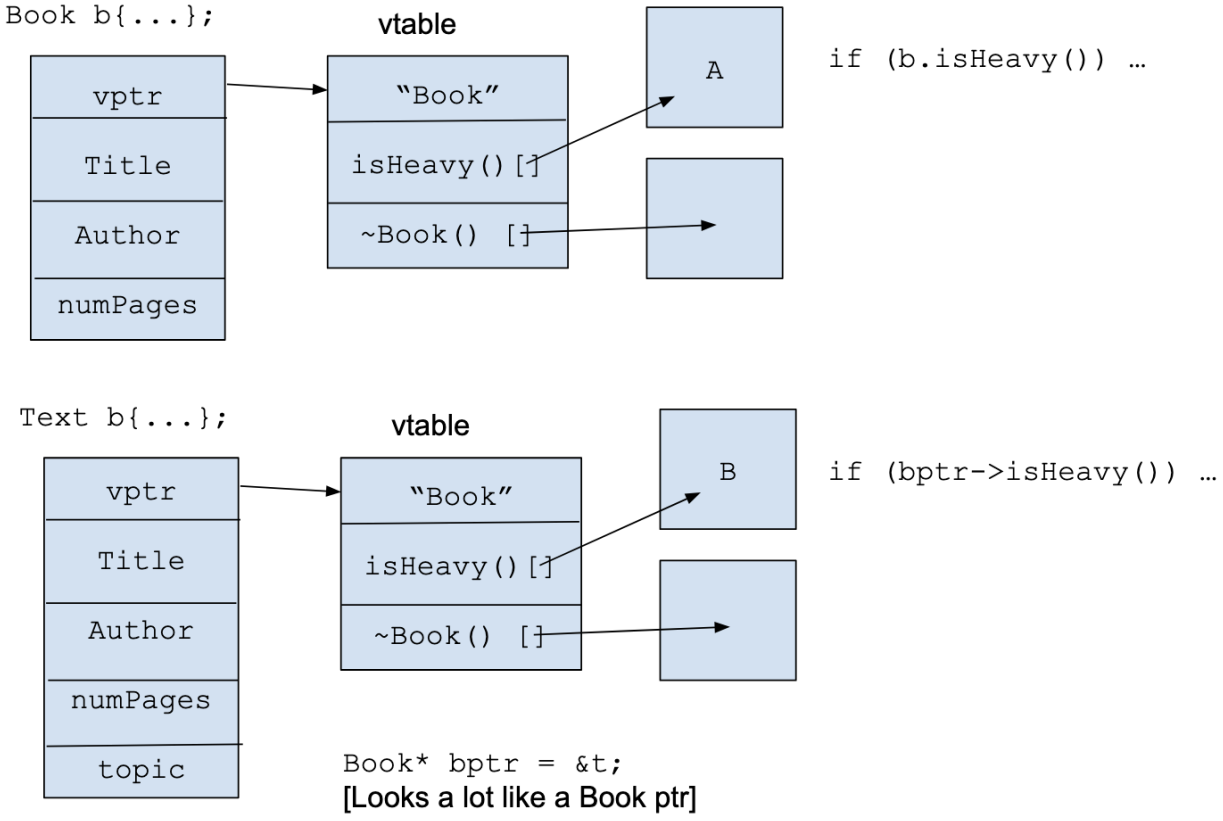


1. Look at `vptr` and to follow to `vtable`
2. Look up location of `f`
3. Go there and execute

Question: 24.2

What happens in an inheritance hierachy with `virtual` methods?

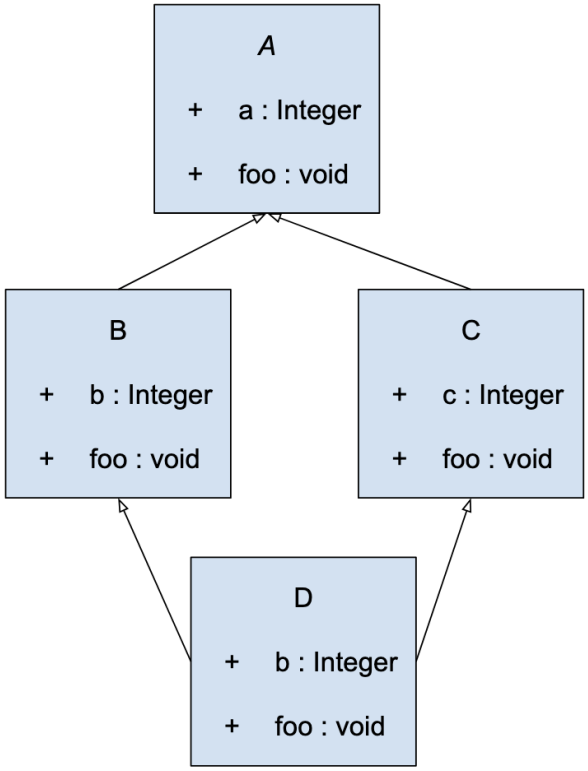
Consider `Book` vs `Text`



24.3 Multiple Inheritance

For example,

- Monotreme which is both a Mammal and a Oviparous
- Conway's Game of Life where a cell is both a subject and an observer



Question: 24.3

Given some object `d` of type `D` how many copies of `a` exists?

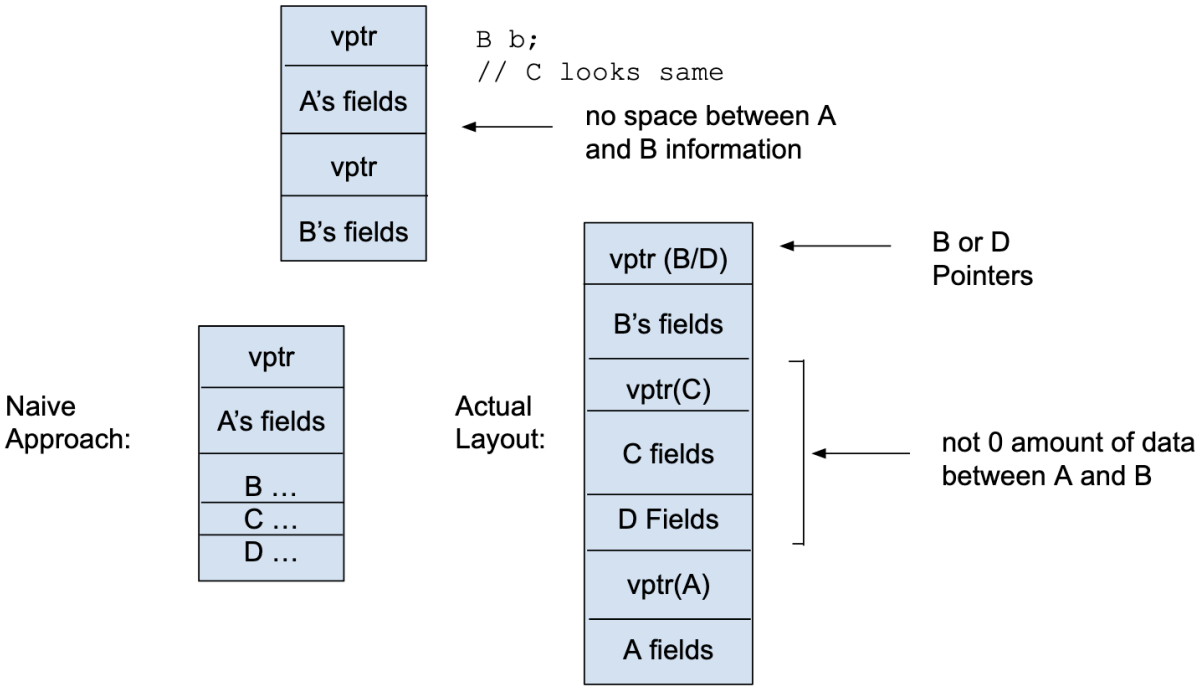
A default compiler choice is 2 copies of `a`. `d.a` is ambiguous! `d.B::a?` `d.C::a?`

Question: 24.4

How do we ensure only 1 copy of `A`'s data?

We use `virtual` inheritance (see C++ IO Modules). We do not change class A but we make B, C inherit virtually from A

```
1 class B : virtual public A { ... };
2 class C : virtual public A { ... };
3 class D : public B, public C { ... };
```



C++20 introduces:

1. Ranges → pair of iterators
2. Views → can be used to eliminate intermediary stage