

# Introduction

## You must use Linux in this course!

Linux is a popular environment for programmers because of its open-source nature and the availability of good tools. The popular Linux distributions usually come with all the tools that you will need for this course, such as a bash shell with script support and a C++ compiler (GCC).

But if you don't have a Linux installation and/or you never used Linux before, *don't worry!* This first module will teach you all that you need to know about Linux to use for the rest of the course.

And you don't have to install Linux by yourself. The Computer Science School provides a teaching environment that already includes a working Ubuntu Linux installation and a C++ compiler. We strongly recommend that you use this environment to run the examples and work on your assignments, even if you have a Linux computer. When you submit your assignments, we will run them on the school's server for automatic grading. Therefore, if you test your code on the school server before submitting it, you can ensure that it will work exactly the same way in your tests and our automatic tests.

## But what if I want to use my own computer?

If you have a Linux computer, you could install the compiler (g++ version 7) and test your code on your own computer. However, a lot of things can go wrong if you try to compile and run your code on different computers with different settings. So, something that worked well in your own computer may work differently (or not work at all) when we run our automatic testing scripts on the school server.

Of course, there are also C++ compilers for macOS and Windows, but those can be even more different than the C++ compiler distributed with Linux. So, the chances of your assignment submissions working differently on your

own computer and the school server are even higher.

If you still prefer to use your own computer to write the code, you can do it, but we do not recommend it and cannot offer you support (i.e., you will have to figure out how to install and setup the compiler on your own). If you do this, at least ensure that you always upload your source files to the school server (we will talk about file transfers shortly) and test them there before you submit your assignments. This way, you can ensure that your code works on the school server before any assignment submission.

In the next part, we will show you how to access the school's Linux environment from anywhere.

# Working on the Teaching Environment

The teaching environment provided by [CSCF](#) (the Computer Science Computing Facility) team can be accessed from anywhere in the world using the SSH (secure shell) protocol.

[SSH](#) is a cryptographic network protocol that provides secure (encrypted) connections between a client (i.e., your computer) and a server (i.e., the School's Ubuntu server). When you connect to the school server using SSH, you can issue commands that are executed in the server, not on your local computer. Only the output of the commands is transmitted back to your computer so you can see the results in a terminal window.

The default communication interface for SSH is text-based. Thus, when you open an SSH connection, you will see a command prompt where you can type commands and see their textual output. It is also possible to enable a graphical interface forwarding over SSH so you can use graphical programs in the server while seeing their interface on your screen, but that's something that we will cover on a future topic.

For now, let's explore the options you have to connect to the school's Ubuntu server using SSH.

## Create your student.cs password

The student.cs (teaching) environment uses its own authentication system, which is different from the Waterloo's WatIAM system. Thus, if you never accessed it before, you need to first create your password. Please use the following URL to set up your new password:

<https://www.student.cs.uwaterloo.ca/password/>

## If you have a Mac or Linux computer

The SSH client program usually comes installed on macOS and most Linux distributions. Therefore, all you need to do is open a Terminal window:

- On macOS, look for the Terminal within Applications -> Utilities
- On Ubuntu, use Ctrl+Alt+T to open a terminal window, or search for 'Terminal' in the launcher

Then, type the following command on the terminal:

```
ssh userid@linux.student.cs.uwaterloo.ca
```

(replace *userid* with your own ID, which you configured when you created your student.cs password).

The server will then ask for your password. Enter the password you created on the step above. Note that for improved security, the characters you're typing do not appear as you type them, but the input is being accepted. Press Enter/Return after typing your password to complete the login process.

## If you have a Windows computer

### Command Prompt

Windows 10 now has a pre-installed SSH client, which you can access using the Command Prompt. If you have Windows 10, this should be the simplest approach. But if this doesn't work or you have an older version of Windows, please check the two options below.

Open the Command Prompt application, which is found in the Start menu (it should be in the Windows System folder in the Start menu). Note that this Command Prompt is not a replacement for bash, which we will teach you to use in this course. You should use the Command Prompt only to connect to the school server via SSH.

Then, type the following command on the terminal:

```
ssh userid@linux.student.cs.uwaterloo.ca
```

(replace *userid* with your own ID, which you configured when you created your student.cs password).

The server will then ask for your password. Enter the password you created on the step above. Note that for improved security, the characters you're typing do not appear as you type them, but the input is being accepted. Press Enter/Return after typing your password to complete the login process.

## Putty

[Putty](#) is an open-source SSH client for Windows. Follow the link to download and install it, then run it.

In the main tab (Session), type the server address in the field 'Host Name (or IP address)':

```
linux.student.cs.uwaterloo.ca
```

(ensure that the *Port* is 22 and the *Connection type* is SSH.)

If you wish, you can also switch to the Connection -> Data tab and type your username into the field 'Auto-login username', so you don't have to type it every time.

Back in the main tab (Session), you can optionally type a session name and click 'Save'. If you do so, next time you open Putty, you just need to select the saved session and click 'Load'. Then, you won't have to type the address and your username again.

Click 'Open' to connect.

The server will then ask for your password. Enter the password you created on the step above. Note that for improved security, the characters you're typing do not appear as you type them, but the input is being accepted. Press Enter/Return after typing your password to complete the login process.

## Windows Subsystem for Linux

If you have Windows 10, you can use this feature, which installs an actual Linux distribution that runs on a terminal inside Windows. Then, you can use the SSH client that comes within Linux.

If you wish to use this feature, please follow the instructions from Microsoft to [install the Windows Subsystem for Linux](#) (unless you are familiar with another Linux distribution, we recommend using Ubuntu).

Then, open your Linux terminal and type the following command:

```
ssh userid@linux.student.cs.uwaterloo.ca
```

(replace *userid* with your own ID, which you configured when you created your student.cs password).

The server will then ask for your password. Enter the password you created on the step above. Note that for improved security, the characters you're typing do not appear as you type them, but the input is being accepted. Press Enter/Return after typing your password to complete the login process.

## If you have a mobile device

Typing code on a mobile device is not very efficient. Therefore, we recommend that you work on a desktop or laptop computer.

But if you need to eventually use a mobile device, you can check [Termius](#). It's an SSH client that has versions for Android and iOS (and also for macOS, Windows and Linux, if for any reason you don't like the options above).

On Termius's main screen, tap the + icon, select New Host. Type the server address in the 'Hostname' field:

```
linux.student.cs.uwaterloo.ca
```

If you wish, type your username in the 'Username' field so you don't have to type it every time. Save this host setup so you can quickly reload it each time you open Termius. Then, open the connection.

The server will then ask for your password. Enter the password you created on the step above. Note that for improved security, the characters you're typing do not appear as you type them, but the input is being accepted. Press Enter/Return after typing your password to complete the login process.

# Verify your Connection to the Teaching Environment

Now that you know how to connect to the teaching environment, take a moment to set up and test your connection before moving to the next topic.

You can go back to the previous lesson to review the connection instructions again if you need it.

Once you complete all the listed instructions for your chosen platform, you should have a terminal open, where you can type commands.

To verify that you have the right connection, enter these two commands (press Enter/Return after each one to execute them).

```
hostname -f  
echo $0
```

The output of `hostname -f` should be something like `ubuntu1804-XXX.student.cs.uwaterloo.ca` (where XXX may change each time you connect), which means that you are correctly connected to one of the servers in the teaching environment.

What is the output of the command `echo $0`?

- bash
- zsh
- tcsh
- other

Yes, that's great! bash is a shell, a program that takes commands from the user and passes them to the operating system. In the next modules, you will learn more about bash and how to interact with it.

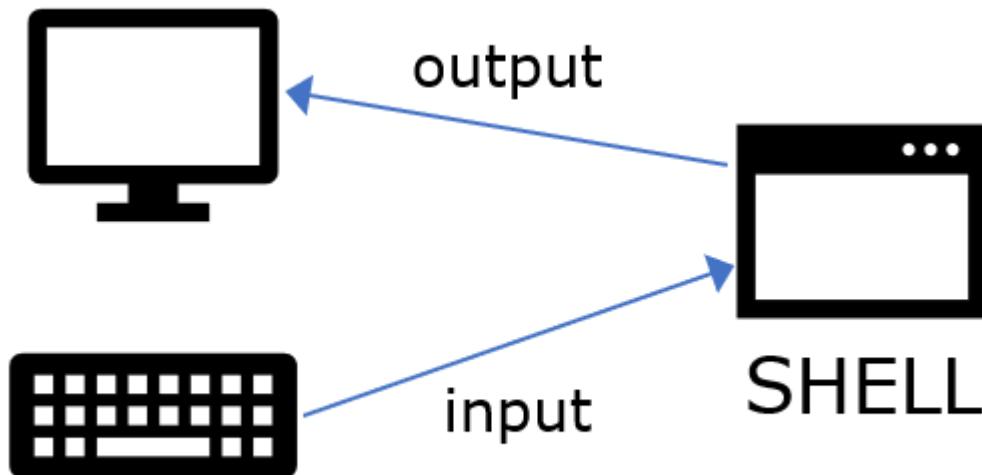
# What is a Shell?

A **shell** is a program that runs in your computer and provides a user interface, which allows you (the user) to communicate with the operating system. Usually, a shell can be a command-line interface (CLI) or a graphical user interface (GUI). In this course, we will focus on command-line interfaces only.

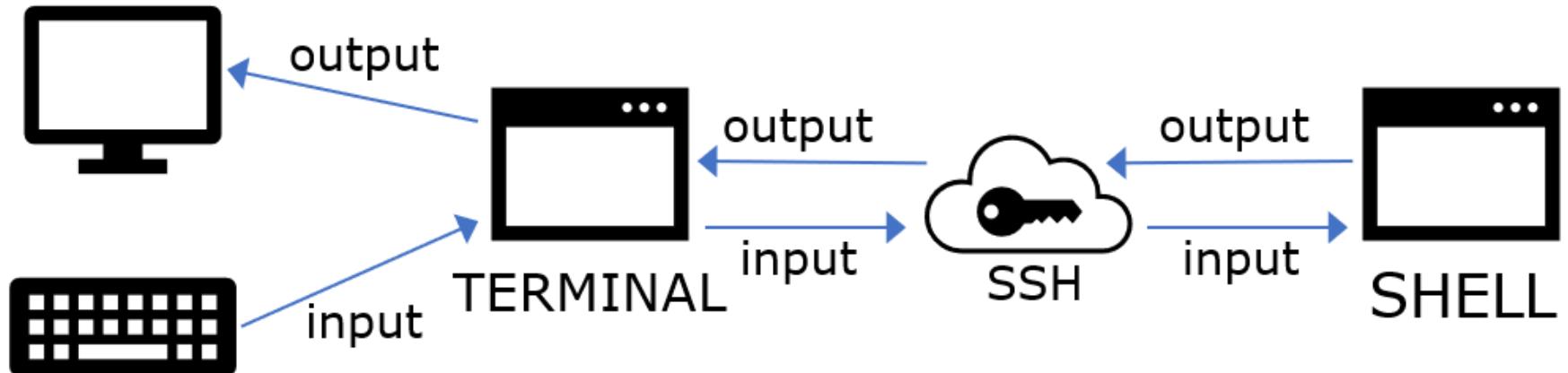
In most Linux distributions, a program called [bash](#) acts as the shell. This name stands for *Bourne Again Shell*, which is an enhanced version of Unix's original shell program, sh. There are other programs that can be installed on a Linux system to act as the shell instead of bash, but we will be focusing only on bash in this course.

## Input and Output

The shell manages user input and output and interprets and executes commands. In the most simple environment, i.e., when you are interacting with a shell running on your own computer, the shell is receiving text input from your keyboard and sending text output to your monitor:



However, when you are using the shell on the teaching environment through an SSH (Secure SHell) connection, what is actually happening is a bit more complex. First, you are not typing commands directly into the shell, you are using a terminal, which is a graphical program that provides a command-line interface (CLI). Additionally, the terminal is sending your input to the shell running on the server through the Internet using the SSH communication protocol. Similarly, the shell is sending the output back to your computer through the Internet using SSH, and this output is received and displayed on your monitor by the terminal. So, your actual environment probably looks more like this:



But regardless of exactly how the environment looks like, from the point of view of the shell, it does not matter. The shell only needs to know that it is receiving text input from an input device and it is writing text output into an output device. So, for the purposes of this course, you can ignore the existence of the terminal and SSH, and think as if the shell was receiving input directly from your keyboard and writing output directly into the terminal window on your monitor.

## Standard Input and Output Devices

As the shell can transparently read input from different devices and write output to different devices, it is useful to employ a common language to refer to the device where it currently is reading from or writing to. In fact, there are three standard input and output devices used by the shell:

- **Standard Input** (or `stdin` for short): is a device from where the shell is reading text input. Typically, the shell reads input from a keyboard physically connected to the computer where it is running or from a remote keyboard whose input is being transferred to the shell via an SSH connection.
- **Standard Output** (or `stdout` for short): is a device to where the shell is writing text output. Typically, the shell writes output to a monitor physically connected to the computer where it is running or to a local or remote

terminal via an SSH connection.

- **Standard Error** (or `stderr` for short): similar to the `stdout`, it is a device where the shell is writing text output. The difference is that the shell and the programs we run usually write normal output to `stdout` and error messages to `stderr`. By default, both output devices write to the same place (the monitor or terminal). However, as we will see shortly, we can redirect these devices so that normal and error messages are displayed in or saved to different places. One advantage of doing so is that error messages do not clutter the output files and corrupt formatting. Also, `stdout` may be *buffered*, meaning that the system may wait to accumulate output before actually displaying (*flushing*) it. On the other hand, `stderr` is never buffered, so error messages are displayed immediately.

# The File System

As you probably know, the files on our computers are organized into **directories** (also known as **folders**). This organization is implemented by the **file system**. A **path** is the general name of a file or directory in a textual format. In Linux (and other Unix-like operating systems), the character / is used as the delimiter between directory and file names.

Note that in Linux, a directory is just a specialized form of file. While it can be edited, it's dangerous to do so.

## Absolute and Relative Paths

The path denoted by only / is the **root**, i.e., the starting point of the file system. Therefore, all paths are somehow relative to the root directory. Therefore, any path that begins with / is an **absolute path** because it clearly identifies where the file or directory is in the file system. For example, these are all valid absolute paths:

/ -> the root of the file system

/file.txt -> a file named file.txt in the root directory of the file system

/etc/ -> a directory named etc in the root directory of the file system (the trailing / is optional)

/etc/file.txt -> a file named file.txt in the directory /etc

/etc/subdir/abc/file.txt -> a file named file.txt in the directory /etc/subdir/abc

As you can see, subdirectories can be nested within other directories.

However, when you write just a file name, such as file.txt, how does the shell know where to find the file? Any path that does not begin with a / is a **relative path**. A relative path is always *relative* to the current directory (or **working directory**). When you are interacting with the shell, it always keeps track of what is the current directory.

There are also two special directory names. A dot (.) identifies the current directory. Two dots (..) identifies the parent of the current directory. So, these are all examples of relative paths:

`file.txt` -> a file with this name in the current directory

`subdir/abc/file.txt` -> a file named `file.txt` within the directory `subdir/abc`, where `subdir` must be in the current directory

`./script.sh` -> a file named `script.sh` in the current directory

`../bin/script.sh` -> a file named `script.sh` in the `bin` directory, which must be in the parent of the current directory

## Changing the Current Directory

The command `cd` can be used to change the current directory. Therefore, after using the command `cd`, any relative paths will be relative to the newly-set current directory. `cd` can receive absolute or relative directory names as the parameter.

The command `pwd` can be used to display the current directory. For example, we use `cd` in the listing below to change to some relative or absolute paths, and use `pwd` after each call to `cd` to verify what is the current directory:

```
$ pwd  
/home/gustavo  
$ cd cs246  
$ pwd  
/home/gustavo/cs246  
$ cd ..  
$ pwd  
/home/gustavo  
$ cd /bin
```

```
$ pwd  
/bin
```

## The Home Directory

In a Linux system, each new user is given their own **home directory**. This is a directory owned by the user, where we can put all our personal files without disturbing other users' files.

In the teaching environment, our home directories are usually in /u/userID or /uX/userID, where X is a number (0-9) and userID logically is the user id. This is also the default current directory each time we connect to the server (unless you specify a different default directory in the connection options).

Note: /u is used to make the entire undergrad environment look like a single machine because there are too many student accounts for them all to be on a single server. Instead, our accounts are distributed across the different machines, i.e., appearing on one of /u0, /u1, etc. So, /u/userID is just another name (an *alias*) for /uX/userID, where X is the number of the server where the directory actually resides. Since we have no idea which machine each account may be on, we usually leave the 0-9 out of our instructions (e.g., on the assignment specifications).

When you write a pathname, the shell replaces the special character ~ with the absolute path of your home directory. You can use this to write absolute paths to files within your home directory. For example, ~/cs246/readme.txt identifies a file named readme.txt in the subdirectory cs246, which is in the current user's home directory. In effect, this will be automatically translated by the shell to /uX/userID/cs246/readme.txt.

**Note:** when you are referring to your home directory, be sure to use the character ~, not ^. The first is the simple tilde character that you get when you type it directly on the keyboard. The second is a fancy formatted tilde that is generated by some text editors when you type a regular tilde. The second version is not recognized by bash. If you are having trouble when trying to reach your home directory, especially if you copied the path from a document or PDF, check your tilde character.

## The \$PATH Variable

We said above that any relative pathname will be relative to the current directory. However, there is one exception, that's when you are trying to run a program or script. When you give a command to the shell that is just the file name (without a named directory) of a program or script, the shell does not look for the file in the current directory. There is a global system variable named \$PATH that specifies where the shell should find executable files. The contents of the \$PATH variable are a list of pathnames separated by the colon character (:). For example:

```
$ echo $PATH  
/home/gustavo/bin:/usr/local/bin:/usr/bin
```

If the contents of the \$PATH variable are like the example above, then any command that only contains a file name will look for executable files in those directories only. If you need to execute a program or script that is not in one of those directories, you must, therefore, use a relative or absolute pathname that contains at least one specific directory name. But what if the program I want to execute is in the current directory? That's when the special directory name . is useful as it identifies the current directory. For example:

```
$ myprogram -> looks for a file named myprogram in one of the directories in the $PATH system variable  
$ ./myprogram -> looks for a file named myprogram in the current directory  
$ ./subdir/myprogram -> looks for a file named myprogram in the directory subdir, which is in the current directory  
$ /opt/program/myprogram -> looks for a file named myprogram in the directory /opt/program
```

Alternatively, if you need to frequently execute a program that is not in one of the directories listed in the \$PATH variable, you can modify the variable to add a new directory. For example, if you want to frequently execute a program that is installed in /opt/program, use this command to add this directory to the \$PATH variable (without removing the current contents of the variable):

```
export PATH=$PATH:/opt/program
```

If you want to learn more about how to set the \$PATH variable, here is an [additional online article](#).

# File Permissions

## The command `ls`

The command `ls` can be used to list the files (and directories) in the current directory (or any directory if you pass a path as a parameter). When used with the argument `-l` (`ls -l`), it gives a "long form" listing, which displays the file permissions and other information. The format is as follows:

```
$ ls -l  
-rw-r----- 1 j2smith cs246 25 May 4 15:27 abc.txt  
type/perm  owner   group size modified name
```

- **type:** - for an ordinary file; d for a directory.
- **permissions:** three groups of three bits, which we will explain below.
- **owner:** The ID of the user that owns the file. This is usually the user who created the file, although ownership can be transferred to another user (using the command `chown`).
- **group:** A user can belong to one or more groups. A file can be associated with one group. When a user and a file are in the same group, the group permissions are in effect for that user (more details below). Group association can also be modified (with `chown`).
- **size:** The size of the file in bytes.
- **modified:** The date and time when the file was last modified.
- **name:** The file name.

# File Permissions

Each file or directory has a set of three types of permissions: user, group, and other:

- **user permissions:** apply to the file's owner user
- **group permissions:** apply to members of the file's group (other than the owner)
- **other permissions:** apply to everyone else

Each one of these types may contain three specific permissions: Read, Write, and eXecute.

| Bit | Meaning for ordinary files                             | Meaning for directories   |
|-----|--|---|
| r   | file's contents can be read                            | directory's contents can be read (e.g., ls, globbing, tab completion) |
| w   | file's contents can be modified                        | directory's contents can be modified (can add/remove files)           |
| x   | file's contents can be executed as a program or script | directory can be navigated into (i.e., can cd into the directory)     |

Note: If a directory's execute bit is not set, there is no access at all to the directory, nor to any file within it, nor to any subdirectory, no matter how the other bits are set.

These permissions are displayed by ls as the nine characters that follow the type bit, in the format: rwxrwxrwx — the first three are the user permissions, the middle three are the group permissions, and the last three are other permissions. If a particular permission is not set, then a - is displayed instead. For example:

rwxrw-r-- means that the permissions rwx are set for the user, rw for the group, and just r for others

rw-r----- means that the permissions rw are set for the user, r for the group, and none for others.

There are two other permission codes that you may sometimes see, s and t. You don't need to know about them for CS246, but if you want to know a bit more about them, you can start with man chmod.

# Changing File Permissions

To change file permissions, use the command: `chmod <mode> <file>`.

`<file>` is the name of the file you want to change (a globbing pattern can be used to change multiple files)

`<mode>` consists of three parts:

1. user types: u (user/owner), g (group), o (other), or a (all)
2. operator: + (add permission), - (subtract permission), or = (set permission exactly)
3. permissions: r (read), w (write), and/or x (execute)

## Examples:

- Give others read permission: `chmod o+r file`
- Make everyone's permission to read and execute (also removing write permissions): `chmod a=rwx file`
- Give owner full control: `chmod u+rwx file` (or `chmod u=rwx file`)

## Exercise:

What is the command to ensure that members of the file's group can read and write, but not execute the file?

- `chmod g+rw file`
- `chmod a=rw file`
- `chmod g=rw file`

Yes, that's correct. This will give the read and write permission to the members of the file's group, and it will remove the execute permission if it was previously set.

### Be careful with permissions in your home directory!

When you are working on the teaching environment, there is probably no reason at all to make your personal files accessible to anyone other than yourself. So, be careful to only ever assign permissions to the user (you) and never assign any permission to the group or others when you are in the teaching environment unless you have a valid reason to do so.

**As we are all working on the same servers, if you make files inside your home directory accessible to others, then other server users (i.e., students, professors, etc.) will be able to see your files. For example, if you make assignment solutions readable by others, and another student finds your solutions, this may lead to an academic integrity violation. Luckily, there is an easy way to protect yourself: once you start working with the course material on your home directory, check the permissions and ensure that the place where you are storing your work (for example, in a folder named cs246) does not have any permissions set to others.**

# Executing Commands

bash can run in an interactive or a non-interactive mode. In the interactive mode, it receives commands directly from the user. In the non-interactive mode, the commands are being read from somewhere else, such as a script file, so user input is not needed. In this module, you will learn how to use the shell in the interactive mode. Later, you will learn how to write shell scripts.

## The Command Prompt

In the interactive mode, when the shell is waiting for user commands, it displays a prompt. The prompt usually displays some useful information, such as the id of the current user, the current working directory, and/or the computer id. It usually ends with a character such as \$ or %. The prompt can be fully customized by the user. We will not go into details on this course about how to customize the prompt, but there are lots of articles available online if you want to customize yours (for example, you can check [this](#) as a starting point).

For example, the Ubuntu prompt (installed as the Windows Subsystem for Linux in my computer) currently displays:

```
gustavo@GFT-PC:~$
```

where gustavo is my user id, GFT-PC is my computer id, and ~ is the current working directory (more on this later). \$ only marks where the prompt ends and my input will begin.

As another example, if I connect to the teaching environment, the prompt currently displays:

```
@ubuntu1804-002%
```

where ubuntu1804-002 is the computer id and % is the character marking where the prompt ends and my input will begin.

In these lessons, we will usually just use the character \$ to represent the prompt, as the actual contents of the prompt may be customized. So, if you see something like this in a lesson:

```
$ ls
```

it means that you should type the command `ls` after the prompt and press Enter/Return to execute it.

## Types of Commands

bash can understand a few native commands, such as those used to set variables or control execution flow, which you will learn about in the [bash scripts module](#). However, most of the time, you will give commands to execute scripts or programs.

**Scripts** are text files that contain commands in a specific programming language. They are not executed directly by the operating system. Instead, their contents are read by a program that interprets and executes the commands (the interpreter). Examples of interpreters are bash (yes, the bash program can also work as a script interpreter when in the non-interactive mode), python, php, etc.

**Programs** are files that contain commands in binary format. This means that the file contents are sequences of bytes that are meant to be understood by the operating system, not by a human. Therefore, programs can be executed directly by the operating system, without needing an interpreter. The usual way to create a program is to write it using a specific programming language and then compile it. You will learn more about [compilation](#) in the C++ module.

When you type a command in the prompt, bash will interpret it. First, it will try to identify if it can be recognized as a native command. If not, then it will look for a script or program on the computer and will execute it if such a script or program is found. Otherwise, it will show a message such as "Command not found."

For example, `ls` is a program normally distributed with Linux. So, go ahead and try it! Type `ls` and press Enter to execute the program. What is the output?

- An error message
- A listing of the files in the current directory
- Instructions about how to execute bash commands

Yes, that's correct. `ls` is a program that displays a list of files and folders in the current working directory.

# Input/Output Redirection

## The command cat

The command `cat` simply reads the contents of standard input and writes it back to standard output.

Try it out by just typing `cat` at the prompt and pressing Enter. The program will run and start waiting for input. Type anything, such as `Hello, Linux`, and press the Enter (or return) key. You will see that the same text is written to standard output.

Press the key combination `Ctrl+D` (usually displayed as `^D` in code listings) when you are done. `^D` is a special character known as EOF (end-of-file). It signals to the program that the input has ended. Therefore, `cat` will terminate its execution when it detects an EOF signal from the input.

```
$ cat  
Hello, Linux  
Hello, Linux  
^D
```

## Redirecting Input and Output

Using `cat` to simply read from the keyboard and output to the monitor does not seem very useful. However, you can use the shell to **redirect** `cat`'s standard input and output devices to read from or write to files.

You can use the character `<` to redirect the standard input. For example:

```
cat < file1.txt
```

In this case, cat is still doing the same, reading from the input and writing back to the output. However, before executing cat, the shell will look for a file named `file.txt`, open it, and redirect cat's standard input to be from this file instead of the keyboard. Therefore, when cat reads from `stdin`, it will now read the contents of the file instead of keyboard input.

Similarly, you can redirect the standard output using the character `>`. For example:

```
cat > file2.txt
```

Now, the shell will create a file named `file2.txt` and redirect cat's standard output to write into this file. So, cat will be reading input from the keyboard, but the output will now be saved into `file2.txt` instead of displayed in the screen.

You can also combine both redirections. For example:

```
cat < file1.txt > file2.txt
```

Now, because of the I/O redirections, cat will be reading input from `file1.txt` and writing its contents into `file2.txt`. In practice, this will create a copy of the contents of `file1.txt` into `file2.txt`.

Finally, you can redirect the standard error device with `2>`. For example:

```
cat < file1.txt > file2.txt 2> log.txt
```

This will make cat read input from `file1.txt` and write the output into `file2.txt`. However, any error messages will be written into `log.txt` instead.

# Command-line Arguments

You can also pass arguments to a program or script by writing each argument separated by a space after the command name. These are called **command-line arguments**. For example:

```
cat -n file1.txt
```

This will make two arguments available to cat when it is executing, `-n` (which tells cat to prefix each line with a number) and `file1.txt` (the name of the file to read).

## Difference Between Arguments and Input Redirection

When cat receives a file name as an argument, it will open the file and read its contents instead of reading from the standard input. So, the command below will have the same practical effect as the one above, i.e., both will cause cat to print the contents of `file1.txt` with the lines numbered.

```
cat -n < file1.txt
```

However, they are not technically the same command. For the first command, cat will receive the string (text) `file1.txt`, then it will have to look for a file with that name and read it. In the second command, the shell will open `file1.txt` and redirect its content into cat's standard input. Therefore, cat will not receive the file name and will read only from standard input.

A program somewhat similar to cat is echo. However, echo does not read from input. It just prints each command-line argument to stdout. For example:

```
$ echo a file1.txt  
a file1.txt
```

```
$ echo a < file1.txt  
a  
$
```

The first command just printed a `file1.txt` into the standard output stream. Note how `echo` does not try to interpret the argument as a file name, it just prints the argument(s) back into `stdout`. The second command only printed the string `a` because it was the only argument. `echo`'s `stdin` was redirected to `file1.txt` by the shell, but because `echo` does not read anything from `stdin`, that input was not used. This demonstrates the difference between standard input and command-line arguments.

When you start writing shell scripts and C++ programs later in the course, you will also see that the way to deal with input or arguments is different, like it was when you wrote C programs if you took CS 136.

Note: `echo` is also a very useful little program to write output (e.g., feedback or error messages) into the standard output stream.

0:00 / 11:17



## Quoting Arguments

When passing command-line arguments to a program, the shell will treat spaces as separators between arguments. If you want to pass a single argument that contains spaces within it, you need to quote the argument with double ("") or single ('') quotes. The difference is that single quotes will not interpolate anything (e.g., variables), whereas double quotes will. For example:

```
$ echo a b c  
a b c  
$ echo "a b c"  
a b c
```

In the first command, a, b, and c will be three separate arguments. Because echo prints back each argument separated by a single white space, the output is a b c.

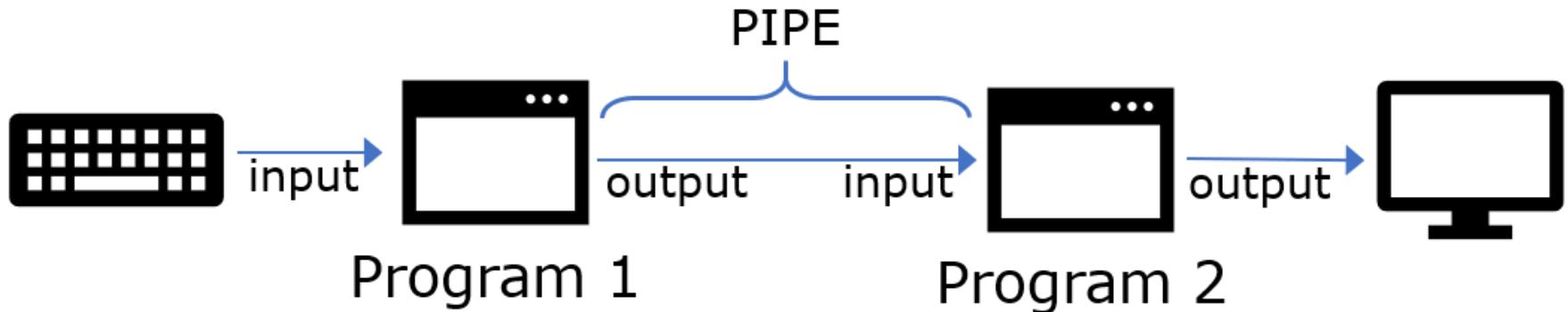
In the second command, a single argument is passed to echo: "a b c" (with the quotes removed). So, echo prints back exactly the contents of the single argument.

```
$ echo "My shell is $0"  
My shell is -bash  
$ echo 'My shell is $0'  
My shell is $0
```

In the first command, the variable \$0 will be replaced by its value when the shell passes the argument to echo because of the double-quotes. In the second command, because of the single quotes, no variable substitution will occur.

# Pipes

**Pipes** allow us to use the output of one program as the input of another by connecting the second program's stdin to the first program's stdout.



For example: how can we count how many words occur in the first 20 lines of `file.txt`?

- The command `head -n` gives the first n lines of a file.
- The command `wc -w` counts the words in the input.

Therefore, we can combine these two programs by piping them. A pipe is made using the character |:

```
head -20 file.txt | wc -w
```

Note that `head` will run first and output the first 20 lines of `file.txt`. However, this output will not be displayed on the screen. Instead, it will be piped as the input to `wc`. `wc` will then count the number of words in the file and return the result.

There is no limit on how you can combine piping and I/O redirection. For example, you can use as many pipes you want on the same command and combine with a redirection:

```
cat file.txt | head -20 | wc -w > words.txt
```

This will produce the same output as before, but first cat will run and output the contents of file.txt; this output will be piped as the input to head, which will output only the first 20 lines; this will be piped as the input to wc, which will count the number of words; and finally, the output of wc will be redirected and saved into the file words.txt.

## Exercise

Suppose that words1.txt and words2.txt contain lists of words, one per line. How can we print a duplicate-free list of all words that occur in any of these files, knowing that:

- uniq removes adjacent duplicate lines from a stream. If the entries are sorted, it therefore removes all duplicates;
- sort sorts lines

- cat words1.txt words2.txt > sort > uniq
- cat words1.txt words2.txt | uniq | sort
- cat words1.txt words2.txt | sort | uniq

Yes, that's correct. cat will read the files and pipe the output to sort. After the lines are sorted, the output will be piped to uniq to remove the duplicates.

## Using output as command-line arguments

It is also possible to use the output of one program as a command-line argument of another. For example:

```
echo "Today is $(date) and I am $(whoami)"
```

Here, the shell executes the commands date and whoami, and substitutes the results into the command line. But be careful:

```
echo 'Today is $(date) and I am $(whoami)'
```

literally prints Today is \$(date) and I am \$(whoami). Substitution occurs only inside double quotes, never inside single quotes.

Students often want to combine the use of echo with the use of cat to output the contents of a file. However echo changes the whitespace of the text passed to it. For example, here's a file with multiple lines of text:

```
$ cat helloworld.txt
hello
    world
```

I can use \$( ) to tell the shell to execute the cat command before passing the result as command-line arguments to echo:

```
$ echo $(cat helloworld.txt)
hello world
```

And the result does not match the original text in the file.

**Don't use echo in conjunction with cat if you need to preserve the original file format!**

**Use cat by itself if you need to preserve the original file format!**



# Globbing Patterns

The shell can automatically expand a few wildcard patterns to match all the files that satisfy the pattern. This is known as **globbing**. For example:

```
cat *.txt
```

The syntax `*.txt` is known as a **globbing pattern**. In the context of globbing patterns, the wildcard character `*` means "match any sequence of characters". When the shell encounters a globbing pattern, it finds all (non-hidden) files in the current directory that fit the pattern and substitutes them onto the command line. For example, the above command line might be replaced with:

```
cat abc.txt bcd.txt cde.txt
```

(This is assuming that the files in the current directory that match the pattern `*.txt` are those given above.)

This is where `cat` gets its name (conCATenate).

The following operators can be used in globbing patterns:

| Operator                 | Meaning  |
|--------------------------|--|
| <code>*</code>           | matches zero or more characters                              |
| <code>?</code>           | matches one character  |
| <code>[abc]</code>       | matches exactly one of the characters in the brackets        |
| <code>[!abc]</code>      | matches any character <i>except</i> the ones in the brackets |
| <code>[a-z]</code>       | matches any character in the given range                     |
| <code>{pat1,pat2}</code> | matches either pat1 or pat2 (note no spaces)                 |

More information on globbing patterns and the history behind the concept can be found through use of the command:

`man 7 glob`

in the student environment. Note that while globbing patterns share certain similarities with [egrep regular expressions](#), they are **not** the same thing.

# Introduction to pattern matching

Now we have the ability to pipe the output of one program into another program. But, what should we do with it? One of the most common things to do with pipes is process data, and one of the most common ways of processing data is **pattern matching**.

In this module, we will look at pattern matching with the tool egrep. egrep looks for lines that match a pattern. You may have heard of grep: grep is egrep's slightly less powerful predecessor, but most of what you learn here will apply to both grep and egrep. The term "grep" has also entered programming jargon as a synonym for "search", because of these tools!

Let's imagine that we have a list of tasks in a text file named `tasks.txt`. Some of the tasks relate to CS246, and we would like to find those tasks. This command will output every line that contains "CS246" in the file `tasks.txt`:

```
$ egrep "CS246" < tasks.txt
```

For example, if the file `tasks.txt` contains the following:

```
CS241: Finish assignment 1
CS246: Assignment 2
cs246 : Exam 1
ANTH221: Exam 1
anth221: Discuss conflict with CS246
```

Then egrep will behave as follows:

```
$ egrep "CS246" < tasks.txt
CS246: Assignment 2
anth221: Discuss conflict with CS246
```

But, note that we've missed something, and included something we may not have wanted to. We missed the line "cs246 : Exam 1" because it was not capitalized in the same way as the line "CS246: Assignment 2", and egrep is case-sensitive. In order to resolve this, we will use a more sophisticated pattern, which allows for the "CS" to be upper- or lower-case:

```
$ egrep "(CS246|cs246)" < tasks.txt  
CS246: Assignment 2  
cs246 : Exam 1  
anth221: Discuss conflict with CS246
```

The patterns that egrep accepts are called **regular expressions**, and they're more powerful than the [globbing patterns](#) we've seen before. A regular expression is a pattern formed by combining smaller patterns. We will go over the various ways of combining patterns here. The syntax  $(X|Y)$  is a pattern that matches either X or Y, and X or Y can themselves be patterns. In the above example, the two patterns we combined with | were similar already, so we can simplify the pattern a bit more by just focusing on the "CS" part:

```
$ egrep "(CS|cs)246" < tasks.txt  
CS246: Assignment 2  
cs246 : Exam 1  
anth221: Discuss conflict with CS246
```

Alternatively, egrep has an optional `-i` argument, which makes its pattern matching *case-insensitive*:

```
$ egrep -i "CS246" < tasks.txt  
CS246: Assignment 2  
cs246 : Exam 1  
anth221: Discuss conflict with CS246
```

However, `-i` is a rather blunt instrument. In this example, it would also match "Cs246" and "cS246", which may not be desirable.

We've now resolved the problem with failing to match "cs246", but we're still matching the line "anth221: Discuss conflict with CS246", which isn't actually a CS246 task. Luckily, we've organized our file in a consistent way. All we

need to do make a pattern not just for "CS246", but specifically for "CS246" at the beginning of a line, followed by a ":". Matching ":" is the simple part: The pattern for a colon is a colon. A special pattern that regular expressions support is ^, which means "beginning of a line". So, to make our pattern match what we want, we can combine it with the ^ and : patterns as follows:

```
$ egrep "^(CS|cs)246:" < tasks.txt  
CS246: Assignment 2
```

But, now we've missed the "cs246 : Exam 1" line again. This time, it's because of the space. We'll tackle that issue and more advanced patterns in the next part.

0:00 / 3:15



# Advanced patterns

In the previous module, we used egrep to match lines in a file containing a list of tasks. When we left off, we'd missed the "cs246 : Exam 1" line. We can read this pattern step by step to understand what went wrong: `^` means "match the beginning of a line". `(CS|cs)` means "match either CS or cs". `246:` means "match 246 followed by a colon". That last part of our pattern didn't allow for the space in our original file. We already have a tool to match for one of two patterns, so one solution to this problem is simply to match for the pattern `" "` or `""` (that is, the empty pattern, which looks for nothing):

```
$ egrep "^(CS|cs)246( |):" < tasks.txt
CS246: Assignment 2
cs246 : Exam 1
```

This solution is a bit unsatisfying, and we can do better. It's common to match for a pattern or nothing, so there's a shorthand that's equivalent, `?:`:

```
$ egrep "^(CS|cs)246 ?::" < tasks.txt
CS246: Assignment 2
cs246 : Exam 1
```

Note that `?` makes a *pattern* optional. In this case, the pattern is just the space before the `?`. If we want to make more than one character optional, we can group patterns together with parentheses. In fact, we already did that, to use `|`. We could make both the `6` and the space optional like so (in this example, we'll also remove the colon, just to match something more interesting):

```
$ egrep "^(CS|cs)24(6 )?:" < tasks.txt
CS241: Finish assignment 1
CS246: Assignment 2
cs246 : Exam 1
```

In this case, we don't need a larger group than the space, but by just making the space optional, we're being fairly limited. What if my file had two spaces, or eight spaces, or twenty spaces? As well as making patterns optional, regular expressions allow us to make patterns *repeat*, using the \* syntax:

```
$ egrep "^(CS|cs)246 *:" < tasks.txt
CS246: Assignment 2
cs246 : Exam 1
```

This version matches both lines, as before, but would also match a line with more than one space. Like ?, \* repeats a *pattern*, so if we wanted to repeat more than one character, we would need to group them. Note how \* did allow zero spaces, and matched the line "CS246: Assignment 2". If we want to match *one or more*, we can use + instead of \*:

```
$ egrep "^(CS|cs)246 +:" < tasks.txt
cs246 : Exam 1
```

Now, let's combine egrep with other tools to perform more interesting tasks. For instance, let's ask how many CS246 tasks we have:

```
$ egrep "^(CS|cs)246 *:" < tasks.txt | wc -l
2
```

Here, egrep is just outputting the same two lines as before, but we're piping that output to wc, which tells us that there are two lines. Since we've written one task per line, that means we have two tasks. If we wanted to see only the first CS246 task, we could combine egrep with head:

```
$ egrep "^(CS|cs)246 *:" < tasks.txt | head -1
CS246: Assignment 2
```

The beauty of pipes is that we can combine any programs to accomplish the task we need accomplished.

# Pattern reference

In the last sub-module, we looked at some advanced patterns for egrep. We haven't yet seen all patterns supported by egrep, though. In this module, we provide a reference to all patterns, as well as examples of their use.

- The basic pattern is all normal characters, and it matches the named character. For instance, the pattern e matches the character "e".
- (XYZ) groups patterns, usually in order to use |, \*, or + (below) on a whole group. For instance, (CS246) matches only "CS246", but (CS246)\* matches "CS246", "" (the empty string), "CS246CS246", etc.
- (X|Y) matches either X or Y, and X or Y may themselves be more sophisticated patterns.
- [...] matches any of the *characters* between the square brackets. For instance, [cC][sS] is equivalent to (c|C)(s|S).
- [...] can also match *ranges* of characters, specified like a-z. For instance, [a-zA-Z] matches all lower-case letters and numbers, and [a-df-zA-Z] matches all lower-case letters except for "e".
- [^...] is an inversion of [...]. For instance, [^a-zA-Z] matches anything *other* than a lower-case character. Be careful, though: It does need to match *something*. For example, CS[^3]46 matches "CS246", and matches "CS446", but does not match "CS46".
- \* matches repetition of a pattern, 0 or more times. For instance, a\* matches "a", "" (the empty string), "aaaaaaaa", etc. Be careful: \* repeats the *pattern*, not the matched string, so (C|c)\* matches "cCCCCccCccccCCC", and any other combination of upper- and lower-case "c"s.
- + matches repetition of a pattern, 1 or more times. For instance, a+ matches "a" and "aaaaaa", but does not match "" (the empty string).
- ? matches repetition of a pattern, 0 or 1 time. For instance, a? matches either "" (the empty string) or "a".
- . matches any single character. This is commonly used to combine patterns, when two things should appear somewhere on the same line, but with anything at all separating them. For example, to match a line that

includes both "CS246" and "ANTH221", we could write (CS246.\*ANTH221|ANTH221.\*CS246). Note the combination of . with \* to match any number of any character.

- The special patterns ^ and \$ match the beginning and end of a line, respectively. For instance, ^CS246 matches lines which start with "CS246", and ANTH221\$ matches lines that end with "ANTH221". ^CS246\$ matches only the exact line "CS246"; if we want to match lines that either begin or end with "CS246", we can use (^CS246|CS246\$).

If we want to match one of these special character, such as (, ), [, or ], we need to precede it with a backslash. For instance, if we want to match "(CS246)", we would use \((CS246\), not (CS246). Be careful about backslashes though: The shell itself handles backslashes if they're inside of double quotes. In this case, you would want to use single quotes:

```
$ egrep '\(CS246\)' < tasks.txt
```

Let's put these new patterns together to search our tasks for assignments in any 200-level CS course. We'll start with the pattern we had before:

```
$ egrep "^(CS|cs)246 *:" < tasks.txt
CS246: Assignment 2
cs246 : Exam 1
```

We want any 200-level course. The course number for a 200-level course will always be a "2" followed by two more numbers. So, we can change the "4" and "6" patterns to match any numbers:

```
$ egrep "^(CS|cs)2[0-9][0-9] *:" < tasks.txt
CS241: Finish assignment 1
CS246: Assignment 2
cs246 : Exam 1
```

Now, let's tackle only matching assignments. Unfortunately, we weren't consistent in how we wrote our tasks: "assignment" can appear anywhere on the line, and may or may not be capitalized. The \* pattern can be used to look anywhere on the line later than the colon:

```
$ egrep "^(CS|cs)2[0-9][0-9] *.*assignment" < tasks.txt
CS241: Finish assignment 1
```

But, now it doesn't support capitalization, so it missed "CS246: Assignment 2". We have a number of ways to support either an upper- or lower-case "a", so let's use one we haven't used yet:

```
$ egrep "^(CS|cs)2[0-9][0-9] *.*[Aa]ssignment" < tasks.txt
CS241: Finish assignment 1
CS246: Assignment 2
```

As you can see, regular expressions are extremely powerful, but can also get extremely confusing! The pattern `^(CS|cs)2[0-9][0-9] *.*[Aa]ssignment` is complex. Like many things in computer science, the trick to reading and writing regular expressions is learning to break down the problem into smaller, independent parts, and becoming good at that simply requires practice. Looking through the manual page for egrep will also show you other useful shortcuts, such as **named character classes**, that may help you simplify your regular expressions.

# bash, egrep and quotation marks

Given that you are just starting to learn bash, we would recommend that you put single quotes around your egrep patterns unless you want bash to perform substitutions such as replacing variables with their contents, since you are not familiar with all the special characters that bash uses.

Note that the single and double quotes apply to bash processing of the argument only. It does not affect how egrep interprets the pattern. It is important to remember that bash and egrep each do their own *separate* processing on the arguments that you pass in.

A very good way to show this is to ask you to look for lines containing \$var. If you tried using:

```
egrep $var file.txt
```

it wouldn't work because bash tries to replace \$var with the corresponding variable value. If the variable var is not defined, then it will replace it with an empty string. You may be tempted to then try:

```
egrep '$var' file.txt
```

which prevents bash from replacing \$var with the value of the variable (or an empty string). But you will then be unable to find anything at all, because egrep will see the pattern \$var, and interpret this as looking for the string var after the end of the line (since \$ means end-of-line to egrep). The correct solution is:

```
egrep '\$var' file.txt
```

where the single quotes prevent bash from replace \$var with the value of the variable (or an empty string), and the backslash ('\') tells egrep to ignore the special meaning of \$. Note that the quotes are there to affect bash's behaviour, and the backslash is there to affect egrep's behaviour.

What about the backslash character and bash? Backslashes work fine in double quotes... sort of. But the shell only uses backslash as an escape when the next character is something the shell cares about. Compare the following commands and their output:

```
$ echo "\hi"  
\hi  
$ echo "\\\"  
\  
$ echo "\\hi"  
\hi  
$ echo "$PATH"  
/usr/bin:/bin  
$ echo "\$PATH"  
$PATH
```

On the other hand, in single quotes, the shell never expands variables or anything like that, so backslashes are treated literally:

```
$ echo '\hi'  
\hi  
$ echo '\\\'  
\\  
$ echo '\\hi'  
\\\hi  
$ echo '$PATH'  
$PATH  
$ echo '\$PATH'  
\$PATH
```

# Creating a script file

bash isn't just an interpreter of commands for the operating system. It also has a simple programming language. By putting a sequence of shell commands into a text file, you can now repeat actions. By convention, though it isn't necessary, you can use ".sh" for the file name suffix. For example, let me create the file `basic.sh` (also available in your course git repository) that contains the commands:

```
#!/bin/bash  
date  
whoami  
pwd
```

The very first line of the file isn't required, but it's a good idea! It's called a **shebang** line (less commonly known as "shabang" or "sh-bang") because it starts with the musical sharp symbol ('#') followed by an exclamation point ('!'), which some people call "bang". This line tells the executing shell which shell language the script is written in, since it could be executed in different environments with a different default shell.

**The shebang line can be modified by putting the argument "-x" on it. This tells bash to produce verbose output, showing all of its substitutions. This is a very helpful way for you to see exactly what your script is doing, and will be very useful for assignment 1. For example:**

```
#!/bin/bash -x  
date  
echo $PATH
```

I can run the program by telling bash to execute it: `bash basic.sh`

0:00 / 0:40



A horizontal progress bar consisting of a thin black line with a thicker grey segment indicating the current playback position.

But specifying bash in front of everything is tedious. It'd be a lot nicer if I could just treat it like any other Linux tool.  
What happens if I try to run it like any other Linux tool?

0:00 / 0:36



A horizontal progress bar consisting of a thin black line with a thicker grey segment indicating the current playback position.

The problem is that the directory `basic.sh` is located in isn't part of the normal bash search path of where it looks for things to execute. How do you know where bash searches? The list of directories bash searches is contained in the system variable `PATH`, so you need to display the contents of the variable. Use the command: `echo $PATH`

You therefore have 3 choices. Put the file somewhere bash already looks (by common convention, your `bin` directory is such a place, though you may need to modify your `PATH` variable to include it), modify the definition of your `PATH` variable to include your current directory, or the easiest option, which is to tell bash where to find your script file.

Information on how to modify your `PATH` variable can be found at <https://opensource.com/article/17/6/set-path-linux>.

Can you use an *absolute path* to the script `basic.sh` in order to tell bash to run it?

- Yes
- No

Yes, that's correct.

The file `basic.sh` is located in `/u/your-userid/cs246/your-term/lectures/bash/scripts/`. So, for example, I might try to run it as `/u/ctkierstead/cs246/1205/lectures/bash/scripts/basic.sh`. If I try to run it as `~/cs246/1205/lectures/bash/scripts/basic.sh`, is this an *absolute* or a *relative* path to the file?

- absolute
- relative

Yes, that's correct.

If you are currently in the CS246 git repository, in the sub-directory `lectures/bash/scripts`, and you want to run the `basic.sh` script, what would be the shortest *relative path*?

- `~/cs246/your-term/lectures/bash/scripts/basic.sh` where *your-term* is replaced by the appropriate value
- `/u/your-userid/cs246/your-term/lectures/bash/scripts/basic.sh` where *your-userid* and *your-term* are replaced by the appropriate value for you
- `lectures/bash/scripts/basic.sh`
- `./basic.sh`
- `../basic.sh`

Yes, that's great! This is indeed the shortest relative path.

# Running a script file

What happens if you try to use the relative path syntax to run `basic.sh`? (You should see something like the following video clip.)

0:00 / 0:47



So, the general rules for setting up a script file are:

1. Insert a shebang line as your first line.
2. Always add user-level execute permission.

# bash as a programming language

bash also has a simple programming language syntax available. This lets you write more complicated scripts. We'll walk you through the basics in the next few modules. You can also look through the manual pages for more details (see `man bash`).

This also means that it may be hard for you to see why your script isn't working as intended. One thing you can do to help is change your shebang line to look like:

```
#!/bin/bash -x
```

The `-x` option tells bash to print every command and its arguments as it executes. This will hopefully let you see where the problem is coming from so that you can correct it. Once you're sure that everything is working properly, **remove** the `-x` from your shebang line.

# Variables

A bash variable name should follow similar naming conventions to C variable names i.e. start with a letter or underscore, and consist of letters, digits and underscores. Be careful, though, to not use reserved words in the language or hyphens. (There's a good list of potential problems at

[https://www.linuxtopia.org/online\\_books/advanced\\_bash\\_scripting\\_guide/gotchas.html](https://www.linuxtopia.org/online_books/advanced_bash_scripting_guide/gotchas.html).)

When a variable name appears on the left-hand side of an assignment, it must never start with a '\$'. The '\$' is used to retrieve values from a variable, and thus should only appear on the right-hand side of an assignment, or in an expression being evaluated. For example:

```
x=1  
echo $x
```

could be a simple bash script that defines the variable named "x". Note that, by default, the "1" assigned to x is a string, and not a number! The whitespace (or lack thereof) on either side of the assignment operator '=' is also crucial. It's an error otherwise.

0:00 / 1:07



A horizontal progress bar consisting of a thin black line with a thicker grey segment indicating the current playback position.

It's also a good idea to use the curly braces "{}" to surround your variable name when you're retrieving the value. It can help prevent unexpected errors. For example:

0:00 / 0:44



A horizontal progress bar consisting of a thin black line with a thicker grey segment indicating the current playback position.

Remember that you also have a number of global variables available to you. You can use the env command to see many of them.

0:00 / 1:01



You also need to remember the shell expansion rules, since they apply to your variables as well.

0:00 / 1:27



A horizontal progress bar consisting of a thin black line with a thicker white segment indicating the current time.

**Be *extremely* careful about which sort of quotation marks you use!**

# Special variables

There are a number of special bash variables that will be useful for your scripts. They are described in more detail in the bash manual pages, but here's a short list of the most important ones:

- \$0 the first word of the command-line and its arguments; i.e., the shell-script name
- \$1, \$2, etc. each command-line argument, based upon its position (\$1 = the first argument, etc.)
- \$# the total number of command-line arguments
- \$? the exit/return value of the most-recently executed command; used to tell us whether it succeeded or failed. 0 means success, while not-0 means failure

## Exercise: Is it a word?

For example, let's say that I want to write a small bash script called `isItAWord` that takes a single word as its command-line argument, and checks to see if it is a word in the dictionary. If the word is in the dictionary, it will print the word; otherwise, it prints nothing. So, if I run it as:

```
./isItAWord dog
```

I expect to see the word "dog" printed, but if I run it as:

```
./isItAWord xxx
```

I should see nothing printed.

The first thing we need is a dictionary. It turns out that in the student environment, there is a text file that consists of a list of words in the English language. The file is called `words`, and it is located in the directory `/usr/share/dict`. So, we'll use that to decide if our command-line argument is a word in the English language or not.

How do we retrieve the first command-line argument to our script, what would be either the word "dog" or "xxx" from our examples?

Now, what tools have you seen so far that will let you search a text file for a word?

If I take these three pieces and combine them, we get the following implementation of our `isItAWord` script:

```
#!/bin/bash  
  
egrep "^$1$" /usr/share/dict/words
```

Can you explain why we surrounded \$1 with '^' and '\$'?

You can find this script in the CS246 git repository, under `lectures/shell/scripts`. Try running it.

# Selection statements

The simplest form of selection statement in bash is the `if` statement. (There is also a `case` statement, which is described at [https://tldp.org/LDP/Bash-Beginners-Guide/html/sect\\_07\\_03.html](https://tldp.org/LDP/Bash-Beginners-Guide/html/sect_07_03.html).) For most of what you'll be doing, the `if` is sufficient.

## General structure

An `if` statement must follow one of the following two forms:

```
if condition1 ; then
    statement1
    statement2
elif condition2 ; then
    statement3
...
else
    statement4
    statement5
fi
```

```
if condition1
then
    statement1
    statement2
elif condition2
then
    statement3
...
else
    statement4
    statement5
fi
```

Note that if you want the keyword `then` to be on the same line as the `if/elif`, you must use the semicolon to separate it from the condition.

## Specifying conditions

A condition is generally specified within a pair of square brackets, "[ ]". Note that bash is very picky about the spacing in that there must be a blank space between the opening square bracket and the condition being tested, as well as a blank space between the condition and the closing square bracket. For example, if I wanted to test that a file named "foo.txt" existed before outputting its contents, I would use the bash command:

```
if [ -e foo.txt ]; then
    cat foo.txt
fi
```

If I wanted to compare two strings, I would use:

```
if [ $foo = "cat" ]; then
    echo found string \"cat\"
fi
```

If foo could contain whitespace such as blanks/spaces/tabs, it would be better to have written the previous example as:

```
if [ "$foo" = "cat" ]; then
    echo found string \"cat\"
fi
```

If I wanted to compare two integers, I would use:

```
if [ $foo -eq 2 ]; then
    echo equals 2
else
    echo not equal 2
fi
```

If you want to combine conditions using "and" or "or", use the keywords -a or -o. For more options, see your Linux command sheet, or the manual pages for test or "[" i.e. `man test` or `man [`.

# Functions

Before we discuss how to write a function in bash, we need to quickly review how bash processes the command-line. Remember that if I execute a script that takes some command-line arguments and I combine that with input- and output-redirection, as in:

```
./exampleScript.sh fish "dog woof" cat < input.txt > output.txt
```

then bash will have already processed the I/O redirection before it executes the script. I can then use \${0}, \${1}, etc. to do something with the command-line arguments.

Now, how do we declare a function in bash? Well, functions can only return a positive integer value in the range 0 to 255, so we don't need to specify a return-type on the function. Neither do we specify the parameters. Instead, we use the same syntax that the script used to access the command-line arguments, since our script invokes the function in the same way. We specify the contents of the function's body by putting our statements inside curly braces i.e. "{ }". So, my exampleScript.sh script could have a function foo, defined like:

```
foo() {
    echo "name is: " ${0}
    echo "foo argument 1 is: " ${1}
    echo "foo argument 2 is: " ${2}
    echo "foo argument 3 is: " ${3}
    if [ ${1} = "cat" ]; then
        return 0
    fi
    return 1
}
```

and it calls foo by saying:

```
foo ${3} ${2} ${1}
echo 'foo returned ${?}' ${?}
```

If `exampleScript.sh` was executed by the command:

```
./exampleScript.sh fish "dog woof" cat < input.txt > output.txt
```

what is the value returned by `foo`?

0

1

Yes, that's correct.

## Exercise: Good password

Let's take our previous `isItAWord` example and use it to build a script that will check if the given word is a good password or not. (A bad password is one where the word can be found in a dictionary!) Our previous bash script consisted of:

```
#!/bin/bash
egrep "^${1}" /usr/share/dict/words
```

Just printing out the word if it's in the dictionary, and nothing if it isn't, isn't very informative for the user. It'd be a lot better if we provided more informative output. So, we're going to first throw away the egrep output by redirecting the standard output to `/dev/null` (this is the Linux equivalent of throwing the standard output into the garbage). Our egrep command now becomes:

```
#!/bin/bash
egrep "^${1}" /usr/share/dict/words > /dev/null
```

We then add on a test for the status code returned by egrep if it found the word. (Always check the manual page to make sure that you're checking the right values!) In this case, 0 indicates success and non-zero indicates failure. Since we're comparing an integer value, and not a string, we'll use `-eq` instead of `=`. Remember that `$?` lets us retrieve the value of the most recently run command. So, our if statement becomes:

```
#!/bin/bash
egrep "^${1}" /usr/share/dict/words > /dev/null
if [ ${?} -eq 0 ]; then
    echo Bad password
else
    echo May be a good password
fi
```

Now, the last piece that we're missing is ensuring that we call our script with a word passed in on the command-line. If we call the script with either 0 or more than 1 command-line argument, we'd like to print a message that explains how to properly invoke the script, and causes the script to terminate with a non-zero value to indicate failure. We'll encapsulate the printing of the usage message into a function. This lets us call it more than once, if necessary, and it's something we can copy into future scripts with only slight changes. We'll use the shell built-in command `exit` to terminate our script with a failure (non-zero) value.

```
usage() {
    echo "Usage: ${0} password"
    exit 1
}

if [ ${#} -ne 1 ]; then
    usage
fi
```

Let's bring all of the pieces together. The final version of the script can be found in your repository under `lectures/shell/scripts` as `goodPasswordUsage`. It looks like:

```
#!/bin/bash
# Answers whether a candidate word might be a good password

usage () {
    echo "Usage: ${0} password" >&2
    exit 1
}

if [ ${#} -ne 1 ]; then
    usage
fi

egrep "^\${1}\$" /usr/share/dict/words > /dev/null

if [ ${?} -eq 0 ]; then
    echo Not a good password
else
```

```
echo May be a good password  
fi
```

Note that, by default, it returns 0 to indicate success if it runs to completion. >&2 is how bash redirects standard output (the default output stream for echo) to standard error.

# Loops

bash provides two main types of loops, a *counted loop*, and a loop that *iterates over a list* of items.

## Counted loop

A counted loop requires a variable to which we add or subtract a value, or modify in some way. The loop terminates when we reach a specific value or some condition evaluates to true. Usual forms of the loop are:

```
while [ cond ]; do  
  ...  
done
```

```
until [ cond ]; do  
  ...  
done
```

```
for (( expr1; expr2; expr3 )); do  
  ...  
done
```

where the last version uses a more C-like syntax.

Examples of use would be:

```
x=1  
while [ ${x} -le 5 ]; do  
  echo ${x}  
  x=$((x + 1))  
done
```

```
x=1  
until [ ${x} -gt 5 ]; do  
  echo ${x}  
  x=$((x + 1))  
done
```

```
for (( x=1; x <= 5; x++ )); do  
  echo ${x}  
done
```

where the special syntax "\$(( ... ))" is used to tell the shell that the information within the doubled-parentheses are to be treated as integers. (Remember, by default bash treats values as being strings!)

You can find a version of this example in your repository as the script count, under lectures/shell/scripts.

## List iteration loop

This is a rather unusual sort of loop. The idea here is that you are looping over a list of (string) values, either written directly into the loop, or generated as the result of executing a command. The general structure of the loop is:

```
for variable in list; do  
    command1  
    command2  
    ...  
done
```

This sort of loop is easier to understand once you see how some examples work. Please see the following video for a few short examples.

0:00 / 2:05



# More bash exercises

Let's walk through a few exercises to practice the concepts we've seen so far.

## Exercise 1

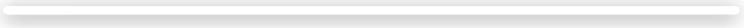
We have a directory that contains a number of C++ files that we've inherited from the previous coop student. Unfortunately, they liked to use the suffix ".cpp" on all of their files, and we much prefer to use ".cc". Since we're planning in incorporating their code into our current project, it would be helpful to rename all of their files to use the suffix ".cc".

What steps must we complete in order to solve this problem?

Well, we first need to create a list of all of the files in the current directory that end in ".cpp". There are several ways that we can do this. The most obvious way is to use the `ls` (list) command to list all files in the directory that end in ".cpp". (You could also have passed the output of `ls` to `egrep` and asked `egrep` for only the strings that ended in ".cpp", but there's no benefit in introducing another tool to the process when one will do the job.)

But, there's an even easier way to do this! You can just use a bash *globbing pattern*.

0:00 / 1:59



We now have a loop that can iterate over all of the files with the ".cpp" suffix in our current directory. We need to change the loop body to rename the file. We know that we can use the `mv` (move) command to rename the file. The first command-line argument to `mv` is the name of the file whose name is to be changed. We can get that from the loop variable. The second command-line argument needs to be the file name to rename the file to i.e. if the file was called `xxx.cpp`, we want it to be called `xxx.cc`. Thus our loop body becomes:

```
for name in *.cpp; do
    mv ${name} do-something-with-${name} ## fix the second command-line argument to mv
done
```

There are several ways to pull apart a file name by removing the suffix. We could use the `basename` command, (see `man basename`) but it turns out that bash has some special syntax for exactly this sort of problem. If I have a variable `${name}`, I can put the string "`%some-pattern`" before the closing curly brace, and that will return the part of the contents of variable name without the string "`some-pattern`". For example, if I have:

```
name="catfish"
echo ${name%fish} # outputs the string "cat"
```

This can be used to create the new file name by removing the suffix "cpp" and adding the suffix "cc" through string concatenation. The loop now becomes:

```
for name in *.cpp; do
    mv ${name} ${name%cpp}cc
done
```

The complete version of this script can be found in your repository as the script file renameCPP, under lectures/shell/scripts.

## Exercise 2

We would like to count the number of times a specified word occurs within the specified file. Both the word and the name of the file will be passed to the script `countWords` as command-line arguments, parameters  `${1}` and  `${2}` respectively. If one or both of the command-line arguments is missing, or if we have more than two command-line arguments, we will print a usage message and return the error code 1.

By this point, you should be able to answer the following questions yourself. If not, please review the previous material.

How do we determine the number of command-line arguments that a script has received?

- `#$`
- `$#`
- `$0`
- `!#`

Yes, that's great!

What is the operator used to test if a variable does not equal the integer value 2?

- `!=`
- `<>`
- `-eq`

-ne

Yes, that's great!

How do I terminate a script with an error code of 1?

- return 1
- exit 1

Yes, that's great!

How do we declare an integer counter, called counter, set to 0 to keep track of the number of matches?

- \$counter=0
- counter = 0
- \$counter=\$0
- counter=0

Yes, that's correct!

How do we write the beginning of a *list iteration* for loop that uses the variable word?

- for word in
- for \$word in

- for word do

Yes, that's correct!

How do we pass in the contents of the file specified by \$2 (not its name!) as the list portion of the *list iteration* for loop?

- 'echo \$2'
- 'cat \$2'
- `cat \$2`
- \$(cat \$2)

Yes, that's correct!

How do we compare two strings, the variable word and the first command-line argument, to see if they are equal?

- \$1 -eq \$word
- \$1 == \$word
- \$1 = \$word
- \$word = \$1
- word = \$1

Yes, that's great! bash treats = as equivalent to ==. When == is used, the string on the right is treated as a pattern and matched according to the pattern matching rules in the bash manual page.

How do we increment the variable counter when we have a match?

- \$counter=\$counter+1
- counter=\$counter+1
- counter=\$((counter+1))
- counter = \$((counter+1))
- counter=\$((\$counter+1))

Yes, that's great!

How do we output the total number of matches at the end?

- echo \$counter
- echo counter
- cat \$counter

Yes, that's great!

Now that you have all the pieces, you should be able to put them together. The result looks like the following:

```

#!/bin/bash
# countWords word file
# Prints the number of times word occurs in file

usage() {
    echo "${0} word file"
    exit 1
}

# Do we have exactly 2 command-line arguments or not?
if [ $# -ne 2 ]; then
    usage
fi

counter=0 # Initialize the counter to 0.

#Iterate over every word in the file, ${2}.
for word in $(cat "${2}"); do
    if [ ${word} == "${1}" ]; then
        counter=$((counter + 1))
    fi
done
echo ${counter}

```

The file can also be found in your repository as the script `countWords`, in the directory `lectures/shell/scripts`. The only thing that may surprise you is the presence of double quotation marks around the command-line argument variables. Putting them around the variables allows the user to specify a string that contains spaces for either the word to be matched or the file name. Note that the former isn't actually that useful since it'll never be matched to any string in the file due to the way the shell reads words; however, it's still a legal string so this prevents potential bash syntax errors. Having a space in the file name isn't common in command-line Linux, but is common everywhere else. This lets us invoke the script as:

```
./countWords cat input.txt
```

or

```
./countWords "peanut butter" "input file.txt"
```

## Exercise 3

As a non-union employee of the University of Waterloo, you are paid monthly on the last Friday of each month. In order to plan your budget, it's important for you to be able to determine the date of the last Friday in any specified month and year. If no month and year are specified, you will assume that the user wants the last Friday of the current month.

The problem can be broken into two main pieces: compute the date, and present the information.

### Compute date

This is where knowing a broad range of Linux tools is very handy. (Or use "man -k calendar" to search the manual page short descriptions and names to see which contain the word "calendar"; you could restrict it further to section 1 of the manual, which covers executable programs and shell commands, by using "man -k calendar -s 1".) The tool that we're interested in using is called `cal`.

Invoked with nothing on the command line, it displays the current month. For example:

```
$ cal
      April 2020
Su Mo Tu We Th Fr Sa
        1  2  3  4
 5  6  7  8  9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29 30
```

Alternately, I can invoke it with the desired month and year to display.

```
$ cal October 2020
October 2020
```

```
Su Mo Tu We Th Fr Sa
      1  2  3
 4  5  6  7  8  9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30 31
```

All we need to do is figure out how to extract the Friday column from the output of `cal`, and then find the last line that contains a number in that column. It's tempting to use the `cut` tool to do it, but the `cal` output uses backspace characters ('\b') as part of the formatting (visible if you use `cal | od -c`), so it won't work reliably. Instead, we're going to use `awk`. `awk` is a scripting language developed by Aho, Weinberger and Kernighan. It's extremely good at pattern-matching and processing records. I just need to tell `awk` to output the 6th column of whatever it received from standard input. I can do this by piping the standard output from `cal` to the standard input of `awk`, using the command:

```
$ cal | awk '{print $6}'
```

```
Fr
```

```
10
17
24
```

I can then use `egrep` to find only the lines that contain a digit:

```
$ cal | awk '{print $6}' | egrep "[0-9]"
10
17
24
```

And then restrict the output to the last line using `tail`:

```
$ cal | awk '{print $6}' | egrep "[0-9]" | tail -1
24
```

## Print information

In order to produce a more human-readable output, I'd like to take the result from the previous part, and check to see what day it is. I know it must be in the range of 22 to 31. If the day isn't 22, 23, or 31, then I'd like the output for the current month to be:

This month's payday is on the *NN*th.

where *NN* is replaced by something in the range of 24 to 30. If it's a 22, 23, 31, I'd like the *NN*th in the output replaced with 22nd, 23rd, or 31st. For example:

This month's payday is on the 31st.

If the month isn't the current month, then the previous output has the string "This month" replaced by the month name (*MMM*) and year (*YYYY*) that was passed in on the command-line. For example, if the month was October, we would print something like:

October *YYYY*'s payday is on the *NN*th.

Let's write a small bash function called `report` that takes the day as an argument, and possibly a month and year, and produces the appropriate output. Note that the `echo` command has a "`-n`" option that prevents the newline character from being printed. You will find this helpful.

1. Write the `echo` statement that outputs the month, followed by the year and no newline.
2. Write the `echo` statement that outputs the text 's payday is on the *NN*th., where *NN* is replaced by the day that was passed in as the first argument to the function `report`.

## Put the pieces together

We should write the standard usage function in case the script is invoked with an incorrect number of parameters i.e. it should have either 0 or exactly 2. If not, we want to exit with a value of 1 after outputting the usage message to

standard output.

```
usage() {
    echo "$0 [month year]"
    exit 1
}
```

Note that the square brackets ("[ ]") are a common Linux convention used to indicate that the command-line arguments are optional i.e. the script can be invoked either as:

```
./payday
```

or as

```
./payday October 2020
```

We want to invoke `usage` if the number of command-line arguments is neither 0 nor 2, so we'd write our `if` statement to use the *and* operator, `-a`.

```
if [ $# -ne 0 -a $# -ne 2 ]; then
    usage
fi
```

If we get past this test, we can then invoke `report` with the appropriate information. Note that passing in non-existent command-line arguments doesn't do anything, so we can just say:

```
report $(cal $1 $2 | awk '{print $6}' | grep "[0-9]" | tail -1) $1 $2
```

The overall script thus looks like this:

```
#!/bin/bash
# Returns the date of the next payday (last Friday of the month)
# Examples:
# payday (no arguments) -- gives this month's payday
# payday June 2020 -- gives payday in June 2020
```

```

usage () {
    echo "$0 [month year]"
    exit 1
}

report () {
    if [ $# -eq 3 ]; then
        echo -n ${2} ${3}
    else
        echo -n "This month"
    fi
    echo -n "'s payday is on the "
    if [ $1 -eq 31 ]; then
        echo "31st."
    elif [ $1 -eq 22 ]; then
        echo "22nd."
    elif [ $1 -eq 23 ]; then
        echo "23rd."
    else
        echo "${1}th."
    fi
}

if [ $# -ne 0 -a $# -ne 2 ]; then
    usage
fi

report $(cal $1 $2 | awk '{print $6}' | grep "[0-9]" | tail -1) $1 $2

```

You can find a copy of it called `payday` in your repository, under `lectures/shell/scripts`.

# What are Software Tests

**Software testing** is an investigation conducted to verify if the software works as intended, identify existing errors, and possibly assess the quality of the software. It is an essential part of software development. Although we usually think about development as "writing code", properly testing the software is as important as writing it.

## Approaches for Testing

Software testing can be conducted manually by a human tester or automatically by a machine.

### Human testing

In **human testing**, a person manually verifies if the software is working as intended and tries to identify existing errors (bugs). This can be done, for example, by inspecting the code and looking for flaws, or by walking through the software operation as a user and verifying the results.

Ideally, a test plan should be created beforehand, based on the requirement specifications. Therefore, the human tester can follow the plan to ensure that all the needed tests are executed. If a plan is not available, the tester can walk through the requirements and do their best to test all possibilities; however, this may result in specific operations not being tested and thus some existing errors may not be discovered.

Human testing may be adequate in some cases. However, it is difficult to scale human testing for large software, as it can be a time-consuming operation. Therefore, tests can be automated so a large number of tests can be executed more effectively.

### Automated (machine) testing

In **automated testing**, test suites are implemented that automatically test the software and compare the results with the expected ones. In general, **test suites** should contain a list of input sets and matching expected outputs, covering different situations according to the requirement specifications. A test suite may also specify the necessary command-line arguments, expected error messages, and return values.

The automated testing software can then run the test suite, running the program with each input set and checking the actual output against the expected output. Any discrepancy is logged, so a human developer can later identify the errors.

## Testing is not Debugging!

Testing the software is different than debugging it. **Testing** is the process that identifies the errors but does not solve them. **Debugging**, on the other hand, is the process that identifies the causes of known errors to solve them.

Therefore, testing occurs both before and after debugging:

1. A test is conducted and an error (bug) is identified.
2. A developer debugs the code to identify the cause of the error.
3. After identifying the cause, the developer fixes the error and generates a new version of the software.
4. A new test is conducted to ensure that the error was indeed corrected.

# Types of Software Tests

Software can be tested on many levels and for different purposes. We will not cover all the different types of tests here, but these are some of the most common.

## Unit tests

**Unit tests** are conducted at the lowest level, testing only one specific module/unit of the software. The goal of unit tests is to verify if the unit of code works as intended and identify any existing errors (bugs). This can be done, for example, by testing if the functions in a module or the methods in a class work as intended and produce the expected results.

## Integration tests

**Integration tests** are conducted a level above the unit tests and aim to verify if the different modules/units of the software work correctly together. Therefore, these tests should perform operations that involve multiple modules or classes and check if they work as intended.

## Functional tests (or system tests)

**Functional tests** focus on the business requirements of the application. They can be conducted by executing the application with a specific set of inputs and verifying if the application produces the correct outputs. Here, we are not concerned with how the results are being produced (which should have already been tested in the unit and integration tests), only if they are correct.

## Acceptance tests

**Acceptance tests** are usually part of a formal process in which the client must verify that the produced software meets all the requirements, so it can be accepted. Client acceptance is generally a condition to move to the next phase of a project, such as installing the software for real users, generating an invoice for payment, etc.

Sometimes, acceptance tests may be detailed into phases such as **alpha testing** (done at the end of the development, by a subset of users, but in the controlled environment of the developer) and **beta testing** (done at the end of the development after the alpha tests, by a subset of real users, in the user's environment).

## Regression tests

**Regression tests** are conducted after any modification in the software, to ensure that it continues working as intended, i.e., that the modification did not introduce new errors. Ideally, regression tests should be automated, so that they can quickly be executed after each code modification. One possible way to conduct regression tests is to create automated unit, integration, and functional tests, so they can be executed again after each modification.

## Performance tests

**Performance tests** are designed to verify if the run-time performance of the system will be adequate. Some systems may work well when tested by the developer in a single-user environment but may have performance issues when tested, for example, with multiple concurrent users or with large data sets.

## Accessibility, Usability, and User Experience tests

The goal of these tests is to verify the quality of the experience of the user interacting with the system. Software may do what it is supposed to do (i.e., it passes all the tests above), but it may be for example hard to use or learn, or may not support users with disabilities. These tests aim to identify this kind of issue.

We will not study accessibility, usability, and user experience tests in this course. They are studied in CS 349.



# White/Black Box Tests

When creating software tests, we can use a white-box or black-box approach, or combine them both, in what is usually known as a grey-box approach.

## White-box tests

In **white-box tests**, tests are created with knowledge of the internal structure of a program. It is generally used when we need to verify the internal structures or workings of a program, instead of the expected functional requirements.

Although white-box testing can be done at any level, it is more common for unit tests, when the goal is to verify that each function or method is working correctly.

## Black-box tests

In **black-box tests**, tests are created based only on the requirement specifications, without any knowledge of the internal structure of the program (i.e., without access to the source code). Logically, the goal is to verify if the software works according to the specification, without being concerned with how it works.

Although black-box tests can be done at any level, they are more common for functional and acceptance tests, when it is more important to verify that the requirements are satisfied.

## Grey-box tests

**Grey-box testing** is just a mix of the two approaches above. So, for example, tests may be created to verify if the software works according to the functional requirements, but with knowledge of the internal structure of the software. This may help the tester make better-informed decisions about how to test each requirement. For example, grey-box testing may be a good approach for performance tests, as the tester is focusing on the expected outcomes, but having knowledge of the code may be useful to decide how to better check the potential performance issues.

## Combining these approaches

All the three approaches above are useful and it is probably a good idea to combine all of them for better test coverage.

Black-box tests are very important to ensure that the software works as intended. When we create white-box tests only, it is common to test just what is implemented in the code, as these tests are usually created based on the existing code. Therefore, if the code already contains a functional error, i.e., if its implementation works but does not produce the correct results according to the specification, the white-box tests may easily fail to catch this error.

Consider this example:

```
int max(int a, int b) {  
    if (a > b) {  
        return a;  
    } else {  
        return b;  
    }  
}
```

If we create a white-box test based on the function above, we might create the following test cases:

- Inputs: a = 5, b = 10 | Output: 10

- Inputs:  $a = -1, b = -2$  | Output: -1

This may seem a correct test. However, by looking only at the source code, we do not know what the specification was for this program, so we do not know if the code is implemented according to the specification, even if it seems to run properly. What if the specification was:

"Compare the absolute values of the two arguments and return the higher of them."

If this was the specification, the implementation is incorrect. If we create a black-box test based on the specification above, without knowledge of the source code, we might create the following test cases instead:

- Inputs:  $a = 5, b = 10$  | Output: 10
- Inputs:  $a = -1, b = -2$  | Output: 2

Note that these new test cases can identify that the implementation above is not producing the correct output in all situations according to the specification.

**Therefore, when creating test cases, a good practice is: start with black-box tests, then supplement with grey-and white-box tests.**

## Creating black-box tests

To create the black-box tests, you must check the specification and try to identify all the different types of inputs that the program should accept. Then, identify what should be the correct output to each set of inputs according to the specification. There is no proven method to identify all the necessary test cases, but these are some useful guidelines:

- identify the different classes of input, e.g., numeric ranges, positive vs. negative, etc.

- test the boundaries of valid ranges (edge cases), e.g., minimum and maximum values
- test multiple simultaneous boundaries (corner cases)
- test extreme cases (within reason)
- intuition: with time and acquired experience, you can begin to guess at the likely errors

## Creating white-box tests

To create white-box tests, you can examine the code and create enough test cases to ensure that all parts work as intended. Ideally, you should ensure that there are enough white-box test cases so that:

- the tests execute all logical paths through the program
- the tests make sure that every function runs

Depending on what language you are using, there may be test coverage tools available, which run the tests and produce a report showing if all lines of code were run at least once during the tests or if there are any lines of code that were not covered in any test.

# Best Practices and Considerations

Now that you know what are software tests and how to create them, these are some essential best practices and considerations, which we suggest you keep in mind while creating your tests.

## Testing is an ongoing operation.

Developers often think about testing only at the end of development, after they have already implemented the code. This is not recommended. Testing should always begin *before* you start coding, by creating test plans and implementing automated tests. Then, they continue *while* you are coding, as new situations appear and you can now create white-box tests. They also continue throughout the whole existence of the software: as features are added, removed, or modified, tests must be created and updated, and regression tests must be executed to ensure that new errors were not introduced to existing code.

## Testing cannot guarantee correctness.

If the tests fail, you can prove wrongness in the software. However, if all tests pass, it is unfortunately not a guarantee that the software is correct. The tests themselves may also be incorrect, or there may be specific cases that were not covered by any test.

## Testing is not easy.

Unfortunately, there is no general formula to create good tests. But the guidelines we provide in this module should help you create effective tests for the assignments in this course. Sometimes, there may be a psychological barrier, for example, you may not want to find out that your program is wrong. This may be particularly true when the tester is not the same person as the developer, i.e., the developer may not enjoy having someone telling that there is

something wrong with the software they wrote. However, testing is a necessity, as errors introduced in the software will need to eventually be identified and fixed. It is better if they are identified during tests than in a real user situation.

**Ideally, tester and developer should be different people.**

When the developer is also the tester, it is very easy to miss existing errors because the developer knows how the software should be used according to the code they wrote. When a different person is testing, they may try to use the software in ways that were not predicted by the developer, thus finding errors that the developer could easily miss.

## Testing in this Course

Learning how to properly test your code and forming the habit of doing so is one of the goals for this course. Therefore, in the assignments, we will often require you to create your own test suites to ensure that your program works according to the specification. In A1, we will ask you to write your own set of scripts to automate testing for the following assignments. For A2 and forward, we will require that you first analyze the specification and create a test suite to submit on the first due date. Then, you will write the code to implement the specification and use the test suite you created to verify that it is working as intended.

We will also be creating our own test suites for the assignments, which will be run automatically by Marmoset to verify that your submission works as required. We will not tell you in advance what our tests will be. However, if you do a good job for the first due date and create a good test suite, there are good chances that your own tests will cover most of the same cases as our tests. Thus, if you create good tests and ensure that your program passes those tests before submitting the code to Marmoset, you can ensure good grades in the assignments.

**Consider creating unit and integration tests in addition to the required functional/regression tests.**

The required tests in the assignment specifications will usually be functional/regression tests, in which you will run a complete version of the program with a specific set of inputs and check if the output is correct. But when you are writing a large program, you should be using unit tests from the start to verify if each small part is working. It may be overwhelming if you wait to start testing after you wrote the whole program and you are suddenly flooded with dozens of errors.

Thus, even though we will not require you to write unit and integration tests, it is in your best interest to do so. The initial time required to create those tests may save a lot of time later as they will help you identify errors early instead of having to deal with a large number of errors on the finished program.

We won't specify how the unit and integration tests should be written and, if the assignment specification does not require it, you won't need to submit your unit tests. But ideally, you should create them anyway. For example, your unit tests may be additional files with a main function that creates an object of the class you want to test (each unit test should be testing only one class at a time) and calls specific methods of the class to check if they are working properly. Once you have tested each class using unit tests, then you can move to test the whole finished program using a test suite as we will require in the assignment specifications.

**In this course, you will be the developer and tester.**

We know, we just said on the previous page that developer and tester should ideally be different people. However, this would be very impractical for this course as each student must be working on their own assignment solutions. So, you will be the developer and tester for your assignments. But remember that this would not be ideal in a real-world situation and ideally you should be able to ask someone else to test the code that you implement.

**In this course, you do not need to test invalid input, unless a behaviour has been prescribed.**

For example, if the assignment specification says that the input must be a number, and it never mentions what should happen if the user enters text instead of a number, then the program behaviour is undefined for text input. We will not test undefined behaviour, so, you do not need to test it either. In this case, all your test cases can just input numbers, you do not need to have any test case with text input.

In a situation like this, we usually receive some questions on Piazza asking (1) what the program should do for text input and/or (2) if it needs to be tested. So, these are the general answers, which you can keep in mind for the whole term: (1) it is undefined, so your program really does not have to deal with that type of input and (2) no, you never need to test undefined program behaviour in this course.

Keep in mind that this is a simplification for practical reasons, to reduce the number of test cases that you need to create when working in the assignments. In a real situation outside of this course, it would probably be necessary to

test with invalid input as well, to ensure that the program does not crash and does something nice, such as printing a message explaining what the input should be.

## Testing Beyond this Course

As explained above, we will explicitly ask you to create and submit test cases for your assignment and you will earn marks for doing so. This is because we want you to learn how to create good tests for your programs. After you pass this course and move forward to the next ones, the instructors will probably just assume that you already know how to test your code, so they will not explicitly ask you to create and submit test cases. But even if it is not an explicit requirement, it is still in your best interest to continue properly testing your code for any of your future courses. Good tests will ensure that your program works as intended and you need your assignment solutions to work as intended so you can earn good marks, so creating good tests will only benefit you. We hope that by working on the assignments for this course you will form the habit of creating good tests before and while you are coding and that you will continue working with this habit even when it is not explicitly requested.

The same considerations are valid for any job that you may have as a software developer. Large teams with well-defined development processes usually have testing as a required step and should ideally enforce that testing and development are done by different people. However, smaller teams (or even large teams with informal development processes) may focus more on having working software at the end than on writing tests. But as you will hopefully have learned by the end of this course, creating good tests is the best way to ensure that you are producing good working software. So, having the habit of creating and conducting tests as part of the development process should always be in your best interest, even when this practice is not being explicitly requested by your boss or client.

# Introduction to C++

You might be wondering why we spend time teaching C++ as an object-oriented programming (OOP) language when there are far newer OOP languages now in existence. C++ has wide-spread acceptance and is continually evolving. It has been extended to contain the majority of the newest features (and is still changing!), while still providing backwards compatibility with C.

We are thus able to program both at a very high level of abstraction, as well as work right down at the machine level if necessary.

Since the required background before taking this course is CS136, we are going to assume a basic proficiency in C from you. The only time we'll discuss C is when we compare how C++ and C handle some particular issue. In particular, most of the C syntax you learned from CS136 (selection, looping, variables, pointers, the struct keyword) also apply to C++.

If you need a review of C, please see `lectures/c++/c-review-slides.pdf`, or one of the following links:

- [https://www.tutorialspoint.com/cprogramming/c\\_quick\\_guide.htm](https://www.tutorialspoint.com/cprogramming/c_quick_guide.htm)
- CS136 handouts, <https://www.student.cs.uwaterloo.ca/~cs136/current/handouts/>

## Good resources

There are many C++ resources available electronically. The course web page has links to some good reference books (available for free electronically through the UW library system, though you will probably need to use a [VPN](#) to access them) and sites under the "Resources" section, which have been duplicated here:

- [C++ in a Nutshell](#) - eBook

- [Practical C++ Programming](#) - eBook
- [C++ Primer Plus](#) - eBook
- [Effective Modern C++](#) - eBook
- [C++ Standard Library](#) - eBook
- [C++ reference](#) and [cplusplus.com](#) - C++ reference
- [FAQ](#) - frequently asked questions regarding C++ (**very useful**)

I would particularly recommend bookmarking the two C++ reference sites, [en.cppreference.com](#), and [cplusplus.com](#). The C++ FAQ site has a lot of interesting material, as well as a discussion on how to go about learning C++ if you know other programming languages (C, C#, Java, or Objective-C). Be cautious of using the information found on Stack Overflow ([stackoverflow.com](#))—not all of it is good.

## C++ standards

Since C++ has evolved over the years, it is sometimes important to know which particular [standard](#) a C++ feature belongs to, since you may be working with a program that was developed under a particular standard. If we specify "C++11", this means that we are referring to the C++ 2011 standard, whereas "C++14" specifies the 2014 C++ standard.

In this course, we will be using the C++14 standard. Most of the things you will learn here should also work with newer standards. However, we won't be discussing the newer standards here.

# Basics through "Hello world!"

The simplest possible program we can write is one that outputs the text "Hello world!" to the standard output stream. You should already be familiar with how this is done in C. Let's compare how the equivalent program would be written in C++.

| C   | C++  |
|---|--|
| #include <stdio.h><br><br>int main() {<br>printf("Hello world!\n");<br>return 0;<br>} | #include <iostream><br><br>int main() {<br>std::cout << "Hello world!" << std::endl;<br>return 0;<br>} |

Notice that we have to include a different library in C++, called `iostream`. [The library name doesn't end in ".h"](#) since that's used to show that the library is a C library, and `iostream` is a C++ library.

C++ is mostly backwards compatible with C. That is, most C programs are valid C++ programs. But, C++ adds many features, and has C++-specific versions of most things in C.

**Note that if there's a C++ way to do things, in this course we want you to exclusively use it, rather than the C way of doing things!**

In C, `printf` always writes to the standard output stream. In C++, we have to specify the name of the standard output stream. The full name of the standard output stream is `std::cout`. ([Pronounced "cee-out".](#)) The "std:::" specifies that `cout` is defined in the *standard (std) namespace*. The same is also true for `std::endl`, which combines the C newline character ('\n') with a request to "flush" the buffer. For efficiency reasons, input and output are

buffered and thus there's no guarantee that it will be written or read immediately. Flushing attempts to speed this up. Note that if you leave off the "std:::" from std::cout, the compiler will complain that "cout" was not declared in this scope: declarations are specific to their namespace, so the compiler can't find a matching declaration. By default, it will not look in the std namespace, so you must explicitly specify std::cout.

Since the main routine **must** be declared in C++ to return an integer, the usual convention is to have it return 0 when the program ran successfully, and non-zero if the program didn't run successfully. Just like bash! Note that if you don't write the "return 0;" line, the compiler will automatically insert it for you.

Writing std:: in front of everything gets tedious very quickly, so there are two other ways to handle this:

| Version 1  | Version 2   |
|--|---|
| <pre>#include &lt;iostream&gt; using std::cout; using std::endl;  int main() {     cout &lt;&lt; "Hello world!" &lt;&lt; endl;     return 0; }</pre> | <pre>#include &lt;iostream&gt; using namespace std;  int main() {     cout &lt;&lt; "Hello world!" &lt;&lt; endl;     return 0; }</pre> |

Note that while the second version requires less typing on your part, there are good reasons [not to do it](#). We'll revisit this when we talk about the [preprocessor](#) in more detail.

By convention, the file names end in one of .cc, .cpp, or .C. We will use .cc in this course. You can find the previous example as the file hello.cc in your repository, under cs246/1205/lectures/c++/01-intro.

# Compilation

Unlike a bash script, you can't run a C or C++ program directly. Instead, you first have to use a compiler to create an executable file.

The compiler takes a high-level programming language file (or files), and produces machine-readable instructions by first preprocessing (we will explain what preprocessing is [later](#)), then compiling (transforming C++ code into Assembler code), assembling (transforming Assembler code into machine code), and linking (combining several pieces of machine code—called object files—into a single executable file). You will see this in more detail if you take CS241 or CS230.

The compiler we will be using is the GCC compiler, where `g++` is a "front-end" to `gcc` that specifies that we're programming in C++ and not in C. Be aware that unless you are programming on precisely the same operating system and architecture with precisely the same library versions, a program compiled on one machine will **NOT** work on another. In fact, not all compilers are implemented the same either. For example, the Mac operating system now uses `clang` as the default C++ compiler, and it's not exactly the same as the `g++` compiler in the CS student environment! So a program that compiles on one will not necessarily compile or run on the other. This is why we tell you to do your final testing in the student environment, since that's where we'll be compiling and running your submitted work.

So how do you create an executable file? You can use a command such as:

```
g++ hello.cc
```

and it will create an executable named `a.out` by default. For certain parts of the course, we're going to want to insist that the compiler follow the C++14 standard (at a minimum). You specify this by using the command:

```
g++ -std=c++14 hello.cc
```

Since you'll be using this a lot, your assignment 0 asks you to set up an alias command for it, g++14. Once you've completed A0, and have logged back in, you can just use the command:

```
g++14 hello.cc
```

Now, a .out isn't a very good name for an executable since all of your programs when compiled will default to that name. It'd be a lot nicer if we used a more meaningful name, such as hello. We can do that by using the command:

```
g++14 hello.cc -o hello
```

0:00 / 1:44



There are other options you can pass into the compiler's command-line. The ones you'll find most useful throughout the course are: -g, -c, -Wall, -Wextra, and -D. There are more that you can use—see the manual page for gcc for more information.

|    |  |
|----|--|
| -g | Produce debugging information in the operating system's native format. The GNU debugger, gdb, can work with this debugging information. Makes your code larger, but useful while trying to get everything working. |
| -c | Produce an object file (ends in .o) that consists of assembler output. Will be useful once we discuss separate compilation.  |
|    |  |

|            |   |
|------------|---|
| -Wall      | This enables all the warnings about constructions that some users consider questionable, and that are easy to avoid (or modify to prevent the warning). Not necessary, but a good idea. |
| -Wextra    | This enables some extra warnings that aren't enabled by -Wall. Not necessary, but a good idea.  |
| -Wpedantic | This ensures that your code follows the strict ISO C++ standard, with no forbidden extensions. Not necessary, but a good idea.  |
| -D         | Lets us define a macro name as a command-line argument to the compiler. Useful for selectively adding/removing code during the compilation process.                                     |

We strongly encourage you to let the compiler help you find mistakes wherever possible!

It is also important that you spend some time learning to use a debugger such as gdb. The time you spend learning it now can save you hours of wasted time trying to track down errors in the future. Plus there are certain errors that using print statements won't catch, and you may be looking in the wrong area if your buffered printing wasn't flushed before the error occurred (and using endl/flush everywhere doesn't guarantee that the buffer is flushed!).

# Input and output

There are three global variables in the standard (std) **namespace** that define the stream objects used for basic input and output. Just as in bash, there are two for output, and one for input:

| C++  | C equivalent | Description                |
|------|--------------|----------------------------|
| cin  | stdin        | Reads from standard input. |
| cout | stdout       | Writes to standard output. |
| cerr | stderr       | Writes to standard error.  |

Output streams use the `<<` operator to send output to the stream. The stream name must be on the left-hand side of the operator, while the information to output must be on the right-hand side.

Input streams use the `>>` operator to read input from the stream. The stream name must be on the left-hand side of the operator, while the variable that will contain the information read must be on the right-hand side.

An easy way to remember this is that the direction of the angle brackets tells you which way the information flows, either *from the stream to the variable*, or *from the variable to the stream*.

In order to be able to use the streams, you must include the `iostream` library.

Let's take a look at a simple example, add .cc from `lectures/c++/02-io`. In particular, pay attention to how the input operator handles "whitespace" (blanks, tabs and newlines) between integer values.

0:00 / 0:46



A horizontal progress bar consisting of a thin black line with a small white square marker positioned near the left end.

What happens if the user doesn't provide a proper integer value?

0:00 / 1:28



A horizontal progress bar consisting of a thin black line with a small white square marker positioned near the left end.

The standard input stream object, `cin`, has several bits that it uses to track if an error has occurred, or if the end of the file has been detected. These bits are not directly accessible—instead, you need to use the methods `fail()` or `eof()`, as in:

```
if ( cin.fail() ) { ... }
```

or

```
if ( cin.eof() ) { ... }
```

**Note that you must attempt to read before you test for end-of-file.**

Let's take a look at the example `readInts.cc` from `lectures/c++/02-io`.

0:00 / 1:35



# Handling input errors

As we saw in the previous `readInts.cc` program, our program isn't terribly robust, and in fact silently fails when it reads a non-integer value. This is terrible from a user perspective, since they have no idea why the program suddenly stopped! It would be much more informative to gracefully handle the error, informing the user of the problem, and still let the program continue.

## Version 2

We're going to start by rewriting `readInts.cc` into `readInts2.cc` (also available in your repository), where the form of the `if` statement test has now been changed to apply the operator `!` to the standard input stream object, `cin`.

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

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

In C++, writing:

```
if ( !cin ) ...
```

is effectively writing:

```
if ( cin.operator!() ) ...
```

where `cin`'s `operator!` has been defined to return the contents of the fail bit by calling `fail()`. This ability to specialize operators is called **operator overloading**, and will be discussed in greater detail in [Overloading](#). In this code, it's just a simpler way of writing `cin.fail()`.

## Version 3

The next version relies upon us knowing some details about the C++ version of the `>>` and `<<` operators (written as `operator>>` or `operator<<` when we're writing the function signatures). In C, `<<` and `>>` are used to shift bits either to the left or to the right by some amount. In C++, the operators `<<` and `>>` are **overloaded** for output and input when the object on the left-hand side of the operator are output and input streams, respectively. It turns out that the first parameter for the input stream operator is an input stream i.e. if my program has the line

```
cin >> i;
```

where `i` is an integer, this is executing the command:

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

What does `operator>>` have as a return type? It needs to be able to return `cin`, and `cin` needs to be read from multiple times so that I can write commands such as:

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

This is functionally equivalent to:

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

In other words, `operator>>` takes the input stream `cin` to read in the variable `x`. It returns the input stream `cin`, modified by the call, and passes it as the first parameter to `operator>>` for the call to read in `y`. It then takes the

modified input stream returned by the call to read in `y` and then uses that to read in `z`. The final modified value of `cin` returned by the call to read in `z` is ignored.

`cin` is of type `std::istream`, so `operator>>` needs to return an `std::istream`; but in order to be able to modify it, the return type is actually `(std::istream&)`, not `(std::istream)`. And so is the parameter to `operator>>`. This type is called a **reference variable**, and we'll be discussing them in [more detail shortly!](#) So, the **signature** of the function `operator>>` that reads in an integer is:

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

This lets us modify our program to now look like `readInts3.cc`:

```
#include <iostream>

using namespace std;

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

(The parentheses are there to clarify the order of operations.)

## Version 4

The next version takes advantage of the fact as of C++11 that an object of type `std::istream` can be implicitly converted (we say that it is "coerced") to a Boolean type (`bool` in C++). In particular, the function is defined to return `(! fail())` i.e. returns true if the stream has no errors and hasn't reached the end of the file. This lets us move the read-and-test into the loop condition test from the loop body.

This lets us modify our program to now look like `readInts4.cc`:

```
#include <iostream>

using namespace std;

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

## Version 5

We will now modify our program to skip all non-integer input, so that we can read all integers and echo them to standard output until the end-of-file signal is detected. Remember that we read in a non-integer character, this sets the fail bit in our input stream object and it stops reading the input stream. For example, if we'd already read in -123, 45, 23, and 1 as in the example below, we would stop upon seeing the character 'x' since it's not an integer.

`_ = space, \t = tab, \n = newline`

`-123_045\t23\n__1xyz_45... EOF`



This tells us that dealing with the error is going to be a two-part process:

1. We will first need to "clear" the fail bit through `cin.clear()`; otherwise, it remains set and our program will still think it's in an error state.

**2. Since the program stopped reading on a non-integer value (any character that cannot be thought of as part of integer, such as the character 'x' in the diagram), we have to "throw away" the offending character through `cin.ignore()`; however, we have no idea how many characters may be part of the problem, so we'll only throw away one character at a time.**

If we do this, we'll be in a "valid" state the next time we try to read an integer.

```
-123_045\t23\n_1xyz_45... EOF
```

In the case of the previous input example, we would immediately stop upon seeing the 'y' character, so we would need to repeat the previous steps in order to be able to throw away the 'y' and be ready to read in another integer. We therefore need to set up our loop so that it can throw away an unknown number of characters.

We also need to be able to detect when end-of-file has been signalled. With our previous version of the loop, we would stop as soon as there is either an error or an end-of-file signal. We *don't* want to exit the loop in the case of an error, only in the case of end-of-file, so we're going to have to go back to an earlier version of the loop:

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

Where should the test for `cin.fail()` be placed in relation to the test for `cin.eof()`?

- before

● after

Yes, that's great!

This lets us modify our program to now look like `readInts5.cc`:

```
#include <iostream>

using namespace std;

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

# File I/O

Reading from a file is (almost) identical to reading from standard input. Instead of including the `iostream` library, you need to include the `fstream` library and use the `ifstream` type. Writing to a file is similar, except that you use the `ofstream` type. For example, take a look at the program `fileIO.cc` in your repository, in the directory `lectures/c++/02-io`:

```
#include <fstream>

int main() {
    std::ifstream infile{ "input.txt" };
    std::ofstream outfile{ "output.txt" };
    int i;
    while ( true ) {
        infile >> i;
        if ( infile.fail() ) break;
        outfile << i << std::endl;
    }
}
```

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`. (For further documentation, you can take a look at:

<http://www.cplusplus.com/doc/tutorial/files/>

The *name* of the file must be of either the type (`const char *`) or the type (`const string &`).

The "curly braces", "`{ }`", are C++11 syntax used to initialize the file stream objects. You will frequently see the same kind of initialization done with parentheses, "`( )`", but we recommend that you use curly brace syntax, since it can be used in the widest variety of situations, and works properly in more situations than parentheses. We will see it again when we discuss *uniform initialization syntax* in the section [Initializing objects](#).

If an input file cannot be opened for input (for example, it doesn't exist or cannot be opened for reading since the user running the program doesn't have read privileges on the file), what does your intuition suggest would happen?

- Program crashes with an error message.
- An exception is raised, but since there is no handler, the program aborts with an error message.
- The `fail` bit is set, and you can check its status by using `fail()`.

If an output file doesn't already exist when you create `outfile`, what do you think will happen?

- Program crashes with an error message.
- The system automatically creates the file `output.txt` with the new contents.

Yes, that's exactly right!

What if the output file already exists and you don't have write privileges on the file?

- Program crashes with an error message.
- An exception is raised, but since there is no handler, the program aborts with an error message.
- The `fail` bit is set, and you can check its status by using `fail()`.

Yes, that's exactly right!

The files will be automatically closed for us when the objects go "out of scope" and are destroyed on the final, closing brace ('}') for the program. This ensures that any final, buffered output will be sent to the output file. If you used the `open()` method to open the file instead, you must remember to call `close()` when you are done.



# Formatting output

We're rarely going to ask you to format your output in this course in any way other than through use of whitespace (spaces, tabs, newlines). It is, however, useful to know that there are ways to do this. Let's start by looking at the file `conversionChart.cc` in your repository, under `lectures/c++/02-io`. Note that we now need to include the library `iomanip` in order to have these features available to us.

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

int main () {
    for (int i=0; i < 20; ++i) {
        cout << dec << setw(3) << i << oct << setw(3) << i << hex << setw(3) << i << endl;
    }
}
```

The output is being formatted in two main ways.

1. We're 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 you can see in the output below, this lets us align the information into columns.
2. We're also using `dec/oct/hex` to make the integer `i` be printed in a different "base" system.
  - o `dec` is your base-10, *decimal* standard.
  - o `oct` is in base 8, and is called *octal*.
  - o `hex` is in base 16, and is called *hexadecimal*. Useful when printing out memory addresses/pointer contents.

If we compile and run the program, it produces the following output:

```
$ ./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 10 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
17 21 11
18 22 12
19 23 13
```

Let's take a look at one more example, `manip.cc` in `lectures/c++/02-io`.

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

int main () {
    int i = 95;
    cout << hex << i << endl;
    cout << i << endl;
    float price = 2.00;
    cout << fixed << showpoint << setprecision(2) << price << endl;
}
$ g++14 manip.cc
$ ./a.out
5f
```

5f  
2.00

Notes:

1. The dec/oct/hex feature is what we would call "sticky" in that once we change the integer value to be printed in a different base, it continues to print in that base until we tell it to be printed in another base system. This is why the integer 95 prints as "5f" twice.
2. showpoint is used to force the printing of the decimal point in our floating point number, price.
3. setprecision specifies the number of significant digits to print. By default, it uses '0' as the fill character.

The use of showpoint combined with setprecision is a good, simple way to print currency. For more information on the formatting features available, see: <https://en.cppreference.com/w/cpp/io/manip>

# Strings

| C  | C++   |
|--|---|
| Array of characters: <ul style="list-style-type: none"><li>• <code>char *</code>, or</li><li>• <code>char []</code></li></ul> Terminated by the null character, ' <code>\0</code> '. | Type is <code>std::string</code> . Requires inclusion of <code>&lt;string&gt;</code> library. You can use:<br><pre>using std::string;</pre> to avoid having to include the entire namespace if you don't want to have the prefix the type with <code>std::</code> everywhere. |
| Memory needs to be explicitly managed. Need to shrink/grow array explicitly.   | Manages memory for you.   |
| Easy to accidentally overwrite ' <code>\0</code> ' and corrupt memory.   | Safer to manipulate.  |
| <pre>char * s = "hello"; char name [10] = "Jane\0";</pre>  | <pre>#include &lt;string&gt; ... std::string s = "hello"; std::string name{ "Jane" };</pre>   |

Since C++ has a **string** library, you are required to use it wherever you would normally use a (`char*`) or (`char[]`) in C.

Note that the literal "hello" that appears on the right-hand-side of the assignment to the `std::string s` in the C++ example is actually a C constant character pointer (`const char*`), and not of type `std::string`. It is passed in to initialize `s`, in the same way that the literal "Jane" is used to initialize the `std::string name` object.

# Operations

We'll only discuss a few of the commonest string operations/operators. The complete reference page for `std::string` can be found at: <http://www.cplusplus.com/reference/string/string/>

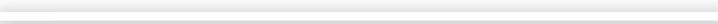
- equality: `s1 == s2`
- inequality: `s1 != s2`
- comparison: `s1 <= s2, s1 < s2, s1 >= s2, s1 > s2`
  - Note that comparisons are *lexicographic* i.e. in dictionary order. There's a nice, simple little example at <https://stackoverflow.com/questions/45950646/what-is-lexicographical-order> that explains the meaning.
- length: `s1.size()` or `s1.length()`
- fetch individual characters: `s1[0], s1[1], ...`, etc.
  - Note that C++ string indexing, just as with C strings, starts at 0 and goes up to `s1.size()-1`.
- concatenation: `s1 = s2 + s3; s3 += s4;`

# Input/Output

The `string` library defines how I/O works on the `std::string` class. There are two examples in the repository, `getStrings.cc` and `getline.cc`, located in `lectures/c++/02-io`, that show you some examples of how to read in and write out strings.

See an example of running `getStrings.cc` (MP4, with subtitles, 0:02:05)

0:00 / 2:04

A horizontal progress bar consisting of a thin black line with a small white dot near the left end.

and running `getline.cc` (MP4, with subtitles 0:01:28)

0:00 / 1:27

A horizontal progress bar consisting of a thin black line with a small white dot near the left end.

Does operator`>>` skip initial whitespace when reading in a `std::string` or not?

yes

no

Yes, that's correct!

Does `getline` take an input stream as a parameter?

yes

no

Yes, that's correct!

Does `getline` skip initial whitespace when reading in a `std::string`?

yes

no

Yes, that's correct!

Can `getline` be coerced to return a boolean value such that a value of `true` indicates success, while `false` indicates failure or end-of-file?

yes

no

Yes, that's correct!

Can reading a string set the fail bit?

yes

no

Yes, that's correct! The closest we can get to failing to read in a string is signalling end-of-file when attempting to read the string. The string *contains whatever value it had previously*, which is still a valid string, though probably not what we want!

# 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 I/O stream classes. It lets you read/write to/from strings using stream operators. (While you can use the `stringstream` for either input or output, we recommend that you use the type explicitly defined for input, [`istringstream`](#), or output, [`ostringstream`](#), as appropriate.)

## Input string streams

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. By default, the `istringstream` separates the words (often called "tokens") by whitespace. (Remember, `whitespace` consists of blanks, tabs, and newlines.) Since the input stream function `std::getline` lets us specify the delimiter to use when reading in lines of input as strings, we can use the same technique to specify a different delimiter to separate our tokens. For example, if I look at the program `istringstream.cc` in `lectures/c++/02-io` (shown below in the left-hand column):

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

int main () {
    string s;
    string s1{ "The quick brown fox\njumped over the lazy\t dog." };
    istringstream ss1{ s1 };
    while ( ss1 >> s ) {
        cout << s << endl;
```

```
$ g++14 istringstream.cc
$ ./a.out
The
quick
brown
fox
jumped
over
the
lazy
dog.
```

```
}

string s2{ "Smith,Jane,99999999,Yu,Yaoliang,99999998" };
istringstream ss2{ s2 };
cout << "***" << endl;
while ( getline( ss2, s, ',' ) ) {
    cout << s << endl;
}
}
```

```
***  
Smith  
Jane  
99999999  
Yu  
Yaoliang  
99999998
```

You can see from the output in the right-hand column that the first string, `s1`, was *tokenized/split* on the whitespace, while the second string, `s2`, was split on the commas.

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

Let's take a look at the programs `getNum.cc` and `readIntsSS.cc` in `lectures/c++/02-io`.

0:00 / 3:30



0:00 / 1:10



## Output string streams

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

There is almost no reason for you to use this in CS246, except for using a library such as X11 for graphics (which you may find yourself using on the final project) that only allows you to draw strings. So, if you wanted to output the final score for a game as an example, you'd have to convert the score integer and accompanying message into a string for the library to use; however, C++11 also provides `std::to_string` from the `std::string` library for the same reason.

See `buildString.cc` in `lectures/c++/02-io/`, whose code is shown below:

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

int main () {
    ostringstream ss;
    int lo {1}, hi {100};
    ss << "Enter a # between " << lo << " and " << hi;
    string s {ss.str()};
    cout << s << endl;
}
```

What should be the content of the `ostringstream` object `ss` after the line `"ss << "Enter a # between " << lo << " and " << hi;"?`

What do you think initializing the `std::string s` with the value of `ss.str()` does?

If you're not sure about any of these, including the contents of the variables `lo` and `hi`, please watch the following short video clip where we use the GDB debugger to look at how `buildString.cc` works.

# Command-line arguments

Command-line arguments in C++ are no different than in C: The `main` function may take two arguments, `argc` and `argv`, with `argc` representing the number of arguments the program received, and `argv` the array of arguments. Unfortunately, they are no different than in 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 short program, `lectures/c++/02-io/args.cc`, demonstrates command-line arguments in C++:

```
#include <iostream>

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

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

Note that, just like in C, `argv[0]` is the name of the program itself, and normal arguments continue from there. This is a convention you will explore in greater detail in CS350.

In cases where a C++ function explicitly requires a `string` as a parameter and not a `char*`, you can easily convert the `char*` into a `string`. For example:

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

## Command-line arguments vs standard input

Now that we've seen both standard input and command-line arguments in C++, it should be clear that command-line arguments and standard input are quite unrelated. However, many programs create the illusion that they're the same, but automatically opening a file named by the command-line argument if one is present. Don't confuse these two behaviours!

0:00 / 8:21



# Functions

Functions in C++ are very much like those in C. The basic format consists of:

```
return-type function-name( type1 arg1, type2 arg2, ... ) {  
    ...  
    return value of appropriate type;  
}
```

Note that forgetting to have a **return statement** when a **return type other than void** is specified can lead to your program behaving erratically. Using **-Wall** when compiling your program will warn you of this error—let the compiler help you!

For example, we could define a function add1 that takes an integer as a parameter, and returns the value with 1 added to it:

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

We could then invoke it as:

```
cout << add1( 3 );
```

## Forward declarations

You should already be familiar with the concept of separating a function into its **declaration** (signature), and its **definition** (implementation). If you don't remember this, or aren't clear on the difference, there's a good discussion of this available at: [https://www.cprogramming.com/declare\\_vs\\_define.html](https://www.cprogramming.com/declare_vs_define.html)

It's very common in C and C++ to give a list of all of 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're going to see this topic again when we get to [separate compilation](#) and the [role of the preprocessor](#).

You have an example of this in your repository called `forwardBad.cc`, in the directory `lectures/c++/03-functions`. As you may imagine from the name, this program won't compile successfully.

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

bool even(unsigned int n) {
    if (n == 0) return true;
    return odd(n - 1);
}

bool odd(unsigned int n) {
    if (n == 0) return false;
    return even(n - 1);
}

int main() {
    cout << boolalpha << even(3) << " " << even(4) << " "
        << odd(3) << " " << odd(4) << endl;
}
```

```
$ g++14 forwardBad.cc
forwardBad.cc: In function "bool even(unsigned int)":  
forwardBad.cc:7:10: error: "odd" was not declared in this scope  
    return odd(n - 1);  
           ^~~  
forwardBad.cc:7:10: note: suggested alternative: "void"  
    return odd(n - 1);  
           ^~~  
                                void
```

As is often the case with C and C++ compilation errors, one error causes multiple other errors.

**The usual correction process is to start with fixing the first error, and then recompile to see what errors are left that need to be fixed.**

In our example, the compiler is telling us that on line 7, we are trying to use something that hasn't yet been declared. And if we look at the program, we are attempting to use the function `odd` before we've declared or defined it.

We can correct this program by putting the declaration of `odd` before the definition of `even`. The revised solution can be found in `forwardGood.cc`, and looks like:

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

```
$ g++14 forwardGood.cc
$ ./a.out
```

```
bool odd(unsigned int n);

bool even(unsigned int n) {
    if (n == 0) return true;
    return odd(n - 1);
}

bool odd(unsigned int n) {
    if (n == 0) return false;
    return even(n - 1);
}

int main() {
    cout << boolalpha << even(3) << " " << even(4) << " "
        << odd(3) << " " << odd(4) << endl;
}
```

false true true false

As you can see, this version both compiles and runs successfully!

# Overloaded functions

Where C++ is different from C is that it allows a function to be [overloaded](#). In other words, I can have more than one function with the exact same function name so long as the number of arguments and/or their types are different. We already saw this with our input and output operators, since we're using the same function names, operator<< and operator>>, and just changing the type of the information being read or written. This is actually very useful, since it means that you don't have to come up with a new function name once you've found a good, meaningful one, just because you need another version that takes a different combination of parameters. Overloading lets the users leverage existing knowledge as well—once you know how to do I/O in C++, it doesn't matter what the type is, you know exactly how to read it in or write it out, using either operator>> or operator<<.

The compiler, however, does **not** distinguish between the overloaded functions based upon return types!

**Note that the decision as to which function is to be called must be made at compile-time.**

| Works  | Doesn't work   |
|--|--|
| <pre>\$ cat t.cc #include &lt;iostream&gt; int add1(int v1, int v2) {     std::cout &lt;&lt; "add1(int,int): ";     return v1+v2; } int add1(int value) {     std::cout &lt;&lt; "add1(int): ";     return value+1; } char add1(char value) {     std::cout &lt;&lt; "add1(char): ";     return value+'a'; }</pre> | <pre>\$ cat t2.cc #include &lt;iostream&gt; int add1(int value) {     std::cout &lt;&lt; "add1(int): ";     return value+1; } char add1(int value) {     std::cout &lt;&lt; "add1(char): ";     return value+'a'; }  int main() {     int i = 7, j = 8;     char c = add1(i);     std::cout &lt;&lt; add1(2) &lt;&lt; std::endl;</pre> |

```

int main() {
    int i = 7, j = 8;
    std::cout << add1(2) << std::endl;
    std::cout << add1('m') << std::endl;
    std::cout << add1(i,j) << std::endl;
}
$ g++14 -Wall t.cc
$ ./a.out
add1(int): 3
add1(char): n
add1(int,int): 15

```

```

$ g++14 t2.cc
t2.cc: In function "char add1(int)": t2.cc:6:6: error: ambiguating new declaration of "char add1(int)"
char add1(int value) {
^~~~
t2.cc:2:5: note: old declaration "int add1(int)"
int add1(int value) {
^~~~

```

Even more interesting, if I try to call `add1` passing in a real number (type `float`), the compiler will complain that it doesn't know which version of `add1` to use, since either could work if the compiler **narrow**s the `float` by throwing away the decimal portion to get an (`int`), and it can take a similar action to create a `char` since a character is really just a (`short unsigned int`). (There is similar but opposite action, called **widening**. These are both examples of [implicit conversions](#).)

## Default parameters

It's pretty common to have more than one version of a function, that mostly does the same thing; but, the addition of one (or more) parameters/flags turns on (or off) some additional behaviour. The key idea here is that the versions are **extremely similar**.

For example, I might have a function `processInput` that reads in information from standard input and processes it. However, in certain circumstances, I might want to copy information to a file, `log.txt`, as it is processed for debugging, or for use elsewhere.

| Version   | Invocation  |
|---|---|
| <pre>void processInput() {     string line;     while ( getline( std::cin, line ) ) {</pre> | <pre>processInput(); processInput( false );</pre> |

```

        // process input
    } // while
} // processInput

```

```

void processInput( bool log ) {
    ofstream logfile{ "log.txt" };
    string line;
    while ( getline( std::cin, line ) ) {
        // process input
        if ( log ) {
            // write processed output to logfile
        } // if
    } // while
} // processInput

```

```
processInput( true );
```

I'd rather not duplicate the code. This is always a bad idea since it means that I now have to maintain two separate versions and any correction/change I make to one has to be made to the other. What I'd prefer is to have one version, where most of the time I would invoke it as `processInput()`, which causes it to behave as if I'd invoked `processInput( false )`. That way, I only need to pass in a boolean value of `true` when I explicitly want to use the log file.

C++ provides a mechanism that lets me do this called **default parameters**. The idea is that some (or all) of your parameters have a default value that the compiler will use, unless the programmer provides a value to take the place of the default one.

**The default values are, by convention, usually listed in the function declaration and not in the function definition.  
They cannot occur in both locations!**

| Function declaration                                | Function definition   |
|---|---|
| <code>void processInput( bool log = false );</code> | <code>void processInput( bool log ) {</code><br><code>    ofstream logfile{ "log.txt" };</code><br><code>    string line;</code><br><code>    while ( getline( std::cin, line ) ) {</code><br><code>        // process input</code> |

```

        if ( log ) {
            // write processed output to logfile
        } // if
    } // while
} // processInput

```

This way, if the programmer invokes `processInput` as `processInput()`, this is treated by the compiler exactly as if the programmer had typed in `processInput( false )`. If the programmer wants a value other than `false` passed in for the parameter `log`, `processInput` would have to be invoked as: `processInput( true )`. We also only have one version of the function!

**There is one additional rule of use: if some of your parameters have default values, and some do not, all of those with default values must come after those without default values in the list of arguments.**

| Right  | WRONG!   |
|--|--|
| <pre>void foo( int i1, int i2,           char delim = ',', bool flag = true ); void foo( int i1, int i2,           bool flag = true, char delim = ',' );</pre> | <pre>void foo( char delim = ',', bool flag = true,           int i1, int i2 ); void foo( char delim = ',', int i1,           bool flag = true, int i2 ); void foo( bool flag = true, char delim = ',',            int i1, int i2 ); void foo( int i1, bool flag = true,           char delim = ',' int i2 );</pre> |

Here's another example from your repository, `default.cc`, located in `lectures/c++/03-functions`.

|  |   |
|--|---|
| <pre>#include &lt;iostream&gt; #include &lt;fstream&gt; using namespace std;  void printSuiteFile( string name = "suite.txt" ) {     ifstream file{ name }; </pre> | <pre>suite.txt</pre> <pre>testA testB testC testD</pre> |
|--|---|

```
string s;
while ( file >> s ) cout << s << endl;
}

int main() {
    printSuiteFile();
    cout << endl;
    printSuiteFile( "suite2.txt" );
}
```

suite2.txt

```
testE
testF
testG
testH
```

If we compile and run this, we should see the contents of suite.txt output, then a newline, then the contents of suite2.txt.

```
$ g++14 default.cc
$ ./a.out
testA
testB
testC
testD

testE
testF
testG
testH
```

# Structures

C++, for compatibility reasons, lets you use the `struct` keyword just as you would in C i.e. as a way to "collect" information that isn't of the same type. (If they were of the same type, you could choose to use an array instead.) If you need a refresher on the basics, please read through: <http://www.cplusplus.com/doc/tutorial/structures/>

In particular, let's take a look at how we'd define a structure for a node in a singly-linked list of integers in C versus how we do it in C++.

| C  | C++   |
|--|---|
| <pre>typedef struct Node_t {     int value;     struct Node_t * next; } Node;  Node * head = NULL; Node n; n.value = 5; n.next = NULL;</pre> | <pre>struct Node {     int value;     Node * next; };  Node * head = nullptr; Node n{ 5, nullptr }, n1 = { 6, head };</pre> |

Note the following differences:

- C++ doesn't require that the type definition start with the keyword `typedef`.
- As well, the keyword `struct` doesn't need to be repeated in all of the variable declarations. It just needs to be done once, in the initial type definition.
- **C++ no longer uses `NULL`; rather you should be using `nullptr` instead. There's a nice discussion of the reason why at [https://en.wikibooks.org/wiki/More\\_C%2B%2B\\_Idioms/nullptr](https://en.wikibooks.org/wiki/More_C%2B%2B_Idioms/nullptr)**

- There is no "short version" of the type name before the closing semicolon.
- There must always be a closing semicolon, after the closing '}'.

Let's take a look at an incorrect C++ definition of Node:

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

Why is it wrong? The problem lies with the line "Node next;". You're in the middle of defining the type, and the definition isn't complete until you reach the closing '}'. But you're trying to specify that next is of type Node, so the compiler needs to know how much space to allocate. But it won't know that until the end of the definition is reached! The only way around this to make next a type that has a known, fixed size. An easy way to do this is to use a pointer (or a reference), since all pointers are of the same size i.e. a pointer just contains a memory address, and the size of memory addresses is known at compilation.

# Constants

Another useful concept is that of having a value that is **immutable**, that 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, as in:

```
const int MAX_GRADE = 100;  
const double PI = 3.14159;
```

**WARNING: stating that a value is `const` means that it must be initialized when it is defined since it cannot be changed later!**

It is also possible to declare a more complex type such as a struct a constant. For example:

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

Here `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. For example, if my program had the line:

```
n2.value = 13;
```

attempting to compile the program would result in the error message:

```
t.cc:NN:MM: error: assignment of member "Node::value" in read-only object  
n2.value = 13;  
^~
```

# Parameter passing

Let's take a look at function that we would like to take in an integer value, increment it by 1, and have the original parameter's value changed. In this situation, we don't want to have to write code such as:

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

We would prefer that the variable `value` be modified *in place*. In other words, we would like to be able to instead write code such as:

```
void inc( int n ) { ++n; }
...
inc( value );
```

We have an example of this program in our repository as `lectures/c++/03-functions/incrementBroken.cc`. Unfortunately, it doesn't work properly. Let's take a closer look at the problem.

0:00 / 2:10

---

The fundamental problem is that we passed the variable `x` "by value", which meant that `inc` was working with a **copy** of `x`. This process is named **call-by-value**. Now, sometimes this is exactly the behaviour we want i.e. modify the copy and leave the original unchanged. But not in this particular case. The C way to solve the problem would be to pass in the address of `x` to `inc`.

This would change the relevant lines of code to:

```
void inc( int * n ) { ++*n; } // same as *n = *n + 1;  
...  
inc( &value );
```

A working version of the program can be found in `lectures/c++/03-functions/incrementFixed.cc`. Note that the *address of x* is still passed by value, but that's okay since we're not trying to change that, just the data where it points **to**.

So, if we can fix the call-by-value problem by taking the address of the parameter and passing that, why don't we have to say

```
cin >> &x;
```

rather than:

```
cin >> x;
```

? Because it turns out that C++ has another type similar to a pointer, called a **reference**, that we will discuss next.

# References

References are a very important topic. Since we're going to be using them a lot, it's very important that you understand them well.

In particular, the general rule in this course is that you should pass information by reference (*pass-by-reference*), rather than by pointer, as much as possible since it makes it easier to write the function code i.e. you don't have to remember to dereference your pointers all of the time.

We first need to introduce two terms that are going to help explain the concept, that of **rvalues** and **lvalues**. Scott Meyers, on page 2 of "Effective Modern C++", gives a simple explanation that works well. If you can take the address of an expression, it is an **lvalue**; otherwise, it's an **rvalue**. You may also find it helpful to think of an lvalue as something that can appear on the left-hand side of an assignment statement. For example, in the following code fragment:

```
int x = 5;
int * ptr = nullptr;
```

x and ptr are both lvalues.

So, what is a **reference**? A reference is an lvalue that acts like a constant pointer but the compiler automatically dereferences it. Since it's a constant, it **must** be initialized when it is defined. For example, in the following code fragment:

```
int x = 5;
int &y = x;
int * ptr = &y;
y += 2;
*ptr += 3;
std::cout << x << std::endl;
```

y is an *lvalue reference*, "bound" to the lvalue x. ptr is another lvalue. We will often use the term "alias" here for a reference i.e. "y is an alias for x" in that y is another name for x and behaves like x in all circumstances.

ptr contains the address of which lvalue? See the video below.

0:00 / 2:03



ptr contains the address of which lvalue?

- x
- y
- Neither x nor y since this is an error.

Note that the type of y is `(int &)`, and not `(int *)` or `(int)`.

**There are a number of things that cannot be done with an lvalue reference:**

1. Cannot leave them uninitialized. For example, `int &x;` is illegal. In fact, it must be initialized with an lvalue, since it's a pointer.

```
int &x = 3; // illegal!
int &x = y + z; // illegal!
int &x = y; // okay
```

2. Cannot create a pointer to a reference. For example, `int &x;` isn't legal, but a reference to a pointer is perfectly legal.

3. Cannot create a reference to a reference. For example, if we have the following code fragment: `int &&x;`, this doesn't actually mean a "reference to a reference", but something completely different. (We'll see this later when we discuss *move semantics*.)

4. Cannot create an array of references, such as `int &refArray[3] = {x, y, z};`.

How would you create a reference to the integer pointer, `ptr`?

- `int &&refPtr = ptr;`
- `int &* refPtr = ptr;`
- `int *& refPtr = ptr;`
- `int ** refPtr = ptr;`

Yes, that's correct!

So, what can you do with a reference? You can pass it as a function parameter type, or as the return type for a function. This will let us fix our increment function, by changing it to the version found in `lectures/c++/03-functions/incrementRef.cc`.

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

void inc(int &n) {
    n = n + 1;
}

int main () {
    int x {5};
    inc(x);
    cout << x << endl;
}
```

If you compile and run it, you will see that it works correctly now i.e. `x` contains the value 6 after the call to `inc`. This then lets us go back and explain how operator`<<` and operator`>>` are defined:

```
std::istream & operator>>( std::istream & in, int & value );
std::ostream & operator<<( std::ostream & out, const int & value );
```

Why is the parameter `value` in operator`<<` being passed as a `(const int &)` i.e. a constant integer reference? Remember that *pass-by-value* copies the contents of the parameter or return value. (It's actually a bit more complicated than that, but we'll explain this more completely when we get to [copy semantics](#)). This means that if the parameter is very large, then copying it can be computationally very expensive. But if we were copying it to avoid allowing the receiving code from being able to change the value, we can make this considerably cheaper by passing a constant reference. Since all references are effectively constant pointers i.e. the address can't change, but the contents of what the address points to can change, if we make the reference a "constant pointer to a constant", then the user can't change the value either! And all references are the size of a pointer, which is a known, fixed, and small size (typically about 8 bytes). So, copying a constant reference is inexpensive, and that's what we're using in the output operator, operator`<<`, for `value`.

To summarize:

```
struct ReallyBig { ... }; // struct that holds a *lot* of information  
  
int f( ReallyBig rb ) { ... } // Copies rb. Potentially slow; but see move/copy ellision later!  
int g( ReallyBig &rb ) { ... } // Copies the alias. Fast; but, allows rb to be changed!  
int h( const ReallyBig & rb ) { ... } // Fast. No copy and rb cannot be changed.
```

**Advice: prefer *pass-by-const-ref* over *pass-by-value* for anything larger than an integer, unless the function needs to make a copy anyways. Then it may be reasonable to use *pass-by-value* instead. There's a good discussion of this at <https://isocpp.org/wiki/faq/references#call-by-reference>.**

One word of warning. What if you want to pass a literal integer into a function that takes a reference? For example, I want to call a function *f* with the value 5, as in *f(5)*. You might think you could define *f* as:

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

but the compiler won't allow that since you're implying that you could change what is considered a constant by the compiler. Instead, you'd have to define *f* as:

```
int f( const int & n ) { ... }
```

Why does this work?

0:00 / 1:20



There is some good information on references at: <https://isocpp.org/wiki/faq/references>

# Dynamic memory allocation

If you need a quick refresher on where information is stored in C, take a look at

<https://www.geeksforgeeks.org/memory-layout-of-c-program/>. This will conceptually also apply to C++, though the C++ standard uses the terms *automatic* versus *dynamic* storage.

You will often need to allocate memory dynamically on the heap, though it is preferable, where possible, to allocate it on the run-time stack.

| C   | C++   |
|---|---|
| <ul style="list-style-type: none"><li>• Uses a library, <code>cstdlib</code>, now known as <code>stdlib.h</code>, that provides functions <code>malloc</code>, <code>calloc</code>, <code>realloc</code> and <code>free</code>.</li><li>• Only works with type <code>(void*)</code>, so <b>not</b> type safe.</li></ul> | <ul style="list-style-type: none"><li>• Uses keywords <code>new</code> and <code>delete</code>.</li><li>• Type safe i.e. allocates space of the appropriate size and returns a pointer of the appropriate type.</li></ul> |

You must use the C++ mechanisms for dynamic memory allocation in this course. Marmoset is configured to reject code that uses the C mechanisms or `std::free`!

If you need a refresher on singly-linked lists, you can look at either the [CS136 lecture material](#) or one of the many websites such as <https://www.geeksforgeeks.org/data-structures/linked-list/>. You can also find a refresher on binary search trees in the CS136 lecture material or at one of the many websites such as <https://www.geeksforgeeks.org/binary-search-tree-data-structure/>.

Let's take a look at a simple program that performs dynamic memory allocation and how memory is managed during its execution.

```
#include <iostream>

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

int main() {
    Node n{ 5, nullptr };
    Node * np = new Node{ 3, &n };
    std::cout << n.data << ' ' << np->data << std::endl;
    delete np;
}
```

The program is stored in memory, in the text segment.  
We are about to allocate the local variable `n` of type  
(`Node`) on the run-time stack in the frame associated  
with the function `main`.

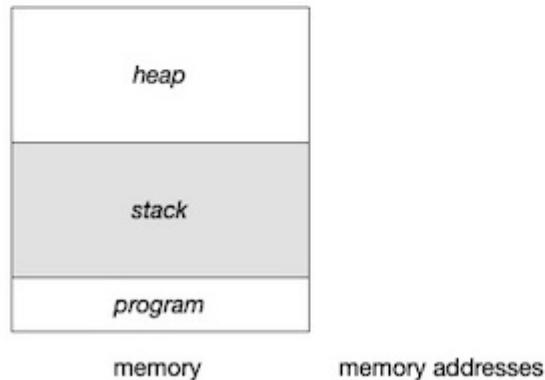
```

#include <iostream>

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

► int main() {
    Node n{ 5, nullptr };
    Node * np = new Node{ 3, &n };
    std::cout << n.data << ' ' << np->data << std::endl;
    delete np;
}

```



Space for `n` has been allocated on the stack, and its data fields have been initialized. We are next going to allocate space for the local variable `np` on the stack.

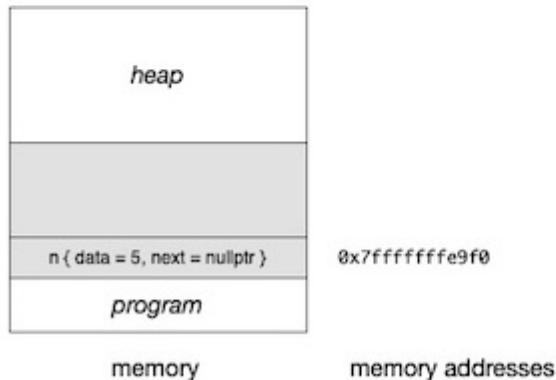
```

#include <iostream>

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

int main() {
    ► Node n{ 5, nullptr };
    Node * np = new Node{ 3, &n };
    std::cout << n.data << ' ' << np->data << std::endl;
    delete np;
}

```



Space for the local variable np is allocated on the run-time stack, on top of the space allocated to the local variable n. np contains the address of sufficient space allocated in the heap to hold a Node, initialized so that its data field contains the integer 3, while its next field contains the address of the node n.

```

#include <iostream>

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

int main() {
    Node n{ 5, nullptr };
    Node * np = new Node{ 3, &n };
    std::cout << n.data << ' ' << np->data << std::endl;
    delete np;
}

```

|                                     |                  |
|-------------------------------------|------------------|
| { data = 5, next = 0xffffffffe9f0 } | 0x555555767e70   |
|                                     |                  |
|                                     |                  |
| np { 0x555555767e70 }               | 0x7fffffff9e8    |
| n { data = 5, next = nullptr }      | 0x7fffffff9f0    |
| program                             |                  |
| memory                              | memory addresses |

We have now marked the heap space that np is pointing to, address 0x555555767e70, as freed. Note that the variable np *still* contains the old address (**not** nullptr), and the contents of that memory location **remain as before** until overwritten!

When the program reaches the closing brace, '}', the local variables are popped from the run-time stack in the reverse order to creation i.e. we first pop np and then pop n.

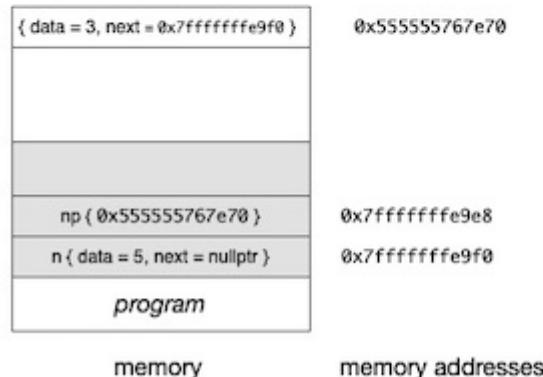
```

#include <iostream>

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

int main() {
    Node n{ 5, nullptr };
    Node * np = new Node{ 3, &n };
    std::cout << n.data << ' ' << np->data << std::endl;
    delete np;
}

```



If you fail to delete all dynamically allocated memory, you have what is called a **memory leak**. If your program has a memory leak, it is considered incorrect in this course since it would eventually fail—not all systems have garbage collection, so this is a problem. In fact, your submitted code will be checked with **valgrind** for memory leaks, and marks will be removed if leaks are detected.

## Dynamic array allocation

The syntax for dynamically allocating and freeing an array is a bit special:

```

Node * nodeArray = new Node[10];
...
nodeArray[0].data = 5;
nodeArray[0].next = nullptr;
...
delete [] nodeArray;

```

Here we've allocated an array of 10 nodes. The variable `nodeArray` is a local variable, stored on the run-time stack. It contains the address of a section of memory on the heap that is large enough to hold 10 nodes. Note that in order to completely free the array, I **must** specify the empty square brackets between the `delete` keyword and the variable name `nodeArray`. If you forget the square brackets, you will only free the first element of the array, and will have a memory leak.

**It is important that you properly match the new and delete statements. If your new used square brackets, your delete must also use square brackets! Mixing the two forms results in undefined, unpredictable behaviour.**

If you need a refresher on how to create multi-dimensional arrays dynamically, there's a good discussion at:  
<https://www.techiedelight.com/dynamic-memory-allocation-in-c-for-2d-3d-array/>

# Pointers and optional files

It's very common in C/C++ to have a program default to reading from standard input and writing to standard output, but allow optional command-line arguments to specify the names of input and output files that should be used instead if they are present. The convention in Linux/UNIX to use square brackets in the documentation to show that these command-line arguments are optional. For example, if the program executable is called `test`, my documentation could specify that `test` is run as:

```
./test [ input-file-name [output-file-name] ]
```

In other words, if there are two command-line arguments present, the first is the name of the input file to use, while the second is the name of the output file. If only one argument is present, then it is the name of the input file to use, and the output should go to standard output. If no arguments are present, then the program reads from standard input and writes to standard output.

C++ uses the same syntax as C for obtaining the command-line arguments. Our `main` routine is modified to show that it now has two arguments. The first is an integer that tells us the number of arguments. By convention, it is called `argc`. The second parameter is an array of NULL terminated C-style strings (`char*`).

If I run the program fragment as:

```
./a.out "abc def" 123
```

then when I examine the contents of `argc` and `argv` in my `main` function, which has been defined as:

```
int main( int argc, char * argv [] ) {  
    ...  
}
```

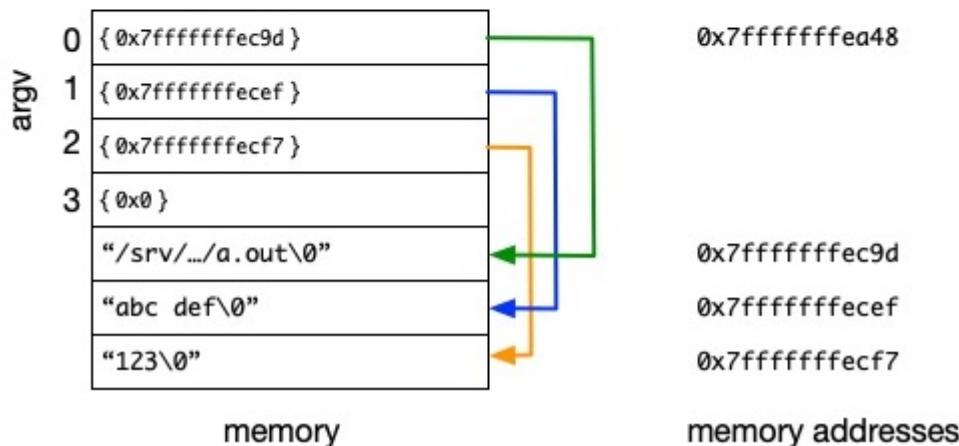
```
#include <iostream>
using namespace std;

int main( int argc, char * argv [ ] ) {
    ...
}
```

`./a.out "abc def" 123`

`argc = 3, argv = 0x7fffffff9d`

`argc` will have the value 3 and `argv` will be an array of 4 C strings, where the final element is the C equivalent to the C++ `nullptr`. Note that each element of `argv` is a pointer to a null-terminated array of characters i.e. a C string. (This also implies that if we want to use `argv[2]` as an actual integer rather than a C string, that we'll have to convert it using either an [`istringstream`](#) or [`std::stoi`](#).



We can then set up our program in the following fashion, where you can find a copy in your repository as `lectures/c++/03-functions/io.cc`:

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

void usage( char * pgmname ) {
    std::cerr << pgmname << " [ input-file-name [output-file-name] ]" << std::endl;
```

```
    } // usage

    int main( int argc, char * argv[] ) {
        std::istream * infile = &std::cin;
        std::ostream * outfile = &std::cout;

        if ( argc > 3 || argc < 1 ) {
            usage( argv[0] );
            return 1;
        } // if

        switch( argc ) {
            case 3:
                outfile = new std::ofstream{ argv[2] };
                if ( outfile == nullptr || outfile->fail() ) {
                    delete outfile;
                    std::cerr << "ERROR: unable to open output file \""
                        << argv[2] << '\"' << std::endl;
                    return 1;
                } // if
                // fall through to open up the input file next
            case 2:
                infile = new std::ifstream{ argv[1] };
                if ( infile == nullptr || infile->fail() ) {
                    if ( outfile != &std::cout ) delete outfile;
                    delete infile;
                    std::cerr << "ERROR: unable to open input file \""
                        << argv[1] << '\"' << std::endl;
                    return 1;
                } // if
                break;
            default:
                // do nothing
                break;
        } // switch

        // Echoes the input to output.
        std::string line;
```

```
while ( std::getline( *infile, line ) ) {
    *outfile << line << std::endl;
} // while

// Closes the I/O files. Don't delete if standard input or output, though.
if ( infile != &std::cin ) delete infile;
if ( outfile != &std::cout ) delete outfile;
} // main
```

We start by setting our `infile` variable to contain the address of `std::cin`, while our `outfile` variable contains the address of `std::cout`. This handles the default case where there are no command-line arguments, and we'll read from standard input and write to standard output.

If we have more than 3 command-line arguments or fewer than 1 (not possible, but makes the program easier to generalize for other situations), we terminate the program with a usage message.

If there are two command-line arguments (`argc == 3`) then we'll make `outfile` point to a new `ofstream` object, initialized with the name of the file contained in `argv[2]`. If `outfile` has its fail bit set, we were unsuccessful in opening the output file (the test for `nullptr` is there since if we accidentally had a value of `nullptr` in `outfile` and tried to dereference `outfile` in order to call its `fail` method, we'd crash the program). Since there is no `break` statement following the `if` statement, we'll "fall through" and execute the code associated with "case 2:", the same situation if we'd only specified a single command-line argument.

We then change `infile` to contain the address of the input file specified in `argv[1]`. If `infile` has its fail bit set, we were unsuccessful in opening the input file.

We then read in each line from `infile`, and output the string to `outfile`. Notice that we've had to dereference them since they are pointers.

If we opened the files by creating them on the heap, then we need to delete the heap-allocated objects in order to properly close the files. It is very important, however, that you **don't** delete the `infile` if it set to contain the address of `std::cin`. The same also holds for `outfile` and `std::cout`.



# Returning information

There are three ways in which we can return information from a function in C++:

1. return by *value*
2. return by *pointer*
3. return by *reference*

We will explore each in turn using our previous Node type declaration as we write a function `getMeANode` that will create a new Node for us to use in our program. Remember that we declared our Node type as:

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

## Return by value

Here our function returns a copy of the local node `n`, copying it onto the run-time stack.

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

Copying `n` is potentially expensive depending upon what is involved when we copy a node. We'll discuss this in more detail when we discuss *copy semantics* versus *move semantics*.

## Return by pointer

Our first version consists of:

```
Node * getMeANode( int value ) {  
    Node n{ value, nullptr };  
    return &n;  
}
```

This is an easy mistake to make, which is why we're showing it to you. We're returning the address of a *local*, stack-allocated variable, n, which is popped from the run-time stack when the function ends. The caller now has the address of an invalid memory location, which will crash the program when used.

Let's change our code to instead allocate the node n on the heap:

```
Node * getMeANode( int value ) {  
    Node *nptr = new Node{ value, nullptr };  
    return nptr;  
}
```

This will work, but our caller is now responsible for calling delete on the received address; otherwise, we'll have a memory leak.

## Return by reference

This is going to look similar to what we saw with pointers. Our first version consists of:

```
Node & getMeANode( int value ) {  
    Node n{ value, nullptr };  
    return n;  
}
```

This is also invalid, and for the same reason i.e. the recipient is receiving a reference to local variable that no longer exists once the function ends.

We can fix it in a similar fashion, by using heap allocation:

```
Node & getMeANode( int value ) {  
    Node *nptr = new Node{ value, nullptr };  
    return *nptr;  
}
```

The real problem, however, is that the caller won't know that the node was allocated on the heap unless they read the code documentation, and thus has no idea that they're supposed to delete it, as in:

```
Node & newNode = getMeANode( 5 );  
...  
// do something with newNode  
...  
delete &newNode;
```

Note that `delete` can only be applied to a pointer, so we have to take the address of `newNode` by using `&newNode`, which will give us the *address of the heap-allocated memory* to free.

## Which technique should we use?

In some cases, returning a pointer is fine, or you have a situation where you need to return a reference (I/O stream operators, for example); however, in most situations, it turns out that return by value is actually the right thing to do since it turns out that as of C++11 (and up), it's not as expensive as it looks.

# Operator overloading

Remember that we earlier discussed that fact that in C++ we can *overload* functions? Well, it turns out that we can also overload operators such as +, -, \*, /, etc. in C++. (For a full list of what operators can be overloaded, see <https://en.cppreference.com/w/cpp/language/operators>.) This lets us create more intuitive function names for some of our types. For example, if I define a vector as a structure that contains two integer data fields, x and y, then if I create two instances of them, v1 and v2, I would have an intuitive understanding of what should happen if I write v1 + v2 in my program. In other words, I expect to receive a new, third vector whose x field contains the sum of v1.x and v2.x, while its y field contains the sum of v1.y and v2.y.

```
struct Vec {  
    int x, y;  
};  
  
Vec operator+( const Vec & v1, const Vec & v2 ) {  
    Vec v{ v1.x + v2.x, v1.y + v2.y };  
    return v;  
}  
  
int main() {  
    Vec v1{3, 4}, v2{6, 7};  
    Vec v3 = v1 + v2;  
}
```

Similarly, I can define operator\* that multiplies a vector by an integer. Note that if I only define operator\* as

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

then it can only be invoked as 2\*v1 and **not** as v1\*2! (The last line in operator\* introduces something new, the ability to create an **anonymous object**, which is an [rvalue](#), to return i.e. if I wrote the function in the same way I did

operator+, the local variable v is only used to create a new instance of Vec to return, and it's not used anywhere else. So, C++ lets us avoid even needing a local variable as an intermediate step!

We can take this one step further, where we define the other version of operator\* that lets us multiply a vector by an integer as v1\*2!

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

The C++ standard nowadays lets the compiler intuitively determine the type of instance to create in the return statement from the return type of the function. So, we can make this even better by recognizing that we've already defined how to multiply k\*v, and multiplication is commutative, so we should instead write our second version of operator\* as:

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

You can see the full example in your repository as `lectures/c++/04-operators/vectors.cc`.

## Overloading << and >>

Overloading the input and output operators is a bit more complex. Let's take a look at an [abstract data type](#) for a course grade. We *could* represent it as an (unsigned) integer, but there's no way to ensure that every time we initialize or change the grade, that we ensure that it stays in the value range of 0 to 100. If we wrap it in an abstract data type, and ensure that it is only ever accessed by the proper functions, we can guarantee that its contents falls in the range 0 to 100 (this will be even easier once we introduce the C++ `class` keyword).

**Aside:** From a software engineering perspective, there's an interesting discussion as to whether an abstract data type should be an [entity object](#) or a [value object](#). This is part of the topic of *domain driven design*. (This is not testable material in CS246,

but if you're interested in software design, this information is worth reading.)

We'll define a Grade structure as:

```
struct Grade {  
    int theGrade;  
};
```

Defining the output operator for a Grade is straightforward compared to the input operator, so we'll start with this operator. We just want to output the value, and then a '%' sign to indicate that it is a percentage i.e. out of 100. Any other formatting will be done by the caller, which is why we're not outputting a newline or anything else.

```
std::ostream & operator<<( std::ostream & out, const Grade & g ) {  
    out << g.theGrade << '%';  
    return out;  
}
```

Note that while we're passing the grade as a constant reference, we are **not** passing the output stream out as a constant reference. That's because we're changing the contents of the output stream out in our function since that's where we're putting the value of the grade. As well, note that we're returning out and not std::cout since we don't know what the caller has passed in as a value for out. It could be std::cerr, or a output file stream, and we don't know! So, we **must** return out.

As well, we're returning out as an (std::ostream&). Why? Remember, this lets us *cascade* calls to the output operator on the stream. For example, this could be part of the following code fragment:

```
Grade sue{95};  
ofstream outfile{ "final_grades.txt" };  
...  
outfile << "Sue's final grade is: " << sue << std::endl;
```

**Rules for writing an output operator:**

1. The return type is always `(std::ostream &)`.
2. The function name is always `operator<<`.
3. The first parameter is always the output stream, `(std::ostream &)`.
4. The second parameter is always the information being output. If it's the size of an integer or smaller, you can pass it by value, but it's *usually* a constant reference.
5. Before you do anything else, write the return statement to return whatever the name of the output stream is.

```
std::ostream & operator<<( std::ostream & out, const typeToPrint & value ) {  
    out << v ; // whatever is appropriate for the type you are outputting  
    return out;  
}
```

The input operator changes both the input stream since it is consuming information from it, and the second parameter that is supposed to hold the information read in from the input stream. This should imply to you that both parameters need to be passed as references, and **not** as constant reference.

```
std::istream & operator>>( std::istream & in, Grade & g ) {  
    in >> g.theGrade;  
    if ( g.theGrade < 0 ) g.theGrade = 0;  
    else if ( g.theGrade > 100 ) g.theGrade = 100;  
    return in;  
}
```

The one thing we haven't done is determine what to do if our attempt to read in `g.theGrade` fails if there is no integer in the input stream. Since we're supposed to ensure that grades are only in the range 0 to 100, and any of

those values are valid, then there's no way we can signal to the caller that our attempt to read fail. We'll find a better way to deal with this once we get to *exception handling*!

### Rules for writing an input operator:

1. The return type is always (`std::istream &`).
2. The function name is always `operator>>`.
3. The first parameter is always the input stream, (`std::istream &`).
4. The second parameter is always the information being read in. It will be a reference.
5. Before you do anything else, write the return statement to return whatever the name of the input stream is.

```
std::istream & operator>>( std::istream & in, typeToRead & value ) {  
    in >> v ; // whatever is appropriate for the type you are reading in  
    return in;  
}
```

You can see the full example in your repository as `lectures/c++/04-operators/grades.cc`.

# The Preprocessor

In C, you saw this:

```
#include <stdio.h>
```

In C++, you've seen this:

```
#include <iostream>
```

But what is that `#include` actually doing?

Both C and C++ use a tool called the **C preprocessor**, which is tasked with handling **preprocessor directives** like `#include`. Running the preprocessor is one of the very first steps the C++ compiler performs, and its behavior is fairly simple. The preprocessor's primary purpose is to control what code is included in your program.

We can see what the preprocessor does by running it directly, without running the rest of the C++ compiler. We will demonstrate this with the following two files:

| world.cc                            | hello.h |
|-------------------------------------|---------|
| <pre>#include "hello.h" WORLD</pre> | HELLO   |

To run only the preprocessor, use the `-E` flag to `g++`. The preprocessor will output to standard out.

```
$ g++ -std=c++14 -E world.cc
# 1 "world.cc"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
```

```
# 1 "<command-line>" 2
# 1 "world.cc"
# 1 "hello.h" 1
HELLO
# 2 "world.cc" 2
WORLD
```

The lines it outputs labelled with a number are simply comments to indicate from where the text is included. More importantly, you can see that "HELLO" (from `hello.h`) and "WORLD" (from `world.cc`) are both included in the preprocessed output! We can use this to separate our code into multiple files, and we'll see in the next subsection why we would want to do that.

When you use `#include <stdio.h>`, the preprocessor includes a built-in file named "stdio.h". However, that is the C name for this file. In C++, there is a new naming convention for the C headers:

```
#include <cstdio>

int main() {
    printf("Hello, C's stdio!\n");
    return 0;
}
```

In most cases, however, this backwards compatibility is unneeded, as there are C++ alternatives. For instance, we can use `iostream` instead of `cstdio` for most purposes. You should compile a file that includes `iostream` or `cstdio` with `-E` to see just how much code is included by these includes!

0:00 / 5:13



## Constant definition

Including isn't all that the C preprocessor can do, however. It can also define its own constants:

```
$ cat define.cc
#include <iostream>
using namespace std;

#define GEESE 15

int main() {
    cout << "I have " << GEESE << " geese!" << endl;
    return 0;
}

$ g++ define.cc -o define
```

```
$ ./define  
I have 15 geese!
```

This is mostly unneeded: In C++ and modern C, you should use `const` definitions instead since they are more [type-safe](#). This kind of definition is common in older C, however. More importantly, since these defines are part of the preprocessor, we can combine them with another preprocessor feature, conditional compilation.

## Conditional compilation

The `#if` directive (which ends with the `#endif` directive) is similar to C's and C++'s `if`, but since it happens in the preprocessor, it controls whether code is included at all. For instance, it is common to use a `#define` to add debugging to your code and easily remove it:

```
$ cat debug.cc  
#include <iostream>  
using namespace std;  
  
#define DEBUG_LEVEL 1  
  
int main() {  
#if DEBUG_LEVEL >= 1  
    cout << "main has started" << endl;  
#endif  
    cout << "Hello, world!" << endl;  
#if DEBUG_LEVEL >= 2  
    cout << "About to end main" << endl;  
#endif  
    return 0;  
}  
  
$ g++ -std=c++14 debug.cc -o debug  
$ ./debug  
main has started  
Hello, world!
```

We can change the amount that this code outputs just by editing the debug level:

```
$ egrep "define DEBUG_LEVEL" debug.cc
#define DEBUG_LEVEL 0
$ g++ -std=c++14 debug.cc -o debug
$ ./debug
Hello, world!
[...]
$ egrep "define DEBUG_LEVEL" debug.cc
#define DEBUG_LEVEL 2
$ g++ -std=c++14 debug.cc -o debug
$ ./debug
main has started
Hello, world!
About to end main
$
```

This is a common way to make features (not just debugging!) optional. `#if` supports `#else` and `#elif` for other conditions:

```
[...]
#if DEBUG_LEVEL > 1
    cout << "Debug mode: Disabling advanced features." << endl;
#elif FEATURE_LEVEL < 1
    cout << "Advanced features disabled by flag." << endl;
#else
    advancedFeatures();
#endif
[...]
```

Finally, an alternative to `#if` is supported, `#ifdef`, which simply checks if a preprocessor variable is set at all, rather than checking its exact value:

```
[...]
#ifndef DEBUG
    cout << "Debugging activated (I don't care what level)" << endl;
#endif
```

The opposite of `#ifdef` is `#ifndef`. It is possible to use `#elif` and `#else` with `#ifdef` and `#ifndef`.

## Preprocessor commenting

One special case of `#if` is `#if 0`. `0` is never true, so code surrounded in `#if 0` is never included at all:

```
$ cat if0.cc
#include <iostream>
using namespace std;

int main() {
#ifndef 0
    cout << "This will never output." << endl;
#endif
    cout << "Hello, world!" << endl;
    return 0;
}

$ g++ -std=c++14 if0.cc -o if0
$ ./if0
Hello, world!
$
```

This is particularly useful for commenting out large segments of code, since `#if 0` nests properly, unlike C++'s and C's multi-line comments using `/*` and `*/`.

## Preprocessor definitions in the command line

Preprocessor definitions can also be given on the command line to `g++`. The argument is `-DVAR=VALUE`, where `VAR` is the name you want to define and `VALUE` is the value you want to set. In particular, it's common to combine this with some default to make easily swappable optional features:

```
$ cat debug.cc
#include <iostream>
using namespace std;

#ifndef DEBUG_LEVEL
#define DEBUG_LEVEL 0
#endif

int main() {
#if DEBUG_LEVEL >= 1
    cout << "main has started" << endl;
#endif
    cout << "Hello, world!" << endl;
#if DEBUG_LEVEL >= 2
    cout << "About to end main" << endl;
#endif
    return 0;
}
```

```
$ g++ -std=c++14 debug.cc -o debug
$ ./debug
Hello, world!
$ g++ -std=c++14 debug.cc -DDEBUG_LEVEL=2 -o debug
$ ./debug
main has started
Hello, world!
About to end main
$
```

# Separate Compilation

Let's recall some definitions:

- **declaration**: Asserts that a function or variable exists, but does not define its content.
- **definition**: Full details on a function or variable. Defines a function's content, and allocates space for both functions and variables.

Until this point, there has been little use for declarations. However, declarations combine with the preprocessor, which we explored in the previous part, to allow us to make our code more modular. We will split our code into two components:

- **interface**: Declarations including function prototypes, with no actual code, as well as type definitions. Put in a separate file from the actual code, typically named with .h (which stands for "header").
- **implementation**: The full definition for every function, as well as space for any global variables. In this course, typically named .cc. C files are always named .c.

That file naming convention should look familiar. In fact, every time we included `stdio.h`, or any other C header, we were including interface files! Although C++'s built-in headers follow a different naming convention without .h (e.g. `iostream`), they too are interface files.

Let's look at a simple example with two implementation files and an interface file.

| main.cc  | vec.h  | vec.cc  |
|--|--|---|
| <pre>#include "vec.h" int main() {     Vec v {1,2};     v = v + v;     return 0; }</pre> | <pre>struct Vec {     int x, y; };  Vec operator+(const Vec &amp;v1, const Vec &amp;v2);</pre> | <pre>#include "vec.h"  Vec operator+(const Vec &amp;v1, const Vec &amp;v2) {     return {v1.x + v2.x, v1.y + v2.y}; }</pre> |

Note how `vec.h` contains no actual implementation. It declares the `Vec` type and the `+` operator over `Vecs`, but does not include the actual implementation of the `+` operator. As seen in the previous part, `#include` will simply include one file in another. So, the `#includes` in both `main.cc` and `vec.cc` include the *declarations* of `Vec` and its `+` operator in both of these files, without including the *definitions*. In the case of `main.cc`, this allows the `main` function to use `Vecs` without declaring them in the same file. In the case of `vec.cc`, this allows it to define the `+` operator over `Vec` without repeating the declaration of `Vec` itself. And everywhere where `vec.h` is included, this allows us to change our types or functions in only one place, and have that change propagate everywhere else automatically.

This, in fact, is exactly what happens when you use `#include <iostream>`, or `#include <cstdio>`, or any other `#include`. `iostream` includes declarations for `std::cout` (and lots more!), so including it gives your code access to those declarations. The actual definition for `std::cout` (and so much more) is in the *C++ standard library*, which is automatically used whenever you compile C++ code.

To see how that works, let's examine how we would compile `main.cc` and `vec.cc`, above. First, let's try to simply compile `main.cc`:

```
$ g++ main.cc  
/usr/bin/ld: /tmp/ccM7Ey4u.o: in function `main':  
main.cc:(.text+0x25): undefined reference to `operator+(Vec const&, Vec const&)'  
collect2: error: ld returned 1 exit status  
$
```

Well, that didn't work. Specifically, this is telling us that it couldn't find the code for the `+` operator over `Vecs`. That should be unsurprising, since that code isn't in `main.cc` at all, it's in `vec.cc`! So, let's compile `vec.cc` instead:

```
$ g++ vec.cc  
/usr/bin/ld: /usr/lib/gcc/x86_64-linux-gnu/8/../../../../x86_64-linux-gnu/Scrt1.o: in function `_start':  
.text+0x20): undefined reference to `main'  
collect2: error: ld returned 1 exit status  
$
```

That first error line might be surprising, or confusing, but the critical part is the second line: It couldn't find `main`, and we can't have a program without `main`. So, let's compile them together:

```
$ g++ main.cc vec.cc -o vecs  
$
```

Much better! This technique works, but can turn quite cumbersome when your program is large. Some large projects take *hours* to compile. The real power of separating our code like this is a technique called **separate compilation**, by which we can compile only one part of our whole program at any time:

```
$ g++ -c main.cc  
$ g++ -c vec.cc  
$ g++ main.o vec.o -o vecs  
$
```

The `-c` option to `g++` requests that a `.cc` file be compiled into a `.o` file, or an **object file**. An object file contains compiled code, but not enough for a full program; in this case, `main.o` contains `main` (from `main.cc`), and `vec.o` contains the `+` operator for `Vecs` (from `vec.cc`). You can use `.o` files like `.cc` files when outputting a program, as in the third line above. Combining multiple `.o` files to create a program is called **linking**. Since `.h` files contain no code, only declarations, they don't need to be (and shouldn't be!) compiled at all!

Importantly, linking is much faster than compiling. Thus, separate compilation lets us make small changes to a large code base much faster, by only recompiling the changed part.

For instance, if we changed `main.cc`, we would only need to recompile `main.cc`; if `vec.cc` doesn't change, then `vec.o` is fine as is:

```
[... change main.cc ...]  
$ g++ -c main.cc  
$ g++ main.o vec.o -o vecs  
$
```

Similarly, if we only changed `vec.cc`, then only `vec.cc` would need to be recompiled.

But what if we need to change the declarations? What if we need to change `vec.h`? Since `main.cc` and `vec.cc` both include `vec.h`, and inclusion just textually includes one file in another, if we update `vec.h`, then we need to recompile both `main.cc` and `vec.cc`.

## Global Variables

Now let's look at global variables. Consider adding a global variable `origin`, and a function `addToOrigin` to our program. We want `origin` to be visible to both `main.cc` and `vec.cc`. Our first attempt will be to declare the variable in `vec.h`, but this will not work:

| main.cc   | vec.h (wrong)   | vec.cc (wrong)   |
|---|---|--|
| <pre>#include "vec.h"<br/><br/>int main() {<br/>    Vec v {1,2};<br/>    origin.x = 12;<br/>    v = addToOrigin(v);<br/>    return 0;<br/>}</pre> | <pre>struct Vec {<br/>    int x, y;<br/>};<br/><br/>Vec origin {0, 0};<br/><br/>Vec operator+(const Vec &amp;v1, const Vec &amp;v2);<br/>Vec addToOrigin(const Vec &amp;v);</pre> | <pre>#include "vec.h"<br/><br/>Vec operator+(const Vec &amp;v1, const Vec &amp;v2) {<br/>    return {v1.x + v2.x, v1.y + v2.y};<br/>}<br/><br/>Vec addToOrigin(const Vec &amp;v) {</pre> |

```
    return v + origin;  
}
```

Let's try to compile this:

```
$ g++ -c main.cc  
$ g++ -c vec.cc  
$ g++ main.o vec.o -o vecs  
/usr/bin/ld: vec.o:(.bss+0x0): multiple definition of `origin'; main.o:(.bss+0x0): first defined here  
collect2: error: ld returned 1 exit status  
$
```

We get the error "multiple definition of `origin`". This is because *defining* a variable allocates space for it. Since that definition was included (through `#include`) in both `main.o` and `vec.o`, we've defined the space twice, and the compiler is unsure which to use. To write a declaration of a global variable, instead of a definition, we need to use a new keyword, `extern`. However, even that is **not enough**:

| main.cc   | vec.h   | vec.cc (wrong)  |
|---|---|---|
| <pre>#include "vec.h"<br/><br/>int main() {<br/>    Vec v {1,2};<br/>    origin.x = 12;<br/>    v = addToOrigin(v);<br/>    return 0;<br/>}</pre> | <pre>struct Vec {<br/>    int x, y;<br/>};<br/><br/>extern Vec origin;<br/><br/>Vec operator+(const Vec &amp;v1, const Vec &amp;v2);<br/>Vec addToOrigin(const Vec &amp;v);</pre> | <pre>#include "vec.h"<br/><br/>Vec operator+(const Vec &amp;v1, const Vec &amp;v2) {<br/>    return {v1.x + v2.x, v1.y + v2.y};<br/>}<br/><br/>Vec addToOrigin(const Vec &amp;v) {<br/>    return v + origin;<br/>}</pre> |

Note that the `extern` declaration has no initializer. It can't have one, since it doesn't actually allocate the variable, and so can't give it a value. This version invokes a new error:

```
$ g++ -c main.cc  
$ g++ -c vec.cc  
$ g++ main.o vec.o -o vecs  
/usr/bin/ld: main.o: in function `main':
```

```

main.cc:(.text+0x18): undefined reference to `origin'
/usr/bin/ld: vec.o: in function `addToOrigin(Vec const&)':
vec.cc:(.text+0x5c): undefined reference to `origin'
collect2: error: ld returned 1 exit status
$
```

This time, we have "undefined reference to origin". `extern` is *only* a declaration, so now, we haven't allocated storage for the variable at all. In order to make `extern` work, we still need the actual variable declaration, but we must put it in a `.cc` file, so that it is not allocated multiple times:

| main.cc   | vec.h  | vec.cc   |
|---|--|--|
| <pre>#include "vec.h"  int main() {     Vec v {1,2};     origin.x = 12;     v = addToOrigin(v);     return 0; }</pre> | <pre>struct Vec {     int x, y; };  <b>extern</b> Vec origin;  Vec operator+(const Vec &amp;v1, const Vec &amp;v2); Vec addToOrigin(const Vec &amp;v);</pre> | <pre>#include "vec.h"  <b>Vec origin {0, 0};</b>  Vec operator+(const Vec &amp;v1, const Vec &amp;v2) {     return {v1.x + v2.x, v1.y + v2.y}; }  Vec addToOrigin(const Vec &amp;v) {     return v + origin; }</pre> |

Finally, this version will actually compile:

```

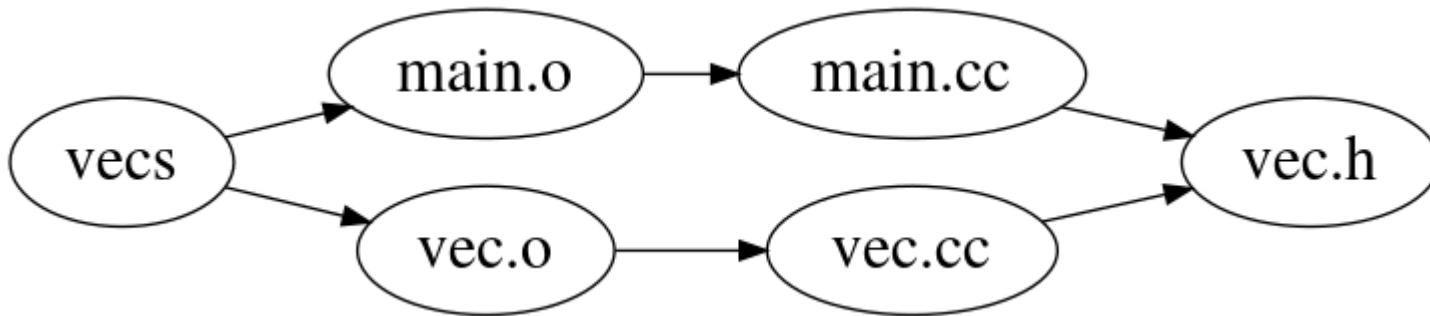
$ g++ -c main.cc
$ g++ -c vec.cc
$ g++ main.o vec.o -o vecs
$
```

This pattern is the same for all global variables which are used in multiple files. Since the header file can only have declarations, we use `extern` there, but the actual variable must still be defined and allocated in an implementation file.

This separation of declarations from definitions creates a system of **dependencies**: Our program is linked from `main.o` and `vec.o`, so it depends on both of those files. `main.o` is compiled from `main.cc`, so `main.o` depends on `main.cc`. `main.cc` includes `vec.h`, so `main.cc` depends on `vec.h`. If `x` depends on `y`, and `y` changes, then `x` needs to be recompiled or relinked.

# Make and Makefiles

In the previous example, the relationship between our files created a system of **dependencies**, and those dependencies were making compiling our program a bit complicated.



Specifically, our program (`vecs`) depends on `main.o` and `vec.o`. `main.o` depends on `main.cc`, and `vec.o` depends on `vec.cc`. Both `main.cc` and `vec.cc` depend on `vec.h`. Typing out each command to separately compile these parts, as well as remembering what depends on what so we know which command we need, can get tedious. Luckily, a tool exists to automate this tedium, by combining the list of commands with the dependency relationship: `make`.

Here's a simple Makefile for our project above:

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

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
```

This must be placed in a file named "Makefile". (By default, `make` looks for a file named either `Makefile` or `makefile`. We suggest that you use `Makefile` so that the file will appear near the beginning of any directory listing.)

# Targets and Recipes

The lines not indented here are *dependencies*. They show that the file before the colon depends on the file(s) after the colon. Each file is a **target**, that is, something this Makefile describes how to create. Since we don't describe how to *create* `main.cc` or `vec.cc`, we include their dependencies (that is, `vec.h`) along with `main.o` and `vec.o`. The targets for this Makefile are `vecs`, `main.o`, and `vec.o`.

**The commands to create a target must be indented with tabs, even if you normally indent with spaces, so be careful!**

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, you can include any valid shell command in the recipes.

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

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

`make` automatically discovered 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:

```
[... 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 you don't 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:

```
$ make main.o
make: 'main.o' is up to date.
[... change vec.h ...]
$ make main.o
g++ -std=c++14 -c main.cc
$
```

## Phony Targets

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

```
$ cat Makefile
vecs: main.o vec.o
    g++ main.o vec.o -o vecs

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: clean

clean:
    rm *.o vecs

$ make clean
rm *.o vecs
$
```

## Make Variables

Two of our recipes are essentially identical. We can reduce the clutter, as well as making our Makefile more flexible, by using **make variables**:

```
$ cat Makefile
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}

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

```
g++ main.o vec.o -o vecs  
$
```

As a bonus, make includes some **default recipes**, including a recipe for compiling .cc files into .o files, which uses the pre-defined make variables CXX and CXXFLAGS, so we didn't have to include those recipes here.

## Automatic Dependency Management

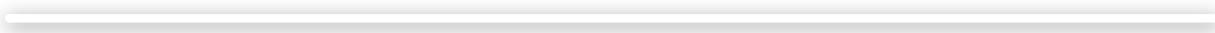
There's one last problem we can deal with: keeping the dependencies between .cc files and .h files up to date. It's easy to forget to update the Makefile every time you add (or remove!) a dependency from a .cc file. Luckily, g++ is capable of generating this dependency information for us! The -MMD flag 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:

```
$ cat Makefile  
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}  
  
$ make  
g++ -std=c++14 -MMD -c -o main.o main.cc  
g++ -std=c++14 -MMD -c -o vec.o vec.cc
```

```
g++ main.o vec.o -o vecs
$ cat main.d
main.o: main.cc vec.h
$ cat vec.d
vec.o: vec.cc vec.h
$
[... edit vec.h ...]
$ make
g++ -std=c++14 -MMD -c -o main.o main.cc
g++ -std=c++14 -MMD -c -o vec.o vec.cc
g++ main.o vec.o -o vecs
$
```

make is capable of much more than this, but this is sufficient to build most normal C++ projects.

0:00 / 7:33



# Preprocessor guards

Because we want to use separate compilation to reduce the amount of time we need to recompile by only recompiling files that have changed, or where the included header file has changed, we are going to find ourselves with many header files, some of which may be included in multiple locations by multiple files.

For example, suppose we want to write a linear algebra module.

```
// linalg.h
#include "vec.h"
. . .

// linalg.cc
#include "linalg.h"
#include "vec.h"
. . .

// main.cc
#include "vec.h"
#include "linalg.h"
. . .
```

By using separate compilation, `linalg.cc` and `vec.cc` should be compiled into the corresponding `linalg.o` and `vec.o`. However, when we try to compile `linalg.cc` or `main.cc`, since they include the header files for both `linalg.h` and `vec.h`, the compilation fails since there are now two copies of the definitions in `vec.h`:

```
$ g++ -std=c++14 -c -o linalg.o linalg.cc
In file included from linalg.h:1:0,
                 from linalg.cc:2:
vec.h:1:8: error: redefinition of ‘struct Vec’
 struct Vec {
           ^
           ~~~
In file included from linalg.cc:1:0:
```

```
vec.h:1:8: note: previous definition of 'struct Vec'  
struct Vec {  
    ^~~
```

(This version of the code can be found in your repository under the directory `lectures/c++/05-separate/example3`)

The proper way to correct the situation is to put **preprocessor guards**, also called **include guards**, into every header file. The general format of an include guard is:

```
#ifndef SOME_UNIQUE_MACRO_NAME_H  
#define SOME_UNIQUE_MACRO_NAME_H  
...  
#endif
```

The preprocessor keyword `#ifndef` checks to see if the following string (`SOME_UNIQUE_MACRO_NAME_H`) is defined or not. If it is, everything up to the `#endif` is omitted by the preprocessor, which means that the compiler won't see it. If the string *isn't* defined, then we use the preprocessor keyword `#define` to define it. This set of steps is guaranteed to ensure that whatever code is wrapped in this set of statements will only be seen once by the compiler. (Some compilers can accomplish the same thing by using the single statement `#pragma once` at the top of the file, but it's not part of the standard and therefore isn't guaranteed to always work. The original approach described above will work in all circumstances.)

Now, what do we use for the macro name? It has to be unique across all of the included system libraries as well as any of your own code. By common convention, it's usually in all capital letters, with underscores separating any words. Since the class names need to be unique, people often use that, sometimes in combination with numbers. You cannot use reserved words such as `NULL`, and it's generally suggested that you avoid names of the form `_A` i.e. an underscore followed by a capital letter or names that use two consecutive underscores. If you end up conflicting with a reserved name or library macro name, you will cause very strange behaviour!

The corrected version of the code can be found in your repository under the directory `lectures/c++/05-separate/example4`. Note that the revised `vec.h` header file now looks like this:

```
#ifndef _VECTOR_H
#define _VECTOR_H

struct Vec {
    int x;
    int y;
};

Vec operator+(const Vec &v1, const Vec &v2);

#endif
```

While it's not strictly speaking necessary to also put the include guard in `linalg.h`, it is considered good practice to always put include guards in your header file so that you don't have to remember to add them later.

Note that use of the `#pragma once` directive is *non-standard* i.e. it isn't supported by every compiler, so if you want your code to be truly portable across every platform, you should use the standard form of the include guards. Do not use it in this course.

# Final Remarks

As we saw before, we can use the following directive to make the compiler automatically resolve the namespace `std::` without us having to type it everywhere:

```
using namespace std;
```

However, you should never do that in a header (.h) file. Why? When someone includes a .h file, all the contents of the file are copied by the preprocessor into the place where it is included. This means that the line `using namespace std;` will also be copied. In effect, you will be forcing anyone that includes your .h file to use this directive even if they did not want to (and may be unaware of it).

**Never write `using namespace std;` in a header (.h) file. Always prefix each type with `std::` in these files.**

Using this directive in an implementation (.cc) file is ok because they are never supposed to be included. So, the directive will only have an effect within that file.

Also:

**Never compile .h files! Header files are to be included only, never compiled.**

**Never include .cc files! Implementation files are to be compiled only, never included.**

# Coupling and Cohesion

Before we explain what classes are, let's explain why we care.

There are two important software design quality measurements, **cohesion** and **coupling**.

**Cohesion** measures the amount of "relatedness" that a module or unit of code contains. (This **module** can be a library package, a file, or as we'll later see, a *class*.) Ideally, we want there to be a "high degree" of cohesion, i.e., everything in the unit is very closely related and serves a single purpose. For example, in C the stdio library contains functions for input and output. So, it's highly cohesive. If I added functions from the C math library, it would have a low degree of cohesion since there's little conceptually in common between I/O and math. The more unrelated purposes a unit serves, the more often it may have to change as I maintain the code. This makes maintenance more difficult and increases the chances of introducing errors.

**Coupling** measures the amount of dependency *between* units/modules. The more dependency between modules, the more often a change in one will cause a change in another. Ideally, therefore, we want the amount of coupling between modules to be low. For example, if I define a Node type that is used by three different units, any change to Node's implementation will cause all three of those units to have to be changed as well. If I can limit the amount of change required, this not only reduces the workload but also reduces the chance for introducing errors.

**Design your code to maximize cohesion and minimize coupling!**

We'll revisit these concepts in [Coupling and Cohesion Revisited](#) (available later in the term).

# Procedural vs Object-Oriented Programming

The style of programming you have been doing so far in C/C++ is called **procedural programming**. It has this name because the code is organized in *procedures* that implement the logic of the program (which in C/C++ are the *functions*) and variables that hold the data. Each variable can hold a single piece of data (e.g., a number, text, date, etc.) or be a structure (a struct in C/C++) that holds complex data with subparts (fields). The variables are passed to the procedures (functions) via parameters or are global variables.

In **object-oriented programming** (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 functions**.

To create objects, we must first define classes. **Classes** are blueprints or type specifications that describe the contents of an object. It is similar to how in C you had to first define a struct type, describing which fields the struct would have, and only after that, you could create an instance of that struct (at which point memory is allocated to store the data fields of the struct). Thus, in OOP, we first need to define a class, which describes the attributes and methods that each object of that type can have. Only after that, we can create objects, which are instances of a class (at which point memory is allocated to store the attributes of the object).

In general, a program that can be written in the procedural style could also be written in the object-oriented style and vice-versa. However, as programs grow in size, it is generally easier to maintain high cohesion and low coupling (which are characteristics of well-designed code) using OOP. This is because when they are designed correctly, classes/objects naturally have high cohesion (as they combine the data and logic related to that data into a single unit) and help decrease the coupling between modules (as you can generally use a class based on its interface without knowing its exact implementation).

These concepts are not specific to C++. Classes and objects are present in all object-oriented languages. Although the precise implementation is sometimes a bit different from language to language, these general OOP concepts are valid for all languages. In the next lessons, you will at the same time be learning the general OOP concepts and the way they work in C++. The OOP concepts and best practices that you will learn can be transferred to any other object-oriented language even though its syntax to implement classes and objects may somewhat differ from the C++ syntax.

# Classes

We know that we can keep related pieces of information together in C/C++ by putting them into a structure, using the `struct` keyword. This improves cohesion. There is, however, another way that C++ lets you improve cohesion that cannot be done in C; we can also put [functions](#) inside of a [structure](#). This lets us keep related data and functions (now called **operations** or **methods** or **member functions**) in the same data structure, which is called a **class**. (We'll see the C++ `class` keyword shortly, in [The class Keyword](#). For now, we're just going to use the `struct` keyword since it provides similar functionality.) A class is a type. An *instance* of a class is called an **object**.

Let's take a look at an example:

| File: student.h   | Explanation   |
|---|---|
| #ifndef _STUDENT_H_<br>#define _STUDENT_H_<br><br>struct Student {<br>int assns, mt, final;<br><br>float grade();<br>};<br><br>#endif | Here we can see that the <code>Student</code> class has three data fields used to hold marks respectively for assignments ( <code>assns</code> ), the midterm ( <code>mt</code> ), and the final exam ( <code>final</code> ). It has a single method, <code>grade</code> , that takes no parameters and returns the final grade in the course. In general, we want to try and avoid giving the client/user too much information about our class and its functionality, which is one of the reasons we only give the function signature in the header file. Ideally, we'd give the client our header file and the <code>student.cc</code> file, which they could then link in with their code, and they'd never need to know how the final course grade was actually calculated! |
| File: student.cc  | Explanation   |
| #include "student.h"  | Note that in the implementation file for the <code>Student</code> , we  |

```

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

```

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. (We've seen this used before to specify that something was part of the std namespace (`std::`). Here we're using it to specify that for some class C, f is in the context of C, by specifying `C::f`.) Since grade is part of Student, and Student is a type, these fields don't actually exist until we create an instance of Student. So, to what fields is grade referring? It uses the fields of the object that invokes 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.

#### File: main.cc client code

```

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

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

```

#### Explanation

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, i.e., the 60 gets stored in `assns`, the 70 gets stored in `mt`, and the 80 gets stored in `final`.) It then calls the `grade` method on the object s, and outputs the calculated final grade.

Note that the method call is being applied to a specific instance of the class. If I create another student and call `grade` on it, as in:

```

Student mohammad{ 80, 76, 98 };
mohammad.grade();

```

then grade is using the data fields of mohammad. 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 object upon which the method is being invoked. So, in the previous example, the value of this inside of grade is the address of mohammad. When you access a class field within a class method, it automatically uses this, so we could have written the implementation of grade as:

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

In practice, this implementation is identical to the one provided in the table above. While we omitted this-> when accessing the fields of the Student object in the example shown in the table, it is automatically added by the compiler, so the actual compiled code will be the same as the example above.

You can find the full code example in [lectures/c++/06-classes/01-constructors/student\\_v1.cc](#).

**Warning: Some people feel very strongly about how this should and shouldn't be used. In particular, some people really dislike it when programmers use this-> unnecessarily and feel it should only ever be used when it is needed to disambiguate parameter or method calls since they feel it clutters up the code; otherwise, its use should be avoided.**

**In this course, we're going to ask you to *only use it when absolutely necessary*.**

# Initializing objects

Our previous Student example used an **initialization list** to copy the data values into the object's data fields.

```
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 a **constructor**, used exactly for this purpose. The general format of the constructor (often abbreviated to *ctor*) is:

```
Name-of-Class-Type( parameter-List ) {  
    // necessary code  
}
```

## Notes:

- There is no return type, since you're building the actual object. The constructor does not return the object it builds.
- The method name is the type name.
- The parameter list may be empty.
- Constructors can be overloaded!
- The method body may be empty, but must be present.

When separated into header and implementation files, our Student example becomes:

|                        |                                 |
|------------------------|---------------------------------|
| Header file: student.h | Implementation file: student.cc |
|------------------------|---------------------------------|

```

#ifndef _STUDENT_H_
#define _STUDENT_H_

struct Student {
    int assns, mt, final;
    Student( int assns, int mt, int final );
    float grade();
};

#endif

```

```

#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;
}

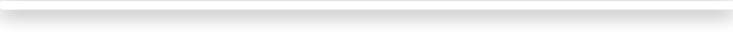
```

You'll notice that we used the `this->` notation in the body of our constructor. We had to use it there since our parameter names are the same as our data field names.

If I don't use `this`, the compiler assumes that I want to use the local parameters, and will assign them to themselves, i.e., `assns` to `assns`, `mt` to `mt`, and `final` to `final`. This leaves my data fields uninitialized (or at least, not initialized with the values I want). The same problem occurs if you initialize local variables rather than the object's data fields.

What happens if we try to create a student object without using the constructor, e.g., `Student s;`? (See the code in `lectures/c++/06-classes/01-constructors/student_v2.`)

0:00 / 3:57



From before, we know that if we create a student object and didn't have a constructor as in the example below:

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

the compiler used the values in braces to initialize the student's data fields. With a constructor, the same syntax calls the constructor and passes the values in as the method's parameters. It turns out that I can also initialize an object in C++ using the assignment operator, operator=, as in:

```
Student s = Student{ 60, 70, 80 };
int x = 5;
std::string word = "hello";
```

And in some circumstances, I can use ( ) to initialize variables, especially when invoking constructors. For example:

```
Student s( 60, 70, 80 );
int x( 5 );
std::string word( "hello" );
```

The rules in C++ as to when we can use = and when we can use ( ) are rather arbitrary. C++ now has **uniform initialization syntax** using {}, which is meant to be used in nearly all initialization situations.

**Try to get into the habit of using uniform initialization syntax. It is the preferred syntax in modern C++ and will work more consistently, more often than the other forms.**

Our initialization code thus becomes:

```
Student s{ 60, 70, 80 };
int x{ 5 };
std::string word{ "hello" };
Node * ptr = new Node{ 10, nullptr };
```

Since we can overload methods in our classes, this means that we can overload our constructors. It then becomes possible to use default parameters on our constructors. (Note that these only go on in the interface, and **not** on the implementation!) So, our constructor in lectures/c++/06-classes/01-constructors/student\_v3 becomes:

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

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

and our implementation doesn't need to change! This way, at the beginning of the term, we can create students who don't have any marks, or who have just the assignments grade, or the assignments and midterm, or all three grades.

```
Student s1{ 60, 70, 80 }, s2, s3{60}, s4{60, 70};
```

**Constructor advantages:**

1. Allows default parameters.
2. Allows overloading.
3. Allows sanity checks.
4. Allows code other than simple field initialization.

# 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 has 0 parameters. That's it.

If you do not provide a constructor of *any sort* to your class (0 or more parameters), then the compiler automatically gives you a default constructor. All it will do is call the default constructor on any data fields in your 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 soon as you declare (and define) any other constructor. Consider the following example:

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

If you remove the constructor, then v2 will become valid and v1 invalid. To make both valid, you would need to write two constructors.

# 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 definition. For example, see `lectures/c++/06-classes/01-constructors/constRef.cc`:

```
int z;

struct MyStruct{
    const int myConst = 5;
    int & myRef = z;
};

MyStruct m1, m2;
```

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 example, a student's id number should be a constant, since it doesn't 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 *have* to be initialized when their space is allocated. If the constructor body is too late, what other choice do we have? Let's take a look at the steps involved in creating an object:

1. allocate space
2. construct the data fields
3. 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.

## Member initialization lists (MIL)

The member initialization list code must be placed as part of 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. So, our previous Student class example becomes:

| Header file: student.h  | Implementation file: student.cc   |
|---|---|
| <pre>#ifndef _STUDENT_H_ #define _STUDENT_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</pre> | <pre>#include "student.h"  int capGrade( int grade ) {     if ( grade &lt; 0 ) return 0;     if ( grade &gt; 100 ) return 100;     return grade; }  Student::Student( const int id, 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; }</pre> |

(A copy of this version can be found in your repository as lectures/c++/06-classes/01-constructors/student\_v4.)

**Notes:**

- The name *outside* of the " {} " in the MIL is the data field, while the name *inside* of the " {} " is the parameter name.
- The MIL can be used to initialize other data fields, not just constants or references.
- 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!
- Using the MIL can be more efficient than initializing in the body of the constructor. This is particularly true when the data field is an object that has a default constructor. The object will be first initialized with the default constructor when it is created, and then initialized again in the body of the constructor using the assignment operator. In other words, you've just initialized it twice!

Consider the following example:

```
struct Student {
    string name;
    Student( const string & name ) {
        this->name = name;
    }
};
```

The data field `name` is initialized with the `std::string` default constructor, and then reassigned in the constructor body.

- MIL initialization takes precedence over initialization in the class. Consider the following example, `lectures/c++/05-classes/01-constructors/vecDef.cc`:

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

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

In this case, the *initialization in the MIL takes precedence*. So, if we declare a vector object v as `Vec v{5};`, the data field x will contain the value 5 but the data field y will contain the value 0. Note that we now also have to explicitly overload the constructors to allow for all three possible ways of initializing the vector object. It would be considerably shorter and more compact to use default parameters in this case!

**Note, however, that the 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 you need sophisticated logic, such as loops or conditions, to initialize your fields, use the body, but otherwise use the MIL.

# Copy constructors

We know from before that if we don't provide any constructors, the compiler gives us a default constructor. It turns out that there are several other special methods that the compiler gives us unless you provide your own. (The rules are actually a little more complicated than that, but we'll discuss them shortly.) They consist of:

1. default constructor: calls the default constructor on all fields that are objects
2. copy constructor: copies all fields from the object passed in
3. copy assignment operator: copies all fields from the object passed in
4. destructor: does nothing by default
5. move constructor: takes data from the object passed in
6. 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 case 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 int id, int assns = 0, int mt = 0, int final = 0 )  
        : id{id}, assns{assns}, mt{mt}, final{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} {}  
};
```

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

When is the built-in copy constructor not correct? Consider the example shown in lectures/c++/05-classes/01-constructors/linkedList.cc:

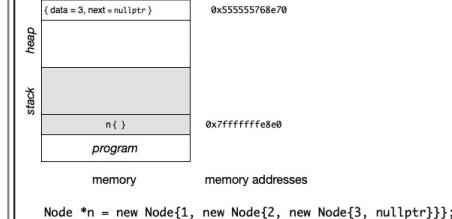
```
struct Node {  
    int data;  
    Node * next;  
  
    Node( int data = 0, Node * next = nullptr ) : data{data}, next{next} {}  
};  
  
ostream &operator<<(ostream &out, const Node &n) {  
    out << n.data;  
    if (n.next) {  
        out << ",";  
        out << *(n.next);  
    }  
    return out;  
}
```

where we have the case of an existing small linked list, pointed to by the local variable n.

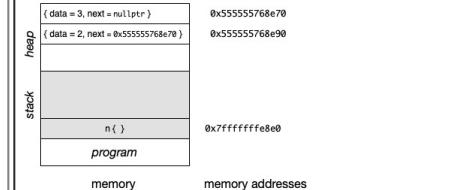
| Code  | Explanation   |
|---|---|
| Node *n = new Node{1, new Node{2, new Node{3, nullptr}}}; | We start by creating a singly-linked list. Each node is |

"anonymous" and allocated on the heap, but we don't lose any of them since the address of the one most recently created is passed in as the value for the `next` field in the node just ahead of it in the list.

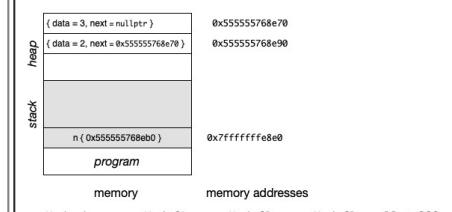
```
Node *n = new Node{1, new Node{2, new Node{3, nullptr}}};
```



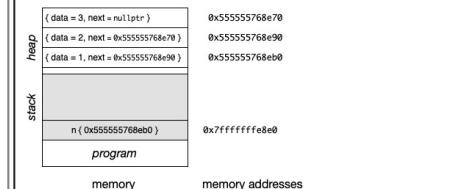
```
Node *n = new Node{1, new Node{2, new Node{3, nullptr}}};
```



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

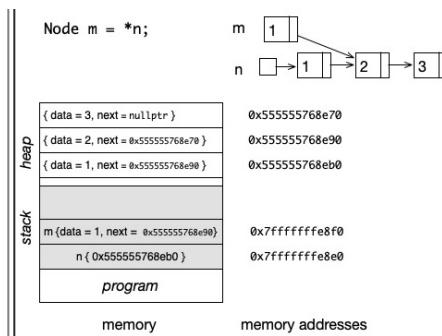


```
Node *n = new Node{1, new Node{2, new Node{3, nullptr}}};
```



```
Node m = *n; // copy ctor; identical to "Node m{*n};"
```

Since we're dereferencing the pointer `n`, we're trying to copy a node to a node. Despite the fact that there's an assignment operator in this line, *since it is part of a declaration for a Node m, this actually invokes a copy constructor!*

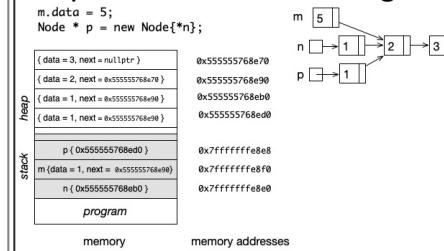


Note that both `m` and `n` are on the stack. However, `n` points to heap memory, as does `m`. `next`.

```
m.data = 5;

Node *p = new Node {*n};
```

Let's change the value in `m`'s data field to 5. And then allocate a new node pointer `p` on the stack, which will point to a newly-allocated node on the heap. That node will be initialized by using the `Node` copy constructor where a dereferenced `n` is passed in as the parameter we're calling other.



```
p->data = 6;

cout << "n: " << *n << endl;
cout << "m: " << m << endl;
cout << "p: " << *p << endl;
cout << endl;

n->next->next->data = 7;

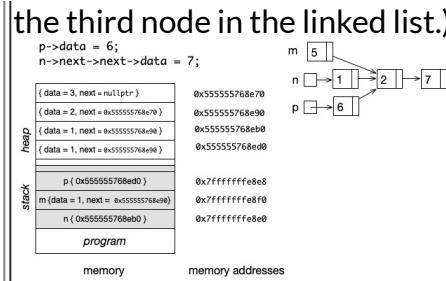
cout << "n: " << *n << endl;
cout << "m: " << m << endl;
cout << "p: " << *p << endl;
cout << endl;
```

We then modify the data field in the node pointed to by `p`. When we output the three lists, we see:

```
n: 1,2,3
m: 5,2,3
p: 6,2,3
```

So, from the outside, it looks as if we have three separate lists.

Let's change the data field in the third node to which `n` points. (Remember, `n->` gets us to the first node in the linked list; `n->next->` gets us to the second node in the linked list; and `n->next->next->` gets us to



When we output the three lists now, we see:

n: 1,2,7  
m: 5,2,7  
p: 6,2,7

Now all three lists are affected by the single change!

So, what happened? Since the default copy constructor only copies the contents of the data field, it copied the existing addresses. We ended up only making a copy of the *first* node and made a **shallow copy** of the rest. If we'd wanted the lists to be independent of each other, we should have made a **deep copy** instead.

**Moral of the story: if you have a dynamically-allocated data structure and want a deep copy, write your own copy constructor**

So, how do you write a copy constructor that performs a deep copy? Here's one approach:

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

Your repository shows an alternative approach in lectures/c++/05-classes/01-constructors/linkedListDeep.cc that uses the C/C++ [conditional operator, also called the ternary operator, ?](#) in combination with the MIL:

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

In particular, n.next is initialized with the value returned by ?, which is either nullptr if n.next == nullptr, or the address of a newly-allocated node on the heap, which is initialized using the copy constructor applied to the dereferenced n.next. By doing this, if we re-did our previous example using the new copy constructor, m is a Node on the stack, and its next field is pointing to a new Node on the heap, initialized with a copy of the original list's second node. When we copy the second node, we end up making a deep copy of the rest of the list, which means that m->next->next is pointing to a new Node on the heap, initialized with the deep copy of n->next->next. That new Node ends up terminating with a nullptr in its next field. Now when we change n->next->next->data to 7, it will only affect the original list.

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 will see exceptions to all three of these cases when we discuss [move/copy ellision](#); but, for now, this is a reasonable approximation.

# The Peril of single-argument constructors

There's one type of constructor with which you need to be careful—single-argument constructors. By definition, they're used to **implicitly convert** an argument of the specified type into an object of the constructor type.

For example, I can extend my Node class definition by saying that:

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

(We've already done this through the use of our default parameters.) However, it now means that I can write code such as:

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

This is fine so long as you're aware of what is happening, and intend for it happen. The problem is that the compiler is silently performing the conversion for you rather than warning you of a potential error. Thus it could happen unintentionally, causing your program to behave strangely.

**Disable implicit conversion by labelling all single-argument constructors with the `explicit` keyword.**

An example of this is shown in `lectures/c++/06-classes/01-constructors/linkedListSingleArg.cc`

```
struct Node {  
    int data;
```

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

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

Now, if we attempt to create a node using implicit conversion:

```
Node myNode = 4;
```

the compiler will produce the error message:

```
linkedListSingleArg.cc: In function "int main()":
linkedListSingleArg.cc:23:17: error: conversion from "int" to non-scalar type "Node" requested
  Node myNode = 4;
                  ^
```

The proper way to create such a node would be via:

```
Node myNode = Node{4};
```

# Destructors

The compiler implicitly provides a **destructor** for your class unless you write your own. Since the compiler knows nothing about your code or your 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:

1. Run the body of the destructor.
2. Invoke destructors for the object's data fields that are themselves objects. This will occur in reverse declaration order.
3. Deallocate the space associated with the object.

The signature for a class destructor follows the format:

```
~cClass-name();
```

It takes no parameters and returns nothing.

So, why do we want a destructor? Used properly, it can help your program avoid memory leaks! For example, the singly-linked list of integer code that we've been using so far has leaked 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's take a look at how we'd implement this. One possible approach would be to write the destructor as:

```
struct Node {  
    ...  
    ~Node() {  
        if ( next != nullptr ) {  
            delete next;  
        }  
    }  
};
```

However, we can simplify this even further by recognizing that:

1. deleting a nullptr does nothing, and
2. the data structure is recursive.

Thus, our revised implementation becomes:

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

...
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. Since it's an object, that automatically invokes the destructor for *that* Node. And repeat until the list is completely freed.

The final version can be found in your repository, under `lectures/c++/06-classes/02-destructors`. Let's examine how we'd separate the header from the implementation when it comes to the destructor:

| File: node.h   | File: node.cc   |
|--|---|
| <pre>#ifndef __NODE_H__ #define __NODE_H__ #include &lt;iostream&gt;  struct Node {     int data;     Node *next;     Node(int data, Node *next);     Node(const Node &amp;n);     explicit Node(int n);     ~Node(); };  std::ostream &amp;operator&lt;&lt;(std::ostream &amp;out, const Node &amp;n); #endif</pre> | <pre>#include "node.h"  Node::Node(int data, Node *next): data{data}, next{next} {}  Node::Node(const Node &amp;n)     : data{n.data},       next{n.next ? new Node{*n.next} : nullptr} {}  Node::Node(int n): data{n}, next{nullptr} {}  Node::~Node() {     delete next; }  std::ostream &amp;operator&lt;&lt;(std::ostream &amp;out, const Node &amp;n) {     out &lt;&lt; n.data;</pre> |

```
    if (n.next) {  
        out << "," << *n.next;  
    }  
    return out;  
}
```

Note the placement of the class scope resolution operator and where the tilde ("~") goes in relation to it.

# 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's look at a quick example using our previous Student class.

```
Student bill{ 60, 70, 80 };
Student jane{ bill }; // copy ctor used
Student fatima{ 78, 81, 76 };
bill = fatima; // copy, but not copy ctor; copy assignment operator
```

We're 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. And just like our copy constructor, there may be times when we instead want to perform a deep copy. 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 right-hand side of the assignment. It returns a reference of the class type, since it needs to be able to modify the object on the left-hand side of the assignment if it's part of a cascade/sequence of assignment operations. For example:

```
jane = bill = fatima;
```

would copy the contents of the `fatima` object into the `bill` object, then copy the contents of the `bill` object into the `jane` object.

Thus, our general signature format will be:

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

An obvious case where we could use this is in our previous Node class for the singly-linked list. We're going to work through several incorrect versions, discussing common mistakes, before we get to our final version.

| Code       | Explanation |
|------------|-------------|
| Version 1: |             |

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

Since our data field is an integer, we can just copy over the value. We perform a deep copy in exactly the same way as previously seen on the next field. We need to return the reference to the modified object. Remember that all object methods implicitly have a hidden first parameter, `this`. We can use that pointer since it contains the address of the object on which this method is being called; however, it's a pointer, and we need a reference, so we'll dereference it with the `*`.

However, this isn't a new object being created! We've over-written the old address in `this->next`, and haven't freed whatever was there. So, we've now introduced a memory leak.

#### Version 2:

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

Well, we can fix our memory leak by just inserting a call to `delete next`, right? But because we're working with pointers, it's entirely possible (though not very common) that we could end up with two pointers to the same `Node` when we try to perform the assignment i.e.

```
Node * p1, *p2;  
...  
*p1 = *p2;
```

And we've just freed the list that both p2 and p1 point to before we attempt the deep copy. Depending upon the state of things, this might work, or we might get a segmentation fault. This problem is called **self-assignment**, and we can fix things by introducing a check for it. (For more detailed information on self-assignment and why it's a problem, see <https://isocpp.org/wiki/faq/assignment-operators>.)

### Version 3:

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

We already know that this contains the address of the object on the left-hand-side of the operation. We can thus just check to see if this == &other. If that's the case, we're in a self-assignment situation, in which case there's nothing to do and we can just return the original, unmodified object.

So, we're done, right? Well, what if the heap ran out of memory when we tried to perform a deep copy? It's not very likely, but it's possible. We don't yet have the

tools to use the preferred mechanism for dealing with this, since that requires exception handling; but, we'll get there in [Introduction to Exceptions](#). For now, we'll ignore the problem since it will currently cause our program to terminate.

Now, we've failed at some point in the deep copy, but have already freed the original contents of `this`. We can improve the situation by changing the order of our actions around a bit.

#### Version 4:

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

This is actually a pretty clever approach. We make a deep copy of the linked list (from its `next` field down, and our current node pointed to by `this` will contain the "head" information of `other`), with the head of the new list stored in a temporary pointer. If the heap allocation fails, and the function terminates, our original node's data fields haven't been changed. Only if we've successfully completed the deep copy will we free the old list that `next` pointed to, before over-writing it with the address in `tmp`.

**Warning: Do not delete the address in tmp! That will destroy the copy of the list that you've just created!**

The final version of the program can be found in your repository, in `lectures/c++/06-classes/03-assign`.

## Copy-swap idiom

An alternative approach is to use the [copy-and-swap idiom](#). (There's an interesting article on it at [https://mropert.github.io/2019/01/07/copy\\_swap\\_20\\_years/](https://mropert.github.io/2019/01/07/copy_swap_20_years/).) 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, 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'll use the `std::swap` routine from the utility library to help write our own swap routine.

```
#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.

This program can be found in your repository, in `lectures/c++/06-classes/03-assign/copy-swap`.

# Move semantics

## Introduction

Before we can cover our last two operations, we need to define what we mean by **move semantics**. In order to do that, we need a quick refresher on **lvalues** and **rvalues**

If you can take the address of an expression, it is an lvalue; otherwise, it's an rvalue. You may also find it helpful to think of an lvalue as something that can appear on the left-hand side of an assignment statement,

Let's illustrate it by examining a function `plusOne` that takes a `Node` by value, and returns a `Node` by value. In the body of the function, we traverse the singly-linked list and increment the `data` field by 1, then return a copy of the `Node` (This version of the code is in `lectures/c++/06-classes/04-rvalue/node.cc`).

```
#include <iostream>

struct Node {
    int data;
    Node *next;

    Node( int data, Node * next = nullptr );
    Node( const Node & other );
    ~Node() { delete next; }
};

std::ostream &operator<<(std::ostream &out, const Node &n);

Node plusOne( Node n ) {
    for ( Node *p{&n}; p ; p = p->next ) {
        ++p->data;
```

```

    }
    return n;
}

int main() {
    Node n{ 1, new Node{2} };
    Node n2{ plusOne(n) };
    std::cout << n << std::endl << n2 << std::endl;
}
// implementations follow

```

The key concept here is that the initialization of `n2` invokes a copy constructor to copy the contents of `plusOne`'s parameter, `n`. When we do this, what is the value of `other` passed as the parameter to `Node`'s copy constructor? The return of the function `plusOne` is an rvalue; however, `Node`'s copy constructor expects to receive an lvalue reference. Thus, the compiler creates a temporary object to hold the result of `plusOne`. `other` is a reference to this temporary object and the copy constructor deep-copies from it. This object will be automatically discarded once the call to `plusOne` and `Node`'s copy constructor ends.

But it is wasteful to copy information from a temporary object that will just be discarded anyway. Why not just "steal" the information from the temporary object instead, before it's destroyed, and save the cost of a copy?

In order to be able to perform this optimization, we have to be able to tell whether or not the parameter `other` is a reference to a temporary object or is instead a stand-alone object. This is done in C++ by specifying that our temporary `Node` object is an *rvalue reference* to a `Node`. The syntax for this is `(Node&&)`. The operations that take an *rvalue reference* as the parameter are similar to the copy operations, but are called *move* operations because they move the data from the temporary object to the object being initialized or assigned, instead of making a copy.

## Move constructor

The introduction of this new notation for an rvalue reference lets us now declare our move constructor signature, where a **move constructor** is designed to steal the information from the rvalue passed into the constructor. In general, the signature format will be:

```
class-type( class-type && parameter-name ) { ... }
```

Since we're creating a brand-new object, we don't have to worry about previous data field values. All we need to do is copy over the information from the rvalue; however, we need to be careful about any shared information we might have copied over that would be destroyed by the automatic invocation of the object's destructor! This means that in our implementation of the Node move constructor, we have to reset the pointer to the singly-linked list in other (whose address we've copied) to `nullptr`; otherwise, the list that we've just stolen will be destroyed when the rvalue other is destroyed.

Using our Node class as an example, the move constructor becomes:

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

So now when we write the following code fragment, the move constructor is run twice:

1. once to take the information being returned from n in the return statement into a location in the run-time stack as an rvalue since plusOne returns a Node by value, and
2. once to move the information from the rvalue into n2.

```

Node plusOne( Node n ) {
    for ( Node *p{&n}; p ; p = p->next ) {
        ++p->data;
    }
    return n;
}

int main() {
    Node n{ 1, new Node{2} };
    Node n2{ plusOne(n) };
    ...
}

```

The full example can be found in `lectures/c++/06-classes/04-rvalue/nodemove.cc`.

## Move assignment operator

**Move assignment** is structurally similar to copy assignment. We need to worry about self-assignment, and we need to make sure that we don't leak memory. We're just now adding to our previous solution the fact that we can just steal the information directly from the rvalue rather than copying it. Since we know that the rvalue is going to be destroyed soon, we can just exchange our old information with the new information from the rvalue, guaranteeing that it'll be properly cleaned up when the rvalue's destructor is run.

This lets our Node move assignment operator become:

```

struct Node {
    ...
    Node & operator=( Node &&other );
    void swap( Node &other );
    ...
};

...

Node & Node::operator=( Node &&other ) {

```

```
    swap(other);
    return *this;
}

void Node::swap(Node &other) {
    std::swap( data, other.data );
    std::swap( next, other.next );
}
```

An example of use is:

```
int main() {
    ...
    Node n2{ ... };
    n2 = plusOne(n2);
    ...
}
```

The full example can be found in `lectures/c++/06-classes/04-rvalue/nodemoveassign.cc`.

# Copy/Move Elision

Let's see again the example introduced in the previous lesson (available in `lectures/c++/06-classes/04-rvalue/node.cc`). Note that it contains two static variables used to count each call of the Node's constructor and copy constructor, and `main` outputs those values at the end of its execution. (We will explain static variables soon. For now, it is enough to know that they count how many times each constructor was called. Or, if you wish, feel free to read ahead about [static members](#).)

```
#include <iostream>

struct Node {
    static int numCtorCalls, numCpyCtorCalls;
    int data;
    Node *next;

    Node( int data, Node * next = nullptr );
    Node( const Node & other );
    ~Node() { delete next; }
    static void printNumCtorCalls();
};

std::ostream &operator<<(std::ostream &out, const Node &n);

Node plusOne( Node n ) {
    for ( Node *p{&n}; p ; p = p->next ) {
        ++p->data;
    }
    return n;
}

int main() {
    Node n{ 1, new Node{2} };
    Node n2{ plusOne(n) };
    std::cout << n << std::endl << n2 << std::endl;
```

```
    Node::printNumCtorCalls();  
}  
  
// implementations follow
```

How many constructor calls should we see if we run this program?

1. In `main`'s first line, we first initialize a `Node` in the heap (1 regular constructor call)
2. Also in `main`'s first line, we initialize `Node n` in the stack (1 regular constructor call)
3. In `main`'s second line, we invoke `plusOne` passing a copy of `n` as the parameter. To initialize `plusOne`'s parameter `n`, the copy constructor is invoked (1 copy constructor call)
4. The body of the copy constructor makes a deep copy. This means that it needs to make a copy of the heap-allocated `Node` pointed by `next` (1 copy constructor call)
5. The last line of `plusOne` returns the parameter `n` by value. This means that a copy of `n` must be made to be returned. Thus, the copy constructor is called again (1 copy constructor call)
6. Again, within the body of the copy constructor, a deep copy of the `Node` pointed by `next` must be made (1 copy constructor call)
7. Now we're back to the second line of `main`. We now need to initialize `n2` with an invocation to `Node`'s copy constructor because we're passing the return of `plusOne` as the parameter to `n2`'s initialization. So, the overloaded version of the constructor that receives a `Node` reference is called (1 copy constructor call)
8. Once more, within the body of the copy constructor, a deep copy of the `Node` pointed by `next` must be made (1 copy constructor call)

Therefore, we would expect to see 8 constructor calls, i.e., 2 calls to the regular constructor and 6 calls to the copy constructor. Let's see if this checks out:

```
$ g++ -std=c++14 -o node node.cc  
$ ./node  
1 2  
2 3
```

```
2 basic ctor calls
4 copy ctor calls
```

As you can see, there were only 4 invocations of the copy constructor instead of 6. Why did this happen? It turns out that the compiler can carry out an optimization called [elision](#).

**What is elision?** As of C++11, the compiler is required to omit move/copy constructor calls in favour of building the information directly into the memory space where it would have been moved/copied, even if omitting these calls changes the behaviour of the program! (For example, if the constructors printed something, or incremented a counter.) This is what we mean by *elision* i.e. we're eliding/removing those constructor calls. Note that this *doesn't* mean that all copy or move constructor calls are removed, the compiler just removes the ones where we would be copying/moving into/from an rvalue. Since an rvalue has no accessible address it is a temporary object designed to be thrown away, so the information may as well be written directly into the final destination.

If we go back to the list of constructor calls above, we can see that steps 5-6-7-8 seem redundant. First, a deep copy of the Node returned by plusOne is made to a temporary location (steps 5-6), then another copy of this temporary Node is made to initialize n2 (steps 7-8). With elision, the compiler omits one of these copies. Therefore, the Node returned by plusOne is copied directly into the space allocated for n2, skipping steps 5 and 6.

If we compile the program with the g++ option `-fno-elide-constructors`, this will turn off the compiler optimizations, and we can see how many copy constructor calls are made if we don't make our program efficient by turning the elision on.

```
$ g++ -std=c++14 -o node -fno-elide-constructors node.cc
$ ./node
1 2
2 3
2 basic ctor calls
6 copy ctor calls
```

As you can see, with elision disabled, the compiled program executed 6 copy constructor calls as described above, instead of skipping the redundant copies.

In this course, we don't expect you to know the exact circumstances in which elision is permitted, just that it is possible. In quizzes, we will specify whether or not elision occurs if it changes the answer to the question.

# Rules for replacing default compiler operations

These rules are copied from: <http://www.cplusplus.com/doc/tutorial/classes2/>

| Member function     | implicitly defined:   | default definition: |
|---------------------|---|---------------------|
| Default constructor | if no other constructors  | does nothing        |
| Destructor          | if no destructor  | does nothing        |
| Copy constructor    | if no move constructor and no move assignment                         | copies all members  |
| Copy assignment     | if no move constructor and no move assignment                         | copies all members  |
| Move constructor    | if no destructor, no copy constructor and no copy nor move assignment | moves all members   |
| Move assignment     | if no destructor, no copy constructor and no copy nor move assignment | moves all members   |

Note that if you don't define a move constructor/move assignment operator, then the copying versions of these operators will be used instead. If they are defined, then they will replace all calls to the copy versions when the argument is an rvalue. If an object being moved contains primitive data types (pointers, characters, integers, real numbers, etc), then the default compiler-provided move operation will copy those data fields.

- See [https://en.cppreference.com/w/cpp/language/move\\_assignment](https://en.cppreference.com/w/cpp/language/move_assignment) for a discussion on how move assignment is implemented.
- See [https://en.cppreference.com/w/cpp/language/move\\_constructor](https://en.cppreference.com/w/cpp/language/move_constructor) for a discussion on how move constructor is implemented.

## Rule of Five

This leads us to the **Rule of Five**. If you write any one of:

1. copy constructor

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

then you usually need to write all five operators.

# Member operators

Some functions *must* be declared members of the class, others are not allowed to be class members, and for some we have a choice as to whether or not they're placed in the class.

So how do we know where to place our functions? If we make it a member of a class, then there implicitly is a first parameter `this` that takes the role of the first operand. Because of the way C++ defines operators, that tells us that the object on which we are invoking the operation **must** be the left-hand-side operand to the operator i.e. it must appear on the left.

For example, if we define `operator+` and `operator*` for our vector class `Vec` as:

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

then we are implying that a `Vec` object must appear on the left of any multiplication operator, as in:

```
Vec v1{ 1, 2}, v2{3, 4};  
...  
v1 = v2 * 3;
```

It would be a syntax error if we tried to write the multiplication as:

```
v1 = 3 * v2;
```

since `int` isn't a class, and we therefore can't make `Vec operator*( const Vec & other )` a member of it.

If we made `Vec operator*( const int k, const Vec & other )` a member of `Vec`, this wouldn't do what we wanted either, since now there are actually *three* parameters to this version of `operator*`: `this`, `k`, and `other`.

The only way we can create a version that works is to move this version of `operator*` outside of `Vec`, as in:

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

What about I/O operators? By definition, the class type *has* to be on the right-hand side of the operator since we expect the stream to be on the left-hand side. Otherwise, this goes against the usual way we use the operators! Therefore, all of the I/O operators must be declared outside of the class.

In summary:

| Must be class members  | Must not be class members   |
|--|---|
| <ul style="list-style-type: none"><li>• constructors</li><li>• destructor</li><li>• <code>operator=</code></li></ul> | Operators whose first parameter <i>isn't</i> an object of the class type. In particular, I/O operators, <code>operator&lt;&lt;</code> and <code>operator&gt;&gt;</code> . |

- `operator[]` : used for classes where the idea of a "subscript" is meaningful
- `operator->` : used for classes where the idea of "points to" is meaningful
- `operator()` : overloads the *function call* operator
- `operator T()` : used to cast the object to the type T

If the function doesn't fall into one of those two categories that forces the placement, then the choice is up to you.

# Static Members

So far, we have used only member fields that are particular for each object. In other words, each instance of a class will have its own copy of the member fields. For example, each object of type `Student` will have a different copy of the fields `assns`, `mt`, and `final`. And when the method `grade()` is called, it will have an implicit parameter `this`, which points to the instance in which the method was called.

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

However, sometimes we need to keep state or data across all objects of a class instead of for each individual object. For example, we may want to track how many `Student` objects are created. This information belongs to the `Student` class, not to each individual object.

**Static members** are associated with the class itself, and not with any particular instance (object) of the class. For example:

```
// In student.h:  
struct Student {  
    . . .  
    static int numInstances;  
    Student(int assns, int mt, int final): assns{assns}, mt{mt}, final{final} {  
        ++numInstances;  
    }  
};  
  
// In student.cc:  
int Student::numInstances = 0;
```

As shown above, static fields must be defined external to the class. At this point, the memory will be allocated for a single copy of the `Student::numInstances` field. This memory will be freed automatically when the program is terminated. Since it is not allocated as part of object instantiation, it should not be freed as part of object destruction.

**Static member functions** don't depend on a specific instance for their computation (they don't have an implicit `this` parameter). Thus, they can only access static fields and call other static member functions.

```
// In student.h:  
struct Student {  
    int assns, mt, final;  
    static int numInstances;  
    . . .  
    // The static method howMany() can access the static field numInstances.  
    // However, it cannot access the instance fields assns, mt, final.  
    static void howMany() {  
        cout << numInstances << endl;  
    }  
};  
  
// In main.cc:  
Student billy {60, 70, 80}, jane {70, 80, 90};  
Student::howMany(); // 2
```

Note how the method `howMany()` is called on the `Student` class, not on any particular object (e.g., `billy`, `jane`).

You can find two different examples of the `Student` class with static members in your repository under `lectures/c++/06-classes/06-static/`.

(If you need a refresher on other ways the `static` keyword can be used in C++, see <https://www.geeksforgeeks.org/static-keyword-cpp/>.

# Object arrays

**Creating an array of objects has some restrictions of which you 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 example, if the class Vec is defined as:

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

then trying to declare an array of Vec objects as either of the following statements:

```
Vec arr1[10];  
Vec * arr2 = new Vec[10];
```

will fail since Vec doesn't have a default constructor. If I change the definition to:

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

then my 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 example:

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

```

};

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

```

However, this won't work if the size of the vector is determined dynamically (for example, read in from input, or retrieved as a command-line argument). It also doesn't scale well—who would want to do this for an array containing hundreds if not thousands of objects?!

**If my class is such that it doesn't have a default constructor (or one cannot/should not be introduced), then my only other alternative 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 element of the array must also be freed.**

You can find two different examples of this approach in your repository under `lectures/c++/06-classes/05-obj-arrays/`:

| File: objarray1.cc  | File: objarray2.cc  |
|---|---|
| <pre> int size; ... Vec *arr[size]; for (int i = 0; i &lt; size; i++ ) {     arr[i] = new Vec{ i, i+1 }; } ... for (int i = 0; i &lt; size; i++ ) {     delete arr[i]; } </pre> | <pre> int size; ... Vec **arr; arr = new Vec*[size]; for (int i = 0; i &lt; size; i++ ) {     arr[i] = new Vec{ i, i+1 }; } ... for (int i = 0; i &lt; size; i++ ) {     delete arr[i]; } delete [] arr; </pre> |

The second version is useful if your array is going to be very large, since otherwise you might run out of space on your run-time stack for it.

# Constant Objects

A **constant object** (or **const object**) is an object whose fields cannot be modified. To make an object constant, you can create it using the keyword `const` or declare a parameter using the keyword `const`. For example:

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

Can we call methods on const objects? Of course, otherwise, the object would not be very useful. However, what happens if the method tries to modify the fields of a const object?

Please take a moment to check the example file `lectures/c++/06-classes/07-const/studentBad.cc` in your repository. In this example, we create a const object `s2`, then try to call its `computeGrade()` method. Try to compile this program. What happens?

- It compiles without any error
- There is a compilation error

We can only call methods on const objects if they promise not to modify fields. This is done by adding the keyword `const` to the end of the method's signature (after the parameter list). For example:

```
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 don't modify fields. *Only const methods may be called on const objects.*

Now, take a moment to check the file `student.cc` in the same directory on your repository, which is a fixed version of `studentBad.cc`. Be sure to notice the difference between them, i.e., we added the `const` keyword to the method `computeGrade()`. Try to compile and run `student.cc`. Does it work now?

- It compiles without any error
- There is still a compilation error

Yes, that's right. The code on `student.cc` will compile and run without errors.

## Mutable Fields

What if, for example, we want to keep track of how many times a method was called? This can be useful to do performance profiling, i.e., if we know how many times each method was called in our program, it can help us find performance bottlenecks. Here is a tentative implementation:

```
struct Student {  
    . . .  
    int numMethodCalls = 0;  
    float grade() { // should be const or not?  
        ++numMethodCalls;  
        return . . . ;  
    }  
};
```

Now `grade()` cannot be a `const` method, so we can't call it on `const Student` objects. But `numMethodCalls` affects only the **physical constness** of the object (i.e., whether the actual bit pattern that makes up the object changes), rather than its **logical constness** (i.e., whether any change to the object is apparent to outsiders).

So, we want to be able to update `numMethodCalls`, even if the object is `const`. We declare the field **mutable**:

```
struct Student {  
    . . .  
    mutable int numMethodCalls = 0;  
    float grade() const { // can be const now  
        ++numMethodCalls; // OK now  
        return . . . ;  
    }  
};
```

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

Please take a moment to check the example file `lectures/c++/06-classes/07-const/studentProfileBad.cc` in your repository. In this example, `computeGrade()` is a `const` method; however, it tries to update the field `numMethodCalls`. If you try to compile it, you will see the message "error: increment of member 'Student::numMethodCalls' in read-only object", which means that we cannot update the value of the field `numMethodCalls` on a `const` object.

Now, check the fixed code in `studentProfile.cc`. Be sure that you notice the difference between the two files, i.e., now the field `numMethodCalls` is declared as `mutable`. Therefore, it can be modified even by a `const` method on a `const` object.

## Best Practices

- Whenever you create a method that you 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.
- Whenever you create a method that receives objects as parameters and you know that the method will not modify those objects, mark the parameter(s) as `const`. This will allow `const` objects to be passed as parameters.
- If you have fields that must be modified even for `const` objects (such as the example `numMethodCalls` above), declare the field as `mutable`. This will allow it to be modified even inside `const` methods.
- Whenever you create (instantiate) a new object and you know that the value of the fields should not be modified anymore after initialization, declare it as a `const` object. This will prevent accidental modification in other parts of the code.

# Invariants

Consider the following example:

```
struct Node {  
    int data;  
    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 behaviour. Quite possibly, the program will crash.

So the class Node relies on an assumption for its proper operation: that next is either nullptr or is a valid pointer to the heap.

This assumption is an example of an **invariant**—a statement that must hold true—upon which Node relies. But with the way that Node is currently implemented, we can't guarantee this invariant will hold. We can't trust the user to use Node properly.

So, *an invariant is a statement that must hold true otherwise our program will not function correctly*. It is hard to reason about programs if you can't rely on invariants.

Consider another example. 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 that 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 you call `pop()` on it. However, the returned item may not be what the client was expecting. This can lead to *logical errors* (or bugs), i.e., errors that do not prevent the program from executing, but they cause the program to behave differently than it should. 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 object-oriented programming, we can do this with *encapsulation*. *This is one of the benefits of using object-oriented programming.*

## Encapsulation

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

- **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 different 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.
- **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.

For example:

```
struct Vec {  
    Vec (int x, int y); // By default, members are public  
  
    private: // What follows is private; cannot be accessed outside struct Vec  
    int x, y;  
  
    public: // What follows is public; accessible to all  
    Vec operator+(const Vec &other);  
    . . .  
};
```

Note that, **by default, all members of structs are publicly accessible.**

So, in the example above, the constructor is public even though its access modifier was not specified. After you use one of the keywords `private` or `public`, all the declared members will use that access modifier until you write one of those keywords again or until the end of the class specification. So, you 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.

In the example above, the fields `x` and `y` are private. So, trying to access them from outside of a `Vec` object is an error. For example, if someone creates a `Vec v` and tries to access `v.x` or `v.y`, the program will not compile. On the other hand, the method `operator+` is public. So, calling `v.operator+(v2)` (or writing `v + v2`, which is the same as calling the `operator+` method) is valid.

So, we can say that we *encapsulated the implementation details of Vec*, so the client cannot access the internal state of the object directly. From the point of view of the client, they can only create a `Vec` passing two `ints` as parameters and then add two vectors using `operator+`. However, the client does not need to know how `Vec` objects store their internal state and cannot access or modify that state directly.

In the next topics, we will provide additional examples of how we can use encapsulation to enforce invariants in classes such as the node or stack examples in the previous section.

# The `class` Keyword

By default, all members of structs 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`:

```
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 (public in `struct`, private in `class`). Everything else that you already know that works `struct` will work the same way with `class`. Therefore, in general, you could use any of these two keywords to define your classes in your programs.

**However, the best practice is to make members private by default and only make public those methods that need to be called from outside of the object. Thus, it is usually preferable to use `class` and then only change the necessary methods to public visibility.**

(For more discussion on this, see <http://www.fluentcpp.com/2017/06/13/the-real-difference-between-struct-class/>.)

# Accessor and Mutator Methods

When you use encapsulation, the clients cannot read or modify the values of the object's fields because they are all private. If you need to allow clients to read or modify those values from outside of the object, you can create **accessor** and **mutator** methods. These are usually simple methods with the goal of allowing clients to read (access) or modify (mutate) the values of the object's fields. For example:

```
class Vec {  
    int x, y;  
public:  
    . . .  
    int getX() const { return x; } // accessor  
    void setX(int z) { x = z; }   // mutator  
  
    int getY() const { return y; } // accessor  
    void setY(int z) { y = z; }   // mutator  
};
```

The accessor and mutator methods could have any name. However, it is a common convention to name accessor methods as `getField` (where `Field` is the actual field's name) and mutator methods as `setField` (again, `Field` is the actual field's name). For this reason, accessors and mutators are also sometimes called *getters* and *setters*. Some people also like to replace `get` with `is` for boolean fields (i.e., `isField` instead of `getField`). Although these are all optional conventions and your program could certainly work without following them, it is strongly recommended to follow this simple naming convention. This will allow other programmers (and yourself!) to easily identify the accessors and mutators when looking at your code.

Note that it is also a good practice and a convention that accessor methods should not modify any state, only read and return the value of a field. Therefore, accessor methods should generally be declared as `const`.

## Enforcing encapsulation with accessors and mutators

In general, you should not automatically create accessor and mutator methods for all the fields in your class. Doing so would allow the client to read and modify any internal value of the object, which would totally break encapsulation. Creating public accessors and mutators for all fields would be almost the same as just making all fields public, which we don't want to do for the reasons we already explained in [Invariants and Encapsulation](#).

Therefore, a good practice is to start by not creating any accessors or mutators. Then, add only those that are really needed for clients to communicate with the objects. Thus, only create accessor methods for fields whose value is of interest to the clients. If your class has fields that store internal state that should only be manipulated inside the object, there is no reason to create accessors for them. Similarly, create only mutator methods for fields whose value should be directly updated by clients. Again, for any internal state that should only be manipulated inside the object, there is no reason to create mutators for them.

It is also very common to have internal state that should be updated only as a result of specific methods calls, not through direct mutators. For example, a stack implementation usually includes some data structure to store the stack elements. This data structure should be a private field. The client can read or modify this data structure only by calling the stack's `push()` and `pop()` methods. To enforce encapsulation (as well as the stack's invariants), the client should never be allowed to directly read or modify the internal data structure. Therefore, the stack implementation should not have other accessors or mutators, only the provided `push()` and `pop()` methods.

```
class Stack {  
    int *elements; // a private field with an internal state that is not directly visible to the client  
    int numElements; // a private field with an internal state that is not directly visible to the client  
public:  
    Stack(); // constructor  
    ~Stack(); // destructor  
    void push(int); // adds an element into the stack (modifies elements and numElements)  
    int pop(); // removes and returns an element from the stack (modifies elements and numElements )  
};
```

Additionally, when you create mutator methods, they may perform additional verifications or actions to enforce the invariants and maintain the consistency of the object. This is a good application of encapsulation to enforce the invariants and avoid logical errors. For example:

```
// Invariant: all grades are between 0 and 100
class Student {
    int assns, mt, final;
    int enforceGradeValue(int g) const { // private method because it doesn't need to be called by clients
        if (g < 0)
            return 0;
        if (g > 100)
            return 100;
        return g;
    }

public:
    . . .
    int getAssns() const { return assns; }           // accessor
    void setAssns(int g) { assns = enforceGradeValue(g); } // mutator
    int getMt() const { return mt; }                 // accessor
    void setMt(int g) { mt = enforceGradeValue(g); }   // mutator
    int getFinal() const { return final; }           // accessor
    void setFinal(int g) { final = enforceGradeValue(g); } // mutator
    . . .
};
```

Note that this Student class has an invariant that all grades must be between 0 and 100. However, we cannot just assume that clients will always provide a valid grade value when they call the mutator methods on a student object. By making the fields private and providing mutators, we can enforce this invariant. We just need to make the mutators check the parameter value before modifying the field value.

Additionally, why is the method `enforceGradeValue` private? This method does not modify any internal state; thus, it would not be a big deal if we made it public. However, it is a good practice to hide the internal implementation details and only display to the client what they need to know to use the class. This makes the public interface of the class smaller and easier for the client to understand. In this case, the client never needs to call that method directly because it

is already called from the mutator methods when needed. By making it private, we hide it from the client, so it is one less thing they have to learn before they can use the `Student` object. Hiding unnecessary details from the client is also part of the encapsulation, even if they would not break an invariant if exposed.

# Fixing the Linked List with Encapsulation

Now that you have learned about encapsulation, we can fix the linked list we showed as an example at the start of this module. As a refresher, this was an initial attempt at implementing the list:

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

The problem is that there is an invariant: `next` should be either `nullptr` or a valid pointer to the heap, which will be freed in the `Node`'s destructor. However, we cannot rely on the clients using our list correctly. So, we need to encapsulate the internal implementation of this linked list to enforce this invariant.

Before reading the solution below, take a moment to think about this issue. How would you modify the linked list implementation to enforce the invariant described above?

- In the constructor of a new `Node`, check if the second parameter is a heap-allocated pointer before initializing the `next` field.
- Create a class that encapsulates (hides) the implementation of the nodes from the client.
- Make the `data` and `next` fields private.

Yes, that's correct! The class should expose only what the client needs to know, not internal implementation details. Thus, the client should only know that it is possible to add, read, and remove elements from the linked list, but they should not know the internal implementation details of the `Nodes`. You can see the full solution below.

# The Solution

To fix the linked list, we need to hide (encapsulate) the implementation details of the nodes from the clients. The public operations of a list for its clients are adding, retrieving, and removing elements from the list. The fact that these elements are stored in nodes allocated in the heap is an internal implementation detail that the client does not need to know. If the nodes are allocated by the list implementation, not by the client, the implementation can guarantee that they will always be allocated in the heap.

## list.h

```
class List {
    struct Node;           // Private nested class; only accessible inside List
    Node *theList = nullptr; // Must be nullptr or pointer to heap-allocated object
public:
    void addToFront(int n); // Adds an element to the front of the list
    int ith(int i);        // Retrieves the element in the ith position of the list
    ~List();
};
```

## list.cc

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

List::~List() { delete theList; }

void List::addToFront(int n) { theList = new Node(n, theList); }

int List::ith(int i) {
```

```
Node *cur = theList;
for (int j = 0; j < i; ++j, cur = cur->next);
return cur->data;
}
```

Note that `Node` is a nested class of `List`. It is first declared privately inside the class `List` in `list.h` and implemented in `list.cc`. There is no public declaration of `Node`. Thus, clients will never be able to use it directly. So, the members of the class `Node` don't need to be private as the class itself is already limited to exist within `List`. The list must be able to access the fields of the nodes. Alternatively, the fields `data` and `next` could be made private and accessor/mutator methods created as necessary. Both options are acceptable in this case.

Now, clients can never create nodes directly. They can only create initially empty lists (using the implicit default constructor) and add `int` elements using `addToFront`. The implementation of this method will always allocate a new `Node` in the heap. Therefore, since only the class `List` can manipulate `Node` objects, we can guarantee the invariant that `next` within class `Node` is always either a null pointer or was allocated by `new`.

It's always a good idea to document invariants. This can be done in the comments in the source files or in another standard document used by the organization. This is done in our example with the comment: `// Must be nullptr or pointer to heap-allocated object`. You don't need to use the word "invariant", but it's important to document the rule that must be followed. By doing so, you ensure that if you or someone else needs to modify the class later, you do not forget about the invariants and you do not break the existing implementation. Note that we may not always include comments about the invariants in our code examples for this course because we want to keep the examples short. But when you are working on your assignments, you should certainly document any invariants in your design and just make it as a good practice for any code you write.

You can find this implementation of the linked list in the directory `lectures/c++/05-classes/08-encapsulation` in your repository. Please take a moment to review and test it.

# Friend Classes

Sometimes, it may be useful to give some class privileged access to another class's private members.

For example, in the previous topic, we created Node as a private nested class within List:

```
class List {  
    struct Node;           // Private nested class; only accessible inside List  
    Node *theList = nullptr; // Internal state  
public:  
    void addToFront(int n); // Adds an element to the front of the list  
    int ith(int i);        // Retrieves the element in the ith position of the list  
    ~List();  
};
```

We did this so that new Nodes could be created only from within the List. This way, we can enforce the invariant that all nodes should be allocated in the heap.

However, now the problem is that clients cannot traverse this list from node to node, as they would usually do with a linked list. Repeatedly calling `ith` to access all of the list items would cost  $\Theta(n^2)$  time. One possible solution is to make Node a public nested class so that clients can retrieve nodes and follow the pointer to the next node in the list, but make the Node constructor private, so clients can never create nodes. But if the constructor is private, List won't be able to create Nodes either. We can solve this by giving class List privileged access to Node by making it a **friend**:

```
class List {  
public:  
    class Node { // Public nested class  
        int data;  
        Node *next;  
        Node (int data, Node *next); // Constructor is now private!  
    public:
```

```

~Node();
int getData() const;
Node *getNext() const;
friend class List; // List has access to all members of Node
}
private:
    Node *theList = nullptr; // Private internal state
public:
    void addToFront(int n); // Adds an element to the front of the list
    int ith(int i) const; // Retrieves the element in the ith position of the list
    Node *getNodeAt(int i) const; // Retrieves the Node in the ith position of the list
    ~List();
};

```

Now, clients can use `List`'s method `getNodeAt` to retrieve a pointer to the `Node` in any position on the list. Then, `Node`'s `getData` method can be used to retrieve the value of the element and `getNext` can be used to traverse to the next node. But note that clients cannot directly access the internal state of the `Node` because the fields are now private. And clients cannot create `Nodes` because the constructor is private. However, class `List` can still access the private fields of `Node` and create new `Nodes` using the private constructor because we declared `List` as a friend class of `Node`.

This is an interesting solution to this problem.

**However, in general, you should give your classes as few friends as possible. Friendships weaken encapsulation and thus should be used only if really necessary.**

Additionally, note that we partially broke encapsulation with this solution because we exposed the implementation details of the `Nodes` to clients. While we did it in a way that the class invariant is still enforced, it is still something we want to avoid if possible. Ideally, clients should be able to traverse a list in the most efficient way possible without knowing what is the internal structure of the list (i.e., if it uses an internal array, a linked list, a tree, etc. should not matter to clients). This is possible by using the [Iterator](#) design pattern. Therefore, after finishing this page, we suggest that you take a moment to follow the link and study about [Iterators](#).

# Friend Functions

For this section, let's use a Vec class as an example. The x and y fields are private and have no accessor methods:

```
class Vec {  
    int x, y;  
};
```

What if we want to implement operator<< to print the values of x and y? So, it needs access to x and y, but it cannot be a member function. If the class defines getX and getY, there is no problem. But if we have no other reason to provide these methods, then maybe we would prefer not to provide them. In that case, we can make operator<< a **friend function**:

```
// vec.h  
class Vec {  
    int x, y;  
    friend std::ostream &operator<<(std::ostream &out, const Vec &v);  
};  
  
//vec.cc  
std::ostream &operator<<(std::ostream &out, const Vec &v) {  
    return out << v.x << ' ' << v.y;  
}
```

Just like friend classes, friend functions are an interesting solution to provide access to private members to a specific function without having to make the members public to everyone. However, they also weaken encapsulation and thus should be used only if really necessary.

# Design Patterns Introduction

A **design pattern** is a codified solution to a common software problem. It specifically deals with a problem in object-oriented software development, and thus focuses on the involved classes and their relationships. A design pattern has 4 key elements:

1. a memorable name that describes the problem or solution,
2. a description of the problem to solve,
3. the general form of the solution, and
4. the consequences (results and trade-offs) of using the pattern.

The guiding principle behind all of the design patterns is: encapsulate change on design decisions i.e. program to the *interface*, not the *implementation*. The class relationships thus rely heavily upon abstract base classes.

Patterns are more abstract than code libraries, but more concrete than design principles (e.g. coupling, cohesion, etc.). They extend the designers' vocabulary so that there is a common way to describe problems and solutions. Design patterns help designers find a suitable, more predictable design more efficiently. They can help code be refactored and evolve more easily.

Since they do make the code and design more complex by hiding information and using indirection in combination with polymorphism, they may not be appropriate for small projects.

We will not be formally studying design patterns in depth, but in this course we will present you with a few useful patterns to give you a taste of the concept.

## History

While we won't test you on this section, the history behind the evolution of design patterns is interesting.

Computer scientists took the concept from the work of architect Christopher Alexander and his co-authors, Sara Ishikawa and Murray Silverstein. In the trilogy: "The Timeless Way of Building", "A Pattern Language: Towns, Buildings, Construction", and "The Oregon Experiment", Alexander proposed a novel way of organizing the construction of towns, down to rooms in a house, by looking at the fundamental principles of what makes living space habitable and inviting i.e. looking at the problem to solve, and proposing a solution with the resulting trade-offs, where each of the 253 patterns fits within a larger web of patterns.

Here's an example that most of you can relate to, the pattern "159. LIGHT ON TWO SIDES OF EVERY ROOM". Alexander notes that:

"When they have a choice, people will always gravitate to those rooms which have light on two sides, and leave the rooms which are lit only from one side unused and empty.

This pattern, perhaps more than any other single pattern, determines the success or failure of a room. The arrangement of daylight in a room, and the presence of windows on two sides, is fundamental. If you build a room with light on one side only, you can be almost certain that you are wasting your money. People will stay out of that room if they can possibly avoid it. Of course, if all the rooms are lit from one side only, people will have to use them. But we can be fairly sure that they are subtly uncomfortable there, always wishing they weren't there, wanting to leave—just because we are so sure of what people do when they do have the choice.

Our experiments on this matter have been rather informal and drawn out over several years. We have been aware of the idea for some time—as have many builders. (We have even heard that "light on two sides" was a tenet of the old Beaux Arts design tradition.) In any case, our experiments were simple: over and over again, in one building after another, wherever we happened to find ourselves, we would check to see if the pattern held. Were people in fact avoiding rooms lit only on one side, preferring the two-sided rooms—what did they think

about it?

We have gone through this with our friends, in offices, in many homes—and overwhelmingly the two-sided pattern seems significant. People are aware, or half-aware of the pattern—they understand exactly what we mean.



With light on two sides .... and without

If this evidence seems too haphazard, please try these observations yourself. Bear the pattern in mind, and examine all the buildings you come across in your daily life. We believe that you will find, as we have done, that those rooms you intuitively recognize as pleasant, friendly rooms have the pattern; and those you intuitively reject as unfriendly, unpleasant, are the ones which do not have the pattern. In short, this one pattern alone, is able to distinguish good rooms from unpleasant ones.

The importance of this pattern lies partly in the social atmosphere it creates in the room. Rooms lit on two sides, with natural light, create less glare around people and objects; this lets us see things more intricately; and most important, it allows us to read in detail the minute expressions that flash across peopled faces, the motion of their hands ... and thereby understand, more clearly, the meaning they are after. The light on two sides allows people to understand each other.

In a room lit on only one side, the light gradient on the walls and floors inside the room is very steep, so that the part furthest from the window is uncomfortably dark, compared with the part near the window. Even worse, since there is little reflected light on the room's inner surfaces, the interior wall immediately next to the window is usually dark, creating discomfort and glare against this light. In rooms lit on one side, the glare which surrounds people's faces prevents people from understanding one another. Although this glare may be somewhat reduced by supplementary artificial lighting, and by well-designed window reveals, the most simple and most basic way of overcoming glare, is to give every room two windows. The light from each window

illuminates the wall surfaces just inside the other window, thus reducing the contrast between those walls and the sky outside. For details and illustrations, see R. G. Hopkinson, *Architectural Physics: Lighting*, London: Building Research Station, 1963, pp. 29, 103. ..." [Excerpt From: Christopher Alexander, Sara Ishikawa, Murray Silverstein. "A Pattern Language."]

There's a short YouTube video, [https://www.youtube.com/watch?v=UWCZUIy2\\_2k](https://www.youtube.com/watch?v=UWCZUIy2_2k) that shows several of them.

## Resources

|   |   |
|---|---|
|    | Despite its age, the classic book on design patterns is still "Design Patterns: Elements of Reusable Object-Oriented Software" by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, published in 1995.  |
|    | A more approachable, though Java-centric book, is the "Head First Design Patterns" book by Eric Freeman and Elisabeth Freeman, published in 2004. It is available electronically through the UW library's license with Safari Books Online. See <a href="http://proxy.lib.uwaterloo.ca/login?url=http://proquestcombo.safaribooksonline.com/0596007124">http://proxy.lib.uwaterloo.ca/login?url=http://proquestcombo.safaribooksonline.com/0596007124</a> |
| Michael Mahemoff has some very readable <a href="#">papers on design patterns</a> , including one on design pattern metaphors:<br><a href="http://mahemoff.com/paper/software/gofMetaphors/">http://mahemoff.com/paper/software/gofMetaphors/</a> |   |
| Hillside.net is a good design patterns resource site: <a href="https://hillside.net/patterns/">https://hillside.net/patterns/</a>   |   |

# Iterator Design Pattern

Previously, when we discussed [Invariants and Encapsulation](#), we showed an example of a linked list that encapsulated (hid) the implementation details of the nodes. As a refresher, here is the interface of the `List` class. However, if you haven't done so yet, you should complete the module about [Invariants and Encapsulation](#) before reading the rest of this page; otherwise, some of the topics we discuss here may not make sense to you.

```
class List {  
    struct Node;           // Private nested class; only accessible inside List  
    Node *theList = nullptr; // Internal state  
public:  
    void addToFront(int n); // Adds an element to the front of the list  
    int ith(int i);        // Retrieves the element in the ith position of the list  
    ~List();  
};
```

But now we can't traverse the list from node to node, as we would with a linked list. Repeatedly calling `ith` to access all of the list items would cost  $\Theta(n^2)$  time. But we can't expose the nodes or we lose 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 you have this situation, then *this* programming technique may solve it.

**Solution:** Use the **Iterator** design pattern. We 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.

## General Format

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

```
class MyClassIterator {  
    // some private field to keep track of where the iterator is currently pointing to  
    explicit MyClassIterator(/* a parameter to initialize it */) // private constructor  
public:  
    MyClass &operator*(); // accesses the item the iterator is currently pointing to (the return type can be different if needed)  
    MyClassIterator &operator++(); // advances the iterator to the next item and returns the iterator  
    bool operator==(const MyClassIterator &other) const; // returns true if both Iterators are equal (point to the same item)  
    bool operator!=(const MyClassIterator &other) const; // returns true if both Iterators are different (do not point to the same item)  
};  
  
class MyClass {  
    // other private members  
public:  
    // other public members
```

```

    MyClassIterator begin(); // returns an Iterator pointing to the first element
    MyClassIterator end();   // returns an Iterator pointint past the last element (not to the last element itself, but after it)
};


```

## Linked List with an Iterator

Here is an excerpt from the linked list including the Iterator class. You can find the complete example in the directory `lectures/se/01-iterator` in your git repository.

```

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.)
};

```

## Using the Iterator

Now the client can use iterator objects to walk the list in  $\Theta(n)$  (linear time):

```

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

```

Note that the client uses `lst.begin()` to obtain an iterator pointing to the first element, where the loop will start. `lst.end()` is used to obtain an iterator that points *past* the last element of the list, to detect when the loop should end. The iterator's `operator++` method (called with `++it`) is used to advance the pointer in the loop. And the iterator's `operator*` method (called with `*it`) is used to obtain the integer element from the current node.

Additionally, the client cannot create an `Iterator` object directly because `Iterator`'s constructor is private. However, `Iterator` declares `List` as a [friend class](#). Therefore, `List` can create new `Iterators` in its `begin` and `end` methods. Consequently, clients can only create list iterators by calling one of those two methods.

## Automatic type deduction

We can shorten the iteration loop somewhat 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. With automatic type deduction, we can write:

```

for (auto it = lst.begin(); it != lst.end(); ++it) {
    cout << *it << endl; // prints 3 2 1
}

```

## 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](#):

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

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

- C has methods begin and end that produce iterators;
- the iterator supports prefix operator++, operator!=, and unary operator\*.

Therefore, a minimal implementation of the Iterator design pattern requires at least these five operations (begin and end in the base class, as well as operator++, operator!=, and operator\* in the iterator class). If you omit any of these operations, 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
}
```

## Iterator Exercise

Now it's your turn to implement an Iterator to check if you understood the concept correctly. For this exercise, you must implement an iterator class to enable a range-based loop between some starting and ending characters. For example:

```
class CharRange {
    char first, last;
public:
    CharRange(char first, char last): first{first}, last{last} {}
    // Add the implementation of the Iterator here
};
```

You must implement an iterator such that the following program works and prints the output described in the comments:

```
int main() {
    CharRange range1{'A', 'E'};
    for (auto c : range1) {
        cout << c << endl; // prints: A B C D E
    }
}
```

```
CharRange range2{'l', 't'};  
for (auto c : range2) {  
    cout << c << endl; // prints: l m n o p q r s t  
}  
}
```

We do not provide a solution to this exercise. Using the explanations on this page and the linked list example, you should be able to understand how to implement this character range iterator. If you have any questions about how to do it, feel free to ask about it on Piazza or office hours.

# Unified Modeling Language

## Introduction

The **Unified Modeling Language**, known as UML for short, is a visual modeling language that lets people design and document a software system. The standard for the latest version can be found at:

<https://www.omg.org/spec/UML/About-UML/>

It is a popular standard, and the notation isn't based upon any particular programming language.

We will be looking at only one subset of its notation, that of class models.

## Class models

A **class model** is a diagram that visually represents a group of classes, and the relationships between them.

We will not be following the full, formal notation in this course, since we'll be treating it as the sort of communication tool that you bring out in meetings i.e. quick sketches on a board to ensure that everybody understands what is being accomplished. However, we do expect you to obtain a reasonable familiarity with the presented notation since you will be expected to be able to read and produce class model diagram on your assignments.

In particular, it will be a key component of how we describe design patterns, since understanding the classes involved and their relationships will be key to understanding how to apply the patterns.

## References

An *extremely* good reference is [Martin Fowler's "UML Distilled: Third Edition"](#). We strongly encourage you to read it, focusing on chapters 3 and 5. It is very short, and very well-written. It is available electronically through UW's library license with the publisher, O'Reilly.

The [UML reference card PDF](#) that you have been provided comes from Fowler's book.

There are also some other UML resources listed on the course web page under [Resources](#), and then under UML.

## Tools

There are a number of software tools that will let you draw UML diagrams. Find one that lets you draw the properly-shaped arrow-heads (they're a crucial part of the notation) and can export the resulting diagrams as a PDF. There's a link to some suggestions listed on the course web page under [Resources](#), and then under UML.

# Class model notation

Let's start by describing the notation used to specify the structure of a class.

## Classes

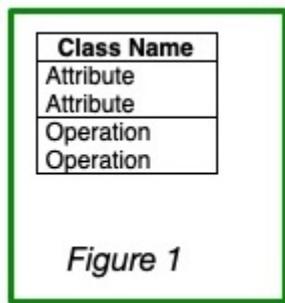


Figure 1

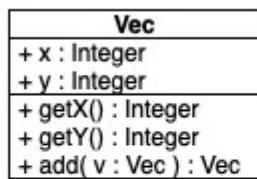


Figure 2

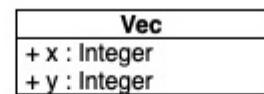


Figure 3

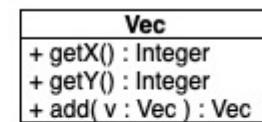


Figure 4

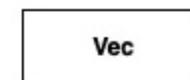
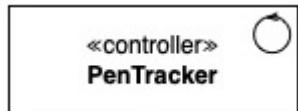


Figure 5

Figure 1, surrounded by a green box, shows the basic structure for a class. The general form of a class has three boxes:

1. The first box contains the **class name**. Figures 2 to 5 show examples of a class diagram using our **Vec** class from previous lecture material.
  - 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*, as in *Banking::CheckingAccount*.
  - 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 intended to be a container for multiple objects of the type.

- The class name may be qualified by an optional stereotype keyword, centered and in the regular typeface, placed above the class name, and within guillemets "«»" (guillemets are special Unicode single characters, and are **not** made up of << and >> but rather Unicode 226A and 226B) and/or an optional stereotype icon (in the upper-right corner). For example:



Some stereotypes are defined by the UML; others may be defined by the user. *We do not expect you to know or use them in this course.*

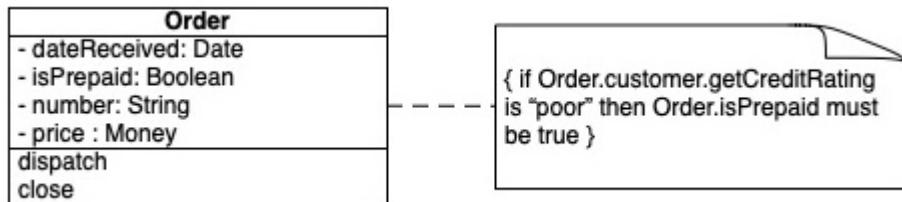
2. The second box is optional, and contains the class **attributes**.
3. The third box is also optional, and contains the class **operations**.

Either or both of the attributes or operation boxes may be omitted, in which case the separation line for the missing box isn't drawn. The class name may never be omitted. Examples of this are shown in figures 3 through 5.

## Comments

A **comment** can be added to the class diagram by joining a rectangle with a bent upper-right corner ("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.

For example:



## 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.

|                   |   |
|-------------------|---|
| <b>visibility</b> | Describes the visibility of the attribute as a punctuation mark, though can be instead described in the property string. There are four markers: <ul style="list-style-type: none"><li>• + public</li><li>• - private</li><li>• # protected</li><li>• ~ package</li></ul>   |
| <b>type</b>       | A string that describes the attribute's type. Usually written to be as language-independent as possible e.g. Boolean rather than the C++ <code>bool</code> , though in this course we may sometimes use C++-specific syntax since we're not using the UML formally. If the type is omitted, so is the preceeding colon. |

|                      |   |
|----------------------|---|
| /                    | Indicates that this attributed is derived from a parent class.  |
| <i>multiplicity</i>  | 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's 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. [ 3 .. 10 ] or [ 0 .. 1 ]), 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++.<br><br>If the upper bound is greater than 1, you can specify ordering and uniqueness on the values in the property string |
| <i>initial-value</i> | Specifies the default initial value as an equal sign followed by the value e.g. = 0   |
| <i>property</i>      | List of comma-separated strings surrounded by { }. Defaults to { changeable }, but can be specified as { readOnly } to indicate that it's a constant. Can be used to annotate multiplicity e.g. 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 form an ordered sequence), set (collection of unique elements) and list (ordered variable-length collection i.e. ordered bag).   |

If the attribute is static, the name and type strings are underlined.

Some examples of attribute declarations are:

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

## 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.

In this course, since all classes will have constructors, destructors, accessors (getters), and mutators (setters), we will not generally specify them in our UML class diagram since we'll assume their presence. Specifying them, however, clearly tells the implementer what is expected in terms of parameters and return-types, and is thus a useful element when drawing up the model.

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 list is optional. The name generally starts with a lower-case letter.

|                             |  |
|-----------------------------|--|
| <code>visibility</code>     | Same as for the attribute.   |
| <code>return-type</code>    | A string containing a comma-separated list of names that describes the operation's return type. (Some languages allow multiple return values.) If the return type is omitted, so is the preceeding colon and the return type is equivalent to the C/C++ <code>void</code> .  |
| <code>parameter-list</code> | <p>Comma-separated list of parameters, enclosed in parentheses. (We will often use C++ syntax for this information rather than the classic UML notation.) Each parameter is of the form:</p> <p><code>direction name : type multiplicity = default-value</code></p> <ul style="list-style-type: none"><li>• <code>direction</code>: specifies direction of information flow and is optional<ul style="list-style-type: none"><li>◦ <code>in</code>: input parameter, passed by value (default)</li><li>◦ <code>out</code>: output parameter with no input value; final value is available to caller</li><li>◦ <code>inout</code>: input parameter that may be modified and whose final value is available to the caller</li><li>◦ <code>return</code>: return value of a call (equivalent to <code>out</code> but available for use inline)</li></ul></li><li>• If the <code>default value</code> is omitted, so is the preceeding equal sign.</li></ul> |

|                 |   |
|-----------------|---|
| <i>property</i> | <p>List of comma-separated strings surrounded by {}. We will rarely use this element, though the ability to specify <code>abstract/virtual</code> or <code>static</code> may be useful if your UML drawing software doesn't let you italicize or underline.</p> <ul style="list-style-type: none"> <li>• A list of raised exceptions may be shown by specifying a comma-separated list of them after the keyword <code>exception</code>.</li> <li>• Instead of italicizing an abstract operation, the keyword <code>abstract</code> may be placed in the property string.</li> <li>• Instead of underlining a static operation, the keyword <code>static</code> may be placed in the property string.</li> <li>• The property <code>isQuery=false</code> indicates that this operation doesn't change the state of the object. It defaults to a value of <code>false</code>. A value of <code>true</code> doesn't guarantee that the state changes, only that it may change.</li> <li>• The property <code>isPolymorphic=true</code> indicates that the operation is overridable and is the default value.</li> </ul> |
|-----------------|---|

An **abstract operation** is italicized. In the classic UML notation, this would only apply to our C++ pure virtual methods, but the motivation here is for the modeller to be aware of *all* virtual methods, and thus we'll be italicizing them as well.

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»**

Some examples of operation declarations are:

```

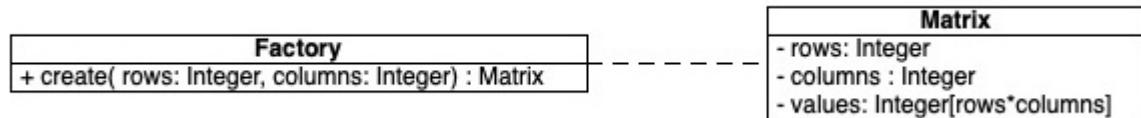
- display () : Location
+ hide ()
<<constructor>> + create ()
- attachXWindow( xwin : XWindow* )
# Matrix::transform (in distance: Vector, in angle: Real = 0) : Matrix

```

## Dependencies

If a class A has a **dependency** upon another class B, it doesn't 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 the line between classes A and B as a dashed line.

For example:



## Static

As mentioned previously, **static** attributes or operations are underlined in the UML class model; otherwise, you might need to tag them with the `{abstract}` property if your drawing software doesn't easily let you underline text.

For example:





# Class relationships

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

## Association

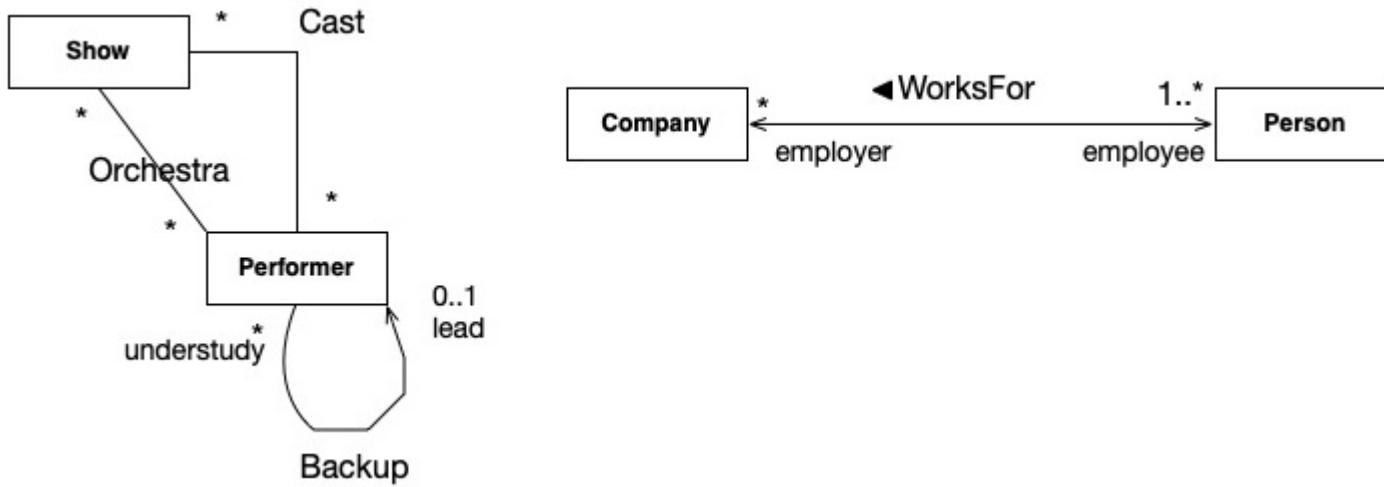
The simplest form of a relationship between two classes is that of an **association**. It is drawn as a solid line (usually straight, but arcs and curves are allowed) between the two classes in the UML diagram. There may be more than one association between the two classes, and a class may be associated with itself. Association lines may cross each other, but an optional "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 triangle to help indicate in which direction the naming relationship is to be read. An example of that is shown in the diagram below, where the triangle tells us that the association name is to be read as: *Person works for Company*.

## Association ends

Each end of the association may have:

- a name, which is equivalent to declaring an attribute in the class (sometimes called a *role name*) e.g. employee, understudy
- a navigation arrow to indicate that it is the holder of the information (the arrowhead must be "open" i.e. a >-shape) e.g. the understudy Performer "knows about"/"has" a lead Performer
- an associated multiplicity e.g. a Person may have 0 or more employers, a Company has at least one Employee
- constraints, such as order



## Aggregation

**Aggregation** is a form of association that describes 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 marked with a special symbol, an open/hollow/white diamond. Only one end of the association may be marked as an aggregate. It is possible to decorate either end of the association with a navigation arrow, or multiplicities, or constraints.

It is possible that the aggregation is shared i.e. a part is shared between one or more aggregates. These parts may also exist independently of the aggregate. An aggregate may or may not be responsible for destroying the parts.

This is often described as a "**Has-A**" relationship, where if A has a B, typically:

- B exists independently outside of A
- if A is destroyed, B lives on
- If A is copied, then B is not deep copied i.e. perform a shallow copy and B is shared

For example, a student may belong to zero or more clubs and exists independently of the clubs. A club recognized by the UW Federation of Students needs at least four students to be a part of it, one each to take the roles of president, vice president, secretary, and treasurer.



This is usually implemented via pointers or reference fields.

## Composition

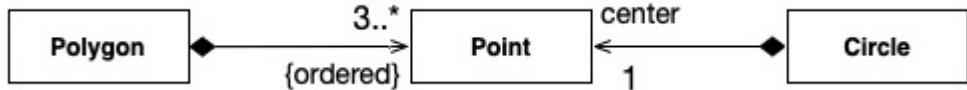
**Composition** is a stricter form of aggregation. In particular, once a "part" is joined to a "whole", **it may not be shared with any other object**. The whole is also responsible for destroying all of its component parts when it is destroyed. It may also be responsible for creating its components. 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 A owns a B, typically:

- B has no identity or independent existence outside of A
- if A is destroyed, B is destroyed
- If A is copied, then B is copied i.e. perform a deep copy

Since the owner has an implicit multiplicity of 1, we often don't bother to specify it (although it is not a problem to write the number 1 anyway). Note that the "owner" end of the association is marked with a solid, black diamond.

An example of a composition relationship is that of a Point to a Polygon or a Circle. An individual Point object may be part of an object of either type, but not part of both simultaneously.



This is usually implemented by a composition of classes. For example, an instance of the class `Basis` is composed of two `Vec` objects:

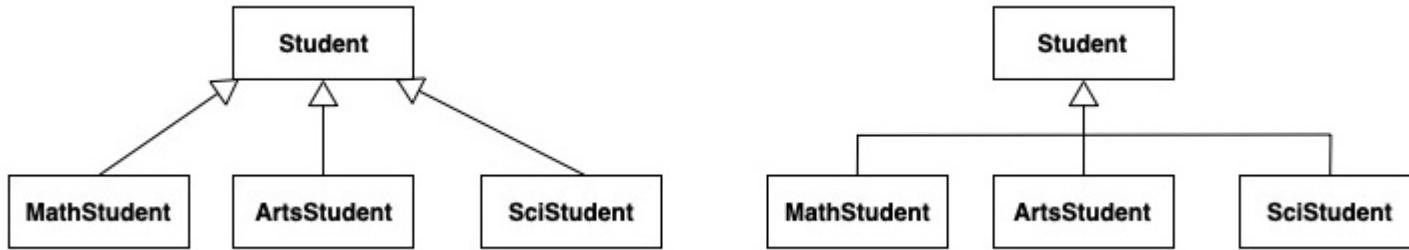
```

class Vec {
    int x, y;
public:
    Vec( int x = 0, int y = 0 );
    int getX();
    int getY();
    ...
};

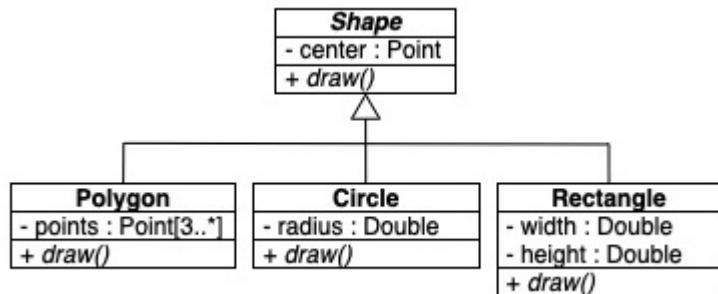
class Basis {
    Vec v1, v2;
public:
    Basis( Vec v1, Vec v2 );
    ...
};
  
```

## Generalization/specialization

The **generalization** (or **specialization**) association between the parent/superclass and the child/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.



In classic UML notation, the name of an abstract base class is italicized, and we italicize only the pure virtual methods. In this course, while we will still italicize the name of the abstract base class, we will italicize **all** virtual methods.



This is often described as an "**Is-A**" relationship, where A is a B. It is implemented through inheritance. We will discuss inheritance in more detail in the next submodule.

# Inheritance example

Lets examine this idea of inheritance more closely by looking at an example to motivate our code structure. We'd like to track our book collection. We have developed a collection of classes for this purpose. Here's the UML class model:

| Book  | Text   | Comic   |
|---|--|---|
| <ul style="list-style-type: none"><li>- title: String</li><li>- author : String</li><li>- length : Integer</li><li>+ Book( title: String, author : String, length : Integer )</li></ul> | <ul style="list-style-type: none"><li>- title: String</li><li>- author : String</li><li>- length : Integer</li><li>- topic : String</li><li>+ Text( title: String, author : String, length : Integer, topic : String )</li></ul> | <ul style="list-style-type: none"><li>- title: String</li><li>- author : String</li><li>- length : Integer</li><li>- hero : String</li><li>+ Comic( title: String, author : String, length : Integer, hero : String )</li></ul> |

Each Book, Text, and Comic class have fields to hold the author, title, and length (number of pages). 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 appropriate header guards and inclusion of the <string>library:

```
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 (const) so that the compiler will notify us if those methods try to change the contents of the data fields.

The problem then becomes "how do we store our actual books, texts, and comics"?

Ideally, we'd like to keep them all in a single array (or other collection type) so that we could iterate/traverse the entire collection in one loop, without having to worry about the underlying types. In order to accomplish this, there are three possible techniques:

1. C unions,

2. C void pointers

3. C++ inheritance

Let's take a look at each of the approaches.

## C unions

C provides a union type declaration that is also understood by C++ for backwards compatibility reasons. The union is defined to contain multiple data fields of potentially different sizes; however, only one data field will be available at any given moment. An instance of the union is allocated the number of bytes required by the largest data field in the union. For example, consider the code `lectures/c++/07-inheritance/union.cc`:

```
#include <iostream>

union ExampleUnion {
    int i;
    char c;
    float f;
};

int main() {
    union ExampleUnion e1, e2, e3;
    e1.i = -1;
    e2.c = 'x';
    e3.f = 3.14159;
    std::cout << "sizeof(e1) = " << sizeof(e1) << "; e1.i = " << e1.i << std::endl
        << "sizeof(e2) = " << sizeof(e2) << "; e2.c = " << e2.c << std::endl
        << "sizeof(e3) = " << sizeof(e3) << "; e3.f = " << e3.f << std::endl;
    std::cout << "e1.c = " << e1.c << " ; e1.f = " << e1.f << std::endl
        << "e2.i = " << e2.i << " ; e2.f = " << e2.f << std::endl
        << "e3.i = " << e3.i << " ; e3.c = " << e3.c << std::endl;
}
```

This program compiles and runs. Each of `e1`, `e2`, and `e3` is exactly 4 bytes long, and we can access the value stored in each respective variable. However, we can also (not always successfully) reinterpret the stored values by accessing them through the other fields:

```
sizeof(e1) = 4; e1.i = -1
sizeof(e2) = 4; e2.c = x
sizeof(e3) = 4; e3.f = 3.14159
e1.c = ý ; e1.f = -nan
e2.i = 120; e2.f = 1.68156e-43
e3.i = 1078530000; e3.c = Đ
```

This means that if we want to use a union, we have to either maintain a parallel data structure that keeps track of which field was used for our corresponding union variable, or we have to wrap the union in a struct (or class) that uses a data field to remember which field of the union was used for that particular instance.

## C void pointers

Another C approach that can be used is to have an array of pointers to the type void i.e. (void\*). Every element that we would want to store could then have its address converted from a pointer of the appropriate type to a (void\*). Once again, we would need a parallel data structure or a wrapper to remember the original value's type information.

For example, consider the code `lectures/c++/07-inheritance/voidptr.cc`:

```
#include <iostream>

struct Element {
    void * data;
    char dataType;
};

int main() {
    Element elems[3];
    int * iptr;
    char * cptr;
    float * fptr;

    elems[0].data = (void *) new int{-1};
    elems[0].dataType = 'i';
    elems[1].data = (void *) new char{'x'};
    elems[1].dataType = 'c';
    elems[2].data = (void *) new float{3.14159};
    elems[2].dataType = 'f';

    for ( int i = 0; i < 3; ++i ) {
        switch( elems[i].dataType ) {
            case 'i':
                iptr = (int*) elems[i].data;
                std::cout << *iptr << std::endl;
                break;
            case 'f' :
                fptr = (float*) elems[i].data;
                std::cout << *fptr << std::endl;
                break;
            case 'c':
                cptr = (char*) elems[i].data;
                std::cout << *cptr << std::endl;
                break;
        }
    }
}
```

```

        std::cout << *cptr << std::endl;
        break;
    } // switch
} // for
} // main

```

Note that this example relies upon the old C method of casting types. This is not the right way of casting types in C++, and should be avoided. We will see the proper C++ casting techniques later.

## C++ inheritance

Neither of the two C approaches are good. They require extra information to be stored and maintained. They also subvert the type system i.e. make it easy to change things to the wrong or invalid type, whether by accident or deliberately.

### Version 1

Instead, we're going to rely upon the C++ notion of **inheritance** and recognize that a *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 *specialization* of a *Book*, or a *Book* as a *generalization* of a *Comic*.

*Book* then becomes our **base class/parent class/superclass**. *Text* (and *Comic*) then becomes our **derived class/child class/subclass**.

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 a 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'll see more on that topic shortly.

The respective class header files are then changed to look like the following, assuming the appropriate header guards and inclusion of the `<string>` library:

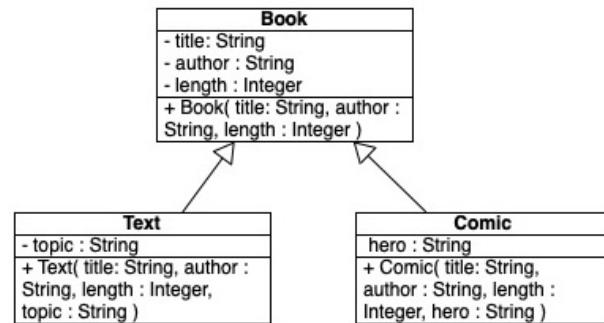
|  |  |  |
|--|--|--|
| <pre> class Book {     std::string author, title;     int length; public:     Book( const std::string &amp;author,           const std::string &amp;title,           int length );     std::string getTitle() const;     std::string getAuthor() const;     int getLength() const;     bool isHeavy() const; }; </pre> | <pre> class Text : public Book {     std::string topic; public:     Text( const std::string &amp;author,           const std::string &amp;title,           int length,           const std::string &amp;topic );     std::string getTopic() const; }; </pre> | <pre> class Comic : public Book {     std::string hero; public:     Comic( const std::string &amp;author,            const std::string &amp;title,            int length,            const std::string &amp;hero );     std::string getHero() const; }; </pre> |
|--|--|--|

---

The keywords ": public Book" after the class name (Text or Comic) tell us that Text (and Comic) *inherit publicly* from Book. ([We can also inherit using the keywords protected and private](#), but we don't cover this topic in CS246 and will only inherit publicly.)

Note that we do not repeat the data fields that we inherit from the base class! Doing so hides the parent's information, and is almost always an error.

Instead, our derived classes **inherit** the title, author and length data fields. They also take advantage of inheriting the parent class methods getTitle, getAuthor, and getLength to avoid re-writing those methods. The corresponding UML class diagram now looks like this:



Note that we don't bother showing the inherited information or methods unless we're creating our own local version. If we set up our classes correctly, then any method that can be called on a Book can also be called on a Text or a Comic.

Who can see the fields of Book?

- Only an object of type Book can see them.
- All objects of any type can see them.

Yes, that's exactly right!

Since `title`, `author`, and `length` are part of the information that the derived classes **Text** and **Comic** inherit, can they see the fields of **Book**?

- Only an object of type Book can see them.
- All objects of any type derived from Book can see them.

Yes, that's exactly right!

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

```
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 won't compile for two reasons:

1. 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, and
2. we're not invoking the `Book` constructor to initialize its fields, and it doesn't 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 consist of:

1. allocate space for the object
2. **invoke the superclass constructor to build the superclass portion of the object (NEW!)**
3. construct the fields
4. run the constructor body

Running the `Book` constructor in the `Text` constructor body is too late (just as it was for constants or references) since the 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.

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

The full version of this example can be found in `lectures/c++/07-inheritance/example1`. Notice that right now, all of the subclasses are using the inherited `isHeavy` method from `Book`. We'll change that next since our definition of what constitutes heaviness changes based upon the class.

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

## Version 2

There are good reasons to keep superclass fields inaccessible to subclasses; but, if you want to give only the derived classes access, you can declare the data fields and/or methods as protected. Our `Book` declaration would then become:

```
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've made the `getLength` method `protected` since it will only be called by the `isHeavy` method. This way, only the base class and the derived classes can call it. We generally don't want to make the data fields `protected`, since then we have no control over how the information can be changed by the subclasses. Instead, we provide `protected` methods to control how the derived classes can manipulate the `private` information.

Our end goal is to find a way to write `isHeavy` so that we can correctly determine if the item is heavy without needing to know what the underlying type of the object is.

Our definition for what constitutes "heavy" is as follows:

- a book over 200 pages
- a text over 400 pages
- a comic over 30 pages

Our `isHeavy` method implementations for the various classes then become:

```

bool Book::isHeavy() const { return length > 200; }
bool Text::isHeavy() { return getLength() > 400; }
bool Comic::isHeavy() const { return getLength() > 30; }

```

This works properly for cases such as:

```

Book b{"War and Peace", "Tolstoy", 5000};
Text t{"Algorithms", "CLRS", 401, "C"};
Comic c{"Robin and his Sidekick Batman", "Robin", 21, "Robin"};

cout << "First book: " << b.getTitle() << "; " << b.getAuthor()
    << "; " << (b.isHeavy() ? "heavy" : "not heavy") << endl;

cout << "Second book: " << t.getTitle() << "; " << t.getAuthor()
    << "; " << (t.isHeavy() ? "heavy" : "not heavy") << endl;

cout << "Third book: " << c.getTitle() << "; " << c.getAuthor()
    << "; " << (c.isHeavy() ? "heavy" : "not heavy") << endl;

```

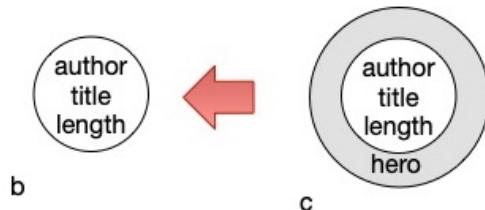
in that all three items correctly returned that they are heavy. Unfortunately, the code from our `main.cc` exposes other problems. Let's look at the first one:

```
Comic c{"Robin Rescues Batman Twice", "Robin", 40, "Robin"};  
Book b = c;  
  
cout << "The comic book: " << c.getTitle() << "; " << c.getAuthor()  
<< "; " << (c.isHeavy() ? "heavy" : "not heavy") << endl;  
  
cout << "The book: " << b.getTitle() << "; " << b.getAuthor()  
<< "; " << (b.isHeavy() ? "heavy" : "not heavy") << endl;
```

when compiled and run produces the output:

```
The comic book: Robin Rescues Batman Twice; Robin; heavy  
The book: Robin Rescues Batman Twice; Robin; not heavy
```

The first line is correct, since `c` is heavy for a `Comic`; however, the second line treated `b` as not being heavy since it ran `Book::isHeavy`. When we initialized the object `b` by copying in the contents of `c`, we've performed an action called **object slicing**. It might help us understand the problem to think of the `Comic` object `c` as having an inner "Book core", with an outer layer of whatever makes it a `Comic`.



When we copy (whether by assignment or constructor, though it's a copy constructor call in this example), we're effectively trying to squeeze a larger object into a smaller amount of memory. We lose the information that makes `c` a `Comic`, and `b` only sees the `Book` portion. So, we can't successfully store objects of our subclass types into variables of the superclass type.

What happens if we use pointers instead?

```
Comic *pc = new Comic{"Spiderman Unabridged", "Peter Parker", 100, "Spiderman"};  
Book *pb = pc;  
  
cout << "The comic book ptr: " << pc->getTitle() << "; " << pc->getAuthor()  
<< "; " << (pc->isHeavy() ? "heavy" : "not heavy") << endl;  
  
cout << "The book ptr: " << pb->getTitle() << "; " << pb->getAuthor()  
<< "; " << (pb->isHeavy() ? "heavy" : "not heavy") << endl;
```

Let's take a look at what happens in this case by using the debugger.

0:00 / 3:11

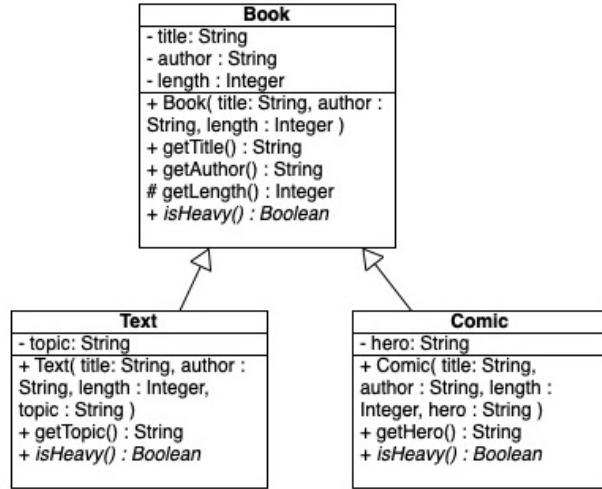


So, this *almost* works!

The full version of this example can be found in [lectures/c++/07-inheritance/example2](#).

## Version 3

The C++ mechanism that lets us force the program to dynamically examine the actual type of the pointed to object (or the actual type of the reference-bound object) rather than rely upon the declared type of the pointer consists of declaring the `isHeavy` method to be **virtual** in the parent class. By default, even if we don't repeat the keyword **virtual** on the method signature in the subclass, it is *still* considered to be **virtual**. 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**. Note that the method signatures must match exactly! So if the method is **constant** in the parent, it must be **constant** in the child as well. 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 (not to the signature of the method in the parent class!).



The respective class header files are then changed to look like the following, assuming the appropriate header guards and inclusion of the `<string>` library:

|  |  |  |
|--|--|--|
| <pre> class Book {     std::string author, title;     int length; protected:     int getLength() const; public:     Book( const std::string &amp;author,           const std::string &amp;title,           int length );     std::string getTitle() const;     std::string getAuthor() const;     <b>virtual</b> bool isHeavy() const; }; </pre> | <pre> class Text : public Book {     std::string topic; public:     Text( const std::string &amp;author,           const std::string &amp;title,           int length,           const std::string &amp;topic );     bool isHeavy() const <b>override</b>;     std::string getTopic() const; }; </pre> | <pre> class Comic : public Book {     std::string hero; public:     Comic( const std::string &amp;author,            const std::string &amp;title,            int length,            const std::string &amp;hero );     bool isHeavy() const <b>override</b>;     std::string getHero() const; }; </pre> |
|--|--|--|

It would be perfectly fine to have instead declared `isHeavy` in the children classes as:

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

Note that we leave the keywords `virtual` and `override` off of the implementations, though:

```

bool Book::isHeavy() const { return length > 200; }
bool Text::isHeavy() const { return getLength() > 400; }
bool Comic::isHeavy() const { return getLength() > 30; }

```

Now when we compile and run using the same pointer example from the previous version:

```
Comic *pc = new Comic{"Spiderman Unabridged", "Peter Parker", 100, "Spiderman"};
Book *pb = pc;

cout << "The comic book ptr: " << pc->getTitle() << ";" << pc->getAuthor()
    << ";" << (pc->isHeavy() ? "heavy" : "not heavy") << endl;

cout << "The book ptr: " << pb->getTitle() << ";" << pb->getAuthor()
    << ";" << (pb->isHeavy() ? "heavy" : "not heavy") << endl;
```

we end up with the correct output for both versions:

```
The comic book ptr: Spiderman Unabridged; Peter Parker; heavy
The book ptr: Spiderman Unabridged; Peter Parker; heavy
```

The full version of this example can be found in [lectures/c++/07-inheritance/example3](#).

## Version 4

This version of the program takes the previous idea, that using pointers (or references) lets us hold objects of either the parent or child types without slicing them, and uses that to build 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's one of the chief benefits to inheritance! This is also why a function `void f(ifstream & in);` can be passed an `ifstream` by reference instead of an `istream`—`ifstream` is a subclass of `istream`.

We've also added another virtual method to our classes, that is used to return true if the item is one of our favourites. What makes an item a favourite is different for each class.

```
// 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++"; }
```

Now, our `main` routine creates an array of (`Book*`), initialized to various books, texts, and comics. It then calls the `printMyFavourites` function, that iterates over the array and calls `favourite` on each object. If `favourite` returns true, the title of the item is printed. Then all of the dynamically allocated memory is freed.

```
// 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];
}
```

The full version of this example can be found in [lectures/c++/07-inheritance/example4](#).

# Arrays of polymorphic objects

If you want to use polymorphism in combination with arrays (or any other container type), it turns out that this will only work correctly if your array holds pointers (an array of references requires that all references be initialized upon the array's declaration, which isn't feasible for large arrays).

Let's use a small example to illustrate the problem. (The code can be found in `lectures/c++/07-inheritance/badArrayObj.cc`).

```
#include <iostream>

class One {
    int x, y;
public:
    One( int x = 0, int y = 0 ) : x{x}, y{y} {}
    int getX() const { return x; }
    int getY() const { return y; }
};

class Two : public One {
    int z;
public:
    Two( int x = 0, int y = 0, int z = 0 ) : One{x,y}, z{z} {}
    int getZ() const { return z; }
};

std::ostream & operator<<( std::ostream & out, const Two & obj ) {
    out << "(" << obj.getX() << "," << obj.getY() << "," << obj.getZ() << ")";
    return out;
}

void f( One * a ) {
    a[0] = One{6,7};
    a[1] = One{8,9};
```

```
}

int main() {
    Two myarray[2]{ Two{0,1,2}, Two{3,4,5} };
    for (int i = 0; i < 2; ++i ) std::cout << myarray[i] << std::endl;
    f( myarray );
    for (int i = 0; i < 2; ++i ) std::cout << myarray[i] << std::endl;
}
```

When we compile and run this, our output consists of:

```
(0,1,2)
(3,4,5)
(6,7,8)
(9,4,5)
```

The first two values are reasonable, since those are the values with which we initialized our Two objects; however, the procedure f received the array as being an array of One objects. When it overwrote the contents of what it thought were two objects of type One, it over-writes only part of the second object. Our data is **misaligned!**

**Never use an array of objects polymorphically!**

**If you want a polymorphic array, use an array of pointers.**

# Polymorphism and destructors

Using polymorphism in combination with dynamic memory allocation poses a special problem.

Let's consider a small example to illustrate the problem. The code for this example can be found in [lectures/c++/07-inheritance/example5](#).

```
#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};

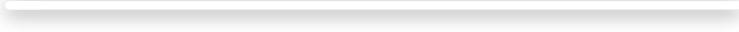
    delete xp;
}
```

You'll notice that each class in the inheritance hierarchy dynamically allocates an array and has defined a destructor to free the array. However, if we run the program through our memory-leak checking software, 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-3.13.0 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)
```

The following video clip shows where the problem occurs, and why:

0:00 / 2:51



**Always make your destructors `virtual`, even if they do nothing!**

# Abstract Classes

Sometimes, we don't have anything to write in the implementation of a virtual method in a base class. For example:

```
class Student {  
protected:  
    int numCourses;  
public:  
    virtual int fees() const; // What should this do?  
    . . .  
};  
  
// There are two kinds of student: regular and co-op.  
  
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`? Not sure... Every student should be either regular or co-op student. So, we should never create objects of the class `Student`, all objects must be created from classes `Regular` or `CoOp`.

Therefore, we can make Student be an *abstract class*. An **abstract class** cannot be instantiated and has at least one method that is not implemented. Its purpose is to organize subclasses.

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

```
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:

```
Student s; // ERROR!
```

**Subclasses of an abstract class are also abstract unless they implement all pure virtual methods.**

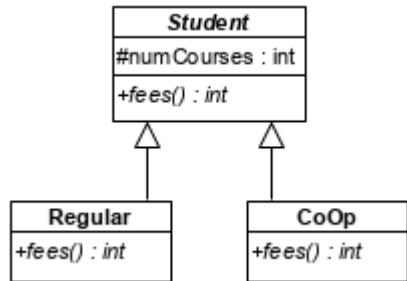
Non-abstract classes are called **concrete**:

```
class Regular: public Student { // concrete class  
public:  
    int fees() const override { return 700 * numCourses; }  
};
```

This example can be found in the directory `lectures/c++/08-purevirt/01-student` in the repository.

## UML Representation

In UML, represent virtual and pure virtual methods using italics. Represent abstract classes by italicizing the class name.



## Abstract classes and methods in other languages

*The following information will not be tested in this course, but it may be interesting to know.*

The nomenclature "pure virtual method" is used in C++ and a few other languages. However, in other languages and object-oriented design in general, methods with no implementation are just called **abstract methods**. In many other languages, the keyword `abstract` is used to declare abstract classes and methods. For example, the abstract class **Student** could be declared like this in Java:

```
public abstract class Student {
    public abstract int fees();
}
```

As you can see, both the class and the method need to be declared abstract in Java, whereas in C++ you do not need to declare a class as abstract; it's inferred automatically by the compiler if the class has at least one pure virtual method. Also, note that the keyword `virtual` does not exist in Java; all non-static public or protected methods are virtual.

# Inheritance and copy/move operations

What happens with the objects' copy and move operations when you 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 behaviour) for the `Text` part. The same is true for other compiler-provided methods.

However, you can also write your own implementation of the constructors and assignment operators. You do this by calling one of `Book`'s constructors/operators first, and then continuing with your implementation. For example, here is how you 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) {
```

```

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: 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 behaviour (i.e., they do the same as the compiler would do as the default behaviour in the initial example when we did not create specific implementations of the operations for class `Text`. Of course, you can specialize those behaviours if your class needs to do anything different.

## Partial Assignment

What happens if you use pointers to the base class 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;

```

```
// What if we do the following, using Book pointers instead of Text pointers?  
*pb2 = *pb1;
```

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 copied (i.e., the value of the field topic is not copied):

| Before the assignment:       |                             | After the assignment (*pb2 = *pb1): |                              |
|------------------------------|-----------------------------|-------------------------------------|------------------------------|
| t1                           | t2                          | t1                                  | t2                           |
| Programming for<br>Beginners | Programming for Big<br>Kids | Programming for<br>Beginners        | Programming for<br>Beginners |
| Niklaus Wirth                | Bjarne Stroustrup           | Niklaus Wirth                       | Niklaus Wirth                |
| 200                          | 300                         | 200                                 | 200                          |
| Pascal                       | C++                         | Pascal                              | C++                          |

You can find an example in the directory `lectures/c++/08-purevirt/02-copy_move` in your repository, which demonstrates the occurrence of partial assignment. Before you continue reading below to learn about potential solutions, we recommend that you take some time to read the code in this example and run it. Be sure you understand what is partial assignment and why it is occurring. Then, continuing reading below to find out how to solve it.

## Solution 1: Virtual operations

As you can see in the example in the table above, partial assignment is not desirable. If only some of the fields are copied when you try to assign different objects, this can potentially lead to bugs in your program. So, let's see how we can fix this.

One potential solution to the problem of partial assignment is making operator= virtual. So, we would change the signatures to:

```

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: `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 won't 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. Yay!

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**:

```

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)!

```

Text t { . . . };
Comic c { . . . };

```

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

In summary, if operator= is non-virtual, then we get partial assignment when assigning through base class pointers. If it is virtual, then the compiler will allow mixed assignment.

You can find an example in the directory `lectures/c++/08-purevirt/03-copy_move_virtual` in your repository, which demonstrates this solution. Before you continue reading below, we recommend that you take some time to read the code in this example and run it. Be sure you understand how partial assignment was fixed by testing `main`. Also, be sure you understand the problem of mixed assignment by running `main_bad`. As it is implemented now, `main_bad` will crash when it attempts to treat a Book or Comic as a Text object. This implementation could be improved by replacing the static casts with dynamic casts (we show this improvement on the next page). It would be safer and would not crash. However, it may still not make sense to allow programmers to try to assign a Comic book into a Text book or vice-versa, even if we take care to avoid crashing the program in this situation.

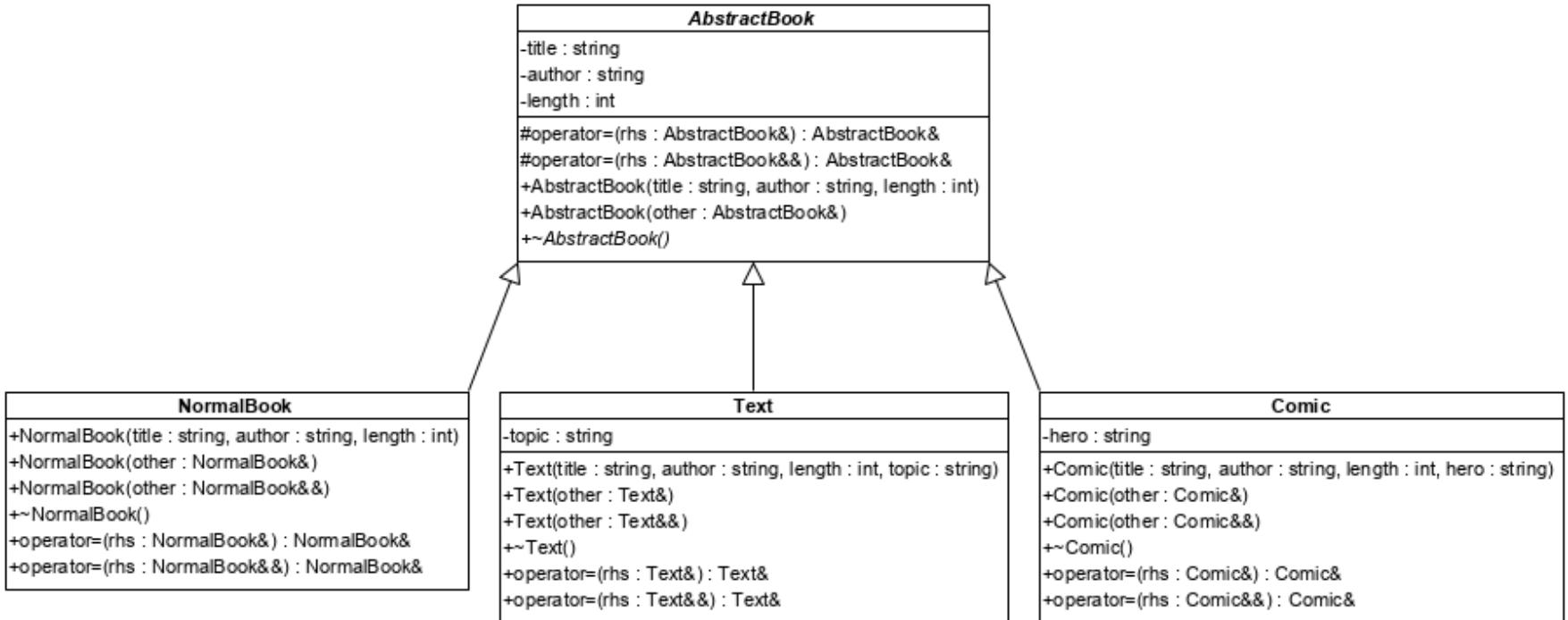
So, continuing reading below to learn about another potential solution.

## Solution 2: Abstract Superclasses

As the previous two sections demonstrated, it is not a good idea to implement a class hierarchy with non-virtual assignment/move operations or you will create a problem of partial assignment. However, it is also not a good idea to just make them virtual or you will create a problem of mixed assignment. Although it is possible to implement those solutions to avoid compile errors or run-time crashes, letting programmers do strange operations like trying to assign a Comic book into a Text book or vice-versa can lead to logical errors and bugs that are hard to debug.

**Recommendation:** all superclasses should be *abstract*.

To implement this, we rewrite the class hierarchy:



```

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
};

```

Remember that in C++, we need to have at least one pure virtual method to make the class abstract. Since we did not have any other pure virtual method, in this case, we made the destructor pure virtual. But note that this is only because we want to make the class abstract. We still need to implement the destructor because it will be called by the subclasses when the objects are destroyed (you can try removing the implementation of the destructor in the example and you will see that the code will not link):

```
// In abstractbook.cc:  
AbstractBook::~AbstractBook() {}
```

(Note: 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.)

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

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

And we can implement the concrete classes `Text` and `Comic` in the same way, i.e., calling `AbstractBook::operator=` as needed and just copying/moving the specific fields of the subclasses.

**This design prevents partial and mixed assignment 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

```

You can find an example in the directory `lectures/c++/08-purevirt/04-copy_move_abstract_superclass` in your repository, which demonstrates this solution. Be sure you understand how partial and mixed assignments were prevented by this solution. Run `main` as it is now to verify that it works correctly. Then, uncomment the lines at the end of the file and try to compile it again. Be sure you understand why those lines won't compile.

## Summary

In the examples above, we presented three different options to implement the copy and move operations together with inheritance:

1. *Public non-virtual operations* do not restrict what the programmer can do but allow partial assignment;
2. *Public virtual operations* do not restrict what the programmer can do but allow mixed assignment;
3. *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 one 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 base class 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. One of the measures to better deal with the problem of mixed assignment is presented in the next topic.

# Compilation dependencies

Now that we're 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 latter is necessary in order to break **include cycles**, where, for example, some file x.h includes file y.h, that in turns include 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.

|   |  |
|---|--|
| <pre>class B : public A {<br/>    ...<br/>};</pre>  | Must include a.h since compiler needs to know exactly how large class A is in order to determine the size of class B.  |
| <pre>class C {<br/>    A myA;<br/>};</pre>  | Must include a.h since compiler needs to know exactly how large class A is in order to determine the size of class C.  |
| <pre>class D {<br/>    A *myAptr;<br/>};</pre>  | 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.   |
| <pre>class E {<br/>    A f(A x);<br/>};</pre>   | 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 only 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.  |
| <pre>class F {<br/>    void f() {<br/>        A x;<br/>        ...<br/>        x.someMethod();<br/>        ...<br/>    }<br/>};</pre> | Because class F wrote the implementation of method F::f <i>inline</i> , 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.<br><br>This is another reason why we discourage you from writing your methods inline. |

Let's take a look at the previous code example, the program that has a stack and a queue class, each implemented using the Node class. (The code can be found in your repository under `lectures/c++/06-classes/09-separate-compilation/version3`.) While we don't have an include cycle here, we can use it as a starting point.

| File: stack.h  | File: queue.h  |
|--|--|
| <pre>#ifndef STACK_H #define STACK_H  <b>struct Node;</b> // forward declaration  class Stack {     Node * ptr; public:     Stack();     ~Stack();     bool isEmpty();     int top();     void pop();     void push( int value ); };  #endif</pre> | <pre>#ifndef QUEUE_H #define QUEUE_H  <b>class Node;</b> // forward declaration  class Queue {     Node * frontPtr, * backPtr; public:     Queue();     ~Queue();     bool isEmpty();     int front();     void dequeue();     void enqueue( int value ); };  #endif</pre> |

Note that we've removed the `#include "node.h"` from both header files. As well, even though `Node` is actually defined as a `struct`, `Queue` has forward-declared it as a `class`. This is perfectly legal when it comes to forward declarations, which just state that "such a type exists", and nothing more. Let's now take a look at the respective implementation files.

| File: stack.cc   | File: queue.cc   |
|--|--|
| <pre>#include "stack.h" <b>#include "node.h"</b>  Stack::Stack() : ptr{nullptr} {} Stack::~Stack() { while ( !isEmpty() ) pop(); } bool Stack::isEmpty() { return ptr == nullptr; } int Stack::top() { return ptr-&gt;data; }  void Stack::pop() {     Node * tmp = ptr;     ptr = ptr-&gt;next;     delete tmp; }</pre> | <pre>#include "queue.h" <b>#include "node.h"</b>  Queue::Queue() : frontPtr{nullptr}, backPtr{nullptr} {} Queue::~Queue() { while ( !isEmpty() ) dequeue(); } bool Queue::isEmpty() { return (frontPtr == backPtr &amp;&amp; frontPtr == nullptr); } int Queue::front() { return frontPtr-&gt;data; }  void Queue::dequeue() {     Node * tmp = frontPtr;     frontPtr = frontPtr-&gt;pnext;     if ( frontPtr == nullptr ) backPtr = nullptr;     delete tmp;</pre> |

```

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

```

Note that we've added the `#include "node.h"` statements to both implementation files, as well as including the actual class header.

**Most of the time, you'll find that if you use forward declarations as much as you can, this automatically removes include cycles. (And as a side-benefit, it declutters your header files.) The times when it won't are in the circumstances where inclusion is absolutely necessary i.e. either inheritance, or an object as a data field. In those circumstances, you'll have to make a design decision.**

Let's start with the case of an include cycle due to inheritance:

| File: a.h  | File: b.h  |
|--|--|
| <pre> #ifndef A_H #define A_H  #include "b.h" class A : public B {     ... };  #endif </pre> | <pre> #ifndef B_H #define B_H  #include "a.h" class B : public A {     ... };  #endif </pre> |

Conceptually, you can't have some class A inherit from some class B and have some class B also inherit from class A. That just doesn't make any sense! However, in many cases, we can replace an "is a" relationship with a "has a" relationship instead i.e. use object composition instead of inheritance. (And in fact, we generally recommend composition over inheritance since it provides flexibility at run-time.) But isn't that the same problem as before?

| File: a.h  | File: b.h  |
|--|--|
| <pre> #ifndef A_H #define A_H  #include "b.h" </pre> | <pre> #ifndef B_H #define B_H  #include "a.h" </pre> |

```
class A {  
    B myB;  
    ...  
};  
  
#endif
```

```
class B {  
    A myA;  
    ...  
};  
  
#endif
```

In the case of a data field being an object, the fix is to either make it a reference 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.

| File: a.h  | File: b.h  |
|--|--|
| <pre>#ifndef A_H<br/>#define A_H<br/><br/>class B;<br/><br/>class A {<br/>    B *myB;<br/>    ...<br/>};<br/><br/>#endif</pre> | <pre>#ifndef B_H<br/>#define B_H<br/><br/>class A;<br/><br/>class B {<br/>    A &amp;myA;<br/>    ...<br/>};<br/><br/>#endif</pre> |

Note that we'll 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.

**General rule: If there is no compilation dependency necessitated by the code, don't introduce one with extraneous #include statements; instead use forward declarations wherever possible and include the necessary headers in the implementation files.**

# Introduction to Templates

Template programming allows us to create parameterized classes (**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.

Note: templates are a large and complex topic. This course will not study all of the aspects related to templates; it's only a basic introduction. If you want to read some more about templates, <https://isocpp.org/wiki/faq/templates> has a lot of good information.

For example, let's say 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 `Nodes`:

```
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. What if we need other list types, such as a list of strings, then a list of Student objects, etc.? Should we keep creating copies of the original List code? Then, if we decide to make a change, we need to find and modify all the copies of the List? No, this is bad!!

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 <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:

```
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");
```

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

You can find the complete example of the List template, including an [Iterator](#), in the directory `lectures/c++/10-templates` in the repository. Be sure to check it and understand how it works.

**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. You can use something other than `T` if you 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, you could use `K` as the name of the type for a key, `V` for a value, `I` for an index, etc. But again, these are just conventions.

Here's an example with two template 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
```

# Introduction to the Standard Template Library (STL)

The **Standard Template Library (STL)** is a large collection of useful templates that already exist in C++.

It contains collection classes such as lists, vectors, maps, deques, etc., 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. You can't derive from them to extend their behaviour because their methods are not virtual. (But you could extend their behaviour using [Decorators](#) if needed.)

Although we will not use all the classes in the STL in this course, there are two classes that may be particularly useful for the assignments: `vector` and `map`. We will introduce these two classes in the next topics.

# STL: std::vector

The class `std::vector` is a generic (template) implementation of dynamic-length arrays.

For example, you can use it to create a dynamic-length array of integers (note that it's defined in the `<vector>` library and in the `std` namespace):

```
#include
using namespace 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.

**So, you don't pass the actual object as a parameter to `emplace_back`, you pass the arguments to be used to construct the object, and `emplace_back` calls the constructor for you. This is extremely useful when you want to create an actual object "in place" in the container rather than first creating the object and then moving (or copying) it into the container.**

For example, we could create a vector of `Vec` objects using `emplace_back`:

```
#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

## Looping over vectors

You can use a for loop to visit a vector's contents by indexing:

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

Vectors also support the [iterator](#) abstraction:

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

Or indeed:

```
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 examples, the [erase](#) method, which removes an item from a vector, works with iterators:

```
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
```

v[i] returns the i-th element of v. It's unchecked. So, if you go out of bounds, the behaviour is undefined. Therefore, you can also use v.at(i), which is a checked version of v[i]. If you go out of bounds, v.at(i) throws an [out\\_of\\_range exception](#). Of course, this comes with a performance penalty because of the additional checking, 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.

## Using vectors instead of dynamic-length arrays

Vectors are guaranteed to be implemented internally as arrays. Now that you know about vectors, (starting with Assignment 4) you should use them whenever you need a dynamic-length array (i.e., avoid using the array versions of new and delete, as using these operators usually indicates an opportunity to use a vector instead).

# 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 (template) class, so we can define any type for the keys and the values. (Well, almost any type. By default, std::map uses operator< to compare and sort the keys. So, the key must be of a type that supports operator<, unless you define a different comparison function as the optional third template parameter. Check the documentation of the std::map class for more details.)

For example, a map of string keys to int values (note that map is in the <map> library and the std namespace):

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

If a key is not found when trying to read it, such as in the last line above (highlighted in red), it is inserted and the value is default-constructed (for an int, the default value is zero).

The [erase](#) method can be used to delete a key and its associated value:

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

The [count](#) method returns one if a key is found in the map, or zero otherwise:

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

## Iterating over a Map

Iterating over a map happens in sorted key order:

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

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

# Introduction to Exceptions

Consider this simple Student class. You are writing the helper function `checkGrade`, which should report an error if the submitted grade is lower than zero or higher than 100:

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

class Student {
private:
    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 assns * 0.4 + mt * 0.2 + final * 0.4;
    }
};

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

What should happen when the function `checkGrade` detects an invalid grade?

**Problem:** Student's code can detect the error but doesn't know what to do about it. The Student class doesn't know what the main program does or what the user interface looks like. Therefore, the decision about what to do with the error can't be made inside the Student class. On the other hand, the client (the `main()` function in this example) can respond, for example, by printing

a message to the user. But it can't detect the error because it doesn't know the internal implementation details of class Student (see [Encapsulation](#)). *Error recovery is, by its nature, a non-local problem.*

**C Solution:** functions return a status code, or set the global variable errno. This leads to awkward programming and encourages programmers to ignore error-checking.

In C++ (and many other object-oriented languages), when an error condition arises, the function **raises** (or **throws**) an **exception** (note: the words throw and raise are generally used interchangeably when referring to exceptions). Then what happens? By default, execution stops. But we can write **handlers** to **catch** exceptions and deal with them.

## Throwing an Exception

Let's first 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's create a class InvalidGrade and throw an exception as an instance of this class when an invalid grade is detected:

```
class InvalidGrade {  
    // We will add fields later  
};  
  
// Helper function  
int checkGrade( int grade ) {  
    if ( grade >= 0 && grade <= 100 ) {  
        return grade;  
    } else {  
        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 appropriately.

## Handling Exceptions

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

```
// In main.cc
int main () {
    try {
        Student s{ 7899, -10, 50, 150 };
        cout << "s.grade() = " << s.grade() << endl;
    } catch (InvalidGrade) {
        cout << "Invalid grade." << 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(s). 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 the `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 won't be executed and the program will terminate immediately just as if we did not have any try-catch. Thus, *if you don't have a catch block that matches the type of the raised exception, it's just like not having any catch block at all.*

If we compile and run the updated program, we will see our error message printed to `cout` instead of the program crashing because of the exception:

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

## Passing Information in the Exception

Our error message is better than the message displayed automatically when we don't 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's 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:

```
class InvalidGrade {  
    private:  
        int grade;  
    public:  
        InvalidGrade(int grade) : grade{grade} {}  
        int getGrade() const { return grade; }  
};  
  
// Helper function  
int checkGrade( int grade ) {  
    if ( grade >= 0 && grade <= 100 ) {  
        return grade;  
    } else {  
        throw InvalidGrade{grade};  
    }  
}
```

Now, let's update our exception handler by catching the thrown object into the variable ex. Then, we can read the information in the object to include it into our error message:

```
// In main.cc  
int main () {  
    try {  
        Student s{ 7899, -10, 50, 150 };  
        cout << "s.grade() = " << s.grade() << endl;  
    } catch (InvalidGrade ex) {  
        cout << "Invalid grade: " << ex.getGrade() << 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 you can include on the object. Of course, for performance optimization, it's 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.

You can find this complete example in the folder `lectures/c++/09-exceptions/01-student` in your Git repository. And just to illustrate that we can pass additional information in the exception, we added an extra type attribute to the `InvalidGrade` class to also display what was the type of the invalid grade (i.e., "assignments", "midterm", or "final").

## Continuing the Program's Execution After the Exception

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

```
int main() {
    Student s;
    while (true) {
        try {
            // assume that we have a method to read a Student object from cin, which may throw an InvalidGrade exception:
            cin >> s;
            cout << "s.grade() = " << s.grade() << endl;
        } catch (InvalidGrade ex) {
            cout << "Invalid grade: " << ex.getGrade() << endl;
        }
        // write some condition to break the while loop
    }
}
```

Because the try-catch block is inside 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 next `Student` from the input.

# Exceptions and the Call Chain

What happens when an exception is raised in a call chain (i.e., when a function calls another one, which calls another one, etc.)? For example:

```
void f() {
    throw out_of_range{"f"};
}
void g() {
    f();
}
void h() {
    g();
}
int main() {
    try {
        h();
    }
    catch (out_of_range) {
        cerr << "Range error in main()" << 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 (*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 more details, this is what happens:

1. An exception is raised in `f`. Because `f` does not have an exception handler, its execution is interrupted and control goes back to `g`.
2. Because `g` does not have an exception handler, its execution is interrupted and control goes back to `h`.
3. Because `h` does not have an exception handler, its execution is interrupted and control goes back to `main`.

4. `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 (please check the complete example in the file `lectures/c++/08-exceptions/02-callchain/callchain.cc` in the repository and try it yourself). Note how functions `h`, `g`, and `f` started, but they never finished properly. That's 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:

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

If there is no matching handler in the entire call chain (for example, if the `main` function in the example above did not have an exception handler of the same type of the exception or no exception handler at all), the program terminates.

## 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 won't 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 when we discuss exception safety).

For example, we could modify functions f, g, and h from the example above to handle and rethrow the exception, like this:

```
void h() {
    cout << "Start h" << endl;
    try {
        g();
    } catch (out_of_range) {
        cerr << "Range error in h()" << endl;
        throw;
    }
    cout << "Finish h" << 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 (please check the complete example in the file `lectures/c++/09-exceptions/02-callchain/callchain_partial_handler.cc` in the repository and try it yourself):

```
$ ./callchain_partial_handler
Start main
Start h
Start g
Start f
Range error in f()
Range error in g()
```

```
Range error in h()
Range error in main()
Finish main
```

## Exceptions in Destructors

**WARNING! NEVER let a destructor throw an exception! By default, the program will terminate immediately (`std::terminate` will be called). Although it is possible to create a throwing destructor (you would tag it with `noexcept(false)`), you still should never do this. If the destructor is being executed during stack unwinding while dealing with another exception, you now have two active, unhandled exceptions and the program will abort immediately (again, by calling `std::terminate`).**

# Catching Exceptions With Subclasses And By Reference

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 (CustomException e) {  
    // handle custom error  
} catch (out_of_range r) {  
    // handle out of range error  
} catch (...) { // literally mean ... here  
    // handle any other exception type  
}
```

So, if an exception of type `CustomException` is raised in the try block, the first catch block will run. If an exception of type `out_of_range` 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.

**Unless you add a catch-all block, then any exception that does not match the type of the existing exception handlers will cause the program to terminate immediately.**

## Catching Exceptions By Subclasses

If your exception classes use inheritance, the class hierarchy is considered by the exception handling blocks. For example, if `E2` is a subclass of `E1`, then a `catch (E1) {}` will handle exceptions of classes `E2` and `E1`. This is because, due to inheritance, an `E2` is a type of `E1`.

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

You can also write multiple catch blocks to create different handlers for specialized exceptions and base exceptions. If you do this, note that the exception handlers are checked in order. So, the handler for the subclass needs to appear before the handler of the superclass. Otherwise, the handler of the superclass will handle the exception first.

The following example illustrates this (you can find this example in the file `lectures/c++/09-exceptions/03-catchref/catchsubclass.cc` in the repository) :

```
class E1 {};
class E2: public E1 {};

int main() {
    try {
        cout << "First block: ";
        throw E2{};
    } catch (E2) {
        // This block will run because E2 will be caught here.
        cout << "caught on E2 block" << endl;
    } catch (E1) {
        // This block will not run because the exception was already caught in the previous block.
        // However, it would run if the exception raised was of type E1.
        cout << "caught on E1 block" << endl;
    }

    try {
        cout << "Second block: ";
        throw E2{};
    }
```

```
    } catch (E1) {
        // This block will run because E2 will be caught here.
        cout << "caught on E1 block" << endl;
    } catch (E2) {
        // This block will never run. Any exception of types E1 or E2 will be handled by the block above.
        cout << "caught on E2 block" << endl;
    }
}
```

In the first try-catch block, if an E2 is raised, the first catch block will run. But if an E1 is raised, the second catch block will run. This is the correct order for the catch blocks.

The second try-catch block is written in the wrong order. Any exception of types E1 or E2 will be handled by the first catch block and the second catch block will never run in any situation. When you compile this code, the compiler will even issue a warning because the last catch block will never run (but the code still compiles despite the warning):

```
$ g++14 catchsubclass.cc
catchsubclass.cc: In function 'int main()':
catchsubclass.cc:27:3: warning: exception of type 'E2' will be caught
    catch (E2) {
    ^~~~~~
catchsubclass.cc:24:3: warning: by earlier handler for 'E1'
    catch (E1) {
    ^~~~~~
```

And if you run the compiled program, you can see that in the first try-catch block, the exception of type E2 was correctly handled by the catch (E2) block. However, in the second try-catch block, the exception E2 was handled by the catch (E1) block:

```
$ ./catchsubclass
First block: caught on E2 block
Second block: caught on E1 block
```

## Catching Exceptions By Reference

When you 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 you have polymorphic methods in the classes, the methods that will run will be those of the superclass. For example:

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

class E2: public E1 {
public:
    void f() override {
        cout << "E2" << 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 want 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".

You can check this example in the file lectures/c++/09-exceptions/03-catchref/catchref.cc in the repository. Compile it with the preprocessor variables BYREF1 and BYREF2 (i.e., use the g++ arguments -DBYREF1 - DBYREF2) to catch the exception by reference. Don't include these preprocessor variables when compiling to see the results when the exception is caught by value.

In order to always treat exception objects like the kind of object that they actually are, and avoid slicing objects 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."

## 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. For example:

```
class SomeErrorType {};
class SpecialErrorType : public SomeErrorType {};
try {
    . . . // some code that raises a SpecialErrorType exception
}
catch (SomeErrorType s) { // note that we're catching a SomeErrorType exception, not a SpecialErrorType
```

```

    ...
    throw;      // will rethrow the original SpecialErrorHandler exception
    // throw s; // would throw the original exception sliced as a SomeErrorHandler
}

```

The exception `s` actually belongs to a subclass of `SomeErrorHandler`, rather than `SomeErrorHandler` itself. The statement `throw s;` throws a new exception of type `SomeErrorHandler` (i.e., `s` is sliced into the type `SomeErrorHandler`). On the other hand, `throw;` rethrows the actual exception that was caught, and the actual type of the exception is retained.

You can check an example in the file `lectures/c++/09-exceptions/03-catchref/catchref.cc` in the repository.

```

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 you run the compiled program, you will see that it prints "E2" twice because the second time the exception is caught, it continues being of type `E2`.

If you don't include this preprocessor variable when compiling, the exception will be thrown again using `throw e;` If you run the program, you will see that it first prints "E2" and then "E1". That's 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 you use `throw e;`, the type of the *reference* (i.e., the static type of the object) determines the handler, not the actual type of the object.

# Exceptions From the C++ Standard Classes

Many of the standard classes in C++ raise exceptions when an error occurs, so you can handle the error in the way that is the most appropriate for your application.

## Input/Output Streams

By default, the [I/O streams](#) such as `cin` and `cout` don't 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 exception instead of checking the `fail()` method. For example, this program catches any failure when reading numbers from the 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
    cin.exceptions(ios::eofbit|ios::failbit);
    int i;
    while (true) {
        try {
            cin >> i;
            cout << i << endl;
        }
        catch (ios::failure &) {
            if (cin.eof()) break;
            cin.clear();
            cin.ignore();
        }
    }
}
```

You can check this example in the directory `lectures/c++/09-exceptions/04-io` in the repository. You can also read the documentation of the member function [exceptions\(\)](#) for more information about how to set it.

## Range Errors

Many classes in the [Standard Template Library](#) 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 you pass an invalid index as the parameter:

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

You can check this example in the directory `lectures/c++/09-exceptions/05-range` in the repository.

## 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 example, the following program tries to allocate too much memory and fails:

```
class C {  
    int a[1000000];  
};  
  
int main () {  
    C *b = new C[10000000];  
}
```

If we run it, we can see that it raises `std::bad_alloc`:

```
$ ./newerror  
terminate called after throwing an instance of 'std::bad_alloc'  
  what(): std::bad_alloc  
Aborted (core dumped)
```

You can check this example in the directory `lectures/c++/09-exceptions/06-newerror` in the repository.

Until now, we have just been using `new` and assuming that it would always succeed. However, as the example above showed, `new` can fail and raise an `std::bad_alloc` exception. Thus, it is a good practice to *always* write a `catch (std::bad_alloc)` block when trying to allocate dynamic memory, so your program can handle the error appropriately if the operation fails.

# Observer

The **Observer** design pattern is also known as **Dependents** or **Publish-Subscribe**.

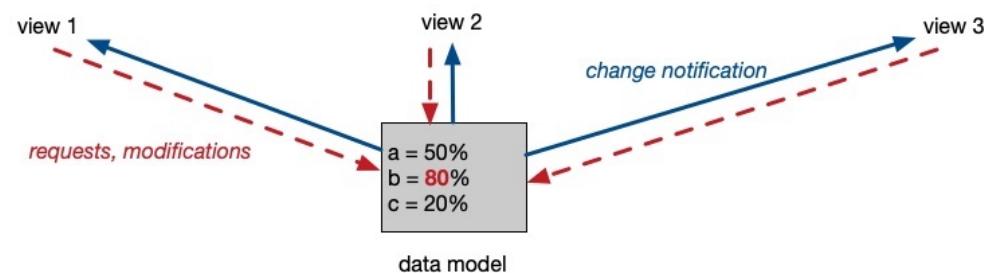
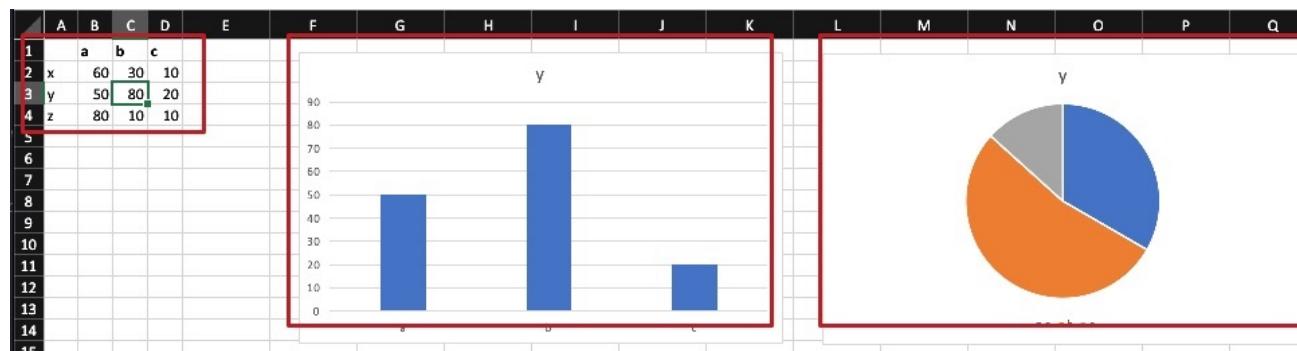
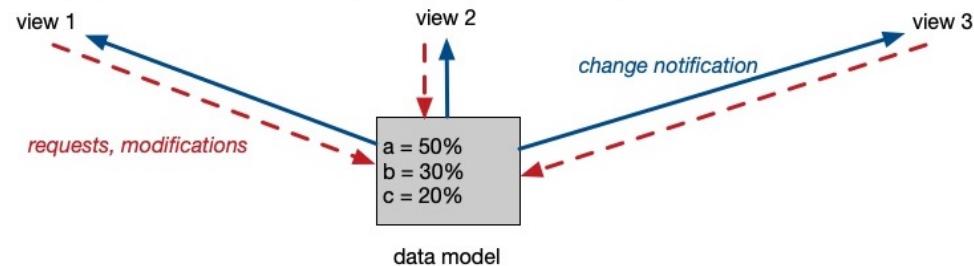
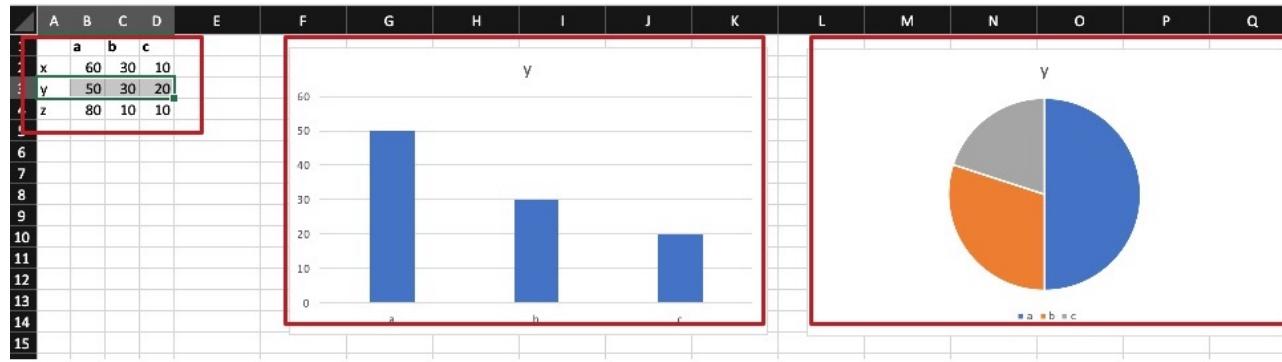
It is intended to create a one-to-many dependency between objects such that all *observers* are notified when the state of the *subject* object being observed changes.

For more reading on the topic, you can read chapter 2 of the "[Head First Design Patterns](#)" book by Freeman and Freeman.

## Examples of use

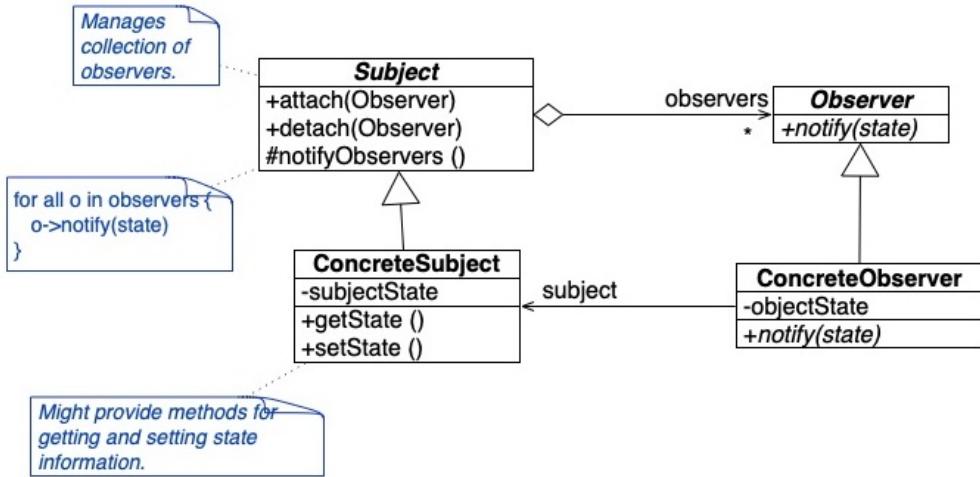
Common examples of use are:

- in graphical libraries such as the Java JDK (Swing, JavaBeans) or GTKmm, where the programmer associates *listeners* (observers) to *widgets* (subjects) such as buttons.
- Java also has a built-in Observer interface and `Observable` class in package `java.util`.
- views of an underlying data model e.g a spreadsheet that displays the information in rows/columns, a pie chart, and a bar chart; whenever the data is updated, all of the views are updated.



## General UML class model

The general form of the Observer design pattern's UML class model diagram looks like:



#### Notes:

- Subject and Observer classes are abstract base classes.
- 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. **Therefore, there is no need to duplicate the information in the concrete subclasses.**
- 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.
- The Subject has an aggregate relationship to the Observer class. Therefore, it is not responsible for destroying them when its destructor is run.
- Concrete observers may dynamically attach or detach themselves from a concrete subject during the program run-time.
- Every time the concrete subject changes state, all of the concrete observers are notified by calling Subject`::notifyObservers`, which calls their virtual `notify` method. (`Observer::notify` is **pure virtual**.)
- 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 don't 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:

1. Concrete subject's state changes.
2. Concrete subject calls `notifyObservers`, which calls the `notify` method for each concrete observer currently subscribed/attached. If we're 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.
3. If we're 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.

## 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. (Think of the notification as being a text message, something that each can receive individually, no matter where they may be in the building or what they may be doing.)

Thus, we'll 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'll define our abstract base classes as `Subject` and `Observer` to make the code more easily reusable later.

| File: subject.h   | File: observer.h  |
|---|---|
| <pre>#ifndef _SUBJECT_H_ #define _SUBJECT_H_ #include &lt;vector&gt; #include "observer.h"  class Subject {     std::vector&lt;Observer*&gt; observers;  public:     Subject();     void attach( Observer *o );     void detach( Observer *o );     void notifyObservers();     virtual ~Subject() = 0; };  #endif</pre>  | <pre>#ifndef _OBSERVER_H_ #define _OBSERVER_H_  class Observer { public:     virtual void notify() = 0;     virtual ~Observer(); };  #endif</pre> |
| File: subject.cc  | File: observer.cc   |
| <pre>#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;         }     } }</pre> | <pre>#include "observer.h"  Observer::~Observer() {}</pre>  |

```

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

```

Our concrete classes then become:

**File: 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

```

**File: horserace.h**

```

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

class HorseRace: public Subject {
    std::ifstream in;
    std::string lastWinner;

public:
    HorseRace( std::string source );
    ~HorseRace();

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

    std::string getState();
};

#endif

```

**File: 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;
}

```

**File: 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:

| File: main.cc  | Explanation   |
|--|---|
| #include <iostream><br>#include "bettor.h"<br><br>int main(int argc, char **argv) {<br>std::string raceData = "race.txt";<br>if ( argc > 1 ) raceData = argv[1];<br><br>HorseRace hr{raceData};<br><br>Bettor Larry{ &hr, "Larry", "RunsLikeACow" };<br>Bettor Moe{ &hr, "Moe", "Molasses" };<br>Bettor Curly{ &hr, "Curly", "TurtlePower" };<br><br>int count = 0;<br>Bettor *Shemp = nullptr;<br><br>while( hr.runRace() ) {<br>if ( count == 2 )<br>Shemp = new Bettor{ &hr, "Shemp", "GreasedLightning" };<br>if ( count == 5 ) delete Shemp;<br>hr.notifyObservers();<br>++count;<br>}<br>if (count < 5) delete Shemp;<br>} | <ul style="list-style-type: none"><li>Since bettor.h already includes horserace.h, we don't need to explicitly include it, though we could.</li><li>The race information is by default contained in the file race.txt, though it can be replaced by a file name supplied as a command-line argument.</li><li>The concrete subject, hr, is initialized with the race results i.e. the name of the winner for each race.</li><li>Three concrete observers, bettors Larry, Moe, and Curly are created, each with the name of the horse that they will bet upon in each race.</li><li>While there are still races being run, loop through the results.</li><li>We then dynamically create a new bettor, Shemp, who will only exist for a maximum of two more rounds. If we exit the loop before this statement gets executed, there's an extra test at the end to ensure we don't leak memory. If we exited before we created Shemp, deleting a nullptr does nothing.</li></ul> |

Our main program consists of:

| File: race.txt  | Execution  | Explanation   |
|---|--|---|
| Molasses<br>RunsLikeACow<br>GreasedLightning<br>GreasedLightning<br>Molasses<br>FinishLine'sThatWay | \$ ./main<br>Winner: Molasses<br>Larry loses.<br>Moe wins! Off to collect.<br>Curly loses.<br>Winner: RunsLikeACow<br>Larry wins! Off to collect.<br>Moe loses.<br>Curly loses.<br>Winner: GreasedLightning<br>Larry loses.<br>Moe loses.<br>Curly loses.<br>Shemp wins! Off to collect.<br>Winner: GreasedLightning | As we can see from the output, every time a winner is declared, the existing bettors state whether they have won or lost based upon their favoured horse.<br><br>Shemp appears for the third race, and disappears in the sixth. |

```
Larry loses.  
Moe loses.  
Curly loses.  
Shemp wins! Off to collect.  
Winner: Molasses  
Larry loses.  
Moe wins! Off to collect.  
Curly loses.  
Shemp loses.  
Winner: FinishLine'sThatWay  
Larry loses.  
Moe loses.  
Curly loses.
```

The full version of the code can be found in [lectures/se/02-observer](#).

## Variations

It is also possible to refine the situation such that the dependents are only notified of certain events rather than all events i.e. the dependents only subscribe to a subset of the possible notifications.

We can also vary the model by the information flow chosen. The information can be either **pushed** by the subject, or **pulled** by the observer. In the first situation, the subject would have to wrap up all of the information to pass to its observers, who thus may receive more information than they need or want. In the second situation, the subject just notifies the observer that the event has occurred, and it's up to the observer to retrieve the desired information from the subject.

# Decorator

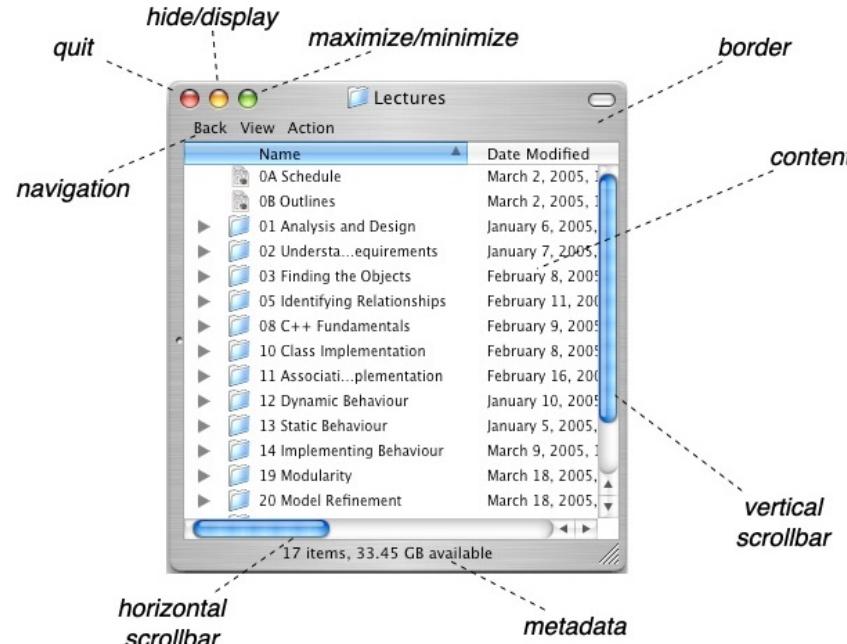
The **Decorator** design pattern is intended to let you add functionality or features to an object at *run-time* rather than to the class as a whole. These functionalities or features might also be withdrawn. Useful when extension by subclassing is impractical. We can also add features incrementally 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're connected.

For more reading on the topic, you can read chapter 3 of the ["Head First Design Patterns" book by Freeman and Freeman](#).

## Examples of use

Common examples of use are:

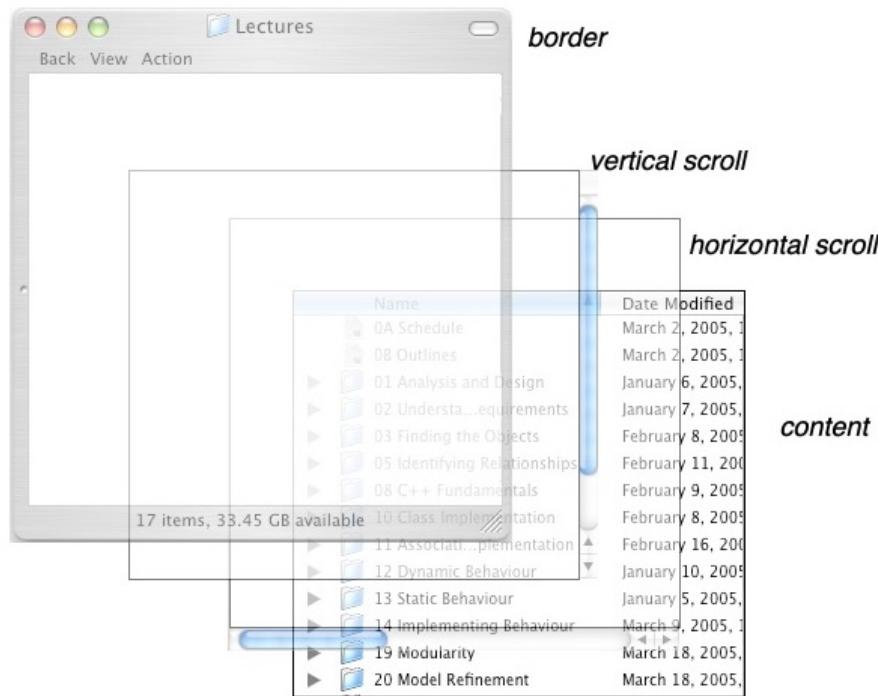
- Windows: navigation, action buttons, metadata, horizontal scroll bar, vertical scroll bar, title bar, border, content



- Telephones: call waiting, caller ID, call forwarding, distinctive ringing, automatic callback, speed dialing, redial
- Word processors: track changes, paragraph format, line spacing, highlighting, text font, text colour, sticky notes, spell checking

- Games: characters' appearance, hair, hat, glasses, beard, moustache, jewelry, piercings, tattoos, clothing

If the features are independent and can be used in any combination, then implementing the features as subclasses effectively means creating a subclass for each possible combination!



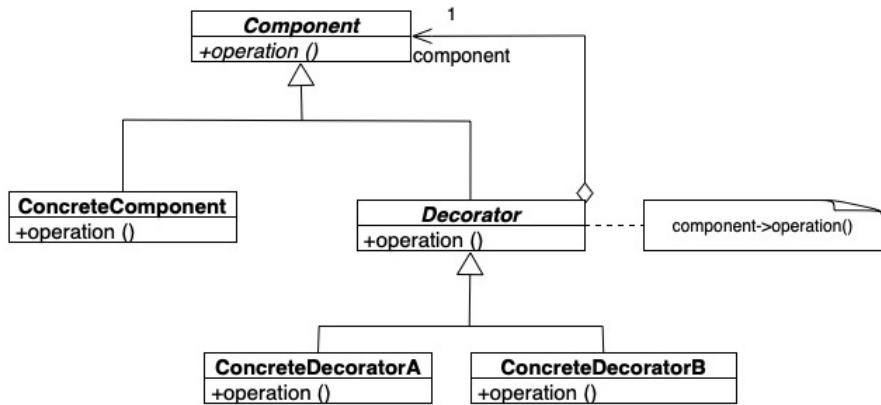
If there are  $n$  features, how many subclasses do we have to create to add the  $(n+1)$ st feature?

- $n+1$
- $2^{n+1}$
- $2^*(n+1)$
- $2^*n+1$

Right! We have  $2^n$  subclasses for  $n$  features, so  $2^{n+1}$  total for adding next feature, so double the existing number!

## General UML class model

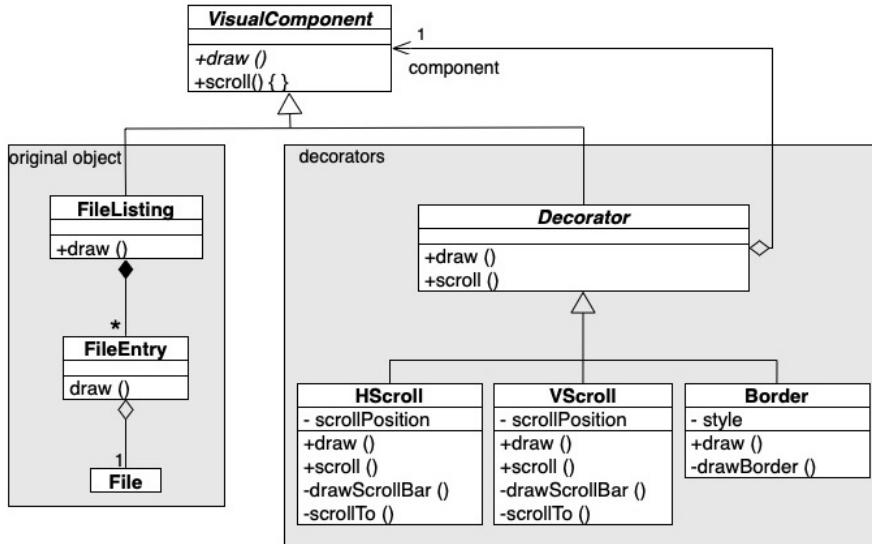
The general form of the Decorator design pattern's UML class model diagram looks like:



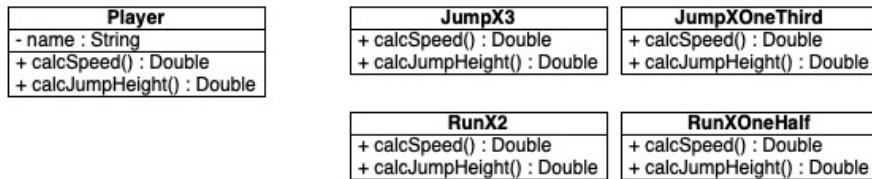
Notes:

- 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 don't need to know what is in the linked list once it's built since the operation we're invoking is *virtual* (likely *pure virtual*) in the Component class.
- Since the ConcreteComponent inherits from Component, the linked list thus terminates with a concrete component object.
- Each call to the operation in a 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.
- New functionalities are introduced by adding new subclasses, not modifying existing code, which reduces the chance of introducing errors. (This is the **Open-Closed design principle** in action! It's not testable material, but anybody interested in software development should read up on the concept—you may already be using it! There's a nice discussion on it at <https://stackify.com/solid-design-open-closed-principle/>)

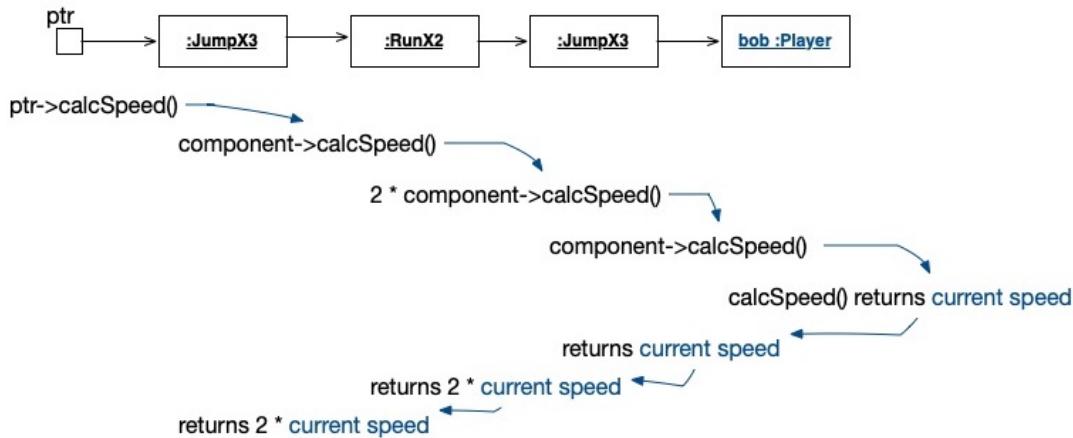
If we apply the Decorator design pattern to our window example, it would look like:



Let's walk through another example to try and explain why the Decorator design pattern is structured the way it is. Consider the situation where we're 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 don't want to write a class for every possible combination of effects, our initial general class structure would look something like:

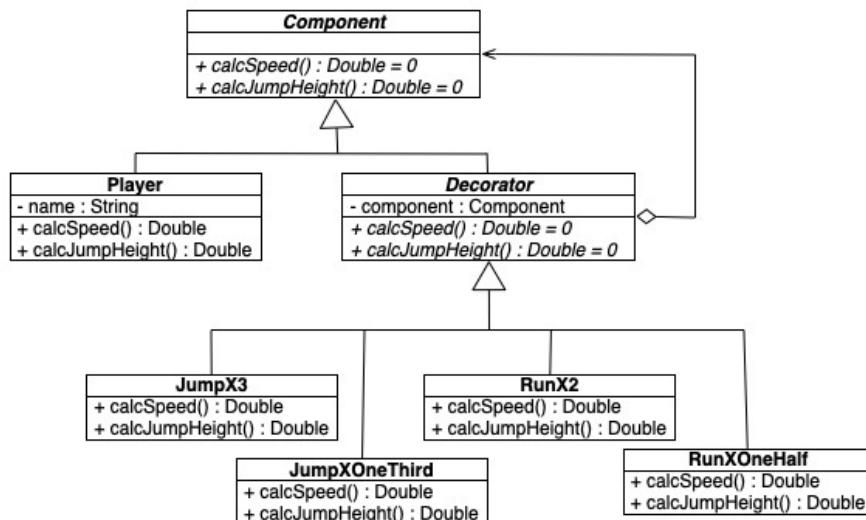


And depending upon the actual game play, we would track the current set of effects by creating a linked list of the objects currently affecting the player's speed or jump height. For example, here's a picture to illustrate a situation where the current player bob has had their speed doubled once, and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



- inherits from Component,
- holds the pointer, and
- still has pure virtual methods for calcSpeed and calcJumpHeight.

Player would then inherit from Component, while the effect classes inherit from Decorator. That lets us restructure our class model to:

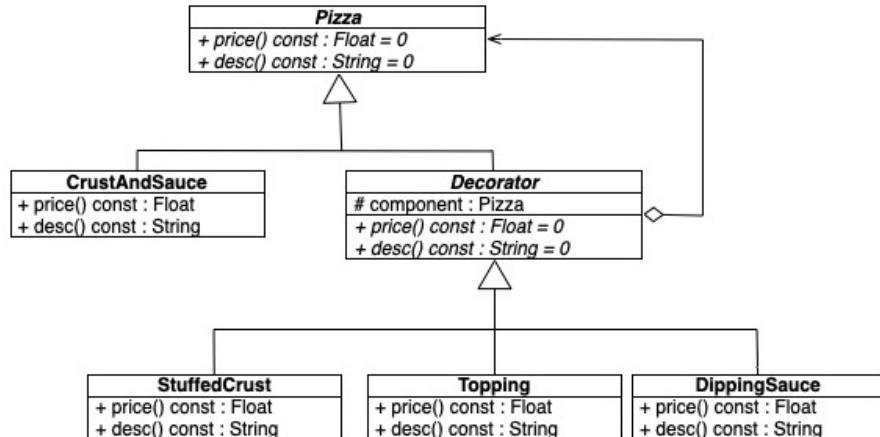


And this version looks structurally like the standard Decorator design pattern!

**Aside:** One thing we've glossed over is the Player class. Structurally speaking, this probably shouldn't be the actual player object, just a handle to the player's abilities since the information in the list might change frequently. It might also be a good idea to wrap this information using the [Strategy](#) design pattern.

## 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 basic 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'll decorate. Our toppings are \$0.75 each and are initialized with a string that describes what the topping type is e.g. mushrooms, or cheese, etc. Each stuffed crust adds \$2.69 to the price while each dipping sauce adds \$0.30. Our class model thus looks like:



Our abstract base classes consist of:

| File: pizza.h  | File: decorator.h   |
|--|---|
| <pre> #ifndef _PIZZA_H_ #define _PIZZA_H_ #include &lt;string&gt;  class Pizza { public:     virtual float price() const = 0;     virtual std::string description() const = 0;     virtual ~Pizza(); };  #endif </pre> | <pre> #ifndef _DECORATOR_H_ #define _DECORATOR_H_ #include "pizza.h"  class Decorator: public Pizza { protected:     Pizza *component; public:     Decorator( Pizza *component );     virtual ~Decorator(); };  #endif </pre> |
| File: pizza.cc   | File: decorator.cc  |
| <pre> #include "pizza.h"  Pizza::~Pizza() {} </pre>  | <pre> #include "pizza.h" #include "decorator.h"  Decorator::Decorator( Pizza *component )     : component{component} {}  Decorator::~Decorator() { delete component; } </pre>   |

Note that, despite the relationship between Pizza and Decorator being shown as aggregation since that's the norm for the Decorator design pattern, in this context composition makes more sense as the relationship, since pizza toppings aren't shared, and when a pizza is eaten (destroyed), so are its toppings.

We'll only show two of our concrete classes, CrustAndSauce and Topping. The others can be looked at in the repository where the full example is available under lectures/se/03-decorator. Our concrete classes consist of:

| File: crustandsauce.h   | File: topping.h   |
|---|---|
| <pre>#ifndef _CRUSTANDSAUCE_H_ #define _CRUSTANDSAUCE_H_ #include "pizza.h"  class CrustAndSauce: public Pizza { public:     float price() const override;     std::string description() const override; };  #endif</pre> | <pre>#ifndef _TOPPING_H_ #define _TOPPING_H_ #include "decorator.h" #include &lt;string&gt; 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</pre>              |
| File: crustandsauce.cc  | File: topping.cc  |
| <pre>#include "crustandsauce.h"  float CrustAndSauce::price() const { return 5.99; }  std::string CrustAndSauce::description() const { return "Pizza"; }</pre>  | <pre>#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-&gt;price() + thePrice; }  std::string Topping::description() const {     return component-&gt;description() + " with " + theTopping; }</pre> |

Our main program consists of:

---

## File: 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];
}
```

## Explanation

- Our pizza order will contain 3 custom pizzas.
- Note that the first object created for each pizza is a `CrustAndSauce` object, whose address is passed as the component to the preceding node in the linked list of decorations. The creation of a decorated pizza looks just like the creation of a singly-linked list because it is a singly-linked list!
- Each `CrustAndSauce` object is decorated with at least 1 item (though it could have been 0 for a plain pizza). The final object's address is what is stored in the array location.
- The overall cost of the order is going to be stored in the float variable `total`, which is initialized to 0.0.
- The description and cost of each pizza is output, and the price is added to the order's total so far.
  - Note that when I ask `myPizzaOrder[0]` for its description, I invoke `Topping::description()` for the `Topping("pepperoni")` object. That invokes the component's `Topping::description()`, which is invoked on the `Topping("cheese")` object. That in turn invokes the component's `CrustAndSauce::description()`, which returns the string "Pizza". The previous `Topping` object, `Topping("cheese")`, takes the "Pizza" string returned by `CrustAndSauce::description()` and adds "with cheese" to the string being returned. The previous `Topping` object to that, `Topping("pepperoni")`, takes the

"Pizza with cheese" string returned by the invocation of component's Topping::description() and adds "with pepperoni" to the string being returned. The final string "Pizza with cheese with pepperoni" is then printed.

- Then the order total is printed.
- Each pizza's memory is then freed.

# Smart Pointers

With your current level of experience in C and C++, you've undoubtedly noticed that memory management is your constant companion in these languages, and a constant pain, too. Luckily, the C++ STL has some features to make memory management less painful.

First, let's identify what makes memory management so difficult in the first place.

In C and C++, you 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, you 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's 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're difficult to correctly manage. In C++ in particular, control flow can be extremely complicated. Consider, for example, the following program fragment:

```
class Point {
public:
    Point(double sx, double sy) : x(sx), y(sy) {}
    double x, y;
};

[...]

double pointDistance(Point *);

[...]

double mydistance(double x, double y) {
    Point *p = new Point(x, y);
    double ret = pointDistance(p);
```

```
    delete p;
    return ret;
}
```

Without knowing the actual implementation of `pointDistance`, it appears impossible for this function to fail to delete its allocated `Point`. But, if `pointDistance` might throw an exception, then this function could end before ever reaching the `delete p;` line. If this function had multiple conditional returns, the situation would be even more complicated.

The `mydistance` example may have worked with stack allocation:

```
double mydistance(double x, double y) {
    Point p(x, y);
    return pointDistance(&p);
}
```

However, this is only the case because `p`'s lifetime is the same as its surrounding function. With a more complex object lifetime, this option would not exist. In addition, we are implicitly assuming in both cases that `pointDistance` does not store its argument, and thus allow the pointer to live past its own deletion, but there is nothing to stop it from doing so.

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.

## unique\_ptr

First, `unique_ptr`. A [unique\\_ptr](#) is exactly what it sounds like: It wraps a pointer, but it's guaranteed, so long as it's 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. We can rewrite our `mydistance` example with a `unique_ptr` like so:

```

#include <memory>

[...]

double pointDistance(std::unique_ptr<Point>);

[...]

double mydistance(double x, double y) {
    auto p = std::make_unique<Point>(x, y);
    // Here, we could access p->x and p->y, just as if p was a pointer
    return pointDistance(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\_ptrs enforce their uniqueness. Thus, a unique\_ptr cannot be copied; to do so would make it non-unique. Since unique\_ptrs cannot be copied, they cannot be passed by value.**

With `pointDistance` 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 mydistance(double x, double y) {
    auto p = std::make_unique<Point>(x, y);
    double ret = pointDistance(std::move(p));
    ret += p->x; // Segmentation fault!
    return ret;
}

```

In this way, `unique_ptrs` have a 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 (or contract) the lifetime, our values may have arbitrarily long lifetimes, like heap-allocated values.

Though `unique_ptr`s cannot be passed *by value*, they can safely be passed *by reference*. This is somewhat confusing, because we now have a reference to a pointer to an object, but keeps the uniqueness, since the `unique_ptr` itself is unique; there are merely multiple references to it. Thus, it's not uncommon for functions to take `unique_ptr<X> &` as arguments.

For many uses, this is sufficient. But, it's not uncommon to bypass `unique_ptr`s and their protections. The `get` method of `unique_ptr`s 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 you've extracted the pointer with `get`. This can still be necessary simply because existing functions expect standard pointers:

```
double pointDistance(Point *);  
[...]  
  
double mydistance(double x, double y) {  
    auto p = std::make_unique<Point>(x, y);  
    return pointDistance(p.get());  
}
```

You should always be careful when using `get` to assure that the underlying pointer is not retained after the `unique_ptr` has gone out of scope.

When considering whether to use `unique_ptr`s, you 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 pointers, and use `get`. If this question does not have a unique answer—that is, if there is no single "owner"—then you may need something more sophisticated than `unique_ptr`.

## shared\_ptr

That more sophisticated option is [shared\\_ptr](#), and once again, its purpose is exactly what it sounds like: It's a pointer that's 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, you may have multiple `shared_ptr` pointing to the same object. `shared_ptr`s can be copied and passed by value, exactly like normal \* pointers, but don't need to be explicitly deleted.

How is this possible? `shared_ptr`s use a technique called [reference counting](#) to determine when there are no more `shared_ptr`s pointing to a given object. When you 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, there are no more `shared_ptr`s pointing to the object, so it's deleted. All of this reference counting is handled automatically in the constructor, copy constructor, and destructor for `shared_ptr` itself, so usually, all a programmer needs to do is use `shared_ptr`s instead of \* pointers.

Let's improve our `mydistance` example by using `shared_ptr`s:

```
double pointDistance(std::shared_ptr<Point>);

[...]

double mydistance(double x, double y) {
    auto p = std::make_shared<Point>(x, y);
    return pointDistance(p);
}
```

Unlike the `unique_ptr` example, `p` does not need to be moved, but can simply be shared, so `p` remains usable:

```
double mydistance(double x, double y) {
    auto p = std::make_shared<Point>(x, y);
    double ret = pointDistance(p);
    ret += p->x; // No problem
```

```
    return ret;
}
```

Unlike the new-based example, no surprising control flow, such as exceptions, will cause us to fail to delete p; it is deleted no matter how the shared\_ptrs to it go out of scope. Unlike the earlier examples, pointDistance may safely retain the pointer; p is deleted only when the *last* shared\_ptr to it goes out of scope, reducing the reference count to 0. Consider some possible implementations of pointDistance, and the mydistance function annotated with the shared pointer's reference count at each point:

```
double pointDistance(std::shared_ptr<Point> p2) {
    return sqrt(p2->x*p2->x + p2->y*p2->y);
}

double mydistance(double x, double y) {
    auto p = std::make_shared<Point>(x, y); // p's reference count is 1
    double ret = pointDistance(p); // During pointDistance, p's reference count is 2
    // As soon as pointDistance ends, p's reference count is reduced to 1 again
    ret += p->x;
    return ret; // After returning, p's reference count is reduced to 0, so the Point is deleted
}

---

std::shared_ptr<Point> lastPoint;

double pointDistance(std::shared_ptr<Point> p2) {
    // Calling this function increases p2's reference count by 1
    double dx = p2->x - lastPoint->x,
           dy = p2->y - lastPoint->y;
    double ret = sqrt(dx*dx + dy*dy);

    /* This assignment increases p2's reference count by 1, but also decreases
     * the reference count of the previous object pointed to by lastPoint. If
     * the previous reference count was reduced to 0, then the previous
     * lastPoint would additionally be deleted. */
    lastPoint = p2;
```

```

        return ret; // Returning reduces p2's reference count by 1, but the increase from lastPoint remains
    }

double mydistance(double x, double y) {
    auto p = std::make_shared<Point>(x, y); // p's reference count is 1

    /* Calling pointDistance increases p's reference count to 2 by copying it
     * to p2. pointDistance then increases it to 3, by copying it to lastPoint. */
    double ret = pointDistance(p);
    /* When pointDistance returns, p's reference count is reduced to 2, because
     * p2 goes out of scope. */

    return ret;
    /* When this function returns, p's reference count is reduced to 1, so it
     * is not deleted. It is still referenced by lastPoint. Its reference count
     * only reaches 0 when lastPoint is replaced, and at that point, it is
     * deleted. */
}

```

## Caveats of shared\_ptrs

If `shared_ptr`s are so great, why have we even bothered with normal pointers and `unique_ptr`s? The answer, quite simply, is that they're not so great; `shared_ptr`s 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::shared_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. Why? Consider this example:

```
void graphWork() {
    auto root = std::make_shared<GraphNode>("Node 1");
    auto n2 = std::make_shared<GraphNode>("Node 2");
    root->addVertex(n2);
    n2->addVertex(root);

    [... do some graph work ...]

    return;
}
```

Adding the vertex from `root` to `n2` increases `n2`'s reference count to 2, because it now has the additional reference from `root`'s `vertices`. Adding the vertex from `n2` to `root` increases `root`'s reference count to 2, because it now has the additional reference from `n2`'s `vertices`. Returning from `graphWork` reduces each of them by 1, because the `root` and `n2` variables have gone out of scope. But,  $2 - 1 = 1$ , and 1 is not 0. These two objects 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 your data so that no such cycles exist, or (b) not to use reference counting (`shared_ptrs`) 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 in the next section.

The second major caveat with `shared_ptrs` is that counting references is extra work, and thus a performance penalty. In many cases, this work is negligible, but if a part of your code with critical performance is constantly passing `shared_ptrs` 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 you to get the underlying pointer. Like `unique_ptr`, the underlying pointer is unprotected, and you can only guarantee that it isn't deleted by making sure you retain a `shared_ptr` for at least as long as the `*` pointer. But, because of these downsides with `shared_ptr`s, it's 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 the reference count. This is occasionally useful for debugging, but should otherwise never be used.

## Summary

Both `unique_ptr`s and `shared_ptr`s are so-called *smart pointers*. Knowing which to use, and whether they're adequately smart for your 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++.

In the next part, we will expand upon the principle behind smart pointers, RAII.

# RAII: Resource Acquisition Is Initialization

Smart pointers are useful for enabling a common C++ idiom: **Resource Acquisition Is Initialization**, or RAII. Resources that must be explicitly cleaned up, such as heap-allocated objects, are bound to resources which are cleaned up automatically. Don't read too much into the name: It's actually cleanup that is simplified by RAII, not acquisition.

The RAII concept extends further than simply using smart pointers, though. More generally, it is the concept that you should always design your classes so that object lifetimes are bound to other objects. Let's recall our `GraphNode` example from the previous module. Since the previous version misused `shared_ptrs`, we'll 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_ptrs` 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.size());
        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 GraphNodes referring directly to other GraphNodes, which created the problem of cyclic dependencies, GraphNodes refer only to indices. The association between indices and actual GraphNodes is only in a single, surrounding Graph object, and vertices can only be resolved to nodes using both GraphNodes 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 GraphNodes come from a single source: the Graph.

The principle of RAI is to build structures such as these, that assure that the lifetimes of all objects in a system are connected.

The 'R' in RAI stands for "Resource", and heap memory is not the only resource that a program may interact with. The RAI principles are also applied to any resource which must be properly cleaned up. Consider, for example, files:

```

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 `GraphNode`s 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's 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 the four likely ways that a `vector` of `Graphs` would be stored.

The first, `d`, adheres to RAII principles, but might be difficult to use. Since the element type of the `vector` is `Graph`, destroying the `vector` will destroy every `Graph`. However, we probably don't intend to copy `Graphs`, and using `Graph` as the basic type encourages such copying: the obvious way to use this `vector` would be `Graph e = d[x];`, which copies the `Graph`. Worse yet, because a `Graph` contains a `vector` of `shared_ptrs` to `GraphNode`s, copying the `Graph` copies a `vector`, and increments a reference count for every `GraphNode`! That single, innocuous-looking operation is actually exceedingly expensive.

The second, `p`, does not adhere to RAII principles. When `p` is destroyed, all of its *pointers* are lost, but the underlying, heap-allocated `Graphs` are not deleted. In order to properly clean up `p`, we would need to loop over it and delete

every element. Worse, if an exception were thrown while `p` was in scope, and we didn't explicitly clean up `p` in the exception handler, then this memory would be leaked.

The third, `u`, adheres to RAII principles, but might be difficult to use. Because a `unique_ptr` is, well, unique, we cannot copy the pointer out of the vector to use it conveniently. That is, `auto g = u[x];` will fail. We can move the graph out of the vector, with `auto g = move(u[x]);`, but this will set the pointer to `nullptr` in the vector, rather than actually removing it (this is how `move` works with `unique_ptrs`). Alternately, we could extract standard pointers with, e.g., `u[x].get()`, so long as we're careful with the lifetime of these pointers.

The last, `s`, adheres to RAII principles and is 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.

Next, we will discuss how we can use RAII principles to write exception-safe functions.

# Pimpl Idiom

In the **Pimpl Idiom**, the term "Pimpl" is short for "**P**ointer to **Im**plementation". Since it's an **idiom** it is a programming technique, not a design pattern.

## Motivation

We've already discussed how one of the strengths of OOP is the ability to encapsulate information and keep it out of the client's hands. However, by its very nature, our header files *have* to show the private data fields and methods, even if they're accessible to the client. They are part of the structure, and take up space. However, this also means that any time we change implementation details, even of the private information and/or methods, the client *has* to recompile all of their code that includes the header file. There's a good discussion of why this is true at <https://en.cppreference.com/w/cpp/language/pimpl>.

In other words, it's not just sufficient to link in the new .o file, which would be the best we could hope for since it takes less time than recompiling everything and linking again.

## Code example

Our motivating example is the Xwindow class, used to provide some simple graphical drawing features in [X11](#) (original graphics library for UNIX). It is often used in the course final project, and sometimes on the last assignment or two. (If you want something more like the Java packages, there's a C++ library called [gtkmm](#), though its learning curve is pretty steep.)

Our initial class is defined as:

| File: xwindow.h   | File: xwindow.cc   |
|---|--|
| <pre>#ifndef __WINDOW_H__ #define __WINDOW_H__ #include &lt;X11/Xlib.h&gt; #include &lt;string&gt;  class Xwindow {     Display *d;     Window w;     int s;     GC gc;     unsigned long colours[10]; public:     Xwindow(int width=500, int height=500); // Displays the window.     ~Xwindow(); // Destroys the window.     Xwindow(const Xwindow&amp;) = delete; // Disallow copy and assignment     Xwindow &amp;operator=(const Xwindow&amp;) = delete;</pre> | <pre>#include &lt;X11/Xutil.h&gt; #include &lt;iostream&gt; #include &lt;cstdlib&gt; #include &lt;string&gt; #include &lt;unistd.h&gt; #include "window.h"  // Allocate the static constant space. const int Xwindow::MAX_NUM_COLOURS = 5;  Xwindow::Xwindow(int width, int height) {     d = XOpenDisplay(nullptr);     ... } Xwindow::~Xwindow() {</pre> |

```

static const int MAX_NUM_COLOURS; // Available colours.
enum class Colour {White=0, Black, Red, Green, Blue};

// Draws a rectangle
void fillRectangle(int x, int y, int width, int height,
    Xwindow::Colour colour=Colour::Black);

// Draws a string
void drawString(int x, int y, std::string msg);
};

#endif

XFreeGC(d, gc);
XCloseDisplay(d);
}

void Xwindow::fillRectangle(int x, int y, int width, int height,
    Xwindow::Colour colour)
{
    XSetForeground(d, gc, colours[static_cast<int>(colour)]);
    XFillRectangle(d, w, gc, x, y, width, height);
    XSetForeground(d, gc, colours[static_cast<int>(Colour::Black)]);
}

void Xwindow::drawString(int x, int y, std::string msg) {
    XDrawString(d, w, DefaultGC(d, s), x, y,
        msg.c_str(), msg.length());
}

```

Note that since the set of available colours is defined as an [enumerated class](#), it's more type-safe, but cannot be treated as integer even though that's the underlying type. We'll use the C++ `static_cast` to convert it to an integer since the X11 library expects that for the colour.

We can replace all of the private data fields with a pointer to a class that we forward-declare in our header file. Since we're no longer declaring data fields whose types are defined in the X11 library, we can also move that particular include statement to our implementation file.

We now need to define in our implementation what `XWindowImpl` actually looks like. We can either define it as a `class`, in which case we have to duplicate the interface of `Xwindow`, or we can define it as a `struct` and take advantage of the public accessibility of the data fields. (Yes, it's perfectly legal in C++ to redefine a `class` as a `struct` and vice versa in the implementation file.) Since the client never sees the contents of the implementation file, and just receives the header file and the `.o` file, they won't know how we actually implemented things.

Every reference to the original data fields in the implementation file now need to go through the pointer to the `struct` to access the fields declared there. Thus, our files are restructured into:

| File: xwindow.h   | File: xwindow.cc  |
|---|---|
| <pre> #ifndef __WINDOW_H__ #define __WINDOW_H__ #include &lt;string&gt;  class XWindowImpl; // forward declaration  class Xwindow {     XWindowImpl * pImpl; // pointer to our implementation class  public:     Xwindow(int width=500, int height=500); // Displays the window.     ~Xwindow(); // Destroys the window. </pre> | <pre> #include &lt;X11/Xlib.h&gt; #include &lt;X11/Xutil.h&gt; #include &lt;iostream&gt; #include &lt;cstdlib&gt; #include &lt;string&gt; #include &lt;unistd.h&gt; #include "window.h"  // Allocate the static constant space. const int Xwindow::MAX_NUM_COLOURS = 5;  <b>struct XWindowImpl {</b> </pre> |

```

Xwindow(const Xwindow&) = delete; // Disallow copy and assignment
Xwindow &operator=(const Xwindow&) = delete;

static const int MAX_NUM_COLOURS; // Available colours.
enum class Colour {White=0, Black, Red, Green, Blue};

// Draws a rectangle
void fillRectangle(int x, int y, int width, int height,
    Xwindow::Colour colour=Colour::Black);

// Draws a string
void drawString(int x, int y, std::string msg);
};

#endif

```

```

Display *d;
Window w;
int s;
GC gc;
unsigned long colours[10];
};

Xwindow::Xwindow(int width, int height) : pImpl{new XWindowImpl} {
    pImpl->d = XOpenDisplay(nullptr);
    ...
}

Xwindow::~Xwindow() {
    XFreeGC(pImpl->d, pImpl->gc);
    XCloseDisplay(pImpl->d);
    delete pImpl;
}

void Xwindow::fillRectangle(int x, int y, int width, int height,
    Xwindow::Colour colour)
{
    XSetForeground(pImpl->d, pImpl->gc,
        pImpl->colours[static_cast<int>(colour)]);
    XFillRectangle(pImpl->d, pImpl->w, pImpl->gc, x, y, width, height);
    XSetForeground(pImpl->d, pImpl->gc,
        pImpl->colours[static_cast<int>(Colour::Black)]);
}

void Xwindow::drawString(int x, int y, std::string msg) {
    XDrawString(pImpl->d, pImpl->w, DefaultGC(pImpl->d, pImpl->s),
        x, y, msg.c_str(), msg.length());
}

```

An alternate implementation choice would be to nest the `XWindowImpl` class in the `Xwindow` class itself. Since they're so closely related, this would be quite reasonable, and further limits access since it could be declared as part of the private class information. That way, even if the client guesses as to implementation details, they have no way of accessing or creating instances of the "pimpl" class. It also helps reduce potential naming conflicts.

File: `xwindow.h`

```

#ifndef __WINDOW_H__
#define __WINDOW_H__
#include <string>

class Xwindow {
    class XWindowImpl; // forward declaration
    XWindowImpl * pImpl; // pointer to our implementation class
}

```

File: `xwindow.cc`

```

...
struct Xwindow::XWindowImpl {
    Display *d;
    Window w;
    int s;
    GC gc;
    unsigned long colours[10];
}

```

```

public:
    Xwindow(int width=500, int height=500); // Displays the window.
    ~Xwindow(); // Destroys the window.
    Xwindow(const Xwindow&) = delete; // Disallow copy and assignment
    Xwindow &operator=(const Xwindow&) = delete;

    static const int MAX_NUM_COLOURS; // Available colours.
    enum class Colour {White=0, Black, Red, Green, Blue};

    // Draws a rectangle
    void fillRectangle(int x, int y, int width, int height,
        Xwindow::Colour colour=Colour::Black);

    // Draws a string
    void drawString(int x, int y, std::string msg);
};

#endif
};

Xwindow::Xwindow(int width, int height)
    : pimpl{new Xwindow::XWindowImpl}
{
    ...
}

...

```

You can find these examples in your repository, under `lectures/se/04-pimpl`.

## Variations

A natural extension to this idea would be to take the "pimpl" class, and turn this into a superclass if there are multiple possible window implementations. As soon as we add a class hierarchy, this leads us to the **Bridge** design pattern, which we will see in the next module.

# Exception Safety

Note: this topic expands on [exceptions](#). If you have not done so yet, please complete reading that module before continuing on this page.

Consider the following example function:

```
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 be deleted and that memory will be leaked. Let's see this with `valgrind`. You can find this example on the file `lectures/c++/09-exceptions/07-safety/01-except_leak.cc` in the repository and try it for yourself.

```
$ 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 (the size of an object of `MyClass`) allocated by operator new in `f()`.

## Solution 1: an exception handler

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

```
void f() {  
    MyClass mc;  
    MyClass *p = new MyClass;  
    try {  
        g();  
    }  
    catch (...) {  
        delete p; // this works, but duplicates the line of code that we already had below  
        throw;  
    }  
    delete p;  
}
```

We can check with valgrind that this worked. You can find this example on the file `lectures/c++/09-exceptions/07-safety/02-except_handler.cc` in the repository and try it for yourself.

```
$ 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's 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 (in this example, only the line `delete p;`, but the duplication could be much larger for a complex program).

It would be better if we could guarantee that something (here, `delete p;`) will happen, no matter how we exit `f` (normally or by exception). It turns out that, as we saw in the previous topic, that's exactly what [RAII](#) does—it guarantees that resources will be freed at the end of the function. So, let's see how we can take advantage of that to provide *exception safety* to our functions.

## Exception Safety

Generally speaking, there are three levels of [exception safety](#) for a function `f`:

1. **Basic guarantee:** if an exception occurs, the program will be in some valid, unspecified state. Nothing leaked, class invariants maintained.
2. **Strong guarantee:** if `f` throws or propagates an exception, the state of the program will be as if `f` had not been called.
3. **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. Therefore, unless the function documentation mentions that it provides a strong or a no-throw guarantee, we can always expect that it will offer at least the basic guarantee. This also means that, when you are writing a function, you must implement it to at least offer a basic exception guarantee. This means that any function that you write should at least guarantee that, if an exception occurs, nothing will be leaked and class invariants will be maintained (unless you can guarantee that the function will never throw an exception).*

Let's discuss each one of these levels of exception safety in more detail.

## 1. 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`:

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

As you can see, 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.

Again, we can check with `valgrind` that this worked. You can find this example on the file `lectures/c++/09-exceptions/07-safety/03-except_raii.cc` in the repository and try it for yourself.

```
$ valgrind --leak-check=full ./03-except_raii
==175== Memcheck, a memory error detector
==175== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==175== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==175== Command: ./03-except_raii
==175==
==175== HEAP SUMMARY:
==175== in use at exit: 0 bytes in 0 blocks
==175== total heap usage: 3 allocs, 3 frees, 72,844 bytes allocated
==175==
==175== All heap blocks were freed -- no leaks are possible
==175==
==175== For counts of detected and suppressed errors, rerun with: -v
==175== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

As expected, no memory leaks this time!

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.

Alternatively, the function can be implemented with a strong exception guarantee.

## 2. 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. Let's see an example:

```
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)
    }
};
```

Is *C*::*f* exception-safe? Let's see:

1. if *a.g()* throws, nothing has happened yet, so OK.
2. if *b.h()* throws, 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 no, *C*::*f* is probably not exception-safe (i.e., it does not offer a strong guarantee).

You can find this example on the file `lectures/c++/09-exceptions/07-safety/04-except_no_strong.cc` in the repository, where `a.g()` runs normally but `b.h()` throws. If we run it:

```
$ ./04-except_no_strong
Before c.f():
a.x = 0, b.y = 0

Now executing c.f()...
Exception caught.

After c.f():
a.x = 1, b.y = 0
```

Note that after trying to execute `c.f()`, the value of `a` was modified, but not the value of `b`. Therefore, `c.f()` made only part of its intended state changes permanent, but not all of them. This is not desirable and, although `C::f` still offers a basic guarantee, *it does not offer a strong guarantee*.

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(); // If these throw, original a and b still intact
        btemp.h();
        a = atemp; // But what if copy assignment throws??
        b = btemp;
    }
};
```

As you can see, we now 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 two 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 have been modified. Therefore, this solution only works if the copy assignment operation guarantees that it won't throw an exception (see [No-throw guarantee](#), below), which is not always possible.

If we cannot guarantee that the assignment operation won't 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's invoking the copy constructor) and invoking `A::g` and `B::h` on the copy. So, if an exception is thrown by any of those methods, we don't 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 `temp` will contain the pointer to the original data before modification (which will be freed automatically at the end of `C::f` by `temp`'s destructor).

***The pointer swap operation never throws an exception because it's 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 modifications 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.

You can find this example on the file `lectures/c++/09-exceptions/07-safety/05-except_strong.cc` in the repository, where `a.g()` runs normally but `b.h()` throws. But now we use the `Plmpl` idiom and `C::f` offers a strong guarantee. If we run it:

```
$ ./05-except_strong
```

```
Before c.f():
```

```
a.x = 0, b.y = 0
```

```
Now executing c.f()...
```

```
Exception caught.
```

```
After c.f():
```

```
a.x = 0, b.y = 0
```

Now, after we tried to run `c.f()` and it threw an exception, none of its effects were permanent. So, the state of the program remained unchanged, as if `c.f()` had never been called, as we would expect from a function that offers a strong guarantee.

**Note that the `plmpl` idiom is not the only way to accomplish this, it's only one of the possible ways to do it.**

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

**In fact, `plmpl` may not be the best solution if your data includes a collection (vector, map, etc.) because making a temporary copy of everything would mean copying the whole collection, which could be inefficient if the collection contains a large number of elements and only a few of them need to be modified. If that's the case, it would be better to just make copies of the objects you need to modify instead of using `plmpl` and copying everything.**

However, note that this only works because we previously said (in the comments) 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 don't 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.

**Generally, a function or method can only offer a strong guarantee if all the functions or methods that it calls offer a strong or a no-throw guarantee.**

**When a function or method offers a strong guarantee, you should always document it.**

Knowing that a function or method is exception-safe allows other functions that use it to also be exception-safe. This documentation can be written as a comment in the function's declaration or in another official documentation. For example, take a look at the documentation for the method [`vector::emplace\_back`](#). There is a section titled "Exception safety", which explains in which situations the method offers a strong or a basic guarantee.

### 3. No-throw Guarantee

Every function in C++ is either *non-throwing* or *potentially throwing*.

**Non-throwing functions** guarantee that they will 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* compiler-provided versions of the: constructor, copy constructor, move constructor, copy assignment operator, move assignment operator, and destructor are non-throwing, although there are some exceptions to this rule (which you can read about [here](#) if interested).

Any other function will be potentially throwing unless you declare it with [`noexcept`](#):

```
void f() noexcept; // the function f() does not throw
```

You can also pass an expression to `noexcept`. If it evaluates to true, then the function is declared as non-throwing:

```
void f() noexcept(true); // the function f() does not throw; same as just noexcept
void f() noexcept(false); // the function f() is potentially throwing; same as if you did not use noexcept at all
```

When you're writing a function that you 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. For example:

```
class MyClass {
    int x;
public:
    int getX() const noexcept { // does not change anything, so never throws
        return x;
    }
    void setX(int v) noexcept { // only copies an int value, so never throws
        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's the case, always declare them with noexcept. Doing so allows collection classes such as `std::vector` to be more efficient when storing objects of that class, as we will see in the next section.

## Exception Safety and the STL vectors

The STL vectors encapsulate a heap-allocated array and follow RAI: when a stack-allocated vector goes out of scope, the internal heap-allocated array is freed. For example:

```
void f() {
    vector<MyClass> v;
    ...
} // v goes out of scope; array is freed, MyClass destructor runs on all objs in the vector
```

But:

```
void g() {
    vector<MyClass*> v;
    ...
} // v goes out of scope; pointers don't have destructors; only the array is freed
```

Pointers don't 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 whether deleting those pointers may be appropriate. The pointers might not own the objects they're pointing at; the objects might not even be on the heap. So if these objects need to be freed, you have to do it manually:

```
for (auto &x : v) delete x;
```

Or alternatively, you can use smart pointers:

```
void h() {
    vector<unique_ptr<MyClass>> v;
    ...
} // array is freed; unique_ptr destructors run, so the objects ARE deleted
```

`unique_ptr`s 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.

Therefore, 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`), that's in summary what it needs to do:

- allocate a new, larger array
- copy the objects over (copy constructor)
  - if a copy constructor throws (strong guarantee):
    - destroy the new array
    - old array still intact
- 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.

Wouldn't moving the objects from the old array to the new array be more efficient?

- allocate a new, larger array
- move the objects over (move constructor)
- delete the old array and replace it with the new, larger array

The problem: if the move constructor throws, then `vector::emplace_back` can't offer a strong guarantee, because the original array would no longer be intact. But `emplace_back` promises a strong guarantee.

Therefore, if the move 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 in the previous subsection, your move operations should provide the no-throw guarantee, if possible, and you should indicate that they do:

```
class MyClass {  
public:  
    MyClass (MyClass &&other) noexcept { . . . }  
    MyClass &operator=(MyClass &&other) noexcept { . . . }  
    . . .  
};
```

If you do this, then whenever `vector::emplace_back` needs to resize its dynamic array for more capacity, it can use your 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.

In the next part, we will discuss the interaction between casting and object-oriented programming in C++, and C++'s many types of casts.

# Advanced Casts

Casts allow you to convert a piece of data from one data type to another. In C, whether this conversion is sensible or not was entirely up to the programmer. For instance, both of the following casts are valid in C:

```
int i = 42;
double d = (double) i;
double *dp = (double *) &i;
```

However, `d` has the value we expect (42.0), while `*dp` has a largely unpredictable value. To dereference `dp` is to reinterpret the bits in and around `i` (as a `double` is larger than an `int`) as a `double`, but those bits were never intended to be interpreted as a `double`. Casting pointers is rarely safe in C, and casting scalars is usually automatic, so the typical advice is simply to avoid casting in C altogether. In C++, because object oriented programming creates relationships between types that do not exist in C—e.g., an `A*` may be a perfectly valid `B*` if `A` and `B` are related types—more types of casts are allowed, and the different types of casts have different behaviors and safety properties.

To consider the various types of casts in C++, we will use these example classes:

```
class Book {
public:
    Book(std::string stitle, std::string sauthor) : title(stitle), author(sauthor), price(0) {}

    void setPrice(int to) { priceCents = to; }
    void setPrice(double to) { priceCents = round(to*100); }

    [...]

private:
    std::string title, author;
    int priceCents;
};
```

```

class FullText : public Book {
public:
    FullText(std::string stitle, std::string sauthor, std::string stext)
        : Book(stitle, sauthor), text(stext) {}

    [...]

private:
    std::string text;
};

class FrenchTranslation {
public:
    WordTranslation(std::string sen, std::string sfr) : en(sen), fr(sfr) {}

    [...]

private:
    std::string en, fr;
};

```

The Book class stores information on a book, namely the title, author, and price in cents. The price can be set to an integer number of cents, but if a double is given as an argument, it's assumed to be dollars. The FullText class stores the full text of a book, so it extends the Book class, and adds a text field. The FrenchTranslation class is unrelated to Book and FullText, and stores the translation of an English word into French.

While C-style casts work in C++, each specialized C++ cast has the following syntax: `the_cast<type>(value)`. `the_cast` is the particular kind of cast below, `type` is the type being cast to, and `value` is the value (which may be an expression) being cast.

## **static\_cast**

Some casts are perfectly sensible. For example, casting an `int` to a `double`, or a `(FullText *)` to a `(Book *)`. A `double` can store all the numbers an `int` can, and every `(FullText *)` is a `(Book *)`. Usually, an explicit cast isn't even needed in these circumstances. However, in this example, if we want to set the price on a `Book`, and our price is in dollars instead of cents, but is stored in an `int`, we need to cast to specify which `setPrice` we actually intend to call (or, of course, we could simply multiply it by 100). `static_cast` performs this sort of safe cast:

```
Book *b = [...];
int priceDollars = 42;
b->setPrice(static_cast<double>(priceDollars));
```

However, `static_cast` is more general than this. As well as safe casts, `static_casts` performs any cast with a well-defined behavior. As well as safe casts, the other kinds of casts with well-defined behavior are **downcasts**: That is, casts from a supertype to a subtype. For example, if we know with certainty that a given `(Book *)` actually points to a `FullText`, we can `static_cast` to get a usable `(FullText *)`:

```
void badlyPriceABook(Book *b, bool isFullText) {
    int p = b->getTitle().length * 120;
    if (isFullText) {
        FullText *f = static_cast<FullText *>(b); // Or auto f = ...
        p += f->getText().length / 100;
    }
    b->setPrice(p);
}
```

This `static_cast` is only correct if `isFullText` is correct—that is, if `b` actually points to a `(FullText *)`, which it can since `FullText` is a subtype of `Book`. In that case, it behaves as you would expect: `FullText *f` is a valid pointer to the `FullText`. If `isFullText` is set incorrectly, this code will crash or have unpredictable behavior. `static_cast` is a promise by the programmer to the compiler that we know the type we're casting to is correct, and a promise by the compiler to the programmer that, so long as that's true, the resulting pointer will behave correctly.

## reinterpret\_cast

[reinterpret\\_cast](#) causes a value to be *reinterpreted* as another type. For instance, all pointers point to some address in memory, and that address has a numeric value. We could use `reinterpret_cast` to reinterpret the pointer as its numeric value:

```
std::cout << "My book is stored at this address: " << reinterpret_cast<long>(b) << std::endl;
```

`reinterpret_cast` can also be used to reinterpret one pointer type as another, e.g.:

```
Book *b = new Book("I Know Why the Caged Bird Sings", "Maya Angelou");
FrenchTranslation *f = reinterpret_cast<FrenchTranslation *>(b);
```

It is of course completely invalid to reinterpret a `Book *` as a `(FrenchTranslation *)`. `reinterpret_cast` is usually unsafe. Worse than that, it often won't crash, but will have surprising and weird behavior. In this example, it is likely (but not guaranteed!) that `f->en` will be "I Know Why the Caged Bird Sings", and `f->fr` will be "Maya Angelou", but attempting to use methods will have odd results.

It is exceedingly rare for `reinterpret_cast` to be needed, but there are some very specialized uses for it, such as just-in-time compilers, and certain dynamic database implementations.

## [const\\_cast](#)

C++ uses `const` to ensure that values are not incorrectly changed. For example, a function `int magic(const int *x)` may read `*x`, but promises not to modify it. Unfortunately, as `const` declarations can get complicated, it's not uncommon for them to be absent or incorrect. If you're certain that a non-`const` pointer won't actually modify the value, you can use [const\\_cast](#) to cast that `const` away. For example:

```
int getTitleLength(Book *); // Does not actually modify the book
```

```
[...]
```

```
const Book *b = [...];
getTitleLength(const_cast<Book *>(b));
```

In most cases, `const` is only guaranteed by the types, so casting `const` away can have surprising results. For instance, if `getTitleLength` in this example did not do what we thought it would do, but modified the `Book`, then those changes would be visible through our `const` pointer.

In principle, `const_casts` should never be necessary, as `const` should always be declared correctly. In practice, vast bodies of existing code do not use `const` correctly, and `const_cast` is needed to interact with such code.

## [dynamic\\_cast](#)

The only unifying property of our above three casts is that they are not guaranteed to be safe; it is up to the programmer to make sure that they are. `dynamic_cast` is a **checked cast**: It allows downcasts, like `static_cast`, but actually checks that the object is of the given type, and so is safe. If the object is not of the specified type, then the `dynamic_cast` resolves to `nullptr`, which makes it possible to check at runtime whether the type was correct. Using `dynamic_cast`, we can rewrite `badlyPriceABook` from above to not require the `isFullText` argument:

```
void badlyPriceABook(Book *b) {
    int p = b->getTitle().length * 120;
    FullText *f = dynamic_cast<FullText *>(b); // Or auto f = ...
    if (f != nullptr) {
        p += f->getText().length / 100;
    }
    b->setPrice(p);
}
```

In most cases, `dynamic_cast` is the safest option, but it has a fairly unpredictable cost. Exactly how the runtime check is implemented varies with different compilers, and different types. In practice, if surrounding information can guarantee a type to be known, `static_cast` will always be a faster option. Furthermore, the information required to perform these checks takes space, and many C++ projects exclude this information to save space, using g++'s `-fno-rtti` option (RTTI stands for Runtime Type Information) and its equivalent in other compilers. If `-fno-rtti` is

specified, then `dynamic_cast` will not work correctly. In spite of these caveats, `dynamic_cast` is the *only* specialized cast which is always safe.

## Smart pointers

C++'s specialized casts work on plain `*` pointers, but if you've read the previous two modules, you would probably prefer to use smart pointers. With the exception of `reinterpret_cast`, each of these has an equivalent which operates on `shared_ptr`s instead of `*` pointers: `static_pointer_cast`, `const_pointer_cast`, and `dynamic_pointer_cast`. Here is `badlyPriceABook`, using a `shared_ptr` and `dynamic_pointer_cast`:

```
void badlyPriceABook(shared_ptr<Book> b) {
    int p = b->getTitle().length * 120;
    shared_ptr<FullText> f = dynamic_pointer_cast<FullText>(b); // Or auto f = ...
    if (f != nullptr) {
        p += f->getText().length / 100;
    }
    b->setPrice(p);
}
```

These casts have all the combined implications of `shared_ptr` and the plain specialized cast. For example, with `dynamic_pointer_cast`, there is a runtime cost to dynamic type checking, and to reference counting.

There is no equivalent for `unique_ptr`s, and such an equivalent would rarely be useful, since `unique_ptr`s's uniqueness make them less likely to *lose* type information, and thus less likely to require recovering that type information with a cast.

# Fixing the polymorphic assignment problem with dynamic casting

In a [previous topic](#), we introduced three different options to deal with the copy/move assignment operations together with inheritance:

1. *Public non-virtual operations* do not restrict what the programmer can do but allow partial assignment;
2. *Public virtual operations* do not restrict what the programmer can do but allow mixed assignment;
3. *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 assignments using base class pointers.

However, our example for option 2 (public virtual operations) created another problem: the program would crash if the programmer attempted to do a mixed assignment, like this:

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

However, there is a better way to implement the copy/move assignment operators that allows the program to better deal with the error above. To do this, we need to use [dynamic casting](#) and [exception handlers](#). If you did not study these two topics yet, please take a moment to read them first before you continue reading the solution explained below.

## Using dynamic casting to safely implement the assignment operators

Remember that when implementing the copy/move operators polymorphically, we need to pass the right side object as a reference to an object with the type of the superclass. In the example introduced in the previous topic, these

were the signatures of the methods within class Text, which is a subclass of Book:

```
Text &operator=(const Book &rhs) override;  
Text &operator=(Book &&rhs) override;
```

We can use a dynamic cast to safely attempt to treat the object passed by reference as the parameter to the copy/move assignment operators, which is of type Book&, as a Text&. If the object passed as parameter is not an instance of a Text, the dynamic cast will throw a std::bad\_cast exception. So, this is how the implementation of the two methods would look like:

```
Text &Text::operator=(const Book &rhs) {  
    if (this == &rhs) return *this;  
    Book::operator=(rhs);  
    // Attempt to treat rhs as a Text object using a dynamic_cast.  
    // If rhs is not a Text, an exception will be thrown.  
    const Text &rhst = dynamic_cast<const Text&>(rhs);  
    topic = rhst.topic;  
    return *this;  
}  
  
Text &Text::operator=(Book &&rhs) {  
    if (this == &rhs) return *this;  
    Book::operator=(std::move(rhs));  
    // Attempt to treat rhs as a Text object using a dynamic_cast.  
    // If rhs is not a Text, an exception will be thrown.  
    Text &rhst = dynamic_cast<Text&>(rhs);  
    topic = std::move(rhst.topic);  
    return *this;  
}
```

## Handling the std::bad\_cast exception

Note that this fix will not completely eliminate the mixed assignment issue. If the programmer tries to do a mixed assignment (for example, by trying to use a Book or Comic object to assign it into a Text variable), the compiler will

not detect the problem. However, the exception `std::bad_cast` will be thrown when the dynamic cast is attempted. But the advantage of this solution is that the exception can now be handled, so the program can recover as appropriate instead of crashing. For example:

```
// Trying to assign a Text from a Book will raise a std::bad_cast exception
try {
    Book b1("Programming for Beginners", "Niklaus Wirth", 200);
    Text t2("Programming for Big Kids", "Bjarne Stroustrup", 300, "C++");
    Book *pb1 = &b1;
    Book *pb2 = &t2;
    *pb2 = *pb1; // std::bad_cast will be thrown
    printTextBook(t2, "Text 2");
} catch (std::bad_cast r) {
    cerr << "Error trying to assign a Text object: " << r.what() << endl;
}
```

You can find this complete example in the directory `lectures/c++/08-purevirt/05-copy_move_virtual_dynamic_cast` in the repository. Please take a moment to review it. Be sure you understand how the dynamic cast was used in the implementation of the copy/move assignment operators. Check and run the `main_exceptions` example. Be sure you understand how the exception handler was implemented. Notice that in comparison with the example in the folder `03-copy_move_virtual`, the program now does not crash when the first error occurs. Instead, a message is printed and the execution of the program continues.

In summary, this fix does not completely eliminate the problem of mixed assignment, but at least it allows the program to gracefully handle the exception and avoid crashing. Therefore, if you decide to implement public virtual copy/move assignment operations in your class hierarchy, be sure to use this form of implementation so your program can handle any errors appropriately.

# Template Functions

You have previously learned to create [template classes](#). In C++, we can also 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 template functions. When we use the function, the template is then instantiated and each instance of the generic type (T in the example above) is replaced with the actual type. For example:

```
int f() {
    int x = 1, y = 2;
    int z = min(x, y);      // T = int
    char w = min('a', 'c'); // T = char
    auto f = min(1.0, 3.0); // T = double
}
```

Note: 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 (e.g., if the function has no arguments), you can always tell it explicitly, for example: `z = min<int>(x, y);`.

For what types T can `min` be used? For what types T does the body of `min` compile? If you look at the function's body, it uses `operator<` between x and y (remember that `x < y` converts to a call to `operator<`). Therefore, `min` will compile for any type T for which `operator<` is defined. Thus, the following code will not compile:

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

```
}

int main() {
    MyClass a{1};
    MyClass b{2};
    MyClass c = min(a, b);
}
```

Let's try to compile it:

```
$ g++ -std=c++14 testTemplate.cc
testTemplate.cc: In instantiation of ‘T min(T, T) [with T = MyClass]’:
testTemplate.cc:10:22:   required from here
testTemplate.cc:5:11: error: no match for ‘operator<’ (operand types are ‘MyClass’ and ‘MyClass’)
  return x < y ? x : y;
           ~~^~~
```

As you can see it, g++ tried to use `MyClass` as the type `T` for function `min`. However, `MyClass` does not define `operator<`. Therefore, it was not possible to instantiate a concrete version of the template function `min` for the type `MyClass`.

**Practice: What types can be used as the argument to the following function `foo`?**

```
template <typename T>
void foo(T &x) {
    T y{x};
    y = y + 5;
    cout << y << endl;
}
```

- `std::string`
- any object (i.e., any instance of a class)
- any type that implements a copy constructor, `operator+(int)`, and `operator<<(std::ostream, T)`
- any type that implements a copy constructor and `operator+(int)`

Yes, that's great!

## Template Iterator Functions

Now that you know how to create template functions, you can write cool functions that work with [Iterators](#).

First, let's remember 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 usually support `operator*`, `operator++`, `operator!=`, and optionally `operator==`.

Knowing this, let's 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;  
    }  
}
```

As you can see, our 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 function. 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) { cout << n << endl; }
. .
int a [] = {1, 2, 3, 4, 5};
. .
for_each(a, a+5, f); // prints the array
```

# Function Objects

If we write the method `operator()` for a class, then we can use objects of that class as functions. For example:

```
class Plus1 {  
public:  
    int operator() (int n) { return n + 1; }  
};  
. . .  
Plus1 p;  
p(4); // produces 5
```

In the code above, it seems like we're calling 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. Remember:

```
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`. For example:

```
class Plus1 {  
public:  
    int operator() (int &n) { ++n; }  
};  
. . .  
int a [] = {1, 2, 3, 4, 5};  
Plus1 p;
```

```
...
for_each(a, a+5, p); // adds one to each element of the array
```

## Lambdas

Imagine you need to answer the question: how many `ints` in a `vector` are even? We could use a generic function like the `for_each` above to iterate over the array and count how many elements are even. For example:

```
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);
```

This works well. However, it seems like a waste to explicitly create the function `even` if it will only be used once. Instead, we can use a *lambda function* (or anonymous function), which is a function not bound to an identifier (such as `even`) and we can pass it as a parameter. So, let's convert `even` into a lambda function (using the same definition of `count_if` above):

```
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 ( ), just like a regular function; and the *body* inside the { }, just a regular function. Please refer to the [documentation of lambdas](#) for more details.

# STL Algorithms

In addition to template classes, the [Standard Template Library \(STL\)](#) also provides a suite of useful template functions, many of which work over iterators. You can use them with `#include <algorithm>` (and they're in the `std` namespace).

We will only mention five of them: `for_each`, `find`, `count`, `copy`, and `transform`, but there are many others. We suggest that you spend some time familiarizing yourself with the [list of available functions](#). When a function is available for what you need to do, it's usually better to use it rather than having to implement the algorithm yourself.

## for\_each

Like the examples that we provided in the previous topics, `for_each` applies a function `fn` to each of the elements in the range `[first, last]`:

```
template<class InputIterator, class Function>
Function for_each(InputIterator first, InputIterator last, Function fn)
{
    while (first!=last) {
        fn (*first);
        ++first;
    }
    return fn;      // or, since C++11: return move(fn);
}
```

Usage example:

```
void addOne(int &n) { ++n; }
. . .
```

```
vector<int> v { ... };
for_each(v.begin(), v.end(), addOne); // calls addOne for each element in the vector
```

## 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.

```
template<class InputIterator, class T>
InputIterator find (InputIterator first, InputIterator last, const T& val)
{
    while (first!=last) {
        if (*first==val) return first;
        ++first;
    }
    return last;
}
```

Usage example:

```
int myints[] = { 10, 20, 30, 40 };
int * p = find (myints, myints+4, 30); // p points to the element 30 in the array
```

## count

[count](#) returns the number of elements in the range [first, last) that compare equal to val.

```
template <class InputIterator, class T>
typename iterator_traits<InputIterator>::difference_type
count (InputIterator first, InputIterator last, const T& val)
{
    typename iterator_traits<InputIterator>::difference_type ret = 0;
    while (first!=last) {
```

```
    if (*first == val) ++ret;
    ++first;
}
return ret;
}
```

Note: The return type (`iterator_traits<InputIterator>::difference_type`) is a signed integral type.

Usage example:

```
int myints[] = {10,20,30,30,20,10,10,20};      // 8 elements
int mycount = count (myints, myints+8, 10); // mycount = 3
```

## 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).

```
template<class InputIterator, class OutputIterator>
OutputIterator copy (InputIterator first, InputIterator last, OutputIterator result)
{
    while (first!=last) {
        *result = *first;
        ++result; ++first;
    }
    return result;
}
```

Note: `copy` does not allocate new memory, so the output container must already have enough space available.

Usage example:

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

But consider this:

```
vector<int> v {1, 2, 3, 4, 5};  
vector<int> w;  
copy(v.begin(), v.end(), w.begin()); // WRONG! No space available in w.
```

This will fail because `copy` doesn't allocate space in `w`. It can't, because it doesn't even know what kind of container `w` iterates over! But what if we had an iterator whose assignment operator inserts a new item? For example:

```
vector<int> v {1, 2, 3, 4, 5};  
vector<int> w;  
copy(v.begin(), v.end(), back_inserter(w)); // works
```

A [back\\_inserter](#) is an iterator that calls the method `push_back` (from `vector` or similar containers). Now, `v` is copied to the end of `w`, by adding new items. Back inserters are available for any container with a `push_back` method.

## transform

[transform](#) applies an operation `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`. It's similar to `copy`, except that it applies the operation `op` to each element being copied.

```
template <class InputIterator, class OutputIterator, class UnaryOperator>  
OutputIterator transform (InputIterator first, InputIterator last,  
                         OutputIterator result, UnaryOperator op)  
{  
    while (first != last) {  
        *result = op(*first);  
        ++result; ++first;  
    }  
    return result;  
}
```

Note: like copy, transform does not allocate new memory, so the output container must already have enough space available, or you can use a back inserter to add new elements to the output container.

Usage example:

```
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); // w = {3, 4, 6, 8, 12}
```

## Iterators

Remember 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, e.g., streams.

```
#include <vector>
#include <iterator>
#include <algorithm>
vector<int> v {1, 2, 3, 4, 5};
ostream_iterator<int> out {cout, ", "};
copy(v.begin(), v.end(), out); // Prints 1, 2, 3, 4, 5,
```

And that's it! Take time to learn to work with iterators and algorithms, and to be comfortable with them. They can dramatically shorten your programs and reduce opportunities for program bugs.

# Factory Method

The **Factory Method** design pattern provides an interface for object creation, but lets the subclasses decide which object to create. It is also known as the **Virtual Constructor**.

For more reading on the topic, you can read chapter 4 of the "[Head First Design Patterns](#)" book by Freeman and Freeman.

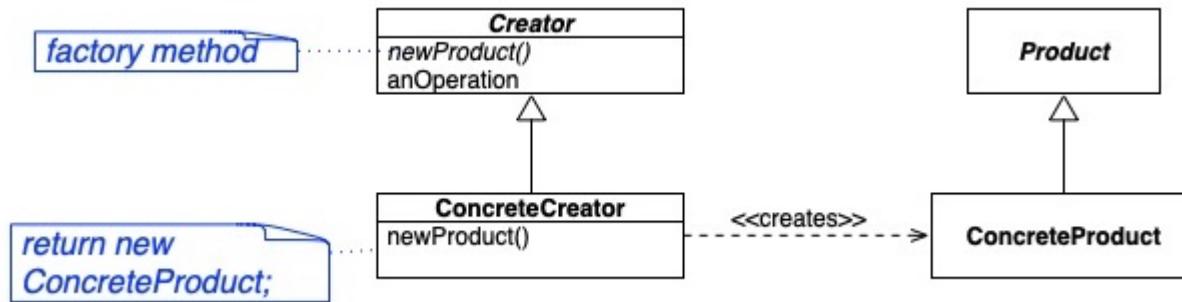
## Examples of use

Common examples of use are:

- Java's XML parser package has some factories: `javax.xml.parsers.DocumentBuilderFactory` or `javax.xml.parsers.SAXParserFactory`.
- [Qt](#) is a free and open-source widget toolkit that lets users build graphical user interfaces across multiple platforms. It has a factory method, [`QMainWindow::createPopupMenu\(\)`](#).
- The Microsoft .NET Framework has a data access component, ADO.NET, that uses the factory method [`IDbCommand.CreateCommand`](#) to connect parallel class hierarchies.
- [HTML5](#)'s Document Object Model (DOM) application programming interface (API) creates specific elements of the `HTMLElement` interface through the factory method [`createElement`](#).

## General UML class model

The general form of the Factory Method design pattern's UML class model diagram looks like:

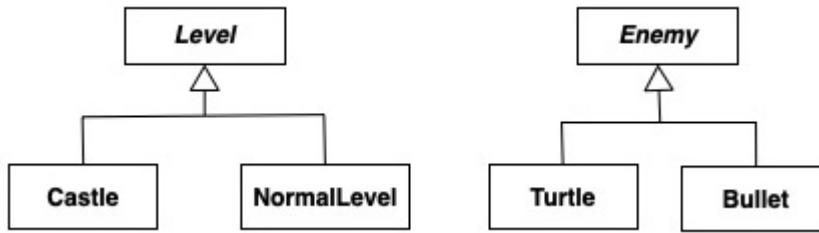


## Notes:

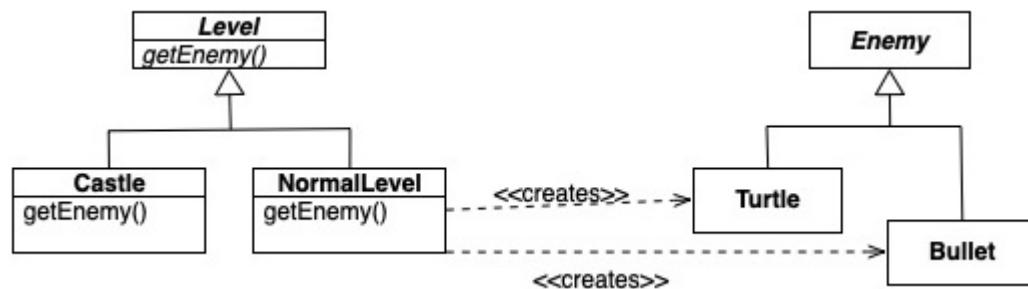
- Encapsulates the object creation as the part of the design that changes, which is why it's a virtual method in the Creator superclass. It may not necessarily be a pure virtual method if it chooses to provide a default implementation to use.
- Creator superclass encapsulates the operations other than creation that can be applied to a product i.e. anOperation.
- A valid variation is to add a parameter to the newProduct method.

## Code example

Our motivating example is going to use a video game with various types of enemies (turtles or bullets), where the type of enemy created varies based upon the current game level. For example, at an easy/normal level, you might want the random creation of enemies to be 70% turtles, and 30% bullets; but, at the hard/castle level, the proportion should instead be 70% bullets and 30% turtles. Our basic inheritance hierarchy looks like:



We can then incorporate the Factory Method into our `Level` class, by adding a pure virtual `getEnemy` method that returns an `Enemy` pointer to either a `Turtle` or `Bullet`, created in the appropriate random ratio encapsulated by the concrete `Level` subclass. Our UML class model would then look like:



Since we don't have an actual, implemented video game, we'll just provide a sketch of the class structures in C++. The code for the `NormalLevel` subclass will be similar to that of the `Castle`, so we'll only show the necessary fragment of the `Castle` implementation.

| File: level.h   | File: castle.h  |
|---|---|
| <pre>#ifndef _LEVEL_H #define _LEVEL_H  class Enemy;  class Level {     ... public:     virtual Enemy * getEnemy() = 0;</pre> | <pre>#ifndef _CASTLE_H #define _CASTLE_H #include "level.h"  class Castle : public Level {     ... public:     virtual Enemy * getEnemy() override;     ...</pre> |

```

}; ...
#endif

```

**File: level.cc**

```

#include "level.h"

// implements whatever isn't pure virtual

```

```

};

#endif

```

**File: castle.cc**

```

#include "castle.h"
#include "turtle.h"
#include "bullet.h"

Enemy * Castle::getEnemy() {
    Enemy * eptr;
    // creates mostly bullets
    return eptr;
}

```

Our main program could create a concrete subclass of the "current" Level, and use that to create the enemies:

```

Level *level = new NormalLevel;
Enemy * enemy = level->getEnemy();

```

without needing to know anything about how the choice of which enemy to create is made. As the game progresses, we would change out what the `level` pointer is pointing to, and the rules for enemy creation would automatically change!

# Abstract Factory

The **Abstract Factory** design pattern provides an interface for creating families of related or dependent objects. It is also known as **Kit**.

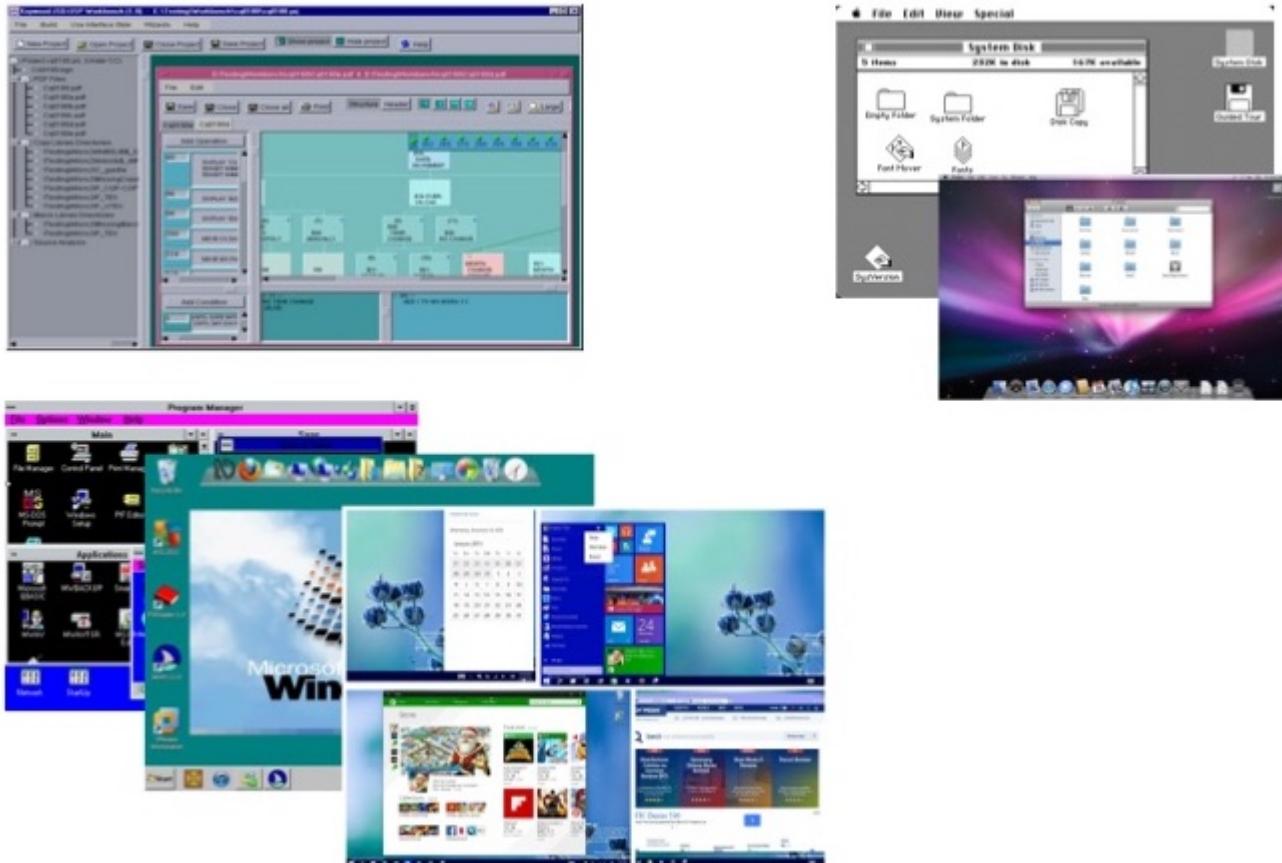
For more reading on the topic, you can read chapter 4 of the "[Head First Design Patterns](#)" book by Freeman and Freeman.

## Examples of use

Common examples of use are:

- Ensuring that the look-and-feel of a user interface toolkit is consistent across operating systems and platforms.  
For example, the look of windows in the Mac OS versus the Windows OS requires that the different title bars,

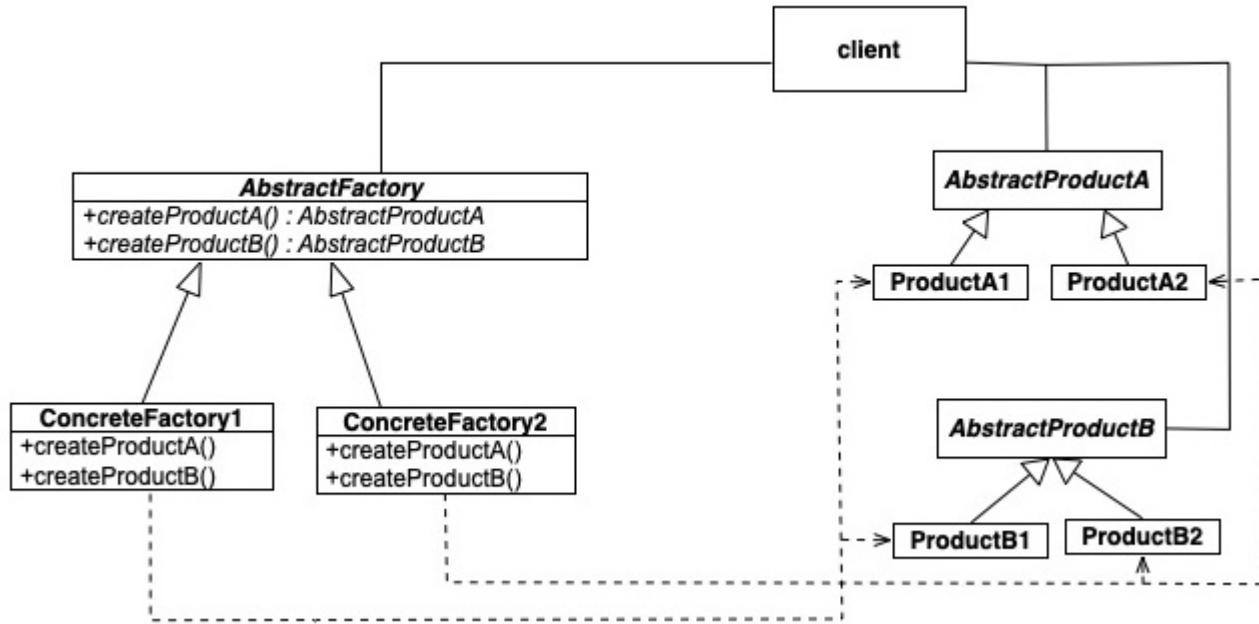
scroll bars, buttons, etc. share a consistent appearance.



- Internationalization of a piece of software. For example, software with English text would need different languages in order to be successfully sold in international markets; however, it wouldn't be useful if some of the text was in Mandarin, and some was in Swahili. An abstract factory could be used to ensure that all elements were generated from the factory for a particular language.

## General UML class model

The general form of the Abstract Factory design pattern's UML class model diagram looks like:



## Notes:

- Normally only a single instance of the concrete factory class is created at run-time.
- Makes it easy to change product families, since you only need to change which concrete factory was created.
- Hard to add new products since all of the concrete factory classes as well as the abstract superclass must be changed.
- The abstract factory methods can be implemented using the Factory Method design pattern!
- Commonly uses object composition to build the final object by composing it of the necessary pieces.

While we won't explore the pattern in any further detail, it complements the Factory Method design pattern nicely, which is why it's mentioned.

# Strategy

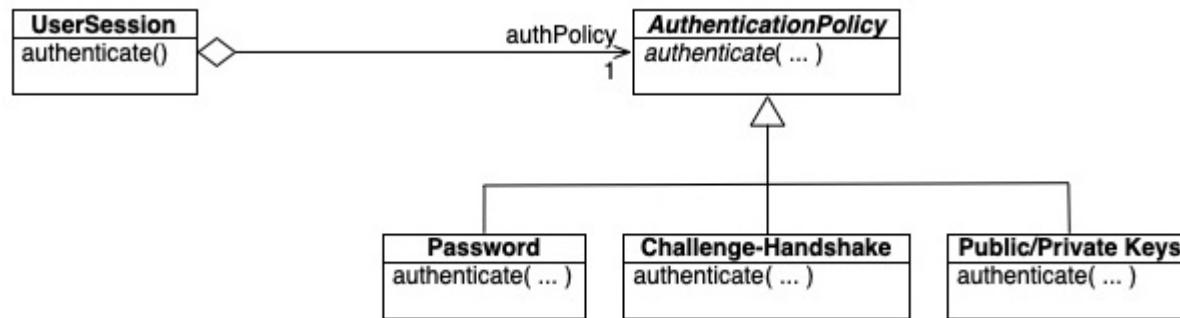
The **Strategy** design pattern is intended to define a family of algorithms, encapsulating each one, and making them interchangeable. Allows the algorithm to vary independently of the client. Also known as **Policy**.

For more reading on the topic, you can read chapter 1 of the "[Head First Design Patterns](#)" book by Freeman and Freeman.

## Examples of use

Common examples of use are:

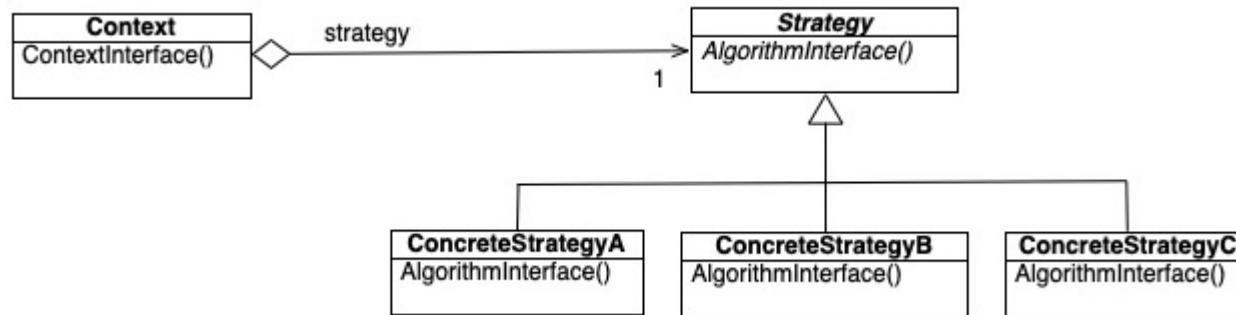
- A class that authenticates users using a variety of authentication policies.



- Choice of sorting algorithm may depend upon data set features.
- Computer player's choice of how to behave may depend upon game context, and change as play progresses.
- How to store or format information may change over time.

## General UML class model

The general form of the Strategy design pattern's UML class model diagram looks like:



#### Notes:

- The client uses composition (not the relationship "composition", but the notion that it has a data field of type `Strategy`, implemented in C++ as a pointer i.e. is built from) to access the current algorithm in use, but isn't dependent upon the concrete class since is using the abstract base class.
- The algorithm in use can be dynamically changed.
- The strategy hierarchy lets inheritance factor out common behaviour from the families of related algorithms.
- Strategy helps eliminate conditional statements to select behaviour.

While we won't explore the pattern in any further detail, it is frequently useful for the course final project, which is why it's mentioned.

# Template Method

The **Template Method** design pattern defines the steps of an algorithm in an operation, but lets the subclasses redefine certain steps though not the overall algorithm's structure.

For more reading on the topic, you can read chapter 8 of the "[Head First Design Patterns](#)" book by Freeman and Freeman.

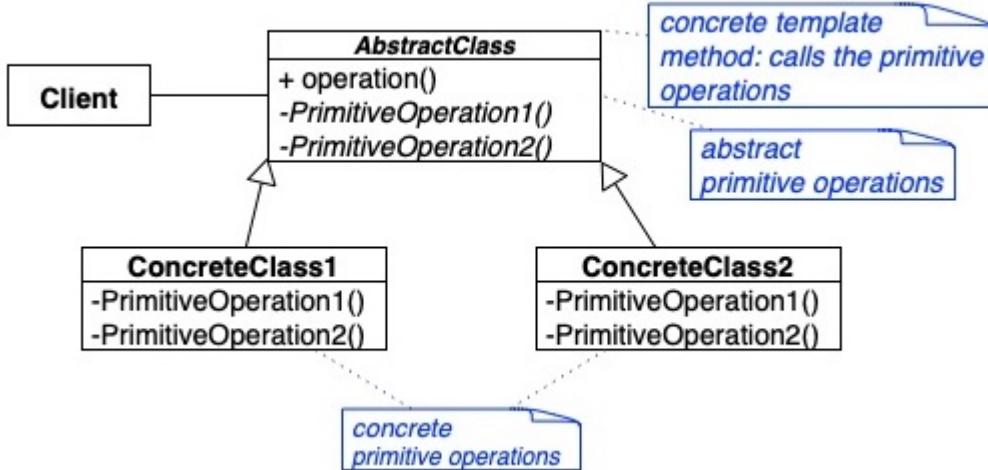
## Examples of use

Common examples of use are in frameworks:

- In the Java Swing library, the `JFrame` class has a `paint()` method that you can override as a "hook" to how to draw the basic frame for a graphical user interface.
- The Java `Applet` class provides several hooks, `init()`, `start()`, `stop()`, `destroy()`, and `paint()`, all of which can be overridden.
- Some other examples can be found at: <https://www.codeproject.com/Articles/307452/common-use-of-Template-Design-pattern-Design-pat>

## General UML class model

The general form of the Template Method design pattern's UML class model diagram looks like:



## Notes:

- The abstract base class contains the refactored common behaviour for all subclasses to remove duplicate code.
- Subclasses can only override the provided *hooks*, the virtual methods.
- The virtual methods may provide a default implementation, or be pure virtual.
- The method that contains the algorithm is not virtual. The steps and their order may not be changed!

## Code example

Our motivating example is going to continue from our video game example. One of our Enemy subclasses was the Turtle class. Let's extend our model to have the video game contain both red and green turtles i.e. the colour of the turtle's shell is either drawn in red, or drawn in green. However, all turtles have a head, a shell, and feet.

| File: turtle.h                                 | File: turtle.cc                                       |
|--|---|
| <pre>#ifndef _TURTLE_H #define _TURTLE_H</pre> | <pre>#include "turtle.h"  void Turtle::draw() {</pre> |

```

class Turtle {
    void drawHead();
    void drawFeet();
    virtual void drawShell() = 0;
public:
    void draw();
};

#endif

```

```

        drawHead();
        drawShell();
        drawFeet();
    }

    void Turtle::drawHead() {
        // draws a head
    }

    void Turtle::drawFeet() {
        // draws feet
    }
}

```

Note that it is perfectly legal to have a private pure virtual method.

Our concrete red and green Turtle subclasses then become:

| File: redturtle.h   | File: greenturtle.h   |
|---|---|
| <pre> #ifndef _RED_TURTLE_H #define _RED_TURTLE_H #include "turtle.h"  class RedTurtle : public Turtle {     virtual void drawShell() override; }; </pre> | <pre> #ifndef _GREEN_TURTLE_H #define _GREEN_TURTLE_H #include "turtle.h"  class GreenTurtle : public Turtle {     virtual void drawShell() override; }; </pre> |
| File: redturtle.cc  | File: greenturtle.cc  |
| <pre> #include "redturtle.h"  void RedTurtle::drawShell() {     // draw red shell } </pre>  | <pre> #include "greenturtle.h"  void GreenTurtle::drawShell() {     // draw green shell } </pre>  |

The subclasses cannot change the way that a turtle is drawn, but they *can* change the way that the shell is drawn.

## Variations

### Non-Virtual Interface Idiom

Let's consider the roles performed by a public `virtual` method.

1. It describes an interface to the client using it. It specifies pre- and post-conditions, the provided behaviour, and the [class invariants](#) i.e. what properties must hold or must be true both before and after the method executes for all instances of the class.
2. It describes the subclass interface, and specifies the **hook** whereby the subclass can insert its own specialized behaviour.

Having these two roles wrapped up in the same function declaration makes it hard to separate these two roles. For example, we may later wish to change the customizable behaviour into two methods, with some non-customizable steps in between, but not change the public interface. It's also problematic as to how we can enforce the class invariants or pre- and post-conditions when the method can be overridden, which lets the person writing the code do whatever they want, including violate the the class invariants or pre- and post-conditions.

The **Non-Virtual Interface (NVI) Idiom** lets us accomplish these goals by specifying the following design decisions:

- All public methods should be non-virtual.
- All virtual methods (**except the destructor**) should be declared either `private`, or at least `protected`.

Let's consider a simple example to illustrate NVI.

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

We now have the ability to change play's behaviour however we wish, **without changing the interface**. We could add code before or after the call to doPlay that cannot be changed, such as checking for copyright before playing the item, or updating the play count afterwards. play could provide additional virtual "hooks" such as displaying the cover art.

It is much easier to provide this sort of control over our methods right from the very beginning, than try to take back control over them later!

The NVI idiom extends the Template Method design pattern by putting **every virtual method inside a non-virtual wrapper**. There is essentially no disadvantage to this, since a good compiler will optimize away the extra function call.

# Visitor

The **Visitor** design pattern allows the programmer to apply an operation to be performed upon the elements contained in an object structure. New operations can be added without changing the elements on which it operates.

For more reading on the topic, you can read chapter 14, the appendix, of the "[Head First Design Patterns](#)" book by Freeman and Freeman.

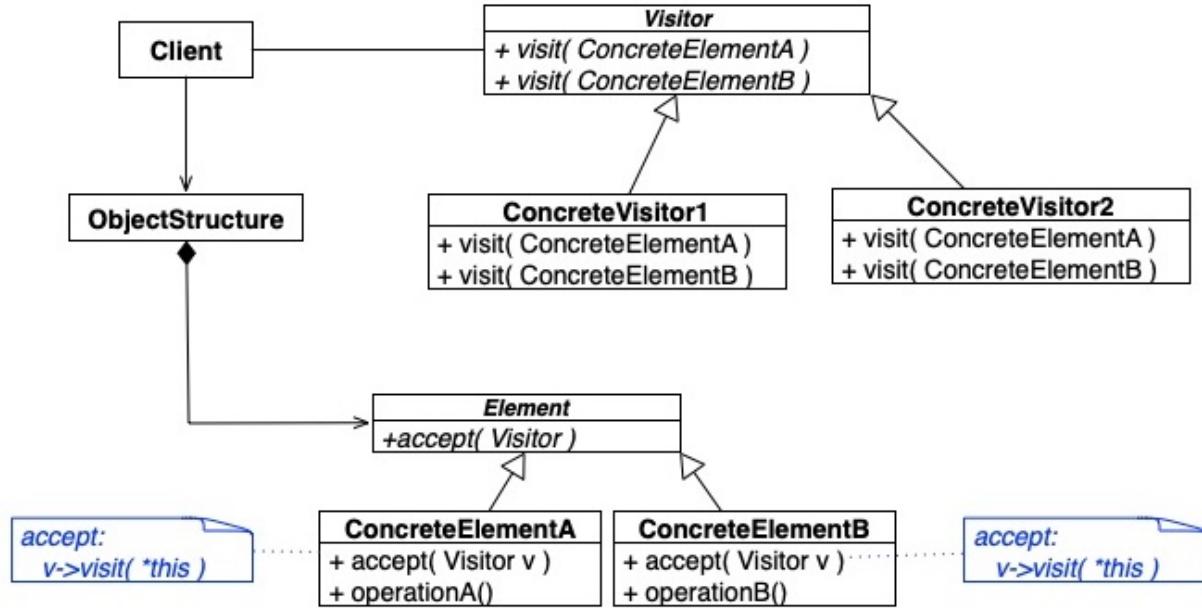
## Examples of use

Common examples of use are:

- The programming language Smalltalk-80 has a Visitor class called `ProgramNodeEnumerator`.
- The IRIS Inventor [[Str93](#)] is a 3-D graphics application toolkit that represents a scene as a tree of nodes. It uses a visitor for rendering the scene, and handling events, etc.
- Corey Coogan describes using the Visitor design pattern in a .NET database application at  
<https://coreycoogan.wordpress.com/2009/06/16/visitor-pattern-real-world-example/>

## General UML class model

The general form of the Visitor design pattern's UML class model diagram looks like:



The Visitor design pattern is one of the more difficult to understand patterns. Let's take a look at a motivating example, and use it to build up to the necessary structure of the Visitor design pattern to better explain how the pieces fit together.

## Motivating example

Let's go back to our video game, where our `Enemy` superclass had `Turtle` and `Bullet` subclasses. In order to defeat our enemies, we need to be able to strike at them with a weapon. What weapon we have available will depend upon what has happened so far in the game, and the various types of weapons available will have varying effects on the enemies i.e. some weapons may be more effective acting upon certain enemies than others.

Our basic structure might look something like the following:



In order to keep our code as flexible as possible, we'd *really* like it if we could just write a routine like:

```
virtual void strike(Enemy & e, Weapon & w);
```

Then, depending upon which actual concrete subclass gets passed in as an Enemy and as a Weapon, the appropriate actions take place. However, since we're writing an object-oriented program, it would make sense to embed the `strike` function as a method in one of our classes.

What happens if we put it in the `Enemy` class as:

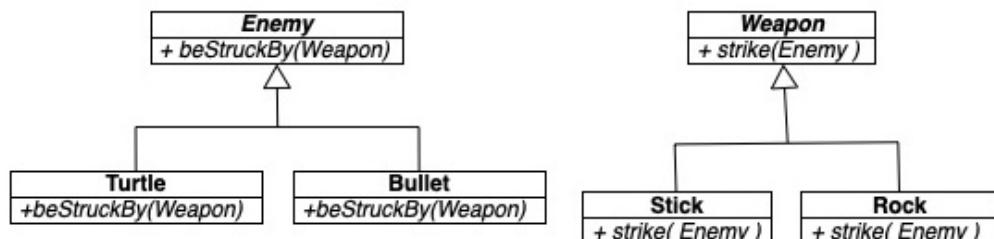
```
virtual void strike( Weapon & w );
```

? We'd be able to choose the appropriate method based upon the actual type of the `Enemy` object (this is called **dynamic dispatch**, and is implemented in C++ as **single dispatch** i.e. can only be performed upon one object by looking up its actual type at run-time), but not also based upon the `Weapon` object type.

If we instead put it in the `Weapon` class as:

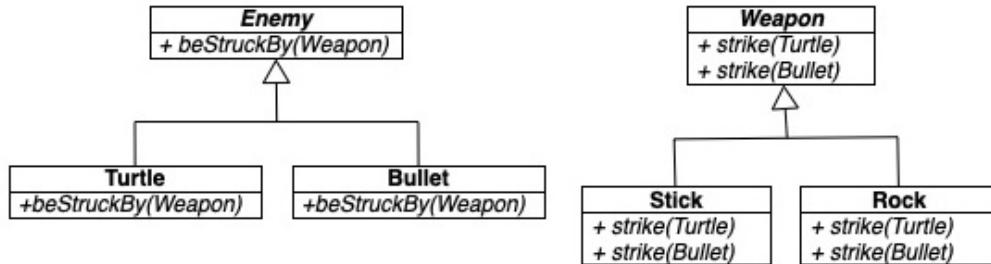
```
virtual void strike( Enemy & e );
```

we'd be able to choose the appropriate method based upon the actual type of the `Weapon` object, but not also based upon the `Enemy` object type.



What we need is a technique called **double dispatch**, which requires us to combine both **overloading** and **overriding** (single dispatch). We'll pick one class, `Enemy`, upon which we'll apply overriding. We'll then apply overloading to the `Weapon` class.

Our revised class model now looks like:



|  |   |
|--|---|
| <b>File: enemy.h</b>   | <b>File: turtle.h</b>   |
| <pre>class Enemy { public:     // override i.e. dynamic dispatch     virtual void beStruckBy( Weapon &amp; w ) = 0;     ... };</pre>   | <pre>class Turtle : public Enemy { public:     virtual void beStruckBy( Weapon &amp; w ) {         w.strike( *this );     } };</pre>  |
| <b>File: weapon.h</b>  | <b>File: stick.h</b>  |
| <pre>class Weapon { public:     // overload for other class hierarchy     virtual void strike( Turtle &amp; t ) = 0;     virtual void strike( Bullet &amp; b ) = 0;     ... };</pre> | <pre>class Stick : public Weapon { public:     virtual void strike( Turtle &amp; t ) {         // strike turtle with stick     }     virtual void strike( Bullet &amp; b ) {         // strike bullet with stick     } };</pre> |

## Notes:

- The **Bullet** class will look just like the **Turtle** class.
- The **Rock** class will look just like the **Stick** class.

Let's say that we have the following code fragment:

```
Enemy * e = new Bullet{ ... }; // create a bullet  
Weapon * w = new Rock{ ... }; // create a rock  
e->beStruckBy( *w );
```

What happens when we call: `e->beStruckBy( *w )`?

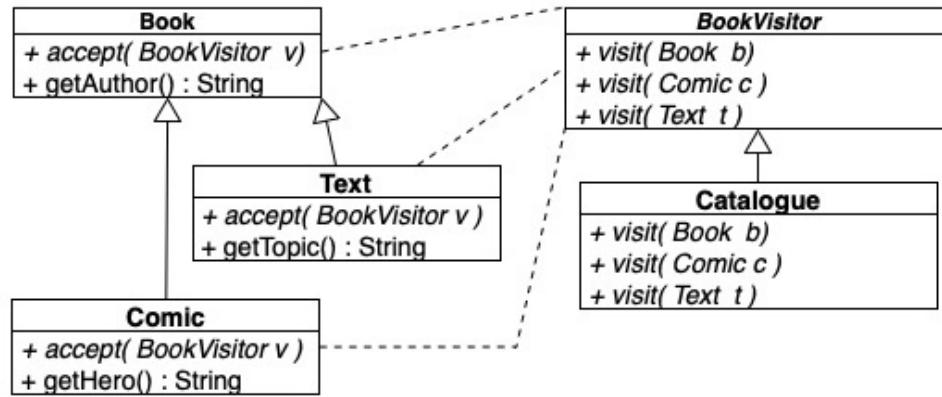
1. First the program has to determine the actual object type for `e`.
2. Since `e` is pointing to a `Bullet` object, it ends up invoking `Bullet::beStruckBy`, passing in `*w` as the parameter called `w`.
3. `Bullet::beStruckBy` will then invoke `w.strike( *this )`.
4. So, we once again have to determine what the actual object type that `w` is bound to, which is a `Rock`.
5. `Rock` has overloaded the `strike` method, so it determines which one to call by looking at the type of `*this`, which is a `Bullet`.
6. We then end up calling `Rock::strike(Bullet & b)`, which performs the appropriate set of actions.

This ability to perform double dispatch is at the core of the Visitor design pattern.

## Code example

Let's now look at using the Visitor design pattern to add functionality to a set of existing classes in a hierarchy without changing or recompiling the classes themselves. We'll use our collection of books to illustrate the idea by adding a `BookVisitor` class and a concrete `Catalogue` class to the hierarchy that can be used to traverse the collection and perform the following actions:

- count the number of distinct book authors,
- count the number of distinct text topics, and
- count the number of distinct comic heroes.



By introducing the abstract base class `BookVisitor`, we can easily change out what operation we want to perform upon the Book hierarchy by creating a different concrete subclass that overrides what action(s) `visit` performs for each concrete class in the Book hierarchy.

| File:BookVisitor.h  | File:catalogue.h   |
|---|--|
| <pre>#ifndef __BOOKVISITOR_H__ #define __BOOKVISITOR_H__  class Book; // Forward declarations class Text; class Comic;  class BookVisitor { public:     virtual void visit( Book &amp; b ) = 0;     virtual void visit( Text &amp; t ) = 0;     virtual void visit( Comic &amp; c ) = 0;     virtual ~BookVisitor(); };  #endif</pre> | <pre>#ifndef __CATALOGUE_H__ #define __CATALOGUE_H__  #include &lt;map&gt; #include &lt;string&gt;  #include "BookVisitor.h"  struct CatalogueVisitor: public BookVisitor {     std::map&lt;std::string, int&gt; theCatalogue;      virtual void visit( Book &amp; b ) override;     virtual void visit( Comic &amp; c ) override;     virtual void visit( Text &amp; t ) override; };  #endif</pre> |
| File: BookVisitor.cc  | File: catalogue.cc   |
| <pre>#include "BookVisitor.h"</pre>   | <pre>#include "catalogue.h"</pre>  |

|   |   |
|---|---|
| <pre>#include "book.h" #include "text.h" #include "comic.h"  BookVisitor::~BookVisitor() {}</pre> | <pre>#include "book.h" #include "text.h" #include "comic.h" using namespace std;  void CatalogueVisitor::visit( Book &amp; b ) { ++theCatalogue[b.getAuthor()]; }  void CatalogueVisitor::visit( Comic &amp; c ) { ++theCatalogue[c.getHero()]; }  void CatalogueVisitor::visit( Text &amp; t ) { ++theCatalogue[t.getTopic()]; }</pre> |
|---|---|

Our main program consists of:

| File: main.cc  | Execution  | Explanation  |
|--|--|--|
| <pre>#include &lt;string&gt; #include &lt;vector&gt; #include "book.h" #include "text.h" #include "comic.h" #include "catalogue.h"  int main() {     std::vector&lt;Book*&gt; 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 Text{ "Annotated Reference Manual", "??", 200, "C++" },         new Comic{ "Aquaman Swims Again", "??", 20, "Aquaman" },         new Comic{ "Clark Kent Loses His Glasses", "??",                   20, "Superman" },         new Comic{ "Superman Saves the Day", "??", 20, "Superman" }     };      CatalogueVisitor v;      for ( auto &amp; b : collection ) b-&gt;accept(v);      for ( auto &amp; i : v.theCatalogue )</pre> | <pre>\$ ./main Aquaman 1 BASIC 1 C++ 2 Potter 1 Superman 2 Tolstoy 1</pre> | <ul style="list-style-type: none"> <li>The program creates a <code>std::vector</code> of <code>Book</code> pointers that is initialized with a set of <code>Book</code>, <code>Text</code>, and <code>Comic</code> objects allocated on the heap.</li> <li>A <code>CatalogueVisitor</code> object, <code>v</code>, is created.</li> <li>We iterate over the book collection, asking each element to accept the concrete visitor object, <code>v</code>. Note that we're using <a href="#">Iterator</a> to iterate over the STL vector and map containers.             <ul style="list-style-type: none"> <li><code>v</code> adds 1 to the author count if it's being accepted by a book object.</li> </ul> </li> </ul> |

```
    std::cout << i.first << " " << i.second << std::endl;  
  
    for ( auto & b : collection ) delete b;  
}
```

- v adds 1 to the hero count if it's being accepted by a comic object.
- v adds 1 to the topic count if it's being accepted by a text object.
- If the author/hero/topic didn't previously exist, the map element is created and the count is set to 1.
- We then iterate through the std::map in v, printing out the information it is indexed by (author/hero/topic), and the associated count.
- We then free the heap-allocated memory.

The full version of the code can be found in [lectures/se/05-visitor](#).

# Bridge

The **Bridge** design pattern decouples an abstraction from its implementation, allowing the two to vary independently. It is also known as **Handle/Body**. Each forms their own inheritance hierarchy. In particular, the two hierarchies are separate from each other, and if we weren't using the pattern, we'd need to have a concrete class for every combination of the abstraction and the implementation. This way, we can extend either side without impacting the other.

For more reading on the topic, you can read chapter 14, the appendix, of the "[Head First Design Patterns](#)" book by Freeman and Freeman.

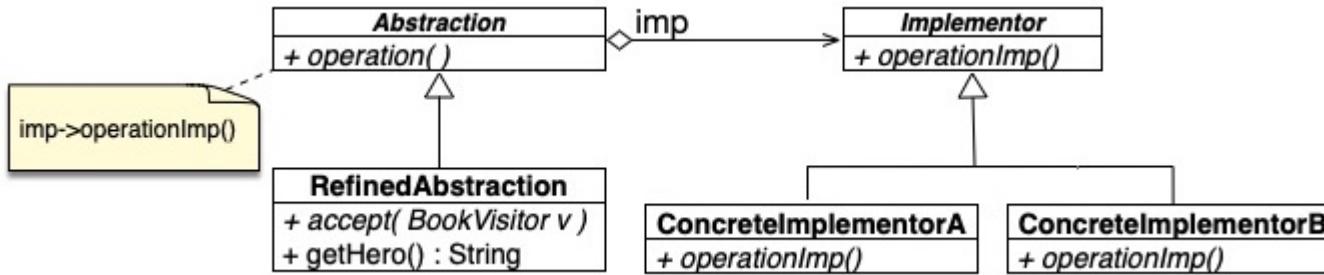
## Examples of use

Common examples of use are:

- The classic example from the "Design Patterns" book is of a portable Window abstraction in a user interface toolkit called [ET++](#). The toolkit needs to work on different platforms, each of which has their own way of drawing windows. So, they provide an abstract base class for the implementation that all of the platforms subclass, and an abstract base for the interface of how to draw a type (e.g. regular window, icon window, transient window).
- The [NeXT AppKit](#) used the bridge pattern for the implementation and display of graphical images.

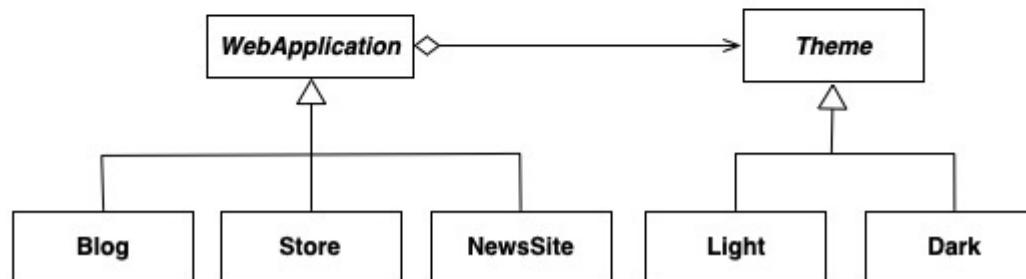
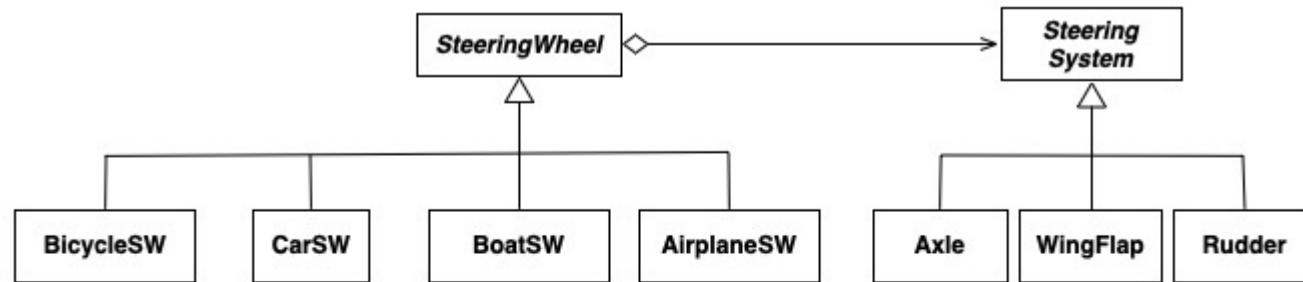
## General UML class model

The general form of the Bridge design pattern's UML class model diagram looks like:



We will not be going into any further details on the design pattern. However, there are several websites that provide nice simple examples of how it might be applied.

- Simple Programmer provides 2 examples at the site <https://simpleprogrammer.com/design-patterns-simplified-the-bridge-pattern/>. The first example describes steering systems (axle, rudder, flap) to which a steering wheel (car, boat, airplane, bicycle) could be applied. The second describes a hierarchy of web applications (blog, store, news site) to present in themes (light, dark).



- Source Making describes a thread-scheduling system for different operating system platforms at [https://sourcemaking.com/design\\_patterns/bridge](https://sourcemaking.com/design_patterns/bridge).
- The "Head First Design Patterns" book has an example of a remote control for different TV brands.

# Coupling and Cohesion Revisited

Now that we've spent some time looking at improving the design of our code using design patterns, let's revisit these concepts through the lens of software design. We've already defined the concepts of [coupling and cohesion](#) before. If you don't remember them, please quickly review them before continuing.

Let's look at some ways of designing modules, and how that lets us judge the amount of coupling and cohesion that exists in our design. In particular, the list starts with techniques that provide either low coupling or low cohesion, and end with techniques that indicate high coupling or high cohesion.

| Coupling   | Cohesion   |
|--|--|
| <ul style="list-style-type: none"><li>• (<i>low</i>) modules communicate via function calls with basic parameters/results</li><li>• modules pass arrays/structs back and forth</li><li>• modules affect each other's control flow</li><li>• modules share global data</li><li>• (<i>high</i>) modules have access to each other's implementations (e.g. friends)</li></ul> | <ul style="list-style-type: none"><li>• (<i>low</i>) arbitrary grouping of unrelated elements (e.g. &lt;utility&gt;)</li><li>• elements share a common theme but are otherwise unrelated; might share base code (e.g. &lt;algorithm&gt;)</li><li>• elements manipulate state over the lifetime of an object (e.g. open, read, and close files)</li><li>• elements pass data to each other</li><li>• (<i>high</i>) elements cooperate to perform exactly one task</li></ul> |

Remember, the goal is to have low coupling, and high cohesion! If we have high coupling, then it becomes harder to reuse individual modules since changes to one require changes to others. If we have low cohesion, then our code is poorly organized, hard to understand, and hard to maintain.

If we apply the goal of low coupling and high cohesion consistently in our designs, then we're effectively applying the design principle called the **Single Responsibility Principle** (SRP). (There's a nice discussion of it at

<https://stackify.com/solid-design-principles/> that you may find interesting.) All of the design patterns that we've seen decoupled classes by introducing abstract base classes, which let the client "program to the interface, and not to the implementation".

Let's take a look at a few design examples and examine how well they apply the Single Responsibility Principle.

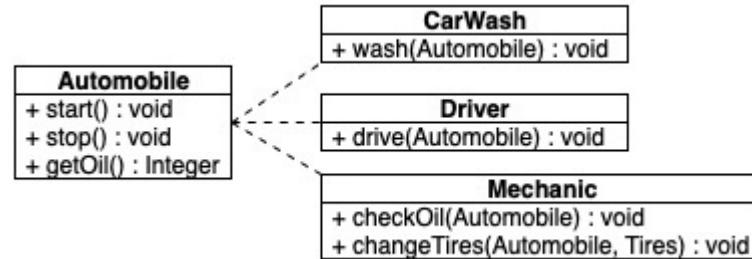
## Cohesion Examples

### Example 1

Our first one consists of an Automobile class with methods to start and stop (used when driving), check the oil, which involves determining how many liters are left and how the oil looks (clean or dirty), and change the tires.

| Automobile                  |
|-----------------------------|
| + start() : void            |
| + stop() : void             |
| + getOil() : Integer        |
| + wash() : void             |
| + checkOil() : void         |
| + changeTires(Tires) : void |

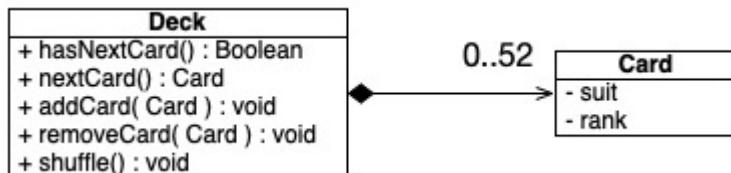
How cohesive is this class? Well, think about how many reasons it might have to change. If the driver changes their driving style; if the actions involved in the car wash change (maybe a deluxe version with detailing? No-touch versus cloth?); if the mechanic changes their process for checking and changing oil or changing tires. Any of these could cause the class to change its implementation. It would be better to separate out these into separate classes by adding CarWash, Mechanic and Driver classes, as in the diagram below:



Now each focuses on a single task. The CarWash implements how to wash a car. The Driver implements the driving functionality. The Mechanic deals with checking oil and changing tires.

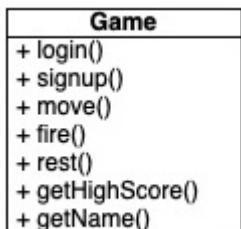
## Example 2

What about a deck of cards? It keeps track of its contents, and their relative ordering. It is possible to iterate through its contents. This is pretty cohesive.



## Example 3

What about the following video game UML class model?



- Game contains multiple independent responsibilities.
- Game is fine as is.

Yes, that's great!

## Example 4

What about the following shopping cart UML class model for an online store?

| ShoppingCart     |
|------------------|
| + add( Item )    |
| + remove( Item ) |
| + checkOut()     |
| + saveForLater() |

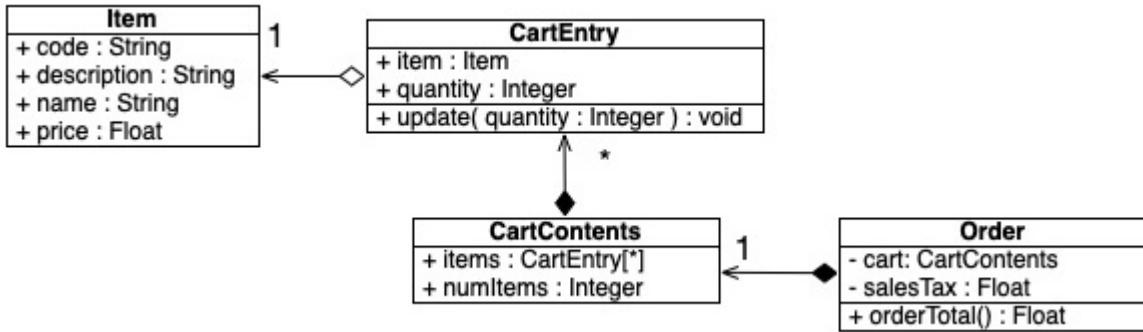
- ShoppingCart contains multiple independent responsibilities.
- ShoppingCart is fine as is.

Yes, that's great!

## Coupling Examples

### Example 1

Consider the shopping cart application example, presented at <https://stackoverflow.com/questions/226977/what-is-loose-coupling-please-provide-examples>. The class model translated to C++ looks like:



The `Order::orderTotal()` method could be implemented as:

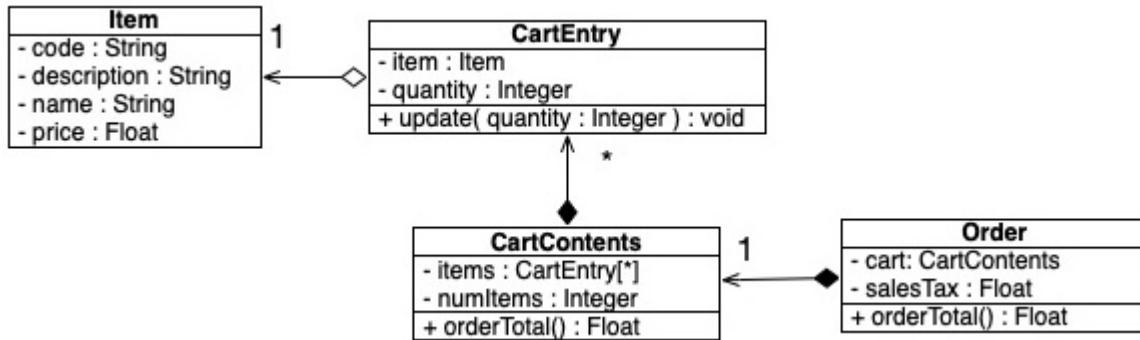
```

float Order::orderTotal() {
    float cartTotal = 0;
    for (int i = 0; i < cart.numItems; i++) {
        cartTotal += cart.items[i].item->price * cart.items[i]->quantity;
    }
    cartTotal += cartTotal*salesTax;
    return cartTotal;
}

```

The implementation depends upon knowing the implementation details of the **Item**, **CartEntry**, and **CartContents** classes. If any of those change, the implementation needs to change. For example, there may be discounts applied if more than a certain quantity of specific items are bought, or the **CartContents** might use an `std::vector` instead of an array to store the cart entries.

We can improve things by encapsulating the information more strictly and delegating the total of the order cost the **CartContents** class. (Assume all proper accessors, constructors and destructors have been provided.) The class model then becomes:



The `Order::orderTotal()` method is now implemented as:

```

float Order::orderTotal() {
    float cartTotal = cart.orderTotal();
    cartTotal += cartTotal*salesTax;
    return cartTotal;
}

```

## Pulling the pieces together

Here's an interesting article by Clare Sudbery that looks at a refactoring case study of a "too-large" class, available at <https://martinfowler.com/articles/class-too-large.html>. It's also a good introduction to the concept of **refactoring**. Martin Fowler has written the classic book on the subject, and you can find a good introduction to the concept at <https://refactoring.com/> as well as his formal definition of the term. You have already been applying some of the ideas without knowing it, so take a look to see what other ideas you might recognize. You might also find some ideas on how to re-structure code that you haven't thought of before!

# Model-View-Controller (MVC)

## Introduction

Decoupling the interface from the program logic is a good way to improve the quality of our code. To keep cohesion high, each one of our classes should be responsible for only one task. This is the **single responsibility principle**: "A class should have only one reason to change."

For example, imagine you are implementing a chess game. You can create a class named `ChessBoard` that implements the logic of a game of chess. However, it should not be printing things, like this:

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

This is bad design, as it inhibits code reuse. What if you want to reuse the `ChessBoard` class, but not have it communicate via `stdout`? A better solution could be to give the class stream objects with which it can do its input/output:

```
class ChessBoard {  
    istream &in;  
    ostream &out;  
public:  
    ChessBoard (istream &in, ostream &out): in{in}, out{out} {}  
    . . .
```

```
    . . . . . out << "Your move"; . . . . .
    . . .
};
```

This seems better. But what if we don't want to use streams at all, e.g., what if we want a graphical interface? So, this is actually not good enough yet. Ideally, your `ChessBoard` class should not be doing any communication at all. Its responsibility is to maintain the *game state*. According to the *single responsibility principle*, it should have no other responsibilities besides that.

Better design: the `ChessBoard` should communicate via parameters/results/exceptions only. Confine the actual user interaction outside the game class. By doing so, we have total freedom to change how the game interacts without modifying the logic in the `ChessBoard` class.

So, should `main` do all of the communication and then call `ChessBoard` methods? No. It is hard to reuse code if the presentation logic is in `main`. The solution is to have a separate class to manage interaction.

## The Model-View-Controller Architecture (MVC)

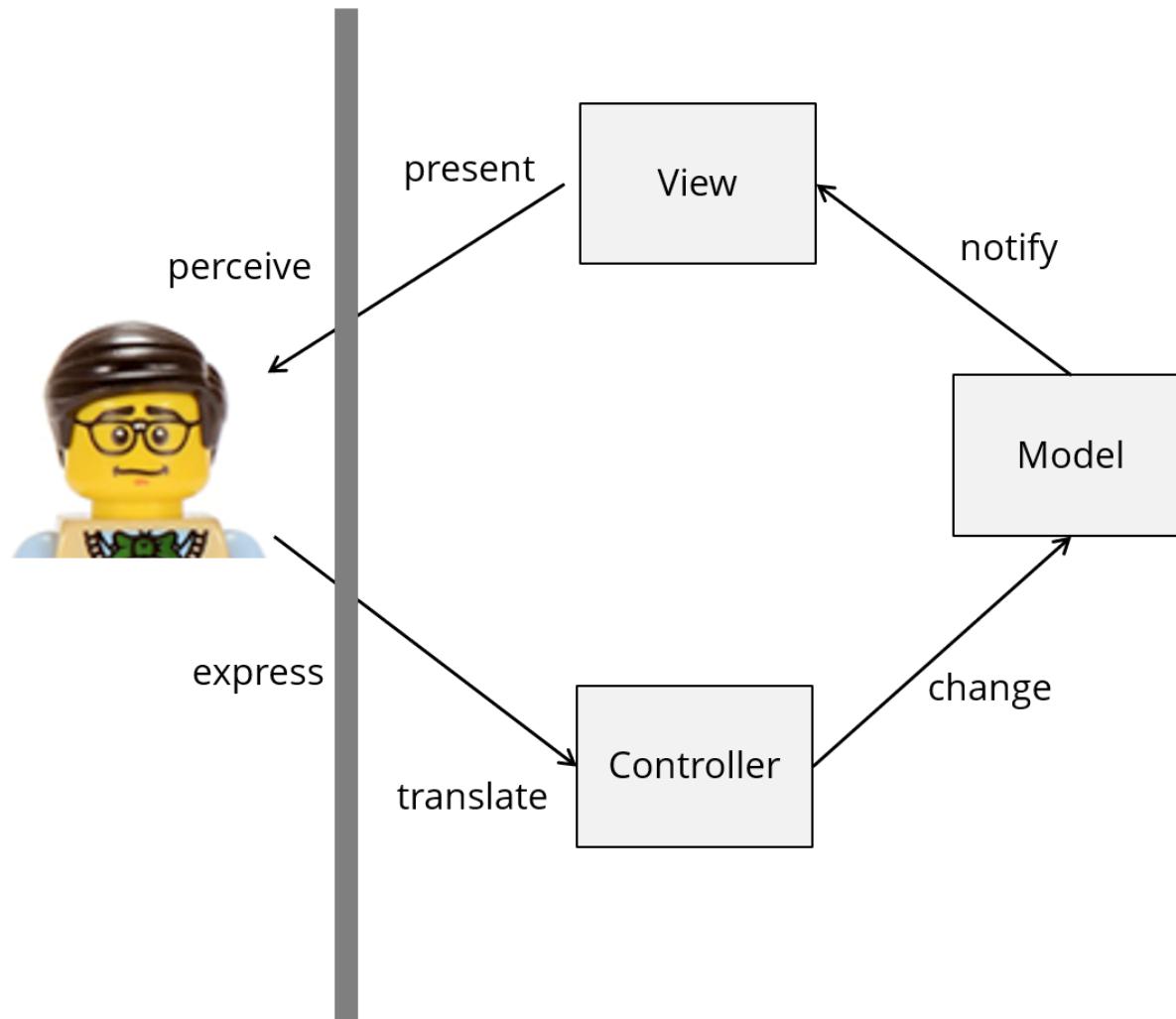
The **Model-View-Controller (MVC)** architecture presents a solution to this common issue. In MVC, the program state, presentation logic, and control logic are all separated. Therefore, it is in theory possible to modify any one of them without modifying the other two.

MVC was developed at Xerox PARC in 1979 by Trygve Reenskaug for the Smalltalk-80 language, the precursor to Java. It is until today one of the most widely used standard design patterns for graphical user interfaces. It is also sometimes used with some variations, like Model-View-Presenter or Model-View-Adapter.

(Note: MVC is widely identified as a design pattern. However, some people argue that it is just a standard architectural idea, but not exactly a design pattern. This is because there is an understanding that a design pattern should be followed accurately, whereas many people follow the basic MVC design loosely and make several

adaptations. If you're interested in the details, we suggest this [long discussion by Martin Fowler](#). This is just for curiosity; for this course, we are more concerned in teaching you to implement MVC in C++, not in the philosophical discussion of how to classify it.)

In MVC, the interface architecture is decomposed into three parts:



- **Model:** manages application data and its modification. It doesn't know anything about how the data is presented to the user.

- **View:** manages the interface to present data. It decides how the data from the model should be presented to the user according to the capabilities of the device (e.g., a graphical interface, a text-based interface, a conversational interface, etc.)
- **Controller:** manages interaction to modify data. It receives user input (e.g., mouse clicks, finger taps, keyboard input, audio input, etc.) and translates it to the actions that operate in the program state stored in the model.

## Implementation

The Model and View in MVC are a classic implementation of the [Observer](#) design pattern, where the Model is the subject and the View is the observer. This allows multiple views to receive notifications every time the model changes. For example, in PowerPoint, you can see a large visual representation of the current slide, as well as a smaller version in the left navigation bar. When one of them changes, the other must be updated too. This can be accomplished by having two views observing the same model data.

1.1 Introduction

Search in Presentation

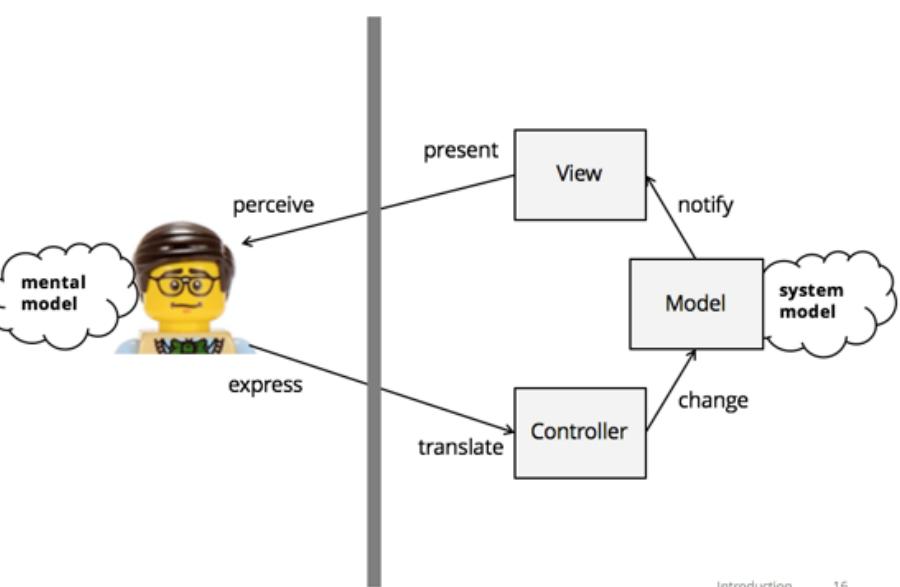
Home Insert Design Transitions Animations Slide Show Review View > Share

Paste Slides Open Sans 22 A A A Paragraph Insert Drawing

B I U abe X<sup>2</sup> X<sub>2</sub> AV Aa A

14 Interactive System Architecture User → Interactive System

15 Interactive System Architecture User → Interactive System

16 Model-View-Controller (MVC) 

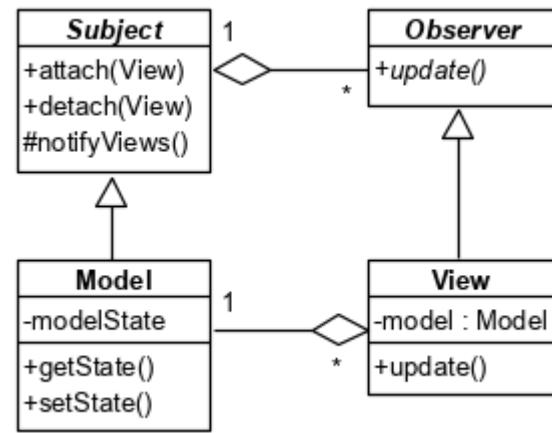
17 Graphical Temperature Control

18 Speech Temperature Control Click to add notes

Slide 16 of 35 Notes Comments E - + 68% 

The view may be responsible for receiving input from the user, for example, if the view contains buttons that capture clicks or text boxes that capture text input. However, the view should not carry out these actions directly; they should all be directed to the Controller. Therefore, the relationship between the View and the Controller is an example of the [Strategy](#) design pattern. (But note that a common adaptation to the pattern is to just implement the Controller within the View.) In turn, the Controller is not responsible for actually making any change in the program state. It should only interpret what the user wants and call one of the methods of the Model.

The following diagram shows the usual implementation of MVC as an instance of the Observer design pattern.



Note: the Model does not need to have methods named `getState` and `setState`; this is just for illustration. Instead, the Model should have accessors, mutators, and public operations just as necessary to perform its responsibility while maintaining [encapsulation](#). However, implementing MVC usually requires partially breaking encapsulation, as the View often needs to read the value of the internal state of the Model so the information can be presented in the graphical interface.

## Example 1

For example, in our chess game, we could display a graphical representation of the board. The player would drag a piece to make a move.

1. The view would have some method that responds to a mouse drag. This method would delegate handling the player action to the Controller. Something like this:

```
void ChessView::handleMouseDown(Point start, Point end) {  
    controller->handleMouseDown(start, end);  
}
```

2. The controller should translate the user interface event to a game model event. So, the controller should identify what piece was dragged into what board position. Then, ask the model to handle this game action. Something like this:

```
void ChessController::handleMouseDown(Point start, Point end) {  
    Piece piece = findPiece(start);  
    Position position = findBoardPosition(end);  
    model->movePieceTo(piece, position);  
}
```

3. The model contains the logic to handle the game action. First, it must validate if the move is a valid one. If it is, then the move should be executed by updating the state. Finally, the views (observers) are notified of the change. Something like this:

```
void ChessModel::movePieceTo(Piece piece, Position position) {  
    if (isValidMove(piece, position)) {  
        // update the state of the internal fields to put the piece in the  
        // new  
        // position; check other consequences of the new piece position,  
        // e.g., if the player won the game, captured an enemy piece, etc.  
    }  
}
```

```
    notifyViews();
}
}

void ChessModel::notifyViews() {
// subject implementation of the observer design pattern
for (int i = 0; i < views.size(); i++) {
    views.at[i].update();
}
}
```

4. Each view has a concrete implementation of the update method (observer design pattern), which should display the new state of the game board. Something like this:

```
void ChessView::update() {
    // read the updated state from the model and
    // redraw the game board in the user interface
}
```

As discussed above, some of the benefits of this approach are increasing cohesion (as each class has only one responsibility), decreasing coupling (as each class communicates with the other just via a public interface), and making reuse easier. For example, we could replace the graphical view with different ones, like a text-based view or a mobile app, without any modification in the model, and just a few modifications in the controller. Or we could instead reuse the view for other types of board games like checkers, by connecting it to a different model and just making the necessary adjustments to the visual display algorithm.

## Example 2

There is a complete example of MVC in C++ in the folder `lectures/se/06-mvc` in our Git repository. Please take some time to study it and understand how it was designed and how it works.

In this example, the Model stores data about a sorted deck of cards. The views display the card currently at the top of the deck. There are two possible user actions: Next (updates the model to the next card in the deck) and Reset (updates the model to go back to the first card). There are two views. The class `View` (used by `main.cc`) displays a graphical interface (a window with a graphical representation of the cards) and receives user input from buttons. The class `TextView` (used by `main_text.cc`) displays a text interface and asks user input from `stdin`.

Note: This example uses GTK, a library for graphical user interfaces. It works (compiles and runs) in the student environment. But you need to have the X Window System working in your computer and you need to use X forwarding to be able to run the program via SSH and see the graphical interface on your computer. Check the specification for Assignment 4 for more information about running graphical programs.

## Caveats of MVC

MVC is a good design pattern that is widely used in the world, particularly when creating graphical user interfaces. However, it is also important to know that it has some caveats. Sometimes, we just need to deal with them while implementing our program. Sometimes, people chose to use variations of the original MVC to handle some of these caveats.

1. **Encapsulation is broken.** Usually, the view needs to have access to the internal state of the model, so it can translate it into the display elements. Therefore, we need to expose accessor methods for fields of the model that would not need to be public otherwise. Similarly, the controller often needs to have access to some information about the view to translate the user intent. For example, in our chess example above, note that the controller's `findPiece` and `findPosition` methods need to identify a piece and a board position based on the

visual coordinates of the mouse click. Therefore, the controller needs to know about the visual coordinates that are part of the internal state of the view. This partial loss of encapsulation is inevitable when using MVC, so we need to be aware of it and work with it.

2. **It may be difficult to separate View and Controller.** For example, the view must often handle input and output because they happen on the same screen. This creates a high coupling between the View and the Controller classes, partly defeating the purpose of the separation between them. For this reason, the Controller class is sometimes eliminated and all the Controller logic is implemented within the View. This generally makes the implementation easier, but reuse is more difficult.
3. **Sometimes, the View needs to manipulate the data.** In theory, the view should just display the data that it reads from the model. But sometimes, there are view-level data manipulations that don't make sense to the model. For example, it may be necessary to format numbers, currency, or dates according to the current user locale, or to check user input for valid formats of a phone or email. Sometimes, this caveat is handled by implementing an Adapter class instead of (or together with) the Controller. The responsibility of this Adapter class is to format the data for the view or to validate the input from the view before passing it to the Controller or the Model.
4. **One common Model doesn't always make sense.** We often have data/state that is specific to a View. e.g. different logic on a screen based on region or language. Again, people sometimes handle this issue by adding an Adapter or Presenter class to deal with these different types of view-related data. There might also be situations where the program data is split between different models. For example, a web browser may need to display, at the same time, a web page (which would be stored in one Model object) and a bar with a history list of the last few visited pages (this history would be stored in another type of Model object). This creates couplings between the View class and more than one Model objects, making reuse more difficult.

# How virtual methods work

One thing that we haven't really discussed is how exactly the correct virtual method is chosen dynamically at run-time. In order to make sense of this, we first need to have some idea of how an object is laid out in memory. Let's start by looking at our Vector class, as defined in your repository in the file lectures/c++/11-vtable/vectors.cc.

```
#include <iostream>
#include <cmath>

class Vector {
public:
    int x, y;
    Vector(int x, int y): x{x}, y{y} {}
    int supNorm() { return max(abs(x), (y)); }
};

class Vector2 {
public:
    int x, y;
    Vector2(int x, int y): x{x}, y{y} {}
    virtual int supNorm() { return max(abs(x), abs(y)); }
};

int main() {
    Vector v{1,2};
    Vector2 w{1,2};

    std::cout << sizeof(int) << " " << sizeof(v) << " " << sizeof(w) << std::endl;

    v.supNorm();
    w.supNorm();

}
```

When we compile and run this, we see the following output:

```
$ ./a.out  
4 8 16
```

You may remember from before than an integer takes up 4 bytes of memory. Given that a `Vector` object has exactly 2 integer data fields, and an integer takes up 4 bytes, then all 8 bytes of the space allocated to `v` is taken up by the two data fields. (We'll talk about why `w` takes up 16 bytes in a moment.) So how does the operating system know where to find the `supNorm` function? Well, it has an address in memory, and any call to the function is replaced by a transfer to that memory location. Let's look at our program in `gdb`.

After compiling, loading our program into `gdb`, setting a breakpoint on line 24 (`v.supNorm()`) and running our program, we can see the following:

```
(gdb) break 24  
Breakpoint 1 at 0xadb: file vectors.cc, line 24.  
(gdb) run  
Starting program: /srv/DFSc/cs-teaching/home/u1/ctkierst/cs246/1205/lectures/c++/11-vtable/a.out  
4 8 16  
  
Breakpoint 1, main () at vectors.cc:24  
24          v.supNorm();  
(gdb) info frame  
Stack level 0, frame at 0x7fffffff960:  
  rip = 0x555555554adb in main (vectors.cc:24); saved rip = 0xfffff7631b97  
  source language c++.  
  Arglist at 0x7fffffff950, args:  
  Locals at 0x7fffffff950, Previous frame's sp is 0x7fffffff960  
  Saved registers:  
    rbp at 0x7fffffff950, rip at 0x7fffffff958  
(gdb) s  
Vector::supNorm (this=0x7fffffff928) at vectors.cc:8  
8          int supNorm() { return std::max(abs(x), (y)); }  
(gdb) info frame  
Stack level 0, frame at 0x7fffffff920:  
  rip = 0x555555554b9c in Vector::supNorm (vectors.cc:8);
```

```
saved rip = 0x555555554ae7
called by frame at 0x7fffffff960
source language c++.
Arglist at 0x7fffffff910, args: this=0x7fffffff928
Locals at 0x7fffffff910, Previous frame's sp is 0x7fffffff920
Saved registers:
    rbp at 0x7fffffff910, rip at 0x7fffffff918
(gdb) bt
#0  Vector::supNorm (this=0x7fffffff928) at vectors.cc:8
#1  0x000055555554ae7 in main () at vectors.cc:24
(gdb)
```

Note that frame 0 is always the activation frame at the top of the stack, and sp is the stack pointer. rbp is the register base pointer, and contains the address of the base of the current stack frame. rsp is the register stack pointer, and contains the address of the top of the current stack frame. rip is the register instruction pointer, which contains the address of the next instruction to execute. (It's not important that you know these registers, this is just explaining what they are.)

The part that we're interested in is the value of rip before the call to v.supNorm() and the value of rip once we're executing v.supNorm(). If you compare the [hexadecimal](#) (base 16) values, 0x55555554adb and 0x55555554b9c, they're actually numerically close to each other (193 bytes apart). So, the address of supNorm is in the segment associated with the program code, and not on the heap or in the run-time stack.

In particular, the compiler changed the method to an ordinary function, and stored it separately from the object. What happens if we declare supNorm to be virtual, as in the class Vector2? Everything else in Vector2 is the same as in Vector, except for the virtual method declaration. But now the size of the w object is 16 bytes, not 8.

Let's take a closer look at it in the debugger by putting a breakpoint on line 25.

```
(gdb) break 25
Breakpoint 1 at 0xae7: file vectors.cc, line 25.
(gdb) run
Starting program: /srv/DFSc/cs-teaching/home/u1/ctkierst/cs246/1205/lectures/c++/11-vtable/a.out
4 8 16
```

```

Breakpoint 1, main () at vectors.cc:25
25          w.supNorm();
(gdb) p w
$1 = {_vptr.Vector2 = 0x555555755d68 <vtable for Vector2+16>, x = 1, y = 2}

```

While `w` still contains the data fields `x` and `y`, it now also contains a `vptr` (**virtual pointer**) to a **vtable**, a table of function pointers to virtual methods. (While the C++ standard doesn't require the compiler to use a vtable to implement virtual tables, most compilers use it.) Every instance of the class now contains a virtual pointer to the vtable.

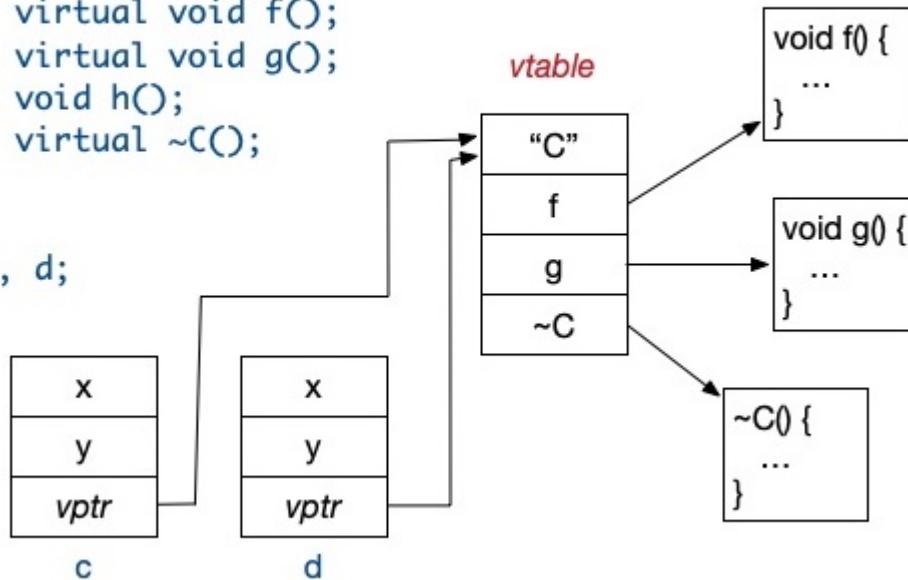
For example, for a simple class `C` that contains several virtual methods, we would think of the objects being laid out in memory as looking something like the following:

```

class C {
    int x, y;
    virtual void f();
    virtual void g();
    void h();
    virtual ~C();
};

C c, d;

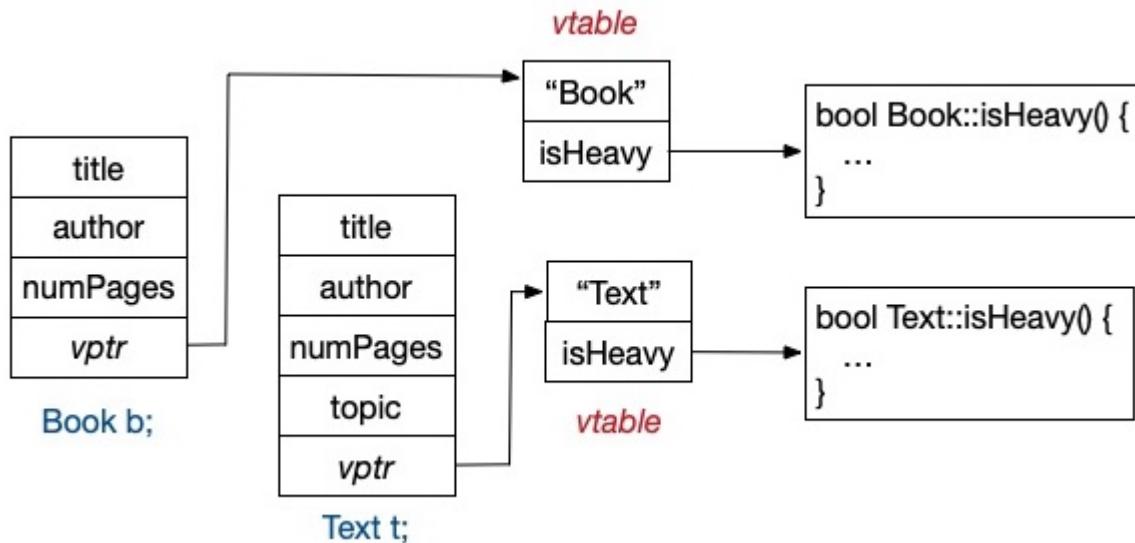
```



Each of the objects `c` and `d`, have a field for the `vptr`, and the two data fields `x` and `y`. Each `vptr` points to the `vtable` (virtual table) for the class. The `vtable` contains an entry for each virtual method, that points to the location of the

method code in memory.

When we add inheritance into the diagram by revisiting our Book and Text example, the diagram looks more like the following:



Each object has the data fields appropriate to its type (a `Book` has `title`, `author`, and `numPages`, while a `Text` has `title`, `author`, `numPages`, and `topic`), and the `vptr`. The `vptr` for a `Book` points to the `Book` vtable, while the `vptr` of the `Text` points to the `Text` vtable. Each of the vtables has an entry for the virtual `isHeavy` method, and they point to the different locations where each class implementation is stored in memory.

Let's say that our code has declared a pointer of type (`Book*`), `bptr`, that happens to currently point to `Text` object when we want to output the result of `bptr->isHeavy()` to standard output.

```
Book * bptr;  
...  
std::cout << bptr->isHeavy() << std::endl;
```

In order to call the correct virtual method for whatever `bptr` is pointing to, we need to:

1. Follow the object's vptr to the vtable.
2. Fetch the address for the `isHeavy` method for that class.
3. Go to the address for that `isHeavy` method.
4. Execute the method code at that address.

All of this happens at run-time. As a result, there is a small performance penalty paid to resolve the virtual method call. As well, each object of a class that has a virtual method now contains a vptr, so an object of a class that has no virtual methods will be smaller than a similar class with one (or more) virtual methods.

How is the information actually laid out in memory? Well, that depends upon the compiler implementation. Let's consider a simple program in your repository, `lectures/c++/11-vtable/inheritance.cc`. The code is shown below:

```
#include <iostream>

class A {
public:
    int a, c;
    A(): a{100}, c{300} {}
    virtual int f (){ return a+1; }
};

class B: public A {
public:
    int b, d;
    B(): A{}, b{200}, d{400} {}
    virtual int g(){ return b-1; }
};

int main () {
    A a;
    B b;
```

```

int *pa = reinterpret_cast<int*>(&a);
int *pb = reinterpret_cast<int*>(&b);

for (int i = 0; i < sizeof(a)/sizeof(int); ++i) std::cout << pa[i] << std::endl;
std::cout << std::endl;
for (int i = 0; i < sizeof(b)/sizeof(int); ++i) std::cout << pb[i] << std::endl;
}

```

In particular, notice that it is using the C++ `reinterpret_cast` to convert the address of each object into the address of an array of integers. This means that we can then iterate over the array, printing out each element as if it was an integer.

In g++, the vptr is the first data field in the object. Then the data fields a and c follow, in that order, for an object of type A. For an object of type B, which inherits from A, the vptr is followed by A's data fields, then B's data fields. So, structurally, it looks like the following diagram:

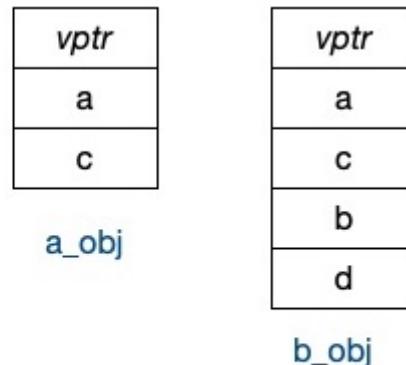
```

class A {
    int a, c;
    virtual void f();
};

class B : public A {
    int b, d;
};

A a_obj;
B b_obj;

```



When we run the program compiled `inheritance.cc` program, we see output similar to the following:

```

$ ./a.out
2004188504
21983
100

```

300

2004188472

21983

100

300

200

400

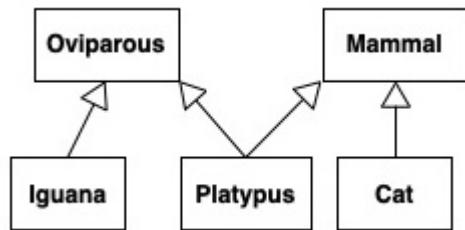
Since the vptr pointers are 8-bytes long in C++, they take up 2 lines worth of information i.e. 2 integers worth since an integer is 4-bytes long. (Note that their values may change on every program execution) Their values in the output are shown in red, and are the first 2 lines output from each for loop. The next two numbers are the initial values of the A object's data fields shown in blue, 100 and 300. The final two values, 200 and 400, are shown in green and are the initial values of the B object's data fields. Thus you can see that the information in the object is laid out consecutively.

# Multiple Inheritance

## Introduction

It turns out that in C++ you can inherit from more than one class. This isn't something that all languages allow. Java, for example, only lets you inherit from one class, though you *can* implement more than one interface.

Let's consider the following simple example where the class `Monotreme` (a mammal that lays eggs, e.g., platypus) inherits from both classes `Oviparous` (Latin for an animal that lays eggs) and `Mammal`. You can find the full code example in your repository under the directory `lectures/c++/11-vtable/multi`.



| File: monotreme.h  | File: monotreme.cc  |
|--|---|
| <pre>#ifndef _MONOTREME_H #define _MONOTREME_H #include "mammal.h" #include "oviparous.h"  class Monotreme: public Mammal, public Oviparous { public:     virtual void description();     virtual ~Monotreme() = 0; };  #endif</pre> | <pre>#include "monotreme.h" #include &lt;iostream&gt;  void Monotreme::description() {     std::cout &lt;&lt; "\tMonotreme:" &lt;&lt; std::endl;     Mammal::description();     Oviparous::description(); }  Monotreme::~Monotreme() {}</pre> |

Let's take a look at the abstract base class `Monotreme`. You will notice that the syntax for multiple inheritance is to give a comma-separated list of the classes which are being inherited from, with the type of inheritance specified for each. The implementation of `Monotreme::description` invokes the `description` method from each superclass; however, since they have the same signature, the calls are considered to be **ambiguous** by the compiler and rejected unless we use the scope resolution operator (`::`) to specify which method we want. We want to call both.

This lets the driver now create an instance of the `Monotreme` concrete subclass `Platypus`, and invoke its `description` method:

```
Monotreme * ptr1 = new Platypus;  
ptr1->description();
```

This results in the output:

```
Platypus:  
    Monotreme:  
        Gives milk to its young.  
        Lays eggs.
```

since `Platypus::description` calls `Monotreme::description` after printing the string "Platypus" to standard output.

## Issues

If multiple inheritance is so useful, why isn't it used more? Let's consider a simple example where the class D inherits from both classes B and C.

```
#include <iostream>  
using namespace std;  
  
class A {  
public:
```

```
int a = 100;
A() {}
virtual ~A(){}
};

class B: public A {
public:
    int b = 200;
    B() {}
    virtual ~B(){}
};

class C: public A {
public:
    int c = 300;
    C() {}
    virtual ~C(){}
};

class D: public B, public C {
public:
    int d = 400;
    D() {}
    virtual ~D(){}
};

int main () {
    D d;
    unsigned int *p = reinterpret_cast<unsigned int *>(&d);

    std::cout << "Size of d: " << sizeof(d) << std::endl << std::endl;

    for (int i=0; i < sizeof(d)/sizeof(int); ++i) {
        std::cout << p[i] << std::endl;
    }
}
```

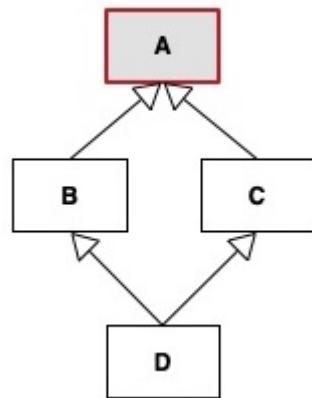
In principle, this seems reasonable enough; however, classes B and C both inherit from class A. So if I have an object of type D, dobj, and I try to access the data field it inherited from A, which version should it use? (This is the same problem we saw in the `Monotreme::description` method, where we needed to disambiguate which of the superclass `description` methods we wanted to call.)

```
D dobj;  
dObj.a; // whose 'a' is this?
```

We can use the scope resolution operator `::` to pick which version of a we'd like to access.

```
dObj.B::a;  
dObj.C::a;
```

Either of these would be legal. However, we have a more fundamental problem. Right now, we have two copies of the data fields for class A since the default behaviour of the compiler is to treat them as being two different things. What if we only wanted one copy?



Our UML class model would look like the above, which is known variously as the **Dreaded Diamond**, the **Deadly Diamond of Death**, or just as the **deadly diamond** due to the shape of the inheritance hierarchy.

The solution is to make A a **virtual base class**, and specify that classes B and C use **virtual inheritance**. This tells the compiler that there will be only one instance of the superclass object in memory for the concrete subobject. (Note

that an actual example of this in "real life" occurs in the C++ I/O library, where `std::basic_ostream` virtually inherits from `std::basic_ios`, and `std::ostream` inherits from `std::basic_ostream`.)

Your repository shows the corrected version as the file `lectures/c++/11-vtable/virtual.cc`. The relevant part of the class declarations is shown below:

```
...
class A {
public:
    int a = 100;
    A() {}
    virtual ~A() {}
};

class B: virtual public A {
public:
    int b = 200;
    B() {}
    virtual ~B() {}
};

class C: virtual public A {
public:
    int c = 300;
    C() {}
    virtual ~C() {}
};

class D: public B, public C {
public:
    int d = 400;
    D() {}
    virtual ~D() {}
};

...

```

The next question to answer is how does virtual inheritance affect the memory layout? Let's start by taking a look at the layout for an object of type B from the above inheritance hierarchy through creative use of the C++ [reinterpret\\_cast](#). The code fragment is shown on the left, and the output on the right.

```
B b;  
  
unsigned int * p = reinterpret_cast<unsigned int *>(&b);  
std::cout << "Size of b: " << sizeof(b) << std::endl << std::endl;  
for (int i=0; i < sizeof(b)/sizeof(int); ++i) {  
    std::cout << p[i] << std::endl;  
}  
std::cout << std::endl;
```

```
Size of b: 32  
1114479800  
22085  
200  
22085  
1114479840  
22085  
100  
0
```

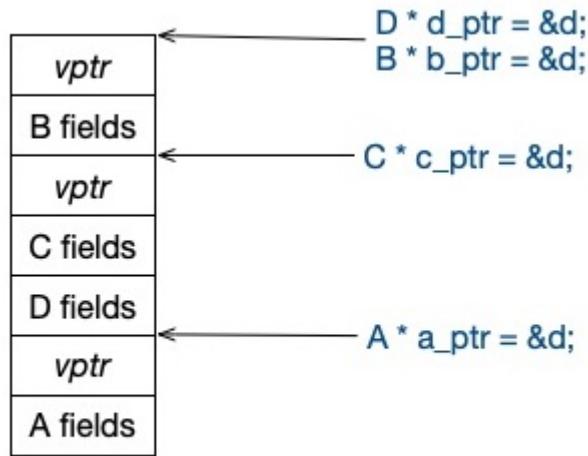
It's possible to spot the data fields for the portion inherited from A and the data field for B, picked out in blue in the output. But there are now other addresses in between the data fields. Let's compare this to what an object of type D looks like.

```
D d;  
  
unsigned int * p = reinterpret_cast<unsigned int *>(&d);  
std::cout << "Size of d: " << sizeof(d) << std::endl << std::endl;  
for (int i=0; i < sizeof(d)/sizeof(int); ++i) {  
    std::cout << p[i] << std::endl;  
}  
std::cout << std::endl;
```

```
Size of d: 48  
3406502760  
21928  
200  
21928  
3406502800  
21928  
300  
400  
3406502840  
21928  
100  
0
```

Here we can see the slices of the various subclasses that make up class D. It turns out that g++ layers the sections of the objects (vptr plus data fields) with an offset i.e. how many bytes away in the object is the section associated with the various superclasses such as A. This happens because at compile-time you don't know how many subclasses are in the hierarchy, and thus how far apart the pieces may be. Instead, we end up storing the location of the base class

part of the object in the vtables—this is where virtual inheritance gets its name. So, the diagram of the memory layout for an object of type D looks roughly like the following:



`D d;`

In particular, if I take the address of the object d and cast it to the pointer type of one of the superclasses (A, B, or C), this changes *which part of the object the pointer points to!* Note that `static_cast` and `dynamic_cast` under multiple inheritance will also adjust the value of the pointer; but, `reinterpret_cast` won't, so be very careful using `reinterpret_cast`.

---

A good resource for more reading on the topic is <https://isocpp.org/wiki/faq/multiple-inheritance>. It includes a very good discussion on alternatives to multiple inheritance, when you might want to use it, and the trade-offs.