1.
Basic C++

Introduction to C++

(1.1)
C++ Standards

There are multiple versions of C++ *standards*. For instance, "C++14" specifies the 2014 C++ standard, which is the version that we are going to use in this course.

(EX 1.2)
Hello World

We are going to write the `helloworld` program in C in C++ to compare two languages. First in C:

helloworld.c

```c
#include <stdio.h>

int main() {
    printf("Hello world!\n");
    return 0;
}
```

And in C++:

helloworld.cc

```cpp
#include <iostream>

int main() {
    std::cout << "Hello world!" << std::endl;
    return 0;
}
```

Note that we included a different library in C++, called `<iostream>`. The library name does not end in `.h`, since `.h` is used to indicate that the library is written in C. Although C++ is mostly backwards compatible with C, whenever possible, it is (obviously) better to use C++ way of writing code, rather than C way of doing things.

Another notable difference between the above two codes is how they write to the output stream. In C, `printf` always writes to the `stdout`. On the other hand, we have to specify the name of the `stdout` in C++. The full name of the `stdout` in C++ is `std::cout`. The `std::` specifis that `cout` is defined in the *standard namespace*, or the namespace `std`. The same is true for `std::endl`, which combines the appropriate newline character[1] with a request to *flush* the buffer.

The `main` routine *must* be declared in C++ to return an integer (i.e. we cannot write `void main()` ...), the usual convention is to have it return 0 when the program ran successfully, and nonzero otherwise. The line `retrun 0;` is optional; if we omit it, the compiler automatically insert it for us.

Writing `std::` in front of everything gets tedious, and there are two ways to handle this:

Omitting `std::` (1)

```cpp
#include <iostream>
using std::cout; // whenever cout occurs, replace it with std::cout
using std::endl; // whenever endl occurs, replace it with std::endl

int main() {
    cout << "Hello world!" << endl;
    return 0;
}
```

[1]Different OS uses different newline characters. For Unix and Unix-like systems (e.g. Linux, MacOS, ...), `\n` is used.

Omitting `std::` (2)

```
#include <iostream>
using namespace std; // replace every name that occurs in std to std::name

int main() {
    cout << "Hello world!" << endl;
    return 0;
}
```

Although the second version requires less typing, there are good reasons not to do it. We shall revisit this when we discuss about the *preprocessor* in more detail.

(1.3) Compilation

Unlike a bash script, we cannot run a C++ program directly. Instead, we first have to use a *compiler* to create an executable file.

Def'n 1.1. **Compiler**

A ***compiler*** takes a high-level programming language files and produces machine-readable instructions by preprocessing, compiling,[a] assembling,[b] and linking.[c]

[a] *Compiling* is transforming a C++ code into an assembler code.
[b] *Assembling* is transforming an assembler code into a machine code.
[c] *Linking* is combining several pieces of machine code – called *object files* – into a single executable file.

The compiler we will be using is the GCC compiler, where `g++` specifise that we are programming in C++ but not in C. Note that unless we are programming on precisely the same OS and architecture with precisely the same library versions, a program compiled in one machine will not work on another.

To make an executable file, we write:

Compiling `helloworld.cc`

```
~$ g++ helloworld.cc
```

on shell, and the compiler will create an executable named `a.out` by default. To specify the compiler to follow the C++14 standard, we write:

Specifying C++14 Standard

```
~$ g++ -std=c++14 helloworld.cc
```

Most times we want our executable program to have a meaningful command. `-o` flag is used for this purpose:

Giving a Name to the Executable File

```
~$ g++ -std=c++14 helloworld.cc -o helloworld
```

There other options that we can pass in, such as `-g`, `-c`, `-Watt`, `-Wextra`, and `-D`. See `man gcc` for more information. It is also recommanded to learn a debugging tool such as `gdb`.

(1.4) Input and Output

There are three global variables in `std` that define the stream objects used for basic input and output.

(a) `cin`: reads from `stdin`.

(b) `cout`: writes to `stdout`.

(c) `cerr`: writes to `stderr`.

Output streams use the `<<` operator to send output to the stream, where the stream name must appear on the LHS of the operator, while the information to output must be on the RHS (e.g. `cout<<"hi"`). Input streams use the `>>` operator to read input from the stream. The position of the stream and information remains the same as `<<`, but the direction of the *flow* of the information is different.

The standard input stream object, `cin`, has several bits that it uses to track if an error has occurred, or the `eof` has been detected. To access these bits, we use

`.fail()` and `.eof()`

```
if ( cin.fail() ) {...} // if cin fails
if ( cin.eof() ) {...} // if cin detects the eof
```

Note that we must *attempt* to read first before testing for `eof`.

(EX 1.5) `readInts.cc`

We can use what we learned about IO so far to create a file `readInts.cc` that reads an integer from the `stdin` and prints back the integer to `stdout`:

`readInts.cc` (1)

```
...

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

A particular problem with the above version is that the program is not robust. In fact, it fails when it reads a non-integer value, and suddenly stops as a result. It would be more informative (to the user) to handle the error, informing the user of the problem, and still let the program continue. We begin by introducing a slight modification.

`readInts.cc` (2)

```
...

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

In C++, writing `if (!cin)` is equivalent to `if (cin.operator!())`, where `cin.operator!` by default returns the fail bit by calling `fail()`. This ability to specialize operators is called *operator overloading*, which we shall discuss later.

The next changes rely upon some details about the C++ versions of the `>>`, `<<` operators. In C, they are used to shift bits; in C++, they are *overloaded* for output and input. It turns out that the first parameter for the input stream operator is an input stream. That is, if a program has the line

```
cin >> i;
```

then this executes the command

```
operator>>(cin, i);
```

What does `operator>>` have as a return type? Note that we can use `>>` multiple times in a line, like:

```
cin >> x >> y >> z;
```

which executes

```
operator>>( operator>>( operator>>(cin, x), y ), z );
```

Thus `operator>>` has to return `cin` back,[2] and we can make our program even more compact:

readInts.cc (3)

```
...

int main(void) {
    int i;
    while ( cin >> i ) {
        cout << i << endl;
    }
}
```

Now we modify our program to skip all non-integer input. Whenever we read in a non-integer character, the fail bit in `cin` is set and it stops reading the input stream. Therefore, we have to perform the following two things:

(a) We first *clear* the fail bit through `cin.clear()`.

(b) Since the program stopped reading on a non-integer value, we have to *throw away* the offending character through `cin.ignore()`.

Also, we need to be able to detect when `eof` has been reached. With our previous version, the loop terminates whenever there is an error (i.e. reading non-integer character) or an `eof` signal. Since we do not want to exit the loop in case of an error, so we have to check for an `eof` only:

Checking `eof`

```
while ( true ) {
    ...
    if ( cin.eof() ) break;
    ...
}
```

Now we are ready to produce a robust version of `readInts.cc`:

[2]Precisely speaking, `operator>>` takes *reference variables*, which we shall discuss later.

Robust `readInts.cc`

```
...

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

(1.6) File IO

Reading from a file is (almost) identical to reading from `stdin`. Instead of including the `<iostream>` library, we need to include the `<fstream>` library and use the `ifstream` type. Writing to a file is similar, except that we use the `ofstream` type. For instance, consider the following program:

`fileIO.cc`

```
1  #include <fstream>
2  using namespace std;
3
4  int main() {
5      ifstream infile{"input.txt"};
6      ofstream infile{"output.txt"};
7      int i;
8      while (true) {
9          infile >> i;
10         if (infile.fail()) {
11             break;
12         }
13         outfile << i << endl;
14     }
15 }
```

Note that object `infile` is mapped to the text input file `input.txt`, while the output object `outfile` is mapped to the text file `output.txt`. Also, the name (e.g. `input.txt`) must be of either the type `const char *` or the type `const string &`.

Observe the use of `{}` for initializing file stream objects (see line 5, 6 of `fileIO.cc`). We shall revisit this when we discuss *uniform initialization syntax*.

Consider the following cases when running `fileIO.cc`:

(a) When an input file cannot be opened for some reason (e.g. no permission), then the `fail` bit is set, which we can check its status by using `fail()`;

(b) When an output file does not already exist when we create `outfile`, then the system automatically creates the file `output.txt` with the new contetns.

(c) When an output file exists but cannot be modified, then the `fail` bit is set, which we can check its status by using `fail()`;

The files will be automatically closed when the objects go *out of scope* and are destroyed on the filan, closing brace (i.e. `}`) for the program. This ensures that any final, buffered output will be sent to the output file. If, however, the method `open()` is used to open the file, then `close()` must be called before the program ends.

(1.7) Formatting Output

Although we would not put much emphesis on formating outputs, it is nevertheless useful to know that there are ways to do this, by using the library `<iomanip>`.

conversionChart.cc

```
#include <iostream>
#include <iomanip>
using namespace std;

int main() {
    for (int i=0; i<17; ++i) {
        cout << dec << setw(3) << i << oct << setw(3)
             << i << hex << setw(3) << i << endl;
    }
}
```

The output is being formatted in two main ways.

(a) We are using `setw` to set the width of the information to 3 characters; by default, the fill-character used to fill out the information to make sure it meets the width is the space character. As we can see below, this enables us to align the information into columns.

(b) We are also using `dec` (base 10, *decimal*), `oct` (base 8, *octal*), `hex` (base 16, *hexadecimal*) to make the integer `i` to be printed in a different *base* system.

When we compile and run the program, it produces:

```
~$ ./a.out
  0  0  0
  1  1  1
  2  2  2
  3  3  3
  4  4  4
  5  5  5
  6  6  6
  7  7  7
  8  8  8
  9 11  9
 10 12  a
 11 13  b
 12 14  c
 13 15  d
 14 16  e
 15 17  f
 16 20 10
```

Here are some notes and other formatting methods:

(a) `dec`, `oct`, `hex` features are *sticky* in a sense that once we change the integer value to b eprinted in a different base, it continues to print in that base until we tell it to be printed in another base system.

(b) `showpoint` is used to force the printing of the decimal point in floating point number.

(c) `setprecision` specifies the number of digits after the decimal point to print. By default it uses `0` as the fill character.

Building Blocks of C++

(1.8) Strings

Here is a list of comparisons between C and C++ strings:

C string	C++ string
An *array* of characters: ◦ `char *` or ◦ `char []` terminated by the null character `\0`.	Of the type `std::string`, which requires inclusion of `<string>` library. We can also write `using std::string;` after the headers to avoid having to include the entire namespace.
Memory needs to be explicitly managed (i.e. need to shrink or grow explicitly)	C++ manages memory for us.
Easy to accidently overwrite `\0` and corrupt memory	Safer to manipulate.

We can initialize strings in C++ in two ways:

String Initialization in C++

```
#include <string>
...
std::string s = "hello";
std::string name{ "George Dantzig" }
```

Note that the literal `"hello"` that appears on line 3 is actually a constant character pointer (`const char *`) but not of type `std::string`. It is *passed in* to initialize `s`, in the same way that the literal `"George Dantzig"` is used to initialize `name`.

Here is a short list of useful (and commonly used) string operations and operators. Let `s1`, `s2` be of type `std::string`.

- *equality*: `s1 == s2`
- *inequality*: `s1 != s2`
- *comparision*:[3] `s1 <= s2`, `s1 < s2`, `s1 >= s2`, `s1 > s2`
- *length*: `s1.size()`, `s1.length()`
- *fetch each character*: `s1[0]`, `s1[1]`, . . . , `s1[s1.length()-1]`
- *concatenation*: `s1 = s1+s2`, `s1 += s2`

The `<string>` library defines how IO works on the `std::string` class. Consider the following examples:

readnprintString.cc

```
#include <iostream>
#include <string>

int main() {
    string s;
    std::cin >> s;
```

[3]C++ string comparisons are *lexicographic* (i.e. in the dictionary order).

```
    std::cout << s << endl;
}
```

getline.cc

```
#include <iostream>
#include <iomanip>
#include <string>

int main() {
    int i = 1;
    string s;
    while (getline(std::cin, s)) {
        std::cout << setw(3) << i++ << ": \"" << s << "\"" << std::endl;
    }
}
```

`readnprintString.cc` reads and prints a string from `stdin`, *ignoring* every whitespace character, whereas `getline.cc` reads the whole line and stores it in `s`.

(1.9) String Streams

There is an unusual type in C++ called `stringstream`, available through the `<sstream>` library. It is a hybrid of both the `std::string` class and the IO stream classes. It lets us read to and write from strings using stream operators, although it is recommanded to use `istringstream` type for inputs and `ostringstream` for outputs. The primary purpose for using an input string stream is to take an existing string, such as the sentence `"The quick brown fox\njumped over the lazy\tdog."` and split it up into separate words (often called `tokens`). By default, the `istringstream` separates the tokens by whitespace. Since the input stream function `std::getline` lets us specify the delimiter to use when reading in lines of inputs as strings, we can use the same technique to specify a different delimeter to separate our tokens. For instance, consider running the following program.

istringstream_ex.cc

```
#include <iostream>
#include <sstream>
#include <string>

int main() {
    string s;
    string s1{"The quick brown fox\njumped over the lazy\t dog."};
    istringstream ss1{s1};
    while (ss1>>s) {
        std::cout << s << std::endl;
    }
    string s2{"George,Dantzig,Henri,Poincare,David,Hilbert"};
    istringstream ss2{s2};
    std::cout << "***" << std::endl;
    while (getline(ss2, s, ',')) {
        std::cout << s << std::endl;
    }
}
```

Another reason to use the `istringstream` is to test if a word extracted from the `istringstream` can be successfully read as an `int`. This lets us make our programs more robust. However, once we learn about `exceptions and exception handling`, we could instead use the `std::stoi` function introduced in C++11.

It is also possible to use an output string stream object `ostringstream` to build up a string from a variety of other types.

(1.10) Command-line Arguments

Command-line arguments in C++ are not different from C: the `main` function may take two arguments, `argc` and `argv`, with `argc` representing the number of arguments the program received, the `argv` the array of arguments. Unfortunately, the are *no* different than C: `argv` is a `char **` (an array of `char` pointers), rather than an array of C++ `strings`. Luckily, C++ usually converts `char *`'s into `std::strings`. The following program demonstrates command-line arguments in C++.

args.cc

```
#include <iostream>

int main(int argc, char **argv) {
    for (int argi=0; argi<argc; argi++) {
        std::cout << argv[argi] << std::endl;
    }
}
```

Running `args.cc` in Terminal

```
~$ g++ args.cc -o args
~$ ./args This is a test
./args
This
is
a
test
~S ./args "This is a test"
./args
This is a test
```

Note that `argv[0]` is the name of the program itself, and normal arguments continue from there.

In case where we need to use a command-line argument as a `string` but not a `char *`, we can convert `char *` into a `string` by writing

```
string s = string(argv[1]);
```

(1.11) Functions

Functions in C++ are very much like those in C. The basic format is the following.

Basic Format for C++ Functions

```
<return-type> <function-name> (<type1> <arg1>, <type2> <arg2>, ...) {
    <function-body>
    return <value-of-appropirate-type> // not required when <return-type> is void
}
```

Note that forgetting to have a `return` statement when a return type other than `void` is specified can lead to the program behaving erratically; using `-Wall` when compiling the program will warn us of this error.

For instance, we can write

```
add1.cc

int add1(int value) {
    return value+1;
}
```

to define a function `add1` which takes an `int` type `value` as a parameter, and returns the value with `1` added to it.

(1.12)
Foward Declarations

Recall that we separate a function into its *declaration* (i.e. *signature*) and *definition* (i.e. *implementation*). It is very common in C++ to give a list of all the necessary declarations at the beginning, and then provide the definitions later. This is especially useful when the functions or types refer to each other. We shall revisit this idea when we get to *separate compilation* and the role of the *preprocessor*.

As an example,

```
1  bool even(unsigned int n) {
2      if (n==0) return true;
3      return odd(n-1);
4  }
5
6  bool odd(unsigned int n) {
7      if (n==0) return false;
8      return even(n-1);
9  }
10 ...
```

would not compile, since the function `odd` is not *declared* when the program reaches line 3. To resolve this issue, we simply declare the `odd` function first:

```
bool odd(unsigned int n);

bool even(unsigned int n) {
...
```

(1.13)
Function Overloading

C++ allows a function to be *overloaded*. That is, we can have more than one function with the exact function name so long as the number of arguments or their types are different. In fact, we have seen this before with our input and output operators, >> and <<. This is convenient, since it means we do not have to come up with a new function name just because we need anotehr version that takes a different combination of paramenters. One should note, however, that the compiler does *not* distinguish between the overloaded functions based upon return types.

(1.14)
Default Parameters

It is a common technique to have more than one version of a function, that mostly does the same thing, but we add one or more parameters (i.e. *flags*) to turn on or off some additional behavior. The key is that the versions are extremely similar. C++ provides a mechanism called *default parameters* that lets us do this. The idea is that some of parameters have a default value that the compiler will use, unless an explicit value is provided.

```
void f(int n=10);
```

By convention, we usually list the default values in the function declaration but not in the function definition.

Note that the parameters with default values must come *after* the parameters without default values in the list of arguments.

(1.15) Structures

C++, for compatibility reasons, lets us use the `struct` keyword just as one would in C. However, there are slight differences. For instance, we would write

Singly-linked List in C

```
1 typedef struct Node_t {
2     int value;
3     struct Node_t * next;
4 } node;
5
6 Node * head = NULL;
7 Node n;
8 n.value = 5;
9 n.next = NULL;
```

to create a `Node` structrue for singly-linked lists in C, whereas

Singly-linked List in C++

```
1 struct Node {
2     int value;
3     Node * next;
4 };
5
6 Node * head = nullptr;
7 Node n{ 5, nullptr };
```

in C++. Note the following differences.

- C++ does not require that the type definition start with the keyword `typedef`.
- Similarly, the keyword `struct` needs not be repeated in all of the variable declarations (compare line 3's). It just needs to be done once, in the initial type definition.
- C++ no longer uses `NULL`; rather we should be using `nullptr` instead.
- There is no *short version* of the type name before the closing semicolon.
- There must always be a closing sermicolon, after the closing brace `}`.

(1.16) Constants

Another useful concept is that of having a value that is *immutable* (i.e. cannot be changed). Even better is to ensure that any attempt to change such a value is flagged by the compiler. We can do this in C++ by using the `const` keyword. A common convention is to name the constant using all capital letters. For instance,

C++ Constants Convention

```
1 const double PI = 3.14159;
2 const double EULER = 2.71828;
```

Note that stating that a value is `const` means that it *must* be initialized when it is defined since it cannot be mutated later. Moreover, it is also possible to declare a more complex type such as a `struct` a constant. For example,

Constant `struct`

```
Node n1{ 5, nullptr };
const Node n2 = n1;
```

where `n1` is copied field-by-field into `n2`, which is declared to be constant. Any attempt to change the values in `n2`'s fields would cause a compilation error message to be produced.

Memory in C++

(1.17) Parameter Passing

Note that when we write

Pass by Value

```
int inc( int n ) { return ++n; }
...
int i = 10;
inc(i);
```

the value of `i` remains unchanged, since we are *passing* parameters to `inc` *by value*. In other words, a *copy* of `i` is passed in to `inc` when the program reaches line 4. In C, we get around this problem by using pointers. However, in C++, there is a type called *reference* that works similar to a pointer.

(1.18) References

(1.19) Make and Makefile

Recall that we can create *dependencies* between files (i.e. the relationship between files) as the size of our program grows, and those dependencies make compiling our program a bit more complicated. For instance, suppose that

- `vecs` depends on `main.o`, `vec.o`;
- `main.o` depends on `main.cc`;
- `vec.o` depends on `vec.cc`; and
- `main.cc`, `vec.cc` depend on `vec.h`.

A tool exists to automate the compilation, by combining the list of commands with the dependency relationship: `make`. Here is a simple `Makefile` for our project.

`Makefile`

```
vecs: main.o vec.o
    g++ main.o vec.o

main.o: main.cc vec.h
    g++ -std=c++14 -c main.cc

vec.o: vec.cc vec.h
    g++ -std=c++14 -c vec.cc
```

The code above must be placed in a file named `Makefile` (or `makefile`). In `Makefile`, the lines not indented are *dependencies*. They show that the file before the colon depends on the files after the colon. Each file is a *targe* (i.e. something `Makefile` describes how to create). Since we do not describe how to create `main.cc` or `vec.cc`, we include their dependencies (i.e. `vec.h`) along with `main.o`, `vec.o`. The targest for this `Makefile` are `vecs`, `main.o`, `vec.o`. Note that the commands to create a target must be indented with *tabs*, not spaces.

The command to create any given target is called a *recipe*. Thus the lines that start with `g++` are the recipes in this example. They are executed as shell commands by `make` to build each target. So we can include any valid shell command in the recipes.

To use this `Makefile`, we need only to use the `make` command.

Executing `Makefile`

```
~$ make
g++ -std=c++14 -c main.cc
g++ -std=c++14 -c vec.cc
g+= main.o vec.o -o vecs
```

`make` automatically discovers which commands it needs to run in order to create `vecs`, including creating each of its dependencies. And, it only rebuilds what needs to be rebuilt, based on what we actually change.

Executing `Makefile` upon Changes to the Files

```
  [... change main.cc ...]

~$ make
g++ -std=c++14 -c main.cc
g++ main.o vec.o -o vecs

  [... change vec.h ...]

~$ make
g++ -std=c++14 -c main.cc
g++ -std=c++14 -c vec.cc
g++ main.o vec.o -o vecs

  [... change nothing ...]

~$ make
make: 'vecs' is up to date.
```

If we do not ask specifically, then `make` creates whatever the first target is in the `Makefile`. In this case, `vecs`. However, we can also specifically ask for a given target.

Specifically Asking for a Target

```
~$ make main.0
make: 'main.0' is up to date

  [... change vec.h ...]
~$ make main.o
g++ -std=c++14 -c main.cc
```

(1.20) Phony Targets

Building things, particularly with separate compilation, leaves detritus in our directories, and it is nice to keep the commands to clearn up after ourselves in our `Makefile` as well. This is done through what is called a *phony target* (i.e. a target that exists only for its recipe, and does not actually build anything). The `clearn` phony target is commonly used for cleanup commands.

`clean` Phony Target

```
vecs: main.o vec.o
    g++ main.o vec.o

main.o: main.cc vec.h
    g++ -std=c++14 -c main.cc

vec.o: vec.cc vec.h
    g++ -std=c++14 -c vec.cc

.PHONY: clearn

clean:
    rm *.o vecs
```

We can use `clean` phony target by

Using `clean`

```
~$ make clearn
rm *.0 vecs
```

(1.21) Make Variables

Note that in `Makefile`, two fo the recipes are essentially identical. We can reduce the clutter, as well as making `Makefile` more flexible, by using `make` *variables*.

Make variables

```
CXX=g++
CXXFLAGS=-std=c++14
OBJECTS=main.o vec.o
EXEC=vecs

${EXEC}: ${OBJECTS}
    ${CXX} ${OBJECTS} -o ${EXEC}

main.o: main.cc vec.h
vec.o: vec.cc vec.h
.PHONY: clean

clean:
    rm ${OBJECTS} ${EXEC}
```

Also note that we are not explicitly writing a recipe for compiling `.cc` files into `.o` files. This is because `make` includes some *default recipes* using the pre-defined `make` variables `CXX`, `CXXFLAGS` (whose values we changed).

(1.22) Automatic Dependency Management

There is one last problem we can deal with: keeping the dependencies between `.cc` files and `.h` files up to date. it is easy to forget to update hte `Makefile` every time we add (or remove) a dependency from a `.cc` file. Luckily, `g++` is capable of generating this dependency information for us. The `-MMD` flag to `g++` causes it to create `.d` files, which are `make` dependencies, whenever it compiles a `.cc` file. We can combine this with the `Makefile` above to create a `Makefile` that keeps its own `.cc` and `.h` dependencies up-to-date.

Automatic Dependency Management

```
CXX=g++
CXXFLAGS=-std=c++14 -MMD
OBJECTS=main.o vec.o
DEPENDS=${OBJECTS:.o=.d}
EXEC=vecs

${EXEC}: ${OBJECTS}
    ${CXX} ${OBJECTS} -o ${EXEC}

-include ${DEPENDS}

.PHONY: clean

clean:
    rm ${OBJECTS} ${DEPENDS} ${EXEC}
```

2.
Object Oriented Programming

Introduction

(2.1) Coupling and Cohesion

Before we explain what classes are, let us provide motivations. There are two important software design quality measurements, *cohesion* and *coupling*.

Def'n 2.1. **Cohesion, Coupling** of a Software

Given a software,

(a) ***cohesion*** measures the amount of *relatedness* that a module[a] or unit of code contains; and

(b) ***coupling*** measures the amount of dependency between units and modules.

[a] A ***module*** is a library package, file, class , ...

It is a rule of thumb to maximize cohesion and minimize coupling.

(2.2) Procedural Programming Versus Object Oriented Programming

(a) *procedural programming*: The style of programming so far is called *procedural programming*. It has this name because the code is organized in *procedures* (which in C++ are the functions) that implement the logic of the program. Each variable can hold a single piece of data or be a structure that holds complex data with subparts (i.e. fields). The variables are passed to the procedures via pameters or are global variables.

(b) *object oriented programming* (*OOP*): In OOP, we implement our programs using objects. ***Objects*** are units of code that contain data and the procedures that implement the logic that operates over the data. Thus objects are self-contained units of data and code. In OOP, the data contained within an object are called *attributes* or *member fields*. The procedures of an object are called *methods*, *operations*, or *member fields*.

To create objects, we must first define ***classes:*** blueprints or type specifications that describe the contents of an object.

In general, a program that can be written in the procedural style could also be written in the object oriented programming style and vice versa. However, as a program grows in size, it is generally easier to maintain high cohesion and low coupling using object oriented programming.

(2.3) Classes in C++

We know that we can keep related pieces of information together in C and C++ by using *struct*. But in C++ we can do better: we can put functions inside of a structure, allowing us to form a *class* (there is the `class` keyword in C++, however, which we shall discuss shortly). Note that a class is a *type* and an *instance* of a class is called an *object*. Consider the following example.

student.h

```
#ifndef _STUDENT_H_
#define _STUdNET_H_

struct Student {
    int assns, mt, final;

    float grade();
};

#endif
```

Here we see that the `Student` class has three data fields used to hold marks respectively for assignments (`assns`), the midterm (`mt`), and the final exam (`final`). It has a single method, `grade`, that takes no parameters and returns the final grade in the course. In general, we want to try and avoid giving the client too much information about our our class and its functionality, which is one of the reasons we only give the function signature in the header `.h` file. Ideally, we would give client our header file and the `student.o` fil,e which they could then link in with their code, and they would never need to know how the final course grade was actually calculated.

student.cc

```
#include "student.h"

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

Note that in the implementation file for the `Student`, we have to use the *scope resolution operator* `::`, prefixed by the class name, to specify that the `grade` function is a *method* of the `Student` class. Since `grade` is part of `Student` and `Student` is a type, these fields do not actually exist until we create an instance of `Student`. So, to what fields is `grade` referring? It uses the fields of the object that invkes it using operator `.`, as in `s.grade()` in the client code. Here, `grade` is being invoked on the object `s`, so it uses the fields of `s` in its calculation.

main.cc

```
#include "student.h"
#include <iostream>

int main() {
    Student s{60,70,80};
    std::cout << s.grade() << std::endl;
}
```

The client includes our header file and creates an object `s` of the `Student` class type. It initializes `s` by passing in three integer values to the *initialization list* delimited by the curly braces `{}`. By default, the compiler maps the values to the data fields in the order they were declared. It then calls the `grade` method on the object `s` and outputs the calculated final grade.

More formally, all class methods have a hidden, extra first parameter called `this` that is a pointer of the class type. The parameter `this` is always set to contain the address of the objeect upon which the method is being invoked. When we access a class field within a class method, it automatically uses `this`. In other words, `student.cc` is equivalent to

student.cc Using this

```
#include "student.h"

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

It is however a good practice to use `this` only when absolutely necessary.

(2.4) Initializing Object

`main.cc` used an *initialization list* to copy the data values into the object's data field:

```
Student s{60,70,80};
```

This is rather limited in functionality, since it only copies the given data. We also have no way of guaranteeing that any `Student` object is actually initialized, and that all of the grades are in the range 0 to 100.

It would be a lot more useful if we had a method that could be used to guarantee that a `Student` object was always initialized upon creation, and that all of the grades were in the correct value range. It turns out that C++ classes have a special type of method, called ***constructor***, used exactly for this purpose. The general format of the constructor (often abbreviated to ctor) is

General Format of the Constructor

```
<name-of-class-type>(<parameter-list>) {
    <body>
}
```

(a) There is no return type for the constructors., since we are building the actual object. The constructor does not return the object it builds.

(b) The method name is the type name.

(c) The parameter list may be empty.

(d) Constructors can be overloaded, just like other functions.

(e) The method body may be empty, but must be present.

When we separated into header and implementation files, the `Student` example becomes as follows.

`student.h` (2)

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

    Student(int assns, int mt, int final);
    float grade();
};
...
```

`student.cc`

```
#include "student.h"

int capGrade(int grade) {
    if (grade<0) return 0;
    if (grade>100) return 100;
    return grade;
}

Student::Student(int assns, int mt, int final) {
    this->assns = capGrade(assns);
    this->mt = capGrade(mt);
    this->final = capGrade(final);
}

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

Note the usage of `this->`. This is necessay since our parameter names are the same as our data field names (by convention).

When we try to create a `Student` object without using the constructor

```
Student s;
```

the compiler complains since we do not have a constructor that takes zero argument.

We can also initialize an object in C++ using the assignment operator = as in

Object Initialization in C++ Using =

```
Student s = Student{60,70,80};
```

In some circumstances, `()` can be used to initilize variables, especially when invoking constructors. However, C++ has *uniform initialization syntax* using `{}`, so which is meant to be used in nearly all initialization situations.

Here are some advantages of creating constructor: creating one allows

(a) default parameters;

(b) overloading;

(c) sanity checks; and

(d) code other than simple field initializations.

Special Class Members

(2.5) Default Constructors

The term *default constructor* is a source of confusion to many, as there are several situations in which there may be a *default*. By definition, however, a default constructor is a constructor that does not have a parameter. If we do not provide a constructor of any sort to our class, then the compiler automatically gives a default constructor. All it will do is call the default construct on any data fields in the object that are themselves objects. Of course, this will only work if those objects also have a default constructor defined in their class type. That *default* default constructor is no longer available as soone as we declare (and define) any other constructor.

Default Constructor

```
struct Vec {
    int x,y;
    Vec(int x, int y) {
        this->x = x;
        this->y = y;
    }
};
Vec v1{5,6}; // this is fine
Vec v2;      // this will not compile
```

If we remove the constructor, then `v2` will become valid and `v1` invalid. To make both valid, we must use constructor overloading.

(2.6) Initializing Constants and References

If all objects of the class will have the same constant or reference, we can initialize the constant or reference in the type defintion. For example,

```
int z;
struct AStruct {
    const float PI = 3.14;
    int & aRef = z;
};

AStruct a1, a2;
```

Things become more complicated if we still want a constant or a reference, but not all objects will use the same constant or reference. For instant, a student's id number should be a constant, since it does not change; however, it should be unique for each student.

```
struct Student {
    const int id;
    ...
};

Student s1{1234,...}, s2{5678,...};
```

We obviously need to pass the constant student id in as a parameter to our constructor. Where do we initialize it? If we initialize it in the constructor body, that's too late since the data fields have already been created (and initialized). Constants need be initialized when their space is allocated. If the constructor body is too late, what other choice do we have? To resolve this, let us take look at the steps involved in creating an object:

(a) allocate space;

(b) construct the data files; and

(c) run the constructor body.

We need to be able to insert our values in the second step. It turns out that C++ has a special form of syntax called the ***member initialization list*** (***MIL***) that will let us do exactly this.

(2.7) Member Initialization Lists

The member initialization list code must be placed as a part o the constructor implementation. It occurs between the closing parenthesis of the parameter list, and the opening curly brace of the constructor body. It consists of a colon **:** followed by a comma-separated list of data field names to be initialized, where each data field is initialized using uniform initialization syntax.

student.h (3)

```
#ifndef _STUDENT_H_
#define _STUdNET_H_

struct Student {
    const int id;
    int assns, mt, final;

    Student(const int id, int assns=0, int mt=0, int final=0);
    float grade();
};

#endif
```

student.cc (2)

```
#include "student.h"

int capGrade(int grade) {
    if (grade<0) return 0;
    if (grade>100) return 100;
    return grade;
}

Student::Student(int assns, int mt, int final)
    : id{id},
      assns{capGrade(assns)},
      mt{capGrade(mt)},
      final{capGrade(final)}
{}

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

(a) The name outside of the `{}` in the MIL is the data field, while the name inside ofthe `{}` is the parameter name. We do not need `this->` here.

(b) The MIL can be used to initialize other data fields, not just constants or references.

(c) Fields in the MIL are initialized in the order in which they are declared in the class, not in the order that they are listed in the MIL.

(d) MIL can only be used on constructors, not on any other method.

It is quite common for constructors to have a MIL and an empty body, and it is usually considered poor practice to use the body to do anything that the MIL could have done. A rule of thumb is that if we need sophisticated logic, such as loops or conditions, to initialize the fields, use the body, but otherwise use the MIL.

(2.8) Copy Constructors

Recall that if we do not provide any construtors, the compiler gives us a default constructor. It turns out that there are several other special methods that the compiler gives us unless we provide our own (the ruls are actually a little more complicated than this, but we shall discuss them shortly). They consist of:

(a) *default constructor*: calls the default constructor on all fields that are objects;

(b) *copy constructor*: copies all fields from the object passed in;

(c) *copy assignment operator*: copies all fields from the object passed in;

(d) *destructor*: does nothing by default;

(e) *move constructor*: takes data from the object passed in; and

(f) *move assignment operator*: takes data from the object passed in.

A ***copy constructor*** is a constructor. So its signature must not return anything, and the name of the method must be the class name. Its only parameter will be a constant reference to an object of the same type. If we consider the ase of defining a copy constructor for the `Student` class, it may be sensible to copy the grades for the assignments, midterm and final exam of the student object passed in, but not the student identity field.

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

    ...
    Student(const Student &other)
        : assns{other.assns}, mt{other.mt}, final{other.final} {}
};
```

For something like a vector, we would actually want to copy every data field:

```
struct Vec {
    int x, y;

    Vec(int x=0, int y=0) : x{x}, y{y} {}
    Vec(const Vec & other)
        : x{other.x}, y{other.y} {}
};
```

which is equivalent in behaviour to the copy constructor that comes by default with every class.

Note that there are some cases where it is not desirable to use the default copy constructor (e.g. when working with linked lists). This is because the default copy constructor only makes a copy of the first node but *shallow copy* (i.e. copying the address) of the rest. If we want the copied list to be independent of the original one, we should explicitly make a copy constructor with a *deoop copy* (i.e. copying the contents). So for instance, if the linked list is built by

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

    Node(int data=0, Node * next=nullptr) : data{data}, next{next} {}
};
```

then

```
Node(const Node &n) {
    data = n.data;
    if (n.next!=nullptr) {
        next = new Node{*n.next};
    }
    else {
        next = nullptr;
    }
}
```

performs the deep copy. More compact version is

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

For now, we are going to say that a copy constructor is called when an object

- is initialized by another object;
- is passed by value; and
- is returned by value.

We shall see exceptions to all three of these cases when we discuss *move and copy ellisions*.

(2.9) Implicit Conversion

There is one type of constructor which we need to be careful – *single-argument constructors*. By definition, they are used to *implicitly convert* an argument of the specified type into an object of the constructor type. For instance, suppose we extand our `Node` class definition by saying

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

int f(Node n) {...}
```

It means that we can write code such as

Implicit Conversion

```
Node n1{4};  // single-argument constructor call
Node n2 = 4; // implicit conversion from int to Node via single-arg constructor
f(4);        // 4 is implicitly converted to a Node
```

This is fine so long as we are aware of what is happening and inted for it happen. The problem is that the compiler is sliently performing the conversion for us rather than warning us of a potential error. Thus it could happen unintentionally, causing our program to behave strangely. For this reason, we have the `explicit` keyword which disables implicit conversion. For instance, we could write

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

(2.10) Destructors

The compiler implicitly provides a ***destructor*** for us unless we write our own. Since the compiler knows nothing about our code or our intentions, the implicit destructor does nothing except for invoking destructors on object data fields that are themselves objects. It is invoked automatically whenever an object on the run time stack goes out of scope, or whenever the heap allocated memory for an object is freed.

The steps that occur when a destructor is run are:

(a) run the body of the destructor;

(b) invoke destructors for the object's data fields that are themselves objects (occuring in reverse declaration order); and

(c) deallocate the space associated with the object.

The signature for a class destructor follows the format

```
~<class name>();
```

It takes no parameters and returns nothing.

The motivation for using destructors is that they can help our program avoid memory leaks. For example, the singly-linked list of integer code that we have been using so far leaks memory. If we write a destructor for the `Node` class, we can prevent memory leaks so long as the initial pointer to the list is freed. Let us take a look at how we would implement this.

Destructor

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

This is possible since `delete nullptr` does nothing. Then given a code

```
Node * n = new Node{1, {new Node{2, new Node{3}}}};
...
delete n;
```

our initial call to `delete n` invokes the destructor of the first `Node` in the linked list. If the `next` data field contains `nullptr`, nothing happens; otherwise, we free the memory allocated to the `next` `Node` in the linked list. This repeats until we reach a `nullptr`.

(2.11) Copy Assignment Operator

Not only can we copy an object during the creation of another object, but we can also copy an object as part of an assignment operation. Let us look at a quick example using our previous `Student` class.

Copy Assienment Operator

```
Student dantzig{60, 80, 100};
Student schrijver{dantzig}; // copy ctor used
Student kuhn{75, 85, 95};
dantzig = kuhn; // copy, but not copy ctor; copy assignment operator
```

We are using the implicit, compiler provided default version of the ***copy assignment operator***. Just like the implicit copy constructor, it performs a shallow copy of the data fields. The name of the method is `operator=`, and by definition it takes a constant reference of the class type as its single parameter. This will be the object that was on the RHS of the assignment. It returns a reference of the class type, since it needs to be able to modify the object on the LHS of the assignment if it is part of a cascade or sequence of assignment operations. For example,

```
dantzig = schrijver = kuhn;
```

would copy the contents of the `kuhn` object into the `schrijver` object, then copy the contents of the `schrijver` object into the `dantzig` object.

Our general signature format is

General Signature Format of Copy Assignment Operators

```
<class type> & operator=(const <class type> & <parameter name>);
```

We demonstrate the usage of copy assignment operators with our `Node` class.

Copy Assignment Operator for `Node` (1)

```
1 Node & operator=(const Node & other) {
2     data = other.data;
3     next = other.next? new Node{*other.next} : nullptr;
4     return *this;
5 }
```

A problem with this version is that `this` is not a newly created object, so potentially we are overwriting the old address in `this->next`, causing a memory leak.

Copy Assignment Operator for `Node` (2)

```
1 Node & operator=(const Node & other) {
2     data = other.data;
3     delete next;
4     next = other.next? new Node{*other.next} : nullptr;
5     return *this;
6 }
```

Although we fixed memory leak by writing `delete next`, it is possible thatt we could end up with two pointers to the same `Node` when we try to perform the same assignment. For instance, suppose that `p1`, `p2` point to the same `Node` but we have

```
1 *p1 = *p2;
```

Then we have just freed the list that both `p1`, `p2` point to before we attempt the deep copy. This problem is called *self-assignment*, and we can fix things by introducing a check for it.

Copy Assignment Operator for `Node` (3)

```
1 Node & operator=(const Node & other) {
2     if (this==&other) return *this;
3     data = other.data;
4     delete next;
5     next = other.next? new Node{*other.next} : nullptr;
6     return *this;
7 }
```

We can check self-assignment by introducing `this&==other`. If that is the case, then there is nothing to do and we can simply return the original object.

One last problem that we have to keep in mind is the possibility of the heap running out of memory when we tried to perform a deep copy. Since we already deleted the contents of `this` on line 4, we face a problem.

Copy Assignment Operator for `Node` (4)

```
1 Node & operator=(const Node & other) {
2     if (this==&other) return *this;
3     data = other.data;
4     Node * tmp = other.next? new Node{*other.next} : nullptr;
5     // some exception handing code goes here
6     delete next;
7     next = tmp;
8     return *this;
9 }
```

One should not delete the address in `tmp`, since that will destroy the copy of the list.

(2.12) Copy and Swap Idiom

An alternative approach is to use the ***copy and swap idiom***. The approach relies upon the fact that a locally created object to the copy assignment operator will be automatically destroyed when the object goes out of scope upon the routine's termination. So we use the local object to make the deep copy, and if that was performed successfully, just swap the data fields of `this` and the local object. We will use the `std::swap` routine from the `utility` library to help write our own `swap` routine.

Copy and Swap Idiom

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

By the end of `tmp`'s initialization, it contains a deep copy of `other`. We then exchange the contents of `this` and `tmp` using `Node::swap`. This way, `tmp.next` ends up with the old value of `this->next`, which is freed when `tmp` goes out of scope and its destructor is run.

(2.13) Move Semantics

Before we cover our last two operations (move constructor and move assignment operator), we need to define what we mean by *move semantics*. In order to do so, we need a quick refresher on *lvalues* and *rvalues*. Intuitively speaking,

- if we can take the address of an expression, then it is an ***lvalue***; and
- otherwise, it is an ***rvalue***.

Let us illustrate it by examining a function `plusOne` that takes a `Node` by value, and returns a `Node` by value.

plusOne

```
#include <iostream>

struct Node {
    int data;
    Node *next;

    Node(int data, Node * next=nullptr); // constructor
    Node(const Node & other);            // copy constructor
    ~Node() {delete next;}               // destructor
}
```

Advanced Object Uses

(2.14)
Array of Objects

Creating an array of objects has some restrictions of which we need to be aware. In particular, it is a requirement of the language that the class must provide a default constructor; otherwise, it is a syntax error. For instance, if the class `Vec` is defined as

Class `Vec`

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

then trying to declare an array of `Vec` objects as either of the following statments

Array of `Vec` Objects

```
Vec arr_1[10];
Vec * arr2 = new Vec[10];
```

will fail since `Vec` does not have a default constructor. However, if we change the definition to

Class `Vec` (2)

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

then the previous array declarations would work.

Another approach that would work without introducing a default constructor is specifying the constructor call for each object in the array. For instance

```
Vec arr[3] = {Vec{0,0}, Vec{1,2}, Vec{3,4}};
```

works without a default constructor for `Vec`. However, this would not work if the size of the vector is determined dynamically. It also does not scale well (i.e. large-sized arrays). An (only) alternative approach is to create an array of pointers to the class type, and initialize each individually by dynamically requesting memory on the heap with the appropriate constructor call. Of course, to prevent memory leaks, each elements of the array must also be freed. Consider the following examples.

Array of Pointers (1)

```
int size; // initialize this later
...
Vec *arr[size];
for (int i=0; i<size; i++) {
    arr[i] = new Vec{i, i+1};
}
...
for (int i=0; i<size; i++) {
    delete arr[i];
}
```

Array of Pointers (2)

```
int size; // initialize this later
...
Vec **arr;
arr = new Vec*[size];
for (int i=0; i<size; i++) {
    arr[i] = new Vec{i, i+1};
}
...
for (int i=0; i<size; i++) {
    delete arr[i];
}
delete [] arr;
```

The second version is useful if the array is going to be very large, since otherwise we may run out of space.

(2.15) Constant Object

A ***constant object*** (or ***const object*** for short) is an object whose fields cannot be modified. To make an object constant, we use the keyword `const` or declare a parameter using the keyword `const`.

Constant Objects

```
Student s(60,70,80);        // regular (i.e. non-const) object
                            // fields can be modified
const Studnet s2 = s;       // the fields of s2 cannot be modified
                            // but s can still be modified
const Studnet s3(50,75,90); // s3 is also a const object
int f(const Node &n);       // n cannot be modified inside f
                            // even if n was not originally created as const
```

Although we can call methods on const objects, calling methods which tries to modify the fields of a const objects causes a compilation error. In other words, we can only call methods on const objects that do not modify fields. This is done by adding the keyword `const` to the end of the method's signature (after the parameter list). For instance

Constant Method

```
struct Student {
    int assns, mt, final;
    float grade () const; // doesn't modify fields, so declare it const
};

// const is part of a function's signature
// so it must be written in both .h and .cc
float Student::grade() const {
    return assns * 0.4 + mt * 0.2 + final * 0.4;
}
```

The compiler checks that const methods do not modify fields. Note that only const methods can be called on const objects.

(2.16) Mutable Fields

Mutable fields are the fields which can be updated even with const methods.

Mutable Fields

```
struct Student {
    ...
    mutable int numMethodCalls = 0; // can be mutated with const methods
    float grade() const {
        ++numMethodCalls;
        ...
    }
}
```

Mutable fields can be always changed, even if the object is const.

(2.17) Summary

Here is a summary of this section.

(a) Whenever we create a method that we know will not modify any internal fields of the object, declare the method as `const`. It will allow the method to be called on const objects.

(b) Whenever we create a method that receives objects as parameters and we know that the method will not modify those objects, mark the parameters as `const`. This will allow const objects to be passed as parameters.

(c) If we have fields that must be modified even for const objects, declare the field as `mutable`. This will allow it to be modified even inside const methods.

(d) Whenever we instantiate a new object and we know that the value of the fields should not be modified anymore after the initialization, declare it as a const object. This will prevent accidental modification in other parts of the code.

This page intentionally left blank.

3. Encapsulation and Design Pattern

Invariants

(3.1) Consider the following example.

```
struct Node {
    int dat;
    Node *next;
    Node (int data, Node *next): data{data}, next{next} {}
    ...
    ~Node() { delete next; }
};
int main() {
    Node n1 {1, new Node{2, nullptr}};
    Node n2 {3, nullptr};
    Node n3 {4, &n2};
}
```

What happens when these objects go out of scope? All three are stack-allocated, so all three have their destructors run. When `n1`'s destructor runs, it reclaims the rest of the list. But when `n3`'s destructor runs, it attempts to delete `n2`, which is on the stack, not the heap. This is undefined behavior. Quite possibly, the program will crash.

So the class `Node` relies on assumption for its proper operation: `next` is either `nullptr` or a valid pointer to the heap. This assumption is an example of an ***invariant***: a statement that must hold true (otherwise the program will not function correctly). It is hard to reason about programs if we do not rely on invariants.

Consider another exmaple. An invariant for a stack is that the last item pushed is the first item popped. But we cannot guarantee this invariant if the client can rearrange the underlying data within the stack.

Note in the example of the stack, violating the invariant may not result in a compiling or execution error. If the client rearranges the items in the stack, the stack will still return an item when we call `pop()` on it. However, the returned item may not be what the client was expecting. This can lead to *logical errors*. Logical errors are usually very difficult to debug as the program may not display any error, it may simply generate the wrong results.

Therefore, we need a way to enforce invariants and avoid logical errors in our programs. In OOP, we can do this with *encapsulation*. This is one of the benefits of using OOP.

Encapsulation

(3.2) Encapsulation

To enforce invariants, we introduce ***encapsulation***, which means that we make clients treat our objects as black boxes that create abstractions in which implementation details are sealed away, and such that clients can only manipulate them via provided methods. Encapsulations regains our control over our objects.

We implement encapsulation by setting the *access modifier* (or *visibility*) for each one of the members (fields or methods) of a class.

(a) ***Private*** class members can only be accessed from within the object (i.e. using the implicit `this` pointer of the object's methods). Therefore, any other objects (of a differnt type) or functions cannot directly access the private members of an object. This means that private fields cannot be read or modified from outside of the object's methods and that private methods cannot be called from outside of the object's methods.

(b) ***Public*** class members can be accessed from anywhere. Thus public fields can be read and modified from outside the object and public methods can be called from outside of the object.

```cpp
struct Vec {
    Vec(int x, int y); // by default, members are public
  private:
    int x,y;
  public:
    Vec operator+(const Vec &other);
    ...
};
```

Note, by default, all members of `struct`s are `public`.

So, in the example above, the constructor is public even though its access modifier was not specified. After we use one of the keywords `private` or `public`, all the declared members will use that access modifier until we write one of those keywords again or until the end of the class specification. So we could write `private:` just once and then list all the private members, then write `public:` once and list all the public members. And note that there is always a colon `:` after one of these keywords.

(3.3) `class`

By default, all members of `struct`s are publicly accessible. But in general, we want fields to be private so clients cannot directly modify the internal state of the objects and potentially violate invariants. In general, only methods should be public. Thus, it would be better if the default visibility were private. We can achieve this by switching from `struct` to `class`.

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

The only difference between `struct` and `class` in C++ is default visibility. Everything else will work the same way.

Iterator

(3.4) Iterator

Consider the following `List` class.

```cpp
class List {
    struct Node;
    Node *theList = nullptr;
  public:
    void addToFront(int n);
    int ith(int i);
    ~List();
}
```

But now we cannot traverse the list from the node to node, as we would with a linked list. Repeatedly calling `ith` to access all of the list items would cost $\Theta\left(n^2\right)$ time. But we cannot expose the nodes or we lost encapsulation.

Certain programming scenarios or problems arise often. We keep track of good solutions to these problems so that we can reuse and adapt them. This is the essence of a *design pattern*: if we have this situation, then *this* programming technique may solve it.

The solution is to use the ***iterator*** design pattern: create a class that manages access to nodes. This iterator class is an abstraction of a pointer, which lets us walk the list without exposing the actual pointers.

(3.5)
General Format

The general format of an iterator in C++ looks like this.

```cpp
class AIterator {
    explicit AIterator(...)
  public
    // access the item in the iterator is currently pointing to
    A &operator*();
    // advances the iterator to the next item and returns the iterator
    AIterator &operator++();
    // returns true if this and other are equal (i.e. points to the same item)
    bool operator==(const AIterator &other);
    // negation of the above
    bool operator!=(const AIterator &other);
};

class A {
    ...
  public:
    ...
    // returns an Iterator pointing to the first element
    AIterator begin();
    // returns an Iterator pointing past the last element (not the last element)
    AIterator end();
};
```

(EX 3.6)
Linked List with an Iterator

Consider the following example.

Linked List with an Iterator

```cpp
class List {
    struct Node;
    Node *theList;

  public:
    // Implementation of the Iterator design pattern
    class Iterator {
        Node *p;
        explicit Iterator(Node *p): p{p} {}  // Private constructor
       public:
        int &operator*() {
            // Access the current item
            return p->data;
        }
        Iterator &operator++() {
            // Advance the pointer and return it
            p = p->next;
            return *this;
        }
        bool operator==(const Iterator &other) const {
            // An iterator is equal to another if their pointers are equal
            return p == other.p;
        }
        bool operator!=(const Iterator &other) const {
            // The reverse of the equals operator
            return !(*this == other);
        }
        friend class List; // List has access to all members of Iterator
    };

    // The client calls this method to obtain an Interator
    // that points to the first element in the list
    Iterator begin() {
        return Iterator{theList};
    }

    // The client calls this method to obtain an Interator
    // that points to the position after the end of the list
    // i.e. a null pointer because a linked list ends when the next pointer is null
     Iterator end() {
        return Iterator{nullptr};
    }

    ... // other list operations (addToFront, ith, etc.)
};
```

Now the client can use the iteartor objects to walk the list in $\Theta(n)$ (i.e. linear time).

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

We can shorten the iteration by taking advantage of ***automatic type deduction***. A definition like `auto x=y;` defines `x` to *have the same type* as `y`. The main advantage of a definition like this is that we do not need to write down the exact type of `x`, which may be complex.

```cpp
for (auto it=lst.begin; it!=lst.end(); ++it) {
    std::cout<<*it<<std::endl;
}
```

(3.7) Range-based for Loops

We can shorten the iteration loop even further by taking advantage of C++'s built-in support for the iterator pattern: the ***range-based*** for loop.

```cpp
for (auto n:lst) {
    std::cout<<n<<endl;
}
```

Range-based for loops are available for any class `C` with the following properties.

(a) `C` has methods `begin` and `end` that produce iterators; and

(b) the iterator supports prefix `operator++`, `operator!=`, and unary `operator*`.

Therefore, a minimal implementation of the iterator design pattern requires at least these five opeartions (`begin` and `end` in the base class, as well ass `operator++`, `operator!=`, and `operator*` in the iterator class). If we omit any of these opeartions, the implementation is incomplete and the iterator will not work properly.

The range-based for loop depicted above declares the variable `n` by value, as a copy of successive items of the list. Hence changes to `n` do not change the actual items in the list. If we wish to mutate the elements of the list (or save the copying, for lists of larger objects), we can declare `n` by reference.

```
for (auto &n:lst) {
    ++n; // mutates the actual list item
}
```

Decorator

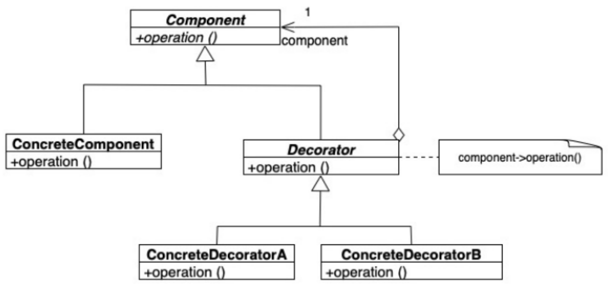
(3.8) Decorator

The ***decorator*** design pattern is intended to let us add functionality or features to an object at *run-time* rather than to the class as a whole. These functionalities or features might also be withdrawn. This is useful when extension by subclassing is impractical. We can also add features incrementally by adding new subclasses. It does mean, however, that we have end up having lots of little objects that look rather similar to each other, and differ only in how they are connected.

(3.9) General Class Model

The general form of the decorator design pattern's UML class model diagram looks is shown in Figure 3.1.

Figure 3.1



Here are some notes.

(a) The `Decorator` abstract base class contains a pointer to a `Component`. This lets us build a linked list of decoration objects by having a pointer to a `Component`. Since all of the decoration classes inherit from `Component`, we do not need to know what is in the linked list once it is built since the `operation` we are invoking is vifrtual (likely pure virtual) in the `Component` class.

(b) Since the `ConcreteComponent` inherits from `Component`, the linked list thus terminates with a concrete component object.

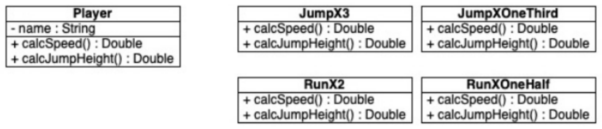
(c) Each call to the `operation` in decoration object ends up invoking the `operation` in the next component. In this way, we can build up values or actions by *delegating* the work to the objects of which the decorated object is composed.

(d) New functionalities are introduced by adding new subclasses, not modifying existing code, which reduces the chance of introducing errors.

(EX 3.10)
Video Game

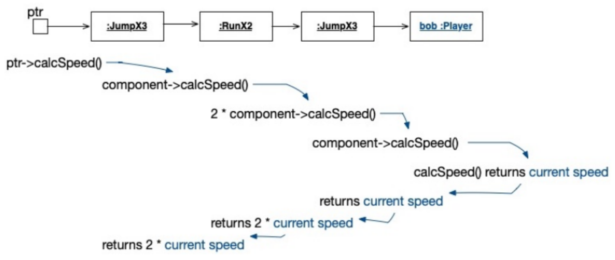
Let us walk through an example to try and explain why the decorator design pattern is structured the way it is. Consider the situation where we are developing a video game. As a character progresses through the game, their ability to run or jump can be either positively or negatively affected by their actions such as consuming certain fruit. The effects would intuitively be cumulative, and could potentially expire over time. Since we do not want to write a class for every possible combination of effects, our initial general class structure would look some thing like Figure 3.2.

Figure 3.2



And depending upon the actual game play, we would track the current set of effects by creating a linked list of the objects currently affecting the player's speed or jump height. For example, here is a picture to illustrate a situation where the current player `bob` has had their speed doubled once, and their abiliity to jump 3 times as high increased twice.

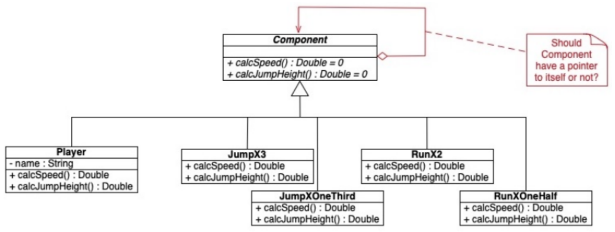
Figure 3.3



In order to calculate `bob`'s current speed, we would traverse the list of effects, asking for the current speed through the `calcSpeed` method. The `JumpX3::calcSpeed` method just returns the speed of the next node in the list, without changes. The `RunX2::calcSpeed` multiplies the speed returned by the next node by 2. So a call to `ptr->calcSpeed()` ends up returning `bob`'s base speed multiplied by 2. If we would wanted to know how high `bob` could currently jump, `ptr->calcJumpHeight()` would have ended up returning `bob`'s current height-jumping ability multiplied by 3, twice (i.e. multiplied by 9 in total).

We would like to perform all of these calculations as trasparently as possible, without having to know what each element of the linked list actually points to, object-wise. This means that we would have to introduce an abstract base class called `Component` with pure virtual methods, from which `Player`, `RunX2`, and `JumpX3` would all inherit. But in order to be able to build a linked-list, all of our effects need to have a poitner to the abstract base class type since they do not know what sort object they will actually point to at run-time.

Figure 3.4


Component
+ calcSpeed() : Double = 0
+ calcJumpHeight() : Double = 0
Should Component have a pointer to itself or not?
Player
- name : String
+ calcSpeed() : Double
+ calcJumpHeight() : Double
JumpX3
+ calcSpeed() : Double
+ calcJumpHeight() : Double
RunX2
+ calcSpeed() : Double
+ calcJumpHeight() : Double
JumpXOneThird
+ calcSpeed() : Double
+ calcJumpHeight() : Double
RunXOneHalf
+ calcSpeed() : Double
+ calcJumpHeight() : Double

Since all of our effects need the pointer, it would be nice to have it in the abstract base class `Component`. However, we do not want our concrete player class to have it, since it is the final node in the list. So we really would prefer to introduce an intermediate abstract base class `Decorator` that

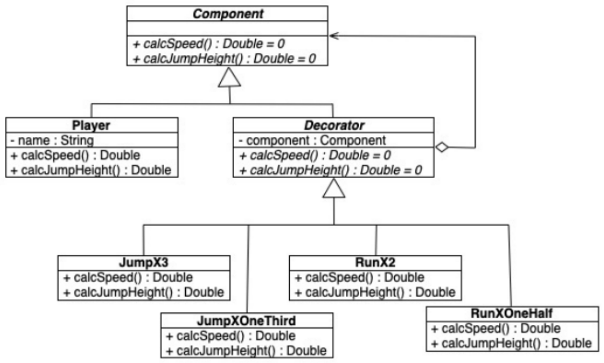
(a) inherits from `Component`;

(b) holds the pointer; and

(c) still has pure virtual methods for `calcSpeed` and `calcJumpHeight`.

`Player` would then inherit from `Component`, while the effect classes inherit from `Decorator`. This lets us restructure our class model to what is shown in Figure 3.5.

Figure 3.5

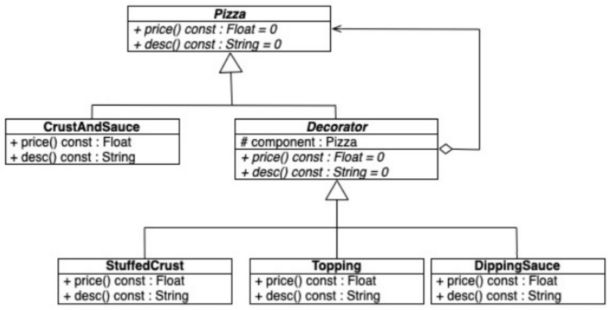

Component
+ calcSpeed() : Double = 0
+ calcJumpHeight() : Double = 0
Player
- name : String
+ calcSpeed() : Double
+ calcJumpHeight() : Double
Decorator
- component : Component
+ calcSpeed() : Double = 0
+ calcJumpHeight() : Double = 0
JumpX3
+ calcSpeed() : Double
+ calcJumpHeight() : Double
RunX2
+ calcSpeed() : Double
+ calcJumpHeight() : Double
JumpXOneThird
+ calcSpeed() : Double
+ calcJumpHeight() : Double
RunXOneHalf
+ calcSpeed() : Double
+ calcJumpHeight() : Double

Note that this version looks structurally like the standard decorator design pattern.

(EX 3.11)
Code Example

Our motivating example is going to use the creation of a pizza with a custom set of toppings, where the cost of the pizza itself depends upon the items ordered. The most baisc pizza we can have is one with a crust and some sauce costing $5.99, so that will be our concrete base class that we will decorate. Our toppings are $0.75 each and are initialized with a string that describes what the topping type is. Each stuffed crust adds $2.69 to the price while each dipping sauce adds $0.30. Our class model thus looks like Figure 3.6.

Figure 3.6



Our abstract base classes consists of the following files.

pizza.h

```
#ifndef _PIZZA_H_
#define _PIZZA_H_
#include <string>

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

#endif
```

pizza.cc

```
#include "pizza.h"

Pizza::~Pizza() {}
```

decorator.h

```
#ifndef _DECORATOR_H_
#define _DECORATOR_H_
#include "pizza.h"

class Decorator: public Pizza {
 protected:
  Pizza *component;
 public:
  Decorator( Pizza *component );
  virtual ~Decorator();
};

#endif
```

decorator.cc

```
#include "pizza.h"
#include "decorator.h"

Decorator::Decorator( Pizza *component )
  : component{component} {}

Decorator::~Decorator() { delete component; }
```

Despite the relationship between `Pizza` and `Decorator` being shown as aggregation, in this context composition makes more sense as the relationship, since pizza toppings are not shared, and when a pizza is eaten (destroyed) so are its toppings.

We will only show two of our concrete classes, `CrustAndSauce` and `Topping`.

crustandsauce.h

```
#ifndef _CRUSTANDSAUCE_H_
#define _CRUSTANDSAUCE_H_
#include "pizza.h"

class CrustAndSauce: public Pizza {
 public:
  float price() const override;
  std::string description() const override;
};

#endif
```

crustandsauce.cc

```
#include "crustandsauce.h"

float CrustAndSauce::price() const { return 5.99; }

```

```
std::string CrustAndSauce::description() const { return "Pizza"; }
```

```
topping.h

#ifndef _TOPPING_H_
#define _TOPPING_H_
#include "decorator.h"
#include <string>
class Pizza;

class Topping: public Decorator {
  std::string theTopping;
  const float thePrice;
 public:
  Topping( std::string topping, Pizza *component );
  float price() const override;
  std::string description() const override;
};

#endif
```

```
topping.cc

#include "topping.h"
#include "pizza.h"

Topping::Topping( std::string topping, Pizza *component )
  : Decorator{component}, theTopping{topping}, thePrice{0.75} {}

float Topping::price() const {
   return component->price() + thePrice;
}

std::string Topping::description() const {
  return component->description() + " with " + theTopping;
}
```

Our main program consists of the following.

```
main.cc

#include <iostream>
#include <string>
#include <iomanip>
#include "pizza.h"
#include "topping.h"
#include "stuffedcrust.h"
#include "dippingsauce.h"
#include "crustandsauce.h"


int main() {
   Pizza *myPizzaOrder[3];

   myPizzaOrder[0] = new Topping{"pepperoni",
      new Topping{"cheese", new CrustAndSauce}};
   myPizzaOrder[1] = new StuffedCrust{
      new Topping{"cheese",
         new Topping{"mushrooms",
            new CrustAndSauce}}};
   myPizzaOrder[2] = new DippingSauce{"garlic",
      new Topping{"cheese",
         new Topping{"cheese",
            new Topping{"cheese",
               new Topping{"cheese",
                   new CrustAndSauce}}}}};

   float total = 0.0;

   for (int i = 0; i < 3; ++i) {
       std::cout << myPizzaOrder[i]->description()
            << ": $" << std::fixed << std::showpoint << std::setprecision(2)
            << myPizzaOrder[i]->price() << std::endl;
       total += myPizzaOrder[i]->price();
   }
   std::cout << std::endl << "Total cost: $" << std::fixed << std::showpoint
      << std::setprecision(2) << total << std::endl;

   for ( int i = 0; i < 3; ++i ) delete myPizzaOrder[i];
}
```

Observer

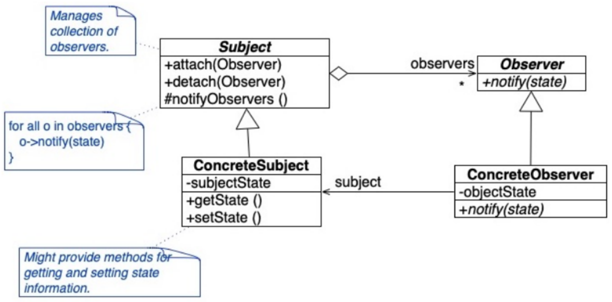
(3.12) Observer

The ***observer*** design pattern is indended to create a one-to-many dependency between objects such that all *observers* are notified when the state of the *subject* object being observed changes.

(3.13) General UML Class Model

The general form of the observer design pattern's UML class model diagram looks as follows. Here are

Figure 3.7



some notes.

(a) `Subject` and `Observer` classes are abstract base classes.

(b) `Subject` contains the code common to all subclasses that they inherit. There is no reason to make the `attach` and `detach` methods `virtual`, since the code will be the same for all subclasses.

(c) `Subject::notifyObservers` is usually declared protected, since it should only be called by the concrete subject objects. In certain cases, it may be made public so that an external source can trigger the notifications.

(d) The `Subject` has an aggregate relationship to the `Observer` class. Therefore, it is not responsible for destroying them when its destructor is run.

(e) Concrete observers may dynamically attach or detach themselves from a concrete subject during the program run-time.

(f) Every time the concrete subject changes state, all of the concrete observers are notified by calling `Subject::notifyObservers`, which calls their virtual `notify` mehtod (`Observer::notify` is pure virtual).

(g) The subject and observer classes are *loosely coupled* since they interact and only need to know that the classes follow the specified interface (i.e. they do not need to know details of the concrete classes other than what is specified by the subject and observer public methods). New observer types only need to inherit from the `Observer` class. Same for new subject types.

The usual sequence of calls for the pattern consists of the following.

(a) Concrete subject's stat changes.

(b) Concrete subject calls `notifyObservers`, which calls the `notify` method for each concrete observer currently subscribed or attached. If we are using the *push* model for data exchange, then all of the information an observer might need will be passed by the `notify` method; otherwise, `notify` passes no data.

(c) If we are using the *pull* model for data exchange, then each concrete observer calls the concrete subject's `getState` method to query the state of the subject; otherwise, it does nothing.

(EX 3.14)
Code Example

Our motivating example is going to use a set of horse races for context. The idea is that people who are betting on the races want to be notified of the outcome of each race upon which they have bet so that they know whether or not to collect their winnings. Thus we will map our concrete observers to the bettors and the concrete subjects to the races. Since one of the key ideas behind the design patterns is code encapsulation and reusability, we will define our abstract base classes as `Subject` and `Observer` to make the code more easily reusable later.

subject.h

```
#ifndef _SUBJECT_H_
#define _SUBJECT_H_
#include <vector>
#include "observer.h"

class Subject {
   std::vector<Observer*> observers;
 public:
   Subject();
   void attach( Observer *o );
   void detach( Observer *o );
   void notifyObservers();
   virtual ~Subject() = 0;
};

#endif
```

subject.cc

```
#include "subject.h"

Subject::Subject() {}
Subject::~Subject() {}
```

```

void Subject::attach( Observer *o ) { observers.emplace_back(o); }

void Subject::detach( Observer *o ) {
  for ( auto it = observers.begin(); it != observers.end(); ++it ) {
    if ( *it == o ) {
      observers.erase(it);
      break;
    }
  }
}

void Subject::notifyObservers() { for (auto ob : observers) ob->notify(); }
```

observer.h

```
#ifndef _OBSERVER_H_
#define _OBSERVER_H_

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

#endif
```

observer.cc

```
#include "observer.h"

Observer::~Observer() {}
```

Our concrete classes then become the following.

bettor.h

```
#ifndef __BETTOR_H__
#define __BETTOR_H__
#include "observer.h"
#include "horserace.h"

class Bettor: public Observer {
  HorseRace *subject;
  const std::string name;
  const std::string myHorse;
 public:
  Bettor( HorseRace *hr, std::string name, std::string horse );
  void notify() override;
  ~Bettor();
};

#endif
```

bettor.cc

```
#include <iostream>
#include "bettor.h"

Bettor::Bettor( HorseRace *hr, std::string name, std::string horse )
    : subject{hr}, name{name}, myHorse{horse}
{ subject->attach( this ); }

Bettor::~Bettor() { subject->detach( this ); }

void Bettor::notify() {
  std::cout << name
    << (subject->getState() == myHorse ? " wins! Off to collect." : " loses.")
    << std::endl;
}
```

horserace.h

```
#ifndef __HORSERACE_H__
#define __HORSERACE_H__
#include <fstream>
#include <string>
#include "subject.h"

class HorseRace: public Subject {
  std::fstream in;
  std::string lastWinner;
 public:
  HorseRace( std::string source );
  ~HorseRace();

  bool runRace(); // Returns true if a race was successfully run.

  std::string getState();
};

#endif
```

horserace.cc

```
#include <iostream>
#include "horserace.h"

HorseRace::HorseRace(std::string source): in{source} {}
HorseRace::~HorseRace() {}

bool HorseRace::runRace() {
  bool result {in >> lastWinner};
  if (result) {
    std::cout << "Winner: " << lastWinner << std::endl;
  }
  return result;
}
std::string HorseRace::getState() {
  return lastWinner;
}
```

Our main program consists of:

main.cc

```
#include <iostream>
#include "bettor.h"

int main(int argc, char **argv) {
  std::string raceData = "race.txt";
  if ( argc > 1 ) raceData = argv[1];

  HorseRace hr{raceData};

  Bettor Larry{ &hr, "Larry", "RunsLikeACow" };
  Bettor Moe{ &hr, "Moe", "Molasses" };
  Bettor Curly{ &hr, "Curly", "TurtlePower" };

  int count = 0;
  Bettor *Shemp = nullptr;

  while( hr.runRace() ) {
    if ( count == 2 )
      Shemp = new Bettor{ &hr, "Shemp", "GreasedLightning" };
    if ( count == 5 ) delete Shemp;
    hr.notifyObservers();
    ++count;
  }
  if (count < 5) delete Shemp;
}
```

4. Relationships and Inheritance

Unified Modeling Language

(4.1) Unified Modeling Language

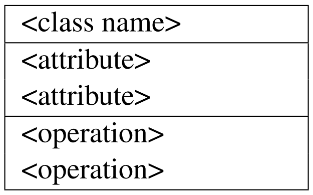
The ***unified modeling language*** (or ***UML*** for short) is a visual modeling language that lets people design and document a software system.

Def'n 4.1. **Class Model**
A ***class model*** is a diagram that visually represents a group of classes, and the relationships between them.

(4.2) Class Model Notation

We start by describing the notation used to specify the structure of a class. Figure 4.1 shows the basic structure for a class.

Figure 4.1



The first box contains the *class name*.

(a) The class name must be unique among all of the classes in that particular scope. Otherwise, it is qualified with a scope operator (e.g. `<package-name>::<class-name>`).

(b) The class name must be capitalized, centered or left-justified, and in bold. Note that, in general, class names are not pluralized unless the class is intedned to be a contained for multiple objects of the type.

(c) The class name may be qualified by an optional stereotype keyword, centered and in the regular tyface, placed above the class name, and within guillemets <<>>[1] and an optional stereotype icon in the upper-right corner.

The second box is optioal, and contains the class *attributes*. The third box is also optional, and contains the class *operations*.

(4.3) Comments

A *comment* can be added to the class diagram by joining a rectangle with a bent upper-right corner (called *dog-ear*) to the item being annotated by a dashed line. The comment is a text string but has no effect on the model, though it may describe a constraint on the annotated item.

(4.4) Attributes

Attributes are left-justified and written in the regular typeface. They are generally shown when needed, though a full description must be provided at least once. They describe the information held by an instance of the class, and may be replaced by *association ends*. The general syntax for an attribute is

```
<<stereotype>> <visibility>/<name>:<type> <multiplicity> <initial-value> {<property>}
```

where every argument except the name is optional. The name generally starts with a lower case letter.

[1]Guillemets are special unicode single characters, 226A, 226B.

`visibility`	Describes the visibility of the attribute as a punctuation mark, though can be instead described in the property string. There are four markers: + *public*, - *private*, # *protected*, and ~ *package*.
`type`	A string that describes the attribute's type. Usually written to be as language-independent as possible (e.g. boolean rather than the C++ `bool`). Since we are discussing about C++ only, we are going to adapt C++ type names.
`/`	Indicates that this attribute is derived from a parent class.
`multiplicity`	Describes the multiplicity constraints on the attribute (i.e. how many of these values will be held). It is enclosed in square brackets `[]`. It is usually omitted if the value is exactly 1 since that is the commonest case. It will consist of either a string specifying the exact number (e.g. `[3]`), a string specifying a precise range (e.g. `[4..10]`), an unknown number (e.g. `[*]`), or a range with a specified lower bound but no known upper bound (e.g. `[2..*]`). Note that `[0..1]` implicitly represents a pointer in C/C++.
`initial-value`	Specifies the default initial value as an qual sign followed by the value (e.g. `= 0`).
`property`	List of comma-separated strings surrounded by `{}`. Defaults to `{changeable}`, but can be specified as `{readOnly}` to indicate that it is a constant. Can be used to annotate multiplicity, such as `ordered` (elements in collection follow an order), `unordered` (items in collection are not in order), `bag` (collection of elements that may have multiple copies of elements), `seq` (elements of the set for an ordered sequence), `set` (collection of unique elements), and `list` (ordered variable length collection).

If the attribute is `static`, the name and type strings are underlined. Here are some examples of attribute declarations.

Examples of Attribute Declarations

```
– colors : Saturation [3]
# points : Point [2..*] {ordered,set}
size : Area = (100, 100)
+ name : String [0..1]
+ name : String {readOnly}
```

(4.5) Operations

Operations are left-justified and written in the regular typeface. They are generally shown when needed, though a full description must be provided at least once. Since all class will have constructors, destructors, accessors (getters), and mutators (setters), we will not generally specify them in our UML class diagram but assume their presence. The general syntax for an operation is:

```
<<stereotype>> <visibility> <name> (<parameter–list>):
<return–type> <multiplicity> <initial–value> <property>
```

where every argument except the name and parameter listt is optional. The `name` generally starts with a lowercase letter.

`visibility`	Same as for the attribute.
`parameter-list`	List of parameters enclosed in parentheses. Each parameter is of the form `<direction> <name> : <type> <multiplicity> = <default-value>`. `direction` specifies direction of information flow and is optional. If the `default-value` is omitted, so is the preceeding equal sign.
`property`	List of strings surrounded by `{}`. A list of raised exceptions may be shown by specifying a comma-separated list of them after the keyword `exception`. Instead of italicizing an abstract operation, the keyword `abstract` may be placed in the property string. Similarly, instead of underlining a static operation, the keyword `static` may be placed in the property string. The property `isQuery=false` indicates that this operation does not change the state of the object. It defaults to a value of `false`. A value of `true` deos not guarantee that the state changes, only that it may change. The property `isPolymorphic=true` indicates that the operation is overridable and is the default value.

An *abstract operation* is italicized. If the operation is `static`, the name and type strings are underlined. Constraints on an operation may be indicated by attaching a note symbol to the specific operation by a dashed line, where the note specifies the constraint as a string in `{}`. The constraint may also be annotated by one of `<<precondition>>`, `<postcondition>>`, or `<<bodycondition>>`. Here are some examples of operation declarations.

Examples of Operation Declarations

```
– display () : Location
+ hide ()
<<constructor>> + create ()
– attachXWindow( xwin : Xwindow* )
# Matrix::transform (in distance: Vector, in angle: Real = 0) : Matrix
```

(4.6) Dependencies

If a class A has a *dependency* upon another class B, it does not actually contain an instance of B but it receives an instance of B as a parameter, or returns an instance of B, or has some sort of other temporary relationship. We draw a dashed line betwen A, B.

(4.7) Static

As mentioned previously, *static* attributes or operations are underlined in the UML class model; otherwise we miht need to tag them with the `{abstract}` property if our drawing software does not easily let you underline text.

(4.8) Class Relationships

There are four main types of class relationships: association, aggregation, composition, and generalization.

(4.9) Associations

The simplest form of a relationship between two classes is that of an *association*. It is drawn as a solid line between two classes in the UML diagram. There may be more than one association between the two classes, and a class may be associated with itself. Association lines may cross each other, but an opertional *line hop* may be used to help avoid ambiguity. The association itself may have a name, but that is optional. The name should not be placed too close to either end in order to avoid confusing it with the association end name if one is present. The name may be decorated with a solid traignle to help indicate in which direction the naming relationship is to be read. Each end of association may have

(a) a name, which is equivalent to declaring an attribute in the class;

(b) a navigation arrow to indicate that it is the holder of the information;

(c) an associated multiplicity; and

(d) constraints, such as order.

(4.10) Aggregations

Aggregation is a form of association that desecribes a *whole-part* relationship where one class, the *aggregate* or *whole*, is made up of the constituent part class. The end of the association with the aggregate is makred with a special symbol, an open/hollow/white diamon. Only one end of the association may be marked as an aggregate. This is often described as a *has-A relationship*, where if B has A, typically

(a) A exists independently outside of B;

(b) if B is destroyed, A lives on; and

(c) if B is copied, then A is not deep copied (i.e. perform a shallow copy and A is shared).

This is usually implemented by pointers or reference fields.

(4.11) Compositions

Composition is a stricter form of aggregation. In particular, once a *part* is joined to a *whole* it may not be shared with any other object. The whole is also responsible for destroying all of its component parts when it is destroyed. It may also be responsible for creating its components. Whether the components exist independently beforehand or are created by the owner is something to be specified in the design. This is often described as an *owns-A relationship*, where if B owns A, typically

(a) A has no identity or independent existence outside of B;

(b) if B is destroyed, A is destroyed; and

(c) if B is copied, then A is deep copied.

Since the owner has an (implicit) multiplicity of 1, we usually do not specify it. Note that the *owner* end of the association is maked with a solid, black diamond. This is generally implemented by a composition of classes.

(4.12) Generalizations

The *generalization* (or *specialization*) association between the parent (or superclass) and the child (or subclass) is indicated by putting a triangular arrowhead on the association end that joins the parent. By definition, there is no multiplicity or navigation arrowhead, though constraints may be added. There may be separate lines from the parent class to each child, or the lines may be drawn as a tree structure. This is often described as an *is-A relationship*, and usually implemented through inheritance.

Inheritance and Polymorphism

(4.13) Inheritance

Let us examine the idea of *inheritance* more closely by looking at an example to motivate our code structure. Here is the UML class model.

Figure 4.2

Book
- title : String
- author : String
- length : Integer
+ Book(title : String, author : String, length : Integer)

Text
- title : String
- author : String
- length : Integer
- topic : String
+ Text(title : String, author : String, length : Integer, topic : String)

Comic
- title : String
- author : String
- length : Integer
- hero : String
+ Comic(title : String, author : String, length : Integer, hero : String)

Each `Book`, `Text`, and `Comic` class have fields to hold the `author`, `title`, and `length`. The text has an additional field, the `topic`, while the `Comic` has an additional `hero` field. The respective class header files look like the following, assuming the appropirate header guards and inclusion of the `<string>` library.

`Book`, `Text`, `Comic` Classes

```cpp
class Book {
    std::string author, title;
    int length;
  public:
    Book( const std::string &author, const std::string &title, int length );
    std::string getTitle() const;
    std::string getAuthor() const;
    int getLength() const;
    bool isHeavy() const;
};
class Text {
    std::string author, title, topic;
    int length;
  public:
    Text( const std::string &author, const std::string &title,
    int length, const std::string &topic );
    std::string getAuthor() const;
    int getLength() const;
    bool isHeavy() const;
    std::string getTopic() const;
};
class Comic {
    std::string author, title, hero;
    int length;
  public:
    Comic( const std::string &author, const std::string &title,
    int length, const std::string &hero );
    std::string getAuthor() const;
    int getLength() const;
    bool isHeavy() const;
    std::string getHero() const;
};
```

Note that the accessor methods are declared constant so that the compiler will notify us if those methods try to change the contents of the data fields. The problem becomes *how do we store our actual books, texts, and comics*? Ideally, we would like to keep them all in a single array (or other collection type) so that we could iterate or traverse the entire collection in one loop, without having to worry about the underlyign types. In order to accomplish this, we use ***inheritance***.

The idea is to recognize `Text` is a `Book`, of sorts, as is the `Comic`. In other words, they are books with other, additional features. Alternatively, we could describe a `Text` as a *sepcialization* of a `Book`, or a `Book` as a *generalization* of a `Comic`. Book then becomes our ***superclass*** (or ***base class***, ***parent class***). `Text` and `Comic` become our ***subclasses*** (or ***derived classes***, ***child classes***).

Inheritance lets us use the information and methods defined in the class from which we inherit, which lets us remove duplicate code and data. This is useful feature, since it reduces the number of places we have to change if we change implementations. Note that it is also possible to *override* the parent's method to replace it with the subclass's own version, which is often desirable. We will see more on that topic shortly. The respective class header files are then changed to look like the following.

`Text`, `Comic` Redefined

```cpp
class Text : public Book {
    std::string topic;
  public:
    Text( const std::string &author, const std::string &title,
          int length, const std::string &topic );
    std::string getTopic() const;
};
class Comic : public Book {
    std::string hero;
  public:
    Comic( const std::string &author, const std::string &title,
           int length, const std::string &hero );
    std::string getHero() const;
};
```

The keywords `: public Book` after the class name (`Text`, `Comimc`) tell us that `Text`, `Comic` *inherit* publicly from `Book`. Although we can also inherit using the keywords `protected` and `private`, we will not cover this topic.

The next problem to overcome is how to initialize a `Text` or `Comic` object. Our temptation is to just use the MIL as normal:

```cpp
Text::Text(const std::string &title, const std::string &author,
           int length, const std::string &topic)
    : title{title}, author{author}, length{length}, topic{topic} {}
```

This would not compile for two reasons.

(a) The data fields `author`, `title`, and `length` are `private` to the `Book` class by default and thus not directly accessible to a `Text` or `Comic` object.

(b) We are not invoking the `Book` constructor to initialize its fields, and it does not have a default constructor.

It turns out that when we create an object, our sequence of steps has changed a bit due to inheritance. Our steps for object creation now consists of the following.

(a) Allocate space for the object.

(b) Invoke the *superclass constructor* to build the superclass portion of the object.

(c) Construct the fields.

(d) Run the constructor body.

Running the `Book` constructor in the `Text` constructor body is too late (just as it was for constants or references) sincethe object has already been constructed by that point. So, just like before, we resort to using the MIL by replacing the initializations of the data fields `author`, `title`, and `length` in the MIL with a call to the `Book` constructor instead.

```cpp
Text::Text(const std::string &title, const std::string &author,
           int length, const std::string &topic)
     : Book{title, author, length}, topic{topic} {}
```

If the superclass has no default constructor, the subclass must invoke the superclass constructor in its MIL.

Another approach is to use `protected` fields or methods.

protected

```cpp
class Book {
    std::string title, author;
    int length;
  protected:
    int getLength() const;
  public:
    Book(const std::string &title, const std::string &author, int length);
    std::string getTitle() const;
    std::string getAuthor() const;
    bool isHeavy() const;
};
```

In this version, we have made the `getLength` method `protected` to make it callable from the subclasses, while remaining uncallable for any other classes. We generally do not use data fields `protected`. Instead, we provide `protected` methods to control how the subclasses can manipulate the `private` information.

(4.14) Object Slicing

When we try to initialize a superclass object by copying a subclass object, C++ performs ***object slicing***. That is, we only copy the information in the *superclass core* inside the subclass object. This can sometimes cause problems. For instance, if we write

```cpp
Comic c(...)
Book b = c;
```

then C++ would recognize `b` as a `Book` object, but not `Comic`. So if we have a method whose implementation is different in the `Book` and `Comic`, then calling the method on `b` would use `Book` version of the method, not `Comic`. To get around this, we use pointers instead (which do not slice the data indeed), and *virtual* methods.

The C++ mechanism that lets us force the program to dynamically examine the actual type of the pointed to object rather than rely upon the declared type of the pointer consists of declaring a method to be `virtual` in the superclass. By default, the keyword `virtual` is repeated on the method signature in the subclass even when omitted. When we have a `virtual` method in the parent class whose implementation is changed or replaced in the child class, we say that the method is ***overridden***. Not that the method signatures must match exactly (e.g. if the method is const in the superclass, then it must be const in the subclass). We can let the compiler help us with catching these sorts of errors by appending the keyword `override` to the signatures of the overridden methods (but not to the signature of the method in the parent class).

`virtual`, `override`

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

It would be perfectly fine to have instead declared `isHeavy` in the subclasses as

```
virtual bool isHeavy() const override;
```

Note that we leave keywords `virtual` and `override` off of the implementations.

(4.15) Polymorphism

The last version takes the previous idea, that using pointers lets us hold objects of either the parent or child types without slicing them, and uses that to buid our container of books that can be either instances of `Book`, `Comic`, or `Text`. The ability to accommodate multiple types under one abstraction is called ***polymorphism***. It is one of the chief benefits to inheritance.

Polymorphism

```
// My favourite books are short books.
bool Book::favourite() const { return length < 100; }

// My favourite comics are Superman comics.
bool Comic::favourite() const { return hero == "Superman"; }

// My favourite textbooks are C++ books
bool Text::favourite() const { return topic == "C++"; }

// Polymorphism in action.
void printMyFavourites(Book *myBooks[], int numBooks) {
  for (int i=0; i < numBooks; ++i) {
    if (myBooks[i]->favourite()) cout << myBooks[i]->getTitle() << endl;
  }
}

int main() {
  Book* collection[] {
   new Book{"War and Peace", "Tolstoy", 5000},
   new Book{"Peter Rabbit", "Potter", 50},
   new Text{"Programming for Beginners", "??", 200, "BASIC"},
   new Text{"Programming for Big Kids", "??", 200, "C++"},
   new Comic{"Aquaman Swims Again", "??", 20, "Aquaman"},
   new Comic{"Clark Kent Loses His Glasses", "??", 20, "Superman"}
  };

  printMyFavourites(collection, 6);
  for (int i=0; i < 6; ++i) delete collection[i];
}
```

(4.16) Arrays of Polymorphic Objects

If we want to use polymorphism in combination with arrays (or any other container type), it turns out that this will only work correctly if our array holds pointers.

(4.17) Polymorphism and Destructors

Using polymorphism in combination with dynamic memory allocation poses a special problem. Let us consider a small example to illustrate the problem.

Polymorphism and Destructors (1)

```
#include <iostream>

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

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

// Run with valgrind
int main () {
  X x{5};
  Y y{5, 10};

  X *xp = new Y{5, 10};

  delete xp;
}
```

Note that each class in the inheritance hierarchy dynamically allocates an array and has defined a destructor to free the array. However, if we rutn the program through `valgrind` after compiling with the `-g` flag, we see the following.

```
$ valgrind ./a.out
==41844== Memcheck, a memory error detector
==41844== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==41844== Using Valgrind-4.14.0 and LibVEX; rerun with -h for copyright info
==41844== Command: ./a.out
==41844==
==41844==
==41844== HEAP SUMMARY:
==41844==     in use at exit: 40 bytes in 1 blocks
==41844==   total heap usage: 7 allocs, 6 frees, 72,860 bytes allocated
==41844==
==41844== LEAK SUMMARY:
==41844==    definitely lost: 40 bytes in 1 blocks
==41844==    indirectly lost: 0 bytes in 0 blocks
==41844==      possibly lost: 0 bytes in 0 blocks
==41844==    still reachable: 0 bytes in 0 blocks
==41844==         suppressed: 0 bytes in 0 blocks
==41844== Rerun with --leak-check=full to see details of leaked memory
==41844==
==41844== For counts of detected and suppressed errors, rerun with: -v
==41844== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

To resolve this issue, we have to make destructors `virtual` (even if they do nothing).

Polymorphism and Destructors (2)

```
class X {
    ...
    virtual ~X() { delete [] x; }
};
class Y: public X {
    ...
    ~Y() override { delete [] y; }
}
```

(4.18) Abstract Classes

Sometimes we do not have anything to write in the implementation of a virtual method in a base class.

```
class Student {
  protected:
    int numCourses;
  public:
    virtual int fees() const; // What should this do?
    . . .
};

class Regular: public Student {
  public:
    int fees() const override; // Computes regular student fees
};


class CoOp: public Student {
  public:
    int fees() const override; // Computes co—op student fees
};
```

But what should we put in the implementation for `Student::fees`? Every student must be either regular or co-op student. In other words, we should never create objects of the class `Student`, all objects must be created from classes `Regular` and `CoOp`. Therefore, we make `Student` an abstract class.

Def'n 4.2. **Abstract** Class
An ***abstract*** class cannot be instantiated and has at least one method that is not implemented.

The purpose of abstract classes is to organize subclasses. In C++, we create them by leaving methods without implementation. So we can explicity give `Student::fees` no implementation. This is done in C++ by adding `= 0` to the end of the declaration of a virtual method.

Creating Abstract Classes

```
class Student {
    ...
  public:
    virtual int fees() const = 0;
};
```

Here the method `fees` has no implementation. It is called a ***pure virtual method***. A class with a pure virtual method cannot be instantiated. That is

```
Student s;
```

produces an error. Note that subclasses of an abstract class are also abstract unless they implement all pure virtual methods.

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

(4.19) Inheritance and Copy/Move Operations

What happens with the objects' copy and move operations when we use inheritance? For example,

```
class Book {
  protected:
    string title, author;
    int length;
  public:
    Book(const string &title, const string &author, int length);
    Book(const Book &b);
    // Let's define the copy ctor:
    Book& operator=(const Book &rhs);
    . . . // other public methods
};

class Text: public Book {
    string topic;
  public:
    Text(const string &title, const string &author,
         int length, const string &topic);
    // Does not define copy/move ctors and operations
    . . .
      // other public methods
};

Text t {"Algorithms", "CLRS", 500, "CS"};
Text t2 = t; // No copy constructor in Text... what happens?
```

This copy initialization `Text t2 = t;` calls `Book`'s copy constructor and then goes field-by-field (i.e. default behavior) for the `Text` part. The same is true for other compiler-provided methods. However, we can also write our own implementation of the constructors and assignment operators. We do this by calling one of `Book`'s constructors/operators first, and then continuing with our implementation. For example, here is how we would implement the operations for class `Text`.

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

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

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

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

Note that even though `other` *points* at an rvalue, `other` itself is an lvalue (so is `other.topic`). Therefore, we use the function `std::move(x)`, which forces an lvalue `x` to be treated as an rvalue, so that the *move* versions of the operators run. This is important in the implementation of the move constructor and assignment operator. Otherwise, the copy versions of the operations would run. The operations given above are equivalent to the default behavior.

(4.20)
Partial Assignment

What happens if we use pointers to the superclass to assign an object to another via copy or move?

```
Text t1("Programming for Beginners", "Niklaus Wirth", 200, "Pascal");
Text t2("Programming for Big Kids", "Bjarne Stroustrup", 300, "C++");
Book *pb1 = &t1;
Book *pb2 = &t2;
```

Because we are using the `=` operator on objects of type `Book*`, `Book::operator=` runs. The result is a ***partial assignment***: only the `Book` part is copied, but `Text`'s fields are not. Figure 4.3 shows the before and after of the above assignment `*pb2 = *pb1`.

Figure 4.3

t1	t2
Programming for Beginners	Programming for Big Kids
Niklaus Wirth	Bjarne Stroustrup
200	300
Pascal	C++

t1	t2
Programming for Beginners	Programming for Beginners
Niklaus Wirth	Niklaus Wirth
200	200
Pascal	C++

(4.21)
Virtual Operations

The first solution to this problem is to make `operator=` virtual. We would change the signatures to

Virtual Operations

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

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

Note that `Text::operator=` is permitted to return (by reference) a subtype object, but the *parameter* types must be the same, or it is not an override and would not compile. With this implementation, the example above, where we assigned a `Text` object to another one using two variables of type `Book*`, is fixed. `Text::operator=` will run and all the fields will be copied correctly.

However, we now created a new problem By the *is-a* principle, if a `Book` can be assigned from another `Book`, then a `Text` can be assigned from another `Book`. Therefore, assignment of a `Book` object to a `Text` object variable would be allowed, which is called ***mixed assignment***.

Mixed Assignment between Superclass and Subclass

```
Text t { . . . };
Book b { . . . };
Text *pt = &t;
*pt = b; // call virtual operator= through pointer; subclass version runs
         // Uses a Book to assign a Text: BAD (but it would compile)
```

Also, it is now possible to use a `Comic` object to assign a `Text` variable (or vice versa).

Mixed Assignment between Subclasses

```
Text t { . . . };
Comic c { . . . };
t = c; // Use Comic object to assign Text object. REALLY BAD
```

(4.22) Abstract Superclass

Another solution is to make all superclasses *abstract*, since, as previous examples demonstrated, it is not a good idea to implement a class hirachy with non-virtual assignment/move operators, but it is also not a good idea to just make them virtual. We are going to create an abstract class `AbstractBook` which is a superclass of `Book`, `Text`, `Comic`.

Abstract Superclass

```
class AbstractBook {
    string title, author;
    int length;
  protected:
    AbstractBook &operator=(const AbstractBook &other); // Assignment now protected
    AbstractBook &operator=(AbstractBook &&other);      // Assignment now protected
   public:
    AbstractBook( . . . );
    virtual ~AbstractBook() = 0; // Need at least one pure virtual method
                                 // If you don't have one, use the dtor
};
```

Note that we are making the destructor pure virtual to make the class abstract. We however need to implement the destructor because it will be called by the subclasses when the objects are destroyed.[2]

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

Now we can create a new concrete class, so we can instantiate normal books. We can just call `AbstractBook::op`
as needed.

Concrete `NormalBook` Class

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

[2]Hence, making a method pure virtual doesn't really mean that there is necessarily no implementation; it means that the method must be overridden by subclasses. But if the base class does have an implementation, the overriding method in the subclass is free to call up to the base class implementation, for default behaviour.

```
        return *this;
    }
};
```

And we can implement the concrete classes `Text` and `Comic` in the same way (i.e. calling `AbstractBook::opera`
as needed and just copying/moving the specific fields of the subclasses). This design prevents partial and mixed assignments because copy/move assignment will not be allowed using base class pointers.

```
Text t1(. . .);
Text t2(. . .);

// The lines below will not compile
// because AbstractBook::operator= is protected, so it cannot be called here.
AbstractBook *pb1 = &t1;
AbstractBook *pb2 = &t2;
*pb2 = *pb1; // compile error

// However, it is possible to assign a Text object to another:
t2 = t1;     // this is fine
// Or using pointers:
Text *pt1 = &t1;
Text *pt2 = &t2;
*pt2 = *pt1; // this is fine
```

In summary,

(a) public non-virtual operations do not restrict what the programmer can do but allow partial assignment;

(b) public virtual operations do not restrict what the programmer can do but allow mixed assignment; and

(c) protected operations in an abstract superclass that are called by the public operations in the concrete subclasses prevent partial and mixed assignments but prevent the programmer from making assignment using base class pointers.

Unfortunately, none of these solutions are perfect as each one of them has a weakness. Generally, the third option is recommended, with abstract superclasses containing protected assignment operations, because it prevents partial and mixed assignments, thus avoiding logical errors in the program. However, it creates a limitation: it is not possible to do an assignment with superclass pointers. For some programs, this limitation may be relevant. In this case, one of the other two solutions may be more appropriate, together with measures to minimize problems of partial or mixed assignment.

Compilation Dependencies

(4.23) Compilation Dependencies

Now that we are trying to keep our header files as separate from our implementation files as possible, we need to examine under what circumstances we absolutely must include one header file in another, and under what circumstances we can simply use a forward declaration in the header file, and just include the header file in the implementation file of the other class. The letter is necessary in order to break *include cycles*, where, for instance, some file `x.h` includes `y.h` which in turn includes `x.h`. Let us start by considering some class `A` defined in the file `a.h`. There are five possible ways that `A` can be used by another class.

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

(a) *inheritance*: Must include `a.h` since compiler needs to know exactly how large class `A` is in order to determine the size of class `B`.

```
class C {
   A myA;
};
```

(b) *non-pointer*: Must include `a.h` since compiler needs to know exactly how large class `A` is in order to determine the size of class `C`.

```
class D {
   A *myAptr;
};
```

(c) *pointer*: All pointers are the same size, so a forward declaration in the header file for class `D` is sufficient, though the implementation file of `D` will need to include `a.h`.

```
class E {
   A f(A x);
};
```

(d) *method*: Despite the fact that the method `E::f` passes a parameter of type `A` by value, and returns an instance of `A` by value, the method signature is noly used for type checking by the compiler. There is thus no true compilation dependency, and a forward declaration is sufficient, though the implementation file of `E` will need to include `a.h`.

```
class F {
   void f() {
      A x;
      ...
      x.someMethod();
      ...
   }
};
```

(e) *method implemented inline*: Because class `F` wrote the implementation of method `F::f` inline, it is using a method that belongs to class `A`. Therefore, it must include the header file for `A` so that the compiler knows what methods `A` has available; however, if we moved the implementation of `F::f` to the implementation file of `F`, then we could use a forward declaration here instead. This is another reason why writing methods inline is discuouraged.

(EX 4.24) Stack and Queue Examples

Consider the following code.

stack.h

```
#ifndef STACK_H
#define STACK_H

struct Node; // forward declaration

class Stack {
    Node * ptr;
public:
    Stack();
    ~Stack();
    bool isEmpty();
    int top();
    void pop();
    void push( int value );
};

#endif
```

Note that we are foward-declaring `struct Node` but `#include "node.h"` is not included. Instead, we write it in the respective implementation file.

```
stack.cc

#include "stack.h"
#include "node.h"

Stack::Stack() : ptr{nullptr} {}
Stack::~Stack() { while ( ! isEmpty() ) pop(); }
bool Stack::isEmpty() { return ptr == nullptr; }
int Stack::top() { return ptr->data; }

void Stack::pop() {
    Node * tmp = ptr;
    ptr = ptr->next;
    delete tmp;
}

void Stack::push( int value ) {
    Node * tmp = new Node{ value, ptr };
    ptr = tmp;
}
```

Note that we have added the `#include "node.h"` statement.

In fact, we can forward declare `Node` as a `class` instead.

```
queue.h

#ifndef QUEUE_H
#define QUEUE_H

class Node; // forward declaration

class Queue {
    Node * frontPtr, * backPtr;
public:
    Queue();
    ~Queue();
    bool isEmpty();
    int front();
    void dequeue();
    void enqueue( int value );
};

#endif
```

This is perfectly legal when it comes to forward declarations, which just state that *such a type exists*, and nothing more. The respective implementation file is the following.

```
queue.cc

#include "queue.h"
#include "node.h"

Queue::Queue() : frontPtr{nullptr}, backPtr{nullptr} {}
Queue::~Queue() { while ( ! isEmpty() ) dequeue(); }
bool Queue::isEmpty() { return (frontPtr == backPtr && frontPtr == nullptr); }
int Queue::front() { return frontPtr->data; }

void Queue::dequeue() {
    Node * tmp = frontPtr;
    frontPtr = frontPtr->pnext;
    if ( frontPtr == nullptr ) backPtr = nullptr;
    delete tmp;
}

void Queue::enqueue( int value ) {
    Node * tmp = new Node{ value, nullptr };
    if ( frontPtr == backPtr && frontPtr == nullptr ) frontPtr = tmp;
    else backPtr->next = tmp;
    backPtr = tmp;
}
```

Most of the time, using forward declarations as much as possible automatically removes `include` cycles (as a side-benefit, it declutters header files also). Let us start with the case of an include cycle due to inheritance.

```
a.h

#ifndef A_H
#define A_H

#include "b.h"
class A : public B {
   ...
};

#endif
```

```
b.h

#ifndef B_H
#define B_H

#include "a.h"
class B : public A {
   ...
};

#endif
```

Conceptually, we cannot have some class `A` inherit from some class `B` and have `B` also inherit from `A`. However, in many cases, we can replace *is a* relationship with a *has a* relationship instead (i.e. use object composition instead of inheritance). But still we face an `include` cycle.

a.h (2)

```
#ifndef A_H
#define A_H

#include "b.h"
class A {
   B myB;
   ...
};

#endif
```

b.h (2)

```
#ifndef B_H
#define B_H

#include "a.h"
class B {
   A myA;
   ...
};

#endif
```

In the case of a data field being an object, the fix is to either make it a referencec to the object, or a pointer to the object. Remember, a reference is really just a constant pointer, and all pointers are the same size, so we just need a forward declaration to be aware of the type name in the header file.

a.h (3)

```
#ifndef A_H
#define A_H

class B;

class A {
   B *myB;
   ...
};

#endif
```

b.h (3i)

```
#ifndef B_H
#define B_H

class A;

class B {
   A &myA;
   ...
};
```

```

#endif
```

Note that we will also have to make a decision as to how the information for those new data fields gets set. It may be that one of them is responsible for creating the other, or the information comes through a constructor parameter in some fashion. The general rule is, then, if there is no compilation dependency necessitated by the code, do not introduce one with extraneous `#include` statements; instead use forward declarations whenever possible and include the necessary headers in the implementation files.

5. Standard Template Library

Introduction

(5.1)
Template Programming

Template programming allows us to create parameterized classes called ***templates*** that are specialized to actual code when we need to use them. The advantage is that we can use the template code to generate many concrete classes without having to duplicate code. For example, suppose that we need to implement a class `List` for `int` data and another for `float` data. We could copy and paste the code and just change the type of the private field within the `Node`s:

`IntList` and `FloatList`

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

class FloatList {
    struct Node {
      float data;
      Node *next;
      ...
    };
    Node *theList;
  public:
    ...
};
```

Of course, copying and pasting code is never a good idea. To avoid this code duplication, we can create a `List` ***template*** with a parameter that corresponds to the type of data stored in the list.

Template

```
template <typename T> class List {
    struct Node {
      T data;
      Node *next;
      ...
    };
    Node *theList;
  public:
    ...
};
```

Now our `List` class can store any type of data. To create a new `List` object, we need to specify the value of the parameter `T` (i.e. the type of data we want to store). When the program is executed, each instance of `T` in the code of the `List` will be replaced with the actual type. For example,

Creating `List` Objects

```
List<int> li;  // int is the value of the template parameter T.
               // So, each T in the List's code will be replaced with int.
li.addToFront(1);
```

```

List<string> ls;  // string is the value of the template parameter T.
                  // So, each T in the List's code will be replaced with string.
ls.addToFront("hello");
```

Roughly speaking, templates work because the compiler specializes the templates into actual code as a source-level transformation and then compiles the resulting code as usual.

Note that because of the way that templates work, the implementation of the template needs to go in the `.h` file instead of the `.cc` file as usual. Additionally, note that in the declaration of the template parameters, `typename` is the required keyword, but `T` is only a common name for the type by convention. We can use something other than `T` if we prefer. When the template has more than one parameter, it is common to use an upper-case character related to the meaning of the type. For example, we could use `K` as the name of the type for a key, `V` for a value, `I` for an index. But again, these are just conventions.

Template with Multiple Parameters

```
template <typename K, V> class Dictionary {
    K key;
    V value;
    ...
}
Dictionary<string, Student> d; // Each K in Dictionary will be replaced with string
                               // and each V will be replaced with Student
```

(5.2) Standard Template Library

The ***standard template library*** (or ***STL*** for short) is a large collection of useful templates that already exist in C++. IT contains collection classes such as lists, vectors, maps, deques, . . . , which we can use to do common tasks, as well as iterators to traverse the elements in those collections and generic functions to operate on them, such as initialization, sorting, searching, and transformation of the elements.

The template classes in the STL are explicitly structured not to allow inheritance. We cannot derive from them to extend their behavior because their methods are not virtual. We are going to use `vector` and `map` very often.

(5.3) `STL:std::vector`

The class `STL:std::vector` is a generic (i.e. template) implementation of dynamic-length arrays. For instance, we can use it to create a dynamic-length array of integers.

Dynamic-length Array of Integers

```
#include <vector>
...
std::vector<int> v; // because it's a template, we need to specify
                    // the type of data to store, which is int in this example
v.emplace_back(6);  // {6}
v.emplace_back(7);  // {6, 7}
```

The methods `emplace_back` or `push_back` can be used to add elements to the vector. The difference is that `emplace_back` creates a new object by using the class constructor before adding it to the array, whereas `push_back` copies or moves the content from an existing object into the array. Hence, we do not pass the actual object as a parameter to `emplace_back`, but we pass the *arguments* to be sued to construct the object, and `emplace_back` calls the constructor for us.

For examples, we could create a vector of `Vec` objects using `emplace_back`.

Vector of Vec Objects

```
#include <iostream>
#include <vector>

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

int main() {
    std::vector<Vec> v;
    for ( int i = 0; i < 5; i++ )
        v.emplace_back( i, i+1 ); // invokes Vec ctor
    for ( const auto & i : v )
        std::cout << "(" << i.x << ", " << i.y << ")" << std::endl;
}
```

`v.pop_back()` removes the last element of `v`.

(5.4) Looping over Vectors

We can use a `for` loop to visit a vector's contents by indexing.

`for` Loop over Vectors

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

Vectors also support the `iterator` abstraction.

`iterator` Abstraction for Vectors

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

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

To iterate in reverse,

```
for (vector<int>::reverse_iterator it = v.rbegin(); it != v.rend(); ++it) {
  ...
}
// Shorter:
for (auto it = v.rbegin(); it != v.rend(); ++it) {
  ...
}
```

Other vector operations are based on iterators. For example, the `erase` method, which removes an item from a vector, works with iterators.

`erase`

```
auto it = v.erase(v.begin()); // erases item 0; returns iterator to first
// item after the erase
it = v.erase(v.begin() + 3); // erases item 3 (4th item)
it = v.erase(it); // erases item pointed to by it
it = v.erase(v.end() - 1); // erases last item
```

Note that `v[i]` returns the `i`th element of `v` and is *unckecked*. So if we go out of bounds, the behavior is undefined. Using `v.at(i)` avoids this problem, which is a checked version of `v[i]`. If we go out of bounds, `v.at(i)` throws an `out_of_range` exception. Of course, this comes with a performace penalty because of the additional cehcking, so `v[i]` is more efficient than `v.at(i)`. Thus it may be a good idea to use `v.at(i)` at the initial stages of development and replace it with `v[i]` for production code after testing the program thoroughly.

Vectors are guaranteed to be implemented internally as arrays.

(5.5) `STL:std::map`

The class `std::map` can be used to implement dictionaries, in which unique keys are mapped to values. It is a generic (i.e. template) class, so we can define any type (comparable by `operator<`; `std::map` uses `operator<` to compare and sort the keys) for the keys and any types for the values. For example, consider a `map` of `string` keys to `int` values.

`map` of `string` Keys to `int` Values

```
 1  #include <map>
 2  using namespace std;
 3  . . .
 4  map<string, int> m;  // string is the key type, and int is the value type
 5  // Setting the values for the keys "abc" and "def":
 6  m["abc"] = 1;
 7  m["def"] = 4;
 8  // Reading the values associated with each key
 9  cout << m["abc"] << endl; // 1
10  cout << m["ghi"] << endl; // 0 (see note below)
```

Note that if a key is not found when trying to read it, such as in the line 10, it is inserted and the value is default-constructed (for an `int`, the default value is `0`).

The `erase` method can be used to delete a key and its associated value:

```
m.erase("abc");
```

The `count` method returns `1` if a key is found in the map or `0` otherwise:

```
if (m.count("def")) ... // 0 = not found, 1 = found
```

(5.6) Iterating over a `map`

Iterating over a `map` happens in sorted key order.

Iterating over a `map`

```
for (auto &p: m) {
  // Note: first and second are fields, not methods
  cout << p.first << ' ' << p.second << endl;
}
```

`p`'s type here is `std::pair<string, int>&` (`std::pair` is defined in `<utility>`).

This page intentionally left blank.

6.
Error Handling with Exceptions

Introduction

(6.1) Consider this simple `Student` class. We are writing the helper function `checkGrade`, which should report an error if the submitted grade is lower than 0 or higher than 100.

```
int checkGrade(int grade) {
    if (grade>=0&&grade<=100) {
        return grade;
    }
    // how should we report this error?
}

class Student {
    const int id;
    int assns, mt, final;
  public:
    student(const int id, int assns=0, int mt=0, int final=0)
      : id{id}, assns{checkGrade(assns)}, mt{checkGrade(mt)}, final{checkGrade(final)} {}
    float grade() const {
        return 0.4*assns+0.2*mt+0.4*final;
    }
};

int main() {
    // how would we detect here if these grades are valid?
    Student s{7899,-10,50,150};
    std::cout<<"s.grade() = "<<s.grade()<<std::endl;
}
```

What should happen when the function `checkGrade` detects an invalid grade? The problem is that `Student`'s code can detect the error but does not know what to do about it. The `Student` class does not know what the main program does or what the user interface looks like. Therefore, the decision about what to do with the error cannot be made inside the `Student` class. On the other hand, the client (i.e. the `main()` function for this example) can respond, but it cannot detect the error because it does not know the internal implementation details of class `Student`. *Error recovery* is, by its nature, a non-local problem

In C++ (and many other object-oriented languages), when an error condition arises, the function ***raises*** (or ***throws***) an ***exception***. Then what happens? Be default, execution stops. But we can write ***handlers*** to ***catch*** exceptions and deal with them.

(6.2) Throwing an Exception

Let us fist complete the code of the function `checkGrade` to throw an exception when it detects an invalid grade. In C++, we can throw anything, but the usual practice is to define specific exception classes. This makes exception handling easier because the client can catch specific exception classes. So let us create a class `InvalidGrade` and throw an exception as an instance of this class when an invalid grade is detected.

Throwing an Exception

```
class InvalidGrade {
    // add fields later
};

int checkGrade(int grade) {
    if (0<=grade&&grade<=100) {
        return grade;
    }
    throw InvalidGrade{};
}
```

Now, if we compile and run the main program, the exception will be raised when the first invalid grade is encountered. Because we did not implement an exception handler, the program execution will be stopped when the exception is raised.

```
~$ ./main
terminate called after throwing an instance of 'InvalidGrade'
Aborted (core dumped)
```

Of course, that message is not user-friendly. So it would be better to implement an exception handler to catch that exception and deal with it appropirately.

(6.3) Handling Exceptions

To handle exceptions, we use a *try-catch block*. Let us modify our main function to include a try-catch block.

```
int main() {
    try {
        Student s{7899,-10,50,150};
        std::cout<<"s.grade() = "<<s.grade<<std::endl;
    } catch (InvalidGrade) {
        std::cout<<"Invalid grade."<<std::endl;
    }
}
```

All the statements that go within the `try {}` block are *protected*, meaning that if an exception is raised while executing any of them, the execution will move to the catch block. In a catch block, we can specify the type of exception that we want to handle. In this case, we are only handling exceptions that are objects of `InvalidGrade` class. If an exception of this type is raised, the statements within the `catch {}` block are executed. After that, the program's execution continues on the next line after the `catch {}` block. But if an exception of any other type is raised, that catch block would not be executed and the program will terminate immediately just as if we did not have any try-catch.

If we comile and run the updated program, we will see our error message printed to `cout` instead of the program crashing because of the execution as follows.

```
~$ ./main
Invalid grade.
```

(6.4) Passing Information in the Exception

Our error message is better than the message displayed automatically when we do not have an exception handler. However, it would be even better if we could pass additional information in the exception, which could be displayed in the error message. As the exception is just an object, we can add fields into the class, which we can populate with information about the error before throwing the exception. For example, let us add a `grade` field to the `InvalidGrade` class. Then we will initialize it with the value of the invalid grade when we throw the exception.

Passing Information in the Exception

```
class InvalidGrade {
  private:
    int grade;
  public:
    InvalidGrade(grade): grade{grade} {}
    int getGrade() const { return grade; }
}

int checkGrade(int grade) {
    if (0<=grade&&grade<=100) {
        return grade;
    }
    throw InvalidGrade{grade};
}

int main() {
    try {
        Student s{7899,−10,50,150};
        std::cout<<"s.grade() = "<<s.grade()<<std::endl;
    } catch (InvalidGrade ex) {
        std::cout<<"Invalid grade: "<<ex.getGrade()<<std::endl;
    }
}
```

And when we compile and run the program, we will see the updated message.

```
~$ ./main
Invalid grade: −10
```

Exceptions can be any complex object; so, there is no limit on the amount of information that we can include on the object. Of course, for performance optimization, it is recommended to pass just the necessary information. However, programmers often use the attributes of the exception to provide enough detail about it, which the program can use to recover from the exception and continue its execution, or to display an appropriate message to the user.

(6.5) Continuing the Program's Execution after the Exception

A great benefit of programming with exceptions is that once we catch an exception, the program does not terminate. Execution continues right after the catch block. For example, in a loop, we could print an error message and continue executing the program after that.

```
int main() {
    Student s;
    while (true) {
        try {
            std::cin>>s;
            std::cout<<"s.grade() = "<<s.grade()<<std::endl;
        } catch (InvalidGrade ex) {
            std::cout<<"Invalid grade: "<<ex.getGrade()<<std::endl;
        }
        // write some condition to break the while loop
    }
}
```

Because the try-catch block is incide the loop, an `InvalidGrade` exception will cause an error message to be printed for that iteration of the loop. But after that, the execution will continue on the next line after the catch block. Assuming that the condition to break the while loop is not reached yet, the loop will then continue and read the nest `Student` from the input.

Exceptions and the Call Chain

(6.6) Exceptions and the Call Chain

What happens when an exception is raised in a *call chain*? For example

Exceptions and the Call Chain

```
void f() {
    throw out_of_range{"f"};
}
void g() {
    f();
}
void h() {
    g();
}
int main() {
    try {
        h();
    }
    catch (out_of_range) {
        std::cerr<<"Range error in main()" << std::endl;
    }
}
```

What happens? `main` calls `h`, then `h` calls `g`, then `g` calls `f`, then `f` raises `out_of_range`.

Control goes back through the call chain (i.e. *unwinds* the stack) until a handler is found. In this case, control goes all the way back to `main`, and `main` handles the exception. In mroe details, this is what happens.

(a) An exception is raised in `f`. Because `f` does not have an exception handler, its execution is interrupted and control goes back to `g`.

(b) Because `g` does not have an exception handler, its execution is interrupted and control goes back to `h`.

(c) Because `h` does not have an exception handler, its execution is interrupted and control goes back to `main`.

(d) `main` has an exception handler. So, the exception is caught and the body of the `catch` block is executed.

If we print something at the start and end of each function in the program above, we can see this happening.

callchain.cc

```
void f() {
    std::cerr<<"Start f"<<std::endl;
    throw out_of_range{"f"};
    std::cerr<<"End f"<<std::endl;
}
void g() {
    std::cerr<<"Start g"<<std::endl;
    f();
    std::cerr<<"End g"<<std::endl;
}
void h() {
    std::cerr<<"Start h"<<std::endl;
    g();
    std::cerr<<"End h"<<std::endl;
}
int main() {
    std::cerr<<"Start main"<<std::endl;
    try {
        h();
    }
    catch (out_of_range) {
        std::cerr<<"Range error in main()" << std::endl;
    }
    std::cerr<<"End main"<<std::endl;
}
```

```
~$ ./callchain
Start main
Start h
Start g
Start f
Range error in main()
End main
```

Note how functions `h`, `g`, `f` started but they never finished properly. That is because when the exception was raised, the function execution was interrupted before the line that printed the output at the end of the function could be executed.

(6.7) Partial Exception Handling

A handler can do part of the recovery job (i.e. execute some corrective code and throw another exception),

```
try {...}
catch (SomeErrorType s) {
    ...
    throw SomeOtherError{...};
}
```

or *rethrow* the same exception.

```
try {...}
catch (SomeErrorType s){
    ...
    throw;
}
```

This is useful when a function needs to do some cleanup, but it would not be able to completely handle the error. For example, if a function allocated dynamic memory, a partial exception handler can free it before rethrowing the original exception. Therefore, the function avoids a memory leak but lets someone else handle the exception. We will get back to this issue shortly when we discuss exception safety.

For instance, we could modify `f`, `g`, `h` from the example above to handle and rethrow the exception.

```
void h() {
    std::cout<<"Start h"<<std::endl;
    try {
        g();
    } catch (out_of_range) {
        std::cerr<<"Range error in h()"<<std::endl;
    }
    std::cout<<"Finish h"<<std::endl;
}
```

Now, if we run the modified program, we will see that each function executes its event handler and prints a message, then rethrows the exception to continue the stack unwinding.

(6.8) Exceptions in Destructors

Never let a destructor throw an exception.

Catching Exceptions with Subclasses and by Reference

(6.9)

For each `try` block, it is possible to add an unlimited number of `catch` blocks to handle different types of exceptions. For example

```
try {
    // do something
} catch (Exception1 e) {
    // handle Exception1
} catch (Exception2 f) {
    // handle Exception2
} catch (...) {
    ...
} ...
```

So if an exception of type `Exception1` is raised in the `try` block, the first `catch` block will run. If an exception of type `Exception2` is raised, the second block will run. And note that the block `catch (...) {}` acts as a *catch-all* that handles any other type of exception not handled by the previous blocks.

(6.10) Catching Exceptions by Subclasses

If exception classes use inheritance, the class hierarchy is considered by the exception handling blocks. For instance, if `E2` is a subclass of `E1`, then a `catch (E1)` will handle exceptions of classes `E1`, `E2`.

```
class E1 {};
class E2: public E1 {};
try {
    ...
} catch (E1) {
    // this will handle exceptions of type E1 or E2
}
```

(6.11) Catching Exceptions by Reference

When we use a `catch` block to catch an exception based on the superclass, such as the `catch (E1)`block in the example above, the object is sliced into the superclass type. This means that, if we have polymorphic methods in the classes, the methods that will run will be those of the superclass.

```
class E1 {
  public:
    virtual void f() {
        std::cout<<"E1"<<std::endl;
    }
};
```

```
class E2: public E1 {
  public:
    void f() override {
        std::cout<<"E2"<std::endl;
    }
};
int main() {
    try {
        throw E2{};
    } catch (E1 e) {
        e.f();
    }
}
```

Even though the exception was raised using the subclass `E2`, the `catch` block needs to assign it to a variable of type `E1`. Therefore, the `E2` object is sliced into the class `E1`. Thus the code above will use the `E1` version of the polymorphic method `f()` and will print `E1`.

However, even when we write an exception handler using a superclass to match the raised exception, we wantt to use the exception object using its correct class. To do this, we need to catch the exception *by reference*.

```
int main() {
    try {
        throw E2{};
    } catch (E1 & e) {
        e.f();
    }
}
```

Now that `e` is just a reference to the thrown exception, the polymorphic version of `f()` implemented in `E2` will run and this program will print `E2`.

In order to always treat exception objects like the kind of object that they actually are, and avoid slicing object into their superclass, catching exceptions by reference is usually the right thing to do. The maxim in C++ is *throw by value, catch by reference*.

(6.12) Rethrowing Exceptions in a Class Hierarchy

An exception can be rethrown by using the statement `throw;` or by using `throw s;`, where `s` is a variable where the caught exception was stored. In general, both approaches are similar. However, they will differ if the exception is a subclass and the exception handler caught it as an object of the superclass.

```
class SomeErrorType {};
class SpecialErrorType : public SomeErrorType {};
try {
    throw SpecialErrorType{};
}
// note that we're catching a SomeErrorType exception, not a SpecialErrorType
catch (SomeErrorType s) {
    ...
    throw;      // will rethrow the original SpecialErrorType exception
    // throw s; // would throw the original exception sliced as a SomeErrorType
}
```

The exception `s` actually belongs to a subclass of `SomeErrorType`, rather than `SomeErrorType` itself. The statement `throw s;` throows a new exception of type `SomeErrorType` (i.e. `s` is sliced into the type

`SomeErrorType`). On the other hand, `throw;` rethrows the actual exception that was caught, and the actual type of the exception is retained.

(EX 6.13) Rethrow Example

Consider the following.

```
int main() {
    try {
        try {
            throw E2{};
        }
        catch (E1 & e) {
            e.f();
        #ifdef RETHROW
            throw;
        #else
            throw e;
        #endif
        }
    } catch (E1 & e) {
        e.f();
    }
}
```

Compile it with the preprocessor variable `REThROW` (i.e. use the `g++` argument `-DREThROW`) to rethrow the original exception with `throw;`. If we run the compiled program, we wll see that it prints `E2` twice because the second time the exception is caught, it continues being of type `E2`.

If we do not include this preprocessor variable when compiling, the exception will be thrown again using `throw e;`. If we run the program, we will see that it first prints `E2` and then `E1`. That is because `throw e;` raised a new exception of type `E1` by slicing `E2` into it.

This happens even if the exception being rethrown was caught by reference because if we use `throw e;`, the type of the *reference* determines the handler, not the actual type of the object.

Exceptions from the C++ Standard Classes

(6.14) Many of the standard classes in C++ raise exceptions when an error occurs, so we can handle the error in the way that is the most appropirate for our application.

(6.15) IO Streams

By default, the IO streams such as `cin` and `cout` do not raise exceptions, so we need to check the return of the `fail()` method. However, we can use the `exceptions()` method to configure them to start raising the `ios::failure` exception when an error occurs. Then we can use a `try-catch` block to handle the xception instead of checking the `fail()` method. For example, this program catches any failure when reading numbers from t he standard input, then just clears the stream and continues reading the next number.

```
int main() {
    // set cin to raise an exception on EOF or failure
    std::cin.exceptions(ios::eofbit|ios::failbit);
    int i;
    while (true) {
        try {
            std::cin>>i;
            std::cout<<i<<std::endl;
        } catch (ios::failure &) {
            if (std::cin.eof()) { break; }
            std::cin.clear();
            std::cin.ignore();
        }
    }
}
```

(EX 6.16)
Range Errors

Many classes in the STL raise exceptions when errors occur. One common type of errors is *out of range* (i.e. when we try to access an element on an invalid index in a collection). The exception that represents this error is `std::out_of_range`. For example, the method `at()` from the class `vector` raises this exception if we pass an invalid index as tha parameter.

```
int main() {
    vector<int> v;
    v.emplace_back(2);
    v.emplace_back(4);
    v.emplace_back(6);
    try {
        std::cout<<v.at(3)<<std::endl; // out of range
        std::cout<<"got here"<<std::endl;
    } catch (out_of_range r) {
        std::cerr<<"bad range "<<r.what()<<std::endl;
    }
    std::cout<<"Done"<<std::endl;
}
```

(EX 6.17)
Dynamic Memory Allocation

When the `new` operation fails to allocate the requested memory (e.g. because there is not enough available memory), it raises the exception `std::bad_alloc`. For instance,

```
class C {
    int a[1000000];
};

int main() {
    C *b = new C[10000000];
}
```

will raise `std::bad_alloc` when run.

It is a good practice to always write a `catch (std::bad_alloc)` block when trying to allocate dynamic memory, so the program can handle the error appropirately if the operation fails.

7.
Advanced C++

Smart Pointers

(7.1) Let us identify what makes memory management so difficult in the first place. In C and C++, we have two options for storing a `struct` or `class`: in the heap or on the stack. Local variables allocate their space on the stack, so to use heap storage, we instead put a pointer on the stack, and use `new` and `delete` to allocate and reclaim its space.

The problem with stack-based storage is that it is too limiting in object lifetime. Since all stack-stored values are destroyed when the relevant function returns, all objects must exist for exactly the duration of the function that declares them. Heap-stored values do not have this restriction, but they are difficult to correctly manage. In C++ in particular, control flow can be extremely complicated.

The C++ STL offers two *wrapper classes* for pointers, which give greater flexibility than stack storage, and while not providing equal flexibility as `new` and `delete`, handle most common lifetimes for objects, as well as providing protection to assure that those lifetimes are correctly implemented.

(7.2) `unique_ptr`

A `unique_ptr` wraps a pointer, but it is guaranteed, so long as it is correctly used, to be the only pointer to the heap-allocated object in question. Since the `unique_ptr` is unique, its own destructor can delete the object pointed to. That is, the pointed-to object is deleted when the `unique_ptr` to it goes out of scope. Consider the following example.

```
class Point {
    ...
}
double pdist(std::unique_ptr<Point>) {
    ...
}
double dist(double x, double y) {
    auto p = std::make_unique<Point>(x,y);
    return pdist(std::move(p));
}
```

Note that the automatic type of *p* is `unique_ptr<Point>`. Superficially, this appears quite similar to the stack-allocated example. However, there is a critical difference.

`unique_ptr` enforce their uniqueness. Thus, a `unique_ptr` *cannot* be copied; to do so would make it non-unique. Since `unique_ptr`s cannot be copied, they cannot be passed by value.

With `pdist` taking a `unique_ptr` as its argument, we must instead *move p*. To move a `unique_ptr` is to transfer the actual, underlying pointer to another `unique_ptr`, and remove it from its starting `unique_ptr`. In this way, the *ownership* of the object may change. If we were to attempt to use *p* after the move, it would no longer be available.

```
double dist(double x, double y) {
    auto p = std::make_unique<Point>(x,y);
    double d = pdist(std::move(p));
    d += p->x; // segfault
    return d;
}
```

In this way, `unique_ptr`s have restricted lifetime, like stack values, but with much more control. Because the object is deleted when the `unique_ptr` goes out of scope, and like stack-allocated values, we do not need to concern ourselves with explicitly deleting the object, or tracking all of the particular paths through the program, including exceptions, that may need to be handled to make sure deletion occurs. But, because we can move a `unique_ptr` and thus transfer the ownership and extend the lifetime, our values may have arbitrarily long lifetimes, like heap-allocated values.

Though `unique_ptrs` cannot be passed *by value*, they can safely be passed *by reference*. This is because creating references to a `unique_ptr` keeps its uniqueness. Thus, it is not uncommon for functions to take `unique_ptr<X> &` as arguments.

For many uses, this is sufficient. But it is not uncommon to bypass `unique_ptrs` and their protections. The `get` method of `unique_ptrs` gets the underlying pointer, but it is important to note that it is still considered owned by the `unique_ptr`. That is, when the `unique_ptr` goes out of scope, the object will be deleted, even if we have extracted the pointer with `get`. This can still be necessary simply because existing functions expect standard poitners.

When considering whether to use `unique_ptrs`, we should consider the idea of *ownership*. Who *owns* this resource? That is, who should be responsible for freeing it? Does this question have a unique answer? If so, that pointer should be a `unique_ptr`. Any other pointers needed can be raw poitners, and use `get`. If this question does not have a unique answer (i.e. if there is no single *owner*), then we may need something more sophisticated than `unique_ptr`.

(7.3) `shared_ptr`

The more sophisticated option is `shared_ptr`: a pointer that is shared. Of course, a simple `*` pointer may also be shared, but a `shared_ptr` is similar to a `unique_ptr`, in that it controls the lifetime of the object it points to. Unlike a `unique_ptr`, however, we may have multiple `shared_ptr` pointing to the same object. `shared_ptrs` can be copied and passed by value, exactly like normal `*` pointers, but do not need to be explicitly deleted.

To make this possible, `shared_ptrs` use a technique called ***reference counting*** to determine when there are no more `shared_ptrs` pointing to a given object. When we copy a `shared_ptr`, it also increases an internal *reference count* by 1. When a `shared_ptr` goes out of scope, the internal reference count is reduced by 1. When it reaches 0, htere are no more `shared_ptrs` pointing to the object, so it is deleted. All of this reference counting is handled automatically in the constructor, copy constructor, and destructor for `shared_ptr` itself, so usually, all we need to do is use `shared_ptrs` instead of `*` pointers.

```cpp
double dist(double x, double y) {
    auto p = std::make_shared<Point>(x,y);
    return pdist(std::move(p));
}
```

Unlike the `unique_ptr` example, `p` needs not be removed, but can simply be shared, so `p` remains usable.

```cpp
double dist(double x, double y) {
    auto p = std::make_shared<Point>(x,y);
    double d = pdist(std::move(p));
    d += p->x; // now works
    return d;
}
```

(7.4) Caveats of `shared_ptrs`

If `shared_ptrs` only have advantages, we would not bother with normal pointers and `unique_ptrs`: `shared_ptrs` come with important caveats that must be understood. First, consider this implementation of a graph.

```
class GraphNode {
    public:
    GraphNode(sname) : name(sname) {}

    void addVertex(std::sahred_ptr<GraphNode> to) {
        vertices.push_back(to);
    }

    ...

    private:
    std::string name;
    std::vector<std::shared_ptr<GraphNode>> vertices;
}
```

This seems like a fine way of storing graph nodes and vertices. But, in many real examples, it will leak memory, failing to delete the graph nodes. For instance

```
void f() {
    auto root = std::make_shared<GraphNode>("Node 1");
    auto node = std::make_shared<GraphNode>("Node 2");
    root->addVertex(node);
    node->addVertex(root);

    ... do some graph work ...

    return;
}
```

Adding the edge from `root` to `node` increases `node`'s reference count to 2, because it now has the additional reference from `root.vertices`. Adding the edge from `node` to `root` increases `root`'s reference count to 2 also. Returning from `f` reduces each of them by 1, because the `root` and `node` have gone out of scope. But $2-1=1\neq 0$, so `root` and `node` are never deleted, and their memory is leaked.

This problem is called ***cyclic references***, and is a fundamental flaw in the reference counting technique. The only solutions to it are

(a) to reorganize the code so that no such cycles exist; or

(b) not to use reference counting (i.e. `shared_ptr`s) at all.

Usually complex data structures like this are implemented with normal `*` pointers, and controlled manually, or reorganized. An example reorganization of `GraphNode` will be presented shortly.

The second major caveat with `shared_ptr`s is that counting references is extra work, and thus a performance penalty. In many cases, this work is negligible, but if a part of the code with critical performance is constantly passing `shared_ptr`s to functions, and thus increasing and decreasing this reference count, it adds up.

Like `unique_ptr`s, `shared_ptr`s have a `get` method, allowing us to get the underlying pointer. Like `unique_ptr`, the underlying pointer is unprotected, and we can only guarantee that it is not deleted by making sure we retain a `shared_ptr` for at least as long as the `*` poitner. But, because of these downsides with `shared_ptr`s, it is not uncommon to use `get` to build segments of code that either require cyclic references, or are performance-critical.

`shared_ptr`s also have a `use_count` method, which returns thereference count. This is occasionally useful for debugging, but should otherwise never be used.

(7.5) Smart Pointers

Both `unique_ptr`s and `shared_ptr`s are so-called ***smart pointers***. Knowing which to use, and whether they are adequately smart for the purposes, requires understanding the implications of the downsides and caveats of each. Many other programming languages, such as Racket, use a technique called ***garbage collection*** to get rid of the need for explicit memory management at all, but at a performance cost similar to (but usually better than) `shared_ptr`s. Generally speaking, garbage collection is infeasible for C++.

RAII: Resource Acquisition Is Initialization

(7.6) RAII

Smart pointers are usefull for enabling a common C++ idiom: ***resource acquisition is initialization*** (or ***RAII***) . Resoureces that must be explicitly cleaned up, such as heap-allocated objects, are bound to resources which are cleaned up automatically. Do not read too much into the name. It is actually cleanup that is simplified by RAII, not acquisitions.

The RAII concept extends futher than simply using smart pointers, though. More generally, it is the concept that we should always design our classes so that object lifetimes are bound to other objects. Let us recall our `GraphNode` example from the previous section. Since the previous version misused `shared_ptr`s, we will rewrite it to use normal pointers instead.

```
class GraphNode{
    public:
    GraphNode(std::string sname) : name(sname) {}

    void addVertex(GraphNode *to) {
        vertices.push_back(to);
    }

    ...

    private:
    std::vector<GraphNode *> vertices;
}
```

With this implementation, it would be exceptionally difficult to properly delete an entire graph, because deleting a node does not delete every node it references. `shared_ptr`s were already demonstrated not to be a solution, because they create cyclic references. So how shall we redefine our graph type so that the lifetime of the nodes is properly managed?

There are many possible solutions, but the simplest for this example is to have a surrounding `Graph` type, and bind the nodes to the `Graph` itself.

```
class Graph {
    public:
    std::shared_ptr<GraphNode> createNode(std::string name) {
        auto node = std::make_shared<GraphNode>(name, nodes.length);
        nodes.push_back(node);
        return node;
    }

    std::shared_ptr<GraphNode> getNode(int index) {
        return nodes[index];
    }

    void addVertex(std::shared_ptr<GraphNode> from,
                     std::shared_ptr<GraphNode> to) {
        from->addVertex(to->getIndex());
    }

    ...

    private:
    std::vector<std::shared_ptr<GraphNode>> nodes;
}

class GraphNode {
    public:
    GraphNode(std::string sname, int sindex) : name(sname), index(sindex) {}

    int getIndex() {
        return index;
    }

    void addVertex(int to) {
        vertices.push_back(to);
    }

    std::vector<std::shared_ptr<GraphNode>>
        getVertices(std::shared_ptr<Graph> graph) {
        std::vector<std::shared_ptr<GraphNode>> ret;
        for (auto index : vertices) {
            ret.push_back(graph->getNode(index));
        }
        return ret;
    }

    ...

    private:
    std::string name;
    int index;
    std::vector<int> vertices;
}
```

In this example, instead of `GraphNode`s referring directly to other `GraphNode`s, which created the problem of cyclic dependencies, `GraphNode`s refer only to indices. The association between indices and actual `GraphNode`s is only in a single, surrounding `Graph` object, and vertices can only be resolved to nodes using both `GraphNode`s and `Graph`. This is certainly more complicated to write, but it vastly simplifies memory management, since the lifetime of all nodes is tied to `Graph`, and that `Graph` can be tied to something with a fixed lifetime, such as a `shared_ptr`. The problem of cycles vanishes, because all references to `GraphNode`s come from a single source: the `Graph`.

The principle of RAII is to build structures such as these, that assure that the lifetimes of all objects in a system are connected. The *R* in RAII stands for *resource*, and heap memory is not the only resource that a program may interact with. The RAII principles are also applied to any resource which must be properly cleaned up.

```
int getIntFromFile(std::string name) {
    std::ifstream f(name);
    int ret;
    f>>ret;
    return ret;
}
```

In C, it would have been necessary to explicitly open and close the file. In C++ with RAII, while we have explicitly opened the file, by initializing an `ifstream`, we do not need to explicitly close the file. This is because `ifstream`'s destructor closes the underlying file, so when `f` goes out of scope, the file is closed automatically.

Importantly, since these different resources are all cleaned up using the same mechanism of RAII, we can use this technique to build systems with many resources, all of which are automatically cleaned up. For instance, if, in our graph example above, each `GraphNode` was associated with a file, we would only need to make an `ifstream` (or whichever RAII-obeying type connects the file to the `GraphNode`) in the `GraphNode` class, and the file's own cleanup would be handled automatically: when the `Graph` goes out of scope, each of its `GraphNodes` are freed, and when the `GraphNode` is freed, each of its members are freed. The file stream would be one such member, so releasing the `Graph` automatically closes the file.

Let us consider several uses of a `vector`, and how they interact with RAII and memory allocation.

```
vector<Graph> d;
vector<Graph *> p;
vector<unique_ptr<Graph>> u;
vector<shared_ptr<Graph>> s;
```

These are four likely ways that a `vector` of `Graph`s would be stored. The last option `s` adheres to RAII principles and relatively easy to use. `auto g = s[x];` will increase a reference count. The `Graph` will only be destroyed when *all* references have gone out of scope. Using the `Graph` itself only increases that single reference count, which is probably an acceptable overhead.

Exception Safety

(7.7) Consider the following.

```
void f() {
    MyClass mc;
    MyClass *p = new Myclass;
    g();
    delete p;
}
```

When everything runs without exceptions, no memory is leaked. `p` is deleted on the last line of the function, and `mc` is stack-allocated, so the destructor will be automatically called during stack unwinding after the end of `f`'s execution. However, what happens if `g()` throws an exception? `mc` will still be deleted during stack unwinding. However, the last line of the function will not execute, so `p` will never deleted and that memory will be leaked. Let us see this with `valgrind`.

```
$ valgrind --leak-check=full ./01-except_leak
==151== Memcheck, a memory error detector
==151== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==151== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==151== Command: ./01-except_leak
==151==
==151== HEAP SUMMARY:
==151== in use at exit: 4 bytes in 1 blocks
==151== total heap usage: 3 allocs, 2 frees, 72,844 bytes allocated
==151==
==151== 4 bytes in 1 blocks are definitely lost in loss record 1 of 1
==151== at 0x4C3017F: operator new(unsigned long)
        (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==151== by 0x108917: f() (01-except_leak.cc:12)
==151== by 0x10893D: main (01-except_leak.cc:19)
==151==
==151== LEAK SUMMARY:
==151== definitely lost: 4 bytes in 1 blocks
==151== indirectly lost: 0 bytes in 0 blocks
==151== possibly lost: 0 bytes in 0 blocks
==151== still reachable: 0 bytes in 0 blocks
==151== suppressed: 0 bytes in 0 blocks
==151==
==151== For counts of detected and suppressed errors, rerun with: -v
==151== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

This output clearly shows us that we leaked 4 bytes (i.e. the size of an object of `MyClass`) allocated by operator `new` in `f()`.

A simple solution to avoid the memory leak is to add an exception handler to `f()`, which will `delete p` and rethrow the exception to continue the stack unwinding.

Exception Handler

```cpp
void f() {
    MyClass mc;
    MyClass *p = new MyClass;
    try {
        g();
    } catch (...) {
        delete p;
        throw;
    }
    delete p;
}
```

We can check with `valgrind` that this worked.

```
$ valgrind --leak-check=full ./02-except_handler
==162== Memcheck, a memory error detector
==162== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==162== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==162== Command: ./02-except_handler
==162==
==162== HEAP SUMMARY:
==162== in use at exit: 0 bytes in 0 blocks
==162== total heap usage: 3 allocs, 3 frees, 72,844 bytes allocated
==162==
==162== All heap blocks were freed -- no leaks are possible
==162==
==162== For counts of detected and suppressed errors, rerun with: -v
==162== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

This solution worked (i.e. the memory leak was solved). However, it is ugly and error-prone. If we forget the exception handler like the one we added to `f`, we will only detect the memory leak if we test a situation when `g` throws an exception. Additionally, we had to duplicate some code (i.e. `delete p;`), where the duplication could be much larger for a complex program.

It would be better if we could guarantee that something (e.g. `delete p;` for this program) will happen, no matter how we exist `f` (i.e. normally or by exception). It turns out that, as we saw in the previous topic, that is exactly what RAII does: it guarantees that resources will be freed at the end of the function. So let us see how we can take advantage of that to provide *exception safety* to our functions.

(7.8) Exception Safety

Generally speaking, there are three levels of ***exception safety*** for a function `f`.

(a) *basic guarantee*: If an exception occurs, the program will be in some valid, unspecified state. Nothing is leaked, class invariants maintained.

(b) *strong guarantee*: If `f` throws or propagates an exception, the state of the program will be as if `f` had not been called.

(c) *no-throw guarantee*: `f` will never throw an exception and will always accomplish its task.

At a minimum, the basic guarantee is expected of any function.

(7.9) Basic Guarantee

As we saw above, we can use simple exception handlers to avoid memory leaks. However, using RAII is an even better approach. For example, we can alter our function `f` to use a `unique_ptr` or `shared_ptr` to allocate memory for `p`.

```cpp
void f() {
    MyClass mc;
    auto p = std::make_unique<MyClass>();
    g();
}
```

This solution looks better than our previous attempt. Now `p` is a stack-allocated smart pointer. So `p`'s destructor will be called automatically as part of stack unwinding, whether `f` ended normally or because of an exception. And of course, `p`'s destructor will free the dynamic memory allocated for a `MyClass` object.

So, using RAII is a good way to avoid memory leaks and ensure a basic exception safety. However, memory leaks are not the only concern for exception safety. It is also necessary to maintain the class invariants. If a function normally makes several changes to the state of a class, an invariant may be broken if some of the changes are kept whereas others are not. Unfortunately, there is no general, easy way to avoid breaking invariants. So when implementing the methods of a class that has invariants, the programmer needs to pay attention to them and, if necessary, create exception handlers that will return the class to a consistent state after an exception.

(7.10) Strong Guarantee

As stated above, a strong exception guarantee means that if a function `f` throws or propagates an exception, the state of the program will be as if `f` had not been called. This means that any modification in the program state made by `f` needs to be undone if an exception is thrown.

```cpp
class A{...};
class B{...};
class C{
    A a;
    B b;
  public:
    void f() {
        a.g(); // may throw (provides strong guarantee)
        b.h(); // may throw (provides strong guarantee)
    }
};
```

Note that `C::f` is not exception-safe since, if `b.h()` throws, then the effects of `a.g()` must be undone to offer a strong guarantee. This is very hard or impossible if `a.g()` has non-local side-effects.

So how do we solve it? If `A::g` and `B::h` do not have non-local side effects, we could try to use copy-and-swap.

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

As shown above, we copy `a` and `b` into temporary variables and execute `A::f` and `B::g` on those temporary objects. So none of the original objects are modified if an exception is thrown by any of those tho methods. Then, after both methods are executed, we copy the modified temporary objects back into our object's `a` and `b` fields.

This solution is almost perfect. However, if the copy assignment operation `b=btemp;` throws an exception, the object `a` will already ahve been modified. Therefore, this solution only works if the copy assignment operation guarantees that it would not throw an exception, which is not always possible.

If we cannot guarantee that the assignment operation would not throw an exception, then a very good solution is to use the pimpl idiom.

```
struct CImpl {
    A a;
    B b;
};
class C {
    unique_ptr<CImpl> pImpl;
  public:
    void f() {
        auto temp = make_unique<CImpl>(*pImpl);
        temp->a.g();
        temp->b.h();
        std::swap(pImpl, temp); // no throw
    }
}
```

Again, we are making a temporary copy of the data of our object (because `make_unique<CImpl>(*pImpl)`) passes `*pImpl` as the constructor parameter, so it is invoking the copy constructor) and invking `A::g` and `B::h` on the copy. So if an exception is thrown by any of those methods, we do not have to worry because our object is still not modified. After both methods finish executing, we just swap the pointers between our `pImpl` field and the temp object. Thus, after that, `pImpl` will contain the pointer to the modified object after the execution of `A::g` and `B::h`, whereas the `temp` will only contain the poitner to the original data before modification (which will be freed automatically at the end of `C::f` by `temp`'s destuctor).

The *pointer swap opeartion* never throws an exception because it is just a swap of the memory addresses stored on each one of the already-allocated variables.

This is what guarantees that the last line of `C::f` will never fail and never throw an exception. Thus, `C::f` now offers a strong exception guarantee: if it executes normally, all the modification to `A` and `B` will be permanent. If it throws an exception at any moment, the temp object will be automatically destroyed as part of stack unwinding, but our object's `pImpl` data will remain unchanged, as if `f` had never been executed.

Note that the pimpl idiom is not the only way to accomplish this; it is one of the possible ways to do it.

The point is that a pointer swap operation never throws an exception. So, if we have pointers to objects, we can always create new pointers to temporary copies, modify the copies, and finally swap the original pointers with the copies. Although using pimpl is an option, we can also just have regular pointers or smart pointers to the objects we need and make it work without using pimpl.

Note that this only works because we previously said that `A::g` and `B::h` offer a strong guarantee. This means that if they throw an exception, the program state will be as if they had never been called. If they did not offer this guarantee, then in general, `C::f` would not be able to offer the guarantee as well. For example, if `B::h` had persistent side-effects even after it threw an exception, it would not be enough to discard the `temp` object. Although the `pImpl` object would still have its original data, we do not know what other side effects `B:h` had caused before it threw. So we would not be able to say that `C::f` offers a strong guarantee in that case.

When a function or method offers a strong guarantee, it should always be documented.

(7.11) No-throw Guarantee

Every function in C++ is either *non-throwing* or *potentially throwing*. Non-throwing functions guarantee that they never throw or propagate an exception. Therefore, if an exception is thrown by a non-throwing function, the program is automatically terminated.

In general, the default constructor, copy constructor, move constructor, copy assignment operator, move assignment operator, and destructor are non-throwing, although there are some exceptions to this rule.

Any other functions will be potentially throwing unless we declare it with `noexcept`.

```
void f() noexcept;
```

We can also pass an expression to `noexcept`. If it evaluates to `true`, then the function is declared as non-throwing.

```
void f() noexcept(true);
void f() noexcept(false); // same as if without noexcept
```

When we are writing a function that we know that can never throw an exception, it is a good idea to declare it with `noexcept`. As explained in the section above, using non-throwing functions allow other functions to also offer the no-throw or the strong guarantee.

```
class MyClass {
    int x;
  public:
    int getX() const noexcept {
        return x;
    }
    void setX(int v) noexcept {
        x = v;
    }
};
```

Pay special attention to the move constructor and move assignment operator. If all they do is swap basic values or pointers, they will never throw an exception. If that is the case, always declare them with `noexcept`. Doing so allows collection classes as `std::vector` to be more efficient when storing objects of that class, as we will see shortly.

(7.12) Exception Safety and the STL Vectors

The STL vectors encapsulate a heap-allocated array and follow RAII: when a stack-allocated vector goes out of scope, the internal heap-allocated array is freed.

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

But what happens if we have a `vector` of pointers instead?

```
void g() {
    vector<MyClass *> v;
    ...
} // v goes out of scope; pointers do not have destructors; only v is freed
```

Pointers do not have destructors. So, in the case of a vector of pointers, any objects pointed to by the pointers in `v` are not freed. The vector `v` has no way of knowing wheter deleting those pointers may be appropirate. The pointers might not own the objects they are pointing at. The objects might not even be on the heap. So if these objects need to befreed, we have to do it manually.

```
for (auto &x:v) delete x;
```

Or alternatively, we can use smart pointers.

```
void h() {
    vector<unique_ptr<MyCLass>> v;
    ...
} // array is freed; unique_ptr destructors run, so the objects are deleted
```

`unique_ptrs` have destructors, which are run by the vector's destructor when `v` goes out of scope. Then, the memory pointed at by the unique pointers are deleted and there are no memory leaks.

Using vectors of smart pointers instead of vectors of regular pointers is a good way to write exception-safe functions.

Consider now the method `vector::emplace_back`. It offers a strong guarantee. So, if the array is full (i.e. `size==cap`), the following is in summary what it needs to do.

(a) Allocate a new, larger array.

(b) Copy the objects over (copy constructor).

(c) If a copy constructor throws (strong guarantee):

  (i) destroy the new array; and

  (ii) old array still intact.

(d) Delete the old array and replace it with the new, larger array.

But copying is expensive, and the old data will just be thrown away as the last step involves deleting the old array. Would not moving the objects from the old array to the new array be more efficient?

(a) Allocate a new, larger array.

(b) Move the objects over (move constructor).

(c) Delete the old array and replace it with the new, larger array.

The problem is that if the move constructor throws, then `vector::emplace_back` cannot offer a strong guarantee, because the original array would no longer be intact. But `emplace_back` promises a strong guarantee.

Therefore, if the mobe constructor offers the no-throw guarantee, `emplace_back` will use the move constructor. Otherwise, it will use the copy constructor, which may be slower. So, as we mentioned before, our move operations should provide the no-throw guarantee, if possible, and we should indicate that they do.

```
class MyClass{
  public:
    MyClass (MyClass &&other) noexcept {...}
    MyClass &operator=(MyClass &&other) noexcept {...}
};
```

If we do this, then whenever `vector::emplace_back` needs to resize its dynamic array for more capacity, it can use the classes' move operations to quickly move all the data from the old array to the new, larger array, and still guarantee a strong exception safety.

Template Functions

(7.13) Template Functions

In C++, we can create *template functions*. For example,

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

Just like with template classes, we specify generic type names for templace functions. When we use the function, the template is then instantiated and each instance of the generic type (`T` in the example above) is repalced with the actual type.

```
int f() {
    int x=1, y=2;
    int z = min(x,y);       // T = int
    char w = mih('a','c'); // T = char
    auto f = min(1.0,3.0); // T = double
}
```

Note that there is no need to say `min<int>`. C++ can infer that `T` is `int` from the types of `x` and `y`. Note also that C++ will only deduce template arguments for template functions, and not for template classes. If C++ is unable to determine `T`, we can always tell it explicitly.

For what types T can min be used? For what types T does the body of min compile? If we look at the function's body, it uses operator< between x and y. Therefore, min will compile for any type T for which operator< is defined.

(7.14) Template Iterator Functions

We can write template functions that work with iterators.

First, recall what the public interface of a generic iterator may look like.

```
template <typename T> class Iterator {
  public:
    T &operator*() const;
    Iterator operator++();
    bool operator==(const Iterator &other) const;
    bool operator!=(const Iterator &other) const;
};
```

So, iterators always support operator*, operator++, operator!= and optionally operator==.

Knowing this, we can write a generic function that executes some other function for each element returned by an iterator.

```
Template <typename Iter, typename Func>
void for_each(Iter start, Iter finish, Func f) {
    while (start!=finish) {
        f(*start);
        ++start;
    }
}
```

The template function for_each works for any type Iter that supports operator*, operator++, and operator!=, and any type Func that can be called as a funciton. So Iter can not only be a class like the Iterator from the example above but any other type that supports these three operations, including raw pointers. For example, we can use for_each to iterate over an array using pointers.

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

Function Objects and Lambdas

(7.15) Function Objects

If we write the method operator() for a class, then we can use objects of that class as functions.

```
class Plus1 {
  public:
    int operator() (int n) { return n+1; }
};
...
Plus1 p;
p(4); // produces 5
```

In the code above, it seems like we are callign the object `p` as a function. Behind the scene, the compiler is translating `p(4)` to `p.operator(4)`.

So, we can use *function objects* with template functions like the `for_each` from the previous topic. Recall:

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

Because `Plus1` can be called as a function, we can use it with `for_each`.

```cpp
class Plus1 {
  public:
    int operator() (int &n) { ++n; }
};
...
int a[] = {1,2,3,4,5};
Plus1 p;
...
for_each(a,a+5,p);
```

(7.16) Lambda Functions

We have ***lambda functions*** (or ***anonymous functions***) in C++, which is a function not bound to an identifier and we can pass it as a parameter. For instance, we can convert

```cpp
bool even (int n) {
    return n % 2 == 0;
}
...
// iterate over the elements and count how many times f returns true
template <typename Iter, typename Func>
int count_if (Iter start, Iter finish, Func f) {
    int n = 0;
    while (start != finish) {
        if (f(*start)) { ++n };
        ++start;
    }
    return n;
}
...
vector <int> v {...};
int x = count_if(v.begin(), v.end(), even);
```

to

```cpp
// iterate over the elements and count how many times f returns true
template <typename Iter, typename Func>
int count_if (Iter start, Iter finish, Func f) {
    int n = 0;
    while (start != finish) {
        if (f(*start)) { ++n };
        ++start;
    }
    return n;
}
...
vector <int> v {...};
int x = count_if(v.begin(), v.end(), [](int n) { return n%2==0; });
```

In the syntax for a lambda function, we can put a list of *captures* inside the `[]`, so we can access outside variables inside the function, a list of `arguments` inside the `()`, and the *body* inside the `{}`.

STL Algorithms

(7.17) In addition to template classes, STL also privdes a suite of useful template functions, many of which work over iterators. We can use them with `include <algorithm>` (and they are in the `std` namespace).

We will only discuss five of them: `for_each`, `find`, `count`, `copy`, and `transform`.

(7.18) `for_each` Like the examples that we provided in the previous section, `for_each` applies a function `fn` to each of the elements in the range `[first,last)`. Here is a usage example.

```
void addOne(int &n) { ++n; }
...
vector<int> v {...};
for_each(v.begin(), v.end(), addOne);
```

(7.19) `find` `find` returns an iterator to the first element in the range `[first,last)` that compares equal to `val`. If no such element is found, the function returns `last`.

```
int myInts[] = {10,20,30,40};
int *p = find(myInts,myInts+4,30);
```

(7.20) `count` `count` returns the number of elements in the range `[first,last)` that compare equal to `val`. Note that the return type is a signed integral type.

```
int myInts[] = {10,20,30,30,20,10,10,20};
int mycount = count(myints,myints+8,10);
```

(7.21) `copy` `copy` copies the elements in the range `[first,last)` into the range beginning at `result`. It returns an iterator to the end of the destination range (which points to the element following the last element copied). Note that `copy` does not allocate new memory, so the output container must already have enough space available.

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

But consider the following.

```
vector<int> v{1,2,3,4,5};
vector<int> w;
copy(v.begin(),v.end(),w.begin());
```

This will fail because `copy` does not allocate space in `w`. We can use `back_inserter` instead, which is an iterator that calls the method `push_back`.

```
vector<int> v{1,2,3,4,5};
vector<int> w;
copy(v.begin(),v.end(),back_inserter(w));
```

`back_inserter` is available for any container with a `push_back` method.

(7.22) `transform`

`transform` applies an opeartion `op` to each of the elements in the range `[first,last)` and stores the value returned by each operation in the range that begins at `result`.

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

(7.23) Iterators

Recall that an iterator is anything that supports the operations `*`, `++`, and `!=`. So we can apply the notion of iteration with the STL algorithms to other data sources or destinations.

```
#include <vector>
#include <iterator>
#include <algorithm>
vector<int> v{1,2,3,4,5}
ostream_iteartor<int> out {std::cout, ", "};
copy(v.begin(),v.end(),out); // prints 1, 2, 3, 4, 5
```