CS 246 - by Jens Schmitz and Caroline Kierstead

Eason Li

2024 S

Contents

0.1 Lecture 1

Lecture 1 - Tuesday, May 7

Here is an example of outputting “Hello world” in C++ code:

Example 0.1

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

Result 0.1: Remark about the main() Function

Note that the `main()` function has to be `int` (it cannot be `void`). Additionally, return for `main()` is optional in C++.

Definition 0.1: Double Arrow Operators

1. `<<` is known as the **put to operator**
2. `>>` is known as the **get from operator**

Code 0.1

1. `cin`, connected with standard input
2. `cerr`, connected with standard error

```
int main() {
    int x, y;
    cin >> x >> y;
}
```

Remind: It is important to note that `cin` always stops at the whitespace.

Result 0.2: Some General Questions

If we pass in a number that is too large, then the value for x or y (or both) would be set to the largest integer possible. If we pass in text values rather than integers, then the program will fail because it cannot recognize the type of our input. A way to test if out input is legit is to use the `cin.fail()` function.

Code 0.2

Here are three ways to check the status of the input stream:

1. `cin.fail()`
2. `cin.eof()`
3. `cin.bad()`

Here is another example:

Example 0.2

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

Discovery 0.1

Instead of passing `cin.fail()` into the if statement, we can also choose to pass in a `bool`-type value, `cin`, which is another built-in feature.

0.1.1 Basic C++ Types with Their Sizes

Result 0.3

```
bool: 8 bits
char: 8 bits
short: 16 bits
int: 32 bits
long: 64 bits
long long: 64 bits

float: 32 bits
double: 64 bits
long double: 128 bits
```

```
unsigned char: 8 bits
unsigned short: 16 bits
unsigned int: 32 bits
unsigned long: 64 bits
unsigned long long: 64 bits
```

0.2 Lecture 2

Lecture 2 - Tuesday, May 9

Discovery 0.2

In C/C++, use `<<` and `>>` to shift bits, while C++ also uses `<<` and `>>` for I/O. Why does this work?
Answer: Function Overloading.

Example 0.3

For bit shifting, we have

$$21_{10} >> 3_{10} \quad \Rightarrow \quad 2_{10}$$

0.2.1 Function Overloading

Definition 0.2: Function Overloading

When the compiler compile, it decides which version of the function to call based upon the number and/or types of parameters, but **not** the return values. Therefore, parameters must be **unambiguous**.

Example 0.4

```
int operator>>(int, int)
std::istream& operator>>(std::istream& in, char& c)
```

Discovery 0.3

If reading in 2 ints and "fail", what happens? (e.g. `cin >> x >> y;`)
Solution:
Could run out of input (EOF) for either x or y;
or read value is larger than `INT_MAX` or smaller than `INT_MIN`.
or even not an integer;

□

Discovery 0.4

How can we detect failure?

Solution: `std::cin` is an instance of `std::istream` has “state bits” that tell us the condition of `cin`. READ FIRST, THEN CHECK.

□

Result 0.4: good(), eof(), fail(), and bad()

We have the following:

1. `cin.good()` ⇒ true if succeded;
2. `cin.eof()` ⇒ true if EOF;
3. `cin.fail()` ⇒ true if either EOF or didn’t get int.
4. `cin.bad()` ⇒ unrecoverable error.

Remark: Remember that a too big/small integer corresponds to `cin.fail()`

Definition 0.3: Operator `bool()`

Refer back to discovery 0.1, `std::istream` has a function that can convert an istream to a bool.

Example 0.5

In particular, `if(cin) ...` would work just fine.

It is also important to point out that we can call `!` on `cin`.

Discovery 0.5

`bool std::istream::operator !()` that returns `std::istream::fail()`.

Now, we can revise program to “throw away” non-ints, output ints, and stop on EOF.

Example 0.6

```
int main() {
    int i;
    while (true) {
        cin >> i;
        if (cin.eof()) break;
        else if (cin.fail()) {
            cin.clear();
            cin.ignore();
        } else {
            cout << i << endl;
        }
    }
}
// e.g. 3 a b 1 2 EOF -> would loop 5 times
//                       it reads 1 and 2 together since
//                       it reads the longest possible value
```

0.2.2 Manipulators – <iomanip>

Definition 0.4: Manipulators

C++ has manipulators to format I/O.

Example 0.7

It can be used to

1. print numbers as hexadecimal/octal/decimal, bool alpha, setw, setfill, skipws, noskipws e.g. `cout << hex << value;`, and all the subsequent numbers are hexadecimal now, and this is what we call "sticky". (Best practice: undo any "sticky" changes.)

Example 0.8: Manipulator

```
cout << 95 << endl;          // prints 95 as a decimal
cout << hex << 95 << endl;   // prints 95 as a hex
         ^ manipulator

cout << 15 << endl;          // gets printed out in hex
                             // until client change it back
cout << dec << 15 << endl;   // prints 95 as a decimal
```

0.2.3 Strings – <string>

C uses (`const char *`) and (`char *`) where every string terminates with, ‘ `\∅` ’, the null character. Append requires reallocation, so we need to worry about memory leak.

Remark: The library is part of std namespace, so with `g++-11`, **always** import and compile string last.

Example 0.9

```
import <string>;
int main() {
    string s1;                     // empty string, s1.size() = 0
    string s2 = "text"             // assign an actual string
    cin >> s1;                     // reads whitespace-delimited word
    cout << s2 << s1 << endl;      // output
    s1 + s2;                       // append two strings
    string s3 = s2 + s2;           // valid
    string line;
    getline(cin, line); // reads an entire line up to '\n' (exclusive)
                        // remember that whitespaces are included
}
```

0.3 Lecture 3

Lecture 3 - Tuesday, May 14

We continue on strings. Recall the example 0.9

Example 0.10: Anonymous Object

```
    void f(std::string s) {cout << s << endl};
    ```
        f(s1);   // this is valid
        f(std::string{"Stuart"});   // anonymous object,
                                     // only exists for the
                                     // span of the call of f
    '''
```

**Definition 0.5: What is a “Stream”**

A **stream** is an abstraction wrapped around input and output. eg. keyboard, files, screen, etc.

We know that C++ already has streams:

1. **input** i.e. `std::cin` (std::istream)
```

2. **output** i.e. `std::cout` and `std::cerr` (std::ostream)

Result 0.5

It is perfectly legal to have

`std::istream *ip = & std::cin;`

0.3.1 Files – <fstream>

Example 0.11: File

```
// C Version
include <stdio.h>
int main() {
    char s[256];
    FILE *f = fopen("file.txt", "r");
    while (1) {
        fscanf(f, "%255s", s);
        if (feof(f)) break;
        printf("%s\n", s);
    }
    fclose(f);
}


// C++ Version
import <iostrem>;
import <fstream>;
import <string>;
using namespace std;
int main() {
    string s;
    ifstream in{"file.txt"};  // automatically open the
                              // file in read mode
                              // in.fail() is true if
                              // couldn't be opened for
                              // reading
    while (in >> s) {
        cout << s << endl;
    }  // by defnition, when we reach here, file is
       // closed and both in and s are destroyed
       // (they are destroyed in reverse order).
}
```

An `std::ifstream` can do *everything* an `std::istream` can.

Example 0.12: Hint for HW1 Q3

```
// the following are all legal
std::istream *ip2 = &in;  // legal
void bar(std::istream* in) {...}

bar(&cin);
bar(&in);
bar(&std::ifstream{"in.txt"});  // anonymous object
```

0.3.2 String Streams – <sstream>

Definition 0.6: sstring

Combination of string and stream.

1. `std::ostringstream` : Used to convert (and possibly concatenate) data that can then be retrieved as a C-style string.

   Example 0.13

   ```
   int main() {
       std::ostringstream oss;
       int num1 = 123, num2 = 456;
       oss << "Numbers are: " << num1 << " and " << num2;
       std::string output = oss.str();
       std::cout << output << std::endl;
       return 0;
   }
   ```

   Example 0.14: int to string

   ```
   std::string convertInt2String(int i) {
       std::ostringstream oss;
       oss << i;
       return oss.str();
   }
   ```

2. `std::istringstream` : Used for input operations from a string.

Example 0.15

```
int main() {
    std::string input = "123 456";
    std::istringstream iss(input);
    int num1, num2;
    iss >> num1 >> num2;
    std::cout << num1 << num2 << std::endl;
    return 0;
}
```

Example 0.16: string to int

```
int covertString2Int(std::string s) {
    std::istringstream iss{s};
    int i = 0;
    iss >> i; // iss.fail() if couldn't read int
    return i;
}
```

Until we cover exceptions, `std::istringstream` is the only way to convert a string to an int. Here is an example of determining whether we actually got an integer as our input.

Example 0.17: Variable Local to `if`

```
int i;
while (true) {
    cout << "Enter a number: ";
    string s;
    cin >> s;
    if (cin.eof()) break;
    if (istringstream iss{s}; iss >> i) break;
    cout << "I said " << i << endl;
}
```

Example 0.18

```
string s;
while (cin >> s) {
    int n;
    if (istringstream is{s}; is >> n) {
        // no clear/ignore since
        // created on every entry to the if
        cout << n << endl;
    }
} // repeats until EOF, outputting ints
```

0.3.3 Command-line Arguments

Example 0.19

`./pgm abc 123 < file.in 2 > err.txt 1 > out.out`.

The part `< file.in 2 > err.txt 1 > out.out` is **bash redirection**, it is *not* part of command-line, while `abc 123` are arguments to the program. The part `./pgm abc 123` is C/C++ “command-line” arguments.

0.3.4 Read Arguments

Example 0.20: Read Arguments

In C++, `argc` and `argv` are parameters of the main function that are used to capture command-line arguments. As long as I have argument other than the file name, I enter the loop. I basically take the string and create a anonymous object, and try to read an integer out of it.

```
// argsSum.cc
import <iostream>;
import <sstream>;
import <string>;
int main(int argc, char* argv[]) {
    int total = 0;
    for (int i = 1; i < argc; i++) {
        std::string arg = argv[i];
        if (std::istringstream{arg} >> n) total += n;
    }
    std::cout << total << std::endl;
}
```

0.4 Lecture 4

Lecture 4 - Tuesday, May 16

0.4.1 Function Overloading

We already saw this when << and >> were used for either bit-shifting or I/O,
eg. `int negInt(int i) {return -i;}` or `bool negBool(bool b {return !b;})`

Definition 0.7: What is "overloading"

Some function name used; but parameter list must differ in number and/or parameter types
Remark: Return values are not part of function resolution.

Example 0.21: Here we overload neg() function

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

int main () {
    cout << neg(3) << " " << boolalpha << neg(true) << endl;
}
```

Discovery 0.6

What if we want a function that either prints the "stem", one per line, for the specified file name, or for the file "suite.txt".
eg. `printSuiteFile();` \\ outputs content of "suite.txt";
eg. `printSuiteFile(‘‘stuart.txt’’);` \\ outputs "stuart.txt";

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

// header
void printSuiteFile(string fname = "suite.txt");
// whenever the function is called without passing in a parameter,
// the filename "suite.txt" will be passed in automatically
```

The only thing we know the actual number of parameters is at the function call site when we compile. The compiler generates any missing information using the specified values. This allows the function code to retrieve the full parameter list of values off of the run-time stack. Otherwise, could try to access missing information from where it shouldn't.

Warning: All default values must be placed at the end of the parameter list. The values must be given in the declaration (if separate compilation).

0.4.2 Struct keyword

- backwards compatible with C

eg. `/* C */`

Example 0.22

```
// C
struct node {
    int value;
    struct node *next;
}
typedef struct node Node;

Node n; n.value = -1; n.next = NULL;

// C++
struct Node {
    int value;
    Node *next;
};  // semicolon is required in C++

Node n; n.value = -1; n.next = nullptr;
```

Discovery 0.7

Why doesn't this Node definition work?

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

This is a compile error since we are missing the "*".

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

0.4.3 Constants

The old C way uses `#define` but it isn't type safe. Meanwhile, C++ uses `const` keyword.
Remark: We cannot change constant values ever.

Example 0.23

```
// following the code above
// aggregate initialization
Node n1{5, nullptr};

const Node n2;
n2.value = 5;  // NO! compile error

const Node n3{n1};  // copies n1 into n3,
                    // which is immutable

const Node n4{-1, &n1};
```

0.4.4 Parameter Passing

Recall what we had in CS136, in C, we either pass by value or pass by address(pointer).

Discovery 0.8

If I have

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

Why don't we have "&" anywhere?
C++ introduces "references" as a third parameter passing mechanism.
eg. `std::istream& operator >> (std::istream& in, int& value);`
A "reference" is a constant pointer that is automatically dereferenced.

0.5 Lecture 5

Lecture 5 - Thursday, May 23

0.5.1 Parameter Passing

Consider the code below:

Example 0.24: Pass in a copy

```
int x{5};                              void inc(int n) {
inc(x);                                    n = n + 1;
cout << x << endl;                     }
```

In the above example, we passed in a copy of x, which is independent to the original x.

Alternatively, we can do the following

Example 0.25: Passing by pointer

```
int x{5};                              void inc(int *np) {
inc(&x);                                   (*np)++;
cout << x << endl;                     }
```

Here we passing by pointer. In this case, we will get 6 as our output.

Theory 0.1: Reference

```
int y = 10;
int &z = y; // reference, z is a reference to y,
            // (z is another name for y)
z = 12;     // now y becomes 12
```

Result 0.6

References must be initialized, i.e.

```
int y = 10;
int &z;
z = y;
```

would gives us an error. Additionally, we cannot change what z is alias of, it behaves like a constant.

Result 0.7

Reference takes no memory.

We can also do the following:

Discovery 0.9

```
int ints[5] = {1, 2, 3, 4, 5};
int &i2 = ints[2];
i2 = 30;
```

Now, *ints*[2] has a value of 30.

Therefore, we can have the following

Discovery 0.10

```
int x{5};                        void inc(int &n) {
inc(x);                              n = n + 1;
cout << x << endl;               }
```

This now prints 6 because we passed in the value by reference. n is bound to x at runtime, so that n is an alias of x.

Example 0.26

```
cin >> x;
```

>> is a read function that takes x by reference and therefore can change x.

Definition 0.8

References are like const pointers with automatic dereferencing.

What cannot we do with reference ?

1. Cannot leave them uninitialized:

```
int &z;          // not ok
int &z = y;      // ok
int &z = x + y;  // not ok
```

2. Cannot create a pointer to a reference:

```
int &*x;  // this would mean x is a pointer to an
          // int reference
```

3. Cannot create a reference to a reference

4. Cannot create an array of references:

```
int &r[3] = { x, y, z };  // this is not valid
```

What can we do with reference ?

1. Use as function parameters:

```
        void inc(int &z);
```

0.5.2 Passing Parameters to function using Reference

```
struct reallyBig {...};

int f(reallyBig rb) {...}     // copy of parameter is made,
                              // potentially slow
int g(reallyBig &rb) {...}    // call by reference, fast
                              // but g can change rb in the caller
int h(const reallyBig &rb) { // preventing function to update rb
                              // in the caller
    reallyBig rbCopy = rb;    // in this way, we can modify rbCopy
}
```

Discovery 0.11

When passing data to functions, prefer pass-by-const-ref for anything larger than a pointer/ integer, unless the function needs to make a copy anyways.

Code 0.3

```
int f(int &n) {...};
f(5);  // this is not ok, because n could potentially
       // be used to update the literal 5
```

Alternatively,

```
int g(const int &n) {...};
g(5);  // this is valid, because we are gauranteed
       // not to change n
```

Theory 0.2

Similarly, (with the similar reason), we can have

```
cin >> x;   // we want to change x
cout << x;  // we don't want to change x,
            // thus this is equivalent to
            ostream &operator<< (ostream &o, const int &x)
```

0.5.3 Dynamic Memory Allocation

Recall that in C, we have

Code 0.4

```
int *p = (int*)malloc(10 * sizeof(int));  // allocate array of
                                          // 10 ints on heap
p[0] = 10;
p[1] = 20; ...
free(p);
```

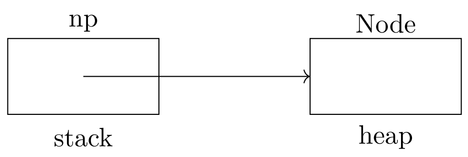
In C++, we have `new` and `delete` instead:

Code 0.5

```
struct Node {...};
Node *np = new Node;  // returns a pointer to the object,
                      // it is type-safe,
                      // it allocates memory from heap
delete np;            // returns memory to the heap
```

Physical Memory Model

code	programs that are loaded and executing
static data	literals / global variables
free store (heap)	memory for new operations
stack	function arguments, local storage



Example 0.27

We can also create an array of 10 Node objects:

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

Result 0.8

In this case, when we call delete, we have to

```
delete [] nodeArray;  // tells delete operator that
                      // nodeArray is an array,
                      // Note that we don't have to
                      // specify how big is the array
```

Theory 0.3

Memory allocated with `new` must be deallocated with `delete`; or
Memory allocated with `new` must be deallocated with `delete []`.

0.6 Tutorial 1

See the question at

CS246 - 2024S/T/tutorial01.pdf

Solution:

```
#include <iostream>
#include <string>
#include <sstream>
using namespace std;

int main() {
    int lenList[2];
    lenList[0] = 2147483647;
    lenList[1] = 0;
    string stringList[2];

    string line;

    while (getline(cin, line)) {
        string s;
        int length = 0;
        istringstream iss(line);
        while (iss >> s) {
            length ++;
        }

        if (length < lenList[0]) {
            lenList[0] = length;
            stringList[0] = line;
        }
        if (length > lenList[1]) {
            lenList[1] = length;
            stringList[1] = line;
        }
    };

    cout << stringList[0] << endl;
    cout << stringList[1] << endl;

    return 0;
}
```

Result: 14/14.

0.7 Lecture 6

Lecture 6 - Tuesday, May 28

Pass by reference is a very powerful mechanism. We can have a function operates directly on an object we pass by reference:

Example 0.28

```
void swap(double &d1, double &d2) {
    double temp = d1;
    d1 = d2;
    d2 = temp;
}
double x = 5;
double y = 10;
cout << x << ", " << y << endl;  // 5, 10
swap(x, y);
cout << x << ", " << y << endl;  // 10, 5
```

Note: the standard library also provides a swap function, which works for any data type.

0.7.1 Returning Values from Functions

– – – Returning by Value:

```
Node getNode() {
    Node n;
    ...;
    return n;  // Node created in the function is copied into n1
}
Node n1 = getNode();
// "Expensive", return by value,
// we will see shortly that this is not so expensive after all
```

Discovery 0.12

Can we avoid the overhead of copy by returning a pointer or a reference instead?

– – – Returning by Pointer Version:

```
Node *getNodePtr() {
    Node n;
    ...;
    return &n;
}
Node *n1 = getNodePtr();  // this returns pointer to stack-
                          // allocated data, which is dead on return
```

– – – Returning by Reference Version:

```
Node &getNodeRef() {
    Node n;
    ...;
    return n;
}
Node &n1 = getNodeRef();  // BAD! Returning a reference to stack-
                          // allocated data, which is dead on return
```

– – – Returning by Pointer Version (Heap):

```
Node *getNodePtrOnHeap() {
    Node *n = new Node;          // Allocate the Node on heap
    ...;
    return n;
}
Node *n1 = getNodePtrOnHeap();   // GOOD! the object is alive
                                 // until we delete it
```

Result 0.9

We have to remember to call delete, the function transfers the ownership of the allocated memory to the caller of the function. That caller is responsible for calling delete (or its own caller).

Theory 0.4

Never call `delete` on a stack-allocated (local) object.

Example 0.29

```
void f() {
    Node n;
    ...;
    delete n;    // BAD! won't even compile
                 // because delete takes in a pointer
    delete &n;   // BAD! n is in stack. Compile warning
    int a[10];   // a is actually a int*
    delete a;    // BAD! a is in stack
}
```

0.7.2 Operator Overloading

This allows us to use built-in operator with user-defined types we create.

```
struct Vec {
    int x, y;
}
vec v1 = {1, 2};
vec v2 = {4, 5};
vec v3 = v1 + v2;
vec v4 = 5 * v3;
```

```
Vec operator+ (const Vec &v1, const Vec &v2) {
    Vec v {v1.x + v2.x, v1.y + v2.y};
    return v;
}  // notice that we pass in by reference

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

Theory 0.5

1. We cannot change operator precedence;
2. Operator must have at least one user-defined type parameters. i.e. cannot override 1 + 2.
3. Cannot change the syntax of how the operators are used;
4. Cannot create new operators like and \$, can only overload existing operators
5. cannot overload
   (a) scope resolution operator `::`, (e.g. `std::cout`);
   (b) member selector `.`, (e.g. `v.x`);
   (c) dereference operator `*`, (e.g. `*p`);
   (d) ternary operator `?  ;`, (e.g. `x==1 ?  true ; false`).

We can also do the following:

Example 0.30: Overload << Operator

```
ostream &operator<< (ostream &out, const Vec &v) {
    out << "(" << v.x << ", " << v.y << ")" << endl;
    return out;
}
Vec v1 = {5, 10};
cout << v1,  // (5, 10)
```

Example 0.31: Overload >> Operator

```
istream &operator>> (istream &in, Vec &v) {
    char p1, c, p2;
    in >> p1 >> v.x >> c >> v.y >> p2;
    return in;
}
```

It is common to overload other arithmetic operators too. There are shortcuts that let us implement a bunch of these with one function, we will explore later.

Discovery 0.13

Why is it okay for `operator<<` to return an `ostream&` ?
Answer: The out varaible we return is the same reference that we pass in, and it refers to an object that exists somewhere other than the stack. This object has a longer life.

0.7.3 Separate Compilation

This allows us to split programs into modules where each module provides two things:

1. an interface:
   (a) type definitions
   (b) function prototypes
2. an implementation: full definitions for every provided function

Recall:

Code 0.6

1. Declaration: asserts the existence of a type, function, global variable, gives it a name;
2. Definition: full details of a type or function; allocates space for variables and function bodies.

Example 0.32

```
int f(int a, string b);
int f(int a, string b);  // declrations can be repeated, but
                         // they need to be the same every time
extern string params;
```

0.8 Lecture 7

Lecture 7 - Thursday, May 30

Recall definitions:

```
int f(int a, string b) {...};
string params = "0, 5, abc";
struct Node {
    int data;
    Node *next;
};
```

Discovery 0.14

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

Code 0.7: Modules, Interface, Implementation

```
// Vec interface file (vec.cc)
export module vec;  // indicates that this is the module interface
                    // file, we can have only one such file
export struct Vec { // anything marked export is made available
    int x, y;       // to the client of the module to use
};
export Vec operator+(const Vec &v1, const Vec &v2);

// vec implementation file (vec-impl.cc)
module vec;  // this file is part of module vec.
             // we can have multiple implementation files
             // Don't need to import vec, compiler implicitly
             // imports the interface.

Vec operator+(const Vec &v1, const Vec &v2)
    return {v1.x + v2.x, v1.y + v2.y}  // we don't need to export
                                       // this, already exported
                                       // in the interface file
```

Result 0.10

You can have types, functions, etc. that are used internally by the module hidden from the user of the module. For instance

```
int helper() {...};  // not exposed to users of module
                     // because it is not exported
```

Code 0.8: Client Program

```
// client program (main.cc)
import vec;  // no <> because this is a user-module,
             // not a system module
```

Theory 0.6

Interface files start with `export module _____;`
Implementation files start with `module _____;`

To compile, we need to follow the dependency order:

Code 0.9

```
g++20m -c vector.cc          // creates vector.o
g++20m -c vector-impl.cc
g++20m -c main.cc

g++20m vector.o vector-impl.o main.o -o main
        // links the programs and creates the executable
```

Theory 0.7

A module itself can import the modules it needs for its own implementation.

Repository provides a bash script called *compile* in the tools directory if students want to partially automate compilation.

0.8.1 Benefits of Modules over Header File

1. Faster compilation. The module interface file is compiled only once and used many times
2. If a module A imports module B to use as part of its implementation, the contents of B are not exposed to users in any order

3. Modules can be imported in any order
4. Non-exported implementation details (like helper functions) are not visible outside the module
5. We can use modules and header files together [†].

0.8.2 Classes

We can put functions inside structs.

Definition 0.9: Class

We call structs with functions **classes**.

Example 0.33

```
// student.cc
export module student;
export struct Student {
    int hw, mt, final;
    float grade();
};

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

// client code
import student;
Student s {60, 70 , 80};
cout << s.grade() << endl;
```

Definition 0.10: Class, Object, Method/Member Function, Scope Resolution Operator

Class is essentially a structer type that can contain functions.
Object is an instance of a class.

```
            Student s {60, 70 , 80};
            ^       ^           ^
        class     object       initialization list
```

`grade` is called a **method** or a **member function**, while `::` is called a **scope resolution operator**.

Recall

[†]In this course, stick to one or the other, but not both

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

The parameters are fields of the receiver object, the object of which `grade()` is called.

```
s.grade();  // s is the receiver object
            // uses the values in object s
```

Theory 0.9: "this"

```
float Student::grade() {     // this function actually has a
                             // hidden parameter called this,
                             // a pointer to the receiver object
    return this->hw * 0.4 + this->mt * 0.2 + this->final * 0.4
}
```

Theory 0.9

Compiler inserts "this" automatically.

We can also define the `grade()` function right inside the struct (not encouraged) because the code ends up being inlined wherever it is used.

Discovery 0.15

```
Student s;
Student *p = &s ;   // &s is equivalent to this
                    // inside the method body.
```

0.8.3 Initializing Objects

Recall

Example 0.34

```
Student s = {60, 70, 80};   // uses the order of the
                            // fields in Student
```

A better way to initialize objects is to use a *Constructor* (sometimes we write ctor for convenience).

0.9 Tutorial 2

CS246 - 2024S/T/tutorial02.pdf

Solution:

```
// Fill in the return type and arguments
istream &operator>>(istream &in, FriendList &fl) {
    // Fill in this operator overload
    if (fl.size < 8) {
        in >> fl.friends[fl.size];
        fl.size++;
    }
    return in;
}

// Fill in the return type and arguments
ostream &operator<<(ostream &out, const FriendList &fl) {
    // Fill in this operator overload
    out << "I have " << fl.size << " friend(s). They are:" << endl;
    for (int i = 0; i < fl.size; i++) {
        out << fl.friends[i];
        if (i < fl.size - 1) out << ", ";
    }
    out << endl;
    return out;
}

// For operator-, argument and return types are given
FriendList operator-(const FriendList& fl, int index) {
    FriendList nfl;
    if (index < 0 || index >= fl.size) {
        nfl = fl;
    } else {
        nfl.size = fl.size - 1;
        for (int i = 0, j = 0; i < fl.size; i++) {
            if (i != index) {
                nfl.friends[j] = fl.friends[i];
                j++;
            }
        }
    }
    return nfl;
}
```

Result: 4/4.

0.10 Lecture 8

Lecture 8 - Tuesday, Jun 4

Code 0.10

```
struct Student {
    int hw, mt, final;
    float grade();
    Student::Student(int hw, int mt, int final);  // constructor
}   // same name as the class
    // can be defined with 0 or more parameters that can be used
    // to initialize new objects
    // no return type, not even void
Student::Student(int hw, int mt, int final) {
    this->hw = hw;          // this is needed to tell compiler
    this->mt = mt;          // which hw we are refering to.
    this->final = final;
}

Student s{60, 70, 80};  // cause constructor to run, the values
                        // are passed to the constructor function
Student s = Student{60, 70, 80};
Student s = Student(60, 70, 80);  // this is not recommended
Student s = {60, 70, 80};
```

Result 0.11

The above programs are all equivalent.

To create the Student on heap, we do:

Code 0.11

```
    Student *sptr = new Student{60, 70, 80};
    ...;
    delete sptr;  // don't forget to delete.
```

Definition 0.11: What are the advantages of a constructor?

1. They are functions, so we can write arbitrarily complex initialization code;
2. We can have default parameters;
3. We can use overloading;
4. We can do validation for sanity check;
5. Ensures out object are initialized properly and that the object is *logically* valid.

Example 0.35: Default Values Example

```
Student::Student(int hw=0; int mt=0; int final=0) {
    this->hw = hw;
    this->mt = mt;
    this->final = final;
}
Student s2{60, 70, 80};  // this is valid now
Student s3;              // this is valid now
```

Theory 0.10: What if we don't write a constructor

Every class comes with a default constructor, default-constructs all fields that are objects.
If you write any constructors, the default goes away.

Code 0.12

```
struct Vec {                          struct Basis {
    int x, y;                             Vec v1, v2;
    Vec(int x, int y) {               }
        this->x = x;                  Basis b;
        this->y = y;                  ^
    }                                 // this wont't compile
};                                    // because Vec doesn't have
Vec v;      // Error                  // a constructor that takes
Vec v{1, 2};  // ok                   // no parameters
```

How about this case?

Code 0.13

```
struct Basis {
    Vec v1, v2;
    Basis() {
        v1 = Vec{0, 1};
        v2 = Vec{0, 2};
    }
};
Basis b;
```

This also doesn't compile, the following theorem is the reason why.

0.10.1 Object Creation Steps

Theory 0.11: Object Creation Steps

When an object is created, there are three steps:

1. Space is allocated;
2. Fields are constructed in declaration order; (constructor tuns for the fields that are objects);
3. Constructor body runs.

The initialization of $v1$ and $v2$ must happen in step 2. But in reality, in the above code, the reason why it does not work is that it instead takes place in step 3. How can we accomplish that then?

MIL is the answer.

Definition 0.12: Member Initialization List (MIL)

```
Student::Student(int hw, int mt, int final)
    : hw{hw}, mt{mt}, final{final} {...}
              ^   ^
        fields|   |values
    \______________ _____________/
                  MIL
```

Theory 0.12

MIL runs in step 2.

Result 0.12

MIL must be in the definition of the constructor, it cannot be in the declaration.

Example 0.36

Refer back to the example we had above, it should instead be like

```
Basis::Basis : v1{v1}, v2{v2} {...};
               \______ _____/ \_ _/
                   Step 2     Step 3
```

0.10.2 Default Values in struct

Example 0.37: Default Values in struct

```
struct Basis {
    Vec v1{0, 1};                 // If MIL doesn't mention a field,
    Vec v2{0, 2};                 // these are used instead
    Basis() {...};                // uses default values
    Basis(const Vec &v1, const Vec &v2) : v1{v1}, v2{v2} {...};
}                                 // uses parameters
Basis b;                          // uses default;
Basis b{Vec{0, 1}, Vec{0, 2}};    // uses parameters
```

Theory 0.13

With a MIL, the fields are initialized in declaration order, not in the order provided in MIL.

Consider the following code:

Code 0.14

```
struct Student {
    int hw, mt, final;   // not object
    string name;         // object
    Student (int h, int m, int f, const string &n) {
        hw = h;
        mt = m;
        final = f;
        name = n;
    }   // runs during step 3, it's actually overwriting the
        // default empty string that was stored in name during
}       // step 2. Reassignment.
```

Instead, we should have

Code 0.15

```
Student(int h, int m, int f, const string &n)
    : hw{h}, mt{m}, final{f}, name{n} {...}
// here n is initialized from n in step 2
// this is more efficient! No reassignment in step 3
```

Theory 0.14

MIL must be used

1. for fields that are objects with no default constructor;

2. for fields that are const or references, those must be initialized in step 2

Example 0.38

We can use field in MIL that were initialized earlier too:

```
: x{rand()}, y{x} {...};   // valid
```

0.11 Lecture 9

Lecture 9 - Thursday, Jun 6

0.11.1 Copy Constructor

The copy constructor constructs one object as a copy of another. The goal of copy constructor is to create an independent copy of the object so that data is shared between the objects.

Code 0.16: Writing our own copy constructor for Student

```
struct Student {
    int hw, mt, final;
    Student(const Student &other)
        : hw{other.hw}, mt{other.mt}, final{other.final} {...}
}
// this is equivalent to the built-in copy constructor
Student s1{60, 70, 80};
Student s2 = s1;  // this invokes the copy constructor
```

Theory 0.15

Every class comes with a default copy constructor.

Example 0.39

If we were to code the **default copy constructor** ourselves, for `Vec` it would look like this:

```
struct Vec {
    int x, y;
    Vec(const Vec &other) : x{other.x}, y{other.y} {}
}
```

Remark: Use either the MIL or the constructor body to copy the data from other into the new object.

Why would we need a copy constructor if the compiler always provides a default one?

Answer: Consider the case when there is a pointer. We do not want them to point the same thing! Built-in copy constructor does a shallow copy. If we want a deep copy, we must implement out own copy constructor. The new one will then replace the built-in one.

Code 0.17

```
struct Node {
    int data;
    Node *next;
}
Node *n = new Node{1, new Node{2, new Node{3, nullptr}}};
Node m = *n;          // this calls copy constructor implicitly
Node *p = new Node{*n};  // calls copy constructor explicitly
```

For the code above, we have the following diagram illustrating what exactly is happening:

Discovery 0.16

The problem here is that the default copy constructor did not create independent copies.

To solve the issue, we need to write our own copy constructor:

Code 0.18

```
Node(const Node &other)
: data{other.data},
  next{other.next == nullptr ? nullptr : new Node{*other.next}}
```

This code recursively copies the rest of the list.

Definition 0.13: Implicit Call of Copy Constructor

The copy constructor is implicitly called when

1. an object is initialized with another object of same type;
2. an object is passed by value to a function;
3. an object is returned by value from a function (the truth is somewhat nuanced).

Theory 0.16

Consider the following code,

```
Node(const Node other) {...}
```

why is it wrong?
Answer: We are passing other by value, which ends up calling the copy constructor recursively.

0.11.2 Copy Constructor that Takes Just One Parameter

Code 0.19

```
struct Node {
    ...
    Node (int data) : data{data} next{nullptr} {}
    Node (int data, Node *next = nullptr) : data{data}, next{next} {}
}
Node n = {1};               // valid
Node m = {1, nullptr};      // valid
Node p = {1, new Node{2}};  // valid
```

The above are all legit, but the problem with single-arg copy constructor is that they create implicit recursions. Why is that a problem?

Example 0.40

```
int f(node n);
f(4);
```

Danger: Accidentally passing an int to a function that expects a Node causing silent conversion. This potentially causes errors that aren't caught (not signaled by the compiler).

Definition 0.14: Explicit

The solution to it is to disable the implicit conversion. In other words, make the constructor explicit by using the **explicit** keyword.

Example 0.41

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

After adding the keyword `explicit`, we have

```
Node n{4};  // valid
Node n = 4;  // Error
f(4);       // Error
f(Node{4});  // valid
```

Result 0.13

It is a good practice that when having a single-arg copy constructor, label it explicit to avoid implicit conversions.

0.11.3 Destructors

Definition 0.15

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

Theory 0.17: Three Steps of Destructors

Class always comes with a default destructor, which invokes the destructors of all fields that are objects.

When an object is destroyed, there are three steps:

1. Destructor body runs;
2. Fields' destructors run in reverse declaration order (for fields that are objects);
3. Space deallocated.

When do we need to write our own destructor?

Code 0.20

Consider

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

if `np` goes out of scope, the pointer is reclaimed, the list is thus leaked. If we say delete np, the first node is reclaimed (`*np`'s destructor is called), which does not do anything (because it is not an object).

The solution to it is writing our own destructor:

Code 0.21

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

Result 0.14

The Node class is responsible for freeing memory.

Theory 0.18

A class has only one destructor method. Cannot have multiple destructors in one class.

Discovery 0.17

Recall the `exit()` function. It terminates the program immediately, so none of the destructors run.

[†]

0.11.4 Copy Assignment Operator

Definition 0.16

A **copy assignment operator** is not a constructor, it overrides the `=` assignment operator.

Code 0.22

```
Student s1{60, 70, 80};     // calls the constructor
Studetn s2 = s2;           // calls the copy constructor
Student s3;               // calls the constructor
s3 = s1;                 // copy but not construct the object,
                        // this is called copy assignment.
```

Discovery 0.18

Copy assignment operator starts with a fully constructed object from earlier.

[†]Not deallocating the memory allocated is considered an incorrect program in the course (might be legit in reality).

0.12 Lecture 10

Lecture 10 - Tuesday, Jun 11

The goal of **copy assignment** is to replace the data in the object with a copy of the data from the other object, without introducing memory leaks. The compiler provides a default copy assignment operator that we can replace.

Code 0.23

```
struct Node {
    int data;
    Node *next;
    ...
    Node &operator=(const Node &other) {
        data = other.data;
        next = other.next ? new Node (*other.next) : nullptr;
                                  ^ copy constructor
        return *this.
    }
};
```

But we might have a memory leak here because the Node might have its own next before assignment. Therefore, we need to delete it first.

Code 0.24

```
Node &operator=(const Node &other) {
    data = other.data;
    delete next;        // free any existing node list
                        // we are pointing to
    next = other.next ? new Node (*other.next) : nullptr;
                              ^ copy constructor
    return *this.
}
```

We couls also have self assignmnet:

Example 0.42: Self-assignment

```
Node n ...
...
n = n;       // self-assignment
```

To fix the issue that we are deleting the next before assigning we encounter in self-assignment, we simply just need to insert another line of code:

```
Node &operator=(const Node &other) {
    if (this == &other) return *this;    // protects against self-assignment
    data = other.data;
    next = other.next ? new Node (*other.next) : nullptr;
                              ^ copy constructor
    return *this.
}
```

Discovery 0.19

Instead of doing `this == &other`, would `*this == other` work?
Answer: No, because you could have two with the same data that would be equal, but this is not a self-assignment. Secondly, you do not get `==` for free, you have to code it yourself.

Result 0.15

A better implementation is (no problem):

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

This version insures that if `new` fails, we do not change our object (cuz we haven't touched it yet). In particular, if `new` fails, we exit the function immediately (more about `except` later in the course).

0.12.1 Copy-and-Swap Idiom

Code 0.25: Alternative using copy and swap

```
import <utility ;

struct Node {
    ...
    void swap(node &other) {
        std::swap(data, other.data);
        std::swap(next, other.next);
    }
    Node &operator=(const Node &other) {
        Node temp = other;  // step1, copy via copy ctor
        swap(temp);         // step2, new temp has old values
        return *this;
    }
}
```

1. Step1: deep copy other to construct temp
2. Step2: after step 2, temp has the old values from our object. When operator= function exists, temp will be destroyed taking our old data with it.

Theory 0.19

For copy-and-swap, the self-assignment test is not needed, but it would be an optimization.

Discovery 0.20

The reason why we return `*this` is that this allows us to do cascading assignment:

```
n1 = n2 = n3 = n4 = n5;     // makes n1 .. n4 the same as n5
```

0.12.2 Move Constructor and Move Assignment Operator

Consider the program that looks like the following:

Example 0.43

```
    Node oddsOrEvens() {     \\ returns a Node object by value
        Node odds{1, new Node{3, new Node{5, nullptr}}};
        Node evens{2, new Node{4, new Node{6, nullptr}}};
        char c;
        cin >> c;
        return c == "0" ? evens : odds;
    }
    Node n = oddsOrEvens();      // copy constructor
    Node m;
    m = oddsOrEvens();           // copy assignment operator
```

Copy constructor/ copy assignment operator are used to copy the data from oddsOrEvens into n or m.

Result 0.16

We can actually steal the data from the temporary odds/ evens objects instead of copying them. The **move constructor** and **move assignment operator** allow us to do it.

Code 0.26: Move Constructor

```
        struct Node {
            ...
            Node(Node &&other)       // rvalue reference (a reference
                                     // to a temporary object)
            : data{other.data}, next{other.next} {
                other.next = nullptr;
            }
        }
```

The temporary Node other will still have its destructor called, but since its next field is now nullptr, the destructor doesn't free anything. The temp object can be destroyed simply and efficiently. No copying of nodes was needed.

Code 0.27: Move Assignment Operator

```
        struct Node {
            ...
            Node &operator=(Node &&other) {
                swap(other);     // same swap used for the copy assign op.
                return *this;    // with the copy-and-swap idom
            }
        }
```

The temporary Node other will still have its destructor called, but since its next field is now nullptr, the destructor doesn't free anything. The temp object can be destroyed simply and efficiently. No copying of nodes was needed.

Theory 0.20: How is move different from copy

1. Other parameter is not const because we need to steal from it and make it "empty". Or we can swap our data into it (for assignment).
2. Other is an rvalue reference type parameter.
3. It is simpler to write a move operator than a copy operator.

Theory 0.21

If you dont define move constructor/ assignment operator, the compiler will use the copy operations when the argument is a temporary object.

Now

```
Node n = oddsOrEvens();
```

```
Node m;
m = oddsOrEvens();
```

uses the move constructor (we didn't have to change the function or the code) and move assignment operator respsectively.

Discovery 0.21

Bottom line, returning objects by value from a function is often very fast and efficient.

Result 0.17: Rule of Five (Big Five)

In summary, we can now state the Rule of Five (or the Big Five): If you need to write any one of the following, you usually need to write all five:

1. destructor;
2. copy constructor;
3. copy assignment operator;
4. move constructor;
5. move assignment operator.

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

0.13 Tutorial 4

CS246 - 2024S/T

0.14 Lecture 11

Lecture 11 - Thursday, Jun 13

0.14.1 Copy/ Move Elision

Code 0.28

```
Vec makeAVec() {     // return by value
    return {0, 0};   // uses the default ctor to create a Vec
}
Vec v = makeAVec(); // copy or move ctor depending on
                    // whether we have a move ctor
```

Theory 0.22

In certain cases, the compiler is required to skip calling the calling the copy/ move constructors. Here, `makeAVec()` writes its result directly into the space occupied by `v` in the caller rather than copying it later.

Here is a slightly more complex example:

Code 0.29

```
void doSomething(Vec v) {
    ...
}
doSomething(makeAVec());
```

Theory 0.23

What is happening here is that the result of `makeAVec()` is written directly into the parameter `v` of `doSomething()`. There is no copy or move.

Result 0.18

Elision happens even if dropping the constructor calls would change the behaviour of the program. For an instance, if the constructors print something while called, if an elision happens, nothing will be printed. Note: you are not expected to know when it happends, rmbr that it could happen.

0.14.2 Member Operators

Notice operator= was a member function and not a stand-alone function. Previous operators we've written were stand-alone functions.

Definition 0.17: Stand-alone Function

Recall that we have

```
struct Vec {
    int x, y;
}
Vec operator+(const Vec &v1, const Vec &v2) {
    // stand-alone function, doesnt have access to ‘this’
}
```

To write it as a member function, we have

```
struct Vec {
    int x, y;

    Vec operator+(const Vec &other) {
        return {x + other.x, y + other.y};
        // doesnt have to say this->x, but we can
    }
    Vec operator*(const int k) {
        return {x * k, y * k};
    }
}
```

However, notice that `operator*` cannot be a member function because we cannot implement the case when we have a vector to the right side of the integer, (i.e, k * v), therefore, we need to implement the `operator*` as a stand-alone function.

Example 0.44: Implement +=

```
Vec &operator+=(Vec &v1, const Vec &v2) {
    v1.x += v2.x;   // normal int += operator, no recursion.
    v1.y += v2.y;
    return v1;
}
```

This is an example of a stand-alone function, but we can indeed change this into a member function. Now we can write the regular operator+ version in terms of the += operator:

Code 0.30

```
Vec operator+(const Vec &v1, const Vec &v2) {
    Vec temp(v1);        // use copy constructor
    return temp += 2;    // use above operator+= to modify/ return temp
}
```

Discovery 0.22

Suppose we wrote our ostream operator« as a member function as following

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

Notice that in this case, we would then need to write

```
v << cout;
```

which is a bit awkward.

Result 0.19

herefore, as a result, define `operator<<` and `operator>>` as stand-alone.

Discovery 0.23

Also notice that some operators must be members:

1. `operator=` ;
2. `operator[]` ;
3. `operator->` ;
4. `operator()` .

```
Vec::operator=(...)      // if member function defined
                         // outside the struct
```

0.14.3 Arrays of Objects

Let's say that Vec is defined as:

Code 0.31

```
struct Vec {
    int x, y;
    Vec(int x, int y) : x{x}, y{y} {}   // replaces the default
                                        // no-arg constructor
};
Vec v;          // won't work
Vec v{5, 4};    // will work
```

Now we would have:

```
Vec vectors[15];         // error
Vec *vp = new Vec[15]    // error
```

The reason why the above two lines of code don't work is that each of the vector in the array of vector(s) needs to be constructed. The compiler will try to default construct each of them, which will fail (default constructor is replaced).

Definition 0.18: How to solve

To fix it, we have several options:

1. Create a default constructor for Vec:
   (a) Create a new constructor; or
   (b) Make x and y parameters have default values.
2. For stack arrays, we can also do:

   ```
   Vec vectors[15] = {Vec{0, 1}, ..., Vec{0, 15}};
   ```

3. For heap arrays, we can create an array of pointers:

   ```
   Vec **vp = new Vec*[15];    // fine, but the pointers are uninitialized
   vp[0] = new Vec{0, 0};
   ```

   Do not forget to delete each of the vectors in the array and the array itself at the end.

Theory 0.24

Btw, you can initialize arrays of built-in types easily:

```
Vec **vp = new Vec*[15] = {nullptr};
```

The compiler initializes the whole array with `\0`.

0.14.4 Const Objects

Example 0.45

```
struct Student {
    ...
    float grade() {
        ...
    }
}
int f(const Student &s) {   // const objects arise often,
                            // especially as parameters
}
```

`const` objects cannot be modified, the compiler guarantees this. Additionally, we cannot call the methods of `s` because the compiler does not know whether those methods modify our objects, so it does not compile.

Theory 0.25

Therefore, in order to be able to call the `grade()` function, we need to write this instead:

```
struct Student {
    ...
    float grade() const {
        ...
    }
}
```

If defined outside (the struct), we need to include the `const` keyword as well.

The compiler checks that `const` methods don't modify fields. In our case, `grade` is not allowed to modify any field.

Result 0.20: Const with const

If you have a `const` object, like `s` as above, you can only call const methods on it. In other words, only `const` methods may be called on `const` objects. On the other hand, if you have a non-`const` object, then anything could be called (even the `const` ones).

0.15 Lecture 12

Lecture 12 - Tuesday, Jun 18

0.15.1 Logical vs. Physical Constness

Code 0.32

```
struct Student {
    mutable int numMethodCalls = 0;

    float grade() const {
        ++numMethodCalls;
        return ...
    }
}
```

Theory 0.26

The mutable numMethodCalls fields affects only the **physical** constness of the object, not its **logical** constness. Mutable fields can be changed even if the object is const. Use `mutable` to indicate that a field does not contribute to the logical constness of the object.

Definition 0.19: Physical vs. Logical Constness

1. Physical Constness: whether the actual bits/ bytes that make up the object have changed;
2. Logical Constness: whether the updated object should logically be regarded as different after the update.

0.15.2 Comparing Object

Code 0.33

This is how we compare in C, where s1 and s2 are char pointers.

```
strcmp(s1, s2) returns      <0 if s1 < s2
                            =0 if s1 = s2
                            >0 if s1 > s2
```

In C++, we can use the three way comparison `operator<=>` (aka. the spaceship operator):

Code 0.34

```
import <utility>;
```

```
string s1, s2;
std::strong_ordering result = s1 <=> s2;     // one comparison
if (result < 0) cout << "less";
else if (result == 0) cout << "equal";
else cout << "greater";
```

Theory 0.27

`strong_ordering` actually returns four possible values:

1. less;
2. equal/ equivalent;
3. greater.

Normally, we could write

```
if (result == std::strong_ordering::less)    // same as result < 0
```

Definition 0.20: Auto

There is a shortcut:

```
        auto result = s1 <=> s2;
```

This uses `auto` keyward, so the compiler automatically deduces the type (replaces `std::strong_ordering`).

0.15.3 Defining <=> for our own Classes

Code 0.35

```
struct Vec {
    int x, y;

    auto operator<=>(const Vec &other) const {
        auto n = x <=> other.x;
        if (n != 0) return n;
        return y <=> other.y;
    }
}
Vec v1{1, 0};
Vec v2{1, 3};
v1 <=> v2;       // strong_ordering::less
```

```
// by implementing <=>, we get all other comparison operators for free
v1 <= v2     =>     (v1 <=> v2) <= 0
v1 == v2     =>     (v1 <=> v2) == 0
v1 > v2      =>     (v1 <=> v2) > 0
```

Discovery 0.24

You can sometimes even get the <=> operator for free:

```
    auto operator<=>(const Vec &other) const = default;
```

This just does the lexicographic comparison on the fields of `Vec`, which is equivalent to what we just wrote.

What about `Node`? What does it mean for a `Node` object to be equal to, less than, or greater than the other one?

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

Algorithm 0.1

1. Step 1: Compare data, if not equal, the one with the less value should be less
2. Step 2: Compare next fields, but we need to consider four cases:
    (a) Both next == nullptr, we return equal;
    (b) node1.next == nullptr and node2.next != nullptr, we return node1 less than node2;
    (c) node1.next != nullptr and node2.next == nullptr, we return node1 greater than node2;
    (d) node1.next != nullptr and node2.next != nullptr, we advance to the next node and repeat.

Code 0.36

```
struct Node {
    auto operator<=>(const Node &other) const {
        // step 1
        auto n = data <=> other.data;
        if (n != 0) return n;
        // step 2
        if (!next && !other.next) return n;                     // case (a)
        if (!next) return std::strong_ordering::less;           // case (b)
        if (!other.next) return std::strong_ordering::greater;  // case (c)
        return *next <=> *other.next;                    // case (d)
                     //^ recursive call to next nodes in both lists
    }
}
```

0.15.4 Invariants and Encapsulation

Consider

Example 0.46

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

Suppose our user does this:

```
Node n1 {1, new Node {11, nullptr}};
Node n2 {2, nullptr};
Node n3 {3, &n2};        // n3's dtor will try to delete n2,
                         // undefined bahaviour
```

The Node class relies on the assumption that next is either `nullptr` or was allocated by `new` (on the heap).

Definition 0.21: Invariant

The above assumption is an example of an **invariant**, which is a statement that must hold true (upon which Node relies).

However, we cannot gaurantee this invariant, we cannot trust the client to use Node properly, because we are exposing the next field, allowing clients to manipulate it directly.

It's hard to reason about programs if you cannot rely on invariants.

Theory 0.28: Encapsulation

To force invariants, we introduce **encapsulation** – we want clients to treat our objects as black boxes (capsules). In this way, it creates abstractions in which implementation details are hidden or sealed away such that clients can only manipulate them in provided methods.

The following is an example:

Example 0.47

```
struct Vec {
    private:    // whatever follows is private,
                // which cannot be accessed outside Vec
        int x, y;
    public:
        Vec(int x, int y) : x{x}, y{y} {}
        Vec operator+(...) {}
};
```

Theory 0.29

By default access to members of a C++ class declared with the keyword `class` is private.

Result 0.21

The only difference between `struct` and `class` is the default visibility.

Let's fix out linked list example by introducing encapsulation: The key is to create a wrapper class `List` that has exclusive access to the underlying Node objects.

Example 0.48

Code 0.37: list.cc

```
export class List {
    struct Node;    // forward declaration, private nested class
                    // only accessible inside List
    Node *theList = nullptr;
    public:
        void addToFront(int data);
        int ith(int i) const;
        ~List();
};
```

Code 0.38: list-impl.cc

```
struct List::Node {      // nested class
    int data;
    Node *next;
    Node(int data, Node *next) : data{data}, next{next} {}
    ~Node() {delete next};
};

List::~List() {delete theList;}
List::addToFront(int data) {theList = new Node{data, theList};}
int List::ith(int i) const {
    Node *cur = theList;
    for (int j = 0; j < i; j++, cur = cur->next) {
        return cur->data;
    }
}
// only List can manipulate Node objects,
// so we can gaurantee the invariant that
// next is always a nullptr or alloced by new.
```

0.16 Lecture 13

Lecture 13 - Thursday, Jun 20

0.16.1 Iterator

Definition 0.22: Iterator

Iterator allows us to easily and efficiently iterate the items in the list. Iterations is a common pattern in programming. C++ has a standard way that iteration works. The general idea is to create a new *Iterator* class that gives us access to the nodes.

1. It is an abstraction of a pointer;
2. It let us walk the list without exposing the actual pointers.

It works like this

Theory 0.30: How Iterator Works

1. The List class has a method called `begin()` that creates and returns an iterator object that points to the first node.
2. Using this iterator object, you can access the data in the node,

3. You can also advance the iterator so that it points to the next node.
4. The List class also has en `end()` method that creates and returns an iterator object pointing to one position past the last node.
5. When your iterator equals the iterator returned by `end()`, you reach the end of the list.

Code 0.39

Here is an example:

```
int main() {
    List list;
    lst.addToFront(1);
                  (2);
                  (3);

    List::Iterator it = lst.begin();     // create iterator object
    while ( it != lst.end() ) {          // check if we've reached the end
        cout << *it << endl;             // access the date using iterator
        ++it;                            // advance the iterator
    }

    // for loop version:
    for ( auto it = lst.begin(); it != lst.end(); ++it ) {
        cout << *it << endl;
    }

    // range-based for loop:
    for (int n : lst) {              // shortcut syntax reads
        cout << n << endl;           // "for each int n in lst"
    }

    // mutate lst items or avoid copying each item
    for (int &n : lst) {         // using reference to avoid copying
        n *= 2;                  // allow us to mutate the data too
    }                            // our list now contains 6 4 2
}
```

Code 0.40

ere is the code to make all of the above work:

```
class List {
    struct Node;
    Node *theList = nullptr;
    ...
    public:
        class Iterator {          // nested class
            Node *p;              // points to the current node;
            public:
                explicit Iterator(Node *p) : p{p} {}     // ctor
                int &operator*() const {         // access data
                    return p->data;
                }
                Iterator &operator++() {         // advance iterator
                    p = p->next;
                    return *this;
                }
                bool operator==(Iterator &other) const {
                    return p == other.p;
                }   // by implementing ==, we get != for free
        };  // class Iterator

    Iterator begin() const {
        return Iterator{theList};
    }
    Iterator end() const {
        return Iterator{nullptr};
    }
    // other list methods, like Big5
}   // class List;
```

Discovery 0.25

Notice that the constructor of the Iterator is still public, so List client's code can still create Iterator objects directly:

```
List::Iterator it = List::Iterator{nullptr};
```

This violates encapsulation. Only `begin()` and `end()` should be allowed to create Iterators. If we make the Iterator constructor private, then the client's code cannot do this, but neither can List.

Theory 0.31: Friend

To give the List class privileged access to the Iterator class, we make the List a **friend**.

Example 0.49

```
class List {
    ..;
    public:
        class Iterator {
            private:              // optional, private by default
                Node *p;
                explicit Iterator(Node *p) p: {p} {};    // private
            friend class List;  // gives List access to the
                                // private members of Iterator
                                // this line can be put anywhere
        };
}
```

Now List can create Iterator objects, and the client code can only create Iterator by calling `begin()` and `end()`.

Result 0.22

The Iterator class is saying that List is a **friend** and can be trusted to access the private members correctly.

0.16.2 Nested Class

Example 0.50

```
--- class List
|
|       --- class Node
|       |
|       -----
|
|       --- class Iterator
|       |
|       -----
-----
```

These classes are "nested" (defined) inside the List class. Their names are `List::Node` and `List::Iterator`.

Theory 0.32

At runtime, you create instances od these classes (objects), the objects can be related in many different ways depending on how you use the classes.

Remark: Do not be confused by nested classes and think that creating a List object automatically creates a Node object and an Iterator object.

Theory 0.33

Nested classes allow you to reuse class names (i.e. Iterator). They also allow you to create "local" classes that are private and used internally as an implementation detail (i.e. Node).

Theory 0.34

An instance of a nested class has access to the private members (fields), and methods of its "wrapping" outer class as long as it has a way to get to an outer object (e.g. through a pointer/ reference/ object).

0.17 Lecture 14

Lecture 14 - Tuesday, Jun 25

Theory 0.35

The range-based for loop is available for any class with

1. methods `begin()` and `end()` that produce iterators;
2. the iterator class must support the `*`, `++`, and `!=` operators.

The compiler "re-writes" the for loop as a traditional for loop, with the `begin()` and `end()`, and the `*`, `++`, and `!=` operators.

0.17.1 Accessors and Mutators

Code 0.41

```
struct Vec {
    private:
        int x, y;
    public:
        int getX() const {return x;};          // accessor
        void setX(const int new_x) {x = new_x}; // mutator
};
```

What about `operator<<`?
It needs access to x and y, but it cannot be a member function.

1. It could use the accessor methods to get x and y;
2. We could make it a friend function.

Example 0.51

```
struct Vec {
    ...
    Friend ostream &operator<<(ostream &out, const Vec &v);
};

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

0.17.2 Equality for Our List Class

If we want to check if two Lists are equal, doing a length check first is quicker and more efficient.

Code 0.42

```
class List {
    Node *theList;
    public:
        auto operator<=>(const List &other) const {
            if (!theList && !other.theList) return strong-ordering::equal;
            if (!theList) return strong-ordering::less;
            if (!other.theList) return strong-ordering::greater;
            return *theList <=> other.theList;      // Node <=> operator
        }
        bool operator==(const List &other) const {
            if (length != other.length) return false;
            return (*this <=> other) == 0;  // implemented ito <=> above
        }
}
```

Discovery 0.26

You can implement both `==` and `!=` operators if you want a custom `!=` operator, but obviously you don't have to.

0.17.3 System Modelling

Definition 0.23: UML

We will use the **UML (Unified Modelling Language)** diagram for showing the structure of a class and the relationships to other classes. This diagram is called a **class diagram**.

Example 0.52: Model a class:

```
|-----------------------|
|         Vec           |        --> class name
|-----------------------|
| -x: integer           |        --> fields:
| -y: integer           |            - for private; + for public; # for protected
|-----------------------|
| +getX(): integer      |        --> methods
| +getY(): integer      |
| +setX(newX: integer)  |
| +setY(newY: integer)  |
|-----------------------|
```

0.17.4 Relationship: Compositon of Classes

Definition 0.24: Composition

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

Code 0.43

Recall

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

A Basis is composed of 2 Vec's. They are **a part** of a Basis, and that is their only purpose.

Result 0.23

Relationship: a Basis "owns" a Vec (actually it owns 2 of them).

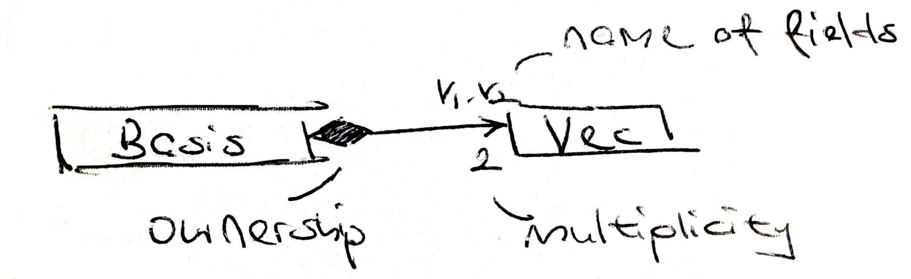
Definition 0.25: "Own a"

If A **owns a** B, then typically:

1. B has no identity outside A (no independent existence);

2. If A is destroyed, B is also destroyed;
3. If A is copied, B is also copied.

Example 0.53: UML - Owns a



0.17.5 Relationship: Aggregation

Definition 0.26: Aggregation, "Has a"

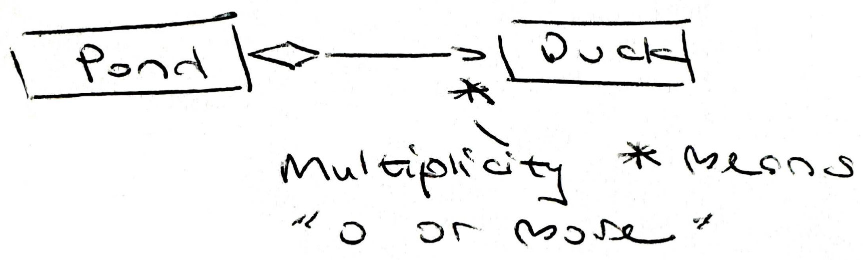
If A **has a** B, then typically:

1. B exists apart from its association with A;
2. If A is destroyed, B lives on;
3. If A is copied, B is not (shallow copy). Copies of A store the same B.

Example 0.54

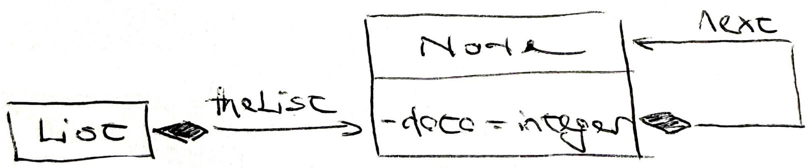
Parts in a catalog, ducks in a pond.

Example 0.55: UML - Has a



Does using a pointer always mean non-ownership (i.e. aggregation)?

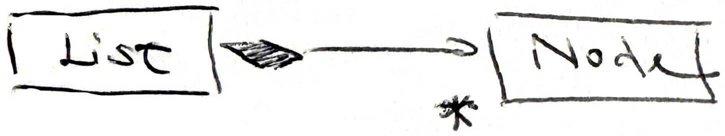
Result 0.24



A Node owns the Nodes that follows it. An implementation of the Big 5 is a good sign of ownership. These ownerships are implemented via pointers.

Discovery 0.27

Another way to view List and Node:



The diagram implies that List has a collection of Nodes and it responsible for all of them.

0.17.6 Relationship: Specialization (or Inheritance)

Suppose you want to track your collection of books:

Code 0.44

```
class Book {
    string title, author;
    int length;     // number of pages
    public:
        Book(...);
        ...
};
class Text {                        class Comic {
    string title, author;               string title, author;
    int length;                         int length;
    string topic;                       string hero;
    public:                             public:
        Text(...);                          Comic(...);
        ...                                 ...
    }                                       }
```

Rather than using techniques like union or `void *`, notice that texts and comics are kinds of books. They are Books with extra features.

Theory 0.36

We can model these using C++ **inheritance**.

Definition 0.27: Base Class and Derived Class

```
class Book {          // base class (or superclass)
    ...
};
class Text : public Book {  // derived class (subclass)
    string topic;           // only the new fields
    public Text(...)
};
class Comic : public Book { // derived class (subclass)
    string hero;            // only the new fields
    public Comic(...)
};
```

Theory 0.37

Derived classes inherit the fields and methods from the base class. So Text and Comic get title, author, and length fields automatically.

Result 0.25: Layout in Memory

```
  Book       Text       Comic
|-------| |-------| |-------|
|-title-| |-title-| |-title-|
|-author| |-author| |-author|
|-length| |-length| |-length|
|-------| |-topic-| |--hero-|   --> extra fields are added
          |-------| |-------|       after base fields
```

Also, any methods that can be called on a Book can also be called on Text and Comic. Anything `private` in Book cannot be seen outside of Book (except any friends of Book). Text and Comic cannot access title, author, and length.

Code 0.45

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

This will not work because

1. title etc. are private and not accessible by Text. The MIL only lets you initialize your own fields;
2. Recall object creation steps (0.11):
   (a) Space is allocated;
   (b) The superclass part is constructed; (new to this case)
   (c) Fields are constructed;
   (d) Constructor body runs.

How do we initialize a Text object using the constructor?
Solution:

Code 0.46

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

0.18 Lecture 15

Lecture 15 - Thursday, Jun 27

0.18.1 Protected Visibility

If you want to give subclasses access to members but not any code outside the class hierarchy, you can use protected visibility

Code 0.47

```
class Book {
    protected:      // accessible to subclasses and sub-subclasses,
                    // but no one else
        string title, author;
        int length;
};
class Text : public Book {
    public:
        void addAuthor{const string &newAuthor} {
            author += ", " + newAuthor; // author field is visible to Text
        }
    ...
};
```

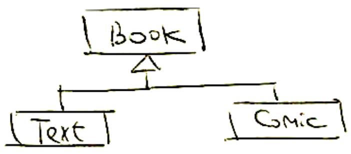
Theory 0.38

If we want to further protect our base class fields, we can use a protected mutator method:

Code 0.48

```
class Book {
    ...             // fields are private
    protected:
        void setAuthor(const string &newAuthor);    // call this method to
                                                    // change author
    publilc:
        string getAuthor() const;                   // accessor method
};
```

Result 0.26: UML Diagram



The arrow shows the “is a” relationship of inheritance.

0.18.2 Virtual Methods

Consider adding an `isHeavy()` method to Book, we will define it as follows:

```
-ordinary book:     heavy means > 200 pages;
-text book:         heavy means > 500 pages;
-comic book:        heavy means > 30  pages;
```

Code 0.49

```
class Book {
    ...
    bool isHeavy() const {return length > 200;}
};
class Comic : public Book {
    ...
    bool isHeavy() const {return length > 30;}  // assume we have access
}
etc.

Book b{"A small book", "Papa Smurf", 50};               // not heavy
Comic c{"A big comic", "Mr. Comic", 40, "Antman"};     // heavy

Book b = Comic{..., 40, ...};
cout b.isHeavy();        // Book.isHeavy() runs
```

Definition 0.28: Sliced

The Comic Object is **sliced** – the hero field is chopped off and the Comic is coerced into a Book.

Theory 0.39

When accessing objects through pointers, **slicing** doesn't happen.

Example 0.56

```
Comic c{..., 40, ...};          Comic *cp = &c;
Book *bp = &c;                  Book &br = c;
cout << c.isHeavy()      // true
     << cp->isHeavy()    // true
     << bp->isHeavy()    // false
     << br.isHeavy()     // false
```

Discovery 0.28

ur Comic Object is acting like a Book when we use a Book pointer or reference. This behaviour of the object depends on what type of pointer (or reference) you access it through.

How do we make a Comic act like a Comic when pointed by a Book pointer?
Answer: Use `virtual` method.

Theory 0.40

Virtual Method – choose which class' method to run based on the *actual* type of the object at runtime.

Code 0.50: Virtual & Override

```
class Book {
    string title, author;
    protected:
        int length;
    public:
        ...
        virtual bool isHeavy() const {return length > 200;}
}
class Comic : public Book {
    ...
    public:
        bool isHeavy() const override {return length > 30;}
}
```

The `override` tells the compiler that we are explicitly overriding the method. It is optional.

Result 0.27

When having the keyword `override`, the compiler will check if there is actually such a `virtual` method in the base class. If there isn't, you get a compile error.

Now

Example 0.57

```
Comic c{..., 40, ...};          Comic *cp = &c;
Book *bp = &c;                  Book &br = c;
cout << c.isHeavy()     // true
     << cp->isHeavy()   // true
```

```
        << bp->isHeavy()    // true, becuz Comic version isHeavy() runs
        << br.isHeavy()     // true, same reson
```

However, it is important to note that

```
Book b = c;
```

is still slicing. We copy the Comic fields into a Book object and run the Book version of `isHeavy()`.
Now we can have a collection of different types of Books:

Code 0.51

```
Book *myBooks[20];        // array of 20 Book*
...
fir (int i = 0; i < 20; ++i) {
    cout << myBooks[i]->isHeavy() << endl;
}
```

The correct version of `isHeavy()` will run for each one.

Definition 0.29: Polymorphism

Accomodating multiple types (Book, Text, Comic) under one abstraction (Book) is called **polymorphism** ("multiple forms").

Discovery 0.29

This is why a function `void f(istream &in)` can be passed an `ifstream` – `ifstream` is a subclass of `istream`.

0.18.3 Destructor Revisited

Code 0.52

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

X *myX = new Y{10, 20};
...
delete myX;
```

This code leaks memory because `delete myX` calls `~X()` but not `~Y()`, so only x is freed, not y.

To solve the above issue, we need to make our destructor `virtual`, similar to how we made `isHeavy()` `virtual`.

Code 0.53: Virtual Destructor

```
virtual ~X() {delete [] x;}
```

Adding the keyword `virtual` causes the correct destructor to be called based on the actual type of the object.

Theory 0.41: Object destruction follows these steps:

1. Destructor body runs;
2. Fields destructor run in reversed declaration order;
3. Repeat step 1-3 for the base class; (new!)
4. Space is deallocated.

   The new step supports subclasses.

Result 0.28: ALWAYS make the destructor virtual

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

Definition 0.30: Final

If a class is not meant to be subclassed, you can declare it `final`.

Example 0.58

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

`final` makes sure that Y cannot be subclassed (so it is at the bottom of the hierarchy).

0.18.4 Pure Virtual Methods and Abstract Classes

Code 0.54

```
class Student {
    protected:
        int numCourses;
    public:
        virtual int fees() const = 0;    // signals that fees() method
                                         // has no implementation
                                         // aka. pure virtual method
                                         // makes Student abstract
        ...
};
class Regular : public Student {
    public:
        int fees() const override;  // computes fees for regular students
};
class Coop : public Student {
    public:
        int fees() const override;  // computes fees for Coop students
};
```

A class with one or more pure virtual methods cannot be instantiated:

```
Student s;  // error! Student cannot be instantiated
```

Definition 0.31: Abstract Class

A class that can't be instantiated is called an **abstract class**. Subclasses (of it) are also abstract unless they implement ALL pure virtual methods.

0.19 Lecture 16

Lecture 16 - Tuesday, Jul 2

Example 0.59

An abstract class is generally used to provide a common set of fields and/ or methods for a series of subclasses.

Definition 0.32: Concrete Class

Non-abstract classes are called **concrete**.

Result 0.29

In UML we represent virtual and pure virtual methods using *italics*. We represent abstract classes by *italicizing* the class name. We can also use asterisks to show this when *italics* are hard to do (i.e. when write on paper).

0.19.1 Templates

Recall the List class we were building:

Code 0.55

```
class List {
    struct Node {
        int data;        <-- data field defines what type of data we store
        Node *next;
    };
    Node *theList;
};
```

Theory 0.42

If we want to store something other than an int, we can turn this List class into a template class. A template class is parameterized by one or more types.

Example 0.60

```
template <typename T>    // <-- new line of code making List
                         //      a template class
class List {
    struct Node {
        T data;
        Node *next;
    };
    ...
    public:
        class Iterator {
            Node *p;
            ...
            public :
                T &operator*() const {...}
        };
        void addToFront(const T &data) {...}
```

```
            T &ith(int i) {...}
            ...
    }
```

Possible client code:

```
    List<int> ints;          // creates a List object with T = int
    List<string> strings;    // creates a List object with T = string
    List<List<int>> loloi;   // creates a List object with T = List<int>

    ints.addToFront(5);
    strings.addToFront("hello world");
    loloi.addToFront(ints);

    for (List<int>::Iterator it = ints.begin(); it != ints.end(); ++it) {
        cout << *it << endl;
    }
    for (auto n : ints) {
        cout << n << endl;
    }
```

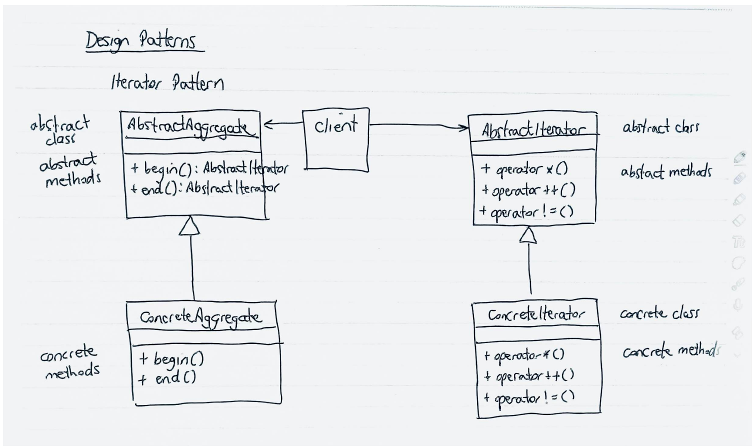
Theory 0.43

The compiler specializes the template into actual code as a source-level transformation and then compiles the resulting code as usual.

0.19.2 The Standard Template Library (STL)

Definition 0.33: STL

Has a large collection of useful templates. One example would be `vector`.



Example 0.61: Vector

Dynamic length arrays:

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

vector<int> v1;          // vector of ints. Creates an empty vector
vector<int> v2{4, 5}     // creates a vector with 4 and 5 as its elements
vector<int> v2(4, 5)     // creates a vector with 4 elements of value 5
vector.emplace_back(1); // addes the value 1 as an element
vector.push_back(2);     // addes the value 2 as an element

vector v4{4, 5, 6, 7};  // compiler deduces the type int from
                         // the initialization list

for (int i = 0; i < v4.size(); ++i) {   // method size() is num of ele
    cout << v4[i] << endl;
}
for (vector<int>::iterator it = v4.begin(); it != v4.end(); ++it) {
    cout << *it << endl;
    // lowercase i for iterator (it's a std template lib)
}
for (int n : v) {
    cout << n << endl;
}

// reverse iterator:
for (vector<int>::reverse_iterator it = v4.rbegin();
     it != v4.rend(); ++it) {
    cout << *it << endl;
}
// more useful methods:
v4.front();      // first element
v4.back();       // last element
v4.pop_back();   // removes the last element. void method
```

Discovery 0.30

Here is a good question about `emplace_back`: link.

Theory 0.44

Vectors are guaranteed to be implemented internally as arrays.

Result 0.30

Use vectors whenever you need a dynamic length array.

0.19.3 Removing Elements From a Vector

Code 0.56

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

The above code is wrong. Consider a vector:

```
v:   1 | 5 | 5 | 2
```

In this case, after `erase()`, all the subsequent element are pushed forward by one spot, while our iterator stays at `v[1]`, so after advancing the iterator, we miss removing the second 5.

Result 0.31

After an insertion or an erase, all iterators pointing after the point of insertion/ erase are considered invalid and must be refreshed.

Example 0.62

So here is the correct implementation:

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

Note that we cannot use range-base for loop because in that case, the iterator is "hidden" from us, and we cannot use the one returned by erase().

0.19.4 Design Patterns - Iterator Pattern

Example 0.63

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

```
            virtual ~AbstractIterator();
    }
```

0.20 Lecture 17

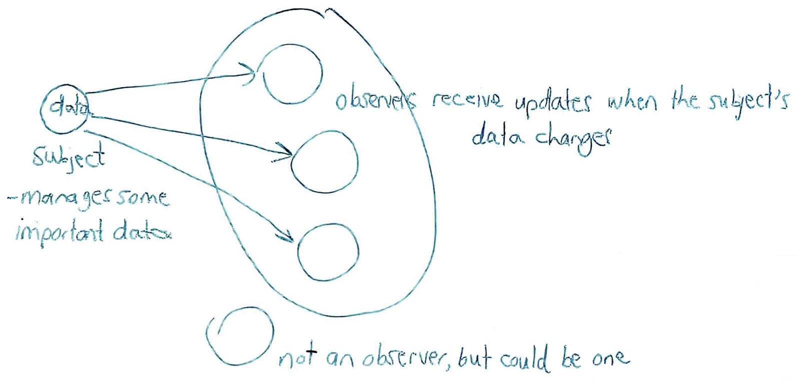
Lecture 17 - Thursday, Jul 4

0.20.1 Observer Pattern

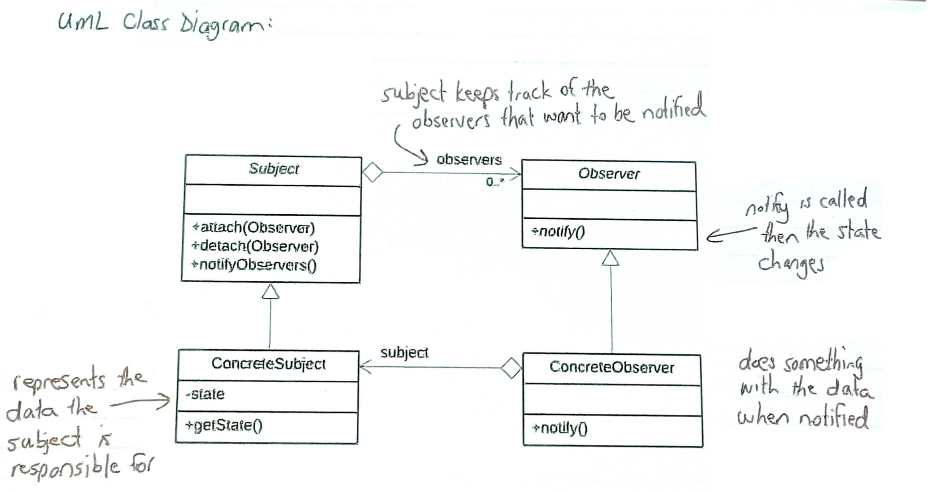
Definition 0.34

An **observer pattern** is used to implement a publish/ subscribe model.

1. One class generates/ updates data: subject/ publisher;
2. Many classes receive data and react to it: observer/ subscriber.


data
Subject
-manages some important data
observers receive updates when the subject's data changes
not an observer, but could be one

Result 0.32

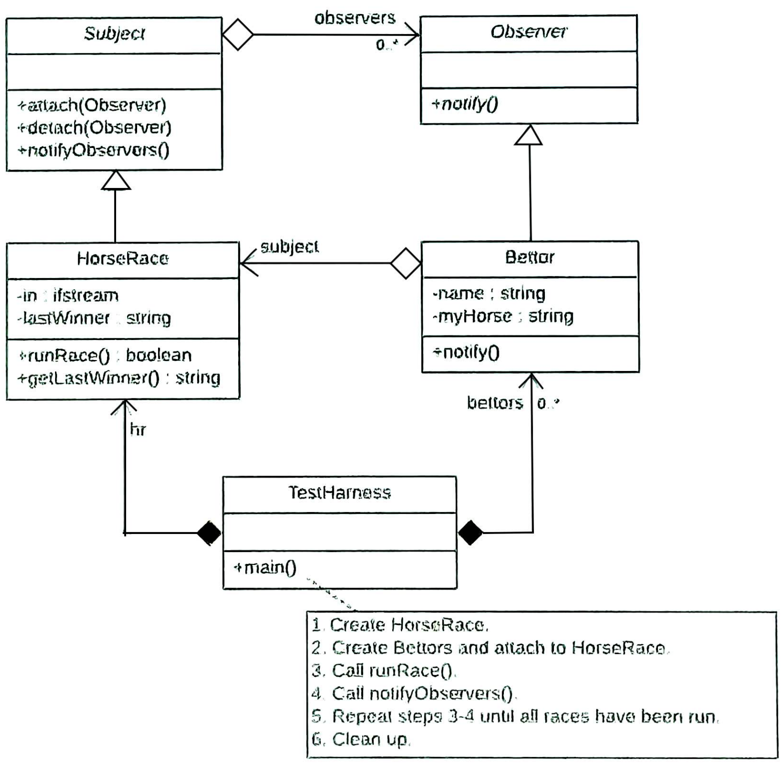

UML Class Diagram:
subject keeps track of the observers that want to be notified
observers
0..*
Subject
+attach(Observer)
+detach(Observer)
+notifyObservers()
Observer
+notify()
notify is called then the state changes
ConcreteSubject
-state
+getState()
subject
ConcreteObserver
+notify()
represents the data the subject is responsible for
does something with the data when notified

Theory 0.45: Sequence of calls:

1. Observers are attached to the subject indicating desire to be notified;
2. Subject's state is updated;
3. `Subject::notifyObserver()` is called - it calls each observer's `notify()` method;
4. Observer calls `ConcreteSubject::getState()` to query the state and react accordingly (may be back to 2).
5. When observer shut dowsn or are no longer interested, they detach from the Subject.

Example 0.64: Actual Code in repository folder /24-observer

Here the `HorseRace` is a concrete subject, while the `Bettor` 's are concrete observers. Black dimonds beside `TestHarness` indicate composition, so the `TestHarness` is responsible for creating/ destroying the HorseRace and Bettor objects.

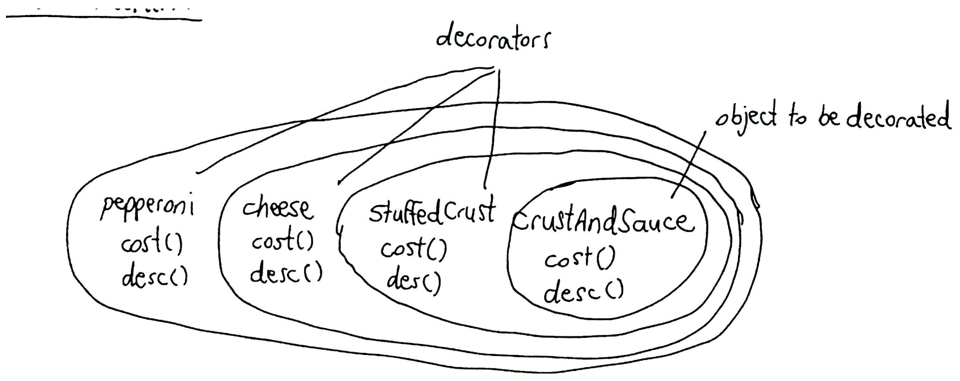


The ~Subject() is pure virtual, and this is what makes Subject abstract.

Result 0.33: Important Points for Observer Pattern

1. The subject does not know anything about the observers other than the fact that they all have the `notify()` method;
2. New types of observers can be created and added at any time without changing the code in the Subject;
3. Order of notification is generally not guaranteed;
4. Subject controls the state. Observers query it. Attenatively the subject can push the data to the observers;
5. State is not defined in the abstract base class Subject because it varies from one subject to the next;
6. You can reuse subjects and observers independently of one another;
7. Concrete subject uses implementation inheritance to get the basic functionality of attaching, detaching, and notifying "for free";
8. Interface inheritance allows all subjects to be used interchangeably and same for observers.

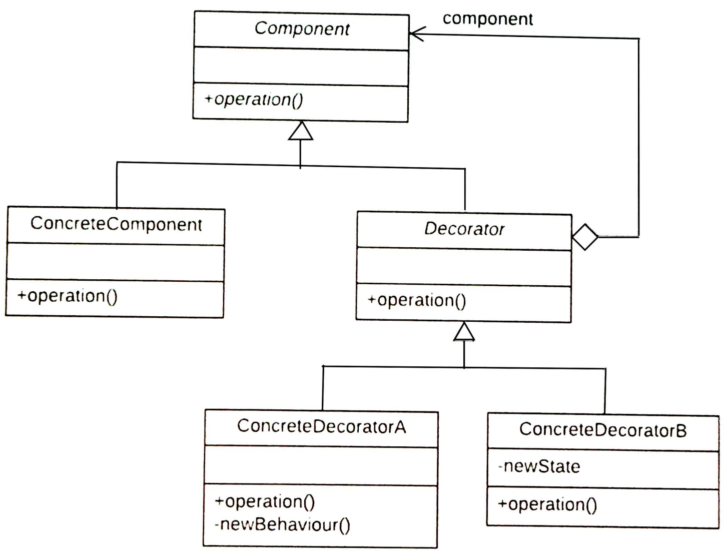
0.20.2 Decorator Pattern



Definition 0.35: General Decorator Pattern UML

Concrete Decorators can add new behaviour and/ or new states.

The general Decorator pattern UML:



Code 0.57

In code, this looks like this:

```
Pizza *p = new CrustAndSauce;
p = new StuffedCrust(p);
p = new Topping("cheese", p);
p = new Topping("pepperoni", p);
cout << "Your " << p->desc() << " pizza costs " << p->cost() << endl;
```

Discovery 0.31

1. Start with a CrustAndSauce object;
2. Decorate it with a StuffedCrust object;
3. . . .;
4. Call the `cost()` method which relied on delegation to add up all the costs.

Theory 0.46

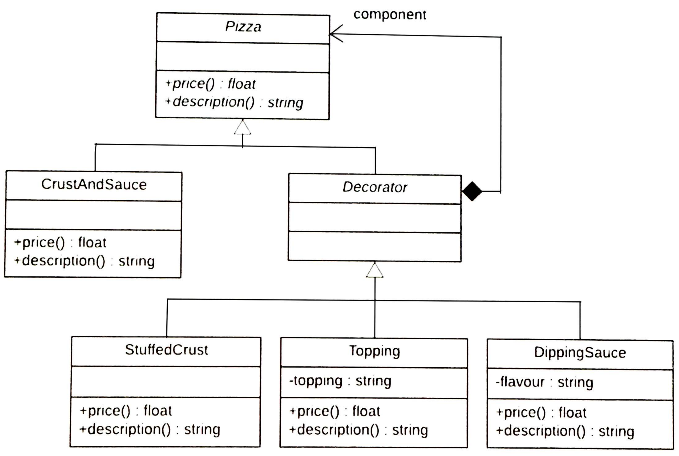
A CrustAndSauce object is a Pizza (through inheritance).
A CrustAndSauce object wrapped in a decorator is still a Pizza.

Code 0.58

```
class Pizza {
    public:
        virtual float cost() const = 0;
};
class CrustAndSauce : public Pizza {
    public:
        float cost() const override { return 5.99; }
};
class Decorator : public Pizza {
    protected:
        Pizza *component;
    public:
        Decorator(Pizza *p) : component{p} {}
        virtual ~Decorator() { delete component; }
};
class StuffedCrust : public Decorator {
    public:
        StuffedCrust(Pizza *p) : Decorator{p} {}
        float cost() const override { return component->cost() + 2.69; }
}
```

Example 0.65: UML for our example

CrustAndSauce is a concrete Pizza class, it is the innermost object we want to add behaviour to. The Decorator class is a Pizza class, but it has one more field - component.



The bottom three classes are decorators which are also Pizzas.

Result 0.34: Important Points for Decorator Pattern

1. Decorator have the same supertype as the object they decorate;
2. You can use one or more decorators to wrap an object;
3. You can pass around a decorated object in place of the original object;
4. The decorator adds its own behaviour before and/ or after delegating to the object it decorates;
5. Objects can be decorated at any time so we can decorate an object at runtime;
6. The Decorator pattern is an alternative to subclassing for extending behaviour.

0.21 Lecture 18

Lecture 18 - Tuesday, Jul 9

When should classes be grouped together in a module, and when should they be in separate modules? To answer this, we can explore two measures of design quality: coupling and cohesion.

0.21.1 Coupling

Definition 0.36: Coupling

Coupling refers to how much distinct program elements (classes, functions, modules) depend on one another:

1. High coupling (tightly coupled) means that the elements are closely connected and changes in one may have a ripple effect to others. This makes it harder to reuse individual elements.
2. Low coupling (loosely coupled) means that the elements are independent and changing in one element has little effect on others.

Result 0.35

You do need some degree of coupling. Loosely coupled elements can interact, but they generally have very little knowledge of each other. Design with loose coupling allow us to build flexible object-oriented systems that can handle change because they minimize the interdependence between objects.

Example 0.66

Observer Pattern is a good example of low coupling.

0.21.2 Cohesion

Definition 0.37: Cohesion

Cohesion refers to the degree to which elements of a module works together to fulfill a single, well-defined purpose

1. High cohesion means that the elements are closely related and focus on a single purpose;
2. Low cohesion: the elements are loosely related and serve multiple purposes.

Discovery 0.32

The elements could be methods of a class or classes in a module, etc.

Result 0.36

Low cohesion indicates poorly organized code. It's harder to reuse something without getting other stuff bundled with it.

Result 0.37: Strive for low coupling and high cohesion

Our goal is strive for low coupling and high cohesion.

High cohesion is related to a design principle:

Theory 0.47: Single Responsibility Principle

A class should have only one reason to change.

This means that you should strive to design your classes so they only have one responsibility, one reason to change. If a class does multiple things, there are more reasons to go back to that class and make changes.

$$\text{Changes} \Rightarrow \text{problems can creep in}$$

When your class does multiple things, it might affect multiple aspects of your designs.

Example 0.67

The iterator pattern is a good example following the Single Responsibility Principle.

0.21.3 What Should Go Into a Module

Code 0.59

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

This won't compile because compiler don't know how big A or B is.

The solution would be:

```
class B;          // forward reference
class A {
    int x;
    B *y;         // and change this to a pointer
};
```

Sometimes one class must come before another:

```
class C { };
class D : public C {...}     // we must know the size of C
```

How should A and B be placed into modules?

Theory 0.48

Modules must be compiled in dependency order. You cannot forward declare another module or any items within another module. Therefore A and B must reside in the same module. (This makes sense since A and B are tightly coupled.)

0.21.4 Decoupling the User Interface (MVC)

What would be wrong with this?

Code 0.60

```
class ChessBoard {       // has the core logic to maintain the
    ...                  // state of the Board
    ... cout << "Your Move" << endl;
    ...
};
```

This makes reuse harder because it interacts with the user directly.

```
class ChessBoard {
    istream &in;
    ostream &out;
```

```
        ...
        ... cout << "Your Move" << endl;
    };
```

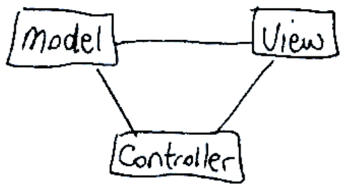
Discovery 0.33

We could possibly put the user interaction in `main`, but that's harder to reuse too.

Seperate the user interaction from the game state. Single Responsibility Principle is a guide.

Definition 0.38: Model View Controller (MVC)

The **Model View Controller (MVC)** is a architecture pattern:



This pattern separates the distinct responsibilities of

1. Model
   (a) state (or data) and the application logic or rules that change the state;
   (b) can have multiple views (e.g. text and graphics views);
   (c) doesn't need to know about their details.
2. View
   (a) the presents the state in the user interface;
   (b) gets the state from the model;
   (c) model and view typically use Observer Pattern.
3. Controller
   (a) mediates the flow of control between the model and view;
   (b) accepts input (from the view or other sources) and converts it to actions on the model or view.

Discovery 0.34

By decoupling application state, presentation, and control, MVC promotes reuse. CS 249 User Interface spends a lot of time talking about MVC.

0.21.5 Exception

Recall accessing elements of a vector v:

`v[i]` accesses the i-th element of v. No bounds checking! Unchecked.

`v.at(i)` accesses the i-th element of v. This checks if i is in bound. Bounds Checked.

What happens if you go out of bounds??

Example 0.68

In C, functions might use special return values or the global `errno` variable. This leads to awkward programming and encourages programmers to ignore error checks.

Definition 0.39

In C++, when an error condition is detected, instead of returning normally, the function **throws an exception** indicating what went wrong.

Theory 0.49

The fundamental idea behind **exception** is to separate detection of an error (which should be done in the called function) from the handling of an error (which should be done in the calling function) while ensuring that the detected error cannot be ignored.

Definition 0.40: "Handler"

When an exception is thrown, by default program execution stops, but we can write handlers to **catch** the exceptions and deal with them.

Example 0.69

`at()` throws an exception of type `std::out_of_range` when i is out of bounds. Handle it as follows:

Code 0.61

```
import <stdexcept>;      // standard exception types

try {   // smth that might throw an exception go in a try block
    ...
    cout << v.at(i) << endl;    // may throw an exception
    cout << "This does not get printed ig exception is thrown";
}
catch(out_of_range e) {      // we watch the exception using a catch handler
    cout << "Range Error: " << e.what() << endl;
}
```

`e.what()` gives the details about the error.

The exception causes execution to transfer to the catch block.

Consider another more complex example:

Example 0.70

```
void f() {
    throw out_of_range{"function f"};   // returned by e.what()
}
void g() { f(); }                   main
void h() { g(); }                   ->  h
                                      ->  g
int main() {                            ->  f
    try {                                 -> throw
        h();
    }
    catch(out_of_range) {...}
}
```

Theory 0.50

Control goes back up the call chain (called **stack unwinding**) untill a handler is found... all the way back to main(). If there is no matching handler in the entire call chain, the program terminates.

Theory 0.51

You can have more than one handler to catch different types of exceptions:

Code 0.62

```
try {
    ...
}
catch (out_of_range e) {...}
catch (bad_alloc) {...} // this is the exception that
                        // new throws when it fails
catch (...) {}          // actual ``...'' denotes the catch-all handler
                        // catches any exception type
```

Theory 0.52

As part of stack unwinding, any stack-allocated objects are destroyed automatically (i.e. their destructors run.)

0.21.6 The Exception Object

`out_of_range` is a class. The statement

```
throw out_of_range{"f"}
      \---------------/
             // costructs the object with "f" as ctor args
```

Definition 0.41: Exception Object

This object is called **the exception object**. Catch handlers can access the exception object through the parameter in the catch declaration.

What is the difference between:

```
catch(out_of_range e);
```

and

```
catch(out_of_range &e);
```

The first one catches by value while the second one catches by reference.

Result 0.38: Throw by value, catch by reference

In C++, we have a maxim “Throw by value, catch by reference”. Catching by reference is usually the right thing to do.

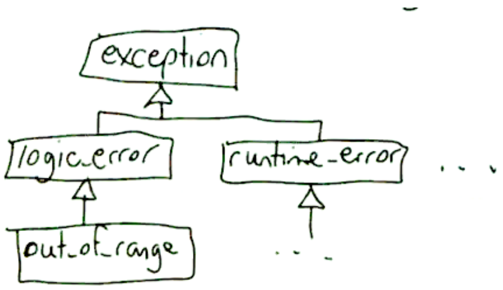
0.22 Lecture 19

Lecture 19 - Thursday, Jul 11

Code 0.63

Catch by value causes the exception object to be copied into the variable, which could cause *slicing* (see more in Definition 0.28) if you throw a derived class, like Text but catch a base class like Book. Catching exceptions by reference will not cause *slicing*.

Exception types are often part of a class hierarchy of exception classes. The `std::out_of_range` class is part of an exception hierarchy in the std library.



Subclasses can add additional fields or new behaviour (such as capturing details about the specific error that occurred).

0.22.1 Rethrowing an Exception

A handler can do part of the recovery job, i.e. executing some corrective code, and then it rethrow the original exception object:

Code 0.64

```
try {
} catch(SomeExceptionType &e) { // recovery/ clean up processing ...
    throw;      // no explicit object being thrown
}
```

or throw a *new* object:

```
try {
} catch (SomeExceptionType &e){
    ...
    throw e;
    // or
    throw SomeOtherExceptionType {...}
}
```

Theory 0.53: `throw object` vs. `throw`

`throw <object>` discards the original exception object and creates and throws a new one. The code above will throw either a SomeExceptionType object or a SomeOtherExceptionType object.
On the other hand, `throw` (without an object) rethrows the original exception object (without doing any copying) and the original type of the exception is retained.

Discovery 0.35

Slicing could occur when you are throwing too.

Example 0.71

The object being thrown might be a subclass of SomeExceptionType rather than SomeExceptionType itself.

Throw always throws an object with the same type as the static type of the expression.

Code 0.65

```
                    [Text]  -->  [Book]
--------------------------------------------------------------
throw Text{...}
catch(Book &e) {    // e is actually a Text object
    throw e;        // object being thrown is sliced into a Book object
                    // because e is a book &
}
```

0.22.2 What can be thrown?

Result 0.39

You can throw anything that you want – you don't have to throw objects.

Example 0.72: Folder: lectures/23-exceptions

Recursive Fib and Factorial functions that use throw to returnn values.

You can also define your own exception classes (or reuse existing ones):

Code 0.66

```
class BadInput{};
...
```

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

Discovery 0.36

Notice that the parameter name in the `catch` declaration is optional if you don't need to refer to it.

0.22.3 Exceptions in Destructors

ADVICE: never let a dtor throw an exception. By default, the program will terminate immediately (`std::terminate` will be called).

Theory 0.54

If a destructor is executed during unwinding while dealing with an exception and the destructor throws, you would now have two active unhandled exceptions, and the program will terminate immediately.

0.22.4 Nested Exception Handlers

If is valid to nest a `try...catch` inside another `try` block or a `catch` handler.

Example 0.73

Both of the following are valid.

```
try {
    ...
    try {               \
        ...             | -->  nested try ... catch
    } catch (...) {}    /
} catch (...) {
    try {               \
        ...             | -->  nested try ... catch
    } catch(...) {}     /
}
```

0.22.5 Why Exceptions?

Result 0.40

Exception makes your code simpler, cleaner, and less likely to miss errors.

See http://isocpp.org/wiki/faq/exceptions (really good resource).

0.22.6 Exception Safety (RAII)

Consider:

Code 0.67

```
void f() {
    C c;                // stack-allocated
    C *cp = new C;      // heap-allocated

    g();

    delete cp;
}
```

This looks correct without any memory leaks. However, what if `g()` throws?

Result 0.41

What is guaranteed? During stack unwinding, all stack-allocated data is cleaned up:

1. destructors run;
2. memory is reclaimed;

But heap-allocated memory is not reclaimed. Thus if `g()` throws, `c` is destroyed, but `cp` is not, which causes memory leak.

What we could do instead is:

Code 0.68

```
void f() {
    C c;                // stack-allocated
    C *cp = new C;      // heap-allocated

    try {
        g();
    } catch(...) {
        delete cp;
        throw;
    }
    delete cp;
}
```

We wrap the call to `g()` in an exception handler, but this is ugly and error-prone and code-duplication.

How can we guarantee that something (i.e. `delete cp`) will happen no matter how we exit f (either normally or by exception)?

Thought: You can count on destructor for stack-allocated data to run, thus use stack-allocated objects with destructors as much as possible. Use that to guarantee your advantage.

Theory 0.55: C++ Idiom

Resource Allocation Is Initialization – RAII

Interpretation: Every resource should be wrapped in a stack-allocated object, whose job is to release the resource.

0.22.7 Smart Pointer

Example 0.74

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

Acquiring the resource (the file) happens by initializing the object f. The file is guaranteed to be released (closed) when f goes out of scope.

This could also be done with dynamic memory:

```
import <memory>;

void f() {      // function rewritten to use RAII
    C c;
    std::unique_ptr<C> cp{new C};    // constructs C on heap,
                                     // return a ptr to it
    g();
}
```

The `unique_ptr`'s destructor will delete the pointer, we can dereference the `cp` object just like a regular pointer:

`cp->_____` or `*cp._____`

Definition 0.42: Smart Pointer

We call this a **smart pointer** because it automatically frees memory (dumb pointers don't do anything) when it goes out of scope.

Theory 0.56

Smart pointers should always be stack-allocated.

Example 0.75

Another way to create a smart pointer object:

```
void f() {
    std::unique_ptr<C> cp = std::make_unique<C>();
}
```

where `std::make_unique` is the function, `<C>` is the type of pointer being wrapped, and what goes inside the parentheses is the constructor arguments for `C`'s constructor.
`std::make_unique` internally calls `new C...`, forwarding the arguments in the parentheses to `C`'s constructor.

What if we try

Code 0.69

```
unique_ptr<C> cp{new C};
unique_ptr<C> cp2 = cp;      // calling copy ctor to copy cp
```

We are trying to copy a `unique_ptr`, but we don't want to delete the same ptr twice. Therefore copying is disabled for `unique_ptr`. They can only be moved.

Theory 0.57

If you need to copy pointers, you first need to answer the question of *ownership*:

1. Who will **own** the resource (who has the responsibility to free it);
2. That ptr should be a unique ptr object. All other ptrs should be **raw** pointers;
3. Can access the underlying raw pointer with `cp.get()`.

0.22.8 New Understanding of Pointers

This leads to a new understanding of pointers:

Result 0.42

1. unique ptr — indicates *ownership*;
2. raw ptr — indicate *non-ownership*, since the raw pointer is considered not to own the resource it points at, you should not delete it;
3. moving a unique ptr (into or out of a function) — transferring *ownership*.

0.23 Lecture 20

Lecture 20 - Tuesday, Jul 16

0.23.1 Pointers as Parameters

Moving a `unique_ptr` into a function:

1. `void f (unique_ptr<C> p)` ;
   f will take over ownership of the object pointed to by *p*. Caller *loses custody* of object.

   ```
   unique_ptr<C> cp {new C};    // cp manages the C object
   f(std::move(cp));            // transfers ownership of the
                                // managed object to f
                                // cp would now be 'empty'
   ```

2. `void g (C *p)` ; *g* will not take ownership of the object pointed to by *p*. Caller's ownership of the object does not change.

0.23.2 Pointers as Function Return Values

Moving a `unique_ptr` out of a function:

1. `unique_ptr<C> f()` ;
   return by value is always a move, so *f* is handing over ownership of the *C* object to the caller;

2. `C g()` ;
   the raw pointer is understood **not** to be deleted by the caller. It might be a pointer to non-heap data or to heap data that someone else already owns.

0.23.3 Shared Ownership: the Shared Ptr

Definition 0.43: `shared_ptr`

When you do need true shared ownership, i.e. any of several pointers might need to free the resource, then we use a `shared_ptr<C>` .

Example 0.76

```
{
    auto p1 = std::make_share<C>();     // allocates a C object on heap
    if (...) {
        auto p2 = p1;    // two ptrs pointing at the same object
                         // copy is allowed
    }   // p2 runs out of scope, but object is not deleted yet
}   // p1 goes out od scope, the object is deleted
```

Theory 0.58

`shared_ptr` maintains a reference count — a count of all shared pointers pointing at the same object. Memory is freed when the number of `shared_ptr`'s pointing to it reaches 0.

Result 0.43

There are three types of pointers: (raw, unique, and shared). They accurately reflects the pointer's onwership role. This leads to dramatically fewer opportunities for memory leaks.

0.23.4 STL Maps

Definition 0.44: Maps

Maps are also known as associative arrays or dictionaries. They are very useful container type. They store (key, value) pairs, and the key in a map is unique, while values can be set and updated.

Code 0.70

```
import <map>;

std::map<string, int> m;     // constructs an empty map
m["a"] = 2;
m["b"] = 3;
cout << m["a"] << endl;      // prints 2
cout << m["b"] << endl;      // prints 3
cout << m["c"] << endl;      // if key is not found, it is inserted and
                             // the value is default constructed
m.erase("a");        // removes the (key, value) pair from map
if (m.count("b"));   // 0 = not found; 1 = found

// interate a map, in sorted key order. Maps store keys in sorted order.
for (auto &p : m) cout << p.first << "=" << p.second << endl;
    // p's type here is std::pair<string, int> &
    // (pair is defined in <utility>):
    struct Pair {
        K first;
        V second;
    }

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

Definition 0.45: Structured Binding

The last iterating approach is called **structured binding**. Decomposes the pair into two local variables, called key and value

Discovery 0.37

Notice that `key` and `value` are references, so we can actually the value using the variable. However, the key cannot be altered.

Theory 0.59

These structured bindings can be used on any structure (class) type with all fields.

Example 0.77

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

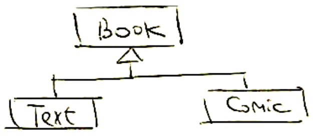
Example 0.78: Using on stack arrays where size is known

```
int a[] = {20, 30, 40};
auto [x, y, z] = a;      // x = 20, y = 30, z = 40
```

0.23.5 Inheritancey and Copy/ Move

When you have an inheritance hierarchy, you need to do a few special things with your copy/ move operations.

Code 0.71



```
class Book {
        string title, author;
        int length;
    public:
        // has copy/ move ctors and copy/ move assignment
};
```

```
class Text : public Book {
        string topic;
    public:
        // does not define copy/ move operations
}

Text t1{"Algorithms", "CLRS", 500, "CS"};
Text t2 = t1;
```

This copy initialization calls Book's copy ctor and then goes field-by-field (i.e. default behaviour) for Text part. The same is true for other compiler-provided operation.
If we were to write our own, they would look like this (note: they are equivalent to the compiler-provided ones):

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

// copy assignment
Text &Text::operator=(const Text &other) {
    Book::operator=(other);     // invoke superclass's copy ass
    topic = other.topic;        // copy text-specific fields
    return *this;
}

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

// move assignment
Text &Text::operator=(Text &&other) {
    Book::operator=(std::move(other));  // invoke superclass's move ass
    topic = std::move(other.topic);     // move text-specific fields
    return *this;
}
```

Theory 0.60

Even though other "points" at an rvalue, other itself is an lvalue (so is other.topic), therefore we use `std::move(x)` to force an lvalue of x to be treated as an rvalue, so the move operations run instead of the copy operations.

Definition 0.46: lvalue — locater value

An lvalue is anything that:

1. can appear on the left hand side of an assignment statement;
2. denotes a storage location;
3. has a name;
4. allows you to take its address using the address-of-operator.

Definition 0.47: rvalue

An rvalue is an temporary object who has none of the properties an lvalue possesses.

Result 0.44

The above is the general pattern for subclass copy/ move operations. Specialize as needed for other classes.

0.23.6 Preventing Partial and Mixed Assignment

Now consider:

Example 0.79

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

What if we do:

```
    *pb1 = *pb2;     // we expect this to be equivalent to t1 = t2
                     // but it's actually not
```

In fact, `Book::operator=()`, and the result is **partial assignment** (Only the Book portion is copied over while the Text portion stays unchanged).

Could we make operator = virtual?

Code 0.72

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

The above won't even compile because the parameter type doesn't match. Suppose we try:

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

This now will compile, but now this is still problematic because this version would except any kind of Book on the right hand side of an assignment (a Text object, but also a Book or a Comic).

Definition 0.48: Mixed Assignment

The above is called **mixed assignment**, which should not be allowed between objects that have different "shapes" (or data fields).
Remark: for a more in-depth treatment of this topic, check out: this link.

Result 0.45

1. If `operator=()` is non-virtual, we get partial assignment when assigning through base class pointers;
2. If `operator=()` is virtual, the compiler will allow mixed assignmnet.

0.24 Lecture 21

Lecture 21 - Thursday, Jul 18

Continue from the last lecture. Our solution to this problem (partial and mixed assignment) is to make all superclasses abstract (making all non-leaf classes abstract).

Result 0.46

Rewrite our Book hierarchy:

We introduce a new supercass that has the common features of Book, Text, and Comic, and we make it abstract.
This eliminates the need to allow assignment between Book objects (i.e. when Book was the base class, it was also a concrete object).
But this means we have to change our code a little bit:

Code 0.73

```
class AbstractBook {
        string title, author;
        int length;
    protected:
        AbstractBook &operator=(const AbstractBook &other); // not virtual
    public:
        AbstractBook(...);
        virtual ~AbstractBook() = 0; // need at least one pure virtual method
                                     // use the destructor if don't have one
}
class Book : public AbstractBook {
    public:
        ...
        Book &operator=(const Book &other) { // Book-to-Book ass (same as b4)
            AbstractBook::operator=(other);
            return *this;
        }
}
class Text : public AbstractBook {
    public:
        ...
        Text &operator=(const Text &other) {
            AbstractBook::operator=(other);
            topic = other.topic;
            return *this;
        }
}
```

This solution gives you everthing you need:

1. you can assign “like” object: `b1 = b2`, `t1 = t2`, or `c1 = c2`;
2. Partial and mixed assignment are prohibited;
3. Assignment through base class pointers is prohibited (avoiding unintentional partial and mixed assignmnet);

4. Derived class assignment operators may call the assignment operator in base class;

Theory 0.61

When you have a pure vitual destructor, it must be implemented even though it's pure virtual, or your program will not link. Most pure virtual functions are never implemented, but pure virtual destructor is a special case. (The subclass destructor will call the base class destructor, so it must exist in program.)

0.24.1 Casting

Recall C-style casting:

```
double d = 3.14;
int x = (int)d;

Node n;
int *np = (int *)&n;     // force C++ to treate a node ptr a an int ptr
```

Theory 0.62

Casting should be avoided, in particular, C-style casts should be avoided in C++.
Aside: If you must casr, C++ provides casting operations.

Code 0.74

1. `static_cast` — this is for "sensible casts" that have well-defined semantics:

   Example 0.80

   (a) `double` to `int`:

   ```
   f(int x);
   f(double d);

   double d;
   f(static_cast<int>(d));      // call int version of f
   ```

   (b) superclass to subclass pointer:

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

   You are taking the responsibility that b actually points to a Text object. You are telling the compiler "trust me"

   This will invoke a conversion constructor if one exists (recall this is a one-argument constructor):

Discovery 0.38

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

Vec v = static_cast<Vec>(5);    // invokes the "int to Vec"
                                // conversion constructor
```

If you try to do a static cast but it is not a valid conversion (the constructor indtead does not exist), you would get a compiler error.

Code 0.75

2 `reinterpret_cast` — unsafe, implementation-dependent, “weird” conversions. Most uses of `reinterpret_cast` result in undefined behaviour.

```
Student s;
Turtle *t = reinterpret_cast<Turtle *>(&s); // forces a Student to be
                                            // treated as a Turtle
```

Tells the compiler to treate the expression as the indicated type. No code (CPU instructions) are generated. Nothing happens at runtime.

Code 0.76

3 `const_cast` — is for converting between const and non-const. This is the only C++ cast that can “cast away const”.

```
void g(int *p);
void f(const int *p) {          // p is a ptr to a const int
    g(p);                       // compiler error, invalid conversion
                                // of const int * to int *
    g(const_cast<int *>(p));    // compile just fine, we know that
                                // g will not modify the int
}
```

Code 0.77

4 `dynamic_cast` — safely converts pointers and references to object up and down the inheritance hierarchy.

```
Book *bp = ...;
```

```
Text *tp = static_cast<Text *>(bp); // not sure if safe, depends on
                                    // what bp is pointing to.
                                    // If its not a Text,
                                    // we get undefined behaviour.
```

It would be better to do a tentative cast: try it and see if it succeeds.

```
Book *bp = ...;
Text *tp = dynamic_cast<Text *>(bp); // attempt cast,
                                     // returns nullptr if invalid
if (tp) {
    cout << tp->getTopic() << endl;
} else {
    cout << "Not a Text" << endl;
}
```

If the cast works, then tp points to the object, otherwise get nullptr.

Theory 0.63

There are smart pointer versions of these, but they are shared pointers instead of unique pointers, and they are functions:

1. `static_pointer_cast`;
2. `reinterpret_pointer_cast`;
3. `const_pointer_cast`;
4. `dynamic_pointer_cast`;

The **functions**, defined in `<memory>`, cast `shared_ptrs`.

Example 0.81: Dynamic cast also works on references

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

If b points to a Text, then t2 is a reference to the same Text. Otherwise, because there is no such thing as a “null reference”, `std::bad_cast` is thrown.

Theory 0.64

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

0.24.2 Static Fields and Methods

Definition 0.49: Static Members

Static members (fields and methods in a class) are associated with the class itself, and not with any particular instance (object) of the class.

Example 0.82

```
struct Student {
    ...
    inline static int numInstances = 0; // inline lets u initialize
                                         // it inline in your class
    student(...) : ... {
        ++numInstances;
    }
}

cout << Student::numInstances << endl;
```

Definition 0.50: Static member function

Static member functions/ methods don't depend on a specific instance for their computation (don't have a this parameter).

Example 0.83

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

Student s1{...};
Student::howMany();     // 2
```

0.24.3 Factory Method Pattern

Suppose we have a problem: we want to write a video game with two kinds of enemies: turtles and bullets. The system sends turtles and bullets our way, but bullets become more frequent in later levels.

We don't want to hardcode the enemy-generation policy. It should be flexible at runtime. We will create a **Factory Method** in Level that creates enemies:

Code 0.78

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

we can have different policies by creating subclasses:

```
class Easy : public Level {
    public:
        Enemy *createEnemy() override {
            // creete mostly turtles
        }
};
class Hard : public Level {
    public:
        Enemy *createEnemy() override {
            // creete mostly bullets
        }
}
```

As a result, we can switch policy at runtime:

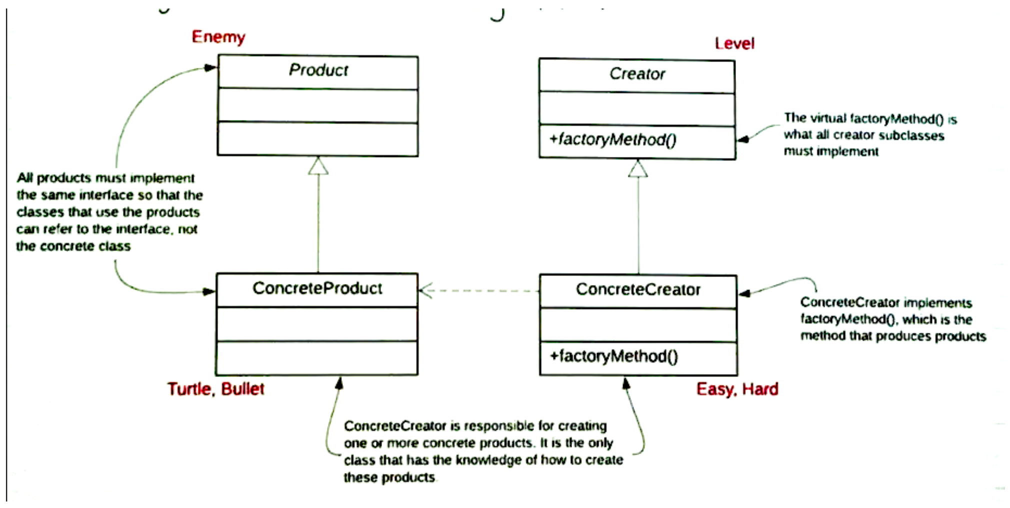
```
Level *easyLvl = new Easy;
Level *hardLvl = new Hard;

Level *currentLvl = easyLvl;     // start easy;

while (true) {
    ...
    if (...) currentLvl = hardLvl;  // at some point we switch
    Enemy *enemy = current->createEnemy();  // factory method
}
```

0.25 Lecture 22

Lecture 22 - Tuesday, Jul 23

Definition 0.51: Factory Method Pattern

The **Factory Method Pattern** defines on interface for creating an object, but lets subclasses decide which class to instantiate. Factory method lets a class defer instantiation to subclasses.



Enemy
Product
Level
Creator
+factoryMethod()
The virtual factoryMethod() is what all creator subclasses must implement
All products must implement the same interface so that the classes that use the products can refer to the interface, not the concrete class
ConcreteProduct
ConcreteCreator
ConcreteCreator implements factoryMethod(), which is the method that produces products
+factoryMethod()
Turtle, Bullet
Easy, Hard
ConcreteCreator is responsible for creating one or more concrete products. It is the only class that has the knowledge of how to create these products

Definition 0.52

This pattern is also known as the **virtual construct pattern** because the pattern that creates the objects is virtual.

Result 0.47: Important

Important Points:

1. Factory method relies on inheritance: object creation is delegated to subclasses, which implement the factory method to create objects;
2. This pattern promotes base coupling by reducing the dependency of your application on concrete classes. Your application only depends on the abstract interface, not on specific concrete classes.

Another example of the oo guiding principle of “program to interfaces, not implementations”.

Theory 0.65

Abstract base classes define the interface. Your client code should be written to use base class pointers and call the methods in the base class interface. The interface is the abstraction. Concrete subclasses can be swapped in and out to provide a variety of behaviours for this abstraction. Promotes loose coupling.

0.25.1 Exception Safety Guarantees

Definition 0.53

This means: “if program execution leaves a function because of an exception, that the program is not left in a broken or unusable state. ”

Theory 0.66

Specifically, the levels of exception safety for a function f (in decreasing order of safety) are:

1. **No-throw Guarantee**: f promises to never throw or propagate an exception and will always complete successfully. If an exception occurs internally, it will be handed internally and won’t be observed by clients. If f can’t fully accomplish its tasks, it may indicate failure in some other ways (such as setting an internal flag or returning a result code).
2. **Strong Guarantee**: If f throws or propagates an exception, the state of the program will be as if f had never been called. If f changes some state, it must undo all of those changes to restore the original state before throwing the exception. In other words, from the outside, it must appear that either:
   (a) f succeded in doing everything it was asked to do (normal exit), or;
   (b) nothing happened except an exception was thrown indicating error.
3. **Basic Guarantee**: If f throws or propagates an exception, the program will be in some valid, maybe unspecified, state. No resources are leaked, no corrupted data structures, and all classes invariantes are intact.
4. **No Exception Safety**: f makes none of the above guaratnees.

Example 0.84: No-throw Guarantee

1. Pointer assignment;
2. delete (to delete an object), recall that destructors should never throw exceptions;
3. `int i; cin >> i;`

   If not successful, this sets the `fail()` flag.
4. Lots of built-in facilities in C++ provide guarantee, they will never throw.

Example 0.85: Basic Guarantee

```
class Node {
    ...
    Node &operator=(const Node &other) {    // copy ass
        if (this == &other) return *this;
        delete next;
        next = nullptr;      // removes the dangling ptr
        next = other.next ? new Node{*other.next} : nullptr;     // may throw
        data = other.data;
        return *this;
    }
};
```

Notice that that line of code may throw because it is a copy constructor. If it does, next stays as `nullptr`.

Example 0.86: Strong Guarantee

```
class A {
    public:
        void g();    // offers the strong guarantee, may throw
};
class B {
    public:
        void h();    // offers the strong guarantee, may throw
};

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

What safety level does f offer?

1. If `a.g()` throws, then any changes it made are "undone", and then f propogates the exception. At this point we have a strong guarantee.
2. If `b.h()` throws, the effects of `a.g()` must be undone to have f offer the strong guarantee, which is hard or even impossible if `a.g()` has non-local side effects.

if `A::g()` or `B::h()` only have local side effects (they only change themselves), we can rewrite f to offer the strong guarantee. One way to do this is to use a copy-swap idiom:

```
    void f() {
        A tempa = a;     // create a copy of a
        B tempb = b;     // create a copy of b
        tempa.g();      // if they throw, the original are still intact
        tempb.h();

        a = tempa;      // swap our a and b with modified ones
        b = tempb;
    }
```

However, this is only mostly correct. The last line is the problem: If it throws (assignment operations could throw), we have already partially updated our state (a has been updated but not b).

It would be better if the last two lines will guarantee not to throw. Reacall that assigning pointers can never throw.

0.25.2 pImpl Idiom

Code 0.79: Access C's internal state through a pointer — pImpl idiom

```
    struct CImpl {
        A a;
        B b;
    };
    class C {
        // construct CImpl on the heap but using a smart ptr to manage
        unique_ptr<CImpl> pImpl{new CImpl};
        public:
            void f() {
                // copy our state using Cimpl's copy ctor
                auto temp = make_unique<CImpl>(*pImpl);

                temp->a.g();
                temp->b.h()

                // swapping ptrs or unique_ptrs is no-throw!
                std::swap(pImpl, temp);
            }
    };
```

Now *f* offers the strong guarantee. It guarantees that if *f* throws an exception, the internal state of the C object is as if *f* had never run.

Discovery 0.39

Note that if `A::g()` or `B::h()` offer no exception safety guarantee, then in general, neither can `C::f()`.

0.25.3 Exception Safety and the Standard Template Library : Vector

Definition 0.54

Remember that the vector class:

1. encapsulates a heap allocated array;
2. when a stack-allocated vector goes out of scope or a heap-allocated vector is deleted, the internal array is freed and the objects in the vector are destroyed (if applicable).

Example 0.87

```
void f() {
    vector<C> v1;                 // vector of objects
    vector<C *> v2;               // vector of ptrs
    vector<unique_ptr<C>> v3;     // vector of smart ptr objects
    ...
}
```

When the function ends, $v1, v2$, and $v3$ go out of scope:

1. v1: C destructor runs on all objects in the vector;
2. v2: ptrs don't have destructors, so objects are not deleted. The vector v2 is not considered as owning the objects.
3. v3: unique ptr destructor runs on all objects, which deletes each smart pointer's C object. Don't need an explicit delete statement.

Result 0.48

In summary,

1. `vector<C>` — owns the objects;
2. `vector<C *>` — does not own the objects;
3. `vector<unique_ptr<C>>` — owns the objects.

How does vector provide the strong guarantee?
Consider `emplace_back`.

Example 0.88

```
If the array is full (size == capacity)
    allocate a new, larger array
        if allocation fails
            the old array is still intact
            exits with exception
    copy the objects over (using copy ctor)
        if copy ctor throws, need to undo the work so far:
            delete objects already copied
            delete new array
            old array is still intact
            exit with an exception
    delete the old array and the old objects
    exit successfully.
```

0.26 Lecture 23

Lecture 23 - Thursday, Jul 25

Result 0.49

The strong guarantee comes at a price: copying is expensive, and the old objects will just be thrown away, so it is wasteful.

Wouldn't *moving* the objects be more efficient?

1. allocate the new array;
2. move the objects over (move constructor);
3. delete the old array.

Discovery 0.40

If a move constructor throws, then `emplace_back` cannot offer the strong guarantee because the objects that have been moved over will no longer be intact.

Definition 0.55

But if the move constructor offers the no-throw guarantee, then `emplace_back` can and will use the move constructor (faster). Otherwise it will use the copy constructor (slower).

Code 0.80

```
class C {
    public:
        C(C &&) noexcept {...}  // noexcept shows we are offering
                                // the no-throw guarantee
        C &operator=(C &&) noexcept {}
}
```

If you know that a function will never throw or propagate an exception, declare it **noexcept**. This facilitates optimization. At a minimum, moves should be **noexcept**.

0.26.1 Template Method Design Pattern

Definition 0.56: Template Method

The **template method pattern** can be used when we want subclasses to override *some* of the superclass behaviour, but other aspects must stay the same.

Example 0.89

noexcept.

```
class Turtle {
    public:
        void draw() {
            drawHead();
            drawShell();
            drawFeet();
        }
    private:
        void drawHead() {...}
        virtual void drawShell() = 0;    // virtual method can be private
        void drawFeet() {...}
};
class RedTurtle : public Turtle {
    void drawShell() override { /* draw red shell; */ }
};
class GreenTurtle : public Turtle {
    void drawShell() override { /* draw green shell; */ }
};
```

The part that must stay the same is the way the turtle is drawn. Subclasses can only change the way the shell is drawn.

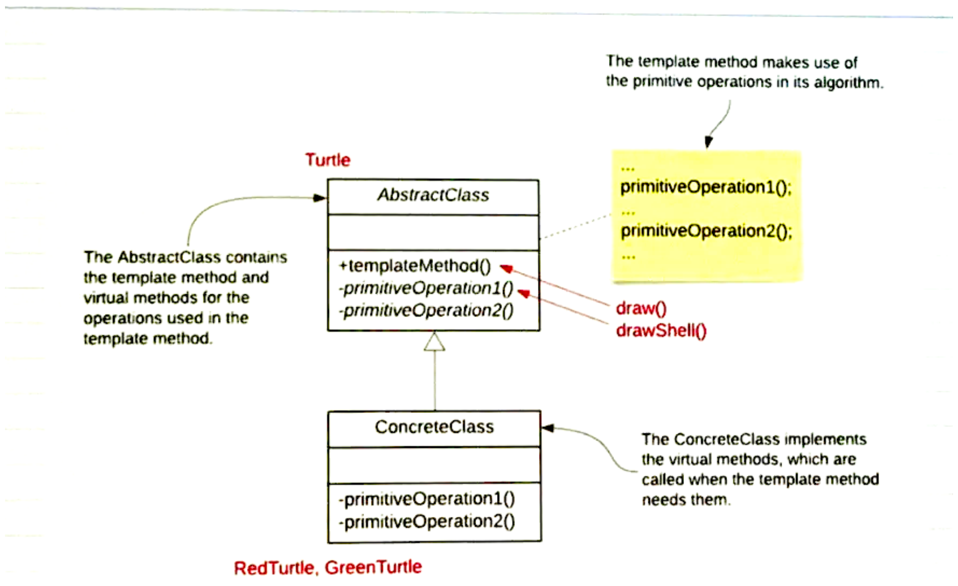
Theory 0.67

This pattern is often used to allow subclasses to customize or provide an implementation *for steps of an algorithm*. The overall algorithm is implemented in the base class as a **template method** — provides a template (or the steps) of the algorithm:

1. `draw` is the template method;
2. `drawHead`, `drawShell`, and `drawFeet` are the steps;
3. `Turtle` controls the overall algorithm, but allows the subclasses to "customize" various steps, like `drawShell`.

Definition 0.57: Template Method

Defines the skeleton of an algorithm in a method, deferring some steps to subclasses. Subclasses can redefine certain steps without changing the algorithm's structure.


The template method makes use of the primitive operations in its algorithm.
Turtle
AbstractClass
+templateMethod()
-primitiveOperation1()
-primitiveOperation2()
...
primitiveOperation1();
...
primitiveOperation2();
...
draw()
drawShell()
The AbstractClass contains the template method and virtual methods for the operations used in the template method.
ConcreteClass
-primitiveOperation1()
-primitiveOperation2()
The ConcreteClass implements the virtual methods, which are called when the template method needs them.
RedTurtle, GreenTurtle

Result 0.50: Non-virtual Interface (NVI) Idiom

Template method leads into another C++ idiom, the Non-virtual Interface (NVI) Idiom

0.26.2 Non-virtual Interface (NVI) Idiom

Discovery 0.41

public virtual method is really two things:

1. public: an interface for the client

- promises certain behaviours to the client, such as pre- and post- conditions, invariants, etc.

2. virtual: an interface for subclasses

   - behaviour can be replaced with anything the subclass wants

These are at odds with one another — making promises that subclasses can break.

Theory 0.68: NVI to the rescue

1. All public methods should be non-virtual;

2. All virtual methods should be private, or at the very least, protected.

This exposes the non-virtual methods to the clients and provides virtual methods for the subclasses to replace or override certain behaviours (just as template methods).

Example 0.90

```
class DigitalMedia {     // ABC (Abstract Base Class) defines
                         // a common interface for playing digital media
    public:
        virtual void play() = 0;
};
```

Translated into NVI:

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

Now if we need to exert extra control over play(), we can do it:

1. we could later decide to add before/ after code around doPlay() that subclasses cannot change (e.g. check copy right before, or update a play count after).

2. we can also add "hooks" by calling additional virtual methods from play (e.g. showCoverArt()).

3. we can do all of this without changing the public interface.

It is much easier to take this kind of control over our virtual methods from the beginning than to try take back control over them later.

Code 0.81

The NVI idiom extends the template methods by providing every virtual method inside a non-virtual wrapper. (This is actaully how our Turtle example was structured.) There is essentially no disadvantage from a performance perspective because a good compiler can optimize away the extra function call.

0.26.3 Template Functions

Similar to template classes, we can create template functions that are parameterized by one more more types.

Code 0.82

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

// using it with different types:
int i1 = ..., i2 = ...;
int i = min(i1, i2);         // T = min

double d1 = ..., d2 = ...;
double d = min(d1, d2);      // T = double
```

Theory 0.69

C++ can infer T from the types of the arguments.

Example 0.91

If C++ cannot determine T, for example if a function has no arguments, you could tell it explicitly:

```
z = min<int>(x, y);      // explicitly specifying the type of T
```

Discovery 0.42

For what types T can min be used? In other words, for what types does the body of min compile?
Answer : for any type for which operator< is defined.

We can generalize a function by making the types of its parameters template arguments:

Code 0.83

```
// sometimes you may see class instead of typename,
// they are interchangeable to compiler
template <typename Iter, typename Fn>
void for_each(Iter start, Iter finish, Fn f) {
    while (start != finish) {
        f(*start);
        ++start;
    } // while loop
} // function
```

`Iter` can be any type that supports `!=`, `*`, and `++`, including raw pointers. And f can be any function that accepts the type returned by the iterator's operator$*$.

```
void writeOut(int n) { cout << n << endl; }
...
int a[] = {1, 2, 3, 4, 5};
for_each(a, a + 5, writeOut);    // prints the array;
                                 // a+5 is just simple ptr arithmetic
```

0.26.4 The Algorithm Library (STL)

Definition 0.58: STL Algorithm Library

The STL Algorithm Library is a suite of template functions (`import <algorithm>;`) many of which work with iterators.

Example 0.92

1. `for_each` — as described above;
2. `find` — search for a value;

   Code 0.84

   ```
   template <typename Iter, typename T>
   Iter find(Iter first, Iter last, const T &val) {
       // returns an iterator to the first element in
       // [first, last) matching val or last if val is not found
   }
   ```

3. `count` — like find(), but instead returns the number of occurrences of val;
4. `copy` — copies one container range to another.

Code 0.85

```
template <typename InIter, typename OutIter>
OutIter copy(InIter first, InIter last, OutIter result) {
    // copies from the range [first, last) to [result, ...)
    // Note: copy does not allocate new memory,
    //       the output container must have the space available
}

// example
vector v{1, 2, 3, 4, 5, 6, 7};
vector<int> w(4);            // space for 4 elements
copy(v.begin() +1, v.begin() +5, w.begin());  // w = {2, 3, 4, 5}
```

0.27 Lecture 24

Lecture 24 - Tuesday, Jul 30

Example 0.93

1. `transform` — transforms values using a user-defined function

Code 0.86

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

// Example:
int add1(int n) { return n+1; }
...
vector v{2, 3, 5, 7, 11};
vector<int> w(v.size());    // create a vector of zeros size matching v
transform(v.begin(), v.end(), w.begin(), add1); // w = {3, 4, 6, 8, 12}
```

0.27.1 Function Objects

What can we use for `f` in transform? We have to see how `f` is used in transform:

Code 0.87

```
*result = f(*first);
```

`f` can be anything that can be called as a function with the appropriate parameter and return types.

Theory 0.70

We can write an `operator()` method in a class (`operator()` is the function call operator).

Example 0.94

```
class Plus1 {
    public:
        int operator()(int n) { return n+1; }
};
Plus1 p;     // constructs a Plus1 object
p(4);        // uses the object like a function, producing 5

transform(v.begin(), v.end(), w.begin(), p);    // using the object as a fn
```

Plus1 can be easily generalized:

```
class Plus {
        int m;
    public:
        Plus(int m) : m{m} {}
        int operator()(int n) { return n+m; }
};
Plus p{7};  // constructs a Plus object
p(4);       // uses the object like a function, producing 11

transform(..., ..., ..., Plus{1});    // last arg is ctor call
```

Definition 0.59: Function Objects

Instances of Plus1 and Plus are called **function objects** — objects that can behave like a function

Result 0.51

Advantages of function objects are: they can be configured with contructors and can maintain state.

Code 0.88

```
class IncreasingPlus {
    int m = 0;
    public:
        int operator()(int n) { return n + (m++); }
        void reset() { m = 0; }
};
vector<int>(5, 0);  // creates a vector of 5 int values of 0
vector<int> w(v.size());    // destination vector
transform(..., IncreasingPlus{});   // w = {0, 1, 2, 3, 4}
```

Discovery 0.43

Function objects are used extensively in the STL. We use them to specify what we are searching for in search functions (such as `find_if`), for defining sort criteria (for function sort), etc.

Example 0.95: Sorting objects by different criteria

```
struct Person {
    string name;
    string address;
};
vector<Person> v;

sort(v.begin(), v.end(), compareByName{});
sort(v.begin(), v.end(), compareByAddress{});
```

Then we can define function objects:

```
struct compareByName {
    bool operator()(const Person &a, const Person &b) const {
        return a.name < b.name;
    }
};
struct compareByAddress {
    //  same as above
```

```
        return a.address < b.address;
    };
```

0.27.2 Lambdas

Definition 0.60: Lambda expressions

Lambda expressions can be a convenient shorthand for function objects or functions.

Example 0.96

Consider the question: how many ints in a vector are even?

Code 0.89

```
vector<int> v;
...
bool even(int n) { return n % 2 == 0; }
...
int num_evens = count_if(v.begin(), v.end(), even); // using a function
```

Using a lambda expression instead:

Code 0.90

```
int num_evens = count_if(v.begin(), v.end(), [](int n){return n%2 == 0;});
```

Definition 0.61

A lambda expression is an unnamed function, specified right where it is needed. The square brackets is known as a lambda introducer, and there are more other lambda introducers:

1. `[]` — lambda introducer, ordinary function that can access its own parameters, its own local variables, and anything in the global scope;
2. `[&]` — can use names from its enclosing scope, by reference;
3. `[=]` — can use names from its enclosing scope, by reference or by value (accessing *copies* of variables from the enclosing scope).

0.27.3 More Uses of Iterators

Definition 0.62: Iterator

An **iterator** is anything that supports the opreations `*`, `++`, and `!=`. We can apply the notion of an iterator to other data sources or destinations such as streams.

Example 0.97

```
import <iostream>;
import <iterator>;

ostream_iterator<int> osi { cout, " " };
// writes ints to cout, ctor takes an output stream
// and an optional string to be printed after each value

*osi = 13;  // writes "13 " to cout
*osi = 42;  // writes "42 " to cout
            // we dont need to advance the iterator;
            // ++osi does nothing

vector<int> v{1, 2, 3, 4, 5};
copy(v.begin(), v.end(), osi);  // write "1 2 3 4 5 " to cout

istream_iterator<int> isi { cin };  // reads from cin
int i1 = *isi;  // read value
++isi;          // advance iterator
int i2 = *isi;
```

Now consider

```
vector<int> w;
copy(v.begin(), v.end(), back_inserter(w));
// constructs an iterator that calls push_back on w
// every time operator* is assigned
```

Remember, `copy` doesn't allocate space in w, it can't because it doesn't even know what kind of container w iterates over. The `back_inserter` inserts a new item in w each time. Now v is copied to the end of w by adding new items. Back inserters are available for any container with a `push_back` method.

0.27.4 Is Dynamic Casting Good Style?

Recall `dynamic_cast`. You can use it to make decisions based on an object's run time type information (RTTI):

Example 0.98

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

Discovery 0.44

Notice that code like this is tightly coupled to the Book hierarchy, and it may indicate bad design. Why? Consider what happens if we introduce a new subclass of Book. We need to add a new case, or else it doesn't work anymore.

Theory 0.71

It is better to use virtual methods.

Code 0.91

Fix the whatIsIt example:

```
class Book {
    ...
    virtual string identify() { return "Normal Book"; }
};
// Override identify in Text and Comic

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

This works by creating an interface function that is uniform across all Book types. Each subclass implements the appropriate logic for the virtual method so you don't need to write code like above that uses `dynamic_cast`.

However, not all dynamic casting are bad design:

Definition 0.63: Dynamic reference casting

Dynamic reference casting offers a possible solution to the polymorphic assignment problem (e.g. if we want to allow assignment through base class pointers).

Code 0.92

Recall

```
Book *pb1 = ...;
Book *pb2 = ...;
*pb1 = *pb2;
```

We would expect this to work for “like” objects (b1 = b2, t1 = t2) but not “unlike” objects (b1 = t1). To make this work,

```
class Book {
    puclib:
        virtual Book operator=(const Book &other);  // virtual
};
class Text : public Book {
    public:
        Text &operator=(const Book &other) override {   // override
            const Text &textother = dynamic_cast<const Text &>(other);
                // implements a run time check if the object copied
                // is actually a Text. Throws if it isn't.
                // Prevents "unlike" assignment.
            Book::operator=(other);
            topic = textother.topic;
            return *this;
        }
}
```