



Object-Oriented Software Development

CS 246E



Brad Lushman

Contents

I Basic C++	4
1 Problem 1: Program Input and Output	5
1.1 Consider C	5
1.2 Consider C++	6
1.2.1 stdin/stdout	6
1.2.2 File access	7
1.3 References	9
1.4 Separate Compilation	10
1.5 Linux make	11
2 Problem 2: Linear Collection + Modularity	13
2.1 C preprocessor	14
2.1.1 Fixing the double-include problem	15
3 Problem 3: Linear Collection and Memory Management	17
3.1 Introduction to Classes	19
3.1.1 Initializing Objects	20
3.1.2 MIL	21
4 Problem 4: Copies	25
4.1 Copy and Swap Idiom	26
5 Problem 5: Moves	27
5.1 Copy/Move Elision	29
6 Problem 6: I want a constant vector	31
7 Problem 7: Tampering	33
8 Problem 8: Efficient Iteration	36
8.1 Iterator Pattern	36
9 Problem 9: Staying in Bounds	41
10 Problem 10: I want a vector of chars	45
11 Problem 11: Better initialization	47

12 Problem 12: I want a vector of Posns	50
13 Problem 13: Less Copying!	52
14 Problem 14: Memory Management is Hard!	56
15 Problem 15: Is vector exception safe?	60
16 Problem 16: Insert/remove in the middle	64
17 Problem 17: Abstraction Over Containers	66
18 Problem 18: Heterogeneous Data	69
19 Problem 19: I'm Leaking	75
20 Problem 20: I want a class with no objects	77
21 Problem 21: The copier is broken	78
22 Problem 22: I want to know what kind of Book I have	80
II A Big Unit on Objected-Oriented Design	83
23 UML	84
24 Measures of Design Quality	86
24.1 Coupling & Cohesion	86
25 SOLID Principles of OO design	88
25.1 <u>S</u> ingle Responsible Principle	88
25.2 <u>O</u> pen-Closed Principle	89
25.3 <u>L</u> iskov Substitution Principle	90
25.4 <u>I</u> nterface Segregation Principle	94
25.5 <u>D</u> ependency Inversion Principle	97
26 Some More Design Patterns	101
26.1 Factory Method Pattern	101
26.2 Decorator Pattern	102
26.3 Visitor Pattern	102
III Abstraction in C++	107
27 Problem 23: Shared OwnerShip	108
28 Problem 24: Abstraction over Iterators	111
28.1 Template Metaprogramming	114
29 Problem 25: I want an even faster vector	116
29.1 Move/Forward Implementation	119
30 Problem 26: Collecting Stats	122
31 Problem 27: Resolving Method Overrides At Compile-Time	124
32 Problem 28: Polymorphic Cloning	126

33 Problem 29: Logging	128
33.1 Mixin inheritance	129
34 Problem 30: Total Control	130
35 Problem 31: Total Control ove Vectors & Lists	133
36 Problem 32: A fixed-size allocator	134
37 Problem 34: I want a (tiny bit) smaller vector class	138
37.1 Something Missing from Last Year	138

Part I
Basic C++

Problem 1: Program Input and Output

Running a program from the command line:

```
./program-name # . means current directory or path/to/program-name
```

Providing inputs in two ways:

1. `./program-name arg1 arg2 ... argn`
Args are written into the program's memory (see Figure 1.1a)
2. `./program-name` Then type sth else
Input comes through standard input stream (stdin) instead (see Figure 1.1b)
Can be connected to other sources / destinations, e.g. files

Redirection `./my-prog < infile > outfile 2> errfile`

1.1 Consider C

```
#include <stdio.h>
void echo(FILE *f) {
    for (int c = fgetc(f); c != EOF; c = fgetc(f)) // fetch from file
        putchar(c); // put to stdout
}
int main(int argc, /* # command line args (always > 1) \ (count) */
         char *argv[] /* command-line args \ (vector)* / ) {
    // argv[0] = program name ... argv[argc] = NULL
    if (argc == 1) echo(stdin);
    else
        for (int i = 1; i < argc; ++i) {
            FILE *f = fopen(argv[i], "r");
            echo(f);
            f(close);
        }
    return 0; // status code convention -0 = OK
}
```

`echo $?` status code of most recently run program.

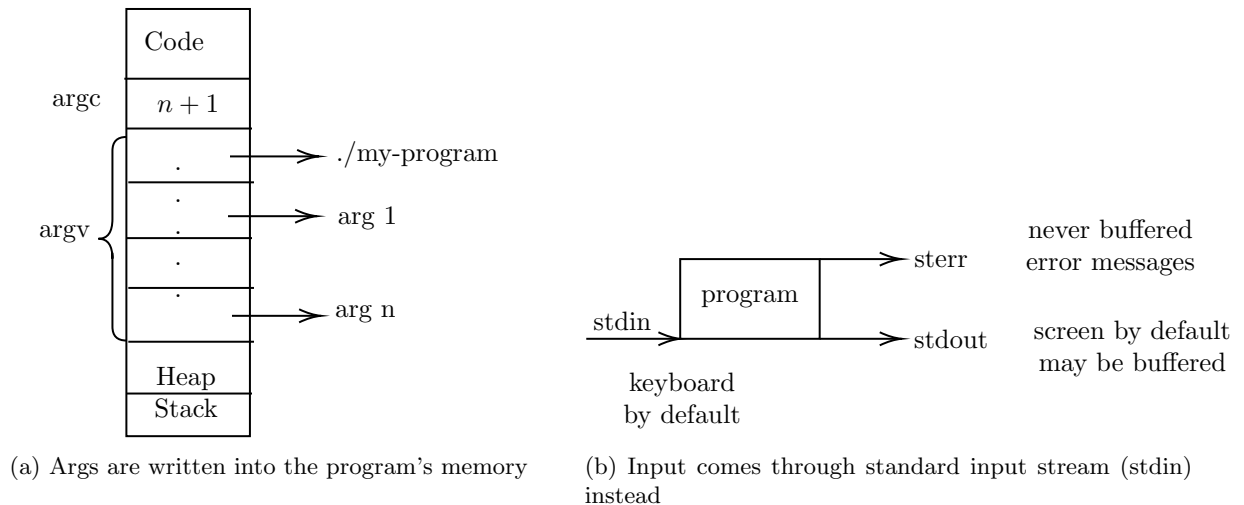


Figure 1.1: two figures

Observe: command-line args/input from stdin - 2 different programming techniques.

To compile: `gcc -std=c99 -Wall myprogram.c -o myprogram` translates C source to executable binary

- `./myprogram`
- `Wall` (warn all) best friend
- By default `a.out`
- This program - simplification of the Linux `cat` command
 - `cat file1 file2 file3`
cat opens file1, file2, file3 prints contents one after the other
 - `cat`
- echoes stdin
 - `cat < file1`
 - * `file1` used as a source for stdin
 - * the shell (not cat) opens the file
 - * displays `file1`

`^D` (ctrl-d) signals end-of-file

1.2 Consider C++

Can we write the cat program in cpp? -already valid cpp
The "cpp way" : command line args - same as C

1.2.1 stdin/stdout

```
#include <iostream>
int main() {
    int x, y;
    std::cin >> x >> y;
```

```

    std::cout << x+y << std::endl;
}

```

std::cin, std::cout, std::cerr - streams types.

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

- >> input operator cin >> x - populates x as a side-effect - returns cin
- << output operator cout << x+y - prints x+y as a side-effect - returns cout

1.2.2 File access

```

#include <fstream>
std::ifstream f{"name-of-file"}; // initialization syntax - also opens the file
//ofstream for output
char c;
while (f >> c) { // evaluates to f
    // - implicitly converted to bool
    // - true if the read succeeded - false if failed
    std::cout << c;
}

// fclose is called when f goes out of scope - automatic

```

Recall: File access (identical to above)

```

Input: the quick brown fox
Output: thequickbrownfox

```

Stem input skips whitespace (just like scanf)

To include whitespace:

```

std::ifstream f{"name-of-file"};
f >> std::noskipws; // io manipulator
char c;
while (f >> c) {
    std::cout << c;
}

```


Note(important):

- no explicit calls to `fopen/fclose`
- initializing `f` opens the file
- when `f`'s scope ends, file is closed

Let's try to write a cat in C++

```
#include <iostream>
#include <fstream>
using namespace std; // Avoids having to say std::
void echo(istream f) {
    char c;
    f >> noskipws;
    while (f >> c) cout << c;
}

int main (int argc, char *argv[]) {
    if (argc == 1) echo(cin);
    else for (int i = 1; i < argc; ++i) {
        ifstream f {argv[i]};
        echo(f);
    }
}
```

Doesn't work - won't even compile - Why?

- `cin` - `istream` - `echo` takes an `istream`
- `f` - `ifstream` - is `echo(f)` a type mismatch?

No! - this is actually fine. `ifstream` is a *subtype* of `istream`

Any `ifstream` can be treated as an `istream`.

- foundational concept in OOP
- details later

That's not error.

The error is - you can't write a function that takes an `istream` the way `echo` does.

Why not? - time for discussion.

Compare:

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

int x;
cin >> x;
```

C and C++ are pass-by-value languages - so `scanf` needs the address of `x`. in order to change `x`, via pointers.

So why not `cin >> &x`?

C++ has another ptr-like type.

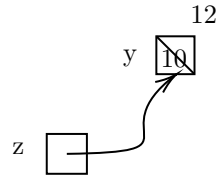


Figure 1.2: z cannot change

1.3 References

See this code in Figure 1.2

```
int y = 10;
int &z = y; // z is an lvalue reference to y
// similar to
int * const z = &y; // bute with auto-deferencing

z = 12; // NOT *z = 12;
// y now 12
int *p = &z; //gives the address of y
```

In all cases, z acts as if it were y.
z is an alias for y.

lvalue refs must be initialized, and must be initialized to something that has an address.

```
int &z = y; // OK
// int &x = 4 //wrong
// int &x = a+b; // wrong
```

- Cannot: - create a ptr a ref: `int &* x;` wrong
- Can create a ref to ptr: `int *&x = ...;` legal
- Can't create a ref to a ref: `int &&r = z;` wrong (will compile - means sth different)
- can't an array of refs: `int &r[3] = ...;` wrong

Can: pass as function parameters:

```
void inc(int &n) /* const ptr to the arg(x). Thus changes affect x */ {
    n = n + 1; // no ptr deref
}

int x = 5;
inc(x);

cout << x; // 6
```

`cin >> x` works because x is passed by ref.
`istream & operator>> (istream &in, int &x);`

Now consider `struct ReallyBig ...`

- `int f(ReallyBig rb) {...}`
note - don't have say `struct ReallyBig` in C++
pass-by-value - copies the struct (slow)
- `int g(ReallyBig &rb) {...}`
reference - no copy (fast)
but - could propagate changes to the caller
- `int h (const ReallyBig &rb) {...}`
fast - no copy- h can't change rb

Prefer pass-by-const-ref over pass-by-value for anything larger than a ptr, unless the function needs to make a copy anyway - then use pass-by-val.

Also `int f(int &n) {...}`, `int g(const int &n) {...}`

- `f(5)`; X - can't initialize an lvalue ref (n) to a literal value (5)
- `g(5)`; OK - since n can never be changed, the compiler allows this. (stored in a temporary location, so n has sth to point at.)

Back to `cat`

- f passed by value
- istream is copied
- copying streams is not allowed
- Works if you pass istream by ref: `void echo(istream &f) { /*as before*/ }`

To compile: `g++ -std=c++14 -Wall myprogram.c -o myprogram`
`./myprogram`

1.4 Separate Compilation

Now we put `echo` in its module

```
// echo.h
void echo(istream &f);
//echo.cc
#include "echo.h"
void echo (istream &f) {...}
//main.cc
#include <iostream>
#include <fstream>
#include "echo.h"
int main(...) {
    ... echo(cin); ...
    ... ifstream f{argv[i]};
        echo(f);
    ...
}
```

Compile separately:

- `g++14 echo.cc` fails - no main
- `g++14 main.cc` fails - no echo

These give us linking errors.

Correct:

- `g++14 -c echo.cc` creates `echo.o`
- `g++14 -c main.cc` creates `main.o`

These are object files.

Linker: `g++14 echo.o main.o -o myprogram`

Advantage only have to compile the parts you change to relink

- change `echo.cc` - recompile `echo.cc`
- change `main.cc` - recompile `main.cc`
- `echo.h` - must recompile both (which includes this header)

And relink.

Change `echo.cc` – recompile `echo.cc`

Change `main.cc` – recompile `main.cc`

Change `echo.h` – recompile both `main.cc` and `echo.cc`

because they include `echo.h`, and link

What if we don't remember what changed. Or what depends on what?

1.5 Linux make

Create a file called `Makefile`

Contents:

```
my-program: main.o echo.o
    g++ main.o echo.o -o myprogram
main.o: main.cc echo.h
    g++ -std=c++14 -Wall -c main.cc
echo.o: echo.cc echo.h
    g++ -std=c++14 -Wall -c echo.cc
PHONY:clean
clean:
    rm myprogram *.o
```

targets: dependencies

\t recipes for building targets from dependencies (Note, can't use aliases)

From the command line: `make` (builds the whole program) Change just `echo.cc`, `make` (recompile `echo.cc`, relink)

`make clean` (remove all binaries)

How? List a directory in "Long form":

```
$ ls -l
-rw-r-----1 j2smith j2smith 25 Sep9 15:27 echo.cc
```

type |owner/group/other permissions| owner | group |size|last modified date/time|name
based on last modified time.

Starting at the leaves of the dependency graph

If the dependency is newer than the target, rebuild the target ... and so on, up to the root target.

e.g. echo.cc newer than echo.o – rebuild echo.o

echo.o newer than myprogram – rebuild myprogram

Shoutcuts use variables

```
CXX = g++ # name of compiler
CXXFLAG = -std=c++14 -Wall # compiler options
EXEC = myprogram
OBJECTS = main.o echo.o

${EXEC}: ${OBJECTS}
    ${CXX} ${OBJECTS} -o ${EXEC}
main.o: main.cc echo.h
echo.o: echo.cc echo.h # omit recipes -make "guess" the right one.
PHONY:clean
clean:
    rm ${OBJECTS} ${EXEC}
#Writing dependencies still hard, but compiler can help.

g++14 -c -MMD echo.cc
# generates echo.o echo.d
cat echo.d
echo.o: echo.cc echo.h

#Just include .d file into Makefile

CXX = g++ # name of compiler
CXXFLAG = -std=c++14 -Wall -MMD # compiler options
EXEC = myprogram
OBJECTS = main.o echo.o
DEPENDS = ${OBJECTS:.o=.d}
${EXEC}: ${OBJECTS}
    ${CXX} ${OBJECTS} -o ${EXEC}
-include ${DEPENDS}
PHONY:clean
clean:
    rm ${OBJECTS} ${DEPENDS} ${EXEC}
```

Always use Makefile

Create the Makefile before you starting coding.

Problem 2: Linear Collection + Modularity

Linear Collection : linked lists + arrays

```
// node.h
struct Node {
    int data;
    Node *next;
};
size_t size(Node *n);

//node.cc
#include "node.h"

size_t size(Node *n){
    size_t count = 0;
    for(Node *cur = n;cur;cur = cur->next) ++count;
    return count;
}

// Note: do not use malloc / free in C++. Use new / delete
// main.cc
int main(){
    Node *n = new Node;
    n->data = 3;
    n->next = nullptr;
    Node *n2 = new Node {2, nullptr};
    Node *n3 = new Node {4, new Node{5, nullptr}};
    delete n;
    delete n2;
    // delete n3->next;
    // delete n3;
    while(n3){
        Node *tmp = n3;
        n3 = n3->next;
        delete tmp;
    }
}

// What happens if we do the following?
```

```
#include "node.h"
#include "node.h"
int main(){...}
```

Won't compile, struct is defined twice.
How can we prevent this?

2.1 C preprocessor

Transform the program before the compiler sees it.

`#include ...` drops the Contents of a file "right here"
Including old C headers - `#include <stdio.h>` become `#include<cstdio>`

`#define VAR VALUE`

- preprocessor variables
- all occurrences of VAR replaced with VALUE.

Eg

```
#define MAX 10
int x[MAX];
//OR
//----myprogram.cc-
int main(){
    int x[MAX];
    ...
}
```

`g++14 -D MAX=10 myprogram.called //make a #define on the cmd line`

```
// conditional compilation choose <= 1 of these to present to the compiler
#if SECURITYLEVEL == 1
short int
#elif SECURITYLEVEL == 2
long long int
#endif
public key;
```

special cases: `-industrial-strength` "comment out"

```
#if 0
...
#endif
```

However,

```

/*... */           // -----> doesn't nest
/*..      /*      /* ... */
| only here are commented out |

```

2.1.1 Fixing the double-include problem

include guard

```

// node.h
# ifndef NODE_H // true if NODE_H is not defined
# define NODE_H // value is the empty string
... file
# endif

```

First time `node.h` is included - symbol `NODE_H` is not defined, so file is subsequently - symbol is defined, so file is suppressed.

Always put `#include` guards in header file.

Never compile `.h` files and include `.cc` files

Now - What if someone writes?

```

// tree.h
struct Node {
    int data;
    Node *left, *right;
};
size_t size(Node *n); //size of the tree

```

Can't use both this and linked list in same program!

Solution: namespaces

```

// list.h:
namespace List {
    struct Node {
        int data;
        Node *next;
    };
    size_t size(Node *n);
} // no ; here

// tree.h
namespace Tree {
    struct Node {
        int data;
        Node *left, right;
    };
    size_t size(Node *n);
} // no ; here

```



```
// list.cc
#include "list.h"

size_t List::size(Node *n) {...} // scope resolution operator

// tree.cc
#include "tree.h"
namespace Tree {
    size_t size(Node *n) {...}
}

// main.cc
#include "list.h"
#include "tree.h"
int main() {
    List::Node *ln = new List::Node {1, nullptr};
    Tree::Node *tn = new Tree::Node {1, nullptr};
    ..
    delete ln;
    delete tn;
}
```

Namespaces are open - anyone can add any namespace:

```
// some-other-file.h
namespace List {
    int some_other_f();
    struct some_other_struct {...};
}
```

Exception adding members to namespace `std` is not permitted. (you will succeed, compiler will allow it) (undefined behaviour, probably mess up)

Problem 3: Linear Collection and Memory Management

Read §7.7.1 §14 §16.2

```
// Arrays:
int a[10]; // on the stack, fixed size
// on the heap
int *p = new int[10];
// TO delete
delete [] p;
```

Use `new` with `delete`, use `new[]` with `delete[]`.
Mismatching these = undefined behaviour.

Problem what if our array isn't big enough?

Note no `realloc` for `new/delete`

Solution Use abstraction to solve the problem

```
#ifndef VECTOR_H
#define VECTOR_H
namespace CS246E {
    struct vector {
        int * thevector;
        size_t size, cap;
    };
    const size_t startsize = 1;
    vector makeVector();
    size_t size(const vector &v);
    int &itemAt(const vector &v, size_t i);
    // return a reference
    void push_back(vector &v, int n);
    void pop_back(vector &v);
    void dispose(vector &v);
}
#endif
//vector.cc
#include "vector.h"
```

```

namespace { //equivalent to static
    void increaseCap(CS246E::vector &v) {
        if (v.size == v.cap) {
            int newVec = new int[2*v.cap];
            for (size_t t=0; t<v.cap; ++t)
                newVec[t]=v.theVector[t];
            delete[] v.theVector;
            v.theVector = newVec;
            v.cap *= 2;
        }
    }
}

CS246E::vector CS246E::makeVector(){
    vector v { new int[startsize], 0, 1};
    return v;
}

size_t CS246E::size(const vector &v) {return v.size;}
int CS246E::itemAt(const Vector &v, size_t i){
    return v.theVector[i];
}
void CS246E::push_back(vector &v, int n) {
    increaseCap();
    v.theVector[v.size++]=n;
}
void CS246E::pop_back(vector &v){if(v.size>0) --v.size;}
void CS246E::dispose(vector &v) {delete[] v.theVector;}

// main.cc
#include "vector.h"
using CS246E::Vector;
int main() {
    Vector v = CS246E::makeVector(); // no arguments here
    push_back(v,1);
    push_back(v,10);
    push_back(v,100);
    // can add as many items as we want - never need to worry about allocation
    itemAt(v,0)=2;
    dispose(v);
}

```

Question why don't we have to say `CS246E::push_back`, `CS246E::itemAt`, `CS246E::dispose`?

Answer Argument-Dependent Lookup (ADL) - also called Koenig Lookup

If the type of the function `f`'s argument belongs to a namespace `n`, then C++ will search namespace `n`, as well as the current scope, for a function matching `f`.

Since `v`'s type (`Vector`) belongs to namespace `CS246E`, C++ will search namespace `CS246E` for `push_back`, `itemAt`, `dispose`

This is the same reason we can say `std::cout<<x`, rather than `std::operator<<(std::cout, x)`;

Problems

- what if we forget to call `makeVector`?
uninitialed object
- what if we forget to call `dispose`?
memory leak?

How can we make this more robust?

3.1 Introduction to Classes

First concept in OOP - functions inside of structs

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

- structures that can contain functions - classes
- functions inside structs - called member functions or methods
- instances of a class - called objects

```
Student bob {90,70,80};
// class object
cout << bob.grade(); // method
```

What do `assns`, `mt`, `final` mean within `grade() {...}`

- fields of the current object - the receiver of the method call, e.g. `bob`

Formally - methods differ from functions in that methods take an implicit parameter called `this` that is a pointer to the receiver object.

`bob.grade()`; → `this` == `&bob`

Could have written (equivalent):

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

3.1.1 Initializing Objects

```
Student Bob {90, 70, 80};
// c-style struct initialization.
// field-by-filed
// Ok, but limited.
```

Better - initialization method - a constructor

```
struct Student {
    int assns, mt, final;
    Student(int assns, int mt, int final) {
        this->assns = assns;
        this->mt = mt;
        this->final = final;
    }
};
Student bob {90, 70, 80};
// Now calls the ctor with args 90,70,80
```

Note once a ctor is defined, c-style field-by-field initialization is no longer available.

Equivalent `Student bob = Student {90,70,80};`

Heap `Student *p = new Student {90,70,80};`

Advantages of ctors

- automatically called
- default parameters
- overloading
- sanity checks

Eg

```
struct Student {
    ...
    Student (int assns=0, int mt=0, int final=0) { ... }
};
Student laura {70}; // 70,0,0
Student newkid; // 0,0,0
```

Note Every class comes with a built-in default (zero-arg) ctor

```
Node n; // calls default ctor
// - default constructs all fields that are objects
// - does nothing in this case
// goes away if you provide any ctor
```

```
// EG:
struct Node {
    int data;
    Node *next;
    Node (int data, Node *next = nullptr) {
        this->data = data; // etc
    }
};
Node n {3}; // 3, nullptr
// Node n; // wrong - no default ctor
```

Object creation protocol when an object is created
4 steps

1. space is allocated
2. later
3. Fields constructed in declaration order
(fields ctors are called for fields that are objects)
4. ctor body runs

Field initialization should happen in step3, but ctor body is in step4.

Consequence object fields could be initialized twice

```
// EG
#include <string>
struct Student {
    string name; //object
    Student(const string &name) {
        this->name = name;
    }
};
Student mike {"mike"};
// name default-initialized("") in step3 then reassigned in step4
```

To fix: Member Initialization List (MIL)

3.1.2 MIL

```
struct Student {
    ...
    Student(const string &name, int assns, int mt, int final):
name{name}, assns{assns}, // must be field names - before{}
mt{mt}, final{final} //step3
    // inside {} following usual scoping rules
    {} // Step4
};
```

MIL Must be used for fields that are

- constants
- references
- objects without a default ctor
- **should be used as much as possible**

Careful single arg ctors

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

Single-arg ctors create implicit conversions

```
Node n{4}; // OK
Node n=4; // OK - int implicitly converted to Node
void f(Node n);
f(4); // OK - maybe trouble
f(x);
```

To fix

```
struct Node {
    ...
    explicit Node (...):...{} // disable the implicit conversion
};
Node n{4}; //OK
// Node n=4;
// f(4);
f(Node{4});
```

Object Destruction

- a method called the destructor (dctor) runs automatically
- built-in dctor - calls dtors on all fields that are objects

Object destruction protocol 4 steps

1. dctor body runs
2. fields destruction (dtors called on fields that are objects) in reverse declaration order
3. later
4. space deallocated

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

What does the built-in dtor do?

- nothing - neither fields is an object

```
Node *n = new Node {3, new Node{4,new Node {5,nullptr}}};
delete n; // only deletes the first node
//writing our own dtor:
struct Node {
    ...
    ~Node() { delete next; }
};
delete n; // now frees the whole list
```

Also

```
{
Node n{1,new Node {2, nullptr}};
// stack ----->|
// |heap-----|
} // scope of n ends; n's dtor will run; whole list is freed
```

Objects

- a ctor always runs when they are created.
- a dtor always runs when they are destroyed

Back to vector problem - turn into a class:

```
//vector.h
#ifndef VECTOR_H
#define VECTOR_H
namespace CS246E {
    struct Vector {
        size_t n,cap;
        int *theVector;
        Vector();
        size_t size();
        int &itemAt(size_t i);
        void push_back(int n);
        void pop_back();
        ~Vector();
    };
}
#endif

//vector.cc
```



```
namespace {
    void increaseCap (const Vector &v) {...}
}

CS246E::Vector::Vector(): tsize{0}, cap{start_size}, theVector{new int[cap]} {}
size_t CS246E::Vector::size() {return n;}
CS246E::Vector::~~Vector() {delete[] theVector;}

// main.cc
int main() {
    Vector v; // default ctor automatically called - no makeVector
    v.push_back(1);
    v.push_back(10);
    v.push_back(100);
    v.itemAt(0)=2;
    // no dispose - dtor cleans v up
}
```

4

Problem 4: Copies

```

Vector v;
v.push_back(100);
...
Vector w=v; // Allowed. Constructs w as a copy of v
w.itemAt(0); //100
w.itemAt(0) = 200;
v.itemAt(0);
// 200 Shallow copy - basis field-by-field copying
//                               - v and w share data
Vector w = v;
// constructs w as a copy of v
// involves the copy ctor

struct Vector {
    ...
    Vector (const Vector &other); // copy ctor
}; // compiler-supplied-copy ctor - copies all fields (shallow copy)

```

If you want a deep copy - write your own copy ctor

```

struct Node {
    int data;
    Node *next;
    ...
    Node (const Node &other): data{other.data},
        next{other.next? new Node{*other.next}: nullptr}
    {}
};

Vector v;
Vector w;
w = v; // copy, but not a constructor
// copy assignment operator
// compiler - supplied - copies each field (shallow)
// compiler-supplied-copies each field (shallow)
// - leaks w's old data

```

Deep copy assignment:

```
struct Node { // vector-exercise
    int data; Node *next;
    ...
    Node &operator=(const Node &other) {
        data = other.data;
        delete next;
        next = other.next? new Node{*other.next}: nullptr;
        return *this;
    }
}; // wrong - dangerous!
```

Consider: Node n{...}; n=n;

Destroys n's data and then tries to copy it.

Must always ensure that `operator=` works in the cases of self-assignment.

```
Node &Node::operator=(const Node &other) {
    if (this == &other) return *this;
    // as before
}
```

Alternative: copy and swap idiom

4.1 Copy and Swap Idiom

```
#include <utility>
struct Node {
    ...
    void swap (Node &other){
        using std::swap;
        swap(data,other.data);
        swap(next,other.next);
    }
    Node &operator=(const Node &other) {
        Node tmp=other;
        swap(tmp);
        return *this;
    }
};
// drawback: always takes a copy
```

Problem 5: Moves

Consider

```
Node plusOne(Node n) {
    for (Node *p = &n; p; p=p->next) ++p->data;
    return n;
}
Node n {1, new Node{2,nullptr}};
Node m = plusOne(n); // copy ctor
                    // but what is "other" here? reference to what?
```

- temporary object → created to hold the result of `plusOne`.
- `other` is a reference to this temporary
 - copy ctor deep-copies the temporary into `m`

But

- the temporary is just going to be discarded as soon as the start `Node m = plusOne(n);` is done
- wasteful to deep copy the temp
- why not just steal the data instead?
 - save the cost of a copy
- Need to be able to tell whether `other` is a ref to a temporary object or a standalone object.

Rvalue references `Node &&` is a reference to a temporary object (rvalue) of type `Node`

```
// version of the ctor that takes a Node &&
struct Node {
    ...
    Node (Node &&other); // called a move ctor
};
Node::Node (Node &&other): data{other.data}, next{other.next}
    {other.next=nullptr;}
```

Similarly `Node m; m = plusOne(n);` assignment from temporary

```
// More assignment operator
struct Node {
    Node &operator=(Node &&other) {
        // steal other's data
        // destroy my old data
        // easy - swap without copy
        using std::swap;
        swap(data, other.data);
        swap(next, other.next);
        return *this;
    }
};
```

Can combine copy/move assignment

```
struct Node {
    ...
    // pass-by-value invokes copy ctor if arg is an lvalue
    // invokes move ctor (if there is one) if arg is an rvalue
    Node &operator=(Node other){
        // unified assignment operator
        swap(other); //copies only if the arg is an lvalue
    }
};
```

Note copy and swap can be expensive - hand-coded operations may do less copying.

But now consider

```
struct Student {
    string name;
    Student (const string &name): name{name} {}
    // copies arg into the field (uses string's copy ctor)
};
// What if name points to an rvalue?
struct Student {
    string name;
    Student(std::string name): // copies if arg was an lvalue. Moves if arg was an
    rvalue
    name{name/*another copy*/}{}
    // name may come from an rvalue, but name is an lvalue
};
struct Student {
    string name;
    Student(string name):
        name{std::move(name)} // now a move construction
    //std::move - forces a value to be treated as an rvalue
    {}
};
struct Student {
```

```

Student(const Student &&other): // move ctor
    name {other.name}{}
Student &operator=(Student other) {
    // unified assignment
    name = std::move(other.name);
    return *this;
}
};

```

If you don't define move operations, copy operations will be used.
If you do define them, they will replace copy operations when the arg is a temporary (rvalue).

5.1 Copy/Move Elision

```

Vector makeAVector() { return Vector {};} // basic ctor
Vector v = makeAVector(); // move ctor? copy ctor?

```

Try with g++: -just the basic ctor - no copy/move In some circumstances, the compiler is allowed to skip calling copy/move ctors (but doesn't have to)

`makeAVector()` writes its result directly into the space occupied by `V` in the caller, rather than copy it over.

```

//Eg
Vector v/*copy ctor*/ = Vector/* basic ctor */{};
// Formally a basic construction + a copy construction

Student bob {90,70,80};
Student bob = Student{90,70,80};
//But the compiler is required to elide this copy construction, so basic ctor only

//EG
void doSomething(Vector v) {...} // pass-by-value
// ->copy/move ctor
doSomething(makeAVector());
// result of makeAVector written directly into the parameter
// no copy

```

This is allowed, even if dropping ctor calls would change the behaviour of the program. (e.g. if the ctor print sth)

If you need all ctors to run: `g++11 -fno-elide-constructors` can slow your program considerably

In summary Rule of 5 (Big 5)

If you need to customize any one of

1. Copy constructor
2. Copy assignment
3. Destructor

4. Move constructor

5. Move assignment

then you usually need to customize all 5.

From now on, we will assume that Node and Vector have the Big 5 defined.

6

Problem 6: I want a constant vector

Say we want to print a vector:

```
ostream &operator <<(ostream &out, const Vector &v){
    for (size_t i=0; i<v.size(); ++i){
        out << v.itemAt(i) << " ";
    }
    return out;
}
```

Won't compile! - can't call `size_t` and `itemAt` on a `const` object - What if these methods change fields?
Since they don't - declare them `const`!

```
struct Vector {
    size_t size() const;
    int &itemAt(size_t i) const;
    ...
};
size_t Vector::size() const {return n;}
int &Vector::itemAt(size_t i) const {return the Vector[i];}
```

Now the loop will work.

But

```
void f(const vector &v) {
    v.itemAt(0) = 4;    // Works!! v not very const...
}
```

- `v` is a `const` object - cannot change `n`, `cap`, `theVector` (ptr)
- You can changed items pointed to by `theVector`

Can we fix this?

```
struct Vector {
    ...
    const int &itemAt(size_t i) const {
```



```

    return theVector[i];
}
};

```

Now `v.itemAt(0)=4` won't compile if `v` is `const`, but it also won't compile if `v` isn't `const`.

To fix: `const` overloading

```

struct Vector {
    ...
    const int &itemAt(size_t i) const;
    // Will be called if the object is const
    int &itemAt(size_t i);
    // Will be called if object is non-const
};

inline const int &Vector::itemAt(size_t i) const { return theVector[i]; }
inline int &Vector::itemAt(size_t i) { return theVector[i]; }
// (inline) suggests replacing the function call with the function body- saves the cost
// of function call

```

More idomatic

```

struct Vector {
    ...
    const int &operator[] (size_t i) const { return theVector[i]; }
    int &operator[] (size_t i) { return theVector[i]; }
    // method body inside class - implicitly declares the method inline
};

```

Problem 7: Tampering

```
Vector v;
v.cap = 100; // sets cap without allocating memory
v.push_back(1);
v.push_back(2); // undefined behaviour - may crash
```

Interfering with ADTs:

1. Forgery
 - creating an object without calling a ctor function
 - not possible once we wrote ctors
2. Tampering
 - accessing the intervals without using a provided interface function

What's big deal? - Invariants → statement will always be true about an abstraction
ADTs provide and rely on invariants

- -stacks- provide the invariant that the last item pushed is the first item popped.
- -vectors- rely on an invariant that elements 0 ... cap-1 are valid to access

Can't rely on invariants if the user can interfere - makes the program hard to reason about.

Fix encapsulation - sealing objects into "black boxes"

```
namespace CS246E {
  struct Vector {
  private:
    size_t n, cap;
    int *theVector;
    // visible only within the Vector class
  public:
    Vector();
    size_t size() const;
    void push_back(int n);
    ...
    // visible to all
  };
}
```

```

};
// If no access specifier given: public
}
// vector.cc
#include "vector.h"
namespace {
    void increaseCap(Vector &v){...}
}
// doesn't work anymore! Doesn't have access to v's internals

```

Try again

```

// vector.h
namespace CS246E {
    struct Vector {
        private:
            size_t n, cap;
            int *theVector;
        public:
            Vector();
            ...
            private:
                void increaseCap(); // now a private method
            // don't need anonymous namespace anymore
    };
}

```

structs

- public default access
- private default access would be better
- class - exactly like struct, except private default access

```

class Vector {
    size_t n, cap;
    int *theVector;
public:
    Vector();
private:
    void increaseCap();
};
// vector.cc same

```

```

Node n {3, nullptr}; // stack-allocated
Node m {4, &n}; // dtor for m will try to delete &n. Undefined Behaviour

```

There was an invariant - `next` is either `nullptr`, or was allocated by `new`.

How can we enforce this?

Encapsulate `Node` inside inside a "wrapper" class

```
class list {
    struct Node {
        int data;
        Node *next;
        // ... methods
    };
    // private nested class
    // - list available outside class List
    Node *theList;
    size_t n;

public:
    list(): theList{nullptr}, n{0} {}
    ~list() { delete theList; }
    size_t size() const {return n;}
    void push_front (int x) {
        theList = new Node {x, theList};
        ++n;
    }
    void pop_front() {
        if (theList) {
            Node *tmp = theList;
            theList = theList->next;
            tmp->next = nullptr;
            delete tmp;
        }
    }
    const int &operator[] (size_t i) const {
        Node *cur = theList;
        for (size_t j = 0; j < i; cur = cur->next);
        return cur->data;
    }
    int &operator[] (size_t i) {...}
};
```

Client cannot manipulate the list directly - no access to next ptrs - invariant is maintained.

Problem 8: Efficient Iteration

```

Vector v;
v.push_back();
//...
for (size_t i = 0; i < v.size(); ++i) {
    cout << v[i] << "\n"; // O(1)
}
// Array access - efficient - O(n) traversal

list l;
l.push_front(..);
// ..
for (size_t i = 0; i < l.size(); ++i) {
    cout << l[i] << "\n"; // O(i)
}
// O(n^2) traversal

```

No direct access to "next" ptrs - how can we do efficient iteration?

Design patterns (more later)

- Well-known solutions to well-studied problems
- Adapted to suit-current needs

8.1 Iterator Pattern

For efficient iteration over a collection without exposing underlying structure.

Idea Create a class that "remembers" where you are in the list -abstraction of a ptr
Implementation: C

```

for (int *p = arr; p != a + size; ++p) printf("%d\n", *p);

```

```

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

```

Now

```

list l;
...
for (list::iterator it = l.begin(); it != l.end(); ++it) {
    cout << *it << "\n";
} // O(n)

```

Question Should `list::begin`, `list::end` be `const` methods?

Consider

```

ostream &operator<<(ostream &out, const list &l) {
    for (list::iterator it = l.begin(); it != end(); ++it) {
        out << *it << '\n';
    }
    return out;
} // won't compile if begin end are not const

```

If `begin/end` are `const`

```

ostream &operator<<(ostream &out, const list &l) {
    for (list::iterator it = l.begin(); it != end(); ++it) {
        out << *it << '\n';
        ++*it; // increment items in the list
    }
    return out;
}

```

- will compile
 - shouldn't - the list is supposed to be const
 - compiles because `operator*` returns a non-const ref
- should compile if `l` is non-const

Conclusion

- iteration over `const` is different from iteration over non-const
- make a second iterator class

```
class list {
    ...
public:
    class iterator {
        // exactly as befor
        int &operator*() const;
    };
    class const_iterator {
        Node *p;
    public:
        const_iterator(Node *p): p{p} {}
        bool operator!=(const const_iterator &other) const;
        const_iterator &operator++();
        const int &operator*() const { return p->data; } // const at the front
    };

    iterator begin() { return iterator{theList}; }
    iterator end() { return iterator{nullptr}; }

    const_iterator begin() const { return const_iterator{theList}; }
    const_iterator end() const { return const_iterator{nullptr}; }
};

ostream &operation<<(ostream &out, const list &l) {
    for (list::const_iterator it = l.begin(); it != l.end(); ++it) {
        out << *it << '␣';
    }
    return out; // works
}
list::const_iterator it != l.begin; // mouthful
```

We got a shorter version!!!

```
ostream &operation<<(ostream &out, const list &l) {
    for (auto it = l.begin(); it != l.end(); ++it) {
        out << *it << '␣';
    }
    return out;
}
```

```
auto x = expr;
```

- saves writing x's type
- x is given the same type as expr

Emm, shorter...

```
ostream &operation<<(ostream &out, const list &l) {
    for (auto n : l) {
        out << n << '␣';
    }
    return out; // Range-based for loop
}
```

Range-based for loop requires

- methods (or functions) begin/end that return an iterator object
- the iterator class must support unary*, prefix++, !=

Note

```
for (auto n : l) ++n;
// n is declared by value, so ++n increments n, not the list items
for (auto &n : l) ++n;
// n is a ref - *will* update list items
for (const auto &n : l) ...
// n by ref (no copy) - cannot be modified
```

One small encapsulation problem...

Question What if client do `list::iterator it {nullptr};`
 - forgery - client can create an end iterator without calling `l.end()`

To fix: make iterator ctors **private**

But then: list can't create iterators either

Solution: friendship

```
class list {
    ...
public:
    class iterator {
        ...
        iterator(Node *p);
    public:
        ...
        friend class list;
    };
    class const_iterator {
        ...
        const_iterator(Node *p);
    public:
        ...
        friend class list;
    };
};
```



```
};  
...  
};
```

Limit friendships: - weaken encapsulation

Can do the same for vectors

```
class Vector {  
...  
public:  
    class iterator {...};  
    class const_iterator {...};  
...  
};
```

OR

```
class Vector {  
    size_t n, cap;  
    int *theVector;  
...  
public:  
    typedef int *iterator;  
    typedef const int *const_iterator;  
    iterator begin() { return iterator {theVector}; }  
    iterator end() { return iterator {theVector+n}; }  
    const_iterator begin() const { return const_iterator {theVector}; }  
    const_iterator end() const { return const_iterator {theVector+n}; }  
};
```

Problem 9: Staying in Bounds

```
Vector v;
v.push_back(2);
v.push_back(3);
v[2]; // Out of bounds! --- undefined behaviour - may be or may not crash
```

Can we make this safer?
Adding bounds checking to `operator[]` would be expensive.

Second, safer fetch operation:

```
class Vector {
...
public:
    int &at(size_t i) {
        if (i < n) return a[i];
        else // ... what???
    }
    ...
};
```

`v.at(2)`

- still not valid - what should happen?
- Returning any `int` - looks like a non-error
- Returning a non-`int` - not type-correct
- Crash the program - can we do better?

Don't return, don't crash

Solution raise an exception

```
class range_error {};
class Vector {
...
};
```

```

public:
    int &at(size_t i) {
        if (i < n) {...}
        else throw range_error {};
        // construct an object of type range_error and "throw" it
    }
};

```

Client options:

1) do nothing

```

Vector v;
v.push_back(1);
v.at(1); // the expression will crash the program

```

2) catch it

```

try {
    Vector v;
    v.push_back(1);
    v.at(1);
}
catch (range_error &r) { // catch by reference. - saves a copy operation
                        // r is the thrown object

    // do something
}

```

3) let your caller catch it

```

int f() {
    Vector v;
    v.push_back(1);
    v.at(1);
    return 0;
}
int g() {
    try {
        f();
    }
    catch (range_error &r) {
        // do sth
    }
}

```

Expression will propagate back through the call chain until a handler is found.

- called unwinding the stack
- if non handler found, program aborts
- control resumes after the catch block (problem code is not retried)

Question What happens if a ctor throws an expression?

- object is considered partially constructed.
- dtor will not run on partially constructed objects.

Eg

```
class C {...};
class D {
    C a;
    C b;
    int *c;
public:
    D () {
        c = new int[10];
        ...
        if (...) throw sth; // *
    }
    ~D () { delete[] c; }
};
D d;
```

At *, the D object is not fully constructed, so ~D() will not run on d.

But

- a & b are fully constructed, so their dtors will run
- So if a ctor wants to throw, it must clean up after itself

```
D () {
    c = new int[10];
    ...
    if (...) {
        delete[] c;
        throw something;
    }
}
```

Question What happens if a dtor throws? Trouble

- By default - program aborts immediately - `std::terminate` is called
- If you really want a throwing dtor - tag it with `noexcept(false)`
- but watch out

```
class myExn {};
class C {
    ...
    ~C() noexcept(false) {
        throw myExn {};;
    }
};
```

```
void h() { C c1; } // dtor for c1 throws at the end of h
void g() {
    C c2;
    h();
    // - unwinding through g
    // - dtor for c2 throws as we leave g
}
void f() { // no handler yet
    try { g(); }
    catch (myExn &e) /* handler */ {
        ...
    }
}
// now 2 unhandled expression
// IMMEDIATE termination guaranteed (std::terminate)
```

Never let dtors throw. Swallow the expressions if necessary
Also note: you can throw any value - not just objects

10

Problem 10: I want a vector of chars

Start over? **NO!!!!**

Introduce a major abstraction mechanism - templates - generalize over types

```
template <typename T> class vector {
    size_t n, cap;
    T *theVector;
public:
    vector ();
    ...
    void push_back(T n);
    T &operator[] (size_t i);
    const T&operator[] const (size_t i);
    typedef T *iterator;
    typedef const T *const_iterator;
    ...
};
template <typename T> vector<T>::vector():
    n{0}, cap{1}, theVector{new T[cap]} {}

template <typename T> void vector<T>::push_back(T n) { ... }
// etc
```

Note must put the implementation in the .h file

```
// main.cc
int main() {
    vector <int> v; // vector of ints
    v.push_back(1);
    ...
    vector <char> w; // vector of chars
    w.push_back('a');
    ...
}
```

Semantics The first time the compiler encounters `vector<int>`, it creates a version of the vector class, where `int` replaces `T`, and then compiles the new class.

- can't do that unless it knows all the details about the class
- so implementation needs to be available, i.e. , in `.h`
- can also write method bodies in line.

11

Problem 11: Better initialization

Long sequence of `push_back` s is clunky.

```
// Arrays
int a[] = {1, 2, 3, 4, 5}; // :)
// Vector
vector<int> v;
v.push_back(1);
... // :(
```

Goal: better initialization

```
template<typename T> class Vector {
...
public:
    Vector() { ... }
    Vector(size_t n, T i = T {}):
        n{n}, cap{n==0 ? 1 : n}, theVector{ new T[cap] } {
        for (size_t j = 0; j < n; ++j) theVector[j] = i;
        }
    ...
};

// Now
Vector <int> v; // empty
Vector <int> {5}; // 0 0 0 0 0
Vector <int> v {3 , 5}; // 5 5 5
```

Note: `T {}` (default ctor) means 0 if `T` is a built-in type
Better - but what about true array-style initialization.

```
#include<initializer_list>
template <typename T> class Vector {
...
public:
    ...
    Vector() {...}
    Vector(size_t n, T i = T{}): ...
```



```

Vector(std::initializer_list<T> init):
    n{init.size()}, cap{n==0 ? 1 : n}, theVector{ new T[cap] } {
        size_t i = 0;
        for (auto i: init) theVector[i++] = t;
    }
};

Vector<int> v {1 , 2, 3, 4, 5}; //1 2 3 4 5
Vector<int> v; // empty (default)
Vector<int> v {5}; // 5
Vector<int> v {3, 5}; // 3 5
// Oops - other constructor not being called

```

Default ctors take precedence over `initializer_list` ctors, which take precedence over all other ctors.

To get the other ctors to run - need round bracket initialization

```

Vector<int> v (5); // 0 0 0 0 0
Vector<int> v (3, 5); // 5 5 5

```

A note on cost: items in an `initializer_list` are stored in contiguous memory. (`begin` method returns a ptr)

So we are using one array to build another - 2 copies in memory

Note

- `initializer_lists` are meant to be immutable.
- do not try to modify their contents
- do not use them as standalone data structures

But also note, there is only one allocation in `Vector` - No doubling and reallocating, as there was with a sequence of `push_backs`

In general - if you know how big your `vector` will be, you can save reallocation cost by requesting the memory up front:

```

template <typename T> class Vector {
...
public:
...
    Vector() {...}
...
    void reserve(suze_t newCap) {
        if (cap < newCap) {
            T *newVec = new T[newCap];
            for (size_t i = 0l i < n; ++i) newVec[i] = theVector[i];
            delete[] theVector;
            theVector = newVec;
            cap = newCap;
        }
    }
};

```

Exercise rewrite `Vector` so that `push_back` uses `reserve` instead of `increaseCap`

```
Vector <int> v;  
v.reserve(10);  
v.push_back(...);  
...  
// can do 10 push_back without needing to reallocate
```

12

Problem 12: I want a vector of Posns

```
struct Posn {
    int x, y;
    Posn (int x, int y): x{x}, y{y} {}
};
int main() {
    Vector <Posn> v; // Won't compile! Why not?
}
```

Take a look at Vector's ctor:

```
template<typename T> vector<T>::Vector():n{0},cap{1},theVector{new T {}} {}
```

creates an array of T objects

Which T object(s) will be stored in array?

C++ always calls a ctor when creating an object.

- Which ctor gets called? Default ctor T {}
- Posn doesn't have one

How can we create a vector of Posns?

Need to separate memory allocation (object creation step 1) from object initialization (step 2 - 4)

Allocation void *operator new(size_t)

- allocates size_t bytes - no initialization.
- returns a void *
- Note:
 - C: void* implicitly converts to any ptr type.
 - C++: the construction requires a cast

Initialization "placement new" `new(addr) type`

- constructs a "type" object at 'addr'
- does not allocate memory

```

template <typename T> class Vector {
...
public:
...
Vector(): n{0}, cap{1}, theVector{static_cast<T*>(operator new(sizeof(T)))} {}
Vector(size_t n, T x): n{n}, cap{n==0? 1 : n},
    theVector{static_cast<T*>(operator new(cap*sizeof(T)))} {
    for (size_t i = 0; i < n; ++i) new (theVector+i) T(x);
}
...
void push_back(T x) {
    increaseCap();
    new (theVector +(n++)) T(x);
}
void pop_back() {
    if (n) {
        theVector[n-1].~T(); // Must explicitly invoke dtor.
        --n;
    }
}
~Vector() {
    destroyItems();
    operator delete (theVector);
}
void destroyItems() {
    for (T *p=theVector; p != theVector+n; ++p) p->~T();
    n=0;
}
};

```

13

Problem 13: Less Copying!

Before: `void push_back(int n);`
Now:

```
void push_back(T x)/*1*/ { // If T is an object, how many times is x being copied?  
    increaseCap();  
    new(theVector + n++) T(x); /* 2 */  
}
```

If arg is an lvalue: ① is copy ctor ② is copy ctor 2 copies - want 1

If arg is an rvalue: ① is a move ctor ② is a copy ctor 1 copies - want 0

Fix

```
void push_back(T x) {  
    increaseCap();  
    new (theVector + n++) T (std::move(x));  
}
```

lvalue: copy + move

rvalue: move + move

If T doesn't have a move ctor: 2 copies

Better: take T by reference:

```
void push_back(const T &x) { // no copy, no move  
    increaseCap();  
    new (theVector + n++) T(x); // copy ctor  
}  
  
void push_back(T &&x) { // no copy, no move  
    increaseCap();  
    new (theVector + n++) T(std::move(x)); // move ctor  
}
```

lvalue: 1 copy rvalue 1 move

If T has no move ctor: 1 copy

Now consider

```
vector <Posn> v;
v.push_back(Posn{3,4});
```

- (1) ctor call to create the object
- (2) copy or move constructor into the vector (depending on whether Posn has a move ctor)
- (3) dtor call on the temp object

Could eliminate (1) and (3) if we could get vector to create the object, instead of the client

- pass ctor args to the vector, not the actual object

A note on template functions:

Consider: `std::swap` - seems to work on all types

Implementation:

```
template<typename T> void swap(T &a, T&b){
    T tmp(std::move(a));
    a = std::move(b);
    b = std::move(tmp);
}

int x = 1, y = 2;
swap(x, y); // Equiv: swap<int>(x, y);
```

Don't have to say `swap<int>`. C++ can deduce this from the types of `x` and `y`

In general, only have to say `f<T>(...)` if `T` cannot be deduced from args.

Type deduction for template args follows the same rules as type deduction for `auto`

Back to vectors - passing ctor args

- we don't know what types ctor args should have
- `T` could be any class, could have several ctors

Idea member template function - like `swap`, it could take anything

Second problem how many ctor args?

Solution variadic templates

```
template<typename T> class Vector {
//...
public:
//...
    template <typename... Args>
    void emplace_back (Args... args) {
        increaseCap();
        new (theVector + n++) T(args...);
    }
};
```

Args is a sequence of type variables denoting the types of the actual arguments of `emplace_back`
 args is a sequence of program variables denoting the actual args of `emplace_back`

```
vector<Posn> v;
v.emplace_back(3, 4);
```

Problem args is taken by value
 Can we take args by reference?
 lvalue or rvalue reference?

```
template<typename... Args> void emplace_back(Arg &&... args) {
    // ...
}
```

Special rules here: `Args &&` is a universal reference

- can point to an lvalue or an rvalue

When is a reference universal? Must have the form `T &&`, where `T` is the type arg being deduced for the current template function call.

Examples

```
template<typename T> class C {
public:
    template<typename U> void f (u &&x); // universal
    template<typename U> void g (const U &&x); // not universal, just rvalue
    void h (T &&x); // not universal, T is not being deduced
```

Now recall

```
class C {...};
void f(C &&x) { // rvalue ref x points to an rvalue, but x is an lvalue
    g(x); // x is passed as an lvalue to g
}
```

If you want to preserve the fact that `x` is an rvalue ref, so that a "move" version of `g` is called (assuming there is one):

```
void f(C &&x) {
    g(std::move(x));
}
```

In the cases of args - don't know if args are lvalues, rvalues, or a mix. Want to call `move` on args iff the args are rvalues

```
template <typename... Args> void emplace_back(Args &&... args) {
    increaseCap();
    new (theVector + n++) T (std::forward<Args>(args) ...);
}
```

`std::forward`

- calls `std::move` if its argument is an rvalue ref
- else does nothing

Now `args` is passed to T's ctor with lvalue/rvalue information preserved - called perfect forwarding.

14

Problem 14: Memory Management is Hard!

No it isn't.

Vectors

- can do everything arrays can
- grow as needed - $O(1)$ amortized time or better
- clean up automatically when they go out of scope
- are tuned to minimize copying

Just use vectors, and you'll never have to manage arrays again.

C++ has enough abstraction facilities to make programming easier than C

But what about single objects?

```
void f() {
    Posn *p = new Posn {1, 2};
    ...
    delete p; // must deallocate the Posn!
```

First ask: Do you really need to use the heap? Could you have used the stack instead?

```
void f() {
    Posn p {1, 2};
    ...
} // no cleanup necessary
```

Sometimes you do need the heap. Calling `delete` isn't so bad. But consider:

```
class BadNews {};
void f() {
    Posn *p = new Posn {1, 2};
    if (some condition) { throw BadNews{}; }
    delete p;
}
```

`p` is leaked if `f` throws. Unacceptable.

Raising and handling an exception should not corrupt the program - we desire exception safety

Leaks are a corruption of the program's heap - will eventually degrade performance and crash.

If a program cannot recover from an exception without corrupting its memory, what's the point of recovering?

What's constitutes exception safety? 3 levels:

- 1) Basis guarantee - once an exception has been handled, the program is in some valid state. No leaked memory, no corrupted data structure, all invariants maintained.
- 2) Strong guarantee - if an exception propagates outside a function `f`, then the state of the program will be as if `f` had not been called. `f` either succeeds completely or not at all.
- 3) Nothrow guarantee - a function `f` offers the nothrow guarantee if it never emits an exception and always accomplished its purpose.

Will revisit.

Coming back to `f`...

```
void f() {
    Posn *p = new Posn {1, 2};
    if (some condition) {
        delete p;
        throw BadNews {};
    }
    delete p;
}
// duplicated effort - memory management is even harder!
```

Want to guarantee that `delete p` happens no matter what.

What guarantees does C++ offer? Only one: that dtors for stack-allocated objects will be called when the objects go out of scope.

So create a class with a dtor `deletes` the ptr.

```
template<typename T> class unique_ptr {
    T *p;
public:
    unique_ptr(T *p): p{p} {}
    ~unique_ptr() { delete p; }
    T *get() const { return p; }
    T *release() {
        T *q = p;
        p = nullptr;
        return q;
    }
};

void f() {
    unique_ptr<Posn> p { new Posn {1, 2} };
    if (some condition) { throw BadNews {}; }
}
```

That's it - less memory management effort than we started with.

Using `unique_ptr` - can use `get` to fetch the ptr.
Better - make `unique_ptr` act like a ptr.

```
template<typename T> class unique_ptr {
    T *p;
public:
    ...
    T &operator*() const { return *p; }
    T *operator->() const { return p; }
    explicit operator bool() const { return p; } // prohibits bool b = p;
    void reset(T *p1) {
        delete p;
        p = p1;
    }
    void swap (unique_ptr<T> &x) {
        std::swap(p, x.p);
    }
};

void f() {
    unique_ptr <Posn> p { new Posn {1, 2} };
    cout << p->x << "\n" << p->y;
}
```

But consider

```
unique_ptr <Posn> p { new Posn {1, 2} };
unique_ptr <Posn> q = p;
```

Two ptrs to the same object! Can't both delete it! Undefined behaviour.

Solution copying `unique_ptr`s is not allowed. Moving OK, though.

```
template <typename T> class unique_ptr {
    ...
public:
    ...
    unique_ptr (const unique_ptr &other) = delete;
    unique_ptr &operator (const unique_ptr &other) = delete;
    // this is how copying of streams is prevented
    unique_ptr (unique_ptr &&other): p{other.p} { other.p = nullptr; }
    unique_ptr &operator= (unique_ptr &&other) {
        swap(other);
        return *this;
    }
};
```

Small exception safety issue. Consider

```
class C {...};
void f(unique_ptr<C> x, int y) {...}
int g() {...}
f(unique_ptr<C> {new C}, g());
```

C++ does not enforce order of argument evaluation. It could be

- (1) `new C`
- (2) `g()`
- (3) `unique_ptr<C> {(1)}`

- then what if `g` throws? (1) is leaked
 - can fix this by making (1) and (3) inseparable. - use a helper function

```
template <typename T, typename... Args> unique_ptr<T> make_unique(Args&&... args) {
    return unique_ptr<T> { new T (std::forward<Args>(args)...)};
}
```

Example becomes `f(make_unique<C>(), g());`; - No leak if `g` throws. `unique_ptr` is an example of the C++ idiom Resource Acquisition Is Initialization (RAII)

- any resource that must be properly released (memory, file handle, etc) should be wrapped in a stack-allocated object whose dtor fixes it.

eg `unique_ptr`, `ifstream`

- acquire the resource when the object is initialized
 - release it when the object's dtor runs

Problem 15: Is vector exception safe?

Consider:

```
template <typename T> class vector {
    size_t n, cap;
    T *theVector;
public:
    vector(size_t n, const T &x): n{n}, cap{n},
        theVector{static_cast<T*>(operator new(n*sizeof(T)))} {
        for (size_t i = 0; i < n; ++i) new (theVector+i) T(x); // copy ctor - what if
        this throws?
    }
};
```

- partially constructed vector - dtor will not run
- broken invariant - the vector does not contain n valid objects.

Note if `operator new` throws - nothing has been allocated - no problem - strong guarantee Fix:

```
template <typename T> vector <T>::vector(size_t n, const T &x): n{n}, cap{n}, theVector
    { /* as before */ } {
    size_t progress = 0;
    try {
        for (size_t i = 0; i < n; ++i) {
            new (theVector +i) T(x);
            ++progress;
        }
    }
    catch (...) { // catch anything
        for (size_t i = 0; i < progress; ++i) theVector[i].~T();
        operator delete(theVector);
        throw; // rethrow
    }
}
```

Abstract the filling part into its own function:

```

template <typename T> void uninitialized_fill (T *start, T *finish, const T &x) {
    T *p;
    try {
        for (p = start; p != finish; ++p) new (p) T (x);
    }
    catch (...) {
        for (T *q = start; q != p; ++q) q->~T();
        throw;
    }
}

template <typename T> vector<T>::vector(size_t n, const T &x): n{n}, cap {n},
    theVector {static_cast<T*>(operator new(n*sizeof(T)))} {
    try {
        uninitialized_fill(theVector, theVector+n, x);
    }
    catch (...) {
        operator delete (theVector);
        throw;
    }
}

```

Can clean this up by using RAII on the array

```

template <typename T> struct vector_base {
    size_t n, cap;
    T *v;
    vector_base (size_t n): n{n}, cap{cap}, v{static_cast <T*> (operator new (n*sizeof(T))
    )} {}
    ~vector_base() { operator delete(v); }
};

template <typename T> class vector {
    vector_base <T> vb; // cleaned up implicitly when vector is destroyed
public:
    vector (size_t n, const T &x): vb{n} {
        uninitialized_fill(vb.v, vb.v+vb.n, x);
    }
    ~vector() { destroy_elements(); }
    void destroy_elements() {
        for (T *p = vb.v, p != vb.v+vb.n; ++p) p->~T();
    }
};

```

Strong guarantee

Copy ctor:

```

template <typename T> vector<T>::vector(const vector &other): vb{other.size()} {
    uninitialized_copy (other.begin(), other.end(), vb.v);
    // similar to uninitialized_fill. Details - exercies
}

```

Assignment: copy and swap is exception safe - as long as `swap` is `nothrow`
`push_back`:

```
template <typename T> vector<T>::push_back (const T &x) {
    increaseCap(); // details - exercise ... done below
    /* new(vb.v + vb.n++) T(x); */
    // don't increment n before you know the construction has succeeded
    // instead
    new(vb.v + vb.n) T(x);
    ++vb.n;
    // if this throws - have the same vector
}
```

What if `increaseCap` throws?

```
void increaseCap() {
    if (vb.n == vb.cap) {
        vector_base vb2 {2 * vb.cap}; // RAI
        uninitialized_copy(vb.v, vb.v + vb.n, vb2.v); // Strong guarantee
        // change this into uninitialized_copy_or_move
        destroy_elements();
        std::swap(vb, vb2); // no throw on ints and ptrs
    }
}
```

Note: only `try` blocks we use are in `uninitialized_fill` and `uninitialized_copy`

Efficiency

- copying from the old array to the new one - moving would be better
- but moving destroys the old array so if an exception is thrown during moving, our vector is corrupted.

So we can only move if we are sure that move operation is `nothrow`.

```
template <typename T> void uninitialized_copy_or_move(T *start, T *finish, T *target) {
    T *p;
    try {
        for (p = start; p != finish; ++p, ++target) {
            new(p) T(std::move_if_noexcept(*target));
        }
    }
    catch (...) { // will never happen if T has a non-throwing move
        for (T *q = start; q != p; ++q) q->~T();
        throw;
    }
}
```

`std::move_if_noexcept(x);`

- produces `std::move(x)` if `x` has non-throwing move ctor

- produces x otherwise

But how should the compiler know if T's move ctor is non-throwing?

You have to tell it.

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

In general, `move` and `swap` should be no-throwing.

Declare them so - will allow more optimization codes to run. Any functions you are sure will never have to throw or propagate an exception, you should declare `noexcept`.

Q: Is `std::swap` `noexcept`?

```
template <typename T> void swap (T &a, T &b) _____ {
    T c (std::move(a));
    a = std::move(b);
    b = std::move(c);
}
```

A: Only if T has a `noexcept` move ctor and move assignment. Thus blank should be filled with

```
noexcept(std::is_nothrow_move_constructible<T>::value &&
         std::is_nothrow_move_assignable<T>::value)
```

Note `noexcept` means the same as `noexcept(true)`

16

Problem 16: Insert/remove in the middle

A method like `vector<T>::insert(size_t y, const T &x)` is easy to write. The same method for `list<T>` would be slow (linear traversal) Using iterators can be faster.

```
template <typename T> class Vector {
public:
    iterator insert (iterator posn, const T &x) {
        increaseCap();
        // - add new items to end
        // - shuffle items down (move_if_noexcept)
        return iterator to point of insertion
    }
};
```

adding the codes...

```
template <typename T> class Vector {
public:
    iterator insert (iterator posn, const T &x) {
        ptrdiff_t offset = posn - begin();
        increaseCap();
        iterator newPosn = begin() + offset;
        new (end()) T (std::move_if_noexcept (* (end()-1)));
        ++vb.n;
        for (iterator it = end() - 1; it != newPosn; --it) {
            *it = std::move_if_noexcept (* (it-1));
        }
        *newPosn = x;
        return newPosn;
    }
};
```

Exception safe?

- assuming T's copy/move options are exception safe (at least basis guarantee)
- may get a partially shuffled vector, but it will still be a valid vector

Note if you have other iterators pointing at the vector:

1	2	3	4	...
it1		here	it2	

and you insert at 'here':

1	2	5	3	4	...
it1		here	it2		

it 2 points at different item.

Convention: after a call to insert or erase, all iterators pointing after the point of insertion/erasure are considered invalid and should not be used.

Similarly if a reallocation happens - all iterators pointing at the vector become invalid.

Exercises: erase

- remove the item pointed to by an iterator
- return an iterator to the point of erasure

emplace - like insert, but takes ctor args.

But - that means we have a problem with `push_back`. If placement new throws, the vector is the same, but iterators were invalidated!

To fix:

- allocate new array
- place the new item
- copy or move old items
- delete

`void pop_back()` Not `T pop_back()`.

- would call copy (or move) ctor on return
- then dtor on item in vector
- if copy ctor throws, dtor call will not happen
- not exception safe

above needs to be changed a little. If the copy ctor throws, the client doesn't get the item and the item is popped, and it can not recover gracefully.

17

Problem 17: Abstraction Over Containers

Recall: map from Racket

```
(map f (list a b c)) --> (list (f a) (f b) (f c))
```

May want to do the same with vectors

```
source [a | b | c | d] → target [f(a) | f(b) | f(c) | f(d)]
```

```
template <typename T1, typename T2>
void transform(const vector <T1> &source, vector<T2> &target, T2 (*f)(T1)) {
    auto it = target.begin();
    for (auto &x : source) {
        *it = f(x);
        ++it;
    }
}
```

OK, but ..

- what if we only want part of source?
- what if we want to send the source to the middle of target, not the beginning?

More general: use iterators.

```
template <typename T1, typename T2>
void transform(vector <T1>::iterator start, vector <T1>::iterator finish, vector<T2>::
    iterator target, T2 (*f)(T1)) {
    while (start != finish) {
        *target = f(*start);
        ++target;
        ++start;
    }
} // ... won't compile
```

(Assume it works) OK, but

- if I want to transform a list, I will write the same code again.

- what if I want to transform a list a vector or vice versa?
- make the type variables stand for the iterators, not the container elements.
- but then how do we indicate the type of f?

```
template <typename InIter, typename OutIter, typename Fn>
void transform (InIter start, InIter finish, OutIter target, Fn f) {
    while (start != finish) {
        *target = f(*start);
        ++target;
        ++start;
    }
}
```

works over vector iterators, list iterators, or any other kinds of iterators.

InIter/OutIter can be any types that support ++, !=, *, including ordinary pointers.

C++ will instantiate this template function with any type that has the operations used by the function. Function can be any type that supports function application.

Eg

```
class Plus {
    int n;
public:
    Plus (int n): n{n} {}
    int operator() (int m) { return m + n; }
};

Plus p {5};
cout << p(7); // 12
// function object

transform(v.begin(),v.end(),w.begin(), Plus{1}); // in last lecture of cs246...
// OR
transform(v.begin(),v.end(),w.begin(), [](int n){return n+1;});
// |<-      lambda      ->|
```

lambda:

```
[../* capture list */..](../* parameter list */..) mutable? noexcept? { ../* body */ ..}
```

semantics:

```
void f(T1 a, T2 b) {
    [a, &b] (int x) { return body; } (args)
}
// translated to
void f(T1 a, T2 b) {
    class ~~~ { // name not known to you
        T1 a;
```

```
T2 &b;  
public:  
    ~~~ (T1 a, T2 &b): a{a}, b{b} {}  
    auto operator() (int x) const { return body; }  
};  
~~~ {a, b}.operator() (args);  
}
```

If the lambda is declared mutable, then `operator()` is not `const`.

capture list - provides access to selected vars in the enclosing scope.

18

Problem 18: Heterogeneous Data

I want a mixture of types in my vector - can't do this with a template

eg

```
vector<template <typename T> T> v; // X
```

- not allowed - templates are compile-time entities - don't exist at run time.

eg fields of a struct

```
class MediaPlayer {  
    ...  
    template <typename T> T nowPlaying; // songs, movies etc  
    // can't do this
```

What's available in C:

```
// unions:  
union Media {Song s; Movie m;};  
Media nowPlaying;  
  
// void *:  
void *nowPlaying;  
  
// both not type-safe
```

Items in a heterogeneous collection will usually have something in common, e.g. provide a common interface.

Can be views as different "kinds" of a more general "thing".

So have a vector of "things" of a field of type "thing".

We'll use the standard CS 246 example (will be covered tomorrow in CS 246) :

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

```
bool isHeavy() const { return length > 100; }
string getTitle() const { return title; }
// etc
};
```

Book	title
	author
	length

```
class Text: public Book { // subclass, derived class
    string topic;
public:
    Text(string title, string author, int length, string topic):
        Book{title, author, length}, topic{topic} {}
    // if we don't do this with book, default ctor will be invoked, but we don't have
    // one
    bool isHeavy const { return length > 200; }
    string getTopic const { return topic; }
};

class Comic: public Book {
    string hero;
public:
    Comic (...) ... // exercise
    bool isHeavy () const { return length > 50; }
    string getHero() const { return hero; }
};
```

Text	title
	author
	length
	topic

Subclasses inherit all members (fields and methods) from their superclasses. All three classes have author, length, methods: getTitle, getAuthor, getLength, isHeavy. ...except that this doesn't work !!!

```
string Text::isHeavy() const { return length > 200; } // private in Book
```

2 options:

```
// 1)
class Book {
    string title, author;
protected:
    int length;
public:
    ...
};

// 2)
string Text::isHeavy() const { return getLength() > 200; }
```

Recommend: option 2)

- You have no control over what subclasses might do.
- protected weakness encapsulation. Cannot enforce invariants on protected fields.

If you want subclasses to have privileged access

- keep fields private
- provide `protected` `get__` and `set__` methods

Updated object creation/destruction steps:

Created

1. Space is allocated
2. Superclass part is constructed
3. Fields constructed
4. Ctor body

Destroyed

1. Dtor body
2. Fields destructed
3. Superclass is destructed
4. Space deallocated

Must now revisit everything we have learned to see the effect of inheritance.

Type compatibility: Texts and Comics are special kinds of Books.

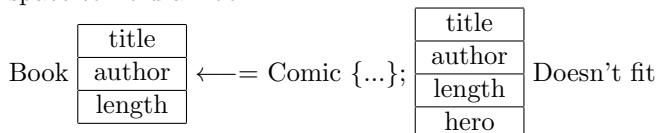
- should be useable in place of Book.

```
Book b = Comic {"_", "_", 75, "_"}; // compiled
b.isHeavy(); // false! 75 is a heavy comic, not a heavy Book.
```

If b is a Comic, why is b acting like a Book?

Because b is a Book. Consequence of stack-allocated objects.

sets aside enough
space to hold a Book



Keeps only the Book part

- comic part is chopped off - "slicing"
- so it really is just a book now.
- so `Book::isHeavy` runs

- slicing happens even if superclass and subclass are the same size.

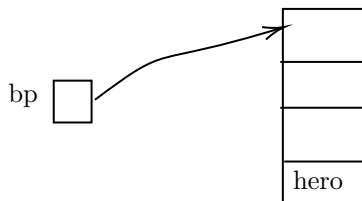
```
vector<Book> library;
library.push_back(Comic{...});
// only the Book parts will be pushed
// not a heterogeneous collection
```

But note:

```
void f(Book books[]); // raw arrays
Comic comics[] = {...};
f(comics);
// will succeed (legal) - but *never* do this!
// array will be misaligned
// will not act like an array of Books
```

Slicing does not happen through ptrs:

```
Book *bp = new Comic {"_", "_", 75, "_"};
```



```
bp->isHeavy(); // still false
```

Rule: the choice of which `isHeavy` to run is based on the type of the pointer (static type), not the object (dynamic type)

Why? Cheaper.

C++ design principle: if you don't use it, you shouldn't have to pay for it. i.e. if you want something expensive, you have to ask for it.

To make `*bp` act like a `Comic` when it is a `Comic`:

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

```
public:
    ...
    bool isHeavy() const override {...}
};
bp->isHeavy(); // true!
```

Assume `isHeavy` is `virtual`.

can now have a true heterogeneous collection:

```
vector<Book*> library;
library.push_back(new Book {...});
library.push_back(new Comic {...});
// even better
vector<unique_ptr<Book>> library;

int howManyHeavy(const vector<Book*> &v) {
    int count = 0;
    for (auto &b : v) {
        if (b->Heavy()) ++count;
        // correct version of isHeavy is always chosen, even though we don't know what's
        // in the vector, and the items are probably not the same type.
    }
    return count;
}

for (auto &b : library) delete b; // Not necessary if you use unique_ptr
```

polymorphism

How do virtual methods "work" and why are they more expensive? Implementation-dependent,
but the following is typical: see Figure 18.1

Non-virtual methods \implies ordinary function calls

If there is at least one virtual method

- compiler creates a table of function pointer - one per class - the vtable.
 - each object contains a ptr to its class' vtable - the vptr
 - calling a virtual method - follow the vptr to the vtable, follow the function ptr to the correct function
- vptr is often the "first" field
- so the subclasses object still looks like a super object
 - so the program knows where the vptr is.

- so virtual methods incur a cost in time (extra ptr dereferences) and in space (each object gets a vptr)
- If a subclass doesn't override a virtual method, its vtable will point at the superclass implementation

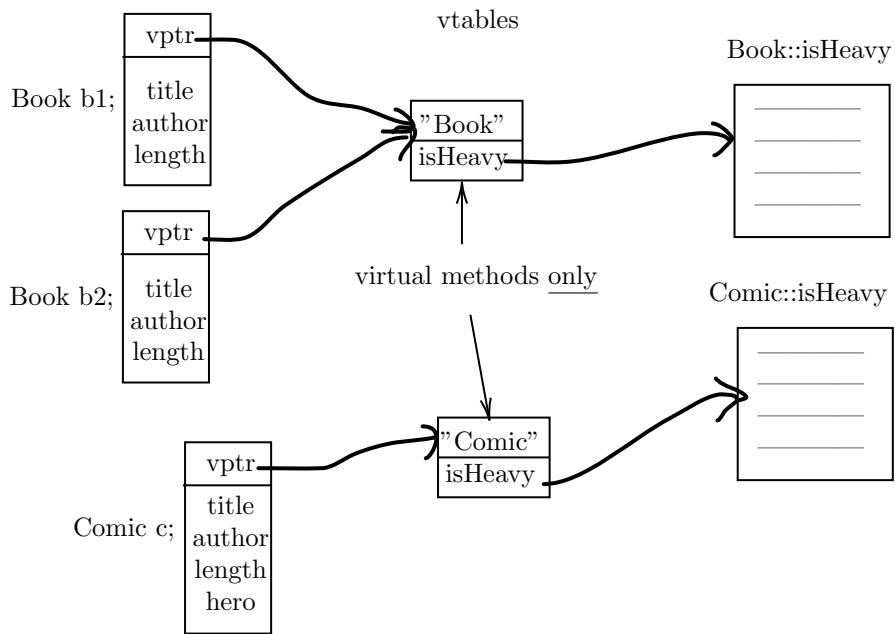


Figure 18.1: An example of vtable

19

Problem 19: I'm Leaking

Consider:

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

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

Note: Y's dtor will call X's dtor (Step 3)

```
X *px = new Y {3, 4};
delete px; // Leaks! - Calls X's dtor, but not Y's
```

Solution: make the dtor virtual

```
class X {
    ...
public:
    ...
    virtual ~X() {delete [] a;}
};
```

- Always make the dtor virtual in classes that are meant to be superclasses - even if the dtor does nothing. - you never know what the subclass might do, so you need to make sure its dtor gets called.
- Also always give your virtual dtor an implementation - even if empty - it will get called by subclass dtor

If a class is not meant to be superclass

- no need to incur the cost of virtual methods needlessly.
- leave the dtor non-virtual, but declare the class final:

```
class X final {  
    ... // cannot be subclassed  
};
```

20

Problem 20: I want a class with no objects

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

class RegularStudent: public Student {
public:
    float fees() const override; // Regular student fees
};

class CoOpStudent: public Student {
public:
    float fees() const override; // Co-op student fees
};
```

What should `Student::fees` do?

Don't know - every `Student` should be `Regular` or `Co-op`.

Solution - explicitly give `Student::fees` no implementation.

```
class Student {
public:
    virtual float fees() const = 0; // called a pure-virtual method.
};
```

Abstract classes cannot be instantiated, but can point to instances of create subclasses.

```
// Student s;
// Student *s = new Student;
Student *s = new Regular Student;
```

Subclasses of abstract classes are abstract unless they implement all pure virtual methods.

Abstract classes

- used to organize concrete classes
- can contain common fields, methods, default implementations

21

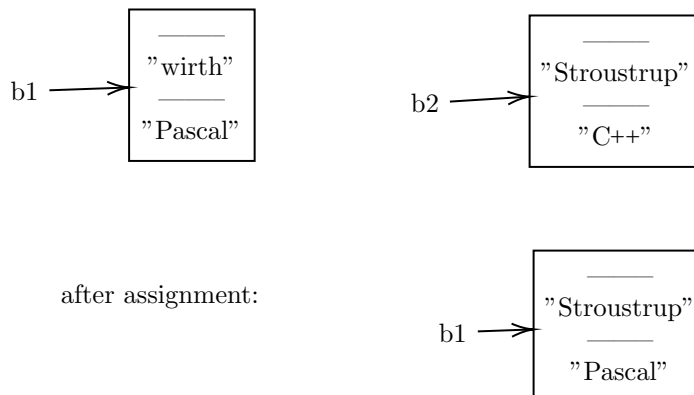
Problem 21: The copier is broken

How do copies and moves interact with inheritance?

```
// copy ctor
Text::Text(const Text &other): Book{other}, topic{other.topic} {}
// move ctor
Text::Text(Text &&other): Book{std::move(other)}, topic{std::move(other.topic)} {}
// copy assign
Text &Text::operator=(const Text &other) {
    Book::operator=(other);
    topic = other.topic;
    return *this;
}
// move assign
Text &Text::operator=(Text &&other) {
    Book::operator=(std::move(other));
    topic = std::move(other.topic);
    return *this;
};
```

But consider:

```
Book *b1 = new Text {...}, *b2 = new Text {...};
*b1 = *b2;
```



What happen? Only the Book part is copied - partial assignment

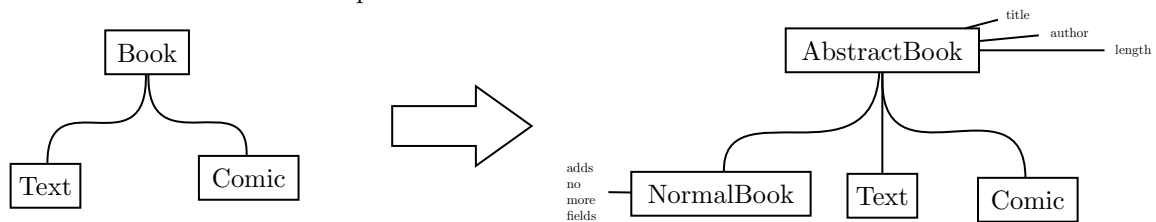
- topic doesn't match title and author - object is corrupted.

Possible solution? Make `operator=` virtual.

```
class Book {
...
public:
    virtual Book &operator=(const Book &other) {...}
};
class Text: public Book {
...
public:
    // Text &operator= (const Text &other) override {...}
    Text &operator= (const Book &other) override {...} // mixed assignment
    // inside () -> must be Book. Or it's not an override.

    // so could pass a Comic - also a problem - will revisit shortly
};
```

Another solution Make all superclasses abstract



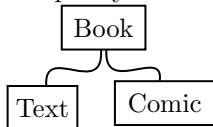
```
class AbstractBook {
protected:
    AbstractBook &operator=(const AbstractBook &other) {...} // non-virtual
    virtual ~AbstractBook() = 0; // to make the class abstract
};
AbstractBook::~AbstractBook() {}

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

`operator=` non-virtual, so no mixed assignment.
 ∴ no partial assignment, i.e. `*b1=*b2`; won't compile.

Problem 22: I want to know what kind of Book I have

For simplicity: assume



C-style casting:

- (type) expr
- forces 'expr' to be treated as type 'type'

```
int addr = ...;
int *p = (int *) addr;
```

The C++ casting operators: - 4 operators

1. `static_cast` for conversions with a well-defined semantics.

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

superclass ptr to subclass ptr

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

You have to know that `b` really points at a `Text` - otherwise undefined behaviour.

2. `reinterpret_cast`

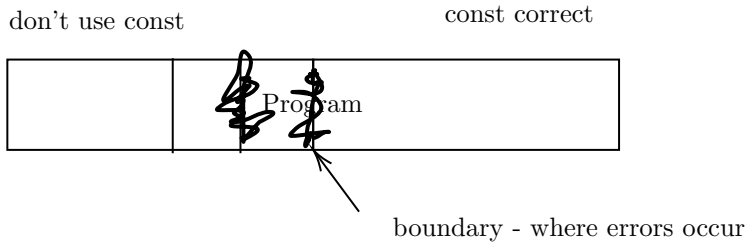
- for cases without a well-defined semantics.
- unsafe, implementation-dependent.

```
Book *b = new Book {...};
int *p = reinterpret_case<int *>(b);
```

3. `const_cast`

- for adding/removing `const`
- the only C++ `const` that can "cast away `const`"

```
void g (Book &b); // assume g doesn't actually modify b.
void f(const Book &b) {
    g(const_cast<Book&>(b));
}
```



4. `dynamic_cast`

```
Book *pb = ...;
```

- what is we don't know whether pb points at a `Text`?
- `static_cast` not safe.

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

If `*pb` is a `Text` or a subclass of `Text`, case succeeds - `pt` points to the object.
 else - `pt = nullptr`

```
if (pt) { ... pt->getTopic() ...
else ...// not a Text
```

eg

```
void whatIsIt (Book *pb) {
    if (dynamic_cast<Text *>(pb)) cout << "Text";
    else if (dynamic_cast<Comic *>(pb)) cout << "Comic";
    else cout << "Book";
}
```

- not good style - What happens when you create a new Book type?

Dynamic reference casting:

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

If *pb is a Text - OK. else raises `std::bad_cast`

Note dynamic casting works by accessing on object's Run-Time Type Information (RTTI)

- this is stored in the vtable for the class.

∴ can only use `dynamic_cast` on objects with at least one virtual method.

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

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

Part II

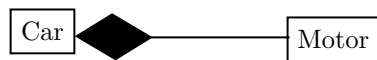
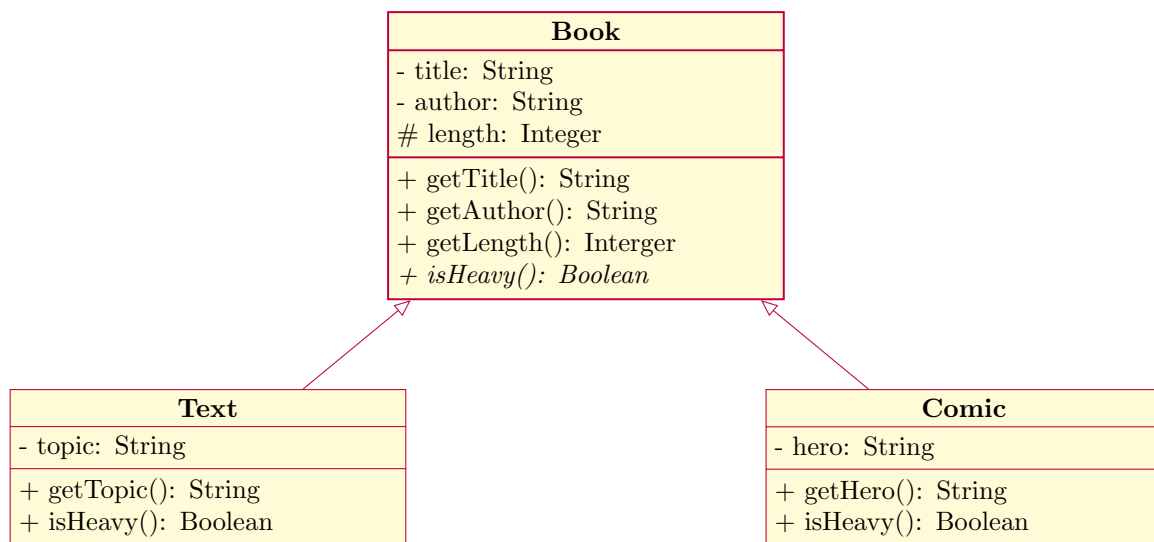
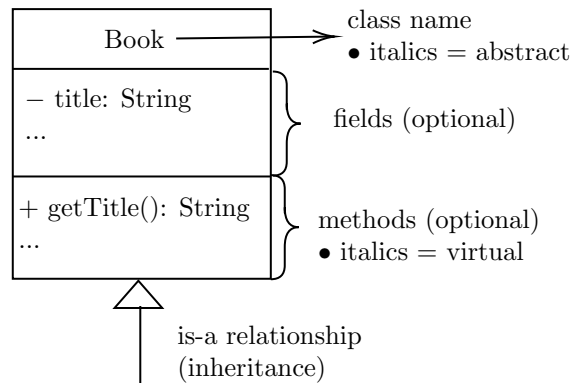
A Big Unit on Objected-Oriented Design

System Modelling - UML (United Modelling Language)

- makes ideas easy to communicate, aids design discussion

Notation

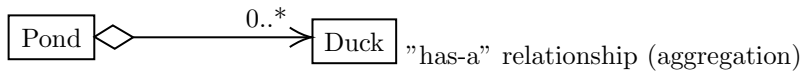
- = private
= protected
+ = public



owns "a" relationship

- means the motor is part of the car - does not have an independent existence
- copy/destroy the car \implies copy/destroy the motor
- typical implementation: class composition, i.e. object fields.

```
class Car {Motor m; };
```



- Duck has its own independent existence
- copy/destroy pond $\not\Rightarrow$ copy/destroy ducks
- typical implementation

```
class Pond { vector<Duck*> ducks; };
```

The concept of ownership is central to OO design in C++. Every resource (memory, file, window, etc.) should be owned by an object that releases it. RAII.

A unique_ptr owns the memory it points to.

A raw ptr should be regarded as not owning the memory it points to.

If you need to point at the same object with several ptrs, one ptr should own it and be unique_ptr. The rest should be raw ptrs.

Moving a unique_ptr = transferring ownership

If you need true shared ownership - later.

24.1 Coupling & Cohesion

Coupling - how strongly different modules depend on each other?

LOW:

- function calls with parameters/results of basic type
- function calls with array/struct parameters
- modules that affect each other's control flow

HIGH:

- modules access each other's implementation (friends)

high coupling \implies

- changes to one module affect other modules
- harder to reuse individual modules

Eg function `WhatIsIt` (`dynamic_cast`)

- tightly coupled to the Book class hierarchy
- must change this function if you create another Book subclass

Cohesion - how closely are elements of a module related to each other?

LOW:

- arbitrary grouping (e.g. `<utility>`)
- common theme, otherwise unrelated, maybe some common base code (e.g. `<algorithm>`)
- elements manipulate state over the lifetime of an object (e.g. open/read/close files)

- elements pass data to each other

HIGH:

- elements cooperate to perform exactly one task

low cohesion \implies poorly organized code; hard to understand, maintain

Want low coupling, high cohesion.

SOLID Principles of OO design

25.1 Single Responsible Principle

- A class should only have one reason to change
i.e. a class should do one thing, not several
- Any change to the problem spec requires a change to the program
- If change to ≥ 2 different part of the spec cause changes to the same class, SRP is violated.

Eg Don't let your classes print things.

Consider:

```
class ChessBoard {  
...  
    cout << "Your_move";  
...  
};
```

- bad design - inhibits code reuse.
- What if you want a version of your program that
 - communicates over different streams (files, network)?
 - works in another language?
 - uses graphics instead of that?
- Major changes to ChessBoard class, instead of reuse.

Violates SRP

- must change this class if there is any change to the specification for
 - game rules
 - strategy
 - interface
 - etc.

- low cohesion

Split these responsibilities up

- one module (not main! can't reuse main) responsible for communication.
- pass info to the communications object and let it do the talking

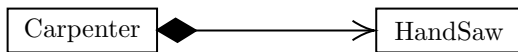
On the other hand

- specifications that are unlikely to change may not need their own class
- avoid needless complexity - judgement call

25.2 Open-Closed Principle

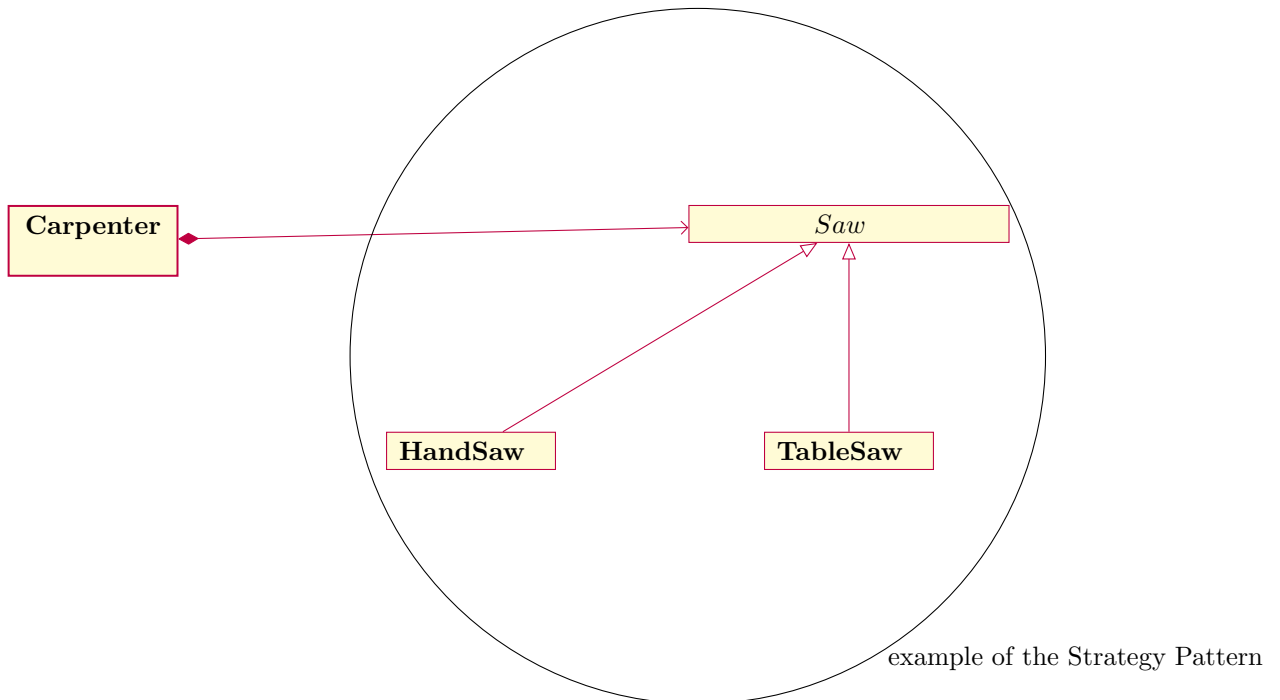
- classes, modules, functions, etc. Should be open for extension and closed for modification.
- Changes in a program's behaviour should happen by writing new code - extending functionality - not changing old code

Eg



What if the carpenter buys a tableSaw?
- this design is not open for extension - must change code.

Solution



No changes need for Carpenter.

Also note countHeavy function

```
int countHeavy(const vector<Book *> &v) {
    int count = 0;
    for (auto &p:v) if (p->isHeavy()) ++count;
}
```

- no changes needed if new Book invented.
- vs. WhatIsIt (dynamic casting) - not closed to modification

Note can't really be 100% closed. Some changes require source modifications. Plan for the most likely changes and make code closed with respect to those changes.

25.3 Liskov Substitution Principle

- Simply put: public inheritance must indicate an "is-a" relationship.

But there is more to it than that:

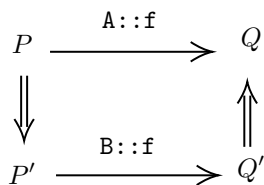
If B is a subtype (subclass) of A, then we should be able to use an object b of type B in any context that requires an object of type A, without affecting the correctness of the program. This is the important point.

C++'s inheritance rules already allow us to use subclass objects in place of superclass objects.

Informally: a program should "not be able to tell", if it is using a superclass object or a subclass object.

More formally

- If an invariant I is true of class A, then it must be true of class B.
- If an invariant I is true of method of $A::f$, and $B::f$ overrides $A::f$, then I must hold for $B::f$
- If $A::f$ has precondition P and postcondition Q , then $B::f$ must have precondition $P' \Leftarrow P$ and postcondition $Q' \Rightarrow Q$.

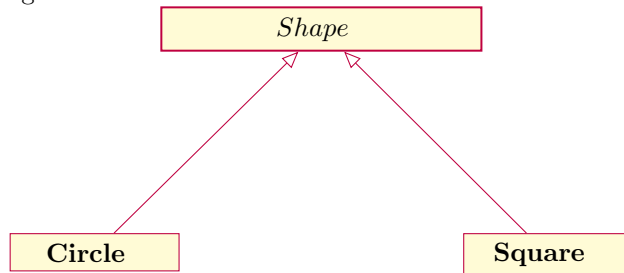


- If $A::f$ and $B::f$ behave differently, the difference in behaviour must fall within what is allowed by the program's correctness specification.

Examples

- contravariance problem
 - arises any time you write a binary operator, i.e. a method with an "other" parameter of the same type as `*this`.

eg



```

class Shape {
public:
    virtual bool operator==(const Shape &other) const;
};
class Circle:public Shape {
public:
    bool operator==(const Circle &other) const override;
};
  
```

- violates Liskov substitution.
 - A Circle is a Shape
 - A Shape can be compared with any other shape.
∴ A Circle can be compared with any other shape.
 - (We saw this with `virtual operator=`)
 - C++ will flag this as a compiler error.

Fix

```

#include <typeinfo>
bool Circle::operator==(const Shape &other) const {
    if (typeid(other) != typeid(Circle)) return false;
    const Circle &other = static_cast<const Circle &> (other);
    //compare fields between *this and other
}
  
```

`dynamic_cast` vs `typeid`

```

dynamic_cast<const Circle &> (other) // is other a Circle or a subclass of Circle?
typeid(other) == typeid(Circle) // is other precisely a circle?
// typeid(other) returns an object of type type_info
  
```

2. Is a square a rectangle?
 - a square has all the properties of the rectangle.

```

class Rectangle {
    int length, width;
public:
    int getLength() const;
    int getWidth() const;
    virtual void setLength (int length);
  
```

```

virtual void setWidth (int width);
int area() const { return length*width; }
};

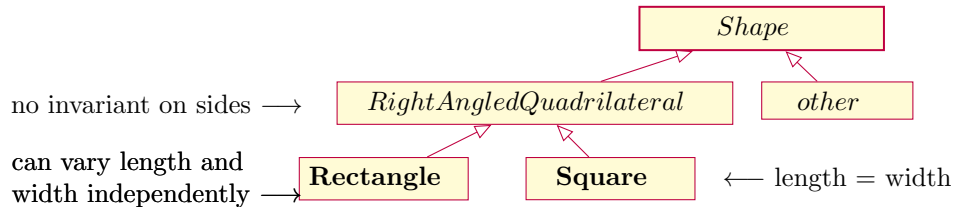
class Square: public Rectangle {
public:
    Square(int side): Rectangle {side, side} {}
    void setLength(int length) override {
        Rectangle::setLength(length);
        Rectangle::setWidth(length);
    }
    // setWidth - similar
};

int f(Rectangle &r) {
    r.setLength(10);
    r.setWidth(20);
    return r.area(); // expect 200
}
Square s{1};
f(s); // 400

```

Rectangles have the property that their length and width can vary independently; square do not. So this violates LSP.

On the other hand, an immutable Square could substitute for an immutable rectangle.



Constraining what subclasses can do.
Consider

```

class Turtle {
public:
    virtual void draw = 0;
};

class RedTurtle: public Turtle {
public:
    void draw () override {
        drawHead();
        drawRedShell();
        drawTail();
    }
};

class GreenTurtle: public Turtle {
public:
    void draw () override {

```

```

    drawHead();
    drawGreenShell();
    drawTail();
}
};
// head and tail are same

```

- code duplication
- Plus - how can we ensure that overrides always do these things?

```

class Turtle {
public:
    void draw () {
        drawHead();
        drawShell();
        drawTail();
    }
private:
    void drawHead();
    virtual void drawShell();
    void drawTail();
};

class RedTurtle: public Turtle {
    void drawShell() override;
};

class GreenTurtle: public Turtle {
    void drawShell() override;
};
// subclass can override a private method, but is not able to call it.

```

Subclasses cannot control the steps of drawing a Turtle, nor the drawing of head & Tail. Can only control the drawing of the shell.

- called the Template Method Pattern

Extension: NVI (Non-Virtual Interface) idiom

- public: virtual methods are simultaneously
 - part of a class' interface - pre/post conditions class invariants, stated purpose.
 - "hooks" for customization by subclasses - overriding code could be anything. These two at odds with each other.
- hard to vary these independently if they are bound to the same function

NVI says: all virtual methods should be private, i.e. all public methods should be non-virtual.

Eg

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

↓

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

- Generalizes Template Method Pattern - puts every virtual function inside a template method.

25.4 Interface Segregation Principle

- many small interfaces is better than one large interface
- if a class has many functionalities, each client of the class should see only the functionality it needs.

Eg video game

```
class Enemy {
public:
    virtual void draw(); // needed by UI
    virtual void strike(); // needed by game logic
};

class UI {
    vector <Enemy *> v;
};

class Battlefield {
    vector <Enemy *> v;
};
```

If we need to change the drawing interface, `Battlefield` must recompile for no reason. If we need to change the combat interface, `UI` must recompile for no reason.

Creates needless coupling between `UI` and `Battlefield`

One solution: multiple inheritance

```

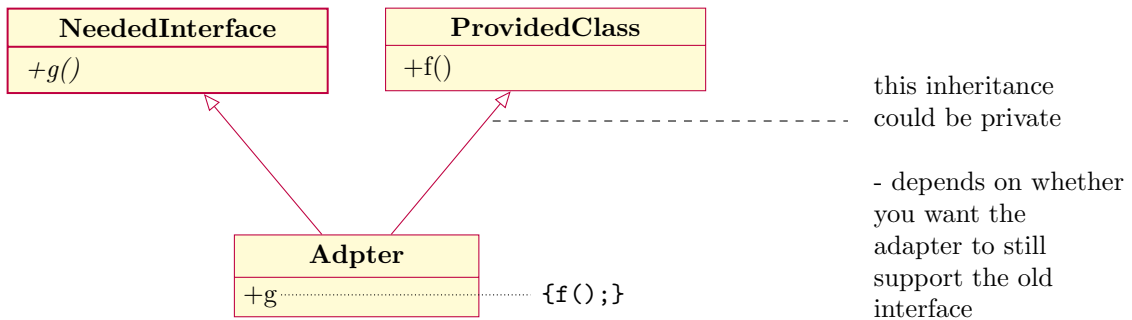
class Draw { // actually means drawable
public:
    virtual void draw() = 0;
};
class Combat { // combatable
public:
    virtual void strike() = 0;
};

class Enemy: public Draw, public Combat { };

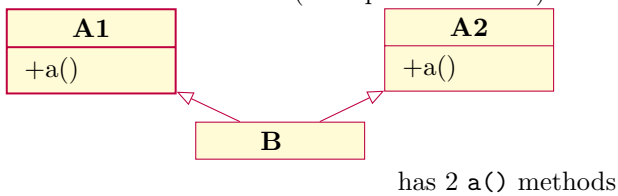
class UI {
    vector <Draw *> v;
};
class Battlefield {
    vector <Combat *> v;
};
    
```

Example of the Adapter Pattern.
 General use of Adapter - when a class provides an interface different from the one you need.

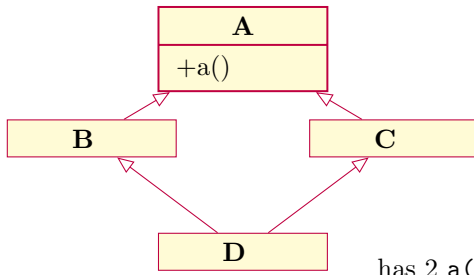
Eg



Detour issue with MI (multiple inheritance)



or indeed



has 2 a() methods, and they are different

```

class D: public B, public C {
    void f() { ... a() ...} // use B::a() or C::a(). ambiguous
};

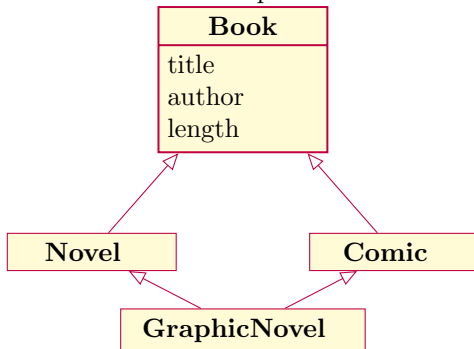
D d;
d.a(); // ambiguous
// use d.B::a() or d.C::a()
  
```

OR may be there should be only one A base and therefore only one a().

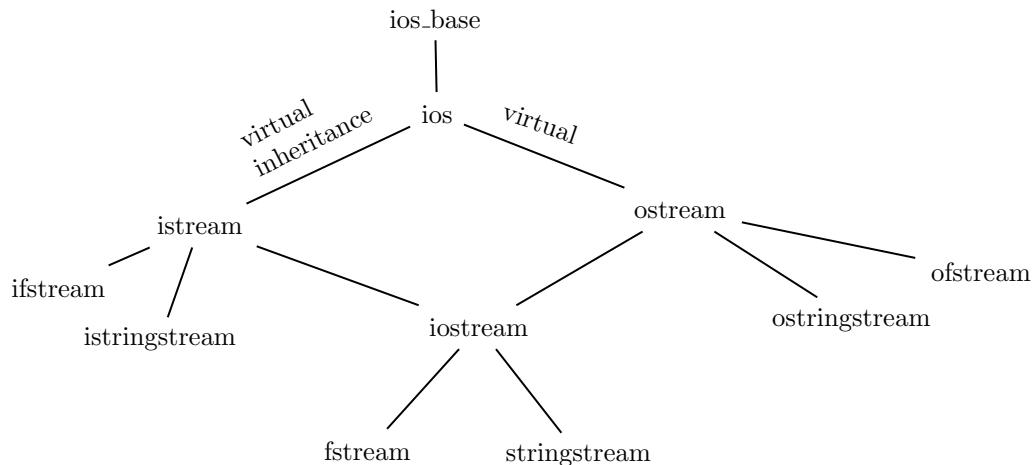
```

class B: virtual public A /* virtual base class */ {...}; // virtual inheritance
class C: virtual public A {...};
d.a(); // No longer ambiguous
  
```

Recall Book example:

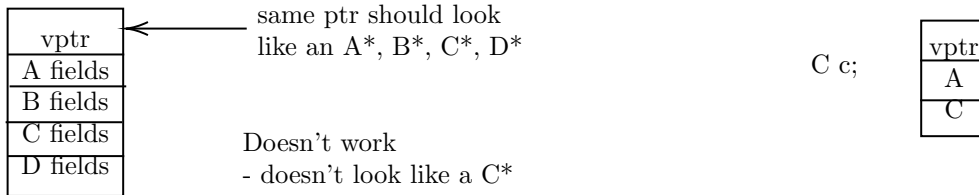


Eg iostream hierarchy

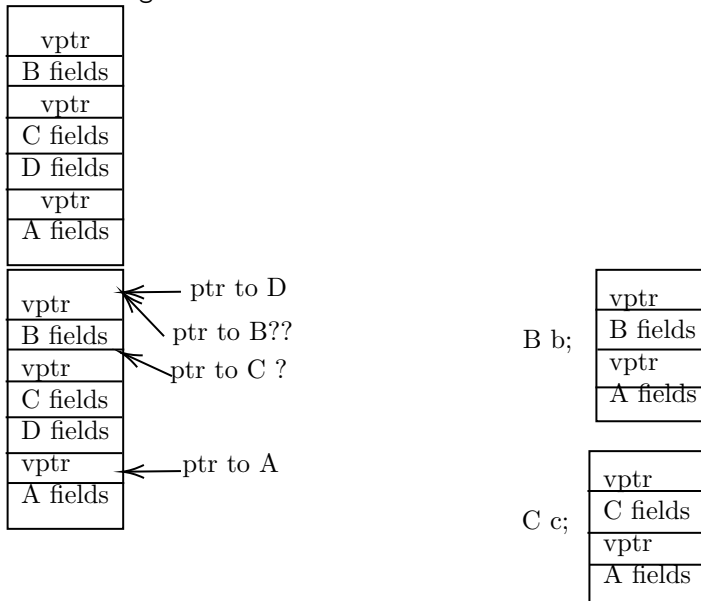


Problem How will a class like D be laid out in memory? (implementation-specific)

Consider



What does g++ do?



B & C need to be laid out so that we can find the A part, but the distance is not known (depends on the run-time type of the object).

Solution: location of the base object stored in vtable.

Also note: diagram doesn't simultaneously look like A, B, C, D.

- but slices of it do.
- ∴ ptr assignment A, B, C, D may change the address stored in the ptr.

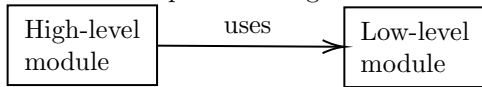
```
D *d = ....;
A *a = d; // changes the address - adds an offset to point to the A part.
```

`static_cast` and `dynamic_cast` under MI will also adjust the value of the ptr. `reinterpret_cast` will not.

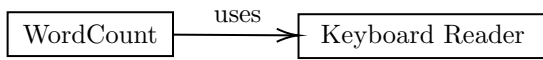
25.5 Dependency Inversion Principle

- High-level modules should not depend on low-level modules. Both should depend on abstractions.
- Abstract classes should never depend on concrete classes.

Traditional top-down design:

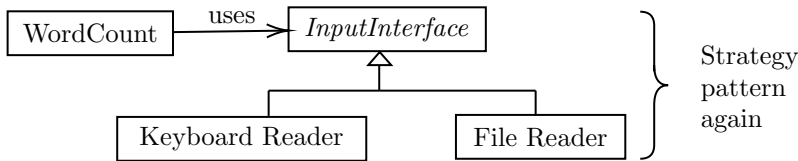
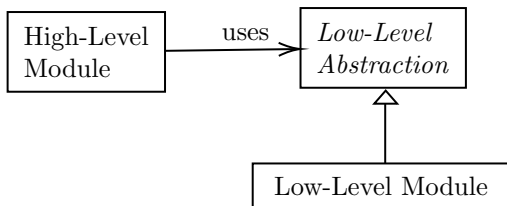


Eg

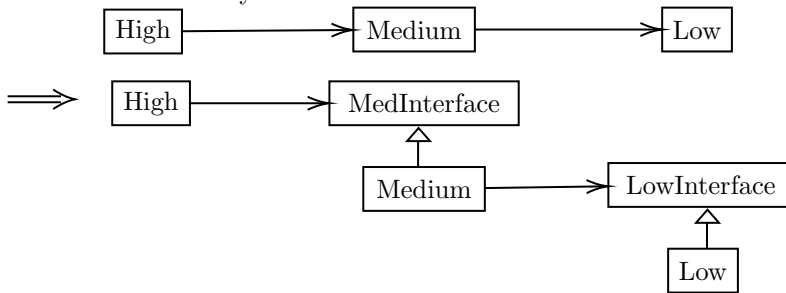


What if I want to use a file reader?
 - changes to details affect higher-level module.

Dependency inversion



Works over several layers as well:

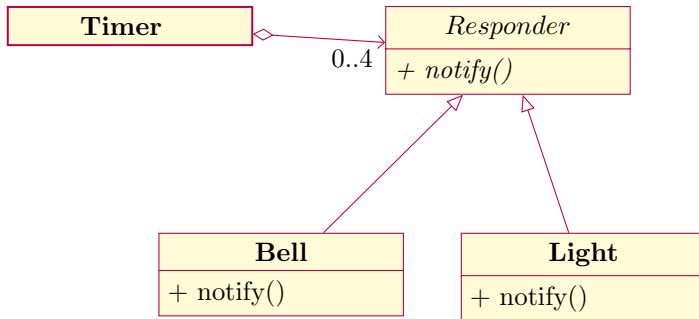


Eg

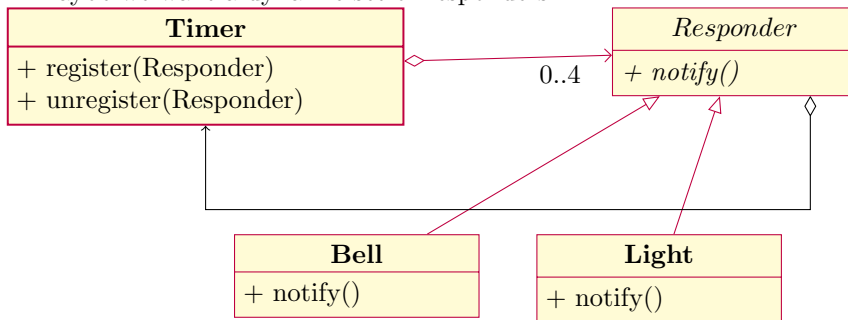


When the timer hits some specified time, it rings the bell (call `Bell::notify`, which rings the bell)

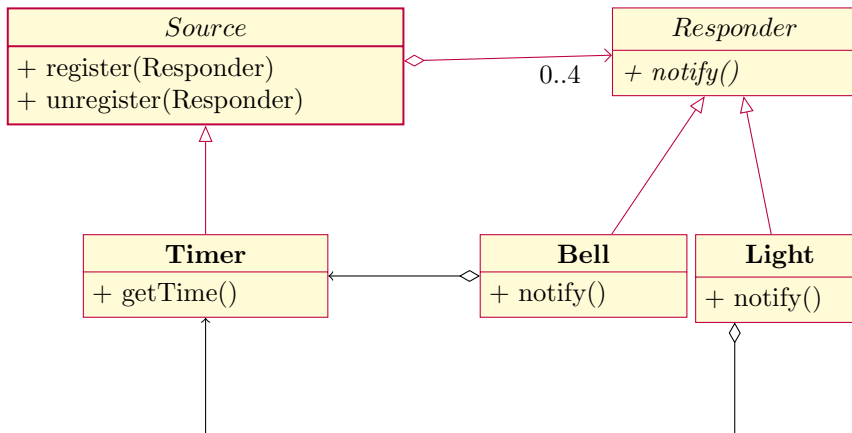
What if we want to trigger other events? - Maybe more than one?



Maybe we want a dynamic set of responders:



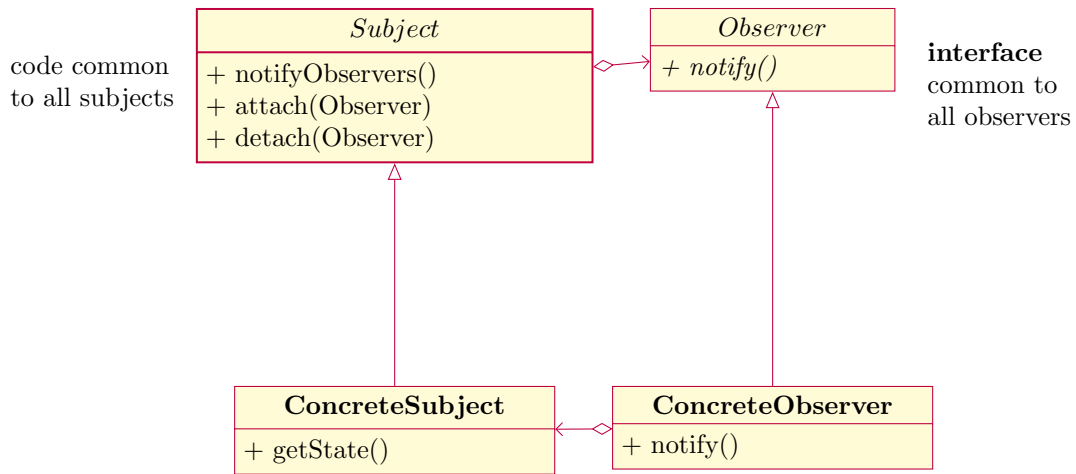
Now Responder is depending on concrete Timer class. Can apply DIP again:



If Light/Bell's behaviour depends on the time, they may need to depend on the concrete timer for a `getTime` method.

- could dependency invert this as well, if you wanted.

General Solution known as Observer Pattern.



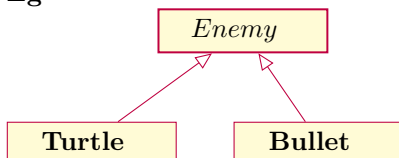
Sequence of calls:

1. Subject's state changes
2. `Subject::notifyObservers` - calls each observer's `notify`
3. Each `Observer` calls `ConcreteSubject::getState` to query the state and react accordingly.

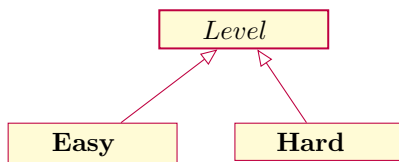
26.1 Factory Method Pattern

- when you don't know exactly what kind of object you want, and your precedences may vary.
- also called the virtual constructor Pattern.
- Strategy Pattern applied to object creation.

Eg



- randomly generated
- more turtles in easy levels
- more bullets in hard levels



```

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

class Easy: public Level {
public:
    Enemy *getEnemy () override {
        // most turtles
    }
};

class Hard: public Level {

```

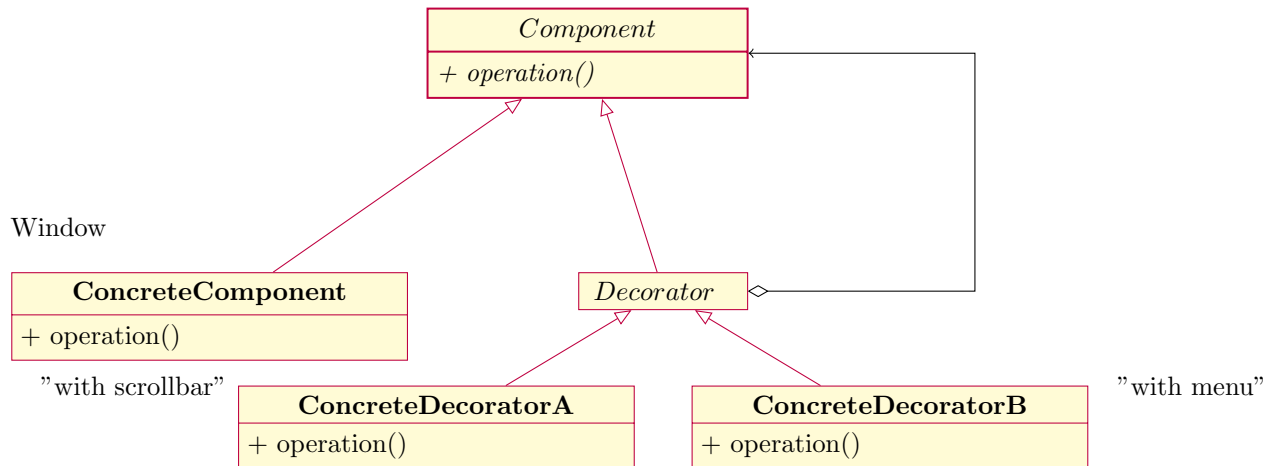
```
public:
    Enemy *getEnemy () override {
        // most bullets
    }
};

Level *l = ...;
Enemy *e = l->getEnemy();
```

26.2 Decorator Pattern

add/remove functionality to/from objects dynamically.

Eg add menu/scroll bar to basic windows - without a combinatorial explosion of subclasses



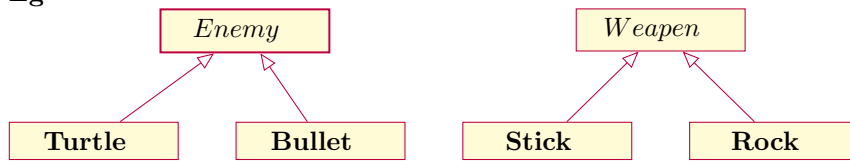
- Every a Decorator IS a Component and HAS a Component.
- WindowWithScrollbar IS a Window and has a ptr to the underlying plain window.
- Window w/ Scrollbar & menu is a Window and has a ptr to Window w/ scrollbar, which has a ptr to a Window.

```
WindowInterface *w = new WindowWithMenu{ new WindowWithScrollbar { new Window {}}};
```

26.3 Visitor Pattern

- for implementing double dispatch
- method chosen based on the runtime type of 2 objects, rather than just one.

Eg



- effect of striking an enemy with a weapon depends on both the enemy and the weapon
- C++
 - virtual methods are dispatched on the type of the receiver object, and not the method parameters.
 - No way to specify 2 receiver objects.

Visitor Pattern - combine overriding with overloading to do 2-stage dispatch

```

class Enemy {
public:
    virtual void beStruckBy(Weapon &w)=0;
};

class Turtle: public Enemy {
    void beStruckBy(Weapon &w) override {w.strike(*this);}
};
class Bullet: public Enemy {
    void beStruckBy(Weapon &w) override {w.strike(*this);}
}; // *this has a different type in each body

class Weapon {
public:
    virtual void strike(Turtle &t)=0;
    virtual void strike(Bullet &b)=0;
};

class Stick: public Weapon {
public:
    virtual void strike(Turtle &t) { /* strike Turtle with Stick */ }
    virtual void strike(Bullet &b) { /* strike Turtle with Bullet */ }
};
// etc

Enemy *e = new Bullet {...};
Weapon *w = new Rock {...};
e->beStruckBy(*w); // What happens?
  
```

Bullet::beStruckBy runs (virtual method dispatch)

- calls Weapon::strike(Bullet &) (known at compile time - overload resolution) (*this is Bullet)
- virtual method resolves to Rock::strike(Bullet &)

Visitor can also be used to add functionality to a class hierarchy without adding new virtual methods. Add a visitor to the Book hierarchy.


```

class Book {
public:
    virtual void accept(Bookvisitor &v) {v.visit(*this);}
    ...
};
class Text: public Book {
public:
    void accept (BookVisitor &v) override { v.visit(*this); }
};
// etc

class BookVisitor {
public:
    virtual void visit (Book &b) = 0;
    virtual void visit (Text &t) = 0;
    virtual void visit (Comic &c) = 0;
};

```

Eg Categorize & count:

	For	by
	Books	Author
	Texts	Topic
	Comics	Hero

Could add a virtual method to the Book hierarchy. OR write a visitor:

```

class Catalogue: public BookVisitor {
public:
    map<string, int> theCat;
    void visit(Book &b) override{ ++theCat[b.getAuthor()]; }
    void visit(Book &b) override{ ++theCat[b.getTopic()]; }
    void visit(Book &b) override{ ++theCat[b.getHero()]; }
};

```

But it won't compile!

- Circular include dependencies book.h, BookVisitor.h include each other
- include guard prevents multiple inclusion
- Whichever ends up occurring first will refer to things not yet defined.

Know when on `#include` is actually needed!

Needless `#includes` create artificial compilation dependencies and slow down compilation - or prevent compilation all together.

Sometimes a forward class declaration is good enough.

Consider

```

class A {...}; // A.h

class B { A a; };

```

```

class C { A *a };
class D: public A { ... };
class E { A f(A); };
class F {
    A f(A a) { a.someMethod; ...}
};
class G {
    t<A> x; // When t is a template
};

```

Which need includes: B, D, F
 forward declaration is OK: C, E
 G - depends on how t uses A - should collapse to one of the other cases

Now that class F only needs an include because method f's implementation is present and use a method of A

- a good reason to keep implementation in .cc file
- where possible: forward declare in .h, include in .cc

Notice: B needs to include; C does not.

If we want to break the compilation dependency of B on A, we could translate to C.

More generally:

```

class A1 {}; class A2 {}; class A3 {};
class B {
    A1 a1;
    A2 a2;
    A3 a3;
}; // compilation dependency #include...

```

⇒

```

// b.h
class BImpl; // forward declaration only
class B {
    unique_ptr<BImpl> pImpl; // ptr to implementation
};

// bimpl.h
struct BImpl {
    A1 a1;
    A2 a2;
    A3 a3;
};

```

b.cc: methods reference pImpl→a1 a2 a3

- b.h is no longer compilation-dependent on a1.h, ...

- called the pImpl idiom

- Another advantage of pImpl - ptrs have a non-throwing swap
 - can provide the strong guarantee on a B method by
 - * copying the impl into a new Bimpl structure (heap-allocated) - method modifies the copy
 - * if anything throws, discard the new structure (easy & automatic with `unique_ptrs`)
 - * if all succeeds, swap impl struct ptrs (nothrow)
 - * previous impl automatically destroyed by smart ptr

```
class B {
    unique_ptr<Bimpl>pImpl;
    ...
    void f() {
        auto temp = make_unique<Bimpl>(* pImpl);
        temp->doSth();
        temp->doSthElse();
        std::swap(pImpl, temp); // nothrow
    } // Strong guarantee
};
```

Back to abstraction in C++

Part III

Abstraction in C++

Problem 23: Shared Ownership

Is C++ hard? No (if you are a client programmer)

But explicit memory management ...

- use vector when you need an array
- use `unique_ptr` when you need a heap object
- use stack-allocated objects as much as possible

If you followed these, you should never have to say `delete` or `delete[]`.
But `unique_ptr`s don't respect is-a.

```
unique_ptr<Base> p = unique_ptr<Derived> {...}; // OK
p->virtual_fn(); // runs derived version. Ok.
```

But

```
unique_ptr<Derived> q = ...;
unique_ptr<Base> p = std::move(q); // X
// type error - no conversion between unique_ptr<Derived> & unique_ptr<Base>
```

Easy to fix

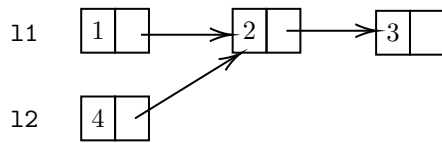
```
template <typename T> class unique_ptr {
    ...
    T *p;
public:
    ...
    template <typename u> unique_ptr (unique_ptr<u> &&q): p{q.p} {q.p=nullptr}
    template <typename u> unique_ptr &operator=(unique_ptr<u> &&q) {
        std::swap(p,q.p);
        return *this;
    }
};
// works for any unique_ptr whose pointer is assignment compatible with this->p.
// e.g. subtypes of T, but not supertypes of T
```

But I want two smart ptrs printing at the same object!
Why?

- the ptr that owns the object should be a `unique_ptr`
- all others can be raw ptrs

When would you want true shared ownership?

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

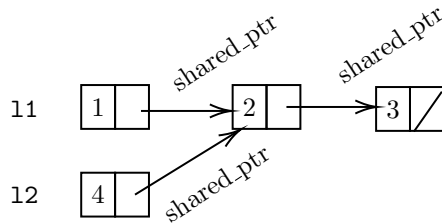


Shared data structures are a nightmare in C.

- How can we ensure each node is freed exactly once?
- Easy in garbage-collected languages.

What can C++ do?

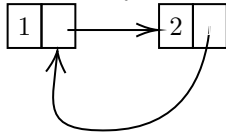
```
template <typename T> class shared_ptr {
    T *p;
    int *refcount; // refcount
    // - counts how many shared_ptrs point to *p
    // - updated each time a refcount is initialized, assigned, or destroyed
    // refcount is shared among all shared_ptrs that point to *p
    // p is only deleted when refcount reaches 0
    // implementation details - left to us
public:
    ...
};
```



```
struct Node {
    int data;
    shared_ptr<Node> next;
};
```

Now deallocation is easy - as easy as garbage collection.

Just watch: cycles



If you have cyclic data - may have to physically break the cycle (or use `weak_ptrs`)

Also watch:

```
Book *p = new ...;
shared_ptr<Book> p1 {p};
shared_ptr<Book> p2 {p};
```

- `shared_ptrs` are not mind-readers
- `p1` & `p2` will not share a refcount. BAD
- if you want to point 2 `shared_ptrs` at an object, create one `shared_ptr` and copy it.

But - you can't dynamic cast `shared_ptrs`.

```
template<typename T, typename U> shared_ptr<T> dynamic_pointer_cast <U> &spu) {
    return shared_ptr<T> (dynamic_cast <T*>(spu.get()));
}

// similarly - const_pointer_cast, static_pointer_cast
```

Just to follow up on the `dynamic_pointer_cast` implementation that Lushman gave us as above. As was observed in class, this implementation doesn't share ref counts with other `shared_ptrs` to the same object. It can be fixed, but it requires knowing about an additional constructor that `shared_ptr` provides:

```
if (dynamic_cast<T*>(spu.get()) return shared_ptr<T>{spu, static_cast<T*>(spu.get())};
else return shared_ptr<T>{nullptr};
```

The constructor that was used here basically says, "share ref counts with this `shared_ptr`, but point here instead".

Problem 24: Abstraction over Iterators

I want to jump ahead n spots in any iterator.

```
template <typename Iter> Iter advance(Iter it, size_t n);
```

How should we do it?

```
for (size_t i = 0; i < n; ++i) ++it;
return it;
```

Slow - $O(n)$ time. Can't just say `it+=n`?
Depends

- for vectors, yes - $O(1)$ time
- for lists, no - `+=` not supported (and if it was, it would still be a loop)

Related - can we go backwards?

- vectors: yes
- lists: no

So all iterators support `!=`, `*` `++`. - but some iterators support other operations.

- `list::iterator` - called a forward iterator - can only go one step forward.
- `vector::iterator` - called a random access iterator - can go anywhere (supports arbitrary ptr arithmetic)

How can we write `advance` to use `+=` for random access iterators, and `loop` for forward iterators?

Since we have different kinds of iterators, create a class hierarchy.

```
struct input_iterator_tag {};
struct output_iterator_tag {};
struct forward_iterator_tag: public input_iterator_tag {};
struct bidirectional_iterator_tag: public forward_iterator_tag {};
```



```
struct random_access_iterator_tag: public bidirectional_iterator_tag {};
```

To associate each iterator class with a tag - could use inheritance.

eg

```
class List {
...
public:
    class iterator: public forward_iterator_tag {...};
};
```

But

- makes it hard to ask what kind of iterator we use (can't `dynamic_cast` - no vtables!)
- doesn't work for iterators that aren't classes (e.g. ptrs)

Instead - make the tag a member:

```
class List {
...
public:
    class iterator {
...
    public:
        typedef forward_iterator_tag iterator_category;
        // convention - every iterator class will define a type member called
        iterator_category.
...
    };
};
```

- also doesn't work for iterators that aren't classes
- but we aren't done yet
- make a template that associates every iterator type with its category.

```
template<typename It> struct iterator_traits {
    typedef It::iterator_category iterator_category;
};
```

won't compile for the same reason as on page 66. Instead, we have

```
template<typename It> struct iterator_traits {
    typedef typename It::iterator_category iterator_category;
};
// (typename) What is this doing here?
```

needed so that C++ know that `It::iterator_category` is a type (remember: compiler doesn't know anything about `It`).

Consider:

```
template <typename T> void f() {
    T::something x; // only makes sense if T::something is a type
}

// But
template <typename T> void f() {
    T::something *x;
    // is this a ptr or a multiplication?
    // Can't know unless you whether T::something is a type.
    // C++ assumes it's a value unless told otherwise.
}

template <typename T> void f() {
    typename T::something x;
    typename T::something *y;
}
```

Need to say `typename` whenever you refer to a member type of a template parameter.

(eg

```
iterator_traits<List<T>::iterator>::iterator_category => forward_iterator_tag
```

)

Specialized version for ptrs:

```
template <typename T> struct iterator_traits<T*> {
    typedef random_access_iterator_tag iterator_category;
};
```

For any iterator type `T`, `iterator_traits<T>::iterator_category` resolves to the tag struct type for `T` (including if `T` is a ptr)

What do we do with this?

```
template <typename Iter> Iter advance (Iter it, int n) {
    if (typeid(typename iterator_traits<Iter>::iterator_category) == typeid(
        random_access_iterator_tag)) {
        return it+=n;
    }
    else if ( ... )
        ...
}
// Doesn't compile.
```

If the iterator is not random access, and `.` doesn't have a `+=` operator, `it+=n` will cause a compilation error, even though it will never be used.

Moreover - the choice of which implementation to use is being made to run-time, when the right choice is known at compile-time.

To make a compile-time decision: [overloading](#)

```
template <typename Iter>
Iter doAdvance(Iter it, int n, random_access_iterator_tag) {
    return it += n;
}

template <typename Iter>
Iter doAdvance(Iter it, int n, bidirectional_iterator_tag) {
    if (n > 0) for (int i = 0; i < n; ++i) ++it;
    else if (n < 0) for (int i = 0; i < -n; ++i) --it;
    return it;
}

template <typename Iter>
Iter doAdvance(Iter it, int n, forward_iterator_tag) {
    if (n >= 0) {
        for (int i = 0; i < n; ++i) ++it;
        return it;
    }
    throw SomeError();
}

template <typename Iter> Iter advance (Iter it, int n) {
    return doAdvance(it, n, iterator_traits<Iter>::iterator_category{});
}
```

Now the compiler will select the fast `doAdvance` for random access iterators, the slow `doAdvance` for bidirectional iterators, and the throwing `doAdvance` for forward iterators.

These choices made at compile-time - no runtime cost.

Using template instantiations to perform compile-time computation.
- called template metaprogramming.

28.1 Template Metaprogramming

C++ templates form a functional language that operates at the level of types.

Express conditions via overloading, repetition via recursive template instantiation.

```
template <int N> struct Fact {
```

```
    static const int value = N*Fact<N-1>::value;
};
template <> struct Fact <0> {
    static const int value = 1;
};

int x = Fact <5>::value; // 120 - evaluated at compile-time
```

But for compile-time computation of values, C++11/14 offer a more straightforward facility.

```
constexpr int fact (int n) {
    if (n==0) return 1;
    else return n*fact(n-1);
}
```

constexpr functions:

- evaluate the function at compile-time if n is a compile-time constant
- else evaluate at run-time
- A constexpr function must be something that actually can be evaluated at compile-time.
 - can't be virtual, can't mutate non-local variables.

Let's solve a problem.

Problem 25: I want an even faster vector

In the good old days of *C*, you could copy an array (even an array of struct!) very quickly by calling `memcpy` (like `strcpy`, but for arbitrary memory, not just strings).

`memcpy` was probably written in assembly, and was as fast as the machine could possibly be.

In C++ - copies invoke copy ctors, which are function calls.

In C++, a type is considered POD (plain old data) if it

- has a trivial default ctor (equiv. `=default`)
- is trivially copyable - copies/moves, dtor have default implementations

AND is standard layout

- no virtual methods or bases
- all members have the same visibility
- no reference members
- no fields in both base class & subclass, or in multiple base classes

For POD types, semantics compatible with *C*, and `memcpy` is safe to use.
How?

```
template <typename T> class vector {
    T *theVector;
    size_t n, cap;
public:
    ...
    vector (const vector &other): ... {
        if (std::is_pod<T>::value) {
            memcpy(theVector, other.theVector, n*sizeof(T));
        }
        else { //loop
        }
    }
};
```

Works, but ... condition is evaluated at run-time, but result is known at compile-time.
2nd option:

```
template <typename T> class Vector {
...
public:
    template <typename X=T> vector (enable_if<std::is_pod<X>::value, const T&>::type
        other): ... {
        memcpy(theVector, other.theVector, n*sizeof(T));
    }
    template <typename X=T> vector (enable_if<!std::is_pod<X>::value, const T&>::type
        other): ... {
        // placement new loop
    }
};
```

How does this work?

```
template<bool b, typename T> struct enable_if;
template<typename T> struct enable_if<true, T> {
    using type=T;
};
```

With metaprogramming, what don't you say is as important as what you do say.

If b is true, `enable_if` defines a struct whose 'type' member typedef is T.

So if `std::is_pod<X>::value = true`, then `enable_if<std::is_pod<x>::value, const T&>::type`
 \implies `const T&`.

If b is false, then the struct is declared, but not defined.

So `enable_if<b, T>` won't compile.

So one of these two versions of the copy ctor (the one with the false condition) won't compile.

Then how is this a valid program?

C++ rule SFINAE - Substitution Failure Is Not An Error

In other words: if t is a type, `template <typename T> .. f (...) {...}` is a template function, and substituting `T=t` resulting in an invalid function, the compiler does not signal on an error - it just removes that instantiation from consideration during overload resolution.

On the other hand - if no version of the function is in scope to handle the overload call, that is an error.

Q Why is this wrong:

```
template <typename T> class vector {
...
public:
    vector(typename enable_if<std::is_pod<T>::value, const T&>::type other): ... { ... }
    vector(typename enable_if<!std::is_pod<T>::value, const T&>::type other): ... { ... }
};
```

i.e. why do we need the extra template out front?

Because SFINAE applies to template functions, and these methods are ordinary functions (ctors), not templates.

- they depend on T, but T's value was determined when you decided what to put in the vector
- if substituting T=t fails, it invalidates the entire vector class, not just the method.

So make the method a separate template, with a new arg X, which can be defaulted to T and do `is_pod<X>`, not `is_pod<T>`.

So it compiles, but when you run it, it crashes.

Why? Hint: if you put debug statement in these copy ctors, they don't print.

Ans: we are getting the compiler-supplied copy ctor, which is doing shallow copies.

- These templates are not enough to suppress the auto-generated copy ctor.
 - and a non-templated match is always preferred to a templated one.

What do we do about it? Could try:

```
template <typename T> class Vector {
...
public:
    vector(const vector &other)=delete; // i.e. disable the copy ctor
    ...
}; // not allowed - can't disable the copy ctor + then write one.
```

Solution that works: overloading.¹

```
template <typename T> class vector {
...
    struct dummy {};
public:
    vector (const vector &other): vector{other, dummy{}} {} // ctor delegation
    template<typename X=T> vector(typename enable_if<std::is_pod<T>::value, const vector
        <T>&>::type other, dummy): ... { ... }
    template<typename X=T> vector(typename enable_if<!std::is_pod<T>::value, const
        vector <T>&>::type other, dummy): ... { ... }
};
```

- overload the ctor with an unused "dummy" arg.
- have the copy ctor delegate to the overloaded ctor.
- copy ctor is inline, so no function call overhead.

this works

Can write some "helper" definitions to make `is_pod` and `enable_if` easier to use

¹The solution below corrected the typos above: missing type vector

```

template <typename T> constexpr bool is_pod_v = std::is_pod<T>::value;
template <bool b, typename T> using enable_if_t = typename enable_if<b, T>::type;
////////////////////////////////////
template <typename T> class vector {
..
public:
..
    template<typename X=T> vector (enable_if_t <is_pod_v<X>, const vector <T>&& other,
        dummy): ... {...}
    template<typename X=T> vector (enable_if_t <!is_pod_v<X>, const vector <T>&& other,
        dummy): ... {...}
};

```

We now have the machinery to implement `std::move` and `std::forward`.

29.1 Move/Forward Implementation

`std::move` - first attempt

```

template <typename T> T&& move(T &&x) {
    return static_cast <T &&> (x);
}

```

- doesn't quite work - `T&&` is a universal reference, not an rvalue ref.
If `T` was an lvalue ref, `T&&` will be an lvalue ref.
- Need to make sure that `T` is not an lvalue ref - if `T` is an lvalue ref, get rid of the ref.

`std::move` - correct version

```

template <typename T> inline constexpr std::remove_reference<T>::type && move(T && x) {
    return static_cast<std::remove_reference<T>::type &&>(x);
    // std::remove_reference<T>::type      turns t&, t&& into t
}

```

Q can we save typing and use `auto`?

```

template <typename T> auto move(T &&x) { ... }

```

No! By-value `auto` throws away refs and other constns.

```

int z;
int &y = z;
auto x = y; // x is int

const int &w = z;
auto v = w; // v is int

```


- uses the type a value would have in exprs if copied.

By-ref `auto&&` is a universal reference.

- need a type declaration rule that doesn't discard refs.

- `decltype(...)` - returns the type that ... was declared to have
- `decltype(var)` - returns the declared type of the var
- `decltype(expr)` - returns an lvalue or rvalue ref, depending on whether expr is an lvalue or an rvalue

Eg

```
int z;
int &y = z;
decltype(y)x = z; // x is int&
x = 4; // affects z

decltype(z) s=z; // s is int
s = 5; // doesn't affect z

auto x = z; // x is int
x = 4; // does not affect z

decltype((z)) s = z; // s is int &
s = 6; // affects z
```

`decltype(auto)`

- perform type deduction like auto, but use the decltype rules.
- i.e. don't throw away ref.

Correct move:

```
template <typename T> inline constexpr decltype(auto) move (T &&x) {
    // as before
}
```

`std::forward` -first attempt

```
template <typename T> T && forward (T &&x) {
    return static_cast<T &&>(x);
}
```

/ Reasoning - if x is an lvalue/rvalue, `T&&` is an lvalue/rvalue reference.

Doesn't work - forward is called on expressions that are lvalues, but may point at rvalues.

eg

```
template <typename T> void f (T &&y) { // lvalue/rvalue ref
    ... forward(y) ... // y is an lvalue
}
```

In order to work, forward must know what types was deduced for T.

So forward<T>(y) would work.

So - can we prevent automatic type deduction for forward?

```
template <typename T> struct identity {
    using type = T;
};
template <typename T> T &&forward (identity<T>::type &&x) { ... }
// identity<T>::type    makes type deduction impossible. but this ref is no longer
// universal.
```

instead - separate lvalue/rvalue cases.

```
template <typename T> inline constexpr T & forward(std::remove_reference_t<T> &x)
    noexcept {
    return static_cast<T &>(x);
}
template <typename T> inline constexpr T && forward(std::remove_reference_t<T> &&x)
    noexcept {
    return static_cast<T &&>(x);
}
```

Problem 26: Collecting Stats

I want to know how many Students I create

```
class Student {
    int assns, mt, final;
    static int count; // static - associated with the class - not one per object
public:
    Student( ... ): ... { ++ count; }
    // accessors.
    static int getCount() { return count; }
    // static methods - have no 'this' parameter.
    // - not really methods - scope functions
};

int Student::count = 0; // must define the variable (.cc file)
Student s1 {...}, s2 {...}, s3 {...};
cout << Student::getCount(); // 3
```

Now I want to count objects in other classes. How can we abstract this solution into reasonable code?

```
template <typename T> struct Count {
    static int count;
    Count () { ++count; }
    Count (const Count &) { ++count; }
    Count (Count &&) { ++count; }
    ~Count() { --count; }
    static int getCount { return count; }
};
template <typename T> int Count <T>::count = 0;
```

```
class Student: Count<Student> { // private inheritance - inherits Count's
    implementation without creating is-a relationship.
    // - members of Count become private in Student
    int assns, mt, final;
public:
    Student( ... ): ... { ... }
    // accessors
```

```
using Count::getCount; // make Count::getCount visible
};

class Book: Count<Book> { ...
public:
    using Count::getCount;
};
```

Why is `Count` a template?

So that for each class `C`, `class C: Count<C>` creates a new, unique instantiation of `Count`, for each `C`. This gives `C` its own counter, vs. sharing one counter over all classes.

This technique - inheriting from a template specialized by yourself - looks weird, but happens enough to have its own name - the Curiously Recurring Template Patter (CRTP).

What is CRTP good for?

Problem 27: Resolving Method Overrides At Compile-Time

Recall: Template Method Pattern

```
class Turtle {
public:
    void draw {
        drawHead();
        drawShell();
        drawTail();
    }
private:
    void drawHead();
    virtual void drawShell()=0; // vtable lookup
    void drawTail();
};

class RedTurtle: public Turtle {
    void drawShell() override;
};
```

Consider

```
template <typename T> class Turtle {
public:
    void draw() {
        drawHead();
        static_cast <T*>(this)->drawShell();
        drawTail();
    }
private:
    void drawHead();
    void drawTail();
};

class RedTurtle: public Turtle<RedTurtle> {
    friend class Turtle;
    void drawShell();
};
```

- no virtual methods, no vtable lookup

```
class GreenTurtle: public Turtle<GreenTurtle> {  
    friend class Turtle;  
    void drawShell();  
};
```

Drawback - no relationship between RedTurtle and GreenTurtle. \implies can't store a mix of them in one container.

Can give Turtle a parent:

```
template <typename T> class Turtle: public Enemy { ... };
```

Then we can store RedTurtles with GreenTurtles

- But can't access draw method
- Could give Enemy a virtual draw.

```
class Enemy {  
public:  
    virtual void draw()=0;  
};
```

- but then there will be a vtable lookup
- On the other hand, if `Turtle::draw` calls several would be virtual helpers, could trade several vtables lookups for one.

Problem 28: Polymorphic Cloning

```
Book *pb = ...;
Book *pb2 = ...; // I want an exact copy of *pb
```

Can't call a ctor directly - don't know what *pb is \implies which ctor to call.

Standard solution virtual clone method

```
class Book {
    ...
public:
    virtual Book *clone { return new Book {*this}; }
};

class Text: public Book {
    Text *clone override { return new Text { *this }; }
};

// comic - similar
```

Can we reuse code?

Works better with an abstract base class:

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

template <typename T> class Book_cloneable: public AbstractBook {
public:
    T *clone() override { return new T { static_cast <T&>(*this)}; }
    // T should be AbstractBook to make it compile
};
```

```
class Book: public Book_cloneable<Book> { ... };  
class Text: public Book_cloneable<Text> { ... };  
class Comic: public Book_cloneable<Comic> { ... };
```

If you are curious, see [enable_shared_from_this](#).

Problem 29: Logging

Want to encapsulate “logging” functionality and “add” it to any class.

```
template <typename T, typename Data> class Logger {
public
    void loggedSet (Data x) {
        cout << "Setting data to " << x << endl;
        static_cast<T *>(this)->set(x);
    }
};

class Box: public Logger <Box, int> {
    friend class Logger;
    int x;
    void set (int y) { x = y; }
public:
    Box(): x{0} {loggedSet(0)};
};

Box b;
b.loggedSet(1);
b.loggedSet(4);
b.loggedSet(7);
```

Another approach:

```
class Box {
    int x;
public:
    Box(): x{0} {}
    void set(int y) { x=y; }
};

template <typename T, typename Data> class Logger: public T {
public:
    void loggedSet(Data x) {
        cout << "Setting data to " << x << endl;
        set(x);
    }
};
```

```
using BoxLogger = Logger <Box, int>;
```

```
BoxLogger b;  
b.loggedSet(1);  
b.loggedSet(4);  
b.loggedSet(7);
```

- again - no virtual method overhead.

33.1 Mixin inheritance

Note if `SpecialBox` is a subclass of `Box`, then `Logger <Box, int>` is not a subclass of `SpecialBox` in the second solution. - Also `Logger <Box, int>` is not a subtype of `Logger <Box, int>`.

But in the first solution - `SpecialBox` is a subtype of `Logger <Box, int>` \implies can specialize behaviour via virtual functions.

Problem 30: Total Control

C++ lets you control how pretty much everything is done - copies, parameter passing, initialization, method call resolution, etc.

Why not go the rest of the way and take control over memory allocation?
Memory allocation are tricky to write. So 2 questions: Why? and How?

Why might you need a custom allocator?

- built-in one is too slow.
 - general purpose - not optimized for any specific use.
 - e.g. if you know you will always allocate objects of the same size, a custom allocator may perform better.
- optimize locality
- to use “special memory”
- to profile your program (collect stats)

How do you customize allocation?

- overload operator `new`
- if you write a global operator `new`, all allocations in the program will use your allocator
- also write operator `delete` - else undefined behaviour.

Eg

```
void *operator new(size_t size) {
    cout << "Request for " << size << " bytes\n";
    return malloc(size);
}
void operator delete(void *p) {
    cout << "Freeing " << p << endl;
    free(p);
}

int main() {
    int *x = new int;
    delete x;
}
```

```
}

```

Works

- but is not correct. Does not adhere to connection. If operator `new` fails, it is supposed to throw `bad_alloc`.
- actually, if operator `new` fails, it is supposed to call the `new_handler` function.
 - the `new_handler` function can
 - free up space (somehow)
 - install a different `new_handler`/ deinstall the current.
 - throw `bad_alloc`
 - abort/exit
- `new_handler` should be called in an ∞ loop.
- if `new_handler` is `nullptr`, then operator `new` throw.
- Also - `new` must return a valid ptr if `size==0`, and `delete(nullptr)` must be safe.

Corrected operator `new`:

```
#include <new>
void *operator new(size_t size) {
    cout << "Request for " << size << " bytes\n";
    while (true) {
        void *p = malloc(size);
        if (p) return p;
        std::new_handler h = std::get_new_handler();
        if (h) h();
        else throw std::bad_alloc {};
    }
}

void operator delete(void *p) {
    if (!p) return;
    cout << "Freeing " << p << endl;
    free(p);
}

```

Replacing global operator `new/delete` affects your entire program.
Probably want to replace these operators on a class-by-class basis.
- especially if optimizing for the size of your objects.

Define operator `new/delete` within the class - must be static members.

```
class C {
public: ...
    static void *operator new(size_t size) {
        cout << "Running C's allocator\n";
        return ::operator new(size);
    }
    static void operator delete(void *p) noexcept {
        cout << "Freeing " << p << endl;
    }
}

```

```

    return ::operator delete(p);
}
};

```

Can we write to an arbitrary stream?

```

class C {
public:
    static void *operator new(size_t size, std::ostream &out) {
        out << ... << endl;
        return ::operator new(size);
    }
    ...
};

C *x = new(cout) C;
ofstream f {...};
C *y = new(f) C;

```

Must also write the corresponding delete:

```

class C {
...
    static void operator delete(void *p, std::ostream &out) noexcept {
        out << ... << endl;
        return ::operator delete(p);
    }
};

```

Won't compile! You also need ordinary delete.

```

class C {
...
    static void operator delete(void *p, std::ostream &out) noexcept {
        out << ... << endl;
        return ::operator delete(p);
    }
};

C *p = new (f) C; // ofstream allocator
delete p; // ordinary delete

```

- Must have ordinary delete - else compile error.

How can you call the specialize one? You can't.

Then why do you need it?

If the ctor that runs after specialized new - throws - then specialized operator delete is called.

If there isn't one \implies no delete is called \implies leak

Problem 31: Total Control over Vectors & Lists

Incorporating custom allocators into containers.

Make the allocator an argument to the template. - With a default value.

```
template <typename T, typename Alloc = Allocator<T>> class Vector { ... };

template <typename T> class allocator {
public:
    // big 5 (no fields)
    T *address(T &x) { return &x; }
    T *allocate(size_t n) { return ::operator new(n*sizeof(T)); }
    void deallocate(T *p) { ::operator delete(p); }
    void construct, destroy // placement versions
}
```

Lists - not so easy

- same Alloc parameter
- want to allocate Node, not T's

```
template <typename T, typename Alloc = Allocator<T>> class lists {
    typename Alloc::rebind<Node>::other alloc;
    // same as before
};

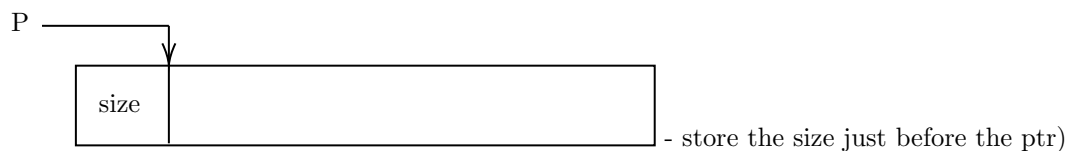
template <typename T> class allocator {
public:
    ...
    template <typename U> struct rebind {
        using other = allocator<U>;
    };
};
```

Problem 32: A fixed-size allocator

Let's write one.

Expect: faster than built-in.

- no need to keep track of sizes (aside: many traditional allocators:



- saves space (no hidden size field)
- saves time (no hunting for a block of the right size)

Create a pool of memory - an array large enough to hold n T objects.

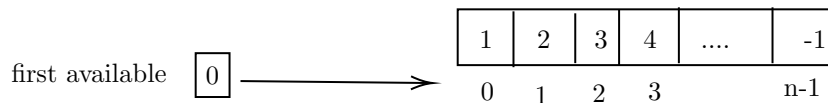
			...	
--	--	--	-----	--

When a slot in the array is given to the client, it will act as a T object.

When we have it, it will act as a node in a linked list.

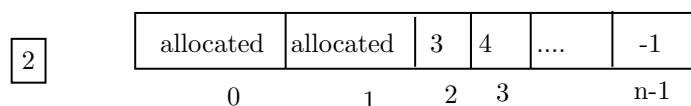
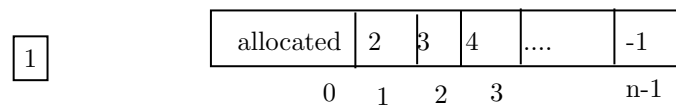
Store an int in each slot - index (not ptr) of the next free slot. Store the index of the first slot.

Initially

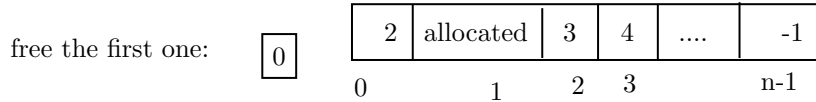


Each node store the index of the next one.

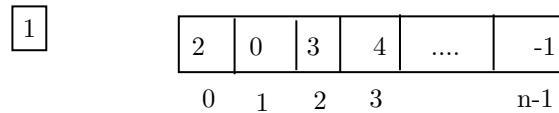
Allocation from the front



Deallocation: add to front of list



free the second one:



Alloc/dealloc: both constant time & very fast

Implementation:

```
template <typename T, int n> class FixedSizeAllocator {
    union Slot {
        int next;
        T data;
        Slot(): next{0} {}
    };
    // Slot - large enough to hold a T. - but also usable as an int.
    Slot theSlots[n];
    int firstAvailable = 0;
public:
    fixedSizeAllocator() {
        for (int i = 0; i < n-1; ++i) theSlots[i].next = i + 1;
        the Slots[n-1] = 1;
    }
    T *allocate() noexcept {
        if (firstAvailable == -1) return nullptr;
        T *result = &(theSlots[firstAvailable].data);
        firstAvailable = theSlot[firstAvailable].next;
        return result;
    }
    void deallocate(void *item) noexcept {
        int index = (static_cast<char*>(item) - reinterpret_cast<char*>(theSlots))/sizeof
        (Slot);
        theSlots[index].next = firstAvailable;
        firstAvailable = index;
    }
};
```

```
class Student final {
    int assns, mt, final;
    static fixedSizeAllocator<Student, SIZE/* How many slots do you want? */> pool;
public:
    Student( ... ) : ...
    static void *operator new(size_t size) {
```



```

    if(size != sizeof(Student)) throw std::bad_alloc{};
    while (true) {
        void *p = pool.alloc();
        if (p) return p;
        auto h = std::get_new_handler();
        if (h) h();
        else throw std::bad_alloc{};
    }
}
static void operator delete (void *p) noexcept {
    if (!p) return;
    pool.deallocate(p);
}
};
fixedSizeAllocator<Student, SIZE> pool;

// main

int main() {
    Student *s1 = new Student;
    Student *s2 = new Student;
    // custom allocator
    delete s1;
    delete s2;
    // custom deallocator
}

```

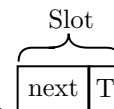
Q where does the memory for s1, s2 reside?

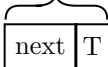
A static Memory (NOT the heap)

- Depending on how you build your allocator, the pool can be in static memory, on the stack, or on the heap.

Notes

- use a union to treat a slot as both an `int` and a T object
- ensures no memory is wasted on bookkeeping.



- alternative: store “next” index adjacent to the T object, i.e.  i.e. struct instead of union.
- advantage of a union: no wasted memory.
- disadvantage: if you access a dangling T ptr, you can corrupt the linked list.

eg

```

Student *s = new Student;
delete s;
s->setAssns( ... ); // will probably corrupt the list

```

(Lesson: following dangling ptrs can be VERY dangerous)

With a strut, 'next' filed is before the ptr, so you have to work harder to corrupt it: e.g. `reinterpret_cast<int*>(s)[-1]=...`

Unions - if one field has a ctor, the union needs a ctor - since it's a union, the ctor should only initialize one filed.

On the other hand - if you have a struct - you will have difficulty if T has no default ctor.

Eg

```
struct Slot {
    int next;
    T data;
};
Slot theSlots[n]; // can't do this if T has no default ctor
```

Can't do operator new/placement new - we're writing operator new!

Also - Why indexes vs ptrs?

- `ints` are smaller that ptrs
- So we waste no memory as long as T is at least as large as an `int`.
- We do waste memory if T is smaller that an `int`.
- Could use short, char for the index (if this allows enough items)
- Could make the index type a parameter of the allocator template.

Why is class Student final?

- fixed-size allocator
- subclass may be larger - won't wok with our allocator, but would still use it.
- options
 1. make the class final
 2. check the size, throw if the size is not right
 3. check the size, use global operator new/delete if size is not right
 4. give subclass its own allocator

Problem 34: I want a (tiny bit) smaller vector class

`vector/vector_base` have an allocator field `a` which itself has no fields.

What is its size? 0? No - not allowed in C++. \therefore all types size > 1 .

Practically - compiler inserts a dummy character.

So the allocator make the vector one byte larger. (possibly more, due to alignment)

To save this space - C++ provides: empty base optimization (EBO).

- an empty base class does not have to occupy space in an object.

So make the allocator a base class of vector. At the same time, make `vector_base` a superclass.

```
template <typename T, typename Alloc=allocator<T>> struct vector_base: Alloc {
    size_t n, cap;
    T *v;
    using Alloc::allocate;
    using Alloc::deallocate;
    // etc
    // Because we don't know anything about Alloc, we need to bring these members into
    // scope.
    ...
};
template <typename T, typename Alloc=allocator<T>> class vector: private /*no is-a*/
    vector_base <T, alloc> {
    using vector_base<T, alloc>::n;
    using vector_base<T, alloc>::cap;
    using vector_base<T, alloc>::v;
    ...
};
```

Details are left as exercise.

37.1 Something Missing from Last Year

So we can eliminate the space cost of an allocator by making it a base class. At the same time, make `vector_base` a base class of a `vector`.

```
template <typename T, typename Alloc = allocator<T>>
struct vector_base: private Alloc { // struct has default public inheritance
```

```

size_t n, cap;
T *v;
using Alloc::allocate;
using Alloc::deallocate;
// etc.
vector_base(size_t n): n{0}, cap{n}, v{allocate(n)} {}
~vector_base() { deallocate(v); }
};

template <typename T, typename Alloc = allocator<T>>
class vector: vector_base<T, Alloc> { // private inheritance - no is-a relation
    using vector_base<T, Alloc>::n;
    using vector_base<T, Alloc>::cap;
    using vector_base<T, Alloc>::v;

    using Alloc::allocate; // or say this->allocate
    using Alloc::deallocate; // this->deallocate
public:
    ... Use n , cap, v instead of vb.n, vb.cap, vb.v
};

```

uninitialized_copy, etc. - need to call construct/destroy
- Simplest - let the take an allocator as a parameter

```

template <typename T, typename Alloc> {
    void uninitialized_fill(T *start, T *finish, const T &x, Alloc a) {
        ...
        a.construct(...)
        ...
        a.destroy(...)
        ...
    }
}

```

How can vector pass an allocator to these functions?

```

uninitialized_fill(v, v + n, x, static_cast<Alloc>(*this)); // Cast yourself to
base class reference

```