

CSE 1325

Week of 08/24/2020

Instructor : Donna French

Canvas



2208-CSE-1325-006

2020 Fall
Home
Announcements
Syllabus
Modules
Discussions
Quizzes
Grades
People
Conferences
Collaborations
Course Evaluations
UTA Libraries

2208-CSE-1325-006-OBJECT-ORIENTED PROGRAMMING ↗

CSE 1325 - Object Oriented Programming



Instructor



Syllabus



Modules



Announcements



Getting Started



Q&A Discussion

View Course Stream

View Course Calendar

View Course Notifications

To Do

- Welcome! Aug 19 at 3:40pm |
- Survey is available! Aug 19 at 3:41pm |
- Survey Aug 20 at 4:54pm |
- First Day of Class Aug 24 at 1:48pm |
- Tomorrow!!! Aug 25 at 6:03pm |
- Tomorrow!!! Aug 25 at 6:05pm |

Welcome!

Welcome to CSE 1325. We will be learning C++ and object oriented programming this semester. We'll build on what you learned in CSE 1320 and some new concepts that will prove useful as you advance in your CSE education.

Syllabus

CSE 1325 – Object Oriented Programming

Fall 2020

Instructor Information

Instructor Donna French

Office Number ERB 652

Office Phone I don't have a phone in my office, but in case of an emergency you can call the CSE department at 817-272-3785.

Email Address donna.french@uta.edu (best way to contact me)

Faculty Profile <https://mentis.uta.edu/explore/profile/donna-french>

Office Hours Monday, Tuesday, Wednesday, Thursday 4:00 pm – 5:30 pm

Syllabus

Course Information

Section Information

CSE 1325 Section 005

CSE 1325 Section 006

CSE 1325 Section 900

Time and Place of Class Meetings

Monday, Wednesday 2:30PM – 3:50PM WH402 Section 005

Monday, Wednesday, Friday 11:00 AM – 11:50 AM WH404 Sections 006 and 900

Hybrid 3

Required Textbooks and Other Course Materials

A specific textbook is not required for this course. We will be referencing a website called learncpp.com. All needed materials will be provided in class.

Syllabus

Technology Requirements

All testing will be online and will require the use of Respondus LockDown Browser and Monitor; therefore, students are required to have access to a computer and a web camera. Cell phone cameras can be configured to act as web cameras with Monitor. You can find more information on the library's website at

<https://ask.uta.edu/friendly.php?slug=faq/294751>

Syllabus

Grading Information

Homework Policy: Programming is learned by doing – not just by reading about it or listening to someone talk about it. How well would you play a sport or a musical instrument if you only read about how to play or listened to someone lecture about how to play? There will be coding assignments and homework assignments almost every week. You will not be able to pass this class with a C or better unless you do the coding assignments. The homework assignments are to reinforce the in class presentations and will serve as your study guides for the exams.

Since all assignments/code will be submitted via Canvas, you will have 24 hours after the due date/time to submit your work but it will incur a 50% penalty. PLEASE remember that 50 points out of 100 is better than ZERO.

Syllabus

Please note – Coding Assignments that do not compile or compile with ANY warnings on the class's VM will be assigned a grade of 0 automatically. No partial credit will be given for code that does not cleanly compile. Code must run in order to be tested/graded. A penalty of 10% will be applied for each day a Coding Assignment is turned in late.

While I do encourage students to work together on understanding Coding Assignments, I expect every student to do their own work and turn in their own code. Coding Assignments are checked for similarity – any student's code that is determined to be too similar to another student's code submission will be assigned a 0 for the first offense and will be referred to the Office of Student Conduct for any subsequent incidents. This policy will be applied to all students involved – does not matter if you are copying someone else or allowing someone else to copy you.

Any assignment from this class that is found posted on the web in any format will be voided for all sections and a new coding assignment will be assigned to all students unless the student who attempted to cheat via the web is identified.

Syllabus

Grading Policy: Letter grades will be assigned as follows: 90-100 = A, 80-89 = B, 70-79 = C, 60-69 = D, 0-59 = F.

Final Exam	10%
Homework/Crash Course	10%
Code	10%
Online quizzes	70%

No make-up exams will be given except for extenuating circumstances beyond the student's control (in the instructor's opinion). Poor planning or forgetfulness on your part won't be considered an emergency.

Departmental Final: Due to the current circumstances, there will not be a Departmental Final in Fall 2020.

Syllabus

Attendance: At The University of Texas at Arlington, taking attendance is not required but attendance is a critical indicator in student success. As the instructor of this section, I will not specifically take attendance, but a significant portion of your final grade will be earned in class; therefore, if you do not attend regularly, your grade will suffer. If you miss an Online Quiz (OLQ) for ANY reason other than a university authorized absence, you will automatically receive a 0 for that quiz. There will be no makeup quizzes unless you were absent due to a university authorized event and can provide documentation of that excused absence prior to missing class and the OLQ.

Syllabus

Important Dates

Wednesday, August 26 th	First day of class
Monday, September 7 th	Labor Day Holiday
Friday, September 11 th	Census date
Friday, November 6 th	Last day to drop classes
November 25 th – 27 th	No classes scheduled – Thanksgiving holidays Students will not return to campus after Thanksgiving holiday – Instruction will be completely online until the last day of classes
Tuesday, December 8 th	Last day of classes at UTA

Canvas and Email

- I use Canvas for everything.
 - Any material presented in class will be posted on Canvas
 - Slides
 - Code examples
 - Extra info like bash commands and Ubuntu alternate instructions
- I use email A LOT
 - I expect that you will check your UTA/Canvas email at least once per day
 - If you send me an email, you can expect to hear back from me within 24 hours or less unless I have announced that I will specifically be out of touch.

Homework

- Homework will be assigned on Mondays at noon and will be due the following Monday by midnight.
- Homework will usually consist of questions covering the previous week's lectures.
- Any homework submitted within 24 hours after the due date will incur a 50% penalty but 50% is better than 0%.
- Homework questions are taken directly from the slides, class discussion and coding assignments.
- Homework will appear in Canvas as “Quizzes” in order to make use of Canvas’s ability to automatically grade that type of submission.
- Completing and understanding the homework will prepare you for the exams.
- Homework is 8% of your grade.

Crash Course

- Crash Course video quizzes will be assigned on Mondays at noon and will be due the following Monday by noon.
- Any quizzes submitted within 24 hours after the due date will incur a 50% penalty but 50% is better than 0%.
- Quiz questions are taken directly from the videos. I highly suggest you turn on Closed Captioning to get exact phrasing and spelling.
- Crash Course quizzes will appear in Canvas as “Quizzess” in order to make use of Canva’s ability to automatically grade that type of submission.
- The purpose of the quiz is to verify that you have watched the video – not to test how much you already know; therefore, alternate terms will not be accepted even though they may be correct.
- This should be an easy 100% every time.

Coding Assignments

- Coding Assignments will be assigned on Mondays at noon and will be due the following Monday by noon.
- Some of the assignments will build on the previous week's assignment. They will start out easy and will get progressively harder.
- Turn in early and ask me to review. Email me what you have and ask questions. Don't get stuck.
- Coding Assignments will be graded by your assigned GA. They will use the provided rubric to grade your programs. Review the rubric yourself before submitting your final version. **Code that does not compile or compiles with warnings in the CSE 1325 VM will automatically receive a grade of 0.**
- Google is not your friend
 - Ask Google – find a thousand places to look
 - Ask your professor – find the right place to look

OnLine Quizzes

Quizzes given in class will make up 70% of your overall grade.

These quizzes will test your ability to read and write code.

OLQs will cover material from class and coding assignments.

The best way to study for the OLQs is to do the Coding Assignments and **understand** them.

Final Exam

The Final Exam is 10% of your overall grade.

The Final Exam will be the same type of 2.5 hour exam that would be given in person.

The 2.5 hour exam will be broken up into 5 thirty minute quizzes called FEQs (Final Exam Quizzes).

Each FEQ will represent a portion of what would have been the Final Exam in person.

Code Formatting

Formatting will count as 10% of the grade for any code you write in this class – Coding Assignments or OLQs.

Indentation and alignment

Code blocks should be indented at least 3 spaces and not more than 5 spaces

If tabs are used, always use tabs and set tab size to be 3-5 spaces

If spaces are used, always use spaces and always use the same number of them

Curly braces { } should align vertically and be on their own line

```
A
{
    B;
    C
    {
        D;
    }
}
```

Code Formatting

Code formatting has several benefits

- allows quick readability – it is easier/faster to understand the gross structure of the code without in depth examination
- allows for less reliance on the editor to match up braces and code blocks
- creates readable code that is easier for someone other than the student to read – for example, when the student is asking the instructor or TAs for assistance
- allows for easier grading of code – both the instructor and student benefit – code that is easier to grade is less likely to be marked as incorrect
- gives the students the experience of applying a given formatting standard which they will likely encounter as a professional programmer

Learning C++

- C++ can run differently depending on what machine you are using
- We will be using a standard setup that everyone will be required to use
- We will be using
 - C++ 11
 - Linux Ubuntu 64 bit

C++ Resources

www.cplusplus.com

is a good resource

Other resources

- Stack Overflow
- O'Reilly books

The screenshot shows the homepage of cplusplus.com. At the top, there's a navigation bar with links for 'Information', 'Tutorials', 'Reference', 'Articles', and 'Forum'. Below the navigation is a large banner with the text 'Welcome to cplusplus.com' and a copyright notice '© The C++ Resources Network, 2016'. The page is divided into several sections:

- Information:** Describes general information about the C++ programming language, including non-technical documents and descriptions. It includes a list of bullet points: 'Description of the C++ language', 'History of the C++ language', and 'F.A.Q., Frequently Asked Questions'. There's also a large blue 'i' icon.
- Tutorials:** Describes the C++ language from its basics up to its most advanced features. It includes a list of bullet points: 'C++ Language', 'Collection of tutorials covering all the features of this versatile and powerful language. Including detailed explanations of pointers, functions, classes and templates, among others...', and 'more...'. There's also a graphic of four interlocking puzzle pieces.
- Reference:** Describes the most important classes, functions and objects of the Standard Language Library, with descriptive fully-functional short programs as examples. It includes a list of bullet points: 'C library', 'The popular C library, is also part of the of C++ language library.', 'IOStream library', 'The standard C++ library for Input/Output operations.', 'String library', 'Library defining the string class.', 'Standard containers', 'Vectors, lists, maps, sets...', and 'more...'. There's also a graphic of a stack of papers.
- Articles:** User-contributed articles, organized into different categories. It includes a list of bullet points: 'Algorithms', 'Standard library', 'C++11', 'Windows API', and 'Other...'. There's also a graphic of a pencil writing on a notepad.
- Forum:** Message boards where members can exchange knowledge and comments. Ordered by topics. It includes a list of bullet points: 'General C++ Programming', 'Beginners', 'Windows', and 'UNIX/Linux'. There's also a graphic of a speech bubble with a 'fa' logo.
- C++ Search:** A search bar with the placeholder 'Search this website:' and a 'Search' button. It also mentions 'Other tools are also available to search results within this website:' and a link 'more search options'. There's also a graphic of a magnifying glass over a globe.

Website for CSE 1325

www.learncpp.com

The screenshot shows the homepage of LearnCpp.com. The header features a dark blue gradient background with the text "LearnCpp.com" in large, white, sans-serif letters. Below it is a subtitle "Tutorials to help you master C++ and object-oriented programming". The left sidebar has a light gray background. It contains links for "Main Page", "Site Index", "Report an Issue", "About / Contact", and "Support LearnCpp". Below these are sections for "SEARCH" with a search bar and a "Google Search" button. The main content area has a white background. It starts with a paragraph about the site's purpose: "LearnCpp.com is a free website devoted to teaching you how to program in C++. Whether you've had any prior programming experience or not, the tutorials on this site will walk you through all the steps to write, compile, and debug your C++ programs, all with plenty of examples." This is followed by a smaller text block: "Becoming an expert won't happen overnight, but with a little patience, you'll get there. And LearnCpp.com will show you the way." At the bottom of the main content area is a yellow callout box containing the text: "Having trouble remembering where you saw something? Not sure where to find something? Use our **site index** to find what you're looking for!" Below this is a large, empty rectangular search results area. The footer navigation bar at the bottom includes links for "Chapter 0", "Introduction / Getting Started", "0.1 Introduction to these tutorials", "0.2 Introduction to programming languages", "0.3 Introduction to C/C++", "0.4 Introduction to C++ development", "0.5 Introduction to the compiler, linker, and libraries", "0.6 Installing an Integrated Development Environment (IDE)", "0.7 Compiling your first program", "0.8 A few common C++ problems", "0.9 Configuring your compiler: Build configurations", and "0.10 Configuring your compiler: Compiler extensions".

Main Page
Site Index

Report an Issue
About / Contact
Support LearnCpp

SEARCH

Google Search

Chapter 0

Introduction / Getting Started

0.1 Introduction to these tutorials

0.2 Introduction to programming languages

0.3 Introduction to C/C++

0.4 Introduction to C++ development

0.5 Introduction to the compiler, linker, and libraries

0.6 Installing an Integrated Development Environment (IDE)

0.7 Compiling your first program

0.8 A few common C++ problems

0.9 Configuring your compiler: Build configurations

0.10 Configuring your compiler: Compiler extensions

CSE 1325

Week of 08/31/2020

Instructor : Donna French

Academic Integrity and Skills Quiz

- Quiz needs to be taken BEFORE the first OLQ which is next Tuesday.
- The number of points that are scored on the quiz will be added to your first OLQ score – up to 4 bonus points.

The screenshot shows a learning management system interface. At the top, there is a navigation bar with various icons. Below the navigation bar, a section titled "Academic Integrity and Skills Quiz" is displayed. This section includes a dropdown arrow icon and the title "Academic Integrity and Skills Quiz". Underneath this, another section is shown with a green rocket ship icon, the title "Academic Integrity and Skills Quiz - Requires Respondus LockDown Browser + Webcam", and the text "4 pts". A vertical green bar is positioned on the left side of the screen, highlighting the quiz section.

- If you do not complete the Academic Integrity and Skills Quiz, 10 points will be deducted from your first OLQ.

<https://mavsvta.sharepoint.com/sites/cse13xx>

The screenshot shows a SharePoint site titled "cse13xx". The top navigation bar includes links for "SharePoint", "Search this site", and various site settings. The page itself has a green header bar with the title "cse13xx". On the left, there's a navigation menu with links to "Home", "CSE 1310", "CSE 1320", "CSE 1325", "VM Download", "VM Information", "Recycle bin", and "Edit". The main content area features a large heading "Welcome to CSE 1310, CSE 1320 and CSE 1325!". Below it, a paragraph explains that users will find course Home Pages and Lab Schedules/Office Hours. A section titled "Getting Help from the GTAs/TAs" provides information about TA/GTA support hours. Another section discusses the lack of in-person lab times due to circumstances, mentioning Microsoft Teams for communication. A "Departmental Final Exam Information" section states that no departmental final exam will be held. The bottom of the page contains three buttons: "CSE 1310 Home Page", "CSE 1320 Home Page", and "CSE 1325 Home Page".

SharePoint Search this site

cse13xx

Not following Share

Published 8/30/2020 Edit

Home + New Page details

CSE 1310

CSE 1320

CSE 1325

VM Download

VM Information

Recycle bin

Edit

Welcome to CSE 1310, CSE 1320 and CSE 1325!

On this site, you will find links to each course's Home Page and links to the Lab Schedules/Office Hours for each course. The Home Page of each course will list specific information about that course.

Getting Help from the GTAs/TAs

Every course/section has a Graduate Teaching Assistant (GTA) or undergraduate Teaching Assistant (TA) who holds office hours to provide support to the students by helping with coding assignments and questions over material from class. You are welcome to visit the TA/GTA from **any** section of your course if you have general questions, but, if you have more specific assignment questions, please look for your specific GTA in the schedule and visit with them during their working hours.

Due to the current circumstances, no in-person lab times are available - everything is online for Fall 2020. Please download the Teams apps (the web version does not support screenshare) so that you can use screen share and/or chat with the GTAs/TAs. Consult the lab schedule for your course to determine what time a TA/GTA will be on duty for your course. You can then use the Chat feature of Microsoft Teams to contact the on duty TA/GTA (use their name in the Search box in Teams) and you will be able to share your screen with them and discuss your questions/issues.

Departmental Final Exam Information

Due to the current circumstances, there will be no Departmental Final for CSE 1310, CSE 1320 and CSE 1325 for the Fall 2020 Semester. Each instructor will provide details on how your individual section will be graded.

SI Sessions

UTSI has provided Supplemental Instruction for CSE 1310, CSE 1320 and CSE 1325. Click [here](#) for a current list of SI sessions. This link will show you where and when to find the SI sessions. Please be aware that even if the SI is not assigned to your specific section, you are still encouraged to utilize the course's SI resource.

CSE 1310 Home Page

CSE 1320 Home Page

CSE 1325 Home Page

What is bash?

Bash is a Unix shell and command language written by Brian Fox for the GNU Project as a free software replacement for the Bourne shell.

Bash is a command processor that typically runs in a text window where the user types commands that cause actions. This is called the Command Line Interface (CLI). It can be easier for programmers to just type out a command rather than digging through menus. Bash can also read and execute commands from a file, called a shell script.

The shell's name is an acronym for Bourne-again shell, a pun on the name of the Bourne shell that it replaces and on the common term "born again".

Ubuntu

What is it?

- Ubuntu is a free and open source operating system and Linux distribution.
- Ubuntu is produced by Canonical.
- Ubuntu is named after the Southern African philosophy of ubuntu (literally, 'human-ness'), which Canonical suggests can be loosely translated as "humanity to others" or "I am what I am because of who we all are".
- Ubuntu is the most popular operating system for the cloud.



Hello World

In C

```
#include <stdio.h>
int main(void)
{
    printf("Hello World\n");
    return 0;
}
```

In C++

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

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

Hello World

Use your favorite editor (I use Notepad++) to write HelloWorld.cpp. Save to the folder you shared in your VM.



```
C:\Users\Donna\Desktop\UTA\VM\HelloWorld.cpp - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
new 1 new 2 HelloWorld.cpp
1 #include <iostream>
2 using namespace std;
3
4 int main()
5 {
6     cout << "Hello World" << endl;
7     return 0;
8 }
9
```

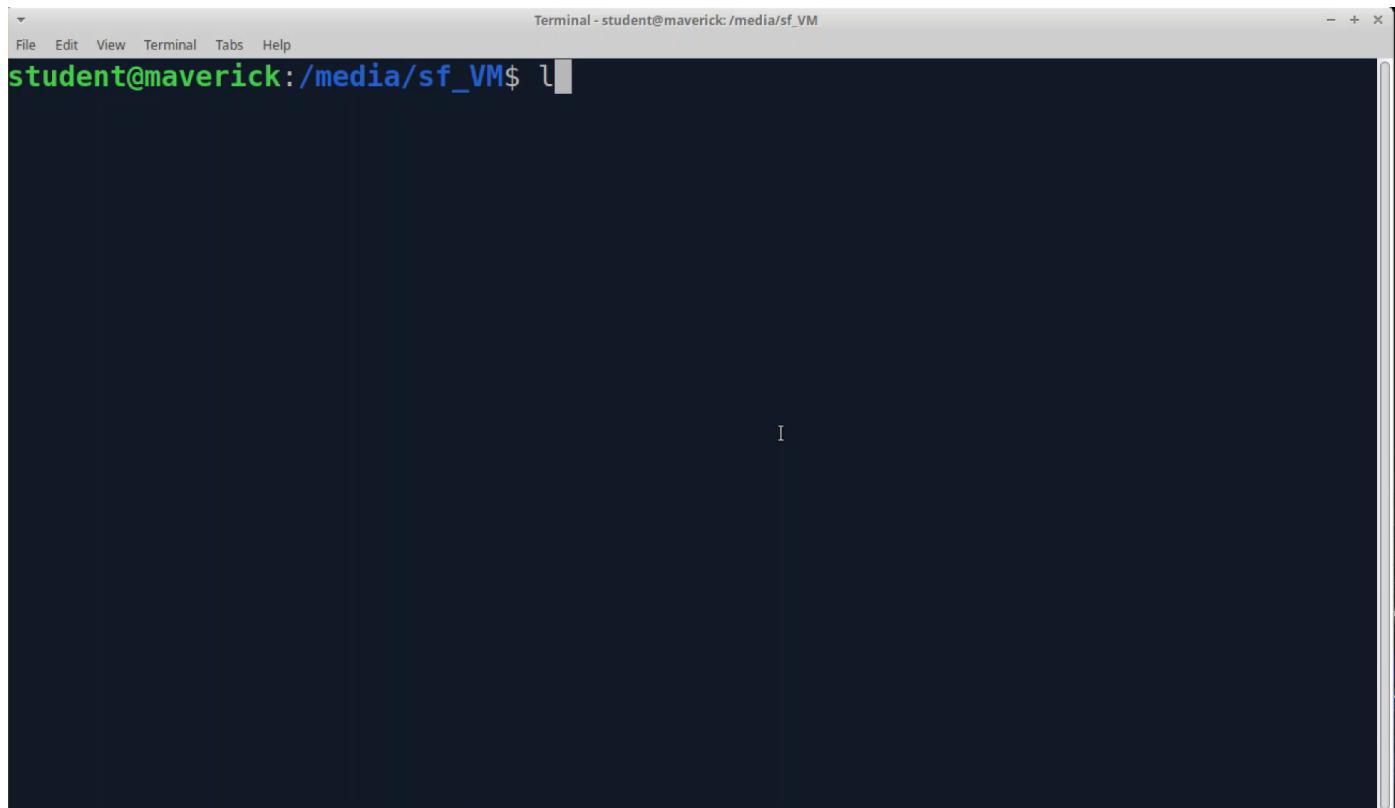
Hello World

You should be able to see it now in your VM when you open your shared folder with the terminal.

```
g++ HelloWorld.cpp
```

Should produce an a.out file.
Run your executable with

```
./a.out
```



A screenshot of a terminal window titled "Terminal - student@maverick:/media/sf_VM". The window has a menu bar with "File", "Edit", "View", "Terminal", "Tabs", and "Help". The main area shows a command-line interface with the prompt "student@maverick:/media/sf_VM\$". A cursor is visible at the end of the line where the user is typing the letter "l".

Hello World

```
#include <iostream>
```

iostream is the header file which contains the functions for formatted input and output including cout, cin, cerr and clog.

C++ standard library packages don't need a .h to reference them.

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

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

Hello World

```
using namespace std
```

The built in C++ library routines are kept in the standard namespace which includes cout, cin, string, vector, map, etc.

Because these tools are used so commonly, it's useful to add "using namespace std" at the top of your source code so that you won't have to type the **std::** prefix constantly.

We use just

```
cout
```

instead of

```
std::cout
```

```
#include <iostream>

using namespace std;

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

Hello World

What is a namespace?

namespace is a language mechanism for grouping declarations. Used to organize classes, functions, data and types.

Namespaces are used to organize code into logical groups and to prevent name collisions that can occur especially when your code base includes multiple libraries.

```
#include <iostream>

using namespace std;

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

I could create a function with the same name and define its own namespace and use the :: scope resolution operator to refer to my version.

We'll get into this more later...

Hello World

```
#include <iostream>  
using namespace std;  
  
int main()  
{  
    cout << "Hello World" << endl;  
    return 0;  
}
```

cout and << and endl

cout is an abbreviation of character output stream.

<< is the output operator

endl puts '\n' into the stream and flushes it

So the line

cout << "Hello World" << endl;

puts the string “Hello World” into the character output stream and flushes it to the screen

Hello World Plus

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

int main()
{
    string first_name;
    cout << "Hello World" << endl;
    cout << "What is your name?" << endl;
    cin >> first_name;
    cout << "Hello " << first_name << endl;
    return 0;
}
```

string is a variable type
that can hold character data

cin is an abbreviation of
character input stream.

>> is the input operator

Hello World Plus

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

int main()
{
    string first_name;
    cout << "Hello World" << endl;
    cout << "What is your name?" << endl;
    cin >> first_name;
    cout << "Hello " << first_name << endl;
    return 0;
}
```

This line
cin >> first_name;
puts whatever you type at the terminal (up to
the first whitespace) into the string variable
first_name
Note that the <ENTER> key (newline) is not
stored in first_name



Hello World Plus

```
student@maverick:/media/sf_VM$ █
```

I



makefile

We are going to start out using a simple makefile.

We will expand on this later as it becomes more necessary.

Please download this template from Canvas and use it with all of your coding assignments.

Course Materials ->

C++ makefile

Add your name and student id as the first line

#FirstName LastName StudentID

```
#makefile for C++ program  
SRC = HelloWorld.cpp  
OBJ = $(SRC:.cpp=.o)  
EXE = $(SRC:.cpp=.e)
```

```
CFLAGS = -g -std=c++11
```

```
all : $(EXE)
```

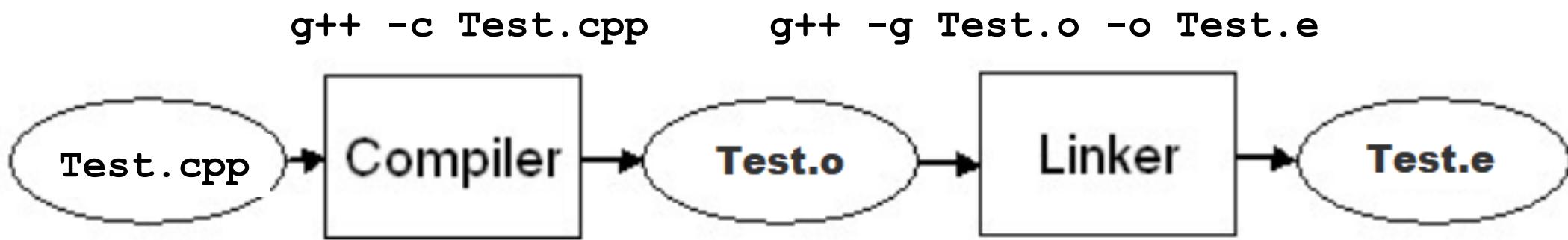
```
$(EXE) : $(OBJ)
```

```
g++ $(CFLAGS) $(OBJ) -o $(EXE)
```

```
$(OBJ) : $(SRC)
```

```
g++ -c $(CFLAGS) $(SRC) -o $(OBJ)
```

makefile



The source file that you type into the editor. This is just a text file, anybody can read.

The object file is an intermediate file. It is only readable by the compiler and the linker.

The executable is the final product. It is a binary file that the operating system can run.

makefile

What is a makefile?

`make` is UNIX utility that is designed to start execution of a makefile.

A makefile is a special file, containing shell commands, that you create and name makefile.

While in the directory containing your makefile, you will type `make` and the commands in the makefile will be executed.

If you create more than one makefile, be certain you are in the correct directory before typing `make`.

makefile

`make` keeps track of the last time files (normally object files) were updated and only updates those files which are required (ones containing changes) to keep the sourcefile up-to-date.

If you have a large program with many source and/or header files, when you change a file on which others depend, you must recompile all the dependent files.

Without a `makefile`, this is an extremely time-consuming task.

makefile

As a makefile is a list of shell commands, it must be written for the shell which will process the makefile. A makefile that works well in one shell may not execute properly in another shell.

The makefile contains a list of rules. These rules tell the system what commands you want to be executed. Most times, these rules are commands to compile(or recompile) a series of files.

The rules, **which must begin in column 1**, are in two parts. The first line is called a dependency line and the subsequent line(s) are system commands or recipes which must be indented with a tab.

makefile

RULE : DEPENDENCIES

[tab] SYSTEM COMMANDS (RECIPE)

A **rule** is usually the name of a file that is generated by a program; examples of rules are executable or object files. A rule can also be the name of an action to carry out, such as "clean". Multiple rules must be separated by a space

A **dependency** (also called *prerequisite*) is a file that is used as input to create the rule. A rule often depends on several files.

The **system command(s)** (also called *recipe*) is an action that make carries out. A recipe may have more than one command, either on the same line or each on its own line. Recipe lines must be indented using a single <tab> character.

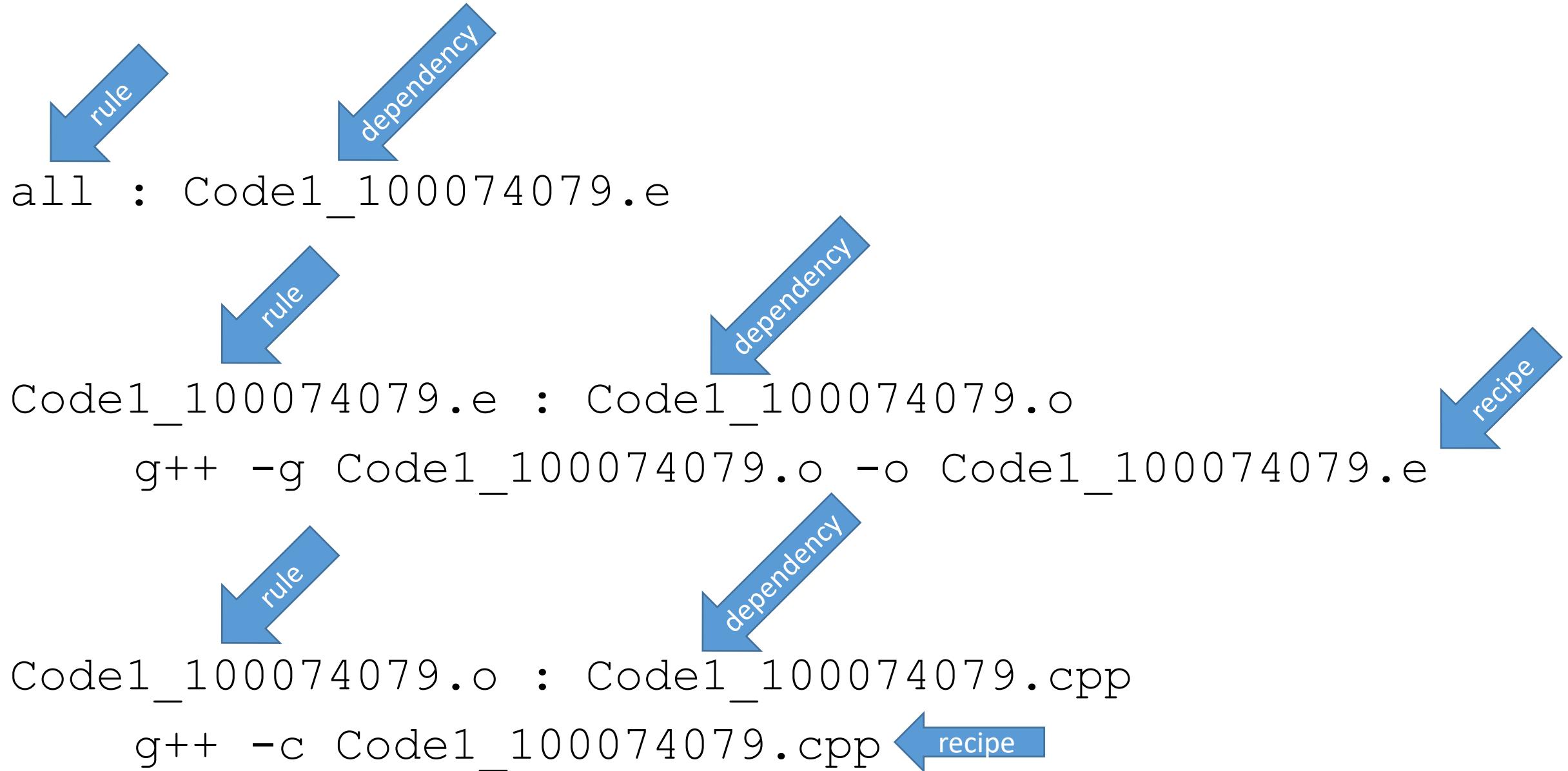
makefile

After the `makefile` has been created, a program can be (re)compiled by typing `make` in the correct directory.

`make` then reads the `makefile` and creates a dependency tree and takes whatever action is necessary. It will not necessarily do all the rules in the `makefile` as all dependencies may not need updated. It will rebuild target files if they are missing or older than the dependency files.

Unless directed otherwise, `make` will stop when it encounters an error during the construction process.

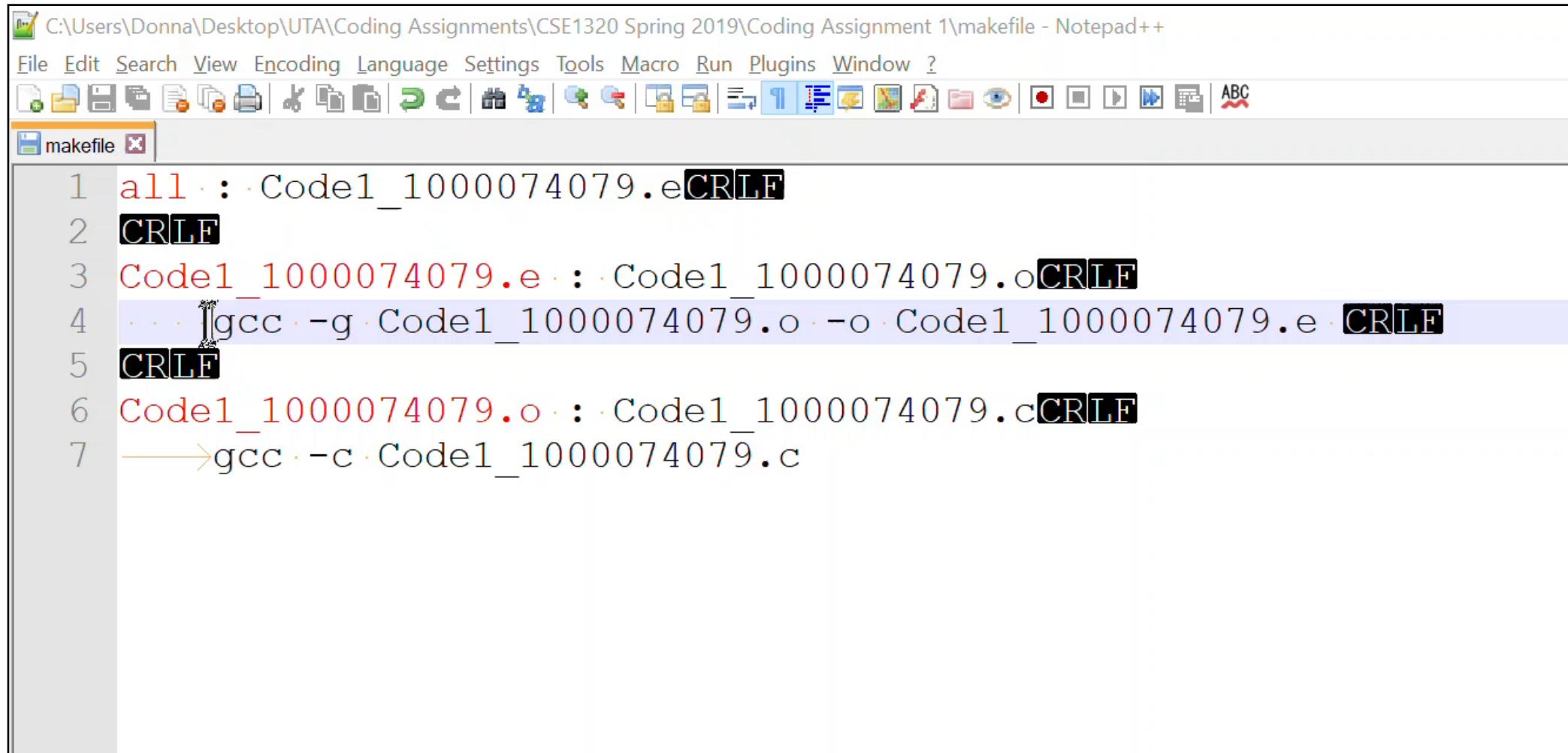
makefile



The screenshot shows a Notepad++ window with the title bar "C:\Users\Donna\Desktop\UTA\Coding Assignments\CSE1320 Spring 2019\Coding Assignment 1\makefile - Notepad++". The menu bar includes File, Edit, Search, View, Encoding, Language, Settings, Tools, Macro, Run, Plugins, Window, and Help. The toolbar contains various icons for file operations like Open, Save, Print, and Find. The status bar at the bottom right shows "ABC". The main editor area has a tab labeled "makefile" with an "X" button. The code is a makefile with the following content:

```
1 all : Code1_1000074079.e
2
3 Code1_1000074079.e : Code1_1000074079.o
4     gcc -g Code1_1000074079.o -o Code1_1000074079.e
5
6 Code1_1000074079.o : Code1_1000074079.c
7     gcc -c Code1_1000074079.c
```

```
[frenchdm@omega CA1]$ make  
makefile:4: *** missing separator. Stop.  
[frenchdm@omega CA1]$
```



C:\Users\Donna\Desktop\UTA\Coding Assignments\CSE1320 Spring 2019\Coding Assignment 1\makefile - Notepad++

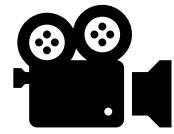
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?

makefile

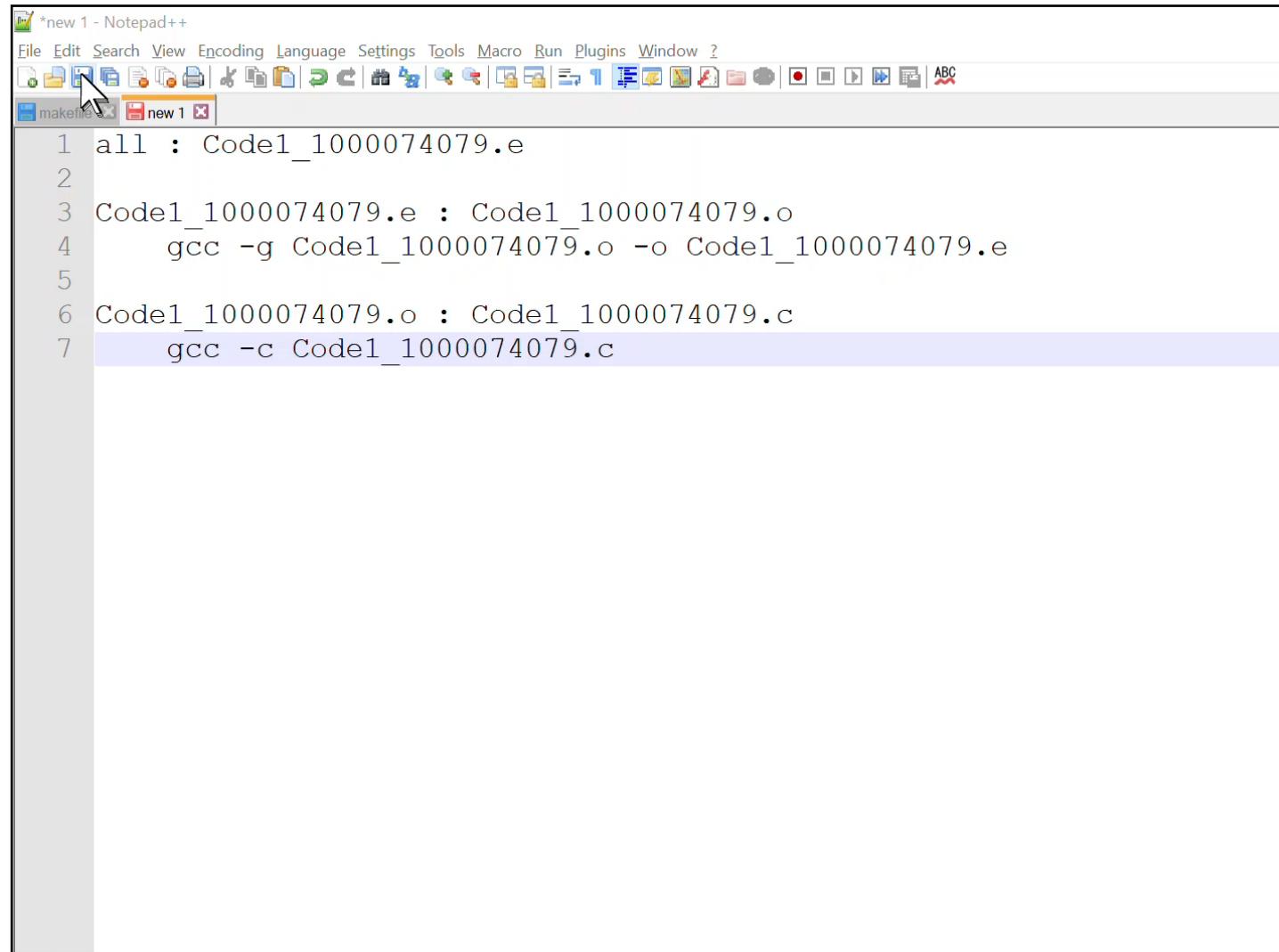
```
1 all : .Code1_1000074079.e CRLF  
2 CRLF  
3 Code1_1000074079.e : .Code1_1000074079.o CRLF  
4 ... [gcc -g .Code1_1000074079.o -o .Code1_1000074079.e] CRLF  
5 CRLF  
6 Code1_1000074079.o : .Code1_1000074079.c CRLF  
7 → gcc -c .Code1_1000074079.c
```

The name of the makefile **MUST BE**

makefile



Save in Notepad++ with a dot on the end to force Notepad++ to not add an extension.



The screenshot shows the Notepad++ interface with a single tab open named "makefile". The tab bar also lists "new 1" and "new 1". The main window displays the following makefile content:

```
1 all : Code1_1000074079.e
2
3 Code1_1000074079.e : Code1_1000074079.o
4     gcc -g Code1_1000074079.o -o Code1_1000074079.e
5
6 Code1_1000074079.o : Code1_1000074079.c
7     gcc -c Code1_1000074079.c
```

A mouse cursor is visible over the toolbar, specifically near the save icon. The code editor uses syntax highlighting to distinguish between different file types and commands.

```
[frenchdm@omega CA1]$ ls  
Code1_1000074079.cpp  makefile.txt  
[frenchdm@omega CA1]$ make  
make: *** No targets specified and no makefile found. Stop.
```

```
[frenchdm@omega CA1]$ mv makefile.txt makefile.mak  
[frenchdm@omega CA1]$ ls  
Code1_1000074079.cpp  makefile.mak
```

```
[frenchdm@omega CA1]$ make  
make: *** No targets specified and no makefile found. Stop.
```

```
[frenchdm@omega CA1]$ mv makefile.mak makefile  
[frenchdm@omega CA1]$ make  
g++ -c Code1_1000074079.cpp  
g++ -g Code1_1000074079.o -o Code1_1000074079.e  
[frenchdm@omega CA1]$
```

makefile

```
all : HelloWorld.e  
  
HelloWorld.e : HelloWorld.o  
    g++ -g -std=c++11 HelloWorld.o -o HelloWorld.e  
  
HelloWorld.o : HelloWorld.cpp  
    g++ -c -g -std=c++11 HelloWorld.cpp -o HelloWorld.o
```

With this explicit makefile, calling just “make” causes execution to start at rule all

Calling “make HelloWorld.e” causes execution to start at rule HelloWorld.e

Calling “make HelloWorld.o” causes execution to start at rule HelloWorld.o

```
student@cse1325:/media/sf_VM$ more makefile
all : HelloWorld.e

HelloWorld.e : HelloWorld.o
        g++ -g -std=c++11 HelloWorld.o -o HelloWorld.e

HelloWorld.o : HelloWorld.cpp
        g++ -c -g -std=c++11 HelloWorld.cpp -o HelloWorld.o

student@cse1325:/media/sf_VM$ make HelloWorld.o
g++ -c -g -std=c++11 HelloWorld.cpp -o HelloWorld.o
student@cse1325:/media/sf_VM$ ls
HelloWorld.cpp  HelloWorld.o  makefile
student@cse1325:/media/sf_VM$ make HelloWorld.e
g++ -g -std=c++11 HelloWorld.o -o HelloWorld.e
student@cse1325:/media/sf_VM$ ls
HelloWorld.cpp  HelloWorld.e  HelloWorld.o  makefile
```

makefile

```
SRC = Code1_100074079.cpp
```

```
OBJ = $(SRC:.cpp=.o)
```

```
EXE = $(SRC:.cpp=.e)
```

```
CFLAGS = -g -std=c++11
```

```
all : $ (EXE)
```

```
$ (EXE) : $ (OBJ)
```

```
    g++ $ (CFLAGS) $ (OBJ) -o $ (EXE)
```

```
$ (OBJ) : $ (SRC)
```

```
    g++ -c $ (SRC)
```

```
all : Code1_100074079.e
```

```
Code1_100074079.e : Code1_100074079.o  
    g++ -g Code1_100074079.o -o
```

```
Code1_100074079.e
```

```
Code1_100074079.o : Code1_100074079.cpp  
    g++ -c Code1_100074079.cpp
```

makefile

```
SRC = Test.cpp
```

```
OBJ = Test.o
```

```
EXE = Test.e
```

```
CFLAGS = -g -std=c++11
```

```
all : Test.e
```

Test.e **Test.o**

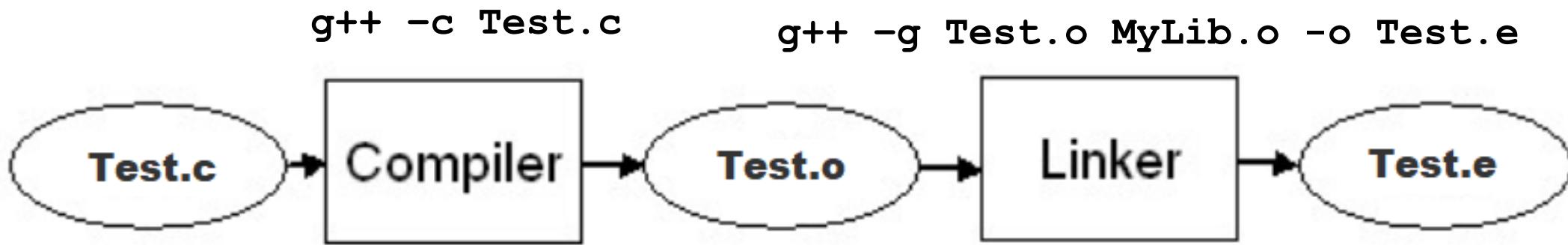
```
g++ -g -std=c++11 Test.o -o Test.e
```

Test.o : **Test.cpp**

```
g++ -c Test.cpp
```

You **DO NOT** do
these
substitutions
yourself – you let
make do its job.

makefile



The source file that you type into the editor. This is just a text file, anybody can read.

The object file is an intermediate file. It is only readable by the compiler and the linker.

The executable is the final product. It is a binary file that the operating system can run.

```
SRC1 = Code1_100074079.cpp
SRC2 = MyLib.cpp
OBJ1 = $(SRC1:.cpp=.o)
OBJ2 = $(SRC2:.cpp=.o)
EXE = $(SRC1:.cpp=.e)
```

```
CFLAGS = -g -std=c++11
```

```
all : $(EXE)
```

```
$(EXE) : $(OBJ1) $(OBJ2)
```

```
        g++ $(CFLAGS) $(OBJ1) $(OBJ2) -o $(EXE)
```

```
$(OBJ1) : $(SRC1)
```

```
        g++ -c $(CFLAGS) $(SRC1) -o $(OBJ1)
```

```
$(OBJ2) : $(SRC2)
```

```
        g++ -c $(CFLAGS) $(SRC2) -o $(OBJ2)
```

makefile

```
SRC = Code1_100074079.cpp
```

```
OBJ = $(SRC:.cpp=.o)
```

```
EXE = $(SRC:.cpp=.e)
```

```
CFLAGS = -g -std=c++11
```

```
all : $(EXE)
```

```
$(EXE) : $(OBJ)
```

```
        g++ $(CFLAGS) $(OBJ) -o $(EXE)
```

```
$(OBJ) : $(SRC)
```

```
        g++ -c $(CFLAGS) $(SRC) -o $(OBJ)
```

```
#Donna French 1000074079 Coding Assigment 6
SRC1 = Code6_1000074079.cpp
SRC2 = TrickOrTreater.cpp
SRC3 = House.cpp
SRC4 = CandyHouse.cpp
SRC5 = ToothbrushHouse.cpp
OBJ1 = $(SRC1:.cpp=.o)
OBJ2 = $(SRC2:.cpp=.o)
OBJ3 = $(SRC3:.cpp=.o)
OBJ4 = $(SRC4:.cpp=.o)
OBJ5 = $(SRC5:.cpp=.o)
EXE = $(SRC1:.cpp=.e)

CFLAGS = -g -std=c++11 -pthread

all : $(EXE)

$(EXE) : $(OBJ1) $(OBJ2) $(OBJ3) $(OBJ4) $(OBJ5)
        g++ $(CFLAGS) $(OBJ1) $(OBJ2) $(OBJ3) $(OBJ4) $(OBJ5) -o $(EXE)

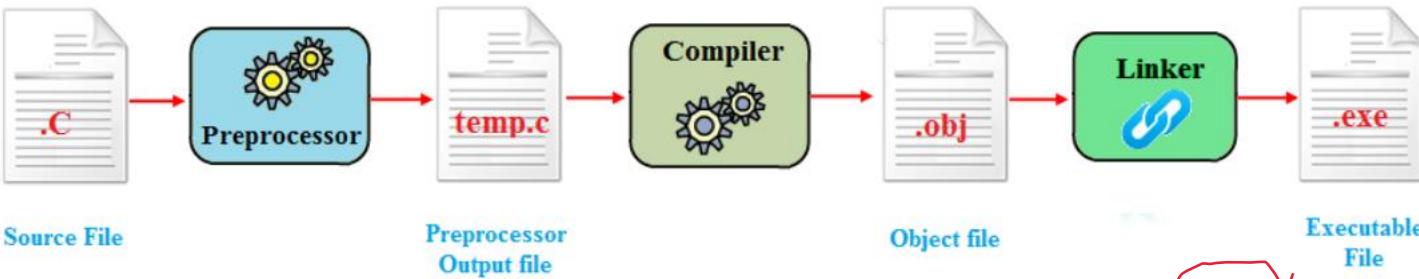
$(OBJ1) : $(SRC1)
        g++ -c $(CFLAGS) $(SRC1) -o $(OBJ1)

$(OBJ2) : $(SRC2)
        g++ -c $(CFLAGS) $(SRC2) -o $(OBJ2)

$(OBJ3) : $(SRC3)
        g++ -c $(CFLAGS) $(SRC3) -o $(OBJ3)

$(OBJ4) : $(SRC4)
        g++ -c $(CFLAGS) $(SRC4) -o $(OBJ4)

$(OBJ5) : $(SRC5)
        g++ -c $(CFLAGS) $(SRC5) -o $(OBJ5)
```



Shell Scripting

compiler

creates an object file

linker

takes in object files and produces an executable file

```

SRC1 = Code2_1000074079.cpp
SRC2 = MyLib.cpp
OBJ1 = $(SRC1:.cpp=.o)
OBJ2 = $(SRC2:.cpp=.o)
EXE = $(SRC1:.cpp=.e)

```

CFLAGS = -g -std=c++11

all : \$ (EXE)

```

$ (EXE) : $ (OBJ1) $ (OBJ2)
g++ $ (CFLAGS) $ (OBJ1) $ (OBJ2) -o $ (EXE)

```

```

$ (OBJ1) : $ (SRC1)
g++ -c $ (CFLAGS) $ (SRC1) -o $ (OBJ1)

$ (OBJ2) : $ (SRC2)
g++ -c $ (CFLAGS) $ (SRC2) -o $ (OBJ2)

```

makefile

***** IMPORTANT INFORMATION *****

After you successfully compile a program, running `make` again will result in

```
student@cse1325:/media/sf_VM$ make  
make: Nothing to be done for 'all'.
```

To force a recompile, use `-B` after `make`

```
student@cse1325:/media/sf_VM$ make -B  
g++ -c -g -std=c++11 HelloWorld.cpp -o HelloWorld.o  
g++ -g -std=c++11 HelloWorld.o -o HelloWorld.e  
student@cse1325:/media/sf_VM$ make  
make: Nothing to be done for 'all'.  
student@cse1325:/media/sf_VM$ make -B  
g++ -c -g -std=c++11 HelloWorld.cpp -o HelloWorld.o  
g++ -g -std=c++11 HelloWorld.o -o HelloWorld.e
```

OLQ1

OLQ1 will be over make and makefile.

1. Given a makefile for a one module program, can you expand it to three modules?
2. Can you explain how to fix the make error
 *** missing separator.
3. Can you identify which parts of a makefile are the compiler and which parts are the linker?

Variables in C++

Familiar variable types from C carry over to
C++

char
short
int
float
double
void
long
unsigned
signed

A new built-in type in C++

bool x

x is a Boolean which can have a
value of true(1) or false(0)

New types defined in the standard library

string xxxx

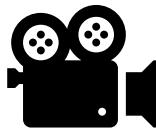
xxxx is stream of characters

These are built-in types



student@cse1325: /media/sf_VM

- + ×



File Edit Tabs Help

```
student@cse1325:/media/sf_VM$ more Bool1Demo.cpp
// Bool1Demo

#include <iostream>

using namespace std;

int main()
{
    bool torf;

    cout << "The current value of torf is " << torf << endl;
    cout << "Enter a value for your bool variable ";
    cin >> torf;
    cout << "The new value of torf is " << torf << endl;

    return 0;
}
student@cse1325:/media/sf_VM$ █
```

String Operators in C++

Initialization

```
string MyString4 = "How are you today?";
```

Assignment

```
MyString2 = MyString1;
```

Concatenation

```
MyString2 = MyString1 + MyString3;
```

Comparison

```
MyString1 < MyString3
```

```
MyString1 <= MyString3
```

```
MyString1 > MyString3
```

```
MyString1 >= MyString3
```

```
MyString1 == MyString3
```

```
MyString1 != MyString3
```

Terminal - student@maverick:/media/sf_VM

File Edit View Terminal Tabs Help

student@maverick:/media/sf_VM\$ █

█

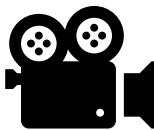
String Operators in C++

```
string MyString1, MyString2, MyString3;  
string MyString4 = "How are you today?";  
  
MyString2 = MyString1;  
  
MyString2 = MyString1 + MyString3;  
  
if (MyString1 < MyString3)  
    cout << MyString1 << " is alphabetically before " << MyString3 << endl;  
else if (MyString1 > MyString3)  
    cout << MyString1 << " is alphabetically after " << MyString3 << endl;  
else if (MyString1 == MyString3)  
    cout << MyString1 << " is alphabetically equal to " << MyString3 << endl;
```



student@cse1325: /media/sf_VM

- + ×



File Edit Tabs Help

```
student@cse1325:/media/sf_VM$ more String1Demo.cpp
#include <iostream>

using namespace std;

int main()
{
    string first_name, last_name, full_name;

    cout << "Hello!\n" << endl;
    cout << "What is your name? (Enter your first name and last name) " << e
ndl;
    cin >> first_name >> last_name;
    cout << "Hello " << first_name << ' ' << last_name << endl;

    return 0;
}
student@cse1325:/media/sf_VM$ █
```

cin

```
cin >> CreamPuff;
```



cout

```
cout << "Happy Birthday";
```

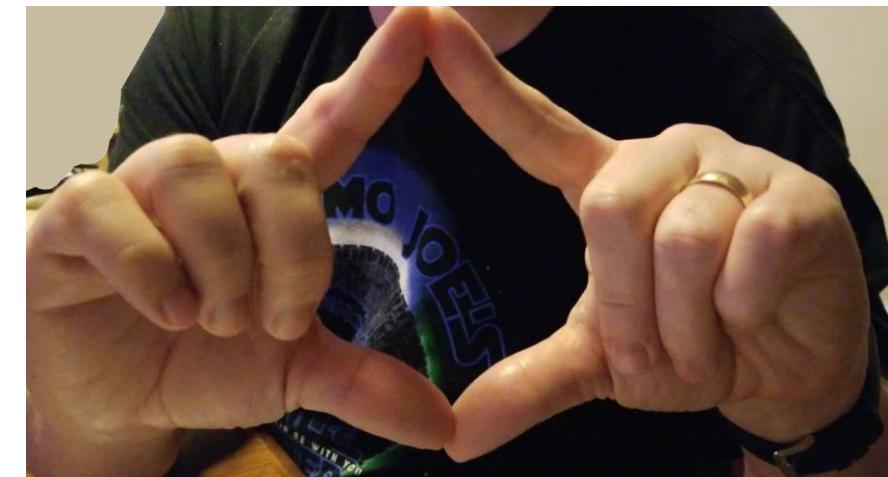


stream insertion vs stream extraction

```
<<  
    stream insertion operator  
>>  
    stream extraction operator
```

Remember the rule in English of "i before e except after c"?

i before e
insertion extraction
 << >>



Uniform Initialization

Unsafe Conversions

C++ allows for (implicit) unsafe conversions.

unsafe = a value can be implicitly turned into a value of another type that does not equal the original value

```
int IntVar1 = 32112;  
char CharVarA = IntVar1;  
int IntVar2 = CharVarA;
```



Uniform Initialization

Unsafe Conversions

To be warned against these unsafe conversions, use the uniform initialization format

```
int IntVar1 {32112};  
char CharVarA {IntVar1};  
int IntVar2 {CharVarA};
```

Terminal - student@maverick: /media/sf_VM

File Edit View Terminal Tabs Help

student@maverick:/media/sf_VM\$





Uniform Initialization

Additional Notes about Uniform Initialization

Type `bool` can be initialized with UI

```
bool b1 {true};  
bool b2 {false};  
bool b3 {!true};  
bool b4 {!false};
```

A function can be called that returns a value inside the {}

Use empty braces {} to initialize a variable to 0.



student@cse1325: /media/sf_VM

- + ×



File Edit Tabs Help

```
student@cse1325:/media/sf_VM$ more uui3Demo.cpp
#include <iostream>

using namespace std;

int getValueFromUser()
{
    cout << "Enter an integer: ";
    int input{};
    cin >> input;

    return input;
}

int main()
{
    int num {getValueFromUser()};

    cout << num << " doubled is: " << num * 2 << '\n';

    return 0;
}
student@cse1325:/media/sf_VM$ █
```

DRY vs WET Coding

DRY

Don't Repeat Yourself

Advantages

Maintainability

Readability

Reuse

Cost

Testing

WET

**Write Everything Twice
We Enjoy Typing**

Advantages

NONE



Abstraction

In order to use a function, you only need to know its name, inputs, outputs, and where it lives.

You don't need to know how it works, or what other code it's dependent upon to use it.

This lowers the amount of knowledge required to use other people's code (including everything in the standard library).

Required Formatting of Code

The opening brace for a function should be given its own line and the closing brace should line up with the opening brace. Any code lines within the braces should be indented the same amount which must be between 3 and 5 spaces.

```
int main()
{
    my first line
    my second line
    my third line
}
```

`std::string`

Just like the Java and C, a string is a collection of sequential characters.

C++ has a `string` type. Just like the Java, `string` is actually an object; therefore, knows things and can do things. This will make more sense once we start talking about classes and member functions.

To use `string` include the `string` header file.

```
#include <string>
```

`std::string`

As we did with `cin` and `cout`, we can either put

```
using namespace std
```

in our .cpp file and not need to preface `string` with `std::` or we can not use the `std` namespace and need to use `std::string`.

```
std::string MyString;  
string MyString;
```

std::string

Declaring and initializing a string in one line

```
string MyString ("Silly");
```

constructing

A string can also be declared and then assigned a value

```
string MyString;  
MyString = "Silly";
```

assignment

std::string

We've already seen the example where `cin` stops reading at whitespace (just like `scanf()`).

```
string first_name, last_name, full_name;

cout << "Hello!\n" << endl;
cout << "What is your name? (Enter your first name and last name) " << endl;
cin >> first_name >> last_name;
cout << "Hello " << first_name << ' ' << last_name << endl;
```

std::string

What if we need to read a line of input including the whitespace into a single variable?

For example, what if I wanted to take whatever name was entered and only store it in one variable?

```
string full_name;  
  
cout << "Hello!\n" << endl;  
cout << "What is your name? " << endl;  
cin >> full_name;  
cout << "Hello " << full_name << endl;
```

If I type

Fred Flintstone

at the prompt, what will print?

std::string

`getline()` is the C++ version of `fgets()` from C. It takes two parameters just like `fgets()`.

The first parameter is the stream to read from – when reading from the screen use `cin`.

The second parameter is `string` variable where you want to store the input.

```
string full_name;  
  
cout << "Hello!\n" << endl;  
cout << "What is your name? " << endl;  
getline(cin, full_name);  
cout << "Hello " << full_name << endl;
```

Hello!

What is your name?

Fred Flintstone

Hello Fred Flintstone

std::string

Mixing `cin` with `getline()` can cause issues

`cin` leaves the newline (`\n`) in the standard input buffer.

```
10      cin >> dog_name;
```

```
(gdb)
```

What is your dog's name? Dino

```
11      cout << "Hi " << dog_name << endl;
```

```
(gdb) p *stdin
```

```
$1 = {_flags = -72539512, _IO_read_ptr = 0x555555769284, "\n", }
```

std::string

Which getline () then reads and uses; therefore, not prompting for more input.

We can use

```
cin.ignore(50, '\n');
```

This function discards the specified number of characters or fewer characters if the delimiter is encountered in the input stream.

Puts a null at the end of the buffer and throws out the newline

New keywords in C++

const

Used to inform the compiler that the value of a particular variable should not be modified.

If a value does not (or should not) change in the body of a function to which it's passed, the parameter should be declared const.

```
const int counter = 1;
```

counter is an integer constant

New keywords in C++

const

const variables must be initialized when you define them and then that value can not be changed via assignment.

const variables can be initialized from other variables (including non-const ones).

We will use const with function parameters when we learn about passing by value in C++.

```
#include <iostream>          #include <iostream>
using namespace std;          using namespace std;
int main()                  int main()
{                           {
    int x;                 const int x;
                           x = 1;
    x = 1;                 return 0;
                           }
    return 0;               }
}

constDemo.cpp: In function 'int main()':
constDemo.cpp:7:12: error: uninitialized const 'x' [-fpermissive]
    const int x;
                           ^
constDemo.cpp:9:4: error: assignment of read-only variable 'x'
    x = 1;
                           ^
makefile:15: recipe for target 'constDemo.o' failed
make: *** [constDemo.o] Error 1
```

```
#include <iostream>

using namespace std;

int main()
{
    const int x = 1;
    x = 1;

    return 0;
}
```

```
constDemo.cpp: In function 'int main()':
constDemo.cpp:9:4: error: assignment of read-only variable 'x'
    x = 1;
               ^
makefile:15: recipe for target 'constDemo.o' failed
make: *** [constDemo.o] Error 1
```

CSE 1325

Week of 09/07/2020

Instructor : Donna French

Uniform Initialization

Unsafe Conversions

C++ allows for (implicit) unsafe conversions.

unsafe = a value can be implicitly turned into a value of another type that does not equal the original value

```
int IntVar1 = 32112;  
char CharVarA = IntVar1;  
int IntVar2 = CharVarA;
```



Uniform Initialization

Unsafe Conversions

To be warned against these unsafe conversions, use the uniform initialization format

```
int IntVar1 {32112};  
char CharVarA {IntVar1};  
int IntVar2 {CharVarA};
```

Terminal - student@maverick: /media/sf_VM

File Edit View Terminal Tabs Help

student@maverick:/media/sf_VM\$





Uniform Initialization

Additional Notes about Uniform Initialization

Type `bool` can be initialized with UI

```
bool b1 {true};  
bool b2 {false};  
bool b3 {!true};  
bool b4 {!false};
```

A function can be called that returns a value inside the {}

Use empty braces {} to initialize a variable to 0.



student@cse1325: /media/sf_VM

- + ×



File Edit Tabs Help

```
student@cse1325:/media/sf_VM$ more uui3Demo.cpp
#include <iostream>

using namespace std;

int getValueFromUser()
{
    cout << "Enter an integer: ";
    int input{};
    cin >> input;

    return input;
}

int main()
{
    int num {getValueFromUser()};

    cout << num << " doubled is: " << num * 2 << '\n';

    return 0;
}
student@cse1325:/media/sf_VM$ █
```

DRY vs WET Coding

DRY

Don't Repeat Yourself

Advantages

Maintainability

Readability

Reuse

Cost

Testing

WET

**Write Everything Twice
We Enjoy Typing**

Advantages

NONE



Abstraction

In order to use a function, you only need to know its name, inputs, outputs, and where it lives.

You don't need to know how it works, or what other code it's dependent upon to use it.

This lowers the amount of knowledge required to use other people's code (including everything in the standard library).

Required Formatting of Code

The opening brace for a function should be given its own line and the closing brace should line up with the opening brace. Any code lines within the braces should be indented the same amount which must be between 3 and 5 spaces.

```
int main()
{
    my first line
    my second line
    my third line
}
```

`std::string`

Just like the Java and C, a string is a collection of sequential characters.

C++ has a `string` type. Just like the Java, `string` is actually an object; therefore, knows things and can do things. This will make more sense once we start talking about classes and member functions.

To use `string` include the `string` header file.

```
#include <string>
```

`std::string`

As we did with `cin` and `cout`, we can either put

```
using namespace std
```

in our .cpp file and not need to preface `string` with `std::` or we can not use the `std` namespace and need to use `std::string`.

```
std::string MyString;  
string MyString;
```

std::string

Declaring and initializing a string in one line

```
string MyString ("Silly");
```

constructing

A string can also be declared and then assigned a value

```
string MyString;  
MyString = "Silly";
```

assignment

std::string

We've already seen the example where `cin` stops reading at whitespace (just like `scanf()`).

```
string first_name, last_name, full_name;

cout << "Hello!\n" << endl;
cout << "What is your name? (Enter your first name and last name) " << endl;
cin >> first_name >> last_name;
cout << "Hello " << first_name << ' ' << last_name << endl;
```

std::string

What if we need to read a line of input including the whitespace into a single variable?

For example, what if I wanted to take whatever name was entered and only store it in one variable?

```
string full_name;  
  
cout << "Hello!\n" << endl;  
cout << "What is your name? " << endl;  
cin >> full_name;  
cout << "Hello " << full_name << endl;
```

If I type

Fred Flintstone

at the prompt, what will print?

std::string

`getline()` is the C++ version of `fgets()` from C. It takes two parameters just like `fgets()`.

The first parameter is the stream to read from – when reading from the screen use `cin`.

The second parameter is `string` variable where you want to store the input.

```
string full_name;  
  
cout << "Hello!\n" << endl;  
cout << "What is your name? " << endl;  
getline(cin, full_name);  
cout << "Hello " << full_name << endl;
```

Hello!

What is your name?

Fred Flintstone

Hello Fred Flintstone

std::string

Mixing `cin` with `getline()` can cause issues

`cin` leaves the newline (`\n`) in the standard input buffer.

```
10      cin >> dog_name;
```

```
(gdb)
```

What is your dog's name? Dino

```
11      cout << "Hi " << dog_name << endl;
```

```
(gdb) p *stdin
```

```
$1 = {_flags = -72539512, _IO_read_ptr = 0x555555769284, "\n", }
```

std::string

Which getline () then reads and uses; therefore, not prompting for more input.

We can use

```
cin.ignore(50, '\n');
```

This function discards the specified number of characters or fewer characters if the delimiter is encountered in the input stream.

Puts a null at the end of the buffer and throws out the newline

New keywords in C++

const

Used to inform the compiler that the value of a particular variable should not be modified.

If a value does not (or should not) change in the body of a function to which it's passed, the parameter should be declared const.

```
const int counter = 1;
```

counter is an integer constant

New keywords in C++

const

const variables must be initialized when you define them and then that value can not be changed via assignment.

const variables can be initialized from other variables (including non-const ones).

We will use const with function parameters when we learn about passing by value in C++.

```
#include <iostream>          #include <iostream>
using namespace std;          using namespace std;
int main()                  int main()
{                           {
    int x;                 const int x;
    x = 1;                  x = 1;
                           return 0;
}
return 0;                  }
                           }

constDemo.cpp: In function 'int main()':
constDemo.cpp:7:12: error: uninitialized const 'x' [-fpermissive]
    const int x;
               ^
constDemo.cpp:9:4: error: assignment of read-only variable 'x'
    x = 1;
               ^
makefile:15: recipe for target 'constDemo.o' failed
make: *** [constDemo.o] Error 1
```

```
#include <iostream>

using namespace std;

int main()
{
    const int x = 1;
    x = 1;

    return 0;
}
```

```
constDemo.cpp: In function 'int main()':
constDemo.cpp:9:4: error: assignment of read-only variable 'x'
    x = 1;
               ^
makefile:15: recipe for target 'constDemo.o' failed
make: *** [constDemo.o] Error 1
```

Passing Parameters to Functions

Two basic methods of passing parameters to functions

- *pass by value*
 - parameter is called *value parameter*
 - a copy is made of the current value of the parameter
 - operations in the function are done on the copy – the original does not change
- *pass by reference*
 - parameter is called a *variable parameter*
 - the address of the parameter's storage location is known in the function
 - operations in the function are done directly on the parameter

Passing Parameters to Functions

In C

all parameters are passed by value

the ability to pass by reference does not exist

Pass by reference can be simulated

- pass the address of the variable
- address cannot be modified
- contents of address can be modified

```
int main(void)
{
    int MyMainNum = 0;

    printf("Before PassByValue call\tMyMainNum = %d\n", MyMainNum);
    PassByValue(MyMainNum);
    printf("After PassByValue call\tMyMainNum = %d\n", MyMainNum);

    printf("Before PassByRef      call\tMyMainNum = %d\n", MyMainNum);
    PassByRef(&MyMainNum);
    printf("After PassByRef      call\tMyMainNum = %d\n", MyMainNum);

    return 0;
}
```

A copy is passed

```
int PassByValue(int MyNum)
{
    MyNum += 100;
    printf("Inside PassByValue\tMyNum      = %d\n", MyNum);
}
```

The address of the actual variable is passed

```
int PassByRef(int *MyNumPtr)
{
    *MyNumPtr += 100;
    printf("Inside PassByRef\tMyRefNum = %d\n", *MyNumPtr);
}
```

```
int MyMainNum = 0;

printf("Before PassByValue call\tMyMainNum = %d\n", MyMainNum);
PassByValue(MyMainNum);
printf("After PassByValue call\tMyMainNum = %d\n", MyMainNum);

int PassByValue(int MyNum)
{
    MyNum += 100;
    printf("Inside PassByValue\tMyNum      = %d\n", MyNum);
}
```

Before PassByValue call MyMainNum = 0
Inside PassByValue MyNum = 100
After PassByValue call MyMainNum = 0

```
int MyMainNum = 0;

printf("Before PassByRef      call\tMyMainNum = %d\n", MyMainNum);
PassByRef(&MyMainNum);
printf("After  PassByRef      call\tMyMainNum = %d\n", MyMainNum);
```

```
int PassByRef(int *MyNumPtr)
{
    *MyNumPtr += 100;
    printf("Inside PassByRef\tMyNumPtr = %d\n", *MyNumPtr);
}
```

```
Before PassByRef      call MyMainNum = 0
Inside PassByRef       MyRefNum   = 100
After  PassByRef      call MyMainNum = 100
```

Pass by Reference in C++

C++ has a specific syntax for passing by reference.

To indicate that a function parameter is passed by reference, follow the parameter's type in the function prototype by an ampersand (&); use the same convention when listing the parameter's type in the function header.

int& number

number is a reference to an int

```
int main(void)
{
    int MyMainNum = 0;

    cout << "Before PassByRefCPlusPlus      call\tMyMainNum = " << MyMainNum << endl;
    PassByRefCPlusPlus(MyMainNum);
    cout << "After  PassByRefCPlusPlus      call\tMyMainNum = " << MyMainNum << endl;

    return 0;
}

int PassByRefCPlusPlus(int & MyNum)
{
    MyNum += 100;
    cout << "Inside PassByRefCPlusPlus\t\tMyNum      = " << MyNum << endl;
}
```

What happens if we remove the &?

Pass By Reference

Pros vs Cons

One disadvantage of pass-by-value is that, if a large data item is being passed, copying that data can take a considerable amount of execution time and memory space.

Pass-by-reference is good for performance reasons, because it can eliminate the pass-by-value overhead of copying large amounts of data.

Pass-by-reference can weaken security; the called function can corrupt the caller's data.

const References

Function setName uses pass-by-value.

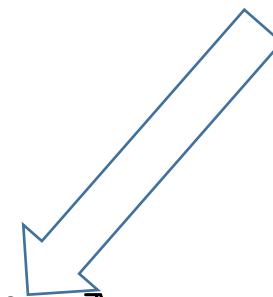
```
void setName(std::string AccountName)
{
    string name = AccountName;
}
```

When this function is called, it receives a copy of its string argument. string objects can be large, so this copy operation degrades an application's performance.

const References

For this reason, string objects (and objects in general) should be passed to functions by reference.

```
void setName(std::string& AccountName)  
{  
    name = AccountName;  
}
```



const References

But, this means that the function can change/corrupt the data.

To specify that a reference parameter should not be allowed to modify the corresponding argument, place the `const` qualifier before the type name in the parameter's declaration.

```
void setName(const std::string& AccountName)
{
    name = AccountName;
}
```

We get the performance of passing the string by reference, but `setName` treats the argument as a constant, so it cannot modify the value in the caller—just like with pass-by-value. Code that calls `setName` would still pass the string argument exactly as before.

namespace

What is a namespace?

A namespace defines an area of code in which all identifiers are guaranteed to be unique.

Namespaces are used to help avoid issues where two independent pieces of code have naming collisions with each other when used together.

namespace

Name Collision Example

If you were told to go to Room 129 BUT not told which building on campus, how would you know which building to choose for Room 129?

Being told NH 129 or ERB 129 makes a big difference in where you end up.

namespace

ERB.h

```
void PrintLocation(int RoomNumber)
{
    std::cout << "ERB " << RoomNumber;
}
```

NH.h

```
void PrintLocation(int RoomNumber)
{
    std::cout << "NH " << RoomNumber;
}
```

namespace

```
1 // namespace demo using ERB and NH
2
3 #include <iostream>
4
5 #include "ERB.h"
6 #include "NH.h"
7
8 int main()
9 {
10     int RoomNumber{ } ;
11
12     std::cout << "Enter Room Number " ;
13     std::cin >> RoomNumber ;
14
15     PrintLocation(RoomNumber) ;
16
17     return 0 ;
18 }
```



g++ xxxx.cpp -E

Terminal - student@maverick:/media/sf_VM/CSE1325

File Edit View Terminal Tabs Help

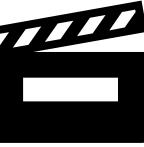
student@maverick:/media/sf_VM/CSE1325\$ █

I

Terminal - student@maverick:/media/sf_VM/CSE1325

File Edit View Terminal Tabs Help

student@maverick:/media/sf_VM/CSE1325\$ █



namespace

ERB.h

```
namespace ERB
{
    void PrintLocation(int RoomNumber)
    {
        cout << "ERB " << RoomNumber;
    }
}
```

NH.h

```
namespace NH
{
    void PrintLocation(int RoomNumber)
    {
        cout << "NH " << RoomNumber;
    }
}
```

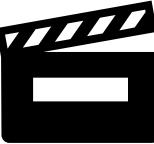
File Edit View Terminal Tabs Help

```
student@maverick:/media/sf_VM/CSE1325$ make
g++ -c -g -std=c++11 namespaceERBNHDemo.cpp -o namespaceERBNHDemo.o
namespaceERBNHDemo.cpp: In function 'int main()':
namespaceERBNHDemo.cpp:15:2: error: 'PrintLocation' was not declared in this scope
15 |   PrintLocation(RoomNumber);
      ^~~~~~
namespaceERBNHDemo.cpp:15:2: note: suggested alternatives:
In file included from namespaceERBNHDemo.cpp:5:
ERB.h:3:7: note:   'ERB::PrintLocation'
 3 | void PrintLocation(int RoomNumber)
    | ^~~~~~
In file included from namespaceERBNHDemo.cpp:6:
NH.h:3:7: note:   'NH::PrintLocation'
 3 | void PrintLocation(int RoomNumber)
    | ^~~~~~
make: *** [makefile:14: namespaceERBNHDemo.o] Error 1
student@maverick:/media/sf_VM/CSE1325@ ]
```

Terminal - student@maverick:/media/sf_VM/CSE1325

File Edit View Terminal Tabs Help

student@maverick:/media/sf_VM/CSE1325\$ m



Scope Resolution Operator

You can tell the compiler to look at a particular namespace by using the scope resolution operator

So what is

::

std::cout

with the name of the namespace.

actually doing?

```
ERB::PrintLocation(RoomNumber);  
NH::PrintLocation(RoomNumber);
```

Scope Resolution Operator

```
3 #include <iostream>
4
5 using namespace std;
6
7 #include "ERB.h"
8 #include "NH.h"
9 using namespace ERB;
10 int main()
11 {
12     int RoomNumber{};
13
14     cout << "Enter Room Number ";
15     cin >> RoomNumber;
16
17     PrintLocation(RoomNumber);
18     NH::PrintLocation(RoomNumber);
19
20     return 0;
21 }
```

Explicit Type Conversion – Casting

C vs C++

Explicit cast in C

```
int i1 = 10;  
int i2 = 4;  
float f = (float)i1 / i2;
```

C-style explicit cast in C++ syntax

```
int i1 = 10;  
int i2 = 4;  
float f = float(i1) / i2;
```

Because these types of casts are not checked by the compiler, they can be misused.

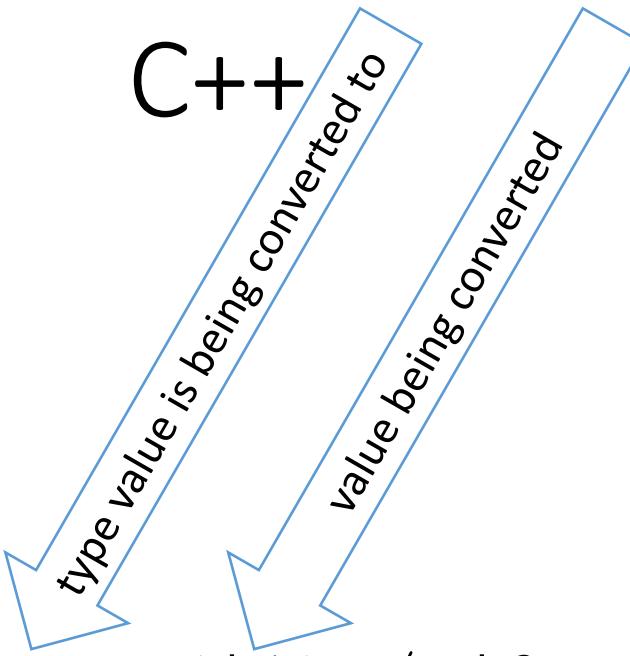
C++ introduced compile-time type checking; therefore, making type casting safer.

Explicit Type Conversion – Casting

C++

static_cast

```
int i1 = 10;  
int i2 = 4;  
float f = static_cast<float>(i1) / i2;
```



static_cast takes a single value as input and outputs the same value converted to the type specified inside the angled brackets.

static_cast is best used to convert one fundamental type into another.

Enumerated Types

One of the simplest user-defined data type is the enumerated type.

An **enumerated type** (also called an **enumeration** or **enum**) is a data type where every possible value is defined as a symbolic constant (called an **enumerator**).

Enumerations are defined via the **enum** keyword.

Enumerated Types

```
enum Color
{
    COLOR_BLACK,
    COLOR_RED,
    COLOR_BLUE,
    COLOR_GREEN,
    COLOR_WHITE,
    COLOR_CYAN,
    COLOR_YELLOW,
    COLOR_MAGENTA
};
```

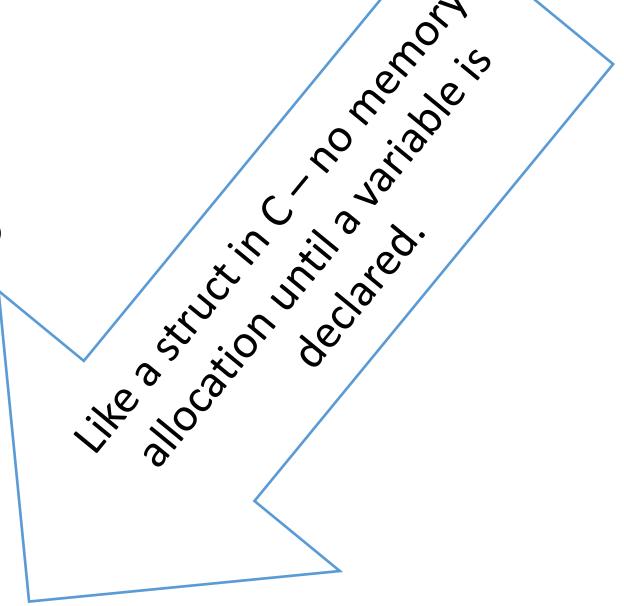
```
Color Banana = COLOR_YELLOW;
```

```
Color Blueberry{COLOR_BLUE} ;
```

```
Color Celery(COLOR_GREEN) ;
```

Providing a name for an enumeration is optional.

Enumerated Types



Like a struct in C - no memory allocation until a variable is declared.

Defining an enumeration does not allocate any memory. When a variable of the enumerated type is defined, memory is allocated for that variable at that time.

Note that each enumerator is separated by a comma and the entire enumeration ends with a semicolon.

Enumerated Types

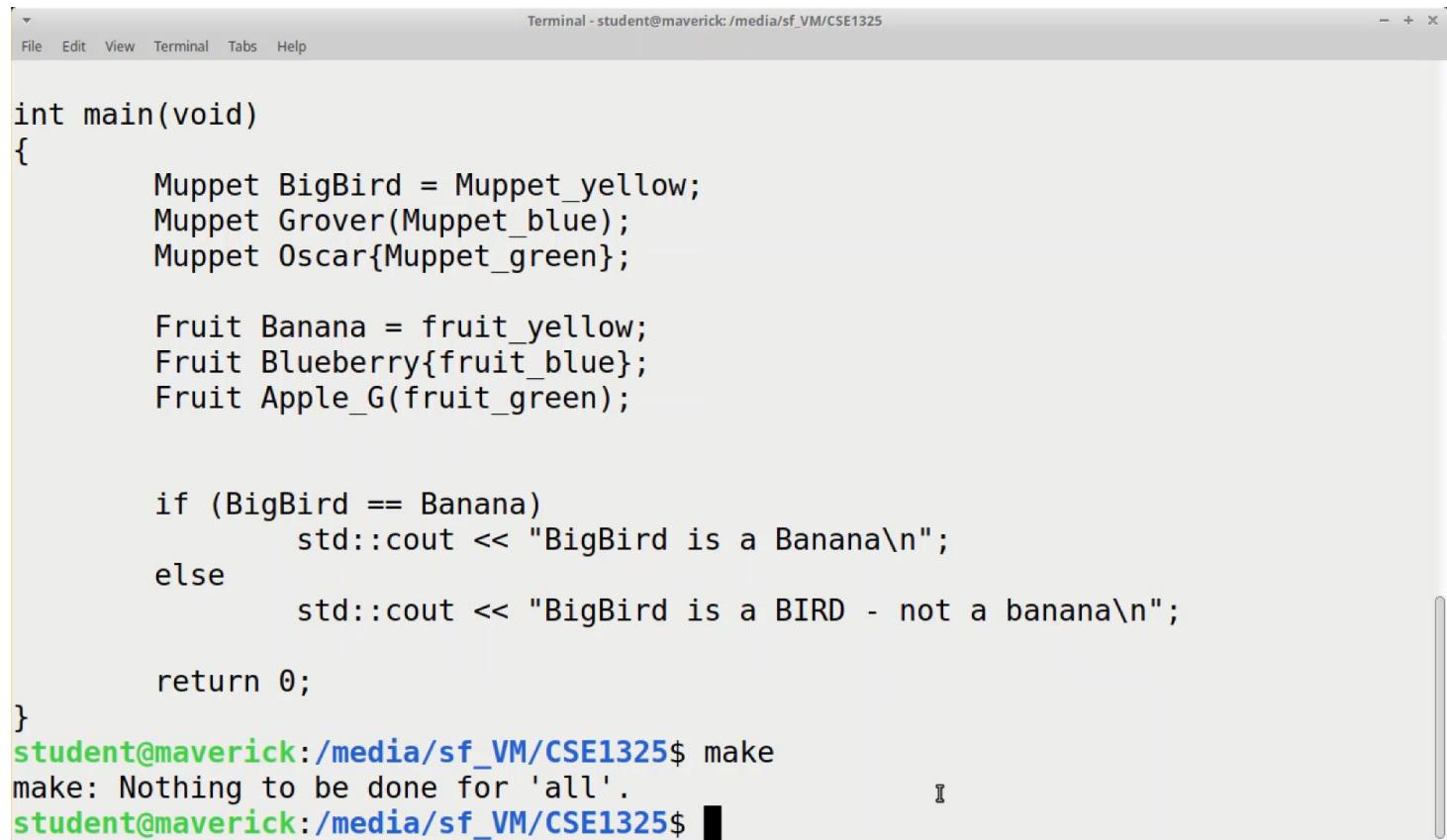
Unintended side effects can occur when we use multiple enumerations within the same program scope.

```
enum Muppet
{
    Muppet_red,
    Muppet_yellow,
    Muppet_blue,
    Muppet_green
};
```

```
enum Fruit
{
    fruit_red,
    fruit_yellow,
    fruit_blue,
    fruit_green
};
```

Enumerated Types

```
Muppet BigBird = Muppet_yellow;  
Muppet Grover(Muppet_blue);  
Muppet Oscar{Muppet_green};  
  
Fruit Banana = fruit_yellow;  
Fruit Blueberry{fruit_blue};  
Fruit Apple_G(fruit_green);  
  
if (BigBird == Banana)  
    std::cout << "BigBird is a Banana";
```



The image shows a terminal window titled "Terminal - student@maverick:/media/sf_VM/CSE1325". The window contains the following C++ code:

```
int main(void)
{
    Muppet BigBird = Muppet_yellow;
    Muppet Grover(Muppet_blue);
    Muppet Oscar{Muppet_green};

    Fruit Banana = fruit_yellow;
    Fruit Blueberry{fruit_blue};
    Fruit Apple_G(fruit_green);

    if (BigBird == Banana)
        std::cout << "BigBird is a Banana\n";
    else
        std::cout << "BigBird is a BIRD - not a banana\n";

    return 0;
}
```

Below the code, the terminal shows the command "make" being run and its output:

```
student@maverick:/media/sf_VM/CSE1325$ make
make: Nothing to be done for 'all'.
student@maverick:/media/sf_VM/CSE1325$
```

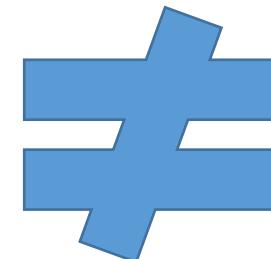
Enumerated Types

When C++ compares two enumerations (`BigBird` and `Banana`), it implicitly converts both of them to integers and compares the integers.

Since `BigBird` and `Banana` were both been set to enumerators that evaluate to 1, this code will logically assume that `BigBird` is equal to `Banana`.

With standard enumerators, there's no way to prevent comparing enumerators from different enumerations. We get a warning, but it still created an executable.

Since this was probably not our intention,
we have a problem.



Another issue :

Because enumerators are placed into the same namespace as the enumeration, an enumerator name can't be used in multiple enumerations within the same namespace:

```
enum Color
{
    RED,
    BLUE,
    GREEN
};
```

```
enum Feeling
{
    HAPPY,
    TIRED,
    BLUE
};
```

Line 28

```
student@cse1325:/media/sf_VM$ make
g++ -c -g -std=c++11 enum1Demo.cpp -o enum1Demo.o
enum1Demo.cpp:28:2: error: redeclaration of 'BLUE'
    BLUE
^~~~
```

enum class/scoped enumeration

enum class (also called a **scoped enumeration**) makes enumerations both strongly typed and strongly scoped.

To make an `enum class`, we use the keyword **class** after the `enum` keyword

```
enum Color
{
    RED,
    BLUE
};
```

```
Color Apple = RED;
```

```
enum class Color
{
    RED,
    BLUE
};
```

```
Color Apple = Color::RED
```

enum class/scoped enumeration

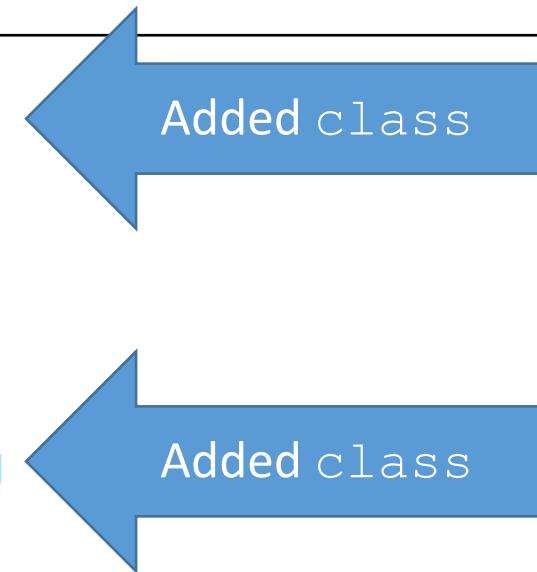
```
enum Color
{
    RED,
    BLUE,
    GREEN
};

enum Feeling
{
    HAPPY,
    TIRED,
    BLUE
};

student@cse1325:/media/sf_VM$ make
g++ -c -g -std=c++11 enum1Demo.cpp -o enum1Demo.o
enum1Demo.cpp:28:2: error: redeclarat
BLUE
^~~~
```

```
enum class Color
{
    RED,
    BLUE,
    GREEN
};

enum class Feeling
{
    HAPPY,
    TIRED,
    BLUE
};
```



```
student@cse1325:/media/sf_VM$ make
g++ -c -g -std=c++11 enum1Demo.cpp -o enum1Demo.o
g++ -g -std=c++11 enum1Demo.o -o enum1Demo.e
```

scoped enumeration

- user defined type
- keywords enum class
- includes a type name and a set of identifiers
- identifiers/enumeration constants must be integer constants
- the value of the enumeration constants start at 0 (unless specified otherwise)
- enumeration constants increment by 1
- identifiers in an enum class must be unique

scoped enumeration

```
keyword           type name          enumeration constants
{ enum class } { Status } { {CONTINUE, WON, LOST} ; }
```

Status is the scoped enum's type name

CONTINUE has a value of 0, WON has a value of 1 and LOST has a value of 2.

```
(gdb) ptype Status
type = enum class Status : int {Status::CONTINUE, Status::WON, Status::LOST}
```

scoped enumeration

```
enum class Status {CONTINUE, WON, LOST};
```

Status is a type so we can declare a variable of that type.

```
Status gameStatus;
```

Then we can assign the enumerated values to our new variable.

```
gameStatus = Status::LOST;
```

```
(gdb) p gameStatus  
$1 = Status::LOST
```

scoped enumeration

```
enum class Status {CONTINUE, WON, LOST};
```

```
gameStatus = Status::LOST;
```

We use the scope resolution operator to "tie" LOST to Status.

If we leave it off

```
enumclassDemo.cpp: In function 'int main()':  
enumclassDemo.cpp:10:15: error: 'LOST' was not declared in this scope  
    gameStatus = LOST;
```

```
gameStatus = LOST;
```

scoped enumeration

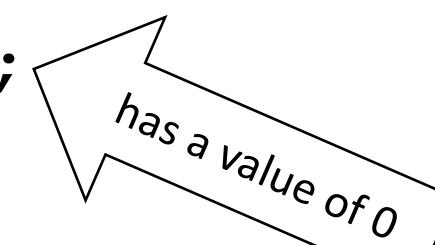
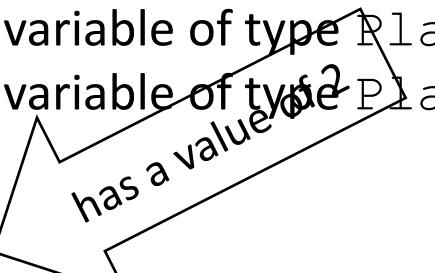
Using the :: allows other enumerated classes to reuse the same enumerated constants with different values.

```
enum class Player1Status {CONTINUE, WON, LOST};  
enum class Player2Status {LOST, CONTINUE, WON};
```

```
Player1Status P1Stat;    P1Stat is a variable of type Player1Status  
Player2Status P2Stat;    P2Stat is a variable of type Player2Status
```

```
P1Stat = Player1Status::LOST;
```

```
P2Stat = Player2Status::LOST;
```



The auto keyword

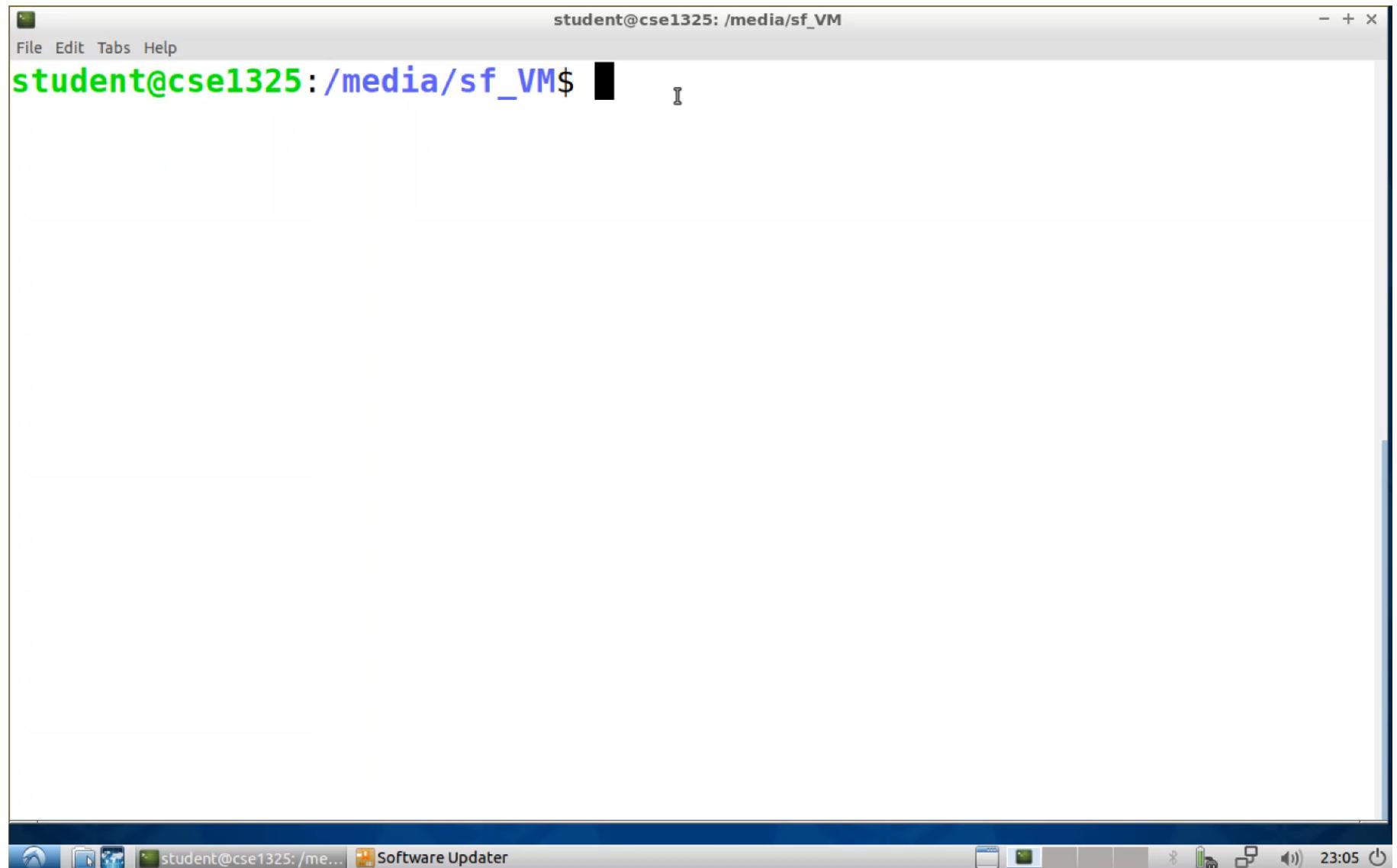
When initializing a variable, the `auto` keyword can be used in place of the variable type to tell the compiler to infer the variable's type from the initializer's type.

This is called **type inference** (also sometimes called type deduction).

```
auto d = 5.0;  
auto i = 1 + 2;
```

The auto keyword

This even
works
with the
return
values
from
functions



A screenshot of a Linux terminal window titled "student@cse1325: /media/sf_VM". The window has a standard title bar with icons and a menu bar labeled "File Edit Tabs Help". The main area of the terminal shows the command prompt "student@cse1325 :/media/sf_VM\$". Below the terminal window, the desktop environment's taskbar is visible, showing other open applications like "Software Updater".

```
student@cse1325 :/media/sf_VM$
```

```
student@cse1325:/media/sf_VM$ more autoDemo.cpp
// auto demo

#include <iostream>

double add(double x, int y)
{
    return x + y;
}

int main()
{
    auto sum = add(5.2, 6);
    std::cout << sum;

    return 0;
}
student@cse1325:/media/sf_VM$
```

```
1 // · auto · demoCRLF
2 CRLF
3 #include <iostream>CRLF
4 CRLF
5 double · add (double · x , · int · y ) CRLF
6 {CRLF
7     ... · return · x · + · y ;CRLF
8 }CRLF
9 .CRLF
10 int · main () CRLF
11 {CRLF
12     ... · auto · sum · = · add (5 .2 , .6 ) ;CRLF
13     → std :: cout << sum ;CRLF
14 CRLF
15     ... · return · 0 ;CRLF
16 }
```

The `auto` keyword

- only works when initializing a variable upon creation. Variables created without initialization values cannot use this feature (as C++ has no context from which to deduce the type).
- the compiler cannot infer types for function parameters at compile time; therefore, `auto` cannot be used for function parameters
- best used when the object's type is hard to type, but the type is obvious from the right hand side of the expression
- using `auto` in place of fundamental data types only saves a few (if any) keystrokes – in the future, we will see examples where the types get complex and lengthy. In those cases, using `auto` can be very nice.

CSE 1325

Week of 09/14/2020

Instructor : Donna French

The auto keyword

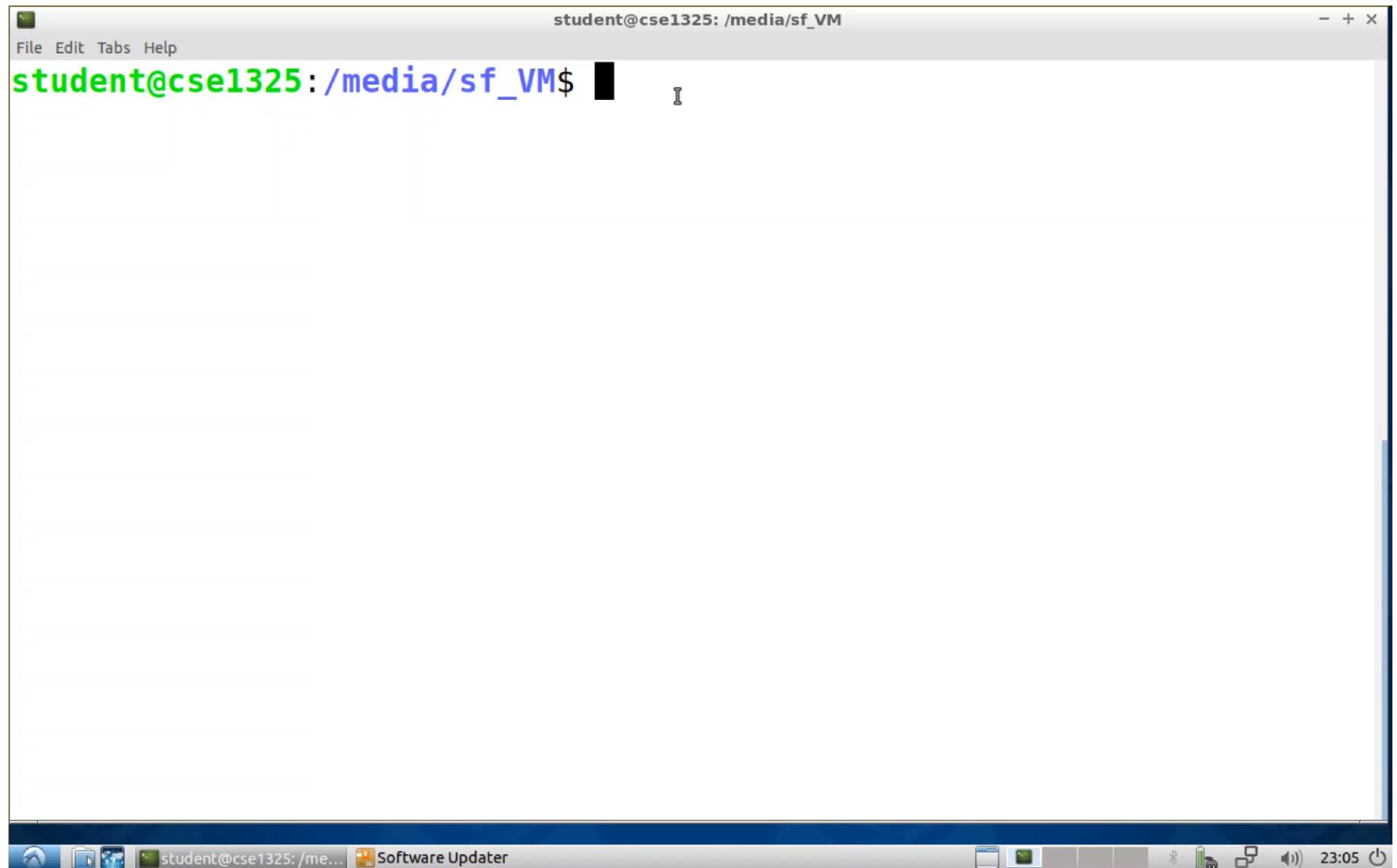
When initializing a variable, the `auto` keyword can be used in place of the variable type to tell the compiler to infer the variable's type from the initializer's type.

This is called **type inference** (also sometimes called type deduction).

```
auto d = 5.0;  
auto i = 1 + 2;
```

The auto keyword

This even
works
with the
return
values
from
functions



A screenshot of a Linux desktop environment showing a terminal window. The terminal window has a title bar with the text "student@cse1325: /media/sf_VM". The menu bar contains "File", "Edit", "Tabs", and "Help". The main area of the terminal shows a command prompt: "student@cse1325 :/media/sf_VM\$". Below the terminal window, the desktop taskbar displays several icons, including a file browser, a terminal icon, and a "Software Updater" icon. The system tray at the bottom right shows the date and time as "23:05".

```
student@cse1325:/media/sf_VM$ more autoDemo.cpp
// auto demo

#include <iostream>

double add(double x, int y)
{
    return x + y;
}

int main()
{
    auto sum = add(5.2, 6);
    std::cout << sum;

    return 0;
}
student@cse1325:/media/sf_VM$
```

```
1 // · auto · demoCRLF
2 CRLF
3 #include <iostream>CRLF
4 CRLF
5 double · add (double · x , · int · y ) CRLF
6 {CRLF
7     ... · return · x · + · y ;CRLF
8 }CRLF
9 .CRLF
10 int · main () CRLF
11 {CRLF
12     ... · auto · sum · = · add (5 .2 , .6 ) ;CRLF
13     → std :: cout << sum ;CRLF
14 CRLF
15     ... · return · 0 ;CRLF
16 }
```

The `auto` keyword

- only works when initializing a variable upon creation. Variables created without initialization values cannot use this feature (as C++ has no context from which to deduce the type).
- the compiler cannot infer types for function parameters at compile time; therefore, `auto` cannot be used for function parameters
- best used when the object's type is hard to type, but the type is obvious from the right hand side of the expression
- using `auto` in place of fundamental data types only saves a few (if any) keystrokes – in the future, we will see examples where the types get complex and lengthy. In those cases, using `auto` can be very nice.

Familiar C++ Libraries

Should look very familiar

atoi(), atof()

```
#include <cstdlib>
```

isdigit(), isalpha(), isalnum(), islower(),
isupper(), isspace(), ispunct()

```
#include <ctype>
```

```
char MyChar;
char MyCharNumber[10];
int MyInt;
float MyFloat;

cout << "Enter a number to be stored in MyCharNumber ";
cin >> MyCharNumber;

MyFloat = atof(MyCharNumber);

cout << "MyCharNumber is " << MyCharNumber << "\tThe float conversion is " << MyFloat << endl;

MyInt = atoi(MyCharNumber);

cout << "MyCharNumber is " << MyCharNumber << "\tThe int conversion is " << MyInt << endl;

cout << "\n\n\nEnter a letter to be UPPERCASED ";
cin >> MyChar;

MyChar = toupper(MyChar);

cout << MyChar << endl;
```

Unary Scope Resolution Operator

C++ provides the unary scope resolution operator (:) to access a global variable when a local variable of the same name is in scope.

The unary scope resolution operator (:) cannot be used to access a local variable of the same name in an outer block.

A global variable can be accessed directly without the unary scope resolution operator if the name of the global variable is not the same as that of a local variable in scope.

Unary Scope Resolution Operator

Global overridden by local in C

```
/* Global version of X */  
int X = 0;  
  
int main(void)  
{  
    /* Local version of X */  
    int X = 123;  
  
    printf("X = %d\n", X);  
  
    return 0;  
}  
  
X = 123
```

Global overridden by local in C++

```
/* Global version of X */  
int X = 0;  
  
int main(void)  
{  
    /* Local version of X */  
    int X = 123;  
  
    cout << "X = " << ::X << endl;  
  
    return 0;  
}  
  
X = 0
```

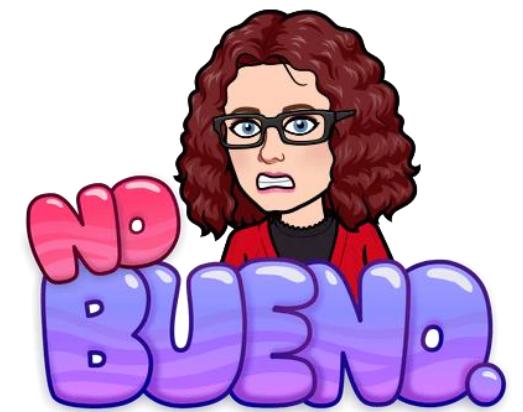
The problem

Programmer Joke

What is the best prefix for a global variable?

//

Global variables are not allowed in Coding Assignments unless specifically included as part of an assignment.



Streams

- C++ I/O occurs in streams of bytes
- A stream is a sequence of bytes
 - Input – bytes flow from a device (e.g., keyboard, drive) to memory
 - Output – bytes flow from memory to a device (e.g., screen, printer)
- C++ provides
 - low-level I/O capabilities
 - unformatted
 - high speed and high volume
 - high-level I/O capabilities
 - formatted
 - people friendly
 - bytes are grouped into meaningful units (integers, floats, characters, strings, etc)
 - type-oriented capabilities

Stream Libraries

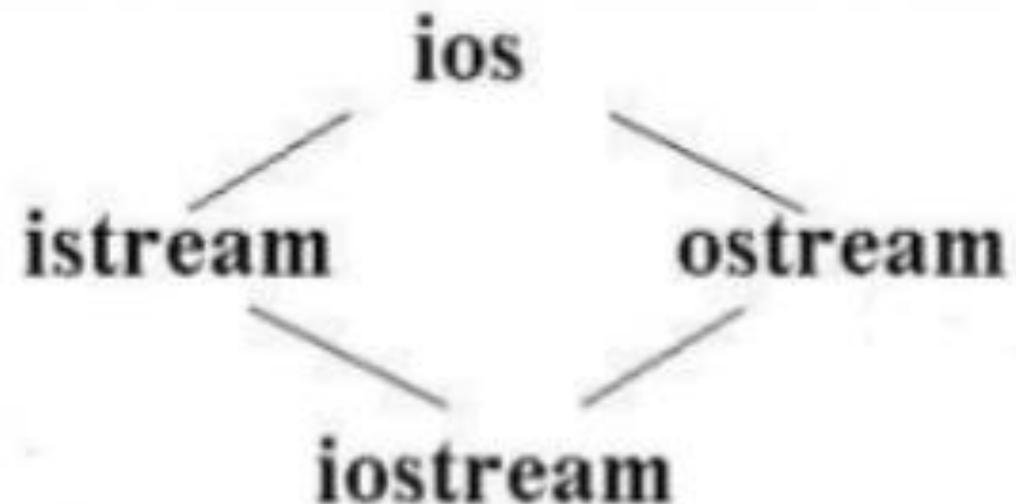
- **iostream**
 - contains objects that perform basic I/O on standard streams
 - cin
 - cout
- **iomanip**
 - contains objects that perform formatted I/O with stream manipulators
- **fstream**
 - contains objects that perform user-controlled file processing operations
- **strstream**
 - contains objects that perform memory formatting



Streams

iostream library

- **istream** class
 - supports stream-input operations
- **ostream** class
 - supports stream-output operations
- **iostream** class
 - supports both stream-input and stream-output operations



Streams

Operator Overloading

<<

>>

left shift operator is overloaded
to be the stream-insertion
operator

right shift operator is overloaded
to be the stream-extraction
operator

cout

object of ostream class
tied to standard output
assumes type of data

cin

object of istream class
tied to standard input
assumes type of data

```
cout << "Hello!" ;
```

```
string first_name, last_name;  
cin >> first_name >> last_name;
```

Streams

Operator Overloading

C++ determines data types automatically – does not require the programmer to supply the type information

printf("%s", MyString);	cout << MyString;
printf("%d", MyInt);	cout << MyInt;
printf("%f", MyDouble);	cout << MyDouble;

Sometimes, this gets in the way...



Operator Overloading

The `<<` operator has been overloaded to print data of type `char*` as a null terminated string. That won't result in the address of a pointer.

```
char MyChar = 'A';
char *MyPtr = &MyChar;

printf("Value of MyChar      %c\n", MyChar);
printf("Address of MyChar    %p\n", MyPtr);

cout << "Value of MyChar " << MyChar << endl;
cout << "Value of MyPtr " << MyPtr << endl;

cout << "Address of MyChar " << (void *)MyPtr << endl;
```

Value of MyChar A
Value of MyPtr A_ ? ? ? ?

File Edit View Terminal Tabs Help

student@maverick:/media/sf_VM/CSE1325\$ gdb ./PrintAddress1Demo.e



Streams

Input/Output Member Functions

`cin.get()`

`get` is a member function of `cin`

retrieves a single character from the standard input stream

returns EOF when the end of file on the stream is encountered

`cout.put()`

`put` is a member function of `cout`

puts one character to the standard output stream

```
#include <stdio.h>

int main(void)
{
    int c;

    while ((c = getchar()) != EOF)
    {
        putchar(c);
    }

    return 0;
}
```

```
#include <iostream>

using namespace std;

int main()
{
    int c;

    while ((c = cin.get()) != EOF)
    {
        cout.put(c);
    }

    return 0;
}
```

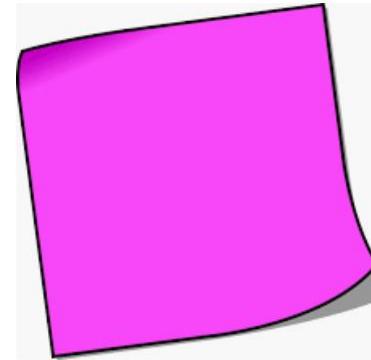
Stream Manipulators

C++ uses stream manipulators to perform formatting tasks

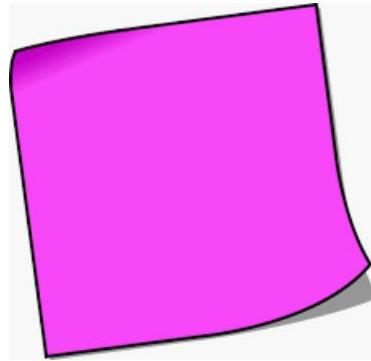
- setting field widths
- setting precision
- setting and unsetting format flags
- setting the fill character in fields
- flushing streams
- inserting a newline in the output stream and flushing the stream
- inserting a null character in the output stream
- skipping whitespace in the input stream

Sticky vs. Non-Sticky Stream Manipulators

A sticky stream manipulator permanently changes stream behavior - permanently until the next change, that is.



A non-sticky stream manipulator only effects the stream for the next value.



Stream Manipulators

Integers

dec, oct , hex , showbase **and** setbase

Integers are normally interpreted as decimal (base 10) values

This interpretation can be altered by inserting a manipulator into the stream.

These only affect integers – using them with other types will have no effect.

```
int MyIntA = 10, MyIntB = 20, MyIntC = 30;  
  
cout << showbase;  
  
cout << "none      " << MyIntA << "\t" << MyIntB << "\t" << MyIntC << endl;  
cout << "decimal   " << dec << MyIntA << "\t" << MyIntB << "\t" << MyIntC << endl;  
cout << "hex       " << hex << MyIntA << "\t" << MyIntB << "\t" << MyIntC << endl;  
cout << "octal     " << oct << MyIntA << "\t" << MyIntB << "\t" << MyIntC << endl;  
cout << "\n\n\n";
```

C++ manipulator `showbase()` function is used to set the `showbase` format flag for the str stream.

none	10	20	30
decimal	10	20	30
hex	0xa	0x14	0x1e
octal	012	024	036

```
cout << oct << MyIntA << "\t" << dec << MyIntB << "\t" << hex << MyIntC << endl;  
cout << noshowbase;  
cout << oct << MyIntA << "\t" << dec << MyIntB << "\t" << hex << MyIntC << endl;
```

```
cout << "\n\n\n";  
cout << setbase(8) << MyIntA << " "  
     << setbase(10) << MyIntB << " "  
     << setbase(16) << MyIntC << endl;
```

setbase () can be called using a variable

setbase (basevalue)

showbase & noshowbase are just flags to enable and disable the program to allow the values to be displayed in various types. Even after noshowbase is used if we use the words such as hex, dec, etc., there is no difference and the original values will be displayed

showbase is STICKY while setbase is not STICKY

setbase () might be better to use since it can be passed a value.

Rather than hardcoding hex, oct, dec, you can just use one setbase ()

Stream Manipulators

Floating Point

`setprecision`, `precision`

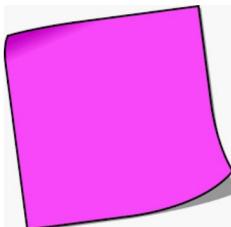
The precision (number of digits displayed) can be controlled for floating point numbers.

`precision`

member function of `cout`

`setprecision`

stream manipulator



precision is reset by capturing current value before altering it and then passing that value to `cout.precision()` to reset the stream

```
double SR1 = sqrt(82.0);
streamsize MyStreamSize = cout.precision(); Means to tell to capture the precision

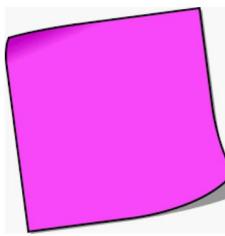
cout << "Square root of 82 with no precision \t\t" << SR1
<< "\n\n" << endl;

for (int i = 1; i < 10; i++)
{
    cout << "Square root of 82 with a precision of " << i
        << "\t\t" << setprecision(i) << SR1 << endl;
}

cout.precision(MyStreamSize); Setting back to the captured precision

cout << "\n\nSquare root of 82 with original precision reset "
<< "\t" << SR1 << endl;
```

Square root of 82 with no precision	9.05539
Square root of 82 with a precision of 1	9
Square root of 82 with a precision of 2	9.1
Square root of 82 with a precision of 3	9.06
Square root of 82 with a precision of 4	9.055
Square root of 82 with a precision of 5	9.0554
Square root of 82 with a precision of 6	9.05539
Square root of 82 with a precision of 7	9.055385
Square root of 82 with a precision of 8	9.0553851
Square root of 82 with a precision of 9	9.05538514
Square root of 82 with original precision reset	9.05539



Stream Manipulation

Floating Point

fixed, scientific

Used to control the output format of floating point numbers

scientific

forces a floating point number to display in scientific notation

fixed

forces a floating point number to display a specific number of digits to the right of the decimal

both of these change the stream – use `defaultfloat` to reset to the default

```
double SR2 = sqrt(82.0);

cout << "Square root of 82 with no stream notation set\t\t"
    << SR2 << "\n\n" << endl;
cout << "Square root of 82 in scientific notation      \t\t" << scientific
    << SR2 << "\n\n" << endl;
cout << "Square root of 82 with no stream notation set\t\t"
    << SR2 << "\n\n" << endl;

cout << "Square root of 82 in fixed notation          \t\t" << fixed
    << SR2 << "\n\n" << endl;
cout << "Square root of 82 with no stream notation set\t\t"
    << SR2 << "\n\n" << endl;
cout << defaultfloat;
cout << "Square root of 82 after resetting to default \t\t"
    << SR2 << "\n\n" << endl;
```

Square root of 82 with no stream notation set	9.05539
Square root of 82 in scientific notation	9.055385e+00
Square root of 82 with no stream notation set	9.055385e+00
Square root of 82 in fixed notation	9.055385
Square root of 82 with no stream notation set	9.055385
Square root of 82 after resetting to default	9.05539

Stream Manipulators

Field Width

`setw, width`

Controls the width - number of character positions in which a value should be output – right justifies text

`width`

member function of `cout`

sets the width for the next `cout`

`setw`

stream manipulator

```
string MyString = "CSE1325";

cout << "Printed with no width specified---" << MyString << endl;
cout.width(40);
cout << "Printed with width set to 40---" << MyString << endl;

for (int i = 10; i < 20; i++)
{
    cout << "Printed with a width of " << i << "----"
        << setw(i) << MyString << endl;
}

cout << "\nPrinted with no width specified---" << MyString << endl;
```

Printed without width set----CSE1325

1 2 3 4 5

12345678901234567890123456789012345678901234567890

Printed with width set to 40---CSE1325

1 2 3 4 5

12345678901234567890123456789012345678901234567890

Printed with a width of 10---- CSE1325

Printed with a width of 11---- CSE1325

Printed with a width of 12---- CSE1325

Printed with a width of 13---- CSE1325

Printed with a width of 14---- CSE1325

Printed with a width of 15---- CSE1325

Printed with a width of 16---- CSE1325

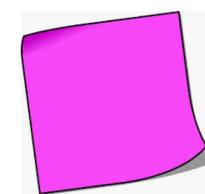
Printed with a width of 17---- CSE1325

Printed with a width of 18---- CSE1325

Printed with a width of 19---- CSE1325

Printed with no width specified---CSE1325

Stream Manipulator boolalpha



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

int main()
{
    cout << "false && false\t" << (false && false) << endl;

    cout << boolalpha
        << "true || false\t" << (true || false) << endl;

    cout << noboolalpha
        << "true ^ true\t" << (true ^ true) << endl;

    return 0;
}
```

false && false	0
true false	true
true ^ true	0

false - keyword that evaluates to zero

true - keyword that evaluates to non-zero

Stream Error State Flags

Each stream object contains a set of **state bits** that represent a stream's state

Stream extraction

- sets the stream's failbit to true if the wrong type of data is input.
- sets the stream's badbit to true if the operation fails in an unrecoverable manner - for example, if a disk fails when a program is reading a file from that disk.

Stream – Error State Flags

eof

- member function of iostream
- used to determine whether end-of-file has been encountered on the stream
- checks the value of the stream's `eofbit` data member
 - set to TRUE for an input stream after end-of-file is encountered after an attempt to extract data beyond the end of the stream
 - set to FALSE if EOF has not been reached

```
cout << "Error State Flags before a bad input operation " << endl
<< "\ncin.eof()      " << cin.eof()
<< "\ncin.fail()    " << cin.fail()
<< "\ncin.good()    " << cin.good()
<< "\ncin.bad()     " << cin.bad();
```

Stream – Error State Flags

fail

- member function of iostream
- used to determine whether a stream operation has failed
- checks the value of the stream's `failbit` data member
 - set to TRUE on a stream when a format error occurs and, as a result, no characters are input
 - when asking for a number and a string is entered
- when `fail()` returns TRUE, the characters are not lost

```
cout << "Error State Flags before a bad input operation " << endl
    << "\ncin.eof()      " << cin.eof()
    << "\ncin.fail()     " << cin.fail()
    << "\ncin.good()    " << cin.good()
    << "\ncin.bad()     " << cin.bad();
```

Stream – Error State Flags

good

- member function of iostream
- used to determine whether a stream operation has failed
- checks the value of the stream's **goodbit** data member
 - set to TRUE for a stream if none of the bits eofbit, failbit or badbit is set to true for the stream

```
cout << "Error State Flags before a bad input operation " << endl
<< "\ncin.eof()      " << cin.eof()
<< "\ncin.fail()     " << cin.fail()
<< "\ncin.good()     " << cin.good()
<< "\ncin.bad()      " << cin.bad();
```

Stream – Error State Flags

bad

- member function of `iostream`
- used to determine whether a stream operation has failed
- checks the value of the stream's `badbit` data member
 - set to TRUE for a stream when an error occurs that results in the loss of data
 - reading from a file when the disk on which the file is stored fails
- indicates a serious failure that is nonrecoverable

```
cout << "Error State Flags before a bad input operation " << endl
<< "\ncin.eof()      " << cin.eof()
<< "\ncin.fail()    " << cin.fail()
<< "\ncin.good()   " << cin.good()
<< "\ncin.bad()    " << cin.bad();
```

Stream – Error State Flags

After an error occurs, you can no longer use the stream until you reset its error state

clear

- member function of iostream
- used to *restore* a stream's state to “good” so that I/O may proceed on that stream
- clears `cin` and sets `goodbit` for the stream

```
cin.clear();
```

```
cout << "Error State Flags before a bad input operation " << endl
    << "\ncin.eof()      " << cin.eof()
    << "\ncin.fail()     " << cin.fail()
    << "\ncin.good()     " << cin.good()
    << "\ncin.bad()      " << cin.bad();
```

cin.eof()	0
cin.fail()	0
cin.good()	1
cin.bad()	0

```
cout << "\n\nEnter a character to cause cin to fail on reading an int ";
```

```
cin >> IntVar;
```

```
cout << "\n\nError State Flags after a bad input operation " << endl
    << "\ncin.eof()      " << cin.eof()
    << "\ncin.fail()     " << cin.fail()
    << "\ncin.good()     " << cin.good()
    << "\ncin.bad()      " << cin.bad();
```

cin.eof()	0
cin.fail()	1
cin.good()	0
cin.bad()	0

```
cin.clear();  
  
cout << "\n\nError State Flags after the clear operation " << endl  
    << "\ncin.eof()      " << cin.eof()  
    << "\ncin.fail()     " << cin.fail()  
    << "\ncin.good()     " << cin.good()  
    << "\ncin.bad()      " << cin.bad() << endl;
```

cin.eof()	0
cin.fail()	0
cin.good()	1
cin.bad()	0

Stream – Error State Flags

`cin` uses the error state flags to terminate a while loop

Input failure

```
int grade;  
while (cin >> grade)  
{  
}
```

<code>cin.eof()</code>	0
<code>cin.fail()</code>	1
<code>cin.good()</code>	0
<code>cin.bad()</code>	0

EOF encountered

```
string MySentence;  
while (cin >> MySentence)  
{  
}
```

<code>cin.eof()</code>	1
<code>cin.fail()</code>	1
<code>cin.good()</code>	0
<code>cin.bad()</code>	0



Streams Input

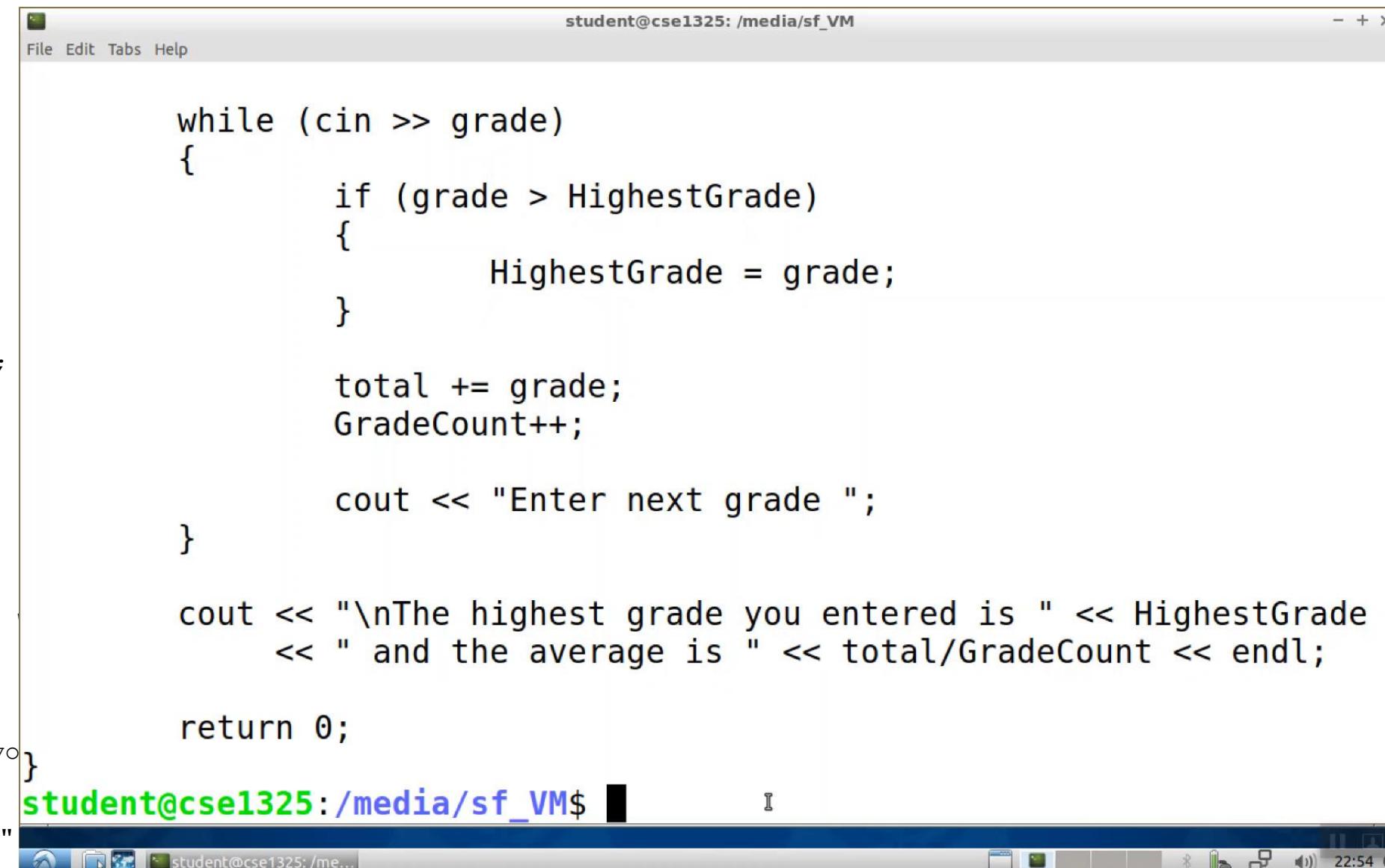
Using `cin` as the condition of a `while` loop

```
while (cin >> grade)
```

Why does this work?

The input to `cin` is converted into a pointer of type `void *`. The value of that pointer is 0 if an error occurred while attempting to read a value or when it reads the EOF indicator. Returning a 0 gives `while` a FALSE causing the condition to fail and the loop to stop.

```
int grade, GradeCount = 0, HighestGrade = -1;  
double total = 0;  
  
cout << "Enter each grade ";  
  
while (cin >> grade)  
{  
    if (grade > HighestGrade)  
    {  
        HighestGrade = grade;  
    }  
  
    total += grade;  
    GradeCount++;  
  
    cout << "Enter next grade "  
}  
  
cout << "\nThe highest grade yo  
    << HighestGrade  
    << " and the average is "  
    << total/GradeCount;
```



A screenshot of a terminal window titled "student@cse1325: /media/sf_VM". The window contains C++ code for calculating the highest grade and average from user input. The code uses a while loop to read grades, an if statement to update the highest grade, and assignment operators to calculate the total and count of grades. It also includes cout statements for prompting the user and displaying the results. The terminal prompt is "student@cse1325:/media/sf_VM\$".

```
while (cin >> grade)  
{  
    if (grade > HighestGrade)  
    {  
        HighestGrade = grade;  
    }  
  
    total += grade;  
    GradeCount++;  
  
    cout << "Enter next grade "  
}  
  
cout << "\nThe highest grade you entered is " << HighestGrade  
    << " and the average is " << total/GradeCount << endl;  
  
return 0;
```

```
#include <iostream>

using namespace std;

int main()
{
    string MySentence;

    cout << "Enter a sentence ";

    while (cin >> MySentence)
    {
        cout << MySentence << endl;
    }
    return 0;
}
```

whilecout2Demo.cpp

Does entering a number cause the `cin` `while` to stop?

No, because a number is treated like a character.

So if letters and numbers are accepted by `cin` into the string `MySentence`, then how to make this loop quit?

Ctrl-D causes `cin` to return 0 which makes the `while` loop quit.

`std::cin`

When we use operator `>>` to get user input and put it into a variable, this is called an “extraction”.

The `>>` operator is called the extraction operator when used in this context.

When the user enters input in response to an extraction operation, that data is placed in a buffer.

std::cin

When the extraction operator is used, the following procedure happens:

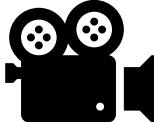
- If there is data already in the input buffer, that data is used for extraction.
- If the input buffer contains no data, the user is asked to input data for extraction (this is the case most of the time). When the user hits <ENTER>, a '\n' character will be placed in the input buffer.
- operator >> extracts as much data from the input buffer as it can into the variable (ignoring any leading whitespace characters, such as spaces, tabs, or '\n').

Any data that cannot be extracted is left in the input buffer for the next extraction.



student@cse1325: /media/sf_VM

- + ×



File Edit Tabs Help

```
#include <iostream>

using namespace std;

int main()
{
    int x = 0, y = 0;

    cout << "Please enter the first integer ";
    cin >> x;
    cout << "You entered " << x << endl;

    cout << "Please enter the second integer ";
    cin >> y;
    cout << "You entered " << y << endl;

    return 0;
}
```

student@cse1325:/media/sf_VM\$ █

█



21:24

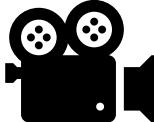


student@cse1325: /media/sf_VM

- + ×

File Edit Tabs Help

student@cse1325:/media/sf_VM\$ gdb ./cinfailDemo.e



student@cse1325:/me...

21:38



student@cse1325: /media/sf_VM

- + ×



File Edit Tabs Help

```
using namespace std;

int main()
{
    int x = 0, y = 0;

    cout << "Please enter the first integer ";
    cin >> x;
    cout << "You entered " << x << endl;

    cin.ignore(32767, '\n');

    cout << "Please enter the second integer ";
    cin >> y;
    cout << "You entered " << y << endl;

    return 0;
}
```

student@cse1325: /media/sf_VM\$



student@cse1325: /media/sf_VM

- + ×

File Edit Tabs Help

```
using namespace std;

int main()
{
    int x = 0, y = 0;

    cout << "Please enter the first integer ";
    cin >> x;
    cout << "You entered " << x << endl;

    cin.clear();
    cin.ignore(32767, '\n');

    cout << "Please enter the second integer ";
    cin >> y;
    cout << "You entered " << y << endl;

    return 0;
}
```

student@cse1325: /media/sf_VM\$



student@cse1325: /me...

21:53



student@cse1325: /media/sf_VM

- + ×

File Edit Tabs Help

```
{  
    int x = 0, y = 0;  
  
    cout << "Please enter the first integer ";  
    cin >> x;  
    cout << "You entered " << x << endl;  
  
    if (cin.fail())  
    {  
        cin.clear();  
        cin.ignore(32767, '\n');  
    }  
  
    cout << "Please enter the second integer ";  
    cin >> y;  
    cout << "You entered " << y << endl;  
  
    return 0;  
}
```

student@cse1325: /media/sf_VM\$!

student@cse1325: /me...



Stream Summary

- C++ I/O occurs in streams which are sequences of bytes
- I/O operations are sensitive to the data type
- <iostream> header – all stream I/O operations
- <iomanip> header – parameterized stream manipulators
- istream
 - cin object
- ostream
 - cout object
- The state of a stream can be tested

Happy Path Testing

Happy Path is the default scenario where no exceptions or error conditions occur.

Happy Path Testing is only testing with well defined test cases that only uses expected inputs; therefore, achieves only expected results. No exceptions or error conditions occur.

Using only Happy Path Testing will result in your program not being robust and not being user friendly.

to_string()

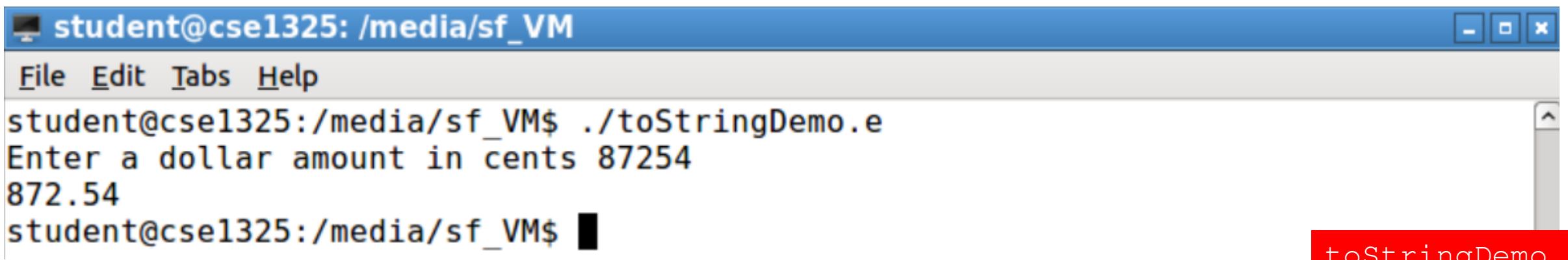
```
long amount;

cout << "Enter a dollar amount in cents ";
cin >> amount;

std::string dollars{std::to_string(amount / 100)};

std::string cents{std::to_string(std::abs(amount % 100))};

cout << dollars + "." + (cents.size() == 1 ? "0" : "") + cents << endl;
```



The screenshot shows a terminal window with a blue header bar. The header bar contains the text "student@cse1325: /media/sf_VM" on the left and standard window control buttons (-, X, and a maximize/minimize button) on the right. Below the header is a menu bar with "File", "Edit", "Tabs", and "Help". The main body of the terminal shows the command "student@cse1325:/media/sf_VM\$./toStringDemo.e" followed by the user's input "Enter a dollar amount in cents 87254" and the program's output "872.54".

```
student@cse1325: /media/sf_VM
File Edit Tabs Help
student@cse1325:/media/sf_VM$ ./toStringDemo.e
Enter a dollar amount in cents 87254
872.54
student@cse1325:/media/sf_VM$
```

String Stream Processing

C++ stream I/O includes capabilities for inputting from, and outputting to, strings in memory

Class `istringstream`

Supports input from a string

Class `ostringstream`

Supports output to a string

`stringstream` is derived from `iostream` and can be used for both input and output.

Header file `<sstream>` must be included in addition to `<iostream>`

String Stream Processing

There are two ways to get data into a stringstream...

1. Use the insertion operator <<

```
stringstream os;
os << "This is silly" << endl;
```

2. Use the str(string) function to set the value of the buffer

```
stringstream os;
os.str("This is silly");
```

String Stream Processing

There are two ways to get data out of a stringstream...

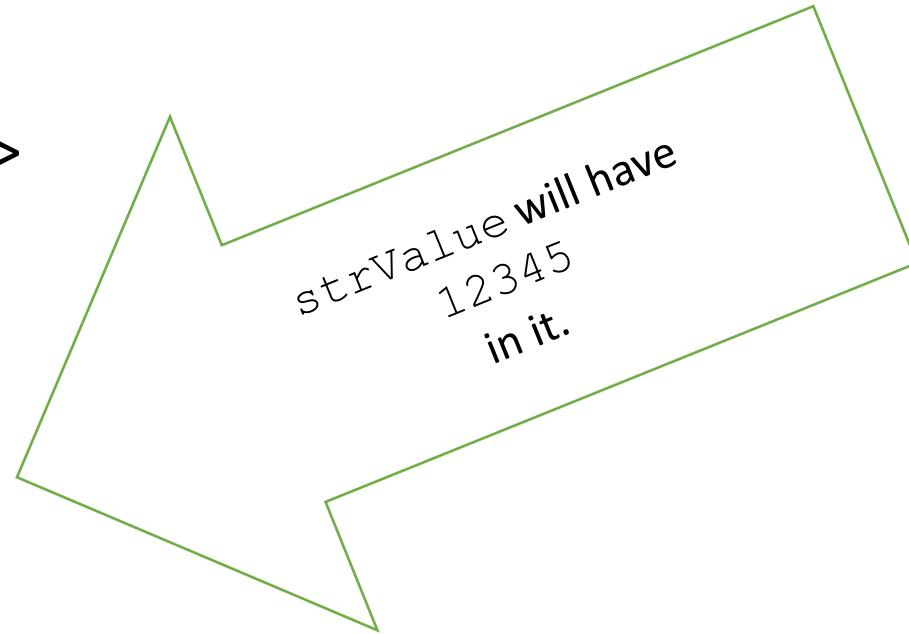
1. Use the `str()` function to retrieve the results of the stringstream

```
stringstream os;
os << "12345 67.89" << endl;
cout << os.str();
```

2. Use extraction operator `>>`

```
stringstream os;
os << "12345 67.89";
```

```
string strValue;
os >> strValue;
```



strValue will have
12345
in it.

String Stream Processing

```
stringstream os;  
os << "12345 67.89";
```

```
string strValue1;  
os >> strValue1;
```

```
string strValue2;  
os >> strValue2;
```

```
cout << strValue1 << " - " << strValue2 << endl;
```

12345 - 67.89

extracts all characters from `os` up to the whitespace

extracts all characters from `os` after the whitespace

String Stream Processing

Note that the `>>` operator iterates through the string...

each successive use of `>>` returns the next extractable value in the stream.

On the other hand, `str()` returns the whole value of the stream, even if the `>>` has already been used on the stream.

String Stream Processing

```
string FullGreeting{"Hello_there! How_are_you? I_am_fine. 1 2 3"};
string Greeting1;
string Greeting2;
string Greeting3;
int GNum1, GNum2, GNum3;
stringstream MySS{FullGreeting};

MySS >> Greeting1 >> Greeting2 >> Greeting3
                >> GNum1 >> GNum2 >> GNum3;

cout << "The " << GNum1 << "st greeting is " << Greeting1 << endl
    << "The " << GNum2 << "nd greeting is " << Greeting2 << endl
    << "The " << GNum3 << "rd greeting is " << Greeting3 << endl;
```

The 1st greeting is Hello_there!
The 2nd greeting is How_are_you?
The 3rd greeting is I_am_fine.

String Stream Processing

```
string String1{"Hello there!"};  
string String2{"How are you?"};  
string String3{"I am fine."};  
  
stringstream SS1;  
  
SS1 << String1 << endl;  
cout << SS1.str();  
  
cout << "-----" << endl;  
  
SS1 << String2 << endl;  
cout << SS1.str();  
  
cout << "-----" << endl;  
  
SS1 << String3 << endl;  
cout << SS1.str();
```

Hello there!

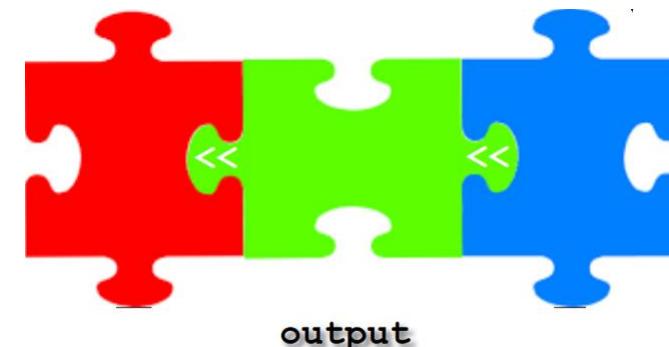
Hello there!

How are you?

Hello there!

How are you?

I am fine.



String Stream Processing

```
string String1{"Hello there!"};  
string String2{"How are you?"};  
string String3{"I am fine."};  
stringstream SS2;  
  
cout << endl << "Printing in reverse with tabs\n";  
SS2 << String3 << "\t" << String2 << "\t" << String1 << endl;  
cout << SS2.str();
```

Printing in reverse with tabs
I am fine. How are you?

Hello there!

String Stream Processing

When reusing a `stringstream` variable, you should set the stream to empty/blanks and call the `clear()` function to clear any flags on the stream.

```
stringstream os;
os << "Hello ";

os.str(""); // erase the buffer
os.clear(); // reset error flags

os << "World!";
cout << os.str();
```

CSE 1325

Week of 09/21/2020

Instructor : Donna French

File Processing

C++ stream I/O includes capabilities for writing to and reading from files.

Class ifstream

- Supports file input (reading from a file)

Class ofstream

- Supports file output (writing to a file)

Class fstream

- Supports file input/output (writing to/reading from a file)

Header file <fstream> must be included in addition to <iostream>

File Processing

- C++ imposes no structure on files
- The concept of a record does not exist in C++
- The program/programmer must enforce a definition on the stream of data

File Processing

Stream of data in a file

Richard Tiffany Gere
Hercules John Ben

Kobe Bean
Geza Affleck

Bryant Matt

Elton Paige
Paige Damon

If we impose a structure on this stream of
characters 1 - 9 = First name
characters 10 - 19 = Middle name
characters 20 – 31 = Last name

1 2 3

1234567890123456789012345678901

Richard Tiffany Gere

Kobe Bean Bryant

Elton Hercules John

Ben Geza Affleck

Matt Paige Damon

File Processing – Opening a File

Open a file for output by creating an `ofstream` object (calling a constructor)

Two arguments

filename

file open mode

```
ofstream MyOutputStream{ "outfile.txt", ios::out};
```

File Processing – File Open Modes

Ios file mode	Meaning
app	Opens the file in append mode
ate	Seeks to the end of the file before reading/writing
binary	Opens the file in binary mode (instead of text mode)
in	Opens the file in read mode (default for ifstream)
out	Opens the file in write mode (default for ofstream)
trunc	Erases the file if it already exists

File Processing – Opening a File

After opening a file, check if the open was successful

`is_open()`

member function of `ofstream`

returns TRUE if file is open and associated with given stream and FALSE if it is not

```
if (MyOutputStream.is_open())
{
    cout << "The file opened" << endl;
}
else
{
    cout << "The file did not open" << endl;
}
```

File Processing – Writing to a File

```
ofstream MyOutputStream("outfile.txt", ios::out);
int Int1 = 10;
double Double1 = 12.34;

if (MyOutputStream.is_open())
{
    MyOutputStream << "I am writing this sentence to outfile.txt";
    MyOutputStream << Int1 << Double1;
}
else
{
    cout << "The file did not open" << endl;
}

MyOutputStream.close();
```



student@cse1325: /media/sf_VM

- + ×

File Edit Tabs Help

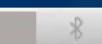
student@cse1325: /media/sf_VM\$ █ I

```
MyOutputStream << "I am writing this sentence to outfile.txt";
MyOutputStream << Int1 << Double1;
```



student@cse1325: /me...

[Software Updater]



12:56





student@cse1325: /media/sf_VM

- + ×

File Edit Tabs Help

student@cse1325:/media/sf_VM\$ I

File Processing – Appending to a File

```
ofstream MyOutputStream{"outfile.txt", ios::app};  
int Int1 = 100;  
double Double1 = 120.34;  
  
if (MyOutputStream.is_open())  
{  
    MyOutputStream << "\nWriting a new line to my existing file opened with append";  
    MyOutputStream << endl << Int1 << endl << Double1;  
}  
else  
{  
    cout << "The file did not open" << endl;  
}  
  
MyOutputStream.close();
```



student@cse1325: /media/sf_VM



File Edit Tabs Help

student@cse1325: /media/sf_VM\$ v

File Processing – Reading from a File

```
ifstream MyInputStream{"makefile"};
string MyLine;
int LineCounter = 0;

if (MyInputStream.is_open())
{
    while (getline(MyInputStream, MyLine))
    {
        cout << "Line " << ++LineCounter << "\t" << MyLine << endl;
    }
}
else
{
    cout << "The file did not open" << endl;
}

MyInputStream.close();
```

File Edit Tabs Help

student@cse1325: /media/sf_VM\$ make

g++ -c -g -std=c++11 ifstream1Demo.cpp -o ifstream1Demo.o

g++ -g -std=c++11 ifstream1Demo.o -o ifstream1Demo.e

student@cse1325: /media/sf_VM\$./ifstream1Demo.e

Line 1 #makefile for C++ program

Line 2 SRC = ifstream1Demo.cpp

Line 3 OBJ = \$(SRC:.cpp=.o)

Line 4 EXE = \$(SRC:.cpp=.e)

Line 5

Line 6 CFLAGS = -g -std=c++11

Line 7

Line 8 all : \$(EXE)

Line 9

Line 10 \$(EXE): \$(OBJ)

Line 11 g++ \$(CFLAGS) \$(OBJ) -o \$(EXE)

Line 12

Line 13 \$(OBJ) : \$(SRC)

Line 14 g++ -c \$(CFLAGS) \$(SRC) -o \$(OBJ)

Line 15

student@cse1325: /media/sf_VM\$ █

```
char MyChar;
int DigitCounter = 0;
ifstream MyPhoneNumberFile;
ifstream MyPhoneNumberFile;
if (MyPhoneNumberFile.is
{
    cout << "eofbit is ";
    cout << "goodbit is ";
while (MyPhoneNumberFile)
{
    if (isdigit(MyChar))
    {
        cout.put(MyChar);
        if (! (++DigitCounter == 10))
            cout << endl;
    }
    cout << endl;
    cout << "eofbit is 1";
    cout << "\n\n";
    cout << "goodbit is 0";
    cout << "goodbit is ";
}
else
    cout << "Unable to open file";
```

student@cse1325:/media/sf_VM\$ more PhoneNumbers.txt

817a415b0687
21c47722d387
907d3f429811

student@cse1325:/media/sf_VM\$./ifstream2Demo.e

eofbit is 0
goodbit is 1

8174150687
2147722387
9073429811

eofbit is 1
goodbit is 0
goodbit is

student@cse1325:/media/sf_VM\$ █

File Processing – Closing a File

When `main()` terminates, the `ofstream` destructor is implicitly called and the file is closed.

Good coding style is to close your own files as soon as you are done using them. In a production environment, files are shared by many processes and should be opened only when needed and closed as soon as possible to prevent conflicts with other processes.

```
MyOutputStream.close();
```

You want to avoid holding files open unnecessarily in a shared environment because other programs – maybe hundreds of other programs may need that same file.¹⁶

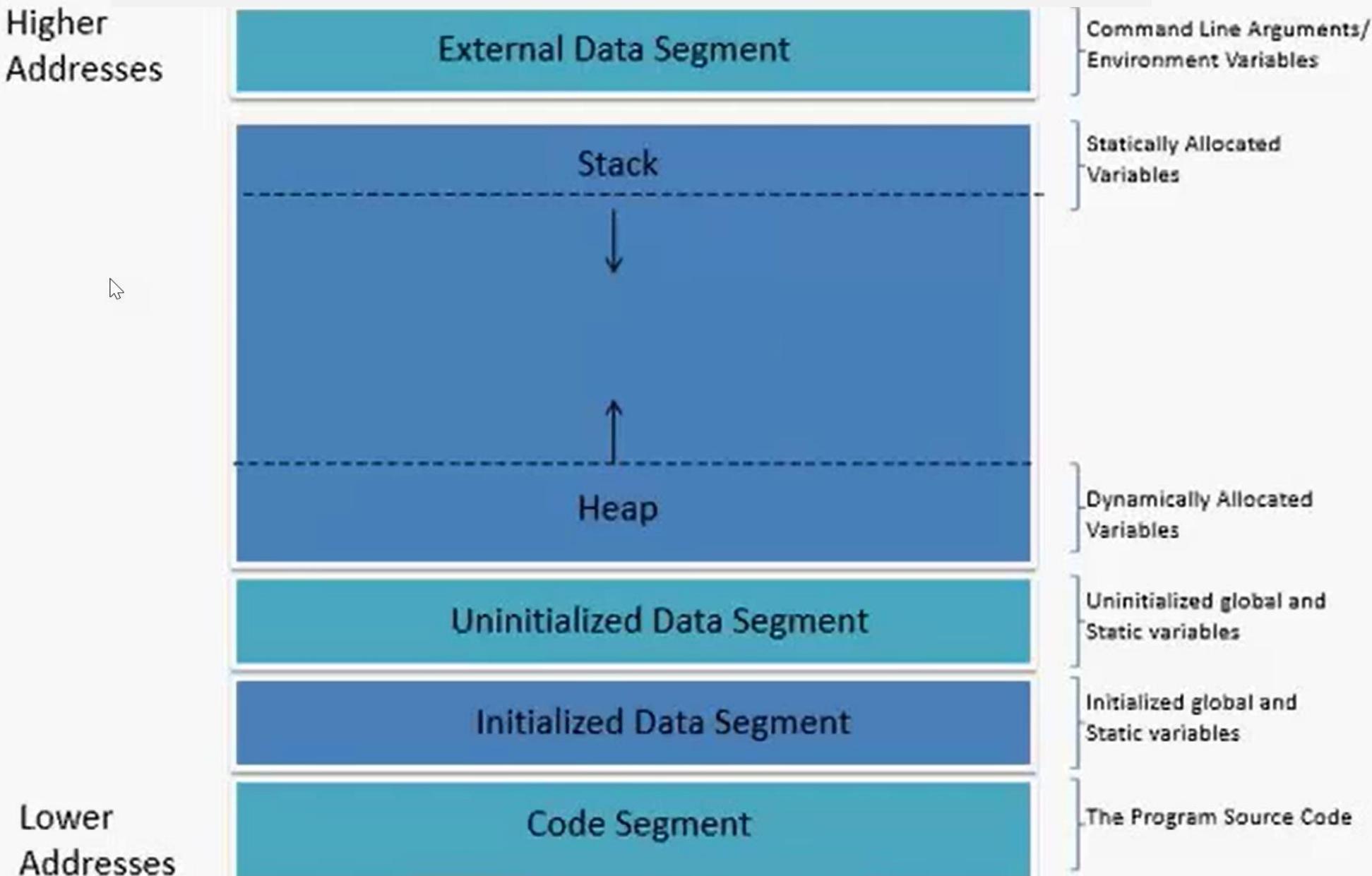
Dynamic Memory Allocation in C++

Dynamic memory allocation is a way for running programs to request memory from the operating system when needed.

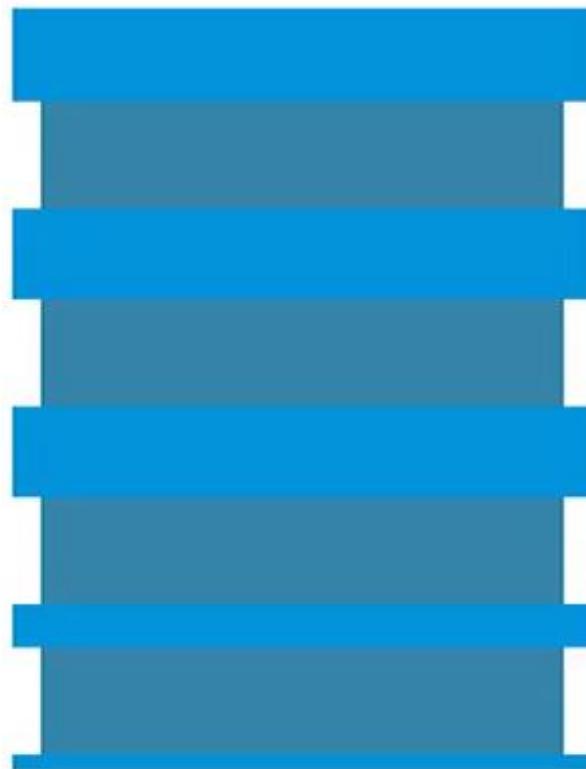
This memory does not come from the program's limited stack memory.

It is allocated from a much larger pool of memory managed by the operating system called the **heap**. On modern machines, the heap can be gigabytes in size.

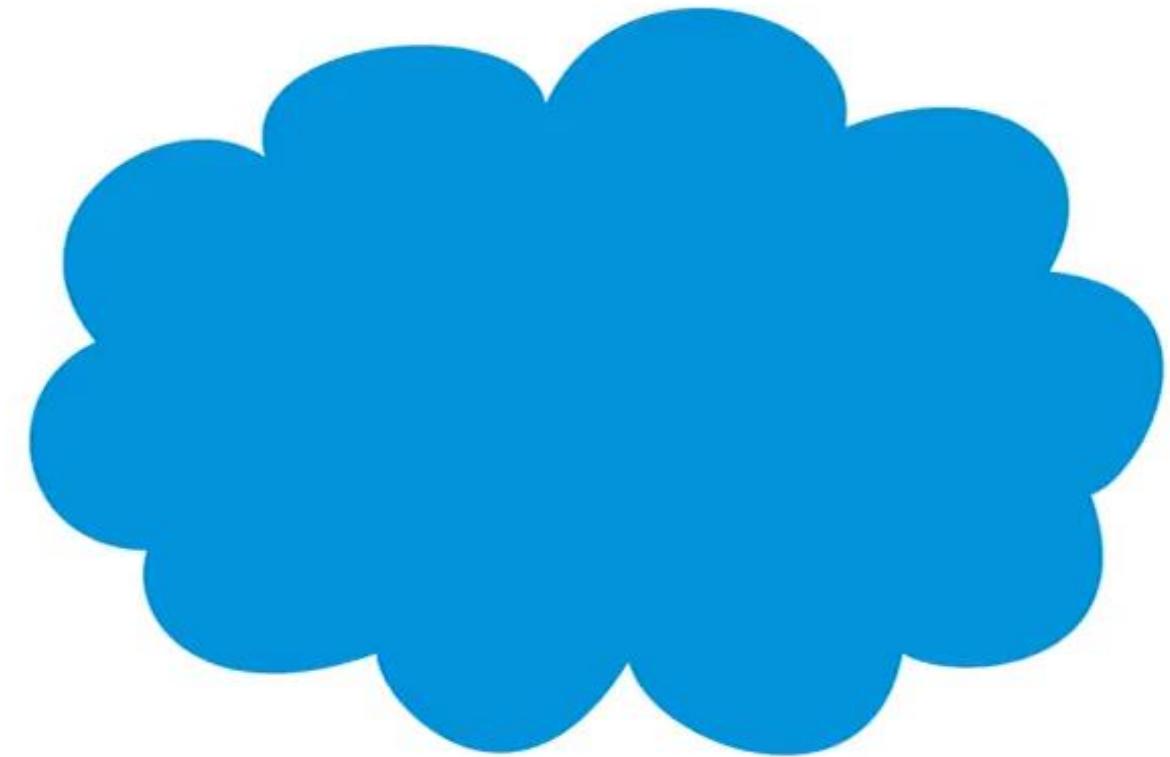
Layout of Memory



Stack vs Heap



Stack



Heap

Heap Memory vs Stack Memory

- **Stack** is used for static memory allocation
 - static variables, strings, local variables, function parameters
 - faster than the heap
 - LIFO
 - do not deallocate variables
 - managed by CPU and will not become fragmented
 - has a predetermined size
- **Heap** is used for dynamic memory allocation
 - dynamically allocated variables
 - slower than the stack
 - random access
 - must free allocated memory
 - managed by programmer – can become fragmented
 - size only limited by machine's memory

Both are stored in the computer's RAM .

`new`

You can control memory *allocation* and *deallocation* in C++ just like we did in C.

This is known as dynamic memory management

C used `malloc()`/`calloc()` and C++ uses `new`

C used `free()` and C++ uses `delete`

C used `realloc()` and C++

`new`

Anything created with `new` is created in the heap

a region of memory assigned to each program for storing dynamically allocated objects.

Once memory is allocated, you can access it via the pointer that operator `new` returns.

Return memory by using the `delete` operator to deallocate it.

new

The new operator

- allocates storage of the proper size
- returns a **pointer** to the type specified to the right of the new operator.

If `new` is unable to find sufficient space in memory, it indicates that an error occurred by “throwing an exception.”

new

To statically allocate memory for a variable

```
int MyStatic;
```

To dynamically allocate memory for a variable

```
int *MyDynamic = new int;
```

```
(gdb) p/a MyDynamic  
$1 = 0x555555767e70  
(gdb) p/a MyStatic  
$2 = 0x7fff  
(gdb) ptype MyStatic  
type = int  
(gdb) ptype MyDynamic  
type = int *
```

Initializing Dynamic Memory

To initialize our static variable

```
MyStatic = 1;
```

```
(gdb) p MyStatic  
$5 = 1
```

```
(gdb) p *MyDynamic  
$6 = 1
```

To initialize our dynamic variable

```
*MyDynamic = 1;
```

Initializing Dynamic Memory

We can initialize a static variable with the declaration

```
int MyStatic{1};          (gdb) p MyStatic  
$5 = 1  
(gdb) p *MyDynamic  
$6 = 1
```

We can do the same with a dynamic variable

```
int *MyDynamic{new int{1}}
```

Initializing Dynamic Memory

Write the statements to

- declare a dynamic variable named Boat of type float and initialize it during the declaration to 5 . 4
- print Boat's value

```
float *Boat{new float{5.4}};  
std::cout << *Boat;
```

delete

To destroy/free dynamically allocated memory, use delete

```
delete MyDynamic;
```

This statement deallocates the memory by returning the memory to the heap.

delete

Do not delete memory that was not allocated by new.
Doing so results in undefined behavior.

```
int MyStatic{1};  
  
int *MyDynamic{new int{1}};  
  
delete &MyStatic;
```

```
7           int MyStatic{1};  
8           (gdb) int *MyDynamic{new int{1}};  
9           (gdb) 10 delete &MyStatic;  
11          (gdb) p &MyStatic  
12          $1 = (int *) 0xfffffffffe07c  
13          (gdb) step  
  
Program received signal SIGSEGV, Segmentation fault.
```

delete

The `delete` operator does not *actually* delete anything.

It simply returns the memory being pointed to back to the operating system.

The operating system is then free to reassign that memory to another application (or to this application again later).

```
          delete  
7           int MyStatic{1};  
(gdb) step  
8           int *MyDynamic{new int{1}};  
(gdb)  
10          delete MyDynamic;  
(gdb) p MyDynamic  
$1 = (int *) 0x555555767e70  
(gdb) step  
13          return 0;  
(gdb) p MyDynamic  
$2 = (int *) 0x555555767e70
```

Memory Leak

Memory leaks happen when your program loses the address of dynamically allocated memory before giving it back to the operating system. Going out of the scope of a dynamically allocated variable is a good way to lose the address.

When this happens, your program cannot delete the dynamically allocated memory because it no longer knows where it is.

The operating system also cannot use this memory because that memory is considered to be still in use by your program.

Memory leaks eat up free memory while the program is running, making less memory available not only to this program, but to other programs as well. Programs with severe memory leak problems can eat all the available memory, causing the entire machine to run slowly or even crash. Only after your program terminates is the operating system able to clean up and “reclaim” all leaked memory.

student@cse1325: /media/sf_VM

File Edit Tabs Help

student@cse1325:/media/sf_VM\$ gdb ./newDemo.e



Null Pointer

After you delete a block of dynamically allocated memory, be sure not to delete the same block again.

One way to guard against this is to immediately set the pointer to `nullptr`.

Deleting a `nullptr` has no effect.

```
Breakpoint 1, main () at newDemo.cpp:7
7           int MyStatic{1};
(gdb) step
8           int *MyDynamic{new int{1}};
(gdb)
10          delete MyDynamic;
(gdb)
11          MyDynamic = nullptr;
(gdb) p MyDynamic
$1 = (int *) 0x555555767e70
(gdb) step
14          return 0;
(gdb) p MyDynamic
$2 = (int *) 0x0
```

C++ Standard Library

C++ programs are typically written by combining “prepackaged” functions and classes available in the C++ Standard Library with new functions and classes you write.

The C++ Standard Library provides a rich collection of functions.

There are three key components of the Standard Library—containers (*templatized* data structures), iterators and algorithms.

C++ Standard Library

Containers are data structures capable of storing objects of almost any data type (there are some restrictions).

We'll see that there are three styles of container classes—first-class containers, container adapters and near containers.

The containers are divided into four major categories—sequence containers, ordered associative containers, unordered associative containers and container adapters.

C++ Standard Library

<i>Unordered associative containers</i>	
Container class	Description
<code>unordered_set</code>	Rapid lookup, no duplicates allowed.
<code>unordered_multiset</code>	Rapid lookup, duplicates allowed.
<code>unordered_map</code>	One-to-one mapping, no duplicates allowed, rapid key-based lookup.
<code>unordered_multimap</code>	One-to-many mapping, duplicates allowed, rapid key-based lookup.

C++ Standard Library

Container class	Description
<i>Sequence containers</i>	
array	Fixed size. Direct access to any element.
deque	Rapid insertions and deletions at front or back. Direct access to any element.
forward_list	Singly linked list, rapid insertion and deletion anywhere. Added in C++11.
list	Doubly linked list, rapid insertion and deletion anywhere.
vector	Rapid insertions and deletions at back. Direct access to any element.

Sequence Containers

arrays

fixed-size collections consisting of data items of the same type

vectors

collections consisting of data items of the same type that can grow and shrink dynamically at execution time.



vector

Simple and useful way to store data

A vector is a sequence of elements that you can access by an index

MyVector

5	9	2	12	22	83
0	1	2	3	4	5

MyVector[0] is 5

MyVector[4] is 22



vector

- Need to add an include to use vectors

```
#include <vector>
```

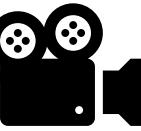
- Declaring a vector

```
vector<type> vectorname;
```

```
vector<type> vectorname(number of elements);
```

- Initializing and declaring a vector

```
vector<type> vectorname{comma delimited list of elements};
```



student@cse1325: /media/sf_VM

- + ×

File Edit Tabs Help

student@cse1325:/media/sf_VM\$ g█

42

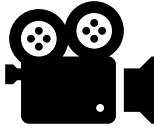


student@cse1325: /media/sf_VM

- + ×

File Edit Tabs Help

student@cse1325: /media/sf_VM\$



student@cse1325: /me...



43

15:28

vector

It is common to process *all* the elements of a vector.

The C++11 [range-based for statement](#) allows you to do this *without using a counter*,

This statement avoids the possibility of “stepping outside” the vector and eliminating the need for bounds checking.

When processing all elements of a vector, if you do not need to access to a vector element’s subscript, use the range based for statement.

vector

for loop

```
vector<int> MyVector = {2, 4, 6, 8};  
  
int i;  
for (i = 0; i < MyVector.size(); i++)  
    cout << setw(5) << MyVector[i];
```

2 4 6 8

range-for-loop

```
vector<int> MyVector = {2, 4, 6, 8};  
  
for (int x : MyVector)  
    cout << setw(5) << x;
```

2 4 6 8

for each iteration, assign the next element of MyVector to int variable x, then execute the following statement

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

using namespace std;

int main()
{
    vector<int> ABunch {1,2,3,4,5,6,7,8,9,10};

    for (int i = 0; i < ABunch.size(); i++)
    {
        cout << "ABunch[" << ABunch[i] <<"]" << endl;
    }

    // read as "for each int banana in ABunch
    for (int banana : ABunch)
    {
        cout << "ABunch[" << banana <<"]" << endl;
    }
}
```

vector

The range-based for statement can be used in place of the counter-controlled for statement whenever code looping through a vector does not require access to the element's subscript.

If a program needs use subscripts for some reason other than simply to loop through a vector

to print a subscript number next to each array element value

use the counter-controlled for statement.

vector

```
int i;  
vector <float> Bank{1.2345,2.3456,3.4567,4.5678,5.6789};  
  
for (i = 0; i < Bank.size(); i++)  
    cout << setprecision(i) << setw(i+2) << Bank[i];  
  
1 2 3.5 4.57 5.679
```

How do we change this to be a range based for statement?

```
student@cse1325:/media/sf_VM$ ./vector5Demo.e
```

```
You are in
```

CSE1325

```
setw() is not sticky
```

```
student@cse1325:/media/sf_VM$ █
```

```
vector <string> ClassName{"CSE1325"};
```

```
cout << "You are in " << endl;
```

```
for (string it : ClassName)
```

```
    cout << setw(50) << it;
```

```
cout << "\nsetw() is not sticky" << endl;
```

```
vector <int> MyList{1,2,3,4,5,6};
```

vector

```
#include <iostream>

int main()
{
    vector<string> CatNames {"Shade", "Appa", "Sylvester", "Josie"};

    for (string it : CatNames)
    {
        cout << it << "\t";
    }

    return 0;
}
```

```
student@cse1325:/media/sf_VM$ make
make: Warning: File 'makefile' has modification time 137 s in the future
g++ -c -g -std=c++11 auto1Demo.cpp -o auto1Demo.o
auto1Demo.cpp: In function 'int main()':
auto1Demo.cpp:7:2: error: 'vector' was not declared in this scope
  vector<string>CatNames{"Shade", "Appa", "Sylvester", "Josie"};
^~~~~~
auto1Demo.cpp:7:2: note: suggested alternative: 'perror'
  vector<string>CatNames{"Shade", "Appa", "Sylvester", "Josie"};
^~~~~~
perror
auto1Demo.cpp:7:9: error: 'string' was not declared in this scope
  vector<string>CatNames{"Shade", "Appa", "Sylvester", "Josie"};
^~~~~~
auto1Demo.cpp:7:9: note: suggested alternatives:
In file included from /usr/include/c++/7/iosfwd:39:0,
                  from /usr/include/c++/7/ios:38,
                  from /usr/include/c++/7/ostream:38,
                  from /usr/include/c++/7/iostream:39,
```

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

int main()
{
    std::vector<std::string> CatNames{"Shade", "Appa", "Sylvester", "Josie"};

    for (auto it : CatNames)
    {
        std::cout << it << "\t";
    }

    return 0;
}
```

Shade Appa Sylvester Josie

vector

push_back()

- member function of vector (like size())
- adds a new element to the end of the vector

```
vector<int> MyVector = {2, 4, 6, 8};
```

```
MyVector.push_back(10);  
MyVector.push_back(12);
```

push_back()

2 4 6 8

The size of MyVector is 4 and the capacity of MyVector is 4

MyVector after push_back(10) MyVector.push_back(10);

2 4 6 8 10

```
for (int x : MyVector)  
    cout << x << "\t";
```

The size of MyVector is 5 and the capacity of MyVector is 8

MyVector after push_back(12)

2 4 6 8 10 12

The size of MyVector is 6 and the capacity of MyVector is 8

```
vector<int> MyVector = {2,4,6,8};

for (int x : MyVector)
    cout << setw(5) << x;

cout << "\nMyVector.size() " << MyVector.size()
    << " MyVector.capacity() " << MyVector.capacity() << endl;
```

2 4 6 8

MyVector.size() 4 MyVector.capacity() 4

capacity()

Vectors may allocate extra capacity

When a vector is resized, the vector may allocate more capacity than is needed.

This is done to provide some “breathing room” for additional elements, to minimize the number of resize operations needed.

Allows the vector to not need to reallocate every time a `push_back` is done.

6

CSE 1325

Week of 09/28/2020

Instructor : Donna French



vector

- Need to add an include to use vectors

```
#include <vector>
```

- Declaring a vector

```
vector<type> vectorname;
```

```
vector<type> vectorname(number of elements);
```

- Initializing and declaring a vector

```
vector<type> vectorname{comma delimited list of elements};
```

vector

How would you declare

a char vector named Frog?
vector<char>Frog;

a float vector named Toad with 7 elements?
vector<float>Toad(7);

a bool vector named Cat initialized to false, true, true, true, false
vector<bool>Cat{0,1,1,true,0};

vector

```
13         vector<char>Frog;
(gdb)
14         vector<float>Toad(7);
(gdb)
15         vector<bool>Cat{0,1,1,true,0};
(gdb)

(gdb) p Frog
$2 = std::vector of length 0, capacity 0
(gdb) p Toad
$3 = std::vector of length 7, capacity 7 = {0, 0, 0, 0, 0, 0, 0}
(gdb) p Cat
$4 = std::vector<bool> of length 5, capacity 64 = {0, 1, 1, 1, 0}
```

vector

Note that in both the uninitialized and initialized case, you do not need to include the length at compile time. This is because `std::vector` will dynamically allocate memory for its contents as requested.

When you add to the end of a vector, memory will be dynamically allocated for the new element.

However, if you try to add PAST the end of the vector, the vector will not resize.

vector

```
13         vector<char>Frog;  
(gdb)  
14         vector<float>Toad(7);  
(gdb)  
15         vector<bool>Cat{0,1,1,true,0};  
(gdb)  
17         cout << Toad[0] << endl;  
(gdb)  
0  
18         cout << boolalpha << Cat[0] << endl;  
(gdb)  
0  
19         cout << Frog[0] << endl;  
(gdb)
```

How would you print the word “false” instead of the 0?

Program received signal SIGSEGV, Segmentation fault.
0x000055555554fca in main () at vector1Demo.cpp:19



vector

A **vector** knows its size

```
vectorname.size()
```

```
vector<int> MyVector{2, 4, 6, 8};  
cout << "MyVector has " << MyVector.size() << " elements\n\n";
```

```
for (int i = 0; i < MyVector.size(); ++i)  
    cout << MyVector[i] << endl;
```

vector

```
9      vector<char>Frog;
(gdb)
10     vector<float>Toad(7);
(gdb)
11     vector<bool>Cat{0,1,1,true,0};
(gdb)
13     cout << Frog.size() << endl;
(gdb)
0
14     cout << Toad.size() << endl;
(gdb)
7
15     cout << Cat.size() << endl;
(gdb)
5
```

vector

We can copy a vector by creating a new vector and initializing it to the vector we want to copy.

```
vector<bool>Cat{0,1,1,true,0};  
vector<bool>Dog{Cat};
```

```
11      vector<bool>Cat{0,1,1,true,0};  
(gdb)  
13      vector<bool>Dog{Cat};  
  
(gdb) p Dog  
$1 = std::vector<bool> of length 5, capacity 64 = {0, 1,  
1, 1, 0}  
(gdb) p Cat  
$2 = std::vector<bool> of length 5, capacity 64 = {0, 1,  
1, 1, 0}
```

vector

We can copy a vector by using the =.

```
11      vector<bool>Cat{0,1,1,true,0};  
(gdb)  
12      vector<bool>Rat;  
(gdb)  
14      vector<bool>Dog{Cat};  
(gdb)  
16      Rat = Cat;  
(gdb) p Cat  
$1 = std::vector<bool> of length 5, capacity 64 = {0, 1, 1, 1, 0}  
(gdb) p Rat  
$2 = std::vector<bool> of length 0, capacity 0  
(gdb) p Dog  
$3 = std::vector<bool> of length 5, capacity 64 = {0, 1, 1, 1, 0}  
(gdb) p Rat  
$4 = std::vector<bool> of length 5, capacity 64 = {0, 1, 1, 1, 0}
```

vector

We can compare two vectors of the same type

```
vector<bool>Cat{0,1,1,true,0};
```

```
vector<bool>Dog{Cat};
```

```
if (Cat == Dog)
    cout << "equal";
else
    cout << "not equal";
```

```
Cat[3] = false;
```

```
if (Cat == Dog)
    cout << "equal";
else
    cout << "not equal";
```



vector

We cannot compare two vectors of the different types

```
student@cse1325:/media/sf_VM$ make
g++ -c -g -std=c++11 tputresetDemo.cpp -o tputresetDemo.o
tputresetDemo.cpp: In function 'int main()':
tputresetDemo.cpp:16:10: error: no match for 'operator==' (operand types are 'std::vector<bool>' and 'std::vector<char>')
 if (Cat == Frog)
 ~~~~~^~~~~~
```

vector

We can use [] just like arrays to access and set individual elements.

```
11         vector<bool>Cat{0,1,1,true,0};  
(gdb)  
13         cout << Cat[2] << endl;  
(gdb) p Cat[2]  
$1 = true  
(gdb) n  
1  
15         if (Cat[2])  
(gdb)  
16             Cat[2] = 0;  
(gdb)  
18         cout << Cat[2] << endl;  
(gdb) p Cat[2]  
$2 = false  
(gdb) n  
0
```

```
vector<bool>Cat{0,1,1,true,0};  
  
cout << Cat[2] << endl;  
  
if (Cat[2])  
    Cat[2] = 0;  
  
cout << Cat[2] << endl;
```

vector

Just like in C, the [] operator will let us walk over memory.

```
11          vector<bool>Cat{0,1,1,true,0};  
(gdb)  
(gdb) p Cat  
$1 = std::vector<bool> of length 5, capacity 64 = {0, 1, 1, 1, 0}  
  
13          Cat[20] = 1;  
  
(gdb) p Cat  
$2 = std::vector<bool> of length 5, capacity 64 = {0, 1, 1, 1, 0}  
  
14          cout << Cat[20] << endl;  
1
```

vector

push_back()

- member function of vector (like size())
- adds a new element to the end of the vector

```
vector<int> MyVector = {2, 4, 6, 8};
```

```
MyVector.push_back(10);  
MyVector.push_back(12);
```

push_back()

2 4 6 8

The size of MyVector is 4 and the capacity of MyVector is 4

MyVector after push_back(10) MyVector.push_back(10);

2 4 6 8 10

```
for (int x : MyVector)  
    cout << x << "\t";
```

The size of MyVector is 5 and the capacity of MyVector is 8

MyVector after push_back(12) MyVector.push_back(12);

2 4 6 8 10 12

The size of MyVector is 6 and the capacity of MyVector is 8

```
vector<int> MyVector = {2,4,6,8};

for (int x : MyVector)
    cout << setw(5) << x;

cout << "\nMyVector.size() " << MyVector.size()
    << " MyVector.capacity() " << MyVector.capacity() << endl;
```

2 4 6 8

MyVector.size() 4 MyVector.capacity() 4

capacity()

Vectors may allocate extra capacity

When a vector is resized, the vector may allocate more capacity than is needed.

This is done to provide some “breathing room” for additional elements, to minimize the number of resize operations needed.

Allows the vector to not need to reallocate every time a push_back is done.

vector

Vector subscripts and at () are based on size/length, not capacity

The range for the subscript operator ([]) and at() function is based on the vector's length, not the capacity.

If a vector has a size/length 3 and a capacity 5, then what happens if we try to access the array element with index 4?

It fails since 4 is greater than the length of the vector.

```
MyVector.push_back(10);

cout << "\n\nMyVector after push_back(10)" << endl;

for (int x : MyVector)
    cout << x << "\t";

cout << "MyVector.size() " << MyVector.size()
    << " MyVector.capacity() " << MyVector.capacity() << endl;
```

```
MyVector after push_back(10)
2      4      6      8      10
MyVector.size() 5 MyVector.capacity() 8
```

```
MyVector.push_back(12);

cout << "\n\nMyVector after push_back(12)" << endl;

for (int x : MyVector)
    cout << x << "\t";

cout << "MyVector.size() " << MyVector.size()
    << " MyVector.capacity() " << MyVector.capacity() << endl;
```

```
MyVector after push_back(12)
2      4      6      8      10     12
MyVector.size() 6 MyVector.capacity() 8
```

2	4	6	8	10	12
---	---	---	---	----	----

```
cout << "The 1st element is " << MyVector.front() << endl;  
cout << "The last element is " << MyVector.back() << endl;  
cout << "The 3rd element is " << MyVector.at(3) << endl;
```

The 1st element is 2
The last element is 12
The 3rd element is 8

front() and back()

Vector member function front() returns the value stored in the first element of the vector

Vector member function back() returns the value stored in the last element of the vector.

```
vector <float> Bank{1.2345,2.3456,3.4567,4.5678,5.6789};
```

```
cout << fixed << Bank.front() << endl  
<< scientific << Bank.back() << endl;
```

1.234500

5.678900e+00

```
MyVector.pop_back();

cout << "\n\nMyVector after pop_back()" << endl;

for (int x : MyVector)
    cout << x << "\t";

cout << "MyVector.size() " << MyVector.size()
    << " MyVector.capacity() " << MyVector.capacity() << endl;
```

```
MyVector after pop_back()
2   4   6   8   10
MyVector.size() 5 MyVector.capacity() 8
```

pop_back()

Vector member function `pop_back()` removes the last element of the vector.

```
vector <float> Bank{1.2345,2.3456,3.4567,4.5678,5.6789};
```

```
Bank.pop_back();
```

```
for (float it : Bank)  
    cout << it << setw(7);
```

```
1.2345 2.3456 3.4567 4.5678 5.6789
```

```
size = 5 and capacity = 5
```

```
1.2345 2.3456 3.4567 4.5678
```

```
size = 4 and capacity = 5
```

```
cout << "size = " << Bank.size()  
    << " and capacity = "  
    << Bank.capacity() << endl;
```

pop_back()

1.2345 2.3456 3.4567 4.5678 5.6789

size = 5 and capacity = 5

1.2345 2.3456 3.4567 4.5678

size = 4 and capacity = 5

1.2345 2.3456 3.4567

size = 3 and capacity = 5

1.2345 2.3456

size = 2 and capacity = 5

1.2345

size = 1 and capacity = 5

size = 0 and capacity = 5

size = 0 and capacity = 5

Note that the capacity did not change – did not shrink

```
for (float it : Bank)  
    cout << it << setw(7);  
  
cout << "size = " << Bank.size()  
    << " and capacity = "  
    << Bank.capacity() << endl;  
  
Bank.pop_back();
```

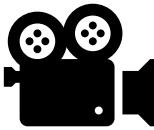
How are we popping an empty vector?

student@cse1325: /media/sf_VM

- + ×

File Edit Tabs Help

student@cse1325:/media/sf_VM\$ tput █



erase()

```
MyVector.erase(MyVector.begin()+1);

cout << "\n\nMyVector after erase()" << endl;

for (int x : MyVector)
    cout << x << "\t";

cout << "\nMyVector.size() " << MyVector.size()
    << " MyVector.capacity() " << MyVector.capacity() << endl;
```

```
MyVector after erase()
2     6     8     10
MyVector.size() 4 MyVector.capacity() 8
```

begin() vs front()

```
MyVector.erase(MyVector.begin()+1);
MyVector.erase(MyVector.front()+1);
```

```
vector3Demo.cpp: In function ‘int main()’:
vector3Demo.cpp:53:35: error: no matching function for call to
‘std::vector<int>::erase(__gnu_cxx::__alloc_traits<std::allocator<int> >::value_type)’
    MyVector.erase(MyVector.front()+1);
```

erase(const_iterator __position)
^~~~~~

```
/usr/include/c++/7/bits/stl_vector.h:1179:7: note:    no known
conversion for argument 1 from
‘__gnu_cxx::__alloc_traits<std::allocator<int> >::value_type {aka
int}’ to ‘std::vector<int>::const_iterator _Alloc>::const_iterator =
```

begin() vs front()

Definition of begin()

```
const_iterator begin() const noexcept;
```

Returns an iterator pointing to the first element in the vector.

Definition of front()

```
const_reference front() const;
```

Returns a reference pointing to the first element in the vector.

Definition of erase()

```
iterator erase (const_iterator position);
```

position

Iterator pointing to a single element to be removed from the vector.

at()

at(n) returns a reference to the element at position n in the vector

```
vector <string> States{"Indiana", "Oklahoma", "Texas";  
cout << States.at(2);
```

Texas

Remember that we start counting at 0

at()

```
vector <string> States{"Indiana", "Oklahoma", "Texas";  
  
cout << "The list of states" << endl;  
  
for (i = 0; i < 3; i++)  
{  
    cout << i+1 << ". " << States[i+1] << "\t";  
}
```

The list of states
Segmentation fault (core dumped)

at()

```
vector <string> States{"Indiana", "Oklahoma", "Texas"};
```

```
cout << "The list of states" << endl;
```

```
for (i = 0; i < 3; i++)
```

```
{
```

```
    cout << i+1 << ". " << States.at(i+1) << "\t";
```

```
}
```

```
The list of states
```

```
terminate called after throwing an instance of 'std::out_of_range'
```

```
  what(): vector::_M_range_check: __n (which is 3) >= this->size()  
(which is 3)
```

```
Aborted (core dumped)
```

Operations on a vector

size()

capacity()

front()

back()

at(n)

pop_back()

erase(n)

begin(n)

end(n)

vector

So did all this discussion on vectors make you think of something from C?

A stack in C++ can be implemented using a vector and

- `push_back()` pushes an element on the stack
- `back()` returns the value of the top element on the stack
- `pop_back()` pops an element off the stack

Command Line Arguments

Command line arguments are optional string arguments that are passed by the operating system to the program when it is launched.

The program can then use them as input (or ignore them).

Much like function parameters provide a way for a function to provide inputs to another function, command line arguments provide a way for people or programs to provide inputs to a *program*.

Command Line Arguments

Passing command line arguments

Executable programs can be run on the command line by invoking them by name.

```
./Code1_1000074079.e
```

In order to pass command line arguments to a program, we list the command line arguments after the executable name

```
./Code1_1000074079.e FileToRead.txt
```

Command Line Parameters

Running a program with command line parameters

```
./Code1_1000074079.e clp1 clp2 clp3
```

Running a program in debug with command line parameters

```
gdb --args ./Code1_1000074079.e clp1 clp2 clp3
```

```
student@cse1325:/media/sf_VM$ gdb --args clp1Demo.e FileToRead.txt
GNU gdb (Ubuntu 8.1-0ubuntu3) 8.1.0.20180409-git
```

```
Reading symbols from clp1Demo.e...done.
```

```
(gdb) break main
```

```
Breakpoint 1 at 0xb73: file clp1Demo.cpp, line 7.
```

```
(gdb) run
```

```
Starting program: /media/sf_VM/clp1Demo.e FileToRead.txt
```

```
Breakpoint 1, main (argc=2, argv=0x7fffffff138) at clp1Demo.cpp:7
```

```
7    {
```

```
(gdb) p argc
```

```
$1 = 2
```

```
(gdb) p *argv
```

```
$2 = 0x7fffffff440 "/media/sf_VM/clp1Demo.e"
```

```
(gdb) p *argv@argc
```

```
$3 = {0x7fffffff440 "/media/sf_VM/clp1Demo.e",
 0x7fffffff458 "FileToRead.txt"}
```

```
(gdb) p argv[0]
```

```
$4 = 0x7fffffff440 "/media/sf_VM/clp1Demo.e"
```

```
(gdb) p argv[1]
```

```
$5 = 0x7fffffff458 "FileToRead.txt"
```

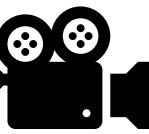
```
#include <iostream>
#include <vector>

int main(int argc, char *argv[])
{
    int i;
    std::vector<std::string> CatNames{};

    for (i = 1; i < argc; i++)
    {
        CatNames.push_back(argv[i]);
    }

    for (auto it : CatNames)
        std::cout << it << "\t";

    return 0;
}
```



File Edit Tabs Help

student@cse1325: /media/sf_VM

- + ×

student@cse1325:/media/sf_VM\$./clp1Demo.e Shade Appa Sylvester Josie



student@cse1325:/me...

Screenshot.txt

[Software Updater]



23:03

```
1 // ICQ 4  
2  
3 #|
```

I

Default Arguments

A **default argument** is a default value provided for a function parameter.

If the user does not supply an explicit argument for a parameter with a default argument, the default value will be used.

If the user does supply an argument for the parameter, the user-supplied argument is used.

Default Arguments

If a function is repeatedly invoked with the same argument value for a particular parameter, then

you can specify that such a parameter has a default argument

Default argument – a default value is passed to that parameter

When a program omits an argument for a parameter with a default argument in a function call, the compiler rewrites the function call and inserts the default value of that argument.

```
unsigned int boxVolume(unsigned int length=1, unsigned int width=1, unsigned int height=1)
{
    return length * width * height;
}

int main()
{
    // nothing is passed - use defaults for all
    cout << "boxVolume() = " << boxVolume() << endl;

    // length is passed - use default width and height
    cout << "\n\nboxVolume(10) = " << boxVolume(10) << endl;

    // length and width are passed - use default height
    cout << "\n\nboxVolume(10,5) = " << boxVolume(10,5) << endl;

    // length and width and height are all passed - no defaults
    cout << "\n\nboxVolume(10,5,2) = " << boxVolume(10,5,2) << endl;

    return 0;
}
```

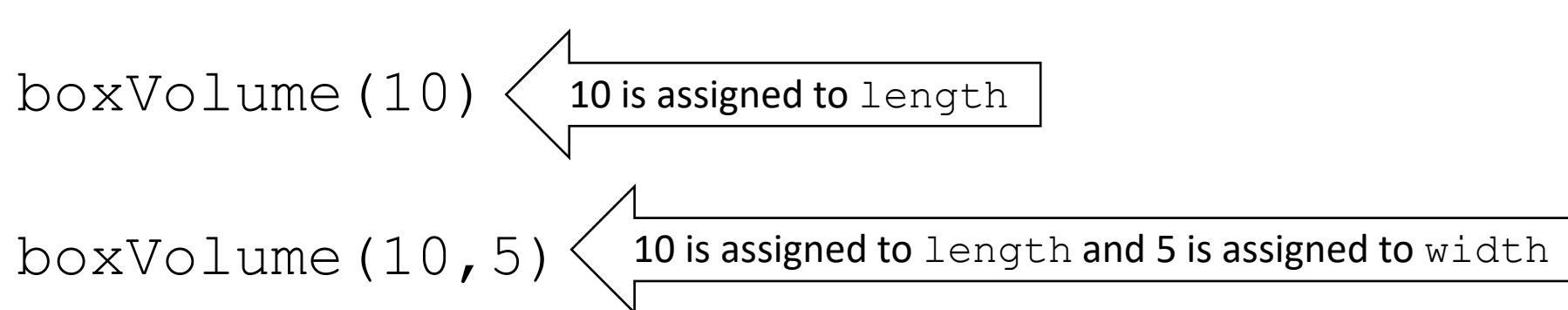
default values

Default Arguments

Any arguments passed to the function explicitly are assigned to the function's parameters from left to right.

`boxVolume`'s parameters (in order) are `length`, `width`, `height`

So when `boxVolume ()` receives one argument, it assigns the value of that argument to its leftmost parameter which is `length`.



Default Arguments

Default values need to be specified in EITHER the prototype or the function **BUT** not both.

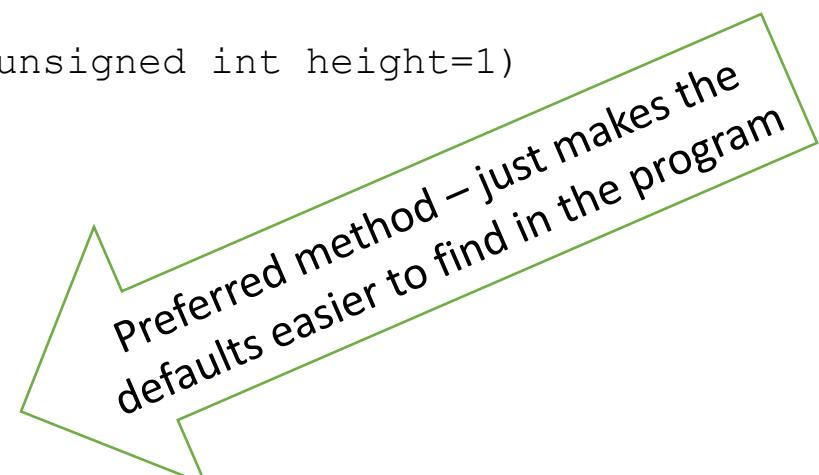
```
unsigned int boxVolume(unsigned int length, unsigned int width, unsigned int height);
```

```
unsigned int boxVolume(unsigned int length=1, unsigned int width=1, unsigned int height=1)
{
    return length * width * height;
}
```

OR

```
unsigned int boxVolume(unsigned int length = 1, unsigned int width = 1, unsigned int height = 1);
```

```
unsigned int boxVolume(unsigned int length, unsigned int width, unsigned int height)
{
    return length * width * height;
}
```



Preferred method – just makes the
defaults easier to find in the program

Default Arguments

```
unsigned int boxVolume(unsigned int length=1,  
                      unsigned int width=1, unsigned int height=1);
```

```
unsigned int boxVolume(unsigned int length=1,  
                      unsigned int width=1, unsigned int height=1)  
{  
    return length * width * height;  
}
```

Default Arguments

All default arguments must be for the rightmost parameters.

```
// Want to use default length but pass in width and height  
cout << "\n\nboxVolume(,5,2) = " << boxVolume(,5,2) << endl;
```

```
student@cse1325:/media/sf_VM$ make  
g++ -c -g -std=c++11 defargDemo.cpp -o defargDemo.o  
defargDemo.cpp: In function 'unsigned int boxVolume(unsigned int, unsigned int,  
unsigned int)':  
defargDemo.cpp:12:90: error: default argument given for parameter 1 of 'unsigned  
int boxVolume(unsigned int, unsigned int, unsigned int)' [-fpermissive]
```

Default Arguments

Default arguments must be the rightmost (trailing) arguments in a function's parameter list.

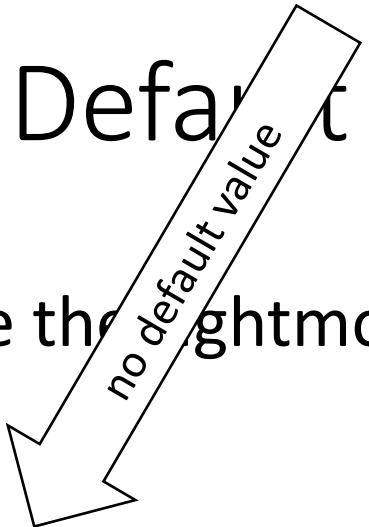
no default value

```
unsigned int boxVolume(unsigned int length=1, unsigned int width=1,unsigned int height)
{
    return length * width * height;
}
```

```
student@cse1325:/media/sf_VM$ make
g++ -c -g -std=c++11 defargDemo.cpp -o defargDemo.o
defargDemo.cpp: In function 'unsigned int boxVolume(unsigned int, unsigned int, unsigned int)':
defargDemo.cpp:13:14: error: default argument missing for parameter 3 of 'unsigned int boxVolume(unsigned int, unsigned int, unsigned int)'
    unsigned int boxVolume(unsigned int length=1, unsigned int width=1, unsigned in
^
makefile:18: recipe for target 'defargDemo.o' failed
make: *** [defargDemo.o] Error 1
```

Default Arguments

Default arguments must be the rightmost (trailing) arguments in a function's parameter list.



```
unsigned int boxVolume(unsigned int length, unsigned int width=1, unsigned int height=1)
{
    return length * width * height;
}
```

OK because length is the leftmost.

```
student@cse1325:/media/sf_VM$ make
g++ -c -g -std=c++11 defargDemo.cpp -o defargDemo.o
g++ -g -std=c++11 defargDemo.o -o defargDemo.e
student@cse1325:/media/sf_VM$ █
```

Default Arguments

Default values can be any expression, including constants, global variables or function calls.

```
#define FROG 2
unsigned int boxVolume(unsigned int length=1, unsigned
int width=FROG, unsigned int height=1)

vector<int>TOAD(4);
unsigned int boxVolume(unsigned int length=1, unsigned
int width=TOAD.size(), unsigned int height=1)
```

Default Arguments

```
#include <iostream>

using namespace std;

void PrintIT(string Word1="University of Texas", string Word2=" at ", string Word3);

void PrintIT(string Word1, string Word2, string Word3)
{
    cout << Word1 << Word2 << Word3;
}

int main()
{
    PrintIT("University of Texas", " at ", "Arlington");
    PrintIT("University of Texas", " at ", "Austin");
    PrintIT("University of Texas", " at ", "Dallas");
    return 0;
}
```

ile? NO

Default Arguments

```
#include <iostream>

using namespace std;

void PrintIT(string Word3, string Word1="University of Texas", string
Word2=" at ");

void PrintIT(string Word3, string Word1, string Word2)
{
    cout << Word1 << Word2 << Word3 << endl;
}

int main()
{
    PrintIT("Arlington");
    PrintIT("Austin");
    PrintIT("Dallas");

    return 0;
}
```

University of Texas at Arlington
University of Texas at Austin
University of Texas at Dallas

Will this compile?

Yes

Default Arguments

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

void PrintIT(string Word3, string Word1="University of Texas", string Word2="at ");

void PrintIT(string Word3, string Word1, string Word2)
{
    cout << Word1 << Word2 << Word3 << endl;
}

int main()
{
    PrintIT("Arlington");
    PrintIT("Austin");
    PrintIT("Dallas");

    return 0;
}
```

University of Texas at Arlington
University of Texas at Austin
University of Texas at Dallas

How to print

University of Tulsa
PrintIT("", "University of Tulsa", "");
Texas A&M University-Commerce
PrintIT("Commerce", "Texas A&M University", "-");
Texas A&M University at Galveston
PrintIT("Galveston", "Texas A&M University");

Function Overloading

Function overloading is a feature of C++ that allows us to create multiple functions with the same name, so long as they have different parameters.

Consider this function...

```
int funA(int X, int Y, int Z)
{
    return X+Y+Z;
}
```

Function Overloading

```
int funA(int X, int Y, int Z)
{
    return X+Y+Z;
}
```

```
int main(void)
{
    cout << funA(2,2,2) << endl;
    cout << funA(3.3,3.3,3.3) << endl;
    return 0;
}
```

This would print “6”

This would print “9”. Why?

Function Overloading

What if we created a function with the same name but used a different type for the parameters?

```
int funA(int X, int Y, int Z)
{
    return X+Y+Z;
```

Prints “6”

```
double funA(double X, double Y, double Z)
{
    return X+Y+Z;
```

Prints “9.9”

Function Overloading

Function overloading is a feature of C++ that allows us to create multiple functions with the same name, so long as they have different parameters.

The C++ compiler selects the proper function to call by examining the number, types and order of the arguments in the call.

The combination of a function's name and its parameters types and the order of them is called a **signature**.

Function Overloading

```
int funA(int X, int Y, int Z) {  
    return X+Y+Z;  
}
```

signature – funA+int+int+int

```
double funA(double X, double Y, double Z) {  
    return X+Y+Z;  
}
```

signature – funA+double+double+double

```
int main(void)  
{  
    cout << funA(2,2,2) << endl;  
    cout << funA(3.3,3.3,3.3) << endl;  
  
    return 0;  
}
```

```
student@cse1325:/media/sf_VM$ make  
g++ -c -g -std=c++11 funplus1Demo.cpp -o funplus1Demo.o  
g++ -g -std=c++11 funplus1Demo.o -o funplus1Demo.e  
student@cse1325:/media/sf_VM$ ./funplus1Demo.e  
6  
9.9
```

Function Overloading

What is the signature of each of these functions?

```
std::string displayMoney(int amount)  
    signature = displayMoney+int
```

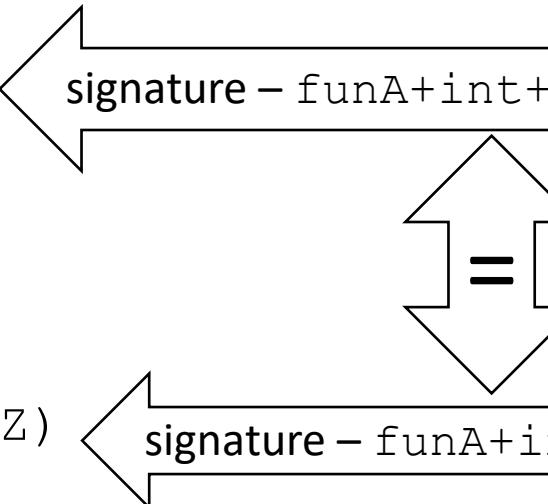
```
bool buyPencils(int payment, std::string& change, int& action,  
const int &quantity, int &inventoryLevel, int &changeLevel)  
    signature=buyPencils+int+string+int+const int+int+int
```

```
int PencilMenu()  
    signature = PencilMenu
```

Function Overloading

Creating overloaded functions with identical parameter lists and different return types is a compilation error.

```
int funA(int X, int Y, int Z) <----- signature - funA+int+int+int  
{  
    return X*Y*Z;  
}  
  
double funA(int X, int Y, int Z) <----- signature - funA+int+int+int  
{  
    return X*Y*Z;  
}
```



```
student@cse1325:/media/sf_VM$ make  
g++ -c -g -std=c++11 funoverDemo.cpp -o funoverDemo.o  
funoverDemo.cpp: In function 'double funA(int, int, int)':  
funoverDemo.cpp:13:32: error: ambiguating new declaration of 'double funA(int, i  
nt, int)'  
    double funA(int X, int Y, int Z)  
  
funoverDemo.cpp:7:5: note: old declaration 'int funA(int, int, int)'  
    int funA(int X, int Y, int Z)
```

Function Overloading

Function overloading can lower a program's complexity significantly while introducing very little additional risk.

Function overloading typically works transparently and without any issues.

The compiler will flag all ambiguous cases, and they can generally be easily resolved.

Conclusion : function overloading can make your program simpler.

Function Overloading and Default Arguments

A function with default arguments omitted might be called identically to another overloaded function.

```
int funA(void)
{
    return 3;
}
```

```
cout << funA() << endl;
cout << funA() << endl;
```

```
student@cse1325:/media/sf_VM$ make
g++ -c -g -std=c++11 funover2Demo.cpp -o funover2Demo.o
funover2Demo.cpp: In function 'int main()':
funover2Demo.cpp:20:15: error: call of overloaded 'funA()' is ambiguous
    cout << funA() << endl;
                                         ^
funover2Demo.cpp:7:5: note: candidate: int funA()
    int funA(void)
                         ^
funover2Demo.cpp:12:8: note: candidate: double funA(double, double, double)
    double funA(double X=3, double Y=3, double Z=3)
                           ^
makefile:18: recipe for target 'funover2Demo.o' failed
make: *** [funover2Demo.o] Error 1
```

```
double funA(double X=3, double Y=3, double Z=3)
{
    return X*Y*Z;
}
```

```
cout << funA(3) << endl;
cout << funA(3) << endl;
27
27
```

Function Overloading and Default Arguments

Functions with default arguments can be overloaded.

```
void print(string MyName);
```

```
void print(char MyInitial='F');
```

```
print();  
print("French");
```

Initial F
Name French

```
void print(string MyName)  
{  
    cout << "Name " << MyName << endl;  
}  
  
void print(char MyInitial)  
{  
    cout << "Initial " << MyInitial << endl;  
}
```

Function Overloading and Default Arguments

It is important to note that default arguments do NOT count towards the parameters that make the function's signature.

```
void print(char Initial1='D', char Initial2='M', char Initial3='F');  
void print(char MyInitial='F');
```

```
student@cse1325:/media/sf_VM@ make  
g++ -c -g -std=c++11 defover2Demo.cpp -o defover2Demo.o  
defover2Demo.cpp: In function ‘int main()’:  
defover2Demo.cpp:23:8: error: call of overloaded ‘print()’ is ambiguous  
    print();  
           ^
```

Function Overloading and Default Arguments

```
void print(char Initial1, char Initial2='M', char Initial3='F');  
void print(char MyInitial='F');
```



```
void print(char Initial1, char Initial2='M', char Initial3='F');  
void print(char MyInitial);
```



```
void print(char Initial1, char Initial2, char Initial3='F');  
void print(char MyInitial='F');
```



```
print('F');
```

File Edit Tabs Help

```
student@cse1325:/media/sf_VM$ make
g++ -c -g -std=c++11 defover1Demo.cpp -o defover1Demo.o
defover1Demo.cpp:8:27: error: stray '\342' in program
void print(char MyInitial=????F');
^

defover1Demo.cpp:8:28: error: stray '\200' in program
void print(char MyInitial=????F');
^

defover1Demo.cpp:8:29: error: stray '\230' in program
void print(char MyInitial=????F');
^

defover1Demo.cpp:8:31: error: stray '\342' in program
void print(char MyInitial='F????');
^

defover1Demo.cpp:8:32: error: stray '\200' in program
void print(char MyInitial='F????');
^

defover1Demo.cpp:8:33: error: stray '\231' in program
void print(char MyInitial='F????');
```

```
1 // default argument function override 1 - DemoCRLF  
2 CRLF  
3 #include <iostream>CRLF  
4 CRLF  
5 using namespace std;CRLF  
6 CRLF  
7 void print(string MyName);CRLF  
8 void print(char MyInitial='F');CRLF  
9 CRLF  
10 CRLF  
11 int main() CRLF I  
12 {CRLF  
13     →print();CRLF  
14     →print("French");CRLF  
15     →CRLF  
16     →return 0;CRLF  
17 }CRLF  
18 CRLF  
19
```

The quotes around the F

Function Templates

Overloaded functions are normally used to perform *similar* operations that involve *different* program logic on different data types.

If the program logic and operations are *identical* for each data type, overloading may be performed more compactly and conveniently by using **function templates**.

You write a single function template definition.

Given the argument types provided in calls to your function, C++ automatically generates separate **function template specializations** to handle each type of call appropriately.

```
int int1, int2, int3;
double double1, double2, double3;
char char1, char2, char3;

// call maximum with int
cout << "Input three integer values: ";
cin >> int1 >> int2 >> int3;
cout << "The max integer value is: " << maximum(int1, int2, int3);

// call maximum with double
cout << "\n\nInput three double values: ";
cin >> double1 >> double2 >> double3;
cout << "The max double value is: " << maximum(double1, double2, double3);

// call maximum with char
cout << "\n\nInput three characters: ";
cin >> char1 >> char2 >> char3;
cout << "The max char value is: " << maximum(char1, char2, char3) << endl;
```

keyword

```
template <typename T>
T maximum( T value1, T value2, T value3)
{
    T maximumValue{value1}; // assume value1 is maximum

    if (value2 > maximumValue)
    {
        maximumValue = value2;
    }

    if (value3 > maximumValue)
    {
        maximumValue = value3;
    }

    return maximumValue;
}
```

<typename T>

is the template parameter which represents a type that has not yet been specified

Function name maximum returns type T

value1, value2 and value3 are of type T

maximumValue is of type T

Function Templates

The compiler creates a separate function definition for each

one expecting three int values

function template specialization for type int

one expecting three double values

function template specialization for type double

one expecting three char values

function template specialization for type char

Function template specialization for type int

```
int maximum(int value1, int value2, int value3)
{
    int maximumValue{value1}; // assume value1 is maximum

    if (value2 > maximumValue)
    {
        maximumValue = value2;
    }

    if (value3 > maximumValue)
    {
        maximumValue = value3;
    }

    return maximumValue;
}
```

7

CSE 1325

Week of 10/05/2020

Instructor : Donna French

Object
Oriented
Programming

Intro to OOP

Going forward...

We are still learning how to program in C++

We will now add studying OO concepts and how to apply them in C++

Other OO languages will have ways of doing almost everything we will do – you are here to learn how to do them in C++

Intro to OOP

- CSE 1310
 - Teaching you how to program
 - Get you in the mindset of a programmer
- CSE 1320
 - Learning more advanced programming
 - What goes on behind the scenes (memory, pointers, debugging) using C
- CSE 1325
 - Learning a different type of programming
 - Objects, inheritance, encapsulation using C++

CSE 1310 and 1320 focused on procedural programming.
CSE 1325 will focus on OO programming

OOP

In traditional programming, programs are basically lists of instructions to the computer that define data and then work with that data.

Data and the functions that work on that data are separate entities that are combined together to produce the desired result.

Because of this separation, traditional programming often does not provide a very intuitive representation of reality.

OOP

It's up to the programmer to manage and connect the properties (variables) to the behaviors (functions) in an appropriate manner. This leads to code that looks like this:

```
driveTo(you, work);
```

A function called `driveTo` that takes `you` and `work` as parameters.

OOP

Object-oriented programming (OOP) provides us with the ability to create objects that tie together both properties and behaviors into a self-contained, reusable package.

This leads to code that looks more like this:

```
you.driveTo(work);
```

You have the ability to drive to work...

OOP

Rather than being focused on writing functions, we focus on defining objects that have a well-defined set of behaviors.

This is why the paradigm is called “object-oriented”.

OOP allows programs to be written in a more modular fashion, which makes them easier to write and understand, and also provides a higher degree of code-reusability.

Simple Analogy – The Automobile as an Object

Suppose you want to drive a car and make it go faster by pressing its accelerator pedal.

What must happen before you can do this?

Step 1 - *design* the car

Simple Analogy – The Automobile as an Object

A car typically begins as engineering drawings, similar to the *blueprints* that describe the design of a house. A **class** is like a blueprint in that it is the template for any object created from it.

These drawings include the design for an accelerator pedal. The pedal *hides* from the driver the complex mechanisms that actually make the car go faster, just as the brake pedal hides the mechanisms that slow the car, and the steering wheel *hides* the mechanisms that turn the car.

This enables people with little or no knowledge of how engines, braking and steering mechanisms work to drive a car easily.

Simple Analogy – The Automobile as an Object

Before you can drive a car, it must be *built* from the engineering drawings that describe it.

A completed car has an *actual* gas pedal to make the car go faster.

But the car won't accelerate on its own, so the driver must *press* the gas pedal to accelerate the car.

Simple Analogy – The Automobile as an Object

Gas pedal in our automobile

 hides the mechanisms of making the car go faster from the driver

Brake pedal

 hides the mechanisms of making the car stop from the driver

Function (also called a Method)

- houses the program statements that actually perform a task
- hides these statements from its user

Member functions

`get()`, `put()`, `getline()`, `size()`, `str()`

Simple Analogy – The Automobile as an Object

A car must be built from its drawings/blueprints before you can drive it.

An *object* must be built from a class before it can be used.

This building process is called *instantiation*.

An object is an instance of its class.

A class can be used many times to build many objects (more than one car is built from a drawing)

Simple Analogy – The Automobile as an Object

Pressing on the gas pedal sends a *message* to the car.

Pressing on the brake sends a different *message* to the car resulting in a different action happening.

Calling a member function is sending a *message* to an object.

Simple Analogy – The Automobile as an Object

What color is your car? What make is your car? What year is your car?

How many miles are on your car? How much gas is in the tank?

These are all *attributes* of your car. Every car not only has functions (gas pedal, brake pedal) but each one also has a unique set of *attributes*. My car knows how many miles are on it, but it does not know how many miles are on the car sitting next to it.

An object's attributes are defined in its class's **data members**.

Car

Attributes?

Functions?

Pencil Machine

Attributes

inventoryLevel

changeLevel

pencilCost

Functions

buyPencil

displayMoney

showInventoryLevel

showChangeLevel

```
struct DateStruct
{
    int year;
    int month;
    int day;
};
```

Today is 2019/9/23
Halloween is 2019/10/31

```
14 void print(DateStruct date)
15 {
16     cout << date.year << "/"
17     << date.month << "/" << date.day;
18 }
19
20 int main()
21 {
22     DateStruct today{2019,9,23};
23
24     cout << "Today is ";
25     print(today);
26     cout << endl;
27
28     today.month = 10;
29     today.day = 31;
30
31     cout << "Halloween is ";
32     print(today);
33     cout << endl;
34
35     return 0;
36 }
```

Why was the choice made to put the endl on a line by itself rather than inside the print() function?

Class

In the world of object-oriented programming, we want our types to not only hold data, but provide functions that work with the data as well.

In C++, this is typically done via the **class** keyword.

Using the **class** keyword defines a new user-defined type called a class.

Struct vs Class

```
struct DateStruct  
{  
    int year;  
    int month;  
    int day;  
};
```

```
class DateStruct  
{  
public :  
    int year;  
    int month;  
    int day;  
};
```

Class

```
class DateClass
{
public :
    int year;
    int month;
    int day;

void print()
{
    cout << year << "/"
        << month << "/"
        << day;
}

};
```

```
int main()
{
    DateClass today{2019, 9, 23};

    cout << "Today is ";
    today.print();
    cout << endl;

    today.month = 10;
    today.day = 31;

    cout << "Halloween is ";
    today.print();
    cout << endl;

    return 0;      Today is 2019/9/23
}                      Halloween is 2019/10/31
```

Class vs Struct

The struct version of the program is pretty similar to the class version.

However, there are a few SIGNIFICANT differences.

In the DateStruct version of print (), the struct itself was passed to the print () function as the first parameter. Otherwise, print () wouldn't know which DateStruct to use. The function then had to reference that parameter inside the function explicitly.

```
DateStruct today{2019,9,23};  
print(today);  
cout << date.year << "/" << date.month << "/" << date.day;
```

Class vs Struct

```
void print()
{
    cout << year << "/" << month << "/" << day;
}
```

Member functions work slightly differently - all member function calls must be associated with an object of the class.

When “today.print ()” is called, the compiler calls the print () member function that is associated with the today object.

So when we call “today.print ()”, the compiler interprets day as today.day, month as today.month, and year as today.year.

If we called “tomorrow.print ()”, day would refer to tomorrow.day instead.

```
int main()
{
    DateClass today{2019, 9, 23};
    DateClass tomorrow{2019, 9, 24};

    cout << "Today is ";
    today.print();
    cout << endl;

    cout << "Tomorrow is ";
    tomorrow.print();
    cout << endl;

    return 0;
}
```

Today is 2019/9/23
Tomorrow is 2019/9/24

Class vs Struct

The associated object is essentially implicitly passed to the member function.

For this reason, it is often called the implicit object.

The key point is that with non-member functions, we have to pass data to the function to work with.

With member functions, we can assume we always have an implicit object of the class to work with.

C++ Structs vs C Structs

C++ Struct

- default access is public
- can have member functions
- can directly initialize structure members
- can use static
- can have a constructor
- `sizeof()` empty struct will be 1

C Struct

- default access is public
- no member functions
- cannot directly initialize structure members
- cannot use static
- cannot have a constructor
- `sizeof()` empty struct will be 0

C++ Class vs C++ Structs

C++ Class

- members of a class are private by default
- when deriving a class, default access specifier is private.

C++ Struct

- members of a struct are public by default
- default access-specifier for a struct is public.

C++ Classes vs C Structs

C++ Class

- default access is private
- constructor/destructor run automatically
- member functions
- can use operator overloading
- inheritance

C Struct

- default access is public
- must be called explicitly
- no member functions

Intro to OOP

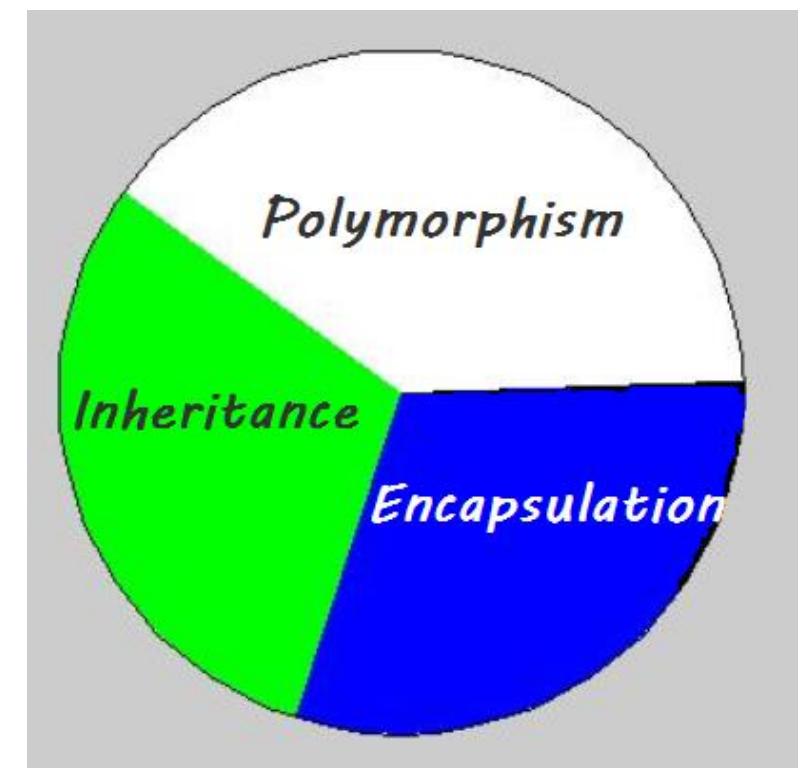
Three main concepts of Object Oriented Programming

OO PIE

Polymorphism

Inheritance

Encapsulation



OOP Vocabulary

Polymorphism

describes the ability of different objects to be accessed by a common interface

Inheritance

describes the ability of objects to derive behavior patterns from other objects

These objects can then customize and add new unique characteristics

Encapsulation

describes the ability of objects to maintain their internal workings independently from the program they are used in

Classes **encapsulate** attributes and member functions into objects created from those classes

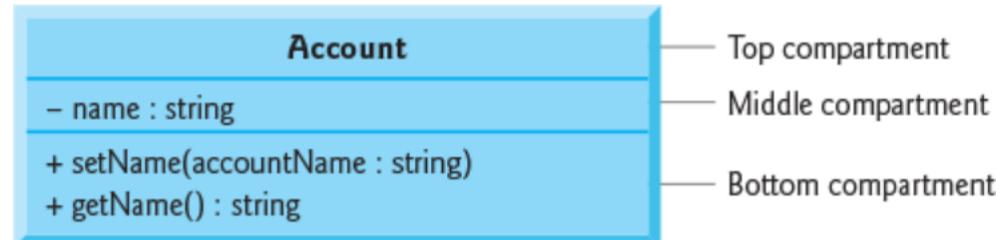
OOP Vocabulary

UML

Unified Modeling Language

widely used graphical scheme for modeling object-oriented systems

We will focus only on class diagrams



Creating a Bank Account Class

Class Account

Data Members

name

balance

Member functions

getBalance

depositFunds

withdrawFunds

Each class you create becomes a new type you can use to create objects.

```
#include <iostream>
#include <string>
#include "Account.h"

using namespace std;

int main()

    string NewAccountName;

    Account MyBankAccount;

    cout << "My bank account's name is " << MyBankAccount.getName();

    cout << "\nEnter a new name for the bank account ";
    getline(cin, NewAccountName);
    MyBankAccount.setName(NewAccountName);

    cout << "My bank account has been renamed " << MyBankAccount.getName() << endl;

    return 0;
}
```

Include file for file
containing class
definition

Instantiate the object
MyBankAccount from class
Account

Member
function
getName()

Member function setName()

Member
function
getName()

TestBankAccount.cpp

```
class Account
{
public :
    void setName(std::string accountName)
    {
        name = accountName;
    }

    std::string getName() const
    {
        return name;
    }

private :
    std::string name;
};
```

using namespace std;

is not used inside classes – using it in a class
would force its usage in any program using the
class – not the job of a class

Class definition

```
class ClassName
{  
};
```

Each class gets its own header file.

```
class Account
{
public :
    void setName(std::string accountName)
    {
        name = accountName;
    }

    std::string getName() const
    {
        return name;
    }

private :
    std::string name;
};
```

setName

- member function
- return type of `void`
- one parameter of type `string`
- set data member `name` to the passed in `accountName`

getName

- member function
- return type of `std::string`
- no parameters
- returns data member `name`

Member function `getName()` is declared as `const` because, in the process of returning `name`, the function does not, and should not, modify its object `Account`.

With the `const` in place, the compiler would issue a warning if any code that modified `name` in that function was accidentally added.

```
class Account
{
public :
    void setName(std::string accountName)
    {
        name = accountName;
    }

    std::string getName() const
    {
        return name;
    }
private :
    std::string name;
};
```

name

- data member
- type `std::string`
- default value is empty string ""

Data members are declared inside a class definition but outside the bodies of the class's member functions.

Programming style – list data members at the end of the class to make them easier to find and to group together.

```
class Account
{
public :
    void setName(std::string accountName)
    {
        name = accountName;
    }

    std::string getName() const
    {
        return name;
    }
private :
    std::string name;
};
```

public
private

- keywords
- access specifier
 - always followed by a :

private

Since name appears after the access specifier private, name is only accessible to class Account's member functions.

This is known as **data hiding** —the data member name is *encapsulated* (hidden) and can be used *only* in class Account's setName () and getName () member functions.

Most data member declarations appear after the private : access specifier.

```
class Account
{
public :
    void setName(std::string accountName)
    {
        name = accountName;
    }

    std::string getName() const
    {
        return name;
    }

private :
    std::string name;
};
```

public
private

- keywords
- access specifier
 - always followed by a :

public

Data members or member functions listed after access specifier public (and before the next access specifier if there is one) are “available to the public.” They can be used by other functions in the program (such as `main`), and by member functions of other classes (if there are any).

By default, everything in a class is private, unless you specify otherwise.

Once you list an access specifier, everything from that point has that access until you list another access specifier.

Programming Style

List public only once, grouping everything that's public.

List private only once, grouping everything that's private.

The access specifiers public and private may be repeated, but this is unnecessary and can be confusing.

Making a class's data members private and member functions public makes debugging more productive because problems with data manipulations are localized to the member functions.

An attempt by a function that's not a member of a particular class to access a private member of that class is a compilation error.

```
student@cse1325:/media/sf_VMs$ more makefile
#Donna French
#makefile for C++ program
SRC = TestBankAccount.cpp
OBJ = $(SRC:.cpp=.o)
EXE = $(SRC:.cpp=.e)

CFLAGS = -g -std=c++11

all : $(EXE)

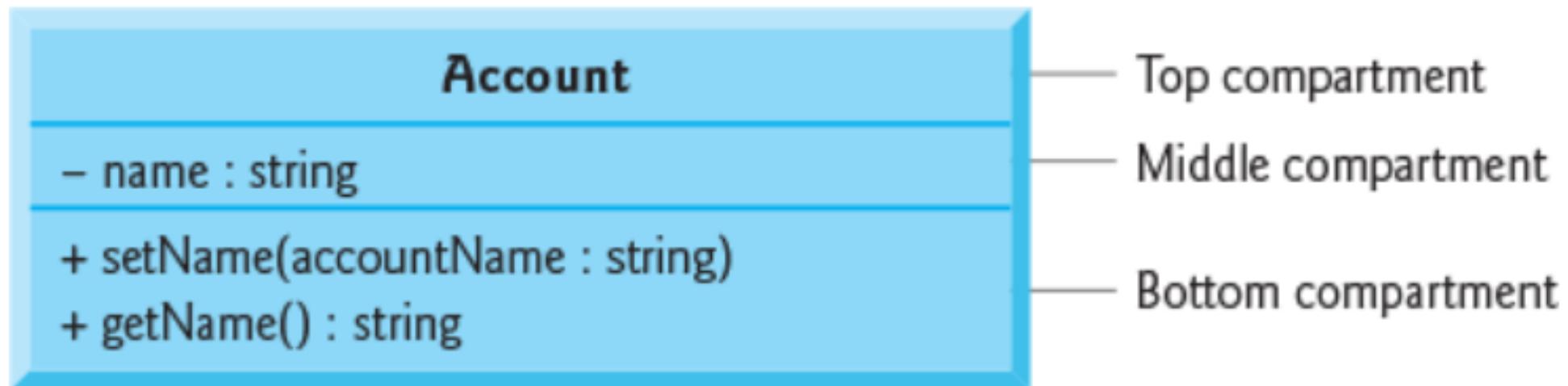
$(EXE): $(OBJ)
        g++ $(CFLAGS) $(OBJ) -o $(EXE)

$(OBJ) : $(SRC)
        g++ -c $(CFLAGS) $(SRC) -o $(OBJ)

student@cse1325:/media/sf_VMs$ make
g++ -c -g -std=c++11 TestBankAccount.cpp -o TestBankAccount.o
g++ -g -std=c++11 TestBankAccount.o -o TestBankAccount.e
student@cse1325:/media/sf_VMs$ ./TestBankAccount.e
My bank account's name is
Enter a new name for the bank account My Bank Account
My bank account has been renamed My Bank Account
student@cse1325:/media/sf_VMs$ █
```

Class Diagram

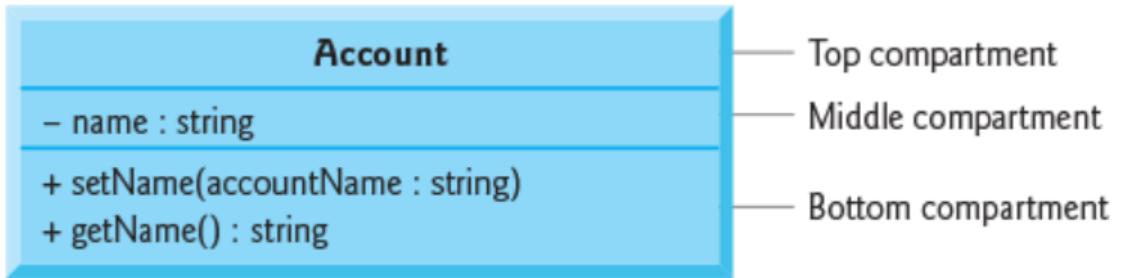
widely used graphical scheme for modeling objects in OOP



```
class Account
{
public :
    void setName(std::string accountName)
    {
        name = accountName;
    }

    std::string getName() const
    {
        return name;
    }

private :
    std::string name;
};
```



Top compartment

Account

Class name
Centered in the compartment

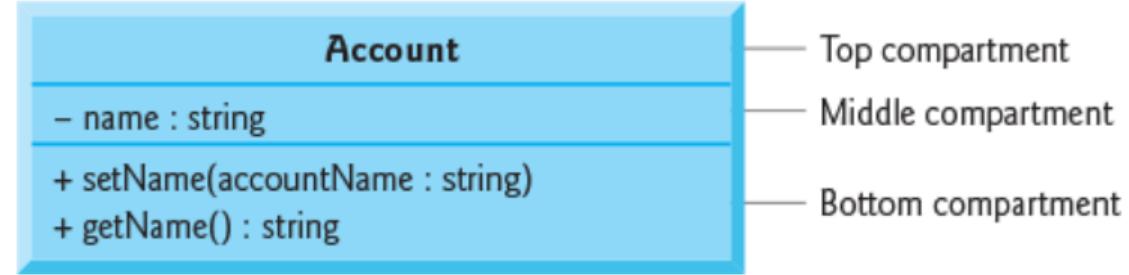
```

class Account
{
public :
    void setName(std::string accountName)
    {
        name = accountName;
    }

    std::string getName() const
    {
        return name;
    }

private :
    std::string name;
};

```



Middle compartment

- name : string
- Class's attribute name
- Minus sign indicates that the attribute is private
- Following the attribute name are a *colon* and the *attribute type*

: string

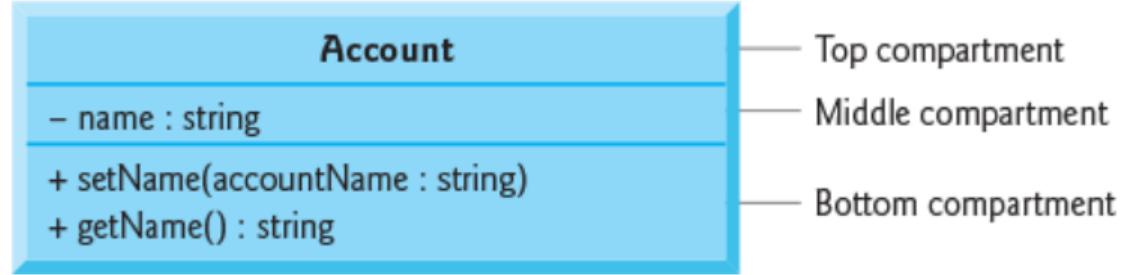
```

class Account
{
public :
    void setName(std::string accountName)
    {
        name = accountName;
    }

    std::string getName() const
    {
        return name;
    }

private :
    std::string name;
};

```



Bottom compartment

Class's operations

Plus sign indicates that the operation is public

The () after the operation name contain

MFN (ParameterName : ParameterType) ReturnType

setName (accountName : string)

getName () : string

No return type - void

No parameters

```
class Account
{
public :
    void setName(std::string accountName)
    {
    }

    std::string getName() const
    {
    }

private :
    std::string name;
};
```

Account

- name : string
- + setName(accountName : string)
- + getName() : string

Circle

- radius : double = 0
+ getRadius() : double
+ setRadius(new_radius : double = 0)
+ calculateCircumference() : double

```
class Circle
{
    public :
        double getRadius()
    {
    }
    void setRadius(double new_radius=0)
    {
    }
    double calculateCircumference()
    {
    }
    private :
        double radius = 0;
};
```

```
class Student
{
    public :
        std::string getStudentID()
        {
        }
        void setStudentID(std::string newStudentID)
        {
        }
        std::string getNetID()
        {
        }
    private :
        std::string studentID;
        std::string netID;
        std::string emailAddress;
};
```

Student

- studentID : string
- netID : string
- emailAddress : string
+ getStudentID() : string
+ setStudentID(newStudentID : string)
+ getNetID() : string
+ setNetID(newNetID : string)
+ getEmailAddress() : string
+ setEmailAddress(newEmailAddress : string)

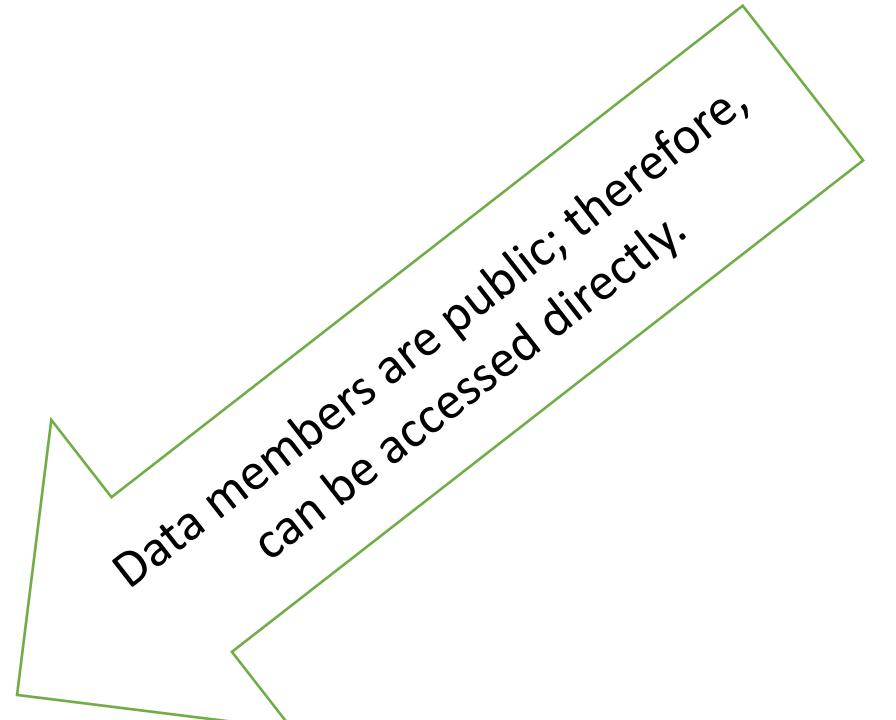
```
class DateClass
{
public :
    int year;
    int month;
    int day;

    void print()
    {
        cout << year << "/"
           << month << "/" << day;
    }
};

int main()
{
    DateClass today{2019,9,25};

    cout << "Today is " << today.month << today.day
       << today.year << endl;
    today.print();
    return 0;
}
```

```
student@cse1325:/media/sf_VM$ ./pvspDemo.e
Today is 9252019
2019/9/25student@cse1325:/media/sf_VM$ █
```



Data members are public; therefore,
can be accessed directly.

```
class Date
{
public:
private:
};

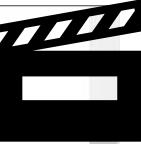
int main()
{
    Date d;
    cout << d;
    today();
    return 0;
}
```

student@cse1325: /media/sf_VM





```
1 // public vs private Demo
2
3 #include <iostream>
4
5 using namespace std;
6
7 class DateClass
8 {
9     public :
10    void print()
11    {
12        cout << year << "/"
13        |   | << month << "/" << day;
14    }
15    void setDate(int mnth, int dy, int yr)
16    {
17        year = yr;
18        month = mnth;
19        day = dy;
20    }
21     private :
22        int year;
23        int month;
24        int day;
25    };
26
27 int main()
28 {
```



```
4
5 using namespace std;
6
7 class DateClass
8 {
9     public :
10    void print()
11    {
12        cout << year << "/" 
13                    << month << "/" << day;
14    }
15    void setDate(int mnth, int dy, int yr)
16    {
17        year = yr;
18        month = mnth;
19        day = dy;
20    }
21    private :
22        int year;
23        int month;
24        int day;
25    };
26
27 int main()
28 {
29     DateClass today;
30
31     today.setDate(9,25,2019);
```

```
class PVSP
{
    public :
        int x;
        char y;
        double z;
    private :
        string a;
        float b;
        long c;
};
```

```
int main ()
{
    PVSP QuizMe;
    QuizMe.x = 1;
    QuizMe.a = "Quiz";
    QuizMe.y = 'A';
    QuizMe.b = 1.23;
    QuizMe.z = 1.23;
    QuizMe.c = 123;
    return 0;
}
```

Which lines will
compile?

Encapsulation

You will begin to notice that objects tend to have mostly private data members and public member functions.

Why?

Take the example of the electronic devices that surround us every day.

They have simple interfaces that allows you to perform actions without know the details behind those actions.

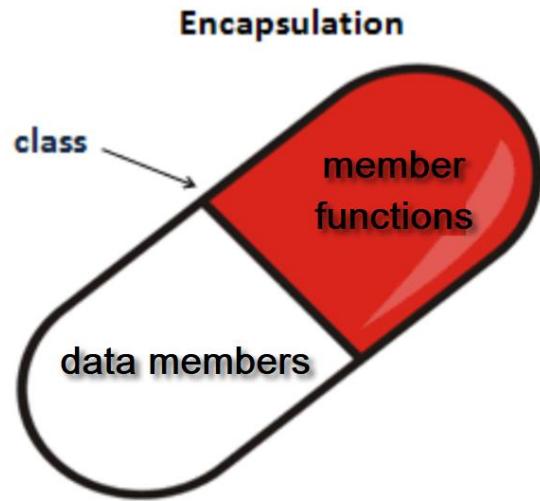
Encapsulation

The separation of interface and implementation is extremely useful because it allows us to use objects without understanding how they work.

This vastly reduces the complexity of using these objects and increases the number of objects we are capable of interacting with.

This same principle is applied to programming – separating implementation from interface.

Generally, data members are made private (hiding implementation details) and member functions are made public (giving the user an interface).



Encapsulation

Classes wrap attributes and member functions into objects created from those classes – an object's attributes and member functions are intimately related.

Objects may communicate with one another, but they are not normally allowed to know how other objects are implemented.

Encapsulation is the technique of information hiding - implementation details are hidden within the objects themselves.

Encapsulation

Benefits of Encapsulation

Encapsulated classes are easier to use and reduce the complexity of programs.

Encapsulated classes help protect your data and prevent misuse

Encapsulated classes are easier to debug

Encapsulation

Access Functions

Getter

A function that returns the value of a private variable

Setter

A function that changes the value of a private variable

```
class DateClass
{
public :
    void print()
    {
        cout << year << "/"
        << month << "/" << day;
    }
private :
    int year;
    int month;
    int day;
};

int main()
{
    DateClass today{2019,9,25};

    cout << "Today is " << today.month << today.day
        << today.year << endl;
    today.print();
    return 0;
}
```

```
classDemo.cpp: In function 'int main()':
classDemo.cpp:24:30: error: no matching function
for call to 'DateClass::DateClass(<brace-
enclosed initializer list>)'
    DateClass today{2019,9,23};
```

We got this error when the data members
were change to private.

If you can't directly
access a variable
(because it's private),
you shouldn't be able to
directly initialize it.

Constructors

A **constructor** is a special kind of class member function that is automatically called when an object of that class is instantiated.

Constructors are typically used to initialize member variables of the class to appropriate default or user-provided values, or to do any setup steps necessary for the class to be used.

Unlike normal member functions, constructors have specific rules for how they must be named:

Constructors must have the same name as the class (with the same capitalization)

Initializing Objects with Constructors

```
My bank account's name is  
Enter a new name for the bank account CSE 1325's Bank Account  
My bank account has been renamed CSE 1325's Bank Account
```

Our initial bank account name prints out as blank because, when our object `MyBankAccount` was created, `name` was initialized to the empty string.

What if you want the freshly created object to have a name already?

A class can define a **constructor** that specifies *custom initialization* for objects of that class.

Initializing Objects with Constructors

Constructor

special member function

must have the same name as the class

have no return type (not even void)

C++ requires a constructor call when *each* object is created

ideal point in program to initialize an object's data members

can have parameters

A constructor's *parameter list* specifies pieces of data required to initialize an object
usually public

CSE 1325

Week of 10/12/2020

Instructor : Donna French

Constructors

A **constructor** is a special kind of class member function that is automatically called when an object of that class is instantiated.

Constructors are typically used to initialize member variables of the class to appropriate default or user-provided values, or to do any setup steps necessary for the class to be used.

Unlike normal member functions, constructors have specific rules for how they must be named:

Constructors must have the same name as the class (with the same capitalization)

Initializing Objects with Constructors

```
My bank account's name is  
Enter a new name for the bank account CSE 1325's Bank Account  
My bank account has been renamed CSE 1325's Bank Account
```

Our initial bank account name prints out as blank because, when our object `MyBankAccount` was created, `name` was initialized to the empty string.

What if you want the freshly created object to have a name already?

A class can define a **constructor** that specifies *custom initialization* for objects of that class.

Initializing Objects with Constructors

Constructor

special member function

must have the same name as the class

have no return type (not even void)

C++ requires a constructor call when *each* object is created

ideal point in program to initialize an object's data members

can have parameters

A constructor's *parameter list* specifies pieces of data required to initialize an object
usually public

Default Constructors

A constructor that takes no parameters (or has parameters that all have default values) is called a **default constructor**.

The default constructor is called if no user-provided initialization values are provided.

Default Constructor

```
Account MyBankAccount;
```

No braces to the right of the object's name causes the default constructor to be called.

In any class that does *not* explicitly define a constructor, the compiler provides a default constructor with no parameters.

```
class Date
{
public :
    std::string getDateMMDDCCYY(void)
    {
        std::string s_month{std::to_string(month)};
        std::string s_day{std::to_string(day)};
        std::string s_year{std::to_string(month)};
        return (s_month.size() == 1 ? "0" : "") + s_month +
               (s_day.size() == 1 ? "0" : "") + s_day +
               s_year;
    }

private :
    int month;
    int day;
    int year;
};
```

```
int main(void)
{
    Date today; // Default constructor with no parameters created by compiler
    cout << today.getDateMMDDCCYY();
    return 0;
}
```

```
student@cse1325:/media/sf_VM$ ./constructorDemo.e
32764147339866832764student@cse1325:/media/sf_VM$
```

The default constructor does *not* initialize the class's fundamental-type data members, but *does* call the default constructor for each data member that's an object of another class.

```
5 class Account
6 {
7     public :
8         std::string getName() const
9     {
10        return name;
11    }
12
13 private :
14     std::string name;
15 }
16
17 using namespace std;
18
19 int main(void)
20 {
21     Account MyAccount;
22
23     cout << "My account's name is \" " << MyAccount.getName() << " \" " << endl;
24
25     return 0;
26 }
```

The default constructor does *not* initialize the class's fundamental-type data members, but *does* call the default constructor for each data member that's an object of another class.

```
student@cse1325:/media/sf_VM$ ./Account1Demo.e
My account's name is ""
```

```
class Account
{
public :
    std::string getName () const
    {
        return name;
    }
    int getBalance () const
    {
        return balance;
    }

private :
    std::string name;
    int balance;
};
```

New member function

New data member

```
int main(void)
{
    Account MyAccount;

    cout << "My account's name is \" " << MyAccount.getName() << "\"
        << " and the balance is " << MyAccount.getBalance() << endl;

    return 0;
}
```

```
student@cse1325:/media/sf_VM$ ./Account1Demo.e
My account's name is "" and the balance is 844865152
```

The default constructor does *not* initialize the class's fundamental-type data members, but *does* call the default constructor for each data member that's an object of another class.

```
class Account
{
public :
    Account()
    {
        balance = 0;
    }
    std::string getName() const
    {
        return name;
    }
    int getBalance() const
    {
        return balance;
    }

private :
    std::string name;
    int balance;
};
```

We provided a default constructor so the compiler does not create one for us.

student@cse1325:/media/sf_VM\$./Account1Demo.e
My account's name is "" and the balance is 0

```
class Account
{
    public :
        Account()
    {
        balance = -100;
        name = "MyPaycheck";
    }
    std::string getName() const
    {
        return name;
    }
    int getBalance() const
    {
        return balance;
    }

    private :
        std::string name;
        int balance;
};
```

student@cse1325:/media/sf_VM\$./Account1Demo.e

My account's name is "MyPaycheck" and the balance is -100

```
class Account
{
public :
    Account()
    {
        balance = -100;
        name = "MyPaycheck";
    }
    Account(int startingBalance, std::string newName="Bank Account")
    {
        balance = startingBalance;
        name = newName;
    }
}

Account MyAccount;
Account YourAccount{1000};
Account HisAccount{1,"His Account"};
```

```
class Account
{
public :
    Account()
    {
        balance = -100;
        name = "MyPaycheck";
    }
    Account(int startingBalance=0, std::string newName="Bank Account")
    {
        balance = startingBalance;
        name = newName;
    }
}
```

What happens if we try to give the balance parameter a default when the name already has a default?

```
student@cse1325:/media/sf_VM$ make
```

```
g++ -c -g -std=c++11 Account1Demo.cpp -o Account1Demo.o
```

```
Account1Demo.cpp: In function 'int main()':
```

```
Account1Demo.cpp:36:10: error: call of overloaded 'Account()' is ambiguous
```

```
    Account MyAccount;  
    ^~~~~~
```

```
Account1Demo.cpp:13:3: note: candidate: Account::Account(int, std::__cxx11::string)  
    Account(int startingBalance=0, std::string newName="Bank Account")  
    ^~~~~~
```

```
Account1Demo.cpp:8:3: note: candidate: Account::Account()  
    Account()  
    ^~~~~~
```

```
makefile:14: recipe for target 'Account1Demo.o' failed  
make: *** [Account1Demo.o] Error 1
```

```
class Account
{
public :
    Account (std::string accountName)
        : name{accountName}
    {
    }

    void setName(std::string accountName)
    {
        name = accountName;
    }

    std::string getName () const
    {
        return name;
    }

private :
    std::string name;
};
```

constructor has
one string
parameter
called accountName

Member initializers appear between a constructor's parameter list and the left brace that begins the constructor's body.

The member initializer list is separated from the parameter list with a colon (:).

Each member initializer consists of a data member's *variable name* followed by {} containing the member's *initial value*.

name is initialized with the parameter AccountName's value.

If a class contains more than one data member, each member initializer is separated from the next by a comma.

The member initializer list executes before the constructor's body executes.

Member Initializer List

Variables in the initializer list are not initialized in the order that they are specified in the initializer list.

They are initialized in the order in which they are declared in the class.

For best results, the following recommendations should be observed:

- 1) Don't initialize member variables in such a way that they are dependent upon other member variables being initialized first (in other words, ensure your member variables will properly initialize even if the initialization ordering is different).
- 2) Initialize variables in the initializer list in the same order in which they are declared in your class. This isn't strictly required so long as the prior recommendation has been followed, but your compiler may give you a warning if you don't do so and you have all warnings turned on.

```
class Account
{
public :
    Account(std::string accountName)
        : name{accountName}
    {
    }

void setName(std::string accountName)
{
    name = accountName;
}

std::string getName() const
{
    return name;
}

private :
    std::string name;
};
```

The constructor and member function `setName()` both have a parameter called `accountName`.

Both of them are local to their functions.

Their scope means that they are not visible to each other.

Default Constructor

In class Account, the class's default constructor calls class string's default constructor to initialize the data member name to the empty string. An uninitialized fundamental-type variable contains an undefined ("garbage") value.

If you define a custom constructor for a class, the compiler will *not* create a default constructor for that class.

Once we created the constructor, we can no longer use the default constructor.

```
g++ -c -g -std=c++11 TestBankAccount.cpp -o TestBankAccount.o
TestBankAccount.cpp: In function 'int main()':
TestBankAccount.cpp:14:10: error: no matching function for call to 'Account::Account()'
    Account MyBankAccount;
```

```
student@cse1325:/media/sf_VMs more makefile
#Donna French
#makefile for C++ program
SRC = TestBankAccount.cpp
OBJ = $(SRC:.cpp=.o)
EXE = $(SRC:.cpp=.e)

CFLAGS = -g -std=c++11

all : $(EXE)

$(EXE): $(OBJ)
        g++ $(CFLAGS) $(OBJ) -o $(EXE)

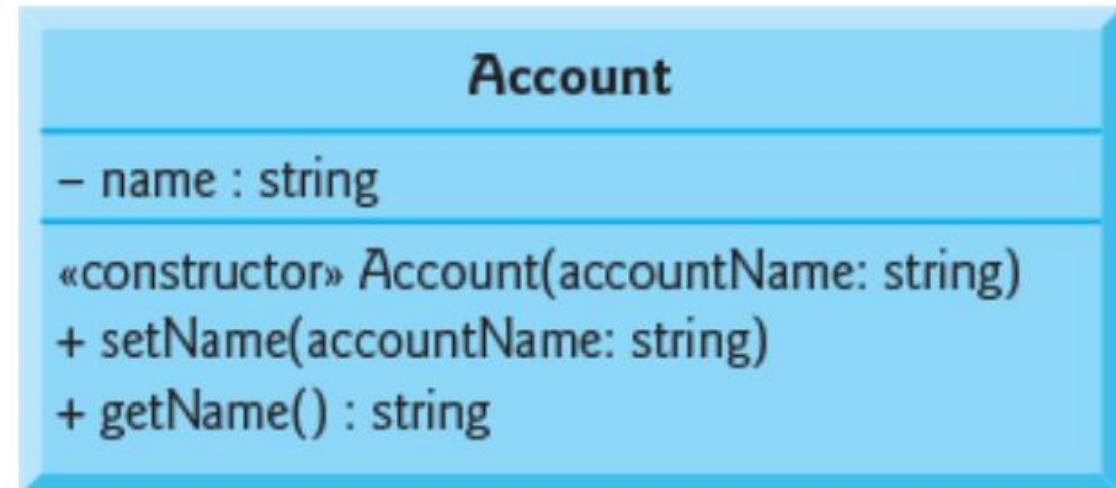
$(OBJ) : $(SRC)
        g++ -c $(CFLAGS) $(SRC) -o $(OBJ)
```

```
student@cse1325:/media/sf_VMs make
g++ -c -g -std=c++11 TestBankAccount.cpp -o TestBankAccount.o
g++ -g -std=c++11 TestBankAccount.o -o TestBankAccount.e
student@cse1325:/media/sf_VMs ./TestBankAccount.e
My bank account's name is Bank Account with Lots of Money
Enter a new name for the bank account My Bank Account
My bank account has been renamed My Bank Account
student@cse1325:/media/sf_VMs █
```

Class Diagram with a Constructor

Like operations, class diagrams show constructors in the *third* compartment of a class diagram.

To distinguish a constructor from the class's operations, the class diagram requires that the word “constructor” be enclosed in `<< and >>` and placed before the constructor’s name. It’s customary to list constructors *before* other operations in the third compartment.

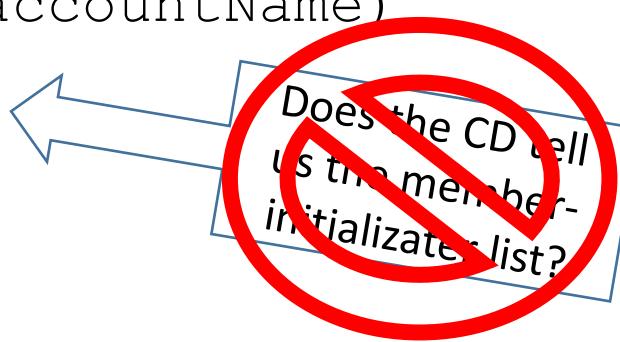


```
class Account
{
public :
    Account(std::string accountName)
        : name{accountName}
    {
    }

    void setName(std::string accountName)
    {
    }

    std::string getName()
    {
    }

private :
    std::string name;
};
```



Account

- name : string

«constructor» Account(accountName: string)

+ setName(accountName: string)

+ getName() : string

```
class Account
{
public :
    Account(std::string accountName)
        : name{accountName}
    {
    }

void setName(std::string accountName)
{
    name = accountName;
}

std::string getName() const
{
    return name;
}

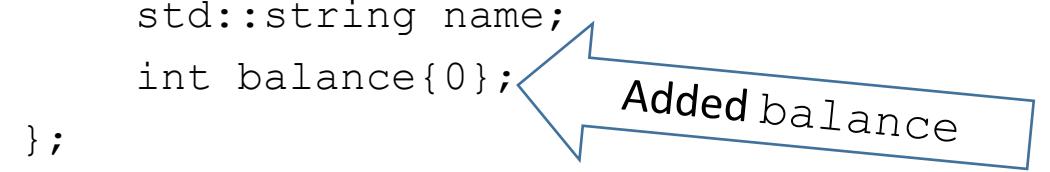
private :
    std::string name;
};
```

```
class Account
{
public :
    Account(std::string accountName)
        : name{accountName}
    {
    }

void setName(std::string accountName)
{
    name = accountName;
}

std::string getName() const
{
    return name;
}

private :
    std::string name;
    int balance{0};
};
```



Added balance

```

class Account
{
public :
    Account(std::string accountName)
        : name{accountName}
    {
    }

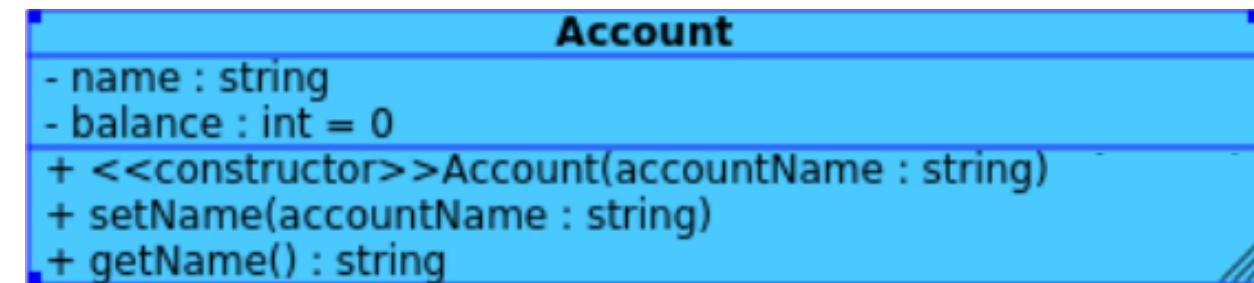
void setName(std::string accountName)
{
    name = accountName;
}

std::string getName() const
{
    return name;
}

private :
    std::string name;
    int balance{0};
};

```

balance is now a variable available to any member function of class Account



```
class Account
{
public :
    Account(std::string accountName, int InitialBalance)
        : name{AccountName}
    {
        if (InitialBalance > 0)
            balance = InitialBalance;
    }

    void setName(std::string accountName)
    {
        name = accountName;
    }

    std::string getName() const
    {
        return name;
    }

private :
    std::string name;
    int balance{0};      otherwise, it stays 0 – the default initial value set in the
                        class's definition.
};
```

Constructor validates that the passed in InitialBalance is greater than zero. If it is, the account's balance to the passed in balance;

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

#include "Account.h"
using namespace std;

int main()
{
    string NewAccountName;

    Account MyBankAccount{"Bank Account with Lots of Money", 1000000};

    cout << "My bank account's name is " << MyBankAccount.getName();

    cout << "\nEnter a new name for the bank account ";
    getline(cin, NewAccountName);
    MyBankAccount.setName(NewAccountName);

    cout << "My bank account has been renamed " << MyBankAccount.getName() << endl;

    return 0;
}
```

We started with the default constructor.

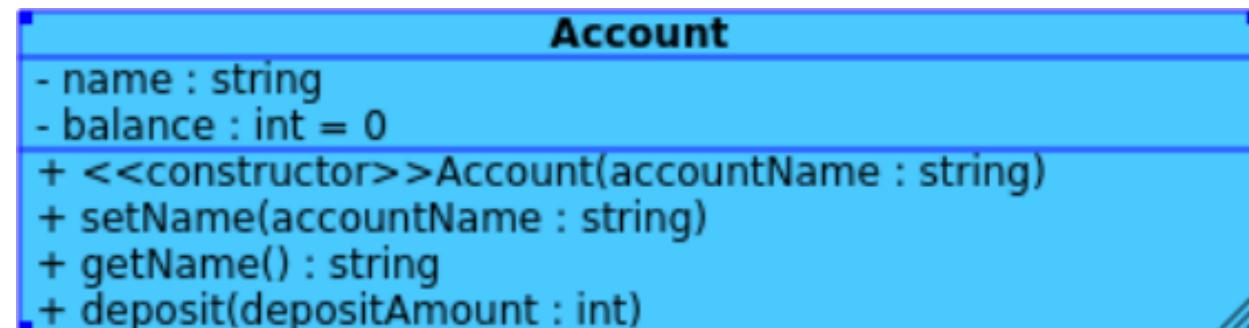
We then created our own constructor with one parameter to set name to the passed in value.

Then, we add a passed in balance to the constructor.

Adding a new member function- deposit

```
3 class Account
4 {
5     public :
6         Account(std::string AccountName, int InitialBalance)
7             : name{AccountName}
8         {
9             if (InitialBalance > 0)
10                 balance = InitialBalance;
11         }
12
13     void setName(std::string AccountName)
14     {
15         name = AccountName;
16     }
17
18     std::string getName() const
19     {
20         return name;
21     }
22
23     void deposit(int depositAmount)
24     {
25         if (depositAmount > 0)
26         {
27             balance += depositAmount;
28         }
29     }
30
31     private :
32         std::string name;
33         int balance{0};
34 }
```

```
void deposit(int depositAmount)
{
    if (depositAmount > 0)
    {
        balance += depositAmount;
    }
}
```



```
3 #include <iostream>
4 #include <string>
5
6 #include "Account.h"
7
8 using namespace std;
9
10 int main()
11 {
12     string NewAccountName;
13     int deposit;
14
15     Account MyBankAccount{"Bank Account with Lots of Money", 1000000};
16
17     cout << "My bank account's name is " << MyBankAccount.getName();
18
19     cout << "\nEnter a new name for the bank account ";
20     getline(cin, NewAccountName);
21     MyBankAccount.setName(NewAccountName);
22
23     cout << "My bank account has been renamed " << MyBankAccount.getName() << endl;
24
25     cout << "How many dollars would you like to deposit? ";
26     cin >> deposit;
27     MyBankAccount.deposit(deposit);
28
29     return 0;
30 }
```

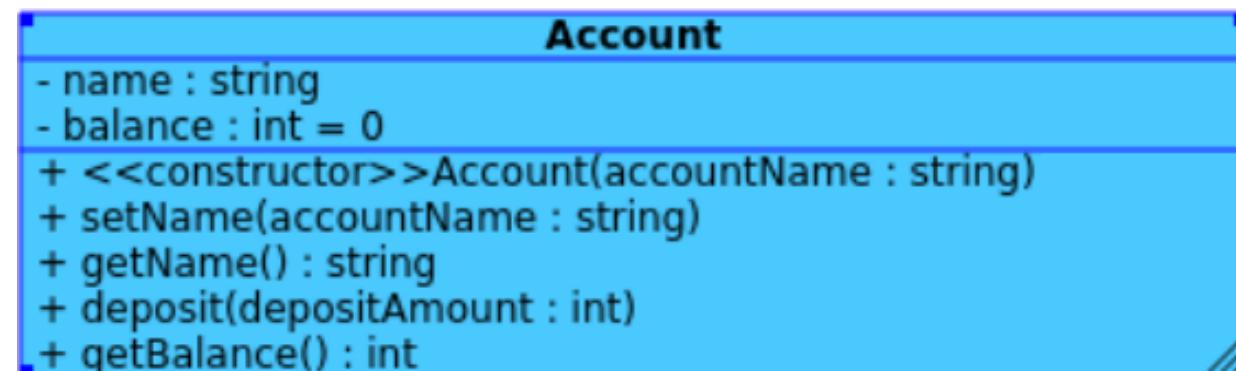
Added member function deposit()

Adding a new const member function - getBalance

```
3 class Account
4 {
5     public :
6         Account(std::string AccountName, int InitialBalance)
7             : name{AccountName}
8         {
9             if (InitialBalance > 0)
10                 balance = InitialBalance;
11         }
12
13     void setName(std::string AccountName)
14     {
15         name = AccountName;
16     }
17
18     std::string getName() const
19     {
20         return name;
21     }
22
23     void deposit(int depositAmount)
24     {
25         if (depositAmount > 0)
26         {
27             balance += depositAmount;
28         }
29     }
30
31     int getBalance const(void)
32     {
33         return balance;
34     }
35
36     private :
37         std::string name;
38         int balance{0};
39 }
```

```
        int getBalance const(void)
{
    return balance;
}
```

getBalance is declared const, because in the process of returning the balance, the function does not, and should not, modify anything in the Account object.



Account.h

```
31
32     {
33         return balance
34     }
```

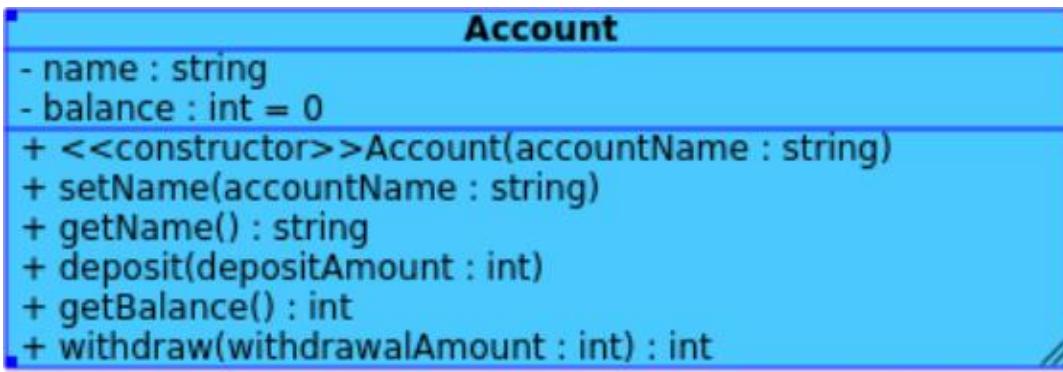
```
student@cse1325:/media/sf_VM$ make
g++ -c -g -std=c++11 TestBankAccount.cpp -o TestBankAccount.o
In file included from TestBankAccount.cpp:6:0:
Account.h: In member function 'int Account::getBalance() const':
Account.h:34:3: error: expected ';' before '}' token
}
^
makefile:15: recipe for target 'TestBankAccount.o' failed
make: *** [TestBankAccount.o] Error 1
student@cse1325:/media/sf_VM$ █
```

```
3 #include <iostream>
4 #include <string>
5
6 #include "Account.h"
7
8 using namespace std;
9
10 int main()
11 {
12     string NewAccountName;
13     int deposit;
14
15     Account MyBankAccount{"Bank Account with Lots of Money", 1000000};
16
17     cout << "My bank account's name is " << MyBankAccount.getName();
18
19     cout << "\nEnter a new name for the bank account ";
20     getline(cin, NewAccountName);
21     MyBankAccount.setName(NewAccountName);
22
23     cout << "My bank account has been renamed " << MyBankAccount.getName() << endl;
24
25     cout << "How many dollars would you like to deposit? ";
26     cin >> deposit;
27     MyBankAccount.deposit(deposit);
28
29     cout << "\n\nYour balance is " << MyBankAccount.getBalance() << endl;
30
31     return 0;
32 }
```

Adding a new member function - withdraw

```
3 class Account
4 {
5     public :
6         Account(std::string AccountName, int InitialBalance)
7             : name{AccountName}
8         {
9             if (InitialBalance > 0)
10                 balance = InitialBalance;
11         }
12
13     void setName(std::string AccountName)
14     {
15         name = AccountName;
16     }
17
18     std::string getName() const
19     {
20         return name;
21     }
22
23     void deposit(int depositAmount)
24     {
25         if (depositAmount > 0)
26         {
27             balance += depositAmount;
28         }
29     }
30
31     int withdraw(int withdrawalAmount)
32     {
33         if (withdrawalAmount <= balance)
34         {
35             balance -= withdrawalAmount;
36             return 1;
37         }
38         else
39         {
40             return 0;
41         }
42     }
43
44     int getBalance (void) const
45     {
46         return balance;
47     }
48
49
50     private :
51         std::string name;
52         int balance{0};
53 }
```

```
int withdraw(int withdrawalAmount)
{
    if (withdrawalAmount <= balance)
    {
        balance -= withdrawalAmount;
        return 1;
    }
    else
    {
        return 0;
    }
}
```



```
cout << "\n\nHow many dollars will be withdrawn? ";
cin >> withdrawal;
if (MyBankAccount.withdraw(withdrawal))
{
    cout << "The new balance is " << MyBankAccount.getBalance() << endl;
}
else
{
    cout << "Insufficient funds" << endl;
}
```

```
int withdraw(int withdrawalAmount)
{
    if (withdrawalAmount <= balance)
    {
        balance -= withdrawalAmount;
        return 1;
    }
    else
    {
        return 0;
    }
}
```

```
int PrintMenu(void)
{
    int Choice = 0;

    cout << "0. Exit menu" << endl;
    cout << "1. Get Account Name" << endl;
    cout << "2. Change Account Name" << endl;
    cout << "3. Check Current Balance" << endl;
    cout << "4. Deposit Funds" << endl;
    cout << "5. Withdraw Funds" << endl;
    cout << "\nEnter choice " << endl;
    cin >> Choice;
    getchar();

    return Choice;
}
```

0. Exit menu
1. Get Account Name
2. Change Account Name
3. Check Current Balance
4. Deposit Funds
5. Withdraw Funds

Enter choice

```
do
{
    Choice = PrintMenu();

    switch (Choice)
    {
        case 1 :
            cout << "\n\nBank account's name is " << MyBankAccount.getName() << endl;
            break;
        case 2 :
            cout << "\n\nEnter a new name for the bank account ";
            getline(cin, NewAccountName);
            MyBankAccount.setName(NewAccountName);
            cout << "\n\nBank account has been renamed " << MyBankAccount.getName() << endl;
            break;
        case 3 :
            cout << "\n\nThe current balance is " << MyBankAccount.getBalance() << endl;
            break;
        case 4 :
            cout << "\n\nHow many dollars will be deposited? ";
            cin >> deposit;
            MyBankAccount.deposit(deposit);
            cout << "The new balance is " << MyBankAccount.getBalance() << endl;
            break;
        case 5 :
            cout << "\n\nHow many dollars will be withdrawn? ";
            cin >> withdrawal;
            if (MyBankAccount.withdraw(withdrawal))
            {
                cout << "The new balance is " << MyBankAccount.getBalance() << endl;
            }
            else
            {
                cout << "Insufficient funds" << endl;
            }
            break;
        default :
            cout << "Goodbye!" << endl;
    }

    cout << "\n\n";
}
while (Choice);
```

to_string()

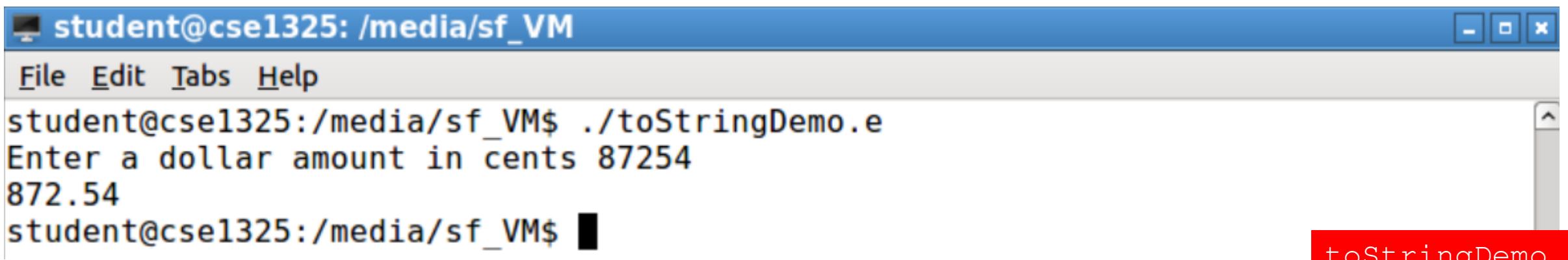
```
long amount;

cout << "Enter a dollar amount in cents ";
cin >> amount;

std::string dollars{std::to_string(amount / 100)};

std::string cents{std::to_string(std::abs(amount % 100))};

cout << dollars + "." + (cents.size() == 1 ? "0" : "") + cents << endl;
```



The screenshot shows a terminal window with a blue header bar. The header bar contains the text "student@cse1325: /media/sf_VM" on the left and standard window control buttons (-, X, and a maximize/minimize button) on the right. Below the header is a menu bar with "File", "Edit", "Tabs", and "Help". The main body of the terminal shows the command "student@cse1325:/media/sf_VM\$./toStringDemo.e" followed by the user's input "Enter a dollar amount in cents 87254" and the program's output "872.54".

```
student@cse1325: /media/sf_VM
File Edit Tabs Help
student@cse1325:/media/sf_VM$ ./toStringDemo.e
Enter a dollar amount in cents 87254
872.54
student@cse1325:/media/sf_VM$
```

```
#include <string>
#include <cmath>

class DollarAmount
{
public :
    DollarAmount(long value) : amount{value}
    {
    }

    long getAmount(void)
    {
        return amount;
    }

    void add(DollarAmount ObjectToAdd)
    {
        amount += ObjectToAdd.amount;
    }

    void subtract(DollarAmount ObjectToSubtract)
    {
        amount -= ObjectToSubtract.amount;
    }

    std::string displayMoney() const
    {
        std::string dollars{std::to_string(amount / 100)};
        std::string cents{std::to_string(std::abs(amount % 100))};
        return dollars + "." + (cents.size() == 1 ? "0" : "") + cents;
    }
private :
    long amount{0};
};
```

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

#include "DollarAmount.h"

using namespace std;

void PrintLine(DollarAmount DAOBJECT1, DollarAmount DAOBJECT2)
{
    cout << "DAOBJECT1 amount in pennies " << DAOBJECT1.getAmount() << endl;
    cout << "DAOBJECT1 amount in dollars and cents " << DAOBJECT1.displayMoney() << "\n\n";

    cout << "DAOBJECT2 amount in pennies " << DAOBJECT2.getAmount() << endl;
    cout << "DAOBJECT2 amount in dollars and cents " << DAOBJECT2.displayMoney() << endl;
}

int main()
{
    DollarAmount DAOBJECT1{12345}; // $123.45 (represents dollar amounts in whole number of pennies
    DollarAmount DAOBJECT2{655}; // $6.55

    PrintLine(DAOBJECT1, DAOBJECT2);

    cout << "\n\nAdding DAOBJECT2 to DAOBJECT1...\n\n" << endl;
    DAOBJECT1.add(DAOBJECT2);

    PrintLine(DAOBJECT1, DAOBJECT2);

    cout << "\n\nSubtracting DAOBJECT1 from DAOBJECT1...\n\n" << endl;
    DAOBJECT1.subtract(DAOBJECT1);

    PrintLine(DAOBJECT1, DAOBJECT2);

    return 0;
}
```

Constructor

```
DollarAmount DAOBJECT1{12345}; // 123.45
```

```
DollarAmount DAOBJECT2{655}; // 6.55
```

```
class DollarAmount
{
    public :
        DollarAmount(long value) : amount{value}
    {
    }

    private :
        long amount{0};
```

getAmount () Member Function

```
cout << "DAOobject1 amount in pennies " << DAOobject1.getAmount() << endl;
```

```
class DollarAmount
```

```
{  
public :  
    long getAmount(void)  
    {  
        return amount;  
    }
```

DA0bject1 amount in pennies 12345

displayMoney() Member Function

```
cout << "DAOobject1 amount in dollars and cents "
    << DAOobject1.displayMoney() << "\n\n";

class DollarAmount
{
public :
    std::string displayMoney() const
    {
        std::string dollars{std::to_string(amount / 100)};
        std::string cents{std::to_string(std::abs(amount % 100))};
        return dollars + "." + (cents.size() == 1 ? "0" : "") + cents;
    }
}
```

DAObject1 amount in dollars and cents 123.45

Member functions add () and subtract ()

```
DAOBJECT1.add(DAOBJECT2);  
DAOBJECT1.subtract(DAOBJECT1);  
  
class DollarAmount  
{  
    public :  
        void add(DollarAmount ObjectToAdd)  
        {  
            amount += ObjectToAdd.amount;  
        }  
  
        void subtract(DollarAmount ObjectToSubtract)  
        {  
            amount -= ObjectToSubtract.amount;  
        }  
}
```

student@student@cse1325: /media/sf_VM

File Edit Tabs Help

DAObject1 amount in pennies 12345
DAObject1 amount in dollars and cents 123.45

DAObject2 amount in pennies 655
DAObject2 amount in dollars and cents 6.55

Adding DAObject2 to DAObject1...

```
DAObject1{12345}; // 123.45  
DAObject2{655}; // 6.55
```

DAObject1 amount in pennies 13000
DAObject1 amount in dollars and cents 130.00

DAObject2 amount in pennies 655
DAObject2 amount in dollars and cents 6.55

Subtracting DAObject1 from DAObject1...

DAObject1 amount in pennies 0
DAObject1 amount in dollars and cents 0.00

DAObject2 amount in pennies 655
DAObject2 amount in dollars and cents 6.55

friend Function and friend Classes

A **friend function** of a class is a non-member function that has the right to access the public *and* non-public class members.

A **friend function** is a function that can access the private members of a class as though it were a member of that class.

Standalone functions, entire classes or member functions of other classes may be declared to be *friends* of another class.

Declaring a friend

To declare a non-member function as a friend of a class, place the function prototype in the class definition and precede it with the keyword `friend`.

The `friend` declaration(s) can appear *anywhere* in a class and are not affected by access specifiers public or private (or protected, which we discuss later).

```
#include <iostream>

using namespace std;

class ClassABC
{
public :
    int getZ() const
    {
        return Z;
    }
private :
    int Z{0};
};

int main()
{
    ClassABC def;

    cout << "def.Z after instantiation: " << def.getZ() << endl;
}
```

Let's add a friend function called setZ that can update private data member Z

```
class ClassABC
{
    friend void setZ(ClassABC&, int);
public :
    int getZ() const
    {
        return Z;
    }
private :
    int Z{0};
};

void setZ(ClassABC& ghi, int newvalue)
{
    ghi.Z = newvalue;
}

int main()
{
    ClassABC def;
    cout << "def.Z after instantiation: "
          << def.getZ() << endl;
    setZ(def, 8);
    cout << "def.Z after call to setZ
friend function: "
          << def.getZ() << endl;
}
```

Properties of friend

Friendship is *granted, not taken*—for class B to be a friend of class A, class A must *explicitly* declare that class B is its friend.

Friendship is not *symmetric*—if class A is a friend of class B, you cannot infer that class B is a friend of class A.

Friendship is not *transitive*—if class A is a friend of class B and class B is a friend of class C, you cannot infer that class A is a friend of class C.

Friendship is not inherited.

Rules of friendship

Even though the prototypes for `friend` functions appear in the class definition, `friends` are not member functions.

Member access notions of private, protected and public are not relevant to `friend` declarations, so `friend` declarations can be placed anywhere in a class definition.

Place all friendship declarations first inside the class definition's body and do not precede them with any access specifier.

Why make friends?

Because everyone should have a friend...



Why does C++ have friend functions?

Allows functionality to be extracted from a class and kept in a non member function for use by multiple classes.

When a database changes, the indexes must be updated. This process can be kept in a friend function rather than a class member function. This type of generic function can then be reused across different database tables.

Functions that are used solely for testing a class can be made friends in the class being tested. This allows for the test code to change without changing the class itself and gives the test function access to the class's private data.

One way to keep classes from inheriting functionality.

Pointers to Structures

In C, it is possible to declare a pointer to any type

This includes pointers to structures.

```
struct tshirt
{
    char size[5];
    char color[10];
    char design[100];
    char fittype;
    float price;
    int inventory;
};
```

```
struct tshirt MyTShirts = {"M", "BLUE", "DISNEY", 'W', 29.99, 1};
struct tshirt *tshirtptr;
tshirtptr = &MyTShirts;

printf("MyTShirts.design\t%s\n", MyTShirts.design);
printf("(*tshirtptr).design\t%s\n\n", (*tshirtptr).design);
```

MyTShirts.design	DISNEY
(*tshirtptr).design	DISNEY

Pointers to Structures

In C, it is possible to declare a pointer to any type

This includes pointers to structures in arrays.

```
struct tshirt DCComicsTShirts[5] = { {"XS", "BLACK", "BATMAN", 'Y', 12.99, 198},  
    { "S", "BLUE", "SUPERMAN", 'M', 24.99, 34},  
    { "M", "RED", "WONDER WOMAN", 'W', 27.99, 87},  
    { "L", "YELLOW", "AQUAMAN", 'M', 26.99, 65},  
    { "XL", "GREEN", "GREEN LANTERN", 'Y', 15.99, 81} };
```

```
struct tshirt *tshirtarrayptr;  
tshirtarrayptr = &DCComicsTShirts[3];
```

```
printf("DCComicsTShirts[3].design\t%s\n", DCComicsTShirts[3].design);  
printf("(*tshirtarrayptr).design\t%s\n", (*tshirtarrayptr).design);
```

DCComicsTShirts[3].design	AQUAMAN
(*tshirtarrayptr).design	AQUAMAN

Pointers to Structures

The () are necessary because the dot selector has precedence over the dereferencing operator *

```
printf("tshirtptr design\t%s\n\n", (*tshirtptr).design);
printf("tshirtarrayptr design\t%s\n", (*tshirtarrayptr).design);
```

Without the (), the compiler complains

```
printf("tshirtptr design\t%s\n\n", *tshirtptr.design);
error: request for member 'design' in something not a structure or
union
```

Pointers to Structures

The concept of a pointer to structure is used so often in C that a special syntax was developed to reference the members of the target structure.

(*struct_pointer) .member can be written as struct_pointer->member

```
printf("tshirtptr design\t%s\n\n", (*tshirtptr).design);
printf("tshirtptr design\t\t%s\n", tshirtptr->design);
```

```
printf("tshirtarrayptr design\t%s\n", (*tshirtarrayptr).design);
printf("tshirtarrayptr design\t\t%s\n", tshirtarrayptr->design);
```

this

One of the questions about classes often asked is,

“When a member function is called, how does C++ keep track of which object it was called on?”.

The answer is that C++ utilizes a hidden pointer named “this”!

this

There's only one copy of each class's functionality, but there can be many objects of a class, so how do member functions know which object's data members to manipulate?

Every object has access to its own address through a pointer called `this` (a C++ keyword).

The `this` pointer is not part of the object itself. The memory occupied by the `this` pointer is not reflected in the result of a `sizeof` operation on the object. Rather, the `this` pointer is passed (by the compiler) as an implicit argument to each of the object's non-static member functions.

```
class Simple
{
public:
    Simple(int id)
    {
        setID(id);
    }
    void setID(int id)
    {
        m_id = id;
    }
    int getID()
    {
        return m_id;
    }
private:
    int m_id;
};
```

this

```
int main()
{
    Simple simple(1);
    simple.setID(2);
    std::cout << simple.getID();

    return 0;
}
```

What would this print?

this

When we call

```
simple.setID(2);
```

C++ knows that function `setID()` should operate on object `simple` and that `m_id` actually refers to `simple.m_id`.

How?

this

Let's look at this line of code

```
simple.setID(2);
```

Although the call to function `setID()` looks like it only has one argument, it actually has two!

When compiled, the compiler converts `simple.setID(2);` into the following

```
setID(&simple, 2);
```

Note that `simple` has been changed from an object prefix to a function argument!

this

```
setID(&simple, 2);
```

setID() is now just a standard function call, and the object simple (which was formerly an object prefix) is now passed by address as an argument to the function.

Since the function call now has an added argument, the member function definition needs to be modified to accept (and use) this argument as a parameter.

this

Since the function call now has an added argument, the member function definition needs to be modified to accept (and use) this argument as a parameter.

```
void setID(int id)
{
    m_id = id;
}
```

is converted by the compiler into

```
void setID(Simple* const this, int id)
{
    this->m_id = id;
}
```

this

```
void setID(Simple* const this, int id)
{
    this->m_id = id;
}
```

When the compiler compiles a normal member function, it **implicitly** adds a new parameter to the function named `this`.

The `this` pointer is a hidden `const` pointer that holds the address of the object the member function was called on.

this

```
void setID(Simple* const this, int id)
{
    this->m_id = id;
}
```

Inside the member function, any class members (functions and variables) also need to be updated so they refer to the object the member function was called on.

This is done by adding a `this->` prefix to each of them.

In the body of function `setID()`, `m_id` (which is a class member variable) has been converted to `this->m_id`.

When `this` points to the address of `simple`, `this->m_id` will resolve to `simple.m_id`.

this

When we call

```
simple.setID(2);
```

the compiler actually calls

```
setID(&simple, 2);
```

Inside setID(), the this pointer holds the address of object simple.

Any member variables inside setID() are prefixed with this->.

So when we say m_id = id, the compiler is actually executing

```
this->m_id = id
```

which in this case updates simple.m_id to id.

Using the `this` Pointer to Avoid Naming Collisions

Member functions use the `this` pointer

implicitly (as we've done so far)

or

explicitly

to reference an object's data members and other member functions. A common explicit use of the `this` pointer is to avoid naming conflicts between a class's data members and member-function parameters (or other local variables)

this

All of this happens automatically.

Just remember is that all normal member functions have a `this` pointer that refers to the object the function was called on

`this` always points to the object being operated on.

So how many “this” pointers exist?

Each member function has a `this` pointer parameter that is set to the address of the object being operated on.

this

```
int main()
{
    Simple A(1); // this = &A inside the Simple constructor
    Simple B(2); // this = &B inside the Simple constructor
    A.setID(3); // this = &A inside member function setID
    B.setID(4); // this = &B inside member function setID

    return 0;
}
```

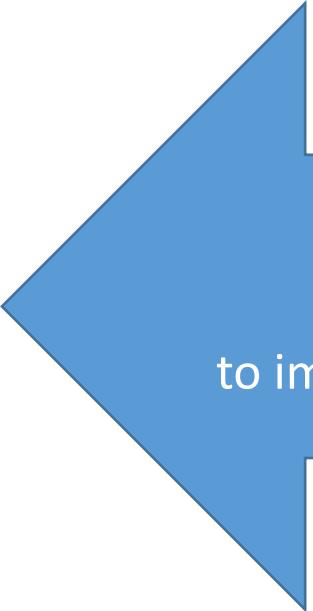
The `this` pointer alternately holds the address of object A or B depending on whether we've called a member function on object A or B.

`this` is just a function parameter - it doesn't add any memory usage to your class

Using the this Pointer to Avoid Naming Collisions

We are familiar with the implicit call

```
class Test
{
public :
    void print(void) const
    {
        cout << "x = " << x;
    }
private :
    int x{ 0 };
};
```



Compiler is translating print() to
void print(&x) const
to implicitly pass the pointer to the object

Using the `this` Pointer to Avoid Naming Collisions

```
void setHour(int hour)
{
    if (hour >= 0 && hour < 24)
    {
        hour = hour;
    }
}
```

hour is passed into the function setHour and then validation is performed on it.

If it passes validation, then we want to update the object's data member hour.

We do this by using this.

Using the this Pointer to Avoid Naming Collisions

So why not just use different names?

```
void setHour(int hourA)
{
    if (hourA >= 0 && hourA < 24)
    {
        this->hour = hourA;
    }
}
```

A widely accepted practice to minimize the proliferation of identifier names is to use the same name for a set function's parameter and the data member it sets, and to reference the data member in the set function's body via this->.

Using the `this` Pointer to Avoid Naming Collisions

We are familiar with the implicit usage so what does the explicit usage look like?

```
class Test
{
public :
    void print(void) const
    {
        cout << "x = " << (*this).x;
        cout << "x = " << this->x;
    }
private :
    int x{0};
};
```

this

Recommendation

Do not add `this->` to all uses of your class members.

Only do so when you have a specific reason to.

We will see more examples of when using `this` is necessary.

Type of this pointer

The type of the `this` pointer depends on the type of the object and whether the member function in which this is used is declared `const`

In a non-`const` member function of class `Employee`, the `this` pointer has the type
`Employee* const`
a constant pointer to a nonconstant `Employee`.

In a `const` member function, this has the type
`const Employee* const`
a constant pointer to a constant `Employee`.

this is a `const` pointer -- you can change the value of the underlying object it points to, but you can not make it point to something else.

CSE 1325

Week of 10/19/2020

Instructor : Donna French

this

One of the questions about classes often asked is,

“When a member function is called, how does C++ keep track of which object it was called on?”.

The answer is that C++ utilizes a hidden pointer named “this”!

this

There's only one copy of each class's functionality, but there can be many objects of a class, so how do member functions know which object's data members to manipulate?

Every object has access to its own address through a pointer called `this` (a C++ keyword).

The `this` pointer is not part of the object itself. The memory occupied by the `this` pointer is not reflected in the result of a `sizeof` operation on the object. Rather, the `this` pointer is passed (by the compiler) as an implicit argument to each of the object's non-static member functions.

```
class Simple
{
public:
    Simple(int id)
    {
        setID(id);
    }
    void setID(int id)
    {
        m_id = id;
    }
    int getID()
    {
        return m_id;
    }
private:
    int m_id;
};
```

this

```
int main()
{
    Simple simple(1);
    simple.setID(2);
    std::cout << simple.getID();

    return 0;
}
```

What would this print?

this

When we call

```
simple.setID(2);
```

C++ knows that function `setID()` should operate on object `simple` and that `m_id` actually refers to `simple.m_id`.

How?

this

Let's look at this line of code

```
simple.setID(2);
```

Although the call to function `setID()` looks like it only has one argument, it actually has two!

When compiled, the compiler converts `simple.setID(2);` into the following

```
setID(&simple, 2);
```

Note that `simple` has been changed from an object prefix to a function argument!

this

```
setID(&simple, 2);
```

setID() is now just a standard function call, and the object simple (which was formerly an object prefix) is now passed by address as an argument to the function.

Since the function call now has an added argument, the member function definition needs to be modified to accept (and use) this argument as a parameter.

this

Since the function call now has an added argument, the member function definition needs to be modified to accept (and use) this argument as a parameter.

```
void setID(int id)
{
    m_id = id;
}
```

is converted by the compiler into

```
void setID(Simple* const this, int id)
{
    this->m_id = id;
}
```

this

```
void setID(Simple* const this, int id)
{
    this->m_id = id;
}
```

When the compiler compiles a normal member function, it **implicitly** adds a new **parameter** to the function named this.

The this pointer is a hidden const pointer that holds the address of the object the member function was called on.

this

```
void setID(Simple* const this, int id)
{
    this->m_id = id;
}
```

Inside the member function, any class members (functions and variables) also need to be updated so they refer to the object the member function was called on.

This is done by adding a `this->` prefix to each of them.

In the body of function `setID()`, `m_id` (which is a class member variable) has been converted to `this->m_id`.

When `this` points to the address of `simple`, `this->m_id` will resolve to `simple.m_id`.

this

When we call

```
simple.setID(2);
```

the compiler actually calls

```
setID(&simple, 2);
```

Inside setID(), the this pointer holds the address of object simple.

Any member variables inside setID() are prefixed with this->.

So when we say m_id = id, the compiler is actually executing

```
this->m_id = id
```

which in this case updates simple.m_id to id.

Using the `this` Pointer to Avoid Naming Collisions

Member functions use the `this` pointer

implicitly (as we've done so far)

or

explicitly

to reference an object's data members and other member functions. A common explicit use of the `this` pointer is to avoid naming conflicts between a class's data members and member-function parameters (or other local variables)

this

All of this happens automatically.

Just remember is that all normal member functions have a `this` pointer that refers to the object the function was called on

`this` always points to the object being operated on.

So how many “this” pointers exist?

Each member function has a `this` pointer parameter that is set to the address of the object being operated on.

this

```
int main()
{
    Simple A(1); // this = &A inside the Simple constructor
    Simple B(2); // this = &B inside the Simple constructor
    A.setID(3); // this = &A inside member function setID
    B.setID(4); // this = &B inside member function setID

    return 0;
}
```

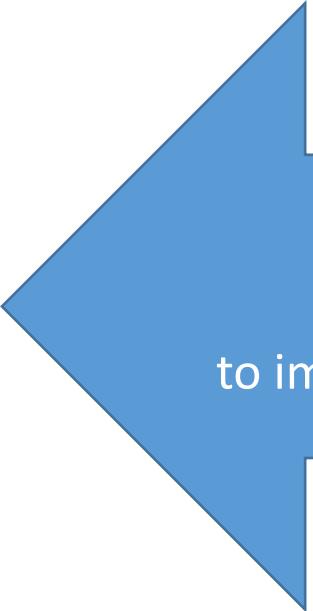
The `this` pointer alternately holds the address of object A or B depending on whether we've called a member function on object A or B.

`this` is just a function parameter - it doesn't add any memory usage to your class

Using the this Pointer to Avoid Naming Collisions

We are familiar with the implicit call

```
class Test
{
public :
    void print(void) const
    {
        cout << "x = " << x;
    }
private :
    int x{ 0 };
};
```



Compiler is translating print() to
void print(&x) const
to implicitly pass the pointer to the object

Using the `this` Pointer to Avoid Naming Collisions

```
void setHour(int hour)
{
    if (hour >= 0 && hour < 24)
    {
        hour = hour;
    }
}
```

hour is passed into the function setHour and then validation is performed on it.

If it passes validation, then we want to update the object's data member hour.

We do this by using this.

Using the this Pointer to Avoid Naming Collisions

So why not just use different names?

```
void setHour(int hourA)
{
    if (hourA >= 0 && hourA < 24)
    {
        this->hour = hourA;
    }
}
```

A widely accepted practice to minimize the proliferation of identifier names is to use the same name for a set function's parameter and the data member it sets, and to reference the data member in the set function's body via this->.

Using the `this` Pointer to Avoid Naming Collisions

We are familiar with the implicit usage so what does the explicit usage look like?

```
class Test
{
public :
    void print(void) const
    {
        cout << "x = " << (*this).x;
        cout << "x = " << this->x;
    }
private :
    int x{0};
};
```

this

Recommendation

Do not add `this->` to all uses of your class members.

Only do so when you have a specific reason to.

We will see more examples of when using `this` is necessary.

Type of this pointer

The type of the `this` pointer depends on the type of the object and whether the member function in which this is used is declared `const`

In a non-`const` member function of class `Employee`, the `this` pointer has the type
`Employee* const`
a constant pointer to a nonconstant `Employee`.

In a `const` member function, this has the type
`const Employee* const`
a constant pointer to a constant `Employee`.

this is a `const` pointer -- you can change the value of the underlying object it points to, but you can not make it point to something else.

Operator Overloading

>>

stream extraction operator
bitwise right shift operator

<<

stream insertion operator
bitwise left shift operator

These are familiar operators that are overloaded.

Operator Overloading

+ and -

Each of these performs differently depending on their context

- integer addition

- floating point arithmetic

- pointer arithmetic

These are familiar operators that are overloaded meaning that the compiler generates the appropriate code based on the types of the operands.

Operator Overloading

The C++ Standard Library's class `string` has lots of overloaded operators.

```
string s1{"happy"};
string s2{"birthday"};
string s3;

cout << "s1\t" << s1 << "\ns2\t" << s2 << "\ns3" << s3 << endl;
```

```
s1    happy
s2    birthday
s3
```

```
string s1{"happy"};
string s2{"birthday"};
```

Operator Overloading

```
cout << "\ns2 == s1\t" << (s2 == s1)
<< "\ns2 != s1\t" << (s2 != s1)
<< "\ns2 > s1\t" << (s2 > s1)
<< "\ns2 < s1\t" << (s2 < s1)
<< endl;
```

s2 == s1	false
s2 != s1	true
s2 > s1	false
s2 < s1	true

```
string s1{"happy"};
string s2{"birthday"};
string s3;
```

Operator Overloading

```
cout << "\n\ns3 = " << s3 << endl;
s3 = s1;
cout << "\ns3 = " << s3 << endl;
```

```
s3[0] = 'H';
s2[0] = 'B';
s3 += s2;
cout << "\n\n" << s3 << endl;
```

```
s3 =
s3 = happy
```

HappyBirthday

stringoverloadDemo.cpp

Operator Overloading

The operators that have been overloaded for strings provide a concise notation for manipulating those string objects.

We can overload operators with our own user-defined types as well.

C++ does not allow new operators to be created but it does allow most existing operators to be overloaded so that when they are used with objects, they have special meanings.

Operator Overloading

Most of C++'s operators can be overloaded.

There are a few exceptions

- .
- . *
- (pointer to member)
- ::
- ? :

Operator Overloading Rules and Restrictions

- An operator's precedence cannot be changed by overloading
 - () can be used to force the order of evaluation of overloaded operators
- An operator's associativity cannot be changed by overloading
 - if an operator normally associates from left to right then so will it when overloaded
- An operator's "ary"ness cannot be changed
 - overloaded unary operators remain unary
 - overloaded binary operators remain binary
 - ternary operator ?: cannot be overloaded (C++'s only ternary operator)
- Only existing operators can be overloaded
 - you cannot create new operators
- You cannot overload operators to change how the operator works on fundamental types
 - cannot make the + operator do subtraction
- Related operators, like + and +=, must be overloaded separately
- When overloading (), [], -> or any assignment operators, the operator overload function must be declared as a class member. All other overloaded operators can be member functions

Operator Overloading

In C++, operators are implemented as functions.

By using function overloading on the operator functions, you can define your own versions of the operators that work with different data types (including classes that you've written).

Using function overloading to overload operators is called operator overloading.

Operator Overloading

```
int x = 2;  
int y = 3;  
std::cout << x + y << '\n';
```

Think of + as a function that adds two values and returns the result.

operator+(x, y)

and cout prints the return value

Operator Overloading

```
Mystring string1 = "Hello, ";
Mystring string2 = "World!";
std::cout << string1 + string2 << '\n';
```

Does this concatenate `string1` and `string2` to make

Hello, World!

No, because `string1` and `string2` are of objects instantiated from class `Mystring`.

Why? Because `+` is not defined for class `Mystring`.

Operator Overloading

When evaluating an expression containing an operator, the compiler uses the following rules

- If *all* of the operands are fundamental data types, the compiler will call a built-in routine if one exists. If one does not exist, the compiler will produce a compiler error.
- If *any* of the operands are user data types, the compiler looks to see whether the type has a matching overloaded operator function that it can call. If it can't find one, it will try to convert one or more of the user-defined type operands into fundamental data types so it can use a matching built-in operator. If that fails, then it will produce a compile error.

Overloading Binary Operator

A binary operator can be overloaded as

- a member function with one parameter
- a non-member function with two parameters
 - one parameter must be either a class object or a reference to a class object

Non-member functions are often declared as a friend of a class to access private data

Binary Overload Operators as Member Functions

When would we use < or > or = (for example) to compare two objects?

...when two objects have an attribute (or a combination of attributes) that makes one object "greater than"/"less than" another object.

The attribute(s) that makes one object < or > or = to another object is defined by you the programmer.

So how do we use < to compare two objects?

```
int main()
{
    Quarterback Cowboys{"Dak", 54, 35, 330, 1};
    Quarterback Texans {"Deshaun", 66, 39, 486, 3};

    if (Texans < Cowboys)
        cout << "YES - Texans < Cowboys" << endl;
    else
        cout << "NO - Texans not < Cowboys" << endl;

    return 0;
}

class Quarterback
{
public :
    Quarterback(std::string name, int att, int comp, int yds, int td)
        : qbName{name}, qbAtt{att}, qbComp{comp}, qbYds{yds}, qbTd{td}
    { }

};
```

```
// binary overload member function Demo
#include <iostream>

class Quarterback
{
public :
    Quarterback(std::string name, int att, int comp, int yds, int td)
    : qbName{name}, qbAtt{att}, qbComp{comp}, qbYds{yds}, qbTd{td}
    { }

    bool operator<(const Quarterback& QB)
    {
        std::cout << "Is " << this->qbName
            << " < " << QB.qbName << std::endl;

        if (qbAtt < QB.qbAtt &&
            qbComp < QB.qbComp &&
            qbYds < QB.qbYds &&
            qbTd < QB.qbTd)
            return true;
        else
            return false;
    }
private :
    std::string qbName;
    int qbAtt;
    int qbComp;
    int qbYds;
    int qbTd;
};

using namespace std;
```

```
int main()
{
    Quarterback Cowboys{"Dak", 54, 35, 330, 1};
    Quarterback Texans{"Deshaun", 66, 39, 486, 3};

    if (Texans < Cowboys)
        cout << "YES - Texans < Cowboys" << endl;
    else
        cout << "NO - Texans not < Cowboys" << endl;

    return 0;
}
```

```
class Quarterback
{
public :
    Quarterback(std::string name, int att, int comp, int yds, int td)
        : qbName{name}, qbAtt{att}, qbComp{comp}, qbYds{yds}, qbTd{td}
    { }

private :
    std::string qbName;
    int qbAtt;
    int qbComp;
    int qbYds;
    int qbTd;
};

};
```

```
class Quarterback
{
public :
    Quarterback(std::string name, int att, int comp, int yds, int td)
    : qbName{name}, qbAtt{att}, qbComp{comp}, qbYds{yds}, qbTd{td}
    { }

    bool operator<(Quarterback& QB)
    {
        std::cout << "Is " << this->qbName
                    << " < " << QB.qbName << std::endl;

        if (qbAtt < QB.qbAtt &&
            qbComp < QB.qbComp &&
            qbYds < QB.qbYds &&
            qbTd < QB.qbTd)
            return true;
        else
            return false;
    }
private :
    std::string qbName;
    int qbAtt;
    int qbComp;
    int qbYds;
    int qbTd;
};
```

```
Line 1     bool operator<(const Quarterback& QB)
Line 2     {
Line 3         std::cout << "Is " << this->qbName
Line 4             << " < " << QB.qbName << std::endl;
Line 5
Line 6     if (qbAtt < QB.qbAtt &&
Line 7         qbComp < QB.qbComp &&
Line 8         qbYds < QB.qbYds &&
Line 9         qbTd < QB.qbTd)
Line 10        return true;
Line 11    else
Line 12        return false;
Line 13 }
```

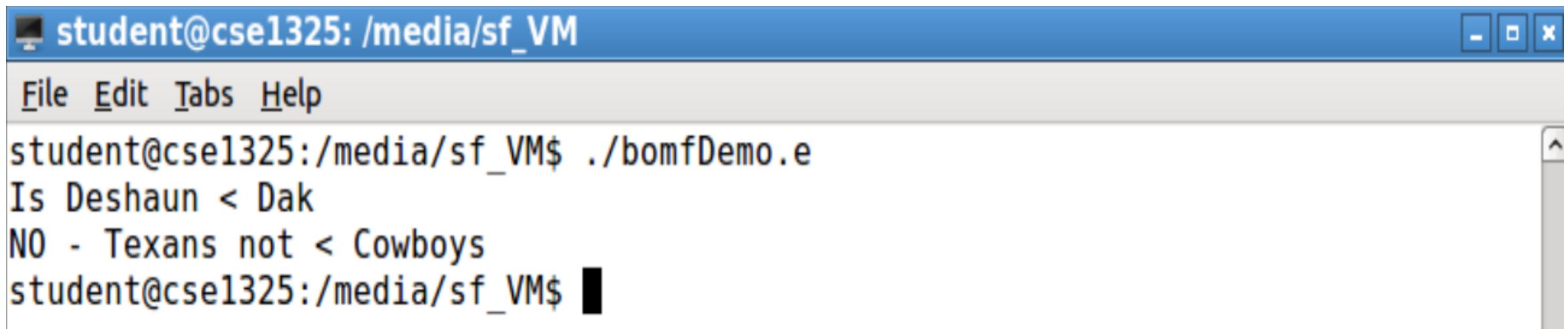
- Line 1 Overloading the < operator. Syntax is the word "operator" and the operator symbol with no space between
Returns bool which is true or false
Object QB is being passed by reference (Quarterback&) so we need to use const to make sure the object is
not changed.
- Line 3 The explicit use of this-> is not required but it is used here for illustration and clarity.
- Line 6-9 this-> is being used implicitly
- Line 10 Return true if all conditions are true
- Line 11 Return false if all conditions are false

```
Line 1     bool operator<(const Quarterback& QB)
Line 2     {
Line 3         std::cout << "Is " << this->qbName
Line 4             << " < " << QB.qbName << std::endl;
Line 5
Line 6     if (qbAtt < QB.qbAtt &&
Line 7         qbComp < QB.qbComp &&
Line 8         qbYds < QB.qbYds &&
Line 9         qbTd < QB.qbTd)
Line 10        return true;
Line 11    else
Line 12        return false;
Line 13 }
```

```
Quarterback Cowboys{"Dak", 54, 35, 330, 1};
Quarterback Texans{"Deshaun", 66, 39, 486, 3};
```

```
if (Texans < Cowboys)
    cout << "YES - Texans < Cowboys" << endl;
else
    cout << "NO - Texans not < Cowboys" << endl;
```

```
Quarterback Cowboys{"Dak", 54, 35, 330, 1};  
Quarterback Texans{"Deshaun", 66, 39, 486, 3};  
  
if (Texans < Cowboys)  
    cout << "YES - Texans < Cowboys" << endl;  
else  
    cout << "NO - Texans not < Cowboys" << endl;
```



A screenshot of a terminal window titled "student@cse1325: /media/sf_VM". The window has standard Linux-style window controls (minimize, maximize, close) at the top right. The menu bar includes "File", "Edit", "Tabs", and "Help". The terminal output shows the execution of a program named "bomfDemo.e". The program compares the statistics of two quarterbacks: Deshaun (66, 39, 486, 3) and Dak (54, 35, 330, 1). The output indicates that Deshaun is not less than Dak, so the program prints "NO - Texans not < Cowboys".

```
student@cse1325: /media/sf_VM$ ./bomfDemo.e  
Is Deshaun < Dak  
NO - Texans not < Cowboys  
student@cse1325: /media/sf_VM$ █
```

Binary Overload Operators as Non-Member Functions

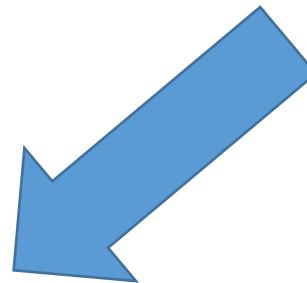
What if we moved member function out of the class and defined it as just a function in the program?

A binary operator can be overloaded as

a member function with one parameter

a non-member function with two parameters

one parameter must be either a class object or a reference to a class object



As a member function with one parameter

```
bool operator<(const Quarterback& QB)
{
    std::cout << "Is " << this->qbName
        << " < " << QB.qbName << std::endl;

    if (qbAtt < QB.qbAtt &&
        qbComp < QB.qbComp &&
        qbYds < QB.qbYds &&
        qbTd < QB.qbTd)
        return true;
    else
        return false;
}
```

As a non member function with two parameters

```
bool operator<(const Quarterback& QB1, const Quarterback& QB2)
{
    std::cout << "Is " << QB1.qbName
                << " < " << QB2.qbName << std::endl;

    if (QB1.qbAtt < QB2.qbAtt &&
        QB1.qbComp < QB2.qbComp &&
        QB1.qbYds < QB2.qbYds &&
        QB1.qbTd < QB2.qbTd)
        return true;
    else
        return false;
}
```

What happens when I make this function a non member function?

It is asking to access private member data.....

How
do
we
fix
this?

```
student@cse1325: /media/sf_VM
File Edit Tabs Help
student@cse1325:/media/sf_VM$ make
g++ -c -g -std=c++11 bonmf2Demo.cpp -o bonmf2Demo.o
bonmf2Demo.cpp: In function 'bool operator<(const Quarterback&, const Quarterbac
k&)':
bonmf2Demo.cpp:13:15: error: 'std::__cxx11::string Quarterback::qbName' is priva
te
    std::string qbName;
               ^
bonmf2Demo.cpp:22:28: error: within this context
    std::cout << "Is " << QB1.qbName
                           ^
bonmf2Demo.cpp:13:15: error: 'std::__cxx11::string Quarterback::qbName' is priva
te
    std::string qbName;
               ^
bonmf2Demo.cpp:23:28: error: within this context
        << " < " << QB2.qbName << std::endl;
                           ^
bonmf2Demo.cpp:14:7: error: 'int Quarterback::qbAtt' is private
    int qbAtt;
               ^
bonmf2Demo.cpp:25:10: error: within this context
    if (QB1.qbAtt < QB2.qbAtt &&
           ^
```

We make a friend

```
class Quarterback
{
    friend bool operator<(const Quarterback& QB1, const Quarterback& QB2);

public :
    Quarterback(std::string name, int att, int comp, int yds, int td)
        : qbName{name}, qbAtt{att}, qbComp{comp}, qbYds{yds}, qbTd{td}
    { }

private :
    std::string qbName;
    int qbAtt;
    int qbComp;
    int qbYds;
    int qbTd;
};

};
```

```
cin >> setw(x) >> MyString;
```

```
string part1;
string part2;
string part3;

cin >> setw(2) >> part1;
cin.ignore(1);

cin >> setw(2) >> part2;
cin.ignore(1);

cin >> setw(4) >> part3;
```

up to date

up-to-date

```
cout << part1 << "-" << part2 << "-" << part3 << endl;
part1 = part1 + "-" + part2 + "-" + part3 + "\n";
cout << part1;
```

Overloading >> and <<

Enter phone number in the form (555) 555-5555:
(219) 772-2387

The phone number entered was

Area code: 219

Exchange : 772

Line : 2387

(219) 772-2387

```
int main()
{
    PhoneNumber phone;

    cout << "Enter phone number in the form (555) 555-5555:" << endl;

    cin >> phone;

    cout << "\nThe phone number entered was\n";

    cout << phone << endl;

    return 0;
}
```

Enter phone number in the form (555) 555-5555:
(219) 772-2387

The phone number entered was
Area code: 219
Exchange : 772
Line : 2387
(219) 772-2387

```
istream& ignore (streamsize n = 1, int delim = EOF);
```

```
cin >> phone;  
(219) 772-2387
```

Overloading >> and <<

```
istream& operator>>(istream& input, PhoneNumber& number)  
{
```

```
    input.ignore();  
    input >> setw(3) >> number.areaCode;  
    input.ignore(2); // skip ) and space  
    input >> setw(3) >> number.exchange;  
    input.ignore(); // skip dash  
    input >> setw(4) >> number.line;  
    return input;
```

```
}
```

Overloading >> and <<

```
ostream& operator<<(ostream& output, const PhoneNumber& number)
{
    output << "Area code: " << number.areaCode
        << "\nExchange : " << number.exchange
        << "\nLine      : " << number.line << "\n"
        << "(" << number.areaCode << ")"
        << number.exchange
        << "-" << number.line << "\n";
    return output;
}
```

```
cin >> phone;
Area code: 219
Exchange : 772
Line      : 2387
(219) 772-2387
```

Overloading a Binary Operator

```
Quarterback Cowboys{ "Dak", 54, 35, 330, 1};  
Quarterback Texans{ "Deshaun", 66, 39, 486, 3};
```

overloaded member function

Is Deshaun(Texan) < Dak(Cowboy)?

```
if (Texans < Cowboys)
```

```
Texans.operator<(Cowboys)
```

```
bool operator<(const  
Quarterback& QB)
```

this-> references Texans object

overloaded friend function

Is Deshaun(Texan) < Dak(Cowboy)?

```
if (Texans < Cowboys)
```

```
operator<(Texans, Cowboys)
```

```
bool operator<(const  
Quarterback& QB1, const  
Quarterback& QB2)
```

no this available or needed

Overloaded Operators as Non-Member friend Functions

The normal use of `cin` and `cout` is

```
cin >> x;  
cout << x;
```

`cin` and `cout` are on the left side of the stream operator.

Therefore, when we overload `>>` and `<<`, we want our object to appear on the right side of the operator.

Overloaded Operators as Non-Member friend Functions

Overload operator functions for binary operators can be member functions if the left operand is an object of the class in which the function is a member.

So, if the overload functions were member functions, the syntax would be

<code>x << cin;</code>	<code>x.operator<<cin;</code>
<code>x >> cout;</code>	<code>x.operator>>cout;</code>

This syntax, while correct, could be mistaken as incorrect since it is not the "normal"/"expected" syntax.

Overloaded Operators as Non-Member friend Functions

To not use this syntax,

```
x << cin;
```

```
x >> cout;
```

```
x.operator<<cin;
```

```
x.operator>>cout;
```

we make our overload function a friend function instead of a member function and we can use put the object on the right.

```
cin >> x;
```

```
cout << x;
```

Operator Overloading

Overloading the relational operators

The test that determines if one object is less than or greater than another object is determined by the programmer. Those tests are written into the operator overload function.

Overloading the stream insertion/extraction operators

`<<` and `>>` can be overloaded to accept input or print output based on rules defined by the programmer. Those test are written into the operator overload function.

Operator Overloading

Overloading the == operator can be used to determine if two objects are equal but the definition of equal is still determined by the programmer. Using a non overloaded == will not determine if two objects are equal.

```
CokeMachine MyCokeMachine{"Bob's Coke Machine", 50, 500, 100};  
CokeMachine YourCokeMachine{"Bob's Coke Machine", 50, 500, 100};  
if (MyCokeMachine == YourCokeMachine)  
    cout >> "They are equivalent" << endl;  
else  
    cout >> "They are not equal" << endl;
```

```
g++ -c -g -std=c++11 equalobjectDemo.cpp -o equalobjectDemo.o  
equalobjectDemo.cpp: In function 'int main()':  
equalobjectDemo.cpp:20:20: error: no match for 'operator==' (operand types are '  
CokeMachine' and 'CokeMachine')  
    if (MyCokeMachine == YourCokeMachine)  
    ^
```

Standard Stream Objects

cin

istream object

"connected to" the standard input device

uses stream extraction operator >>

```
int grade;  
cin >> grade;
```

cout

ostream object

"connected to" the standard output device

```
cout << grade;
```

uses stream insertion operator <<

Standard Stream Objects

`cerr`

ostream object

"connected to" the standard error device (normally the screen)

uses stream insertion operator `<<`

outputs to object `cerr` are unbuffered

each stream insertion to `cerr` causes its output to appear immediately

```
#include <iostream>

using namespace std;

int main()
{
    cout << "Hello there. How are you?";
    cerr << "\nNot well - feeling a bit erroritable";
    cout << "Sorry to hear that";

    return 0;
}
```

```
student@maverick:/media/sf_VM/CSE1325$ ./cerrDemo.e
Not well - feeling a bit erroritable. Hello there. How are you? Sorry to hear
that. student@maverick:/media/sf_VM/CSE1325$ █
```

```
#include <iostream>

using namespace std;

int main()
{
    cout << "Hello there. How are you?" << endl;
    cerr << "\nNot well - feeling a bit erroritable. ";
    cout << "Sorry to hear that" << endl;

    return 0;
}
```

```
student@maverick:/media/sf_VM/CSE1325$ ./cerrDemo.e
Hello there. How are you?
Not well - feeling a bit erroritable. Sorry to hear that.
student@maverick:/media/sf_VM/CSE1325$ █
```

Standard Stream Objects

clog

ostream object

"connected to" the standard error device (normally the screen)

uses stream insertion operator <<

outputs to object clog are buffered

each stream insertion to clog is held in an internal
memory buffer until the buffer is filled or until the buffer is
flushed

```
#include <iostream>

using namespace std;

int main()
{
    cout << "Hello there.  How are you?";
    clog << "Not well - feeling a bit erroritable";
    cout << "Sorry to hear that";

    return 0;
}
```

```
8         cout << "Hello there.  How are you?  ";
(gdb)
9         clog << "Not well - feeling a bit erroritable.  ";
(gdb)
Not well - feeling a bit erroritable.  10          cout << "Sorry to hear that.
";
(gdb)
12         return 0;
(gdb)
13     }
(gdb)
__libc_start_main (main=0x5555555555189 <main()>, argc=1, argv=0x7fffffff0d8,
    init=<optimized out>, fini=<optimized out>, rtld_fini=<optimized out>,
    stack_end=0x7fffffff0c8) at ../csu/libc-start.c:342
342     ../csu/libc-start.c: No such file or directory.
(gdb)
Hello there.  How are you?  Sorry to hear that.  [Inferior 1 (process 11913) exited
normally]
(gdb)
The program is not being run.
```

10

CSE 1325

Week of 10/26/2020

Instructor : Donna French

```
#include <iostream>

bool CallFunA()
{
    int a = 1, b = 2;
    if (a >= b)
    {
        return true;
    }
    else if (a < b)
    {
        return false;
    }
}
```

```
int main(void)
{
    CallFunA();
    return 0;
}
```

```
ifelseDemo.cpp: In function 'bool CallFunA()':
ifelseDemo.cpp:17:1: warning: control reaches end
of non-void function [-Wreturn-type]
    17 | }
           | ^
```

Standard Stream Objects

cin

istream object

"connected to" the standard input device

uses stream extraction operator >>

```
int grade;  
cin >> grade;
```

cout

ostream object

"connected to" the standard output device

```
cout << grade;
```

uses stream insertion operator <<

Standard Stream Objects

`cerr`

ostream object

"connected to" the standard error device (normally the screen)

uses stream insertion operator `<<`

outputs to object `cerr` are unbuffered

each stream insertion to `cerr` causes its output to appear immediately

```
#include <iostream>

using namespace std;

int main()
{
    cout << "Hello there.  How are you?";
    cerr << "\nNot well - feeling a bit erroritable";
    cout << "Sorry to hear that";

    return 0;
}
```

```
student@maverick:/media/sf_VM/CSE1325$ ./cerrDemo.e
Not well - feeling a bit erroritable. Hello there. How are you? Sorry to hear
that. student@maverick:/media/sf_VM/CSE1325$ █
```

```
#include <iostream>

using namespace std;

int main()
{
    cout << "Hello there. How are you?" << endl;
    cerr << "\nNot well - feeling a bit erroritable. ";
    cout << "Sorry to hear that" << endl;

    return 0;
}
```

```
student@maverick:/media/sf_VM/CSE1325$ ./cerrDemo.e
Hello there. How are you?
Not well - feeling a bit erroritable. Sorry to hear that.
student@maverick:/media/sf_VM/CSE1325$ █
```

Standard Stream Objects

clog

ostream object

"connected to" the standard error device (normally the screen)

uses stream insertion operator <<

outputs to object clog are buffered

each stream insertion to clog is held in an internal
memory buffer until the buffer is filled or until the buffer is
flushed

```
#include <iostream>

using namespace std;

int main()
{
    cout << "Hello there.  How are you?";
    clog << "Not well - feeling a bit erroritable";
    cout << "Sorry to hear that";

    return 0;
}
```

```
8         cout << "Hello there.  How are you?  ";
(gdb)
9         clog << "Not well - feeling a bit erroritable.  ";
(gdb)
Not well - feeling a bit erroritable.  10             cout << "Sorry to hear that.
";
(gdb)
12         return 0;
(gdb)
13     }
(gdb)
__libc_start_main (main=0x5555555555189 <main()>, argc=1, argv=0x7fffffff0d8,
    init=<optimized out>, fini=<optimized out>, rtld_fini=<optimized out>,
    stack_end=0x7fffffff0c8) at ../csu/libc-start.c:342
342     ../csu/libc-start.c: No such file or directory.
(gdb)
Hello there.  How are you?  Sorry to hear that.  [Inferior 1 (process 11913) exited
normally]
(gdb)
The program is not being run.
```

scope resolution operator ::

A class's member functions can be defined outside of the class itself by using the scope resolution operator :: to "tie" the function to the class.

```
ClassName::memberFunction()
```

The ClassName:: tells the compiler that the member function is within that class's scope and its name is known to other class members.

```
class CokeMachine
{
public :
    std::string getMachineName(void)
    {
        return machineName;
    }
};
```

```
class CokeMachine
{
public :
    std::string getMachineName(void);
};

std::string           getMachineName(void)
{
    return machineName;
}
```

```
CokeMachine.h: In function 'std::__cxx11::string getMachineName()':
CokeMachine.h:137:9: error: 'machineName' was not declared in this scope
    return machineName;
               ^
makefile:17: recipe for target 'Code2_1000074079.o' failed
make: *** [Code2_1000074079.o] Error 1
```

```
class CokeMachine
{
public :
    std::string getMachineName(void)
    {
        return machineName;
    }
};
```

```
class CokeMachine
{
public :
    std::string getMachineName(int);
};

std::string CokeMachine::getMachineName(void)
{
    return machineName;
}
```

CokeMachine.h:135:13: error: prototype for 'std::__cxx11::string CokeMachine::getMachineName()' does not match any in class 'CokeMachine'
std::string CokeMachine::getMachineName(void)



UML Relationships

There are many methods of showing relationships between classes.
We are going to focus on four specific relationships.

Association 

Composition 

Aggregation 

Inheritance 

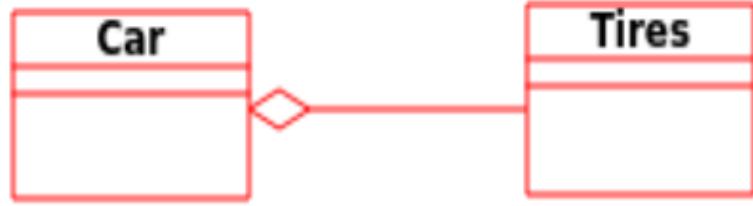
UML Relationships

Association



- Represents the "has a" relationship
- a linkage between two classes
- shows that classes are aware of each other and their relationship
- uni-directional or bi-directional

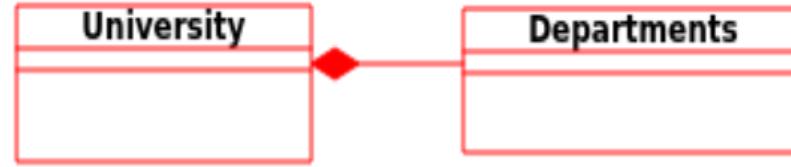




Aggregation



- Special type of association
- Represents the "has a" /"whole-part" relationship.
- Describes when a class (the whole) is composed of/has other classes (the parts)
- Diamond on "whole" side



Composition



- An association that represents a very strong aggregation
- Represents the "has a" /"whole-part" relationship.
- Describes when a class (the whole) is composed of/has other classes (the parts) BUT the parts cannot exist without the whole.
- Diamond on "whole" side

UML Relationships

Inheritance



- Represents the "is a" relationship
- Shows the relationship between a super class/base class and a derived/subclass.
- Arrow is on the side of the base class



"is a" or "has a"?

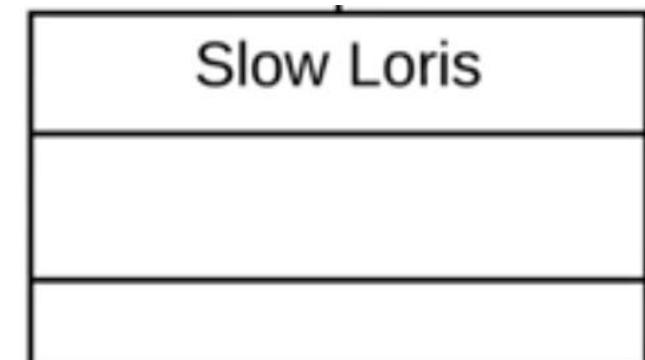
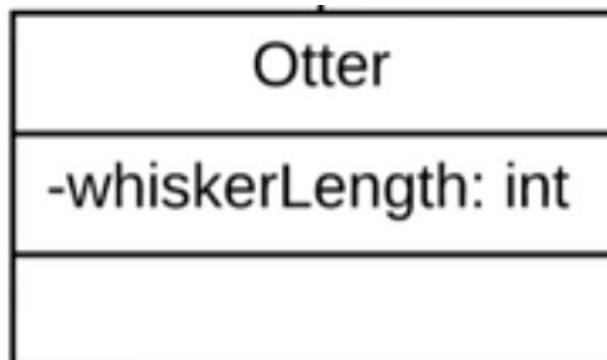
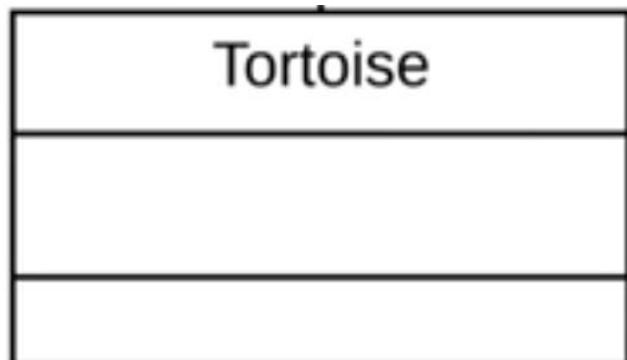
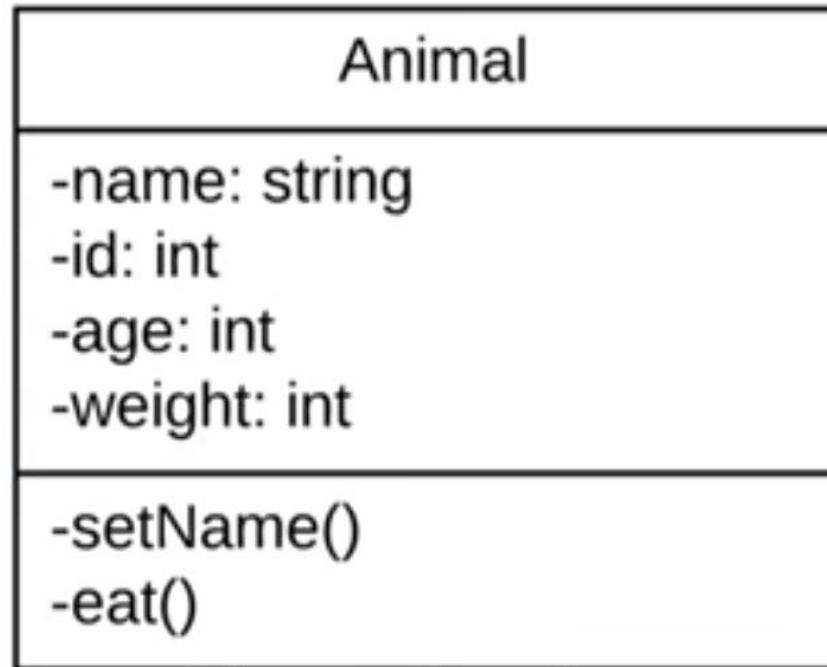
Tortoise is an Animal

Otter is an Animal

Slow Loris is an Animal

"is a" relationship

Inheritance



"is a" or "has a"?

Otter is a Sea Urchin?

Otter has a Sea Urchin?

"has a" relationship

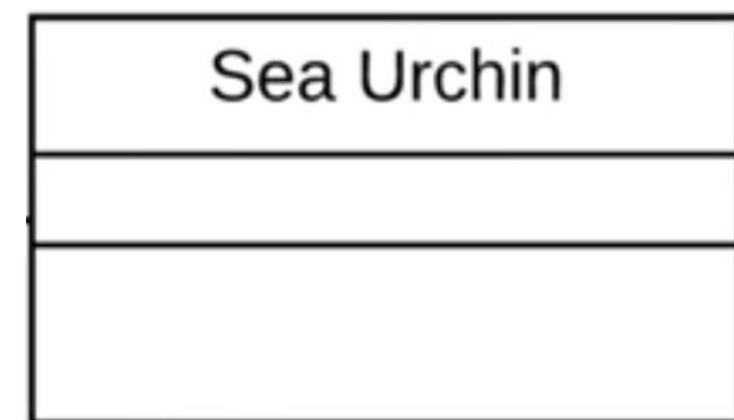
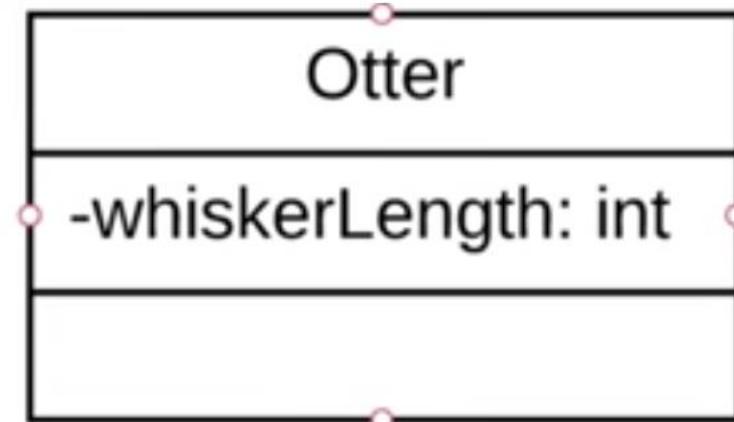
association/composition/aggregation?

"whole/part" relationship?

Otter is part of Sea Urchin?

Sea Urchin is part of Otter?

not "whole/part"



Association

"is a" or "has a"?

Creep is a Tortoise?

Creep has a Tortoise?

"has a" relationship

association/composition/aggregation?

"whole/part" relationship?

Creep is part of Tortoise?

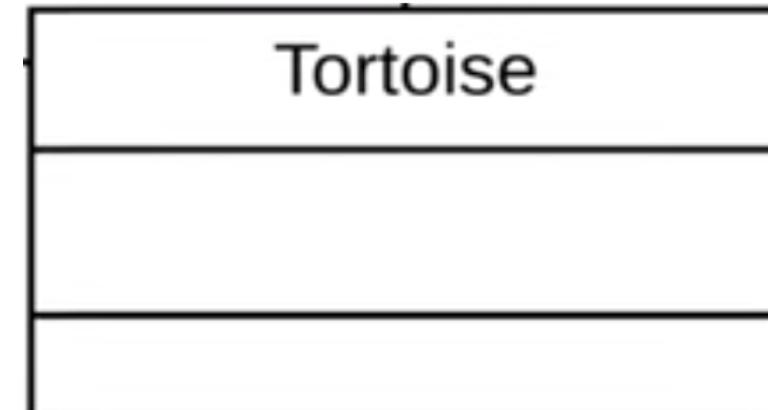
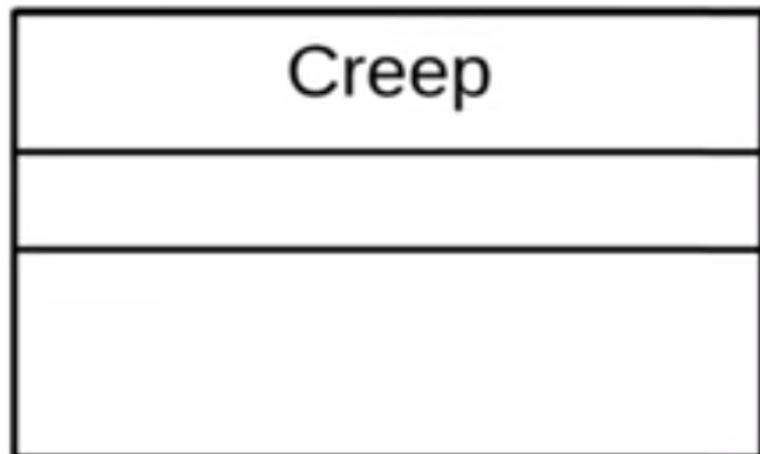
Tortoise is part of Creep?

Yes – Creep is "whole" and Tortoise is "part"

Can the Creep exist without the Tortoise?

Yes

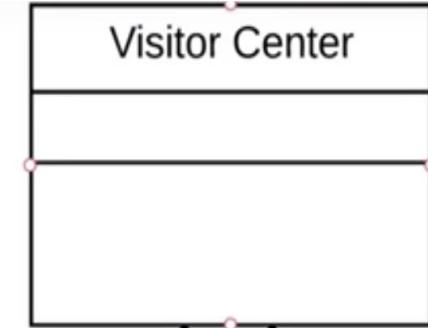
Aggregation



"is a" or "has a"?

Lobby is a Visitor Center? Bathroom is a Visitor Center?

Visitor Center is a lobby or bathroom?



Lobby/Bathroom has a Visitor Center?

Visitor Center has a Lobby and a Bathroom?

"has a" relationship

association/composition/aggregation?

"whole/part" relationship?

Visitor Center is part of Bathroom/Lobby?

Bathroom/Lobby is part of Visitor Center?



Can the "parts" – Bathroom and Lobby – exist without the "whole" - Visitor Center?

Yes – Visitor Center is "whole" and Bathroom/Lobby is "part"

Composition

namespace

A program may include many identifiers defined in different scopes.

Sometimes a variable of one scope will collide with a variable of the same name in a different scope which could possibly create a naming conflict.

Identifier overlapping occurs frequently in third-party libraries that happen to use the same names for global identifiers such as function names.

Example - multiple classes using a function named `getName()` and a private data member named `name`.

namespace

C++ solves this conflict with namespace.

Each namespace defines a scope in which identifiers and variables are placed.

To use a namespace member

member's name must be qualified with the namespace name and the scope resolution operator ()::

MyNameSpace::member

using directive must appear before the name is used in the program

```
using namespace MyNameSpace;
```

namespace

To use a namespace member

member's name must be qualified with the namespace name and the scope resolution operator (::)

```
MyNameSpace::member
```

using directive must appear before the name is used in the program

```
using namespace MyNameSpace;
```

```
std::cout << "Hello";
```

```
using namespace std;  
cout << "Hello";
```

member's name must be qualified with the namespace name and the scope resolution operator (::)

```
#include <iostream>

int main()
{
    std::string name;

    std::cout << "Please enter your name ";
    getline(std::cin, name);

    std::cout << "Your name is "
        << name << std::endl;

    return 0;
}
```

using directive must appear before the name is used in the program

```
#include <iostream>

using namespace std;

int main()
{
    string name;

    cout << "Please enter your name ";
    getline(cin, name);

    cout << "Your name is " << name << endl;

    return 0;
}
```

namespace

Creating your own namespace

```
namespace MySpace
{
    int cin;
    std::string name{"Fred"};

    void getline(int cin, std::string& Name)
    {
        Name = name;
    }
}

using namespace MySpace;
```

namespace

Nesting namespaces

```
using namespace std;
```

```
namespace MySpace
```

```
{
```

```
    int cin;
```

```
    string name{"Fred"};
```

```
    void getline(int cin, string& Name)
```

```
{
```

```
    Name = name;
```

```
}
```

```
}
```

```
using namespace MySpace;
```

Don't need to use std::string since using namespace std;
was already used.

namespace

```
using namespace std;                                int main()  
  
namespace MySpace                                {  
  
{  
  
    int cin;  
  
    string name{"Fred"};  
  
    void getline(int cin, string& Name)  
    {  
  
        Name = name;  
  
    }  
  
}  
  
using namespace MySpace;  
  
                                string name;  
  
                                cout << "Please enter your name ";  
  
                                getline(cin, name);  
  
                                cout << "Your name is " << name << endl;  
  
                                return 0;  
  
}
```

student@cse1325: /media/sf_VM

File Edit Tabs Help

```
student@cse1325:/media/sf_VM$ make
g++ -c -g -std=c++11 namespaceDemo.cpp -o namespaceDemo.o
namespaceDemo.cpp: In function 'int main()':
namespaceDemo.cpp:22:10: error: reference to 'cin' is ambiguous
    getline(cin, name);
               ^
namespaceDemo.cpp:7:6: note: candidates are: int MySpace::cin
    int cin;
               ^
In file included from namespaceDemo.cpp:1:0:
/usr/include/c++/5/iostream:60:18: note:                         std::istream std::cin
    extern istream cin; // Linked to standard input
               ^
makefile:17: recipe for target 'namespaceDemo.o' failed
make: *** [namespaceDemo.o] Error 1
```

How do we fix this error?

namespace

```
using namespace std;                                int main()
{
namespace MySpace
{
    int cin;
    string name{"Fred"};
    cout << "Please enter your name ";
    getline(cin, name);
}

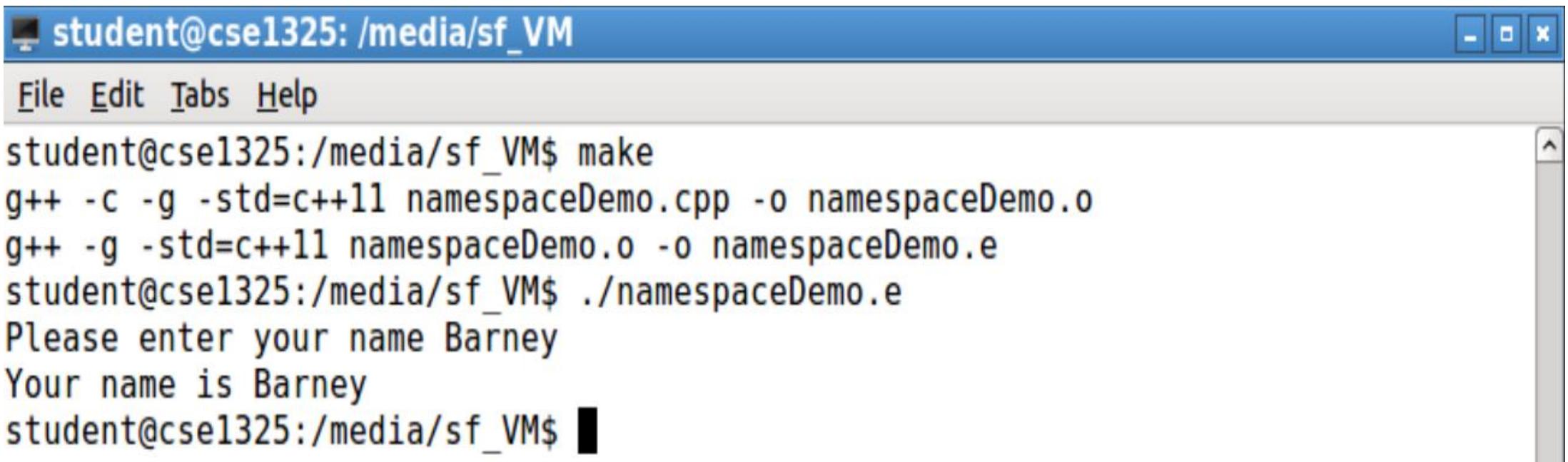
void getline(int cin, string& Name)
{
    Name = name;
    cout << "Your name is " << name << endl;
}

}
}

using namespace MySpace;
```

namespace

```
getline(cin, name);  
  
getline(std::cin, name);
```



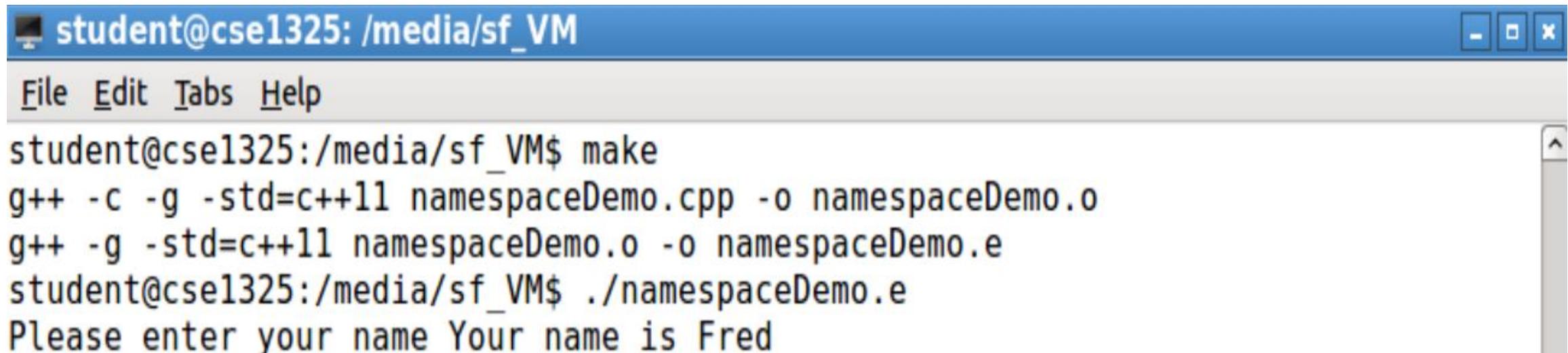
The screenshot shows a terminal window titled "student@cse1325: /media/sf_VM". The window contains the following text:

```
student@cse1325:/media/sf_VM$ make  
g++ -c -g -std=c++11 namespaceDemo.cpp -o namespaceDemo.o  
g++ -g -std=c++11 namespaceDemo.o -o namespaceDemo.e  
student@cse1325:/media/sf_VM$ ./namespaceDemo.e  
Please enter your name Barney  
Your name is Barney  
student@cse1325:/media/sf_VM$
```

namespace

```
getline(cin, name);
```

```
getline(MySpace::cin, name);
```



The screenshot shows a terminal window with the following details:

- Title Bar:** student@cse1325: /media/sf_VM
- Menu Bar:** File Edit Tabs Help
- Output:**

```
student@cse1325:/media/sf_VM$ make
g++ -c -g -std=c++11 namespaceDemo.cpp -o namespaceDemo.o
g++ -g -std=c++11 namespaceDemo.o -o namespaceDemo.e
student@cse1325:/media/sf_VM$ ./namespaceDemo.e
Please enter your name Your name is Fred
```

namespace

```
using namespace std;                                int main()
{
    namespace MySpace
    {
        int cin;
        string name{"Fred"};
        void getline(int cin, string& Name)
        {
            Name = name;
        }
    }
    using namespace MySpace;
}
```

```
    {
        string name;
        cout << "Please enter your name ";
        getline(MySpace::cin, name);
        cout << "Your name is " << name << endl;
    }
    return 0;
}
```

Please enter your name Your name is Fred

namespace

using namespace should not be placed in header files

If I defined MySpace in a header file and included the usage of

```
using namespace MySpace;
```

and then included that header file in my program, all of my program's usages of cin would need to be qualified with std:: even though I also included using namespace std;

Introduction to Exception Handling

An **exception** indicates a problem that occurs while a program executes.

Should be a problem that occurs infrequently; hence; exception.

Exception handling allows you to create fault-tolerant programs that can handle exceptions.

This may mean allowing the program to finish normally even if an exception occurred – like trying to access an out-of-range subscript in a vector.

More severe problems might require that the program notify the user of the problem and then terminate immediately.

Introduction to Exception Handling

When a function detects a problem, like trying to access a subscript out of bounds, it **throws** an exception.

We can see this if we try to if we use member function `at()` to try to access vector elements out of range.

```
vector<int> WholeNumbers={ 0,1,2,3,4 };  
  
for (int i = 0; i <= WholeNumbers.size(); i++)  
{  
    cout << WholeNumbers.at(i) << endl;  
}
```

```
vector<int> WholeNumbers={ 0,1,2,3,4 } ;  
  
for (int i = 0; i <= WholeNumbers.size(); i++)  
{  
    cout << WholeNumbers.at(i) << endl;  
}  
cout << "Even if an exception occurs, life goes on" << endl;
```

What happens when WholeNumbers.at(5) tries to print?

```
student@cse1325:/media/sf_VMs ./tryCatchDemo.e  
0  
1  
2  
3  
4
```

```
terminate called after throwing an instance of 'std::out_of_range'  
what(): vector::_range_check: __n (which is 5) >= this->size() (which is 5)  
Aborted (core dumped)
```

the program terminated after throwing
an error.

```
4
12         for (int i = 0; i <= WholeNumbers.size(); i++)
(gdb)             cout << WholeNumbers.at(i) << endl;
17
(gdb)
terminate called after throwing an instance of 'std::out_of_range'
what(): vector::_M_range_check: __n (which is 5) >= this->size() (which is 5)
```

```
Program received signal SIGABRT, Aborted.
0x00007ffff74ab428 in __GI_raise (sig=sig@entry=6)
    at ../sysdeps/unix/sysv/linux/raise.c:54
54     ../sysdeps/unix/sysv/linux/raise.c: No such file or directory.
(gdb)
```

```
Program terminated with signal SIGABRT, Aborted.
The program no longer exists.
(gdb) █
```

```
vector<int> WholeNumbers={ 0,1,2,3,4 };
```

```
for (int i = 0; i <= WholeNumbers.size(); i++)
```

```
{  
try {  
    try block – contains the code that might throw an exception
```

```
    cout << WholeNumbers.at(i) << endl;  
}
```

```
} catch (out_of_range& ex) {  
    catch block – contains the code that handles the exception
```

```
    cerr << "An exception occurred: " << ex.what() << endl;  
}
```

```
}
```

```
cout << "Even if an exception occurs, life goes on" << endl;
```

Introduction to Exception Handling

```
catch (out_of_range& ex)
{
    cerr << "An exception occurred: " << ex.what() << endl;
}
```

Accessing an `out of range` `vector` element triggers the `vector` member function `at()` to throw an exception of `out_of_range`.

When `at()` throws the exception, the code in the `try` block terminates immediately and the code in the `catch` block begins executing.

Any variables declared in a `try` block are out of scope and not accessible in the `catch` block.

Introduction to Exception Handling

```
catch (out_of_range& ex)
{
    cerr << "An exception occurred: " << ex.what() << endl;
}
```

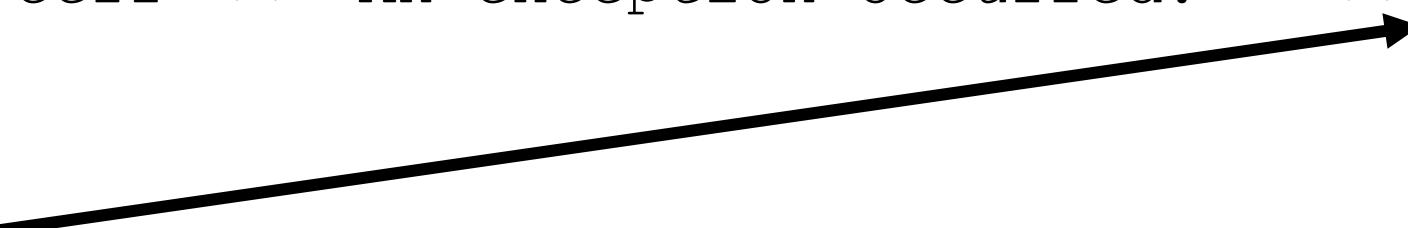
The catch block declares a type (`out_of_range`) and an exception parameter (`ex`) that it receives as a reference.

`ex` is the caught exception object

catching an exception by reference increases performance by preventing the exception object from being copied when it is caught

Introduction to Exception Handling

```
catch (out_of_range& ex)
{
    cerr << "An exception occurred: " << ex.what() << endl;
}
```



what () is a member function of the exception object

what () will get the error message that is stored in the exception object and display it

Once the message is displayed, the exception is considered handled and the program continues with the next statement after the catch block's closing brace.

```
vector<int> WholeNumbers={0,1,2,3,4};

for (int i = 0; i <= WholeNumbers.size(); i++)
{
    try
    {
        cout << WholeNumbers.at(i) << endl;
    }
    catch (out_of_range& ex)
    {
        cerr << "An exception occurred: " << ex.what() << endl;
    }
}

cout << "Even if an exception occurs, life goes on" << endl;
```

```
student@cse1325:/media/sf_VMs$ ./trycatchDemo.e
```

```
0
```

```
1
```

```
2
```

```
3
```

```
4
```

```
An exception occurred: vector::_M_range_check: __n (which is 5) >= this->size()  
(which is 5)
```

```
Even if an exception occurs, life goes on
```

The `try` block passes control to the `catch` block when `vector`'s member function `at()` throws an exception.

The `catch` block uses the exception object's `what()` member function to print out the exception and allows the program to continue and print the statement immediately after the `catch` block and the program ends normally.

In debug, we can see that the program exited normally.

```
4
13         for (int i = 0; i <= WholeNumbers.size(); i++)
(gdb)             cout << WholeNumbers.at(i) << endl;
17
(gdb)
An exception occurred: vector::_M_range_check: __n (which is 5) >= this->size()
(which is 5)
Even if an exception occurs, life goes on
[Inferior 1 (process 3462) exited normally]
```

```
(gdb)
std::vector<int, std::allocator<int> >::_M_range_check (this=0x7fffffff0c0,
    __n=5) at /usr/include/c++/5/bits/stl_vector.h:802
802         if (__n >= this->size())
(gdb)
std::vector<int, std::allocator<int> >::size (this=0x7fffffff0c0)
    at /usr/include/c++/5/bits/stl_vector.h:655
655     { return size_type(this->_M_impl._M_finish - this->_M_impl._M_star
t); }
(gdb)
std::vector<int, std::allocator<int> >::_M_range_check (this=0x7fffffff0c0,
    __n=5) at /usr/include/c++/5/bits/stl_vector.h:803
803         __throw_out_of_range_fmt(__N("vector::_M_range_check: __n "
(gdb)
std::vector<int, std::allocator<int> >::size (this=0x7fffffff0c0)
    at /usr/include/c++/5/bits/stl_vector.h:655
655     { return size_type(this->_M_impl._M_finish - this->_M_impl._M_star
t); }
(gdb)
An exception occurred: vector::_M_range_check: __n (which is 5) >= this->size()
(which is 5)
Even if an exception occurs, life goes on
[Inferior 1 (process 3466) exited normally]
```

Separating Interface from Implementation

Each of our prior class definition examples put the class in a header file for reuse and then included the header in a source code file containing main().

CokeMachine.h

Code2_1000074079.c

This arrangement of files allowed us to create and manipulate objects of the class.

This arrangement also reveals the entire implementation of the class to the class's clients since a header file is a text file that can be opened and read.

Separating Interface from Implementation

The client code actually should only know 3 things about a class

- what member functions to call
- what arguments to provide to each member function
- what return type to expect from each member function

The client code does not need to know how those functions are implemented.

Separating Interface from Implementation

When the client code does know how a class is implemented, then the programmer might write client code based on the class's implementation details.

Ideally, if the class's implementation changes, the class's clients should not have to change.

Hiding the class's implementation details makes it easier to change the class's implementation while minimizing, and hopefully, eliminating changes to the client code.

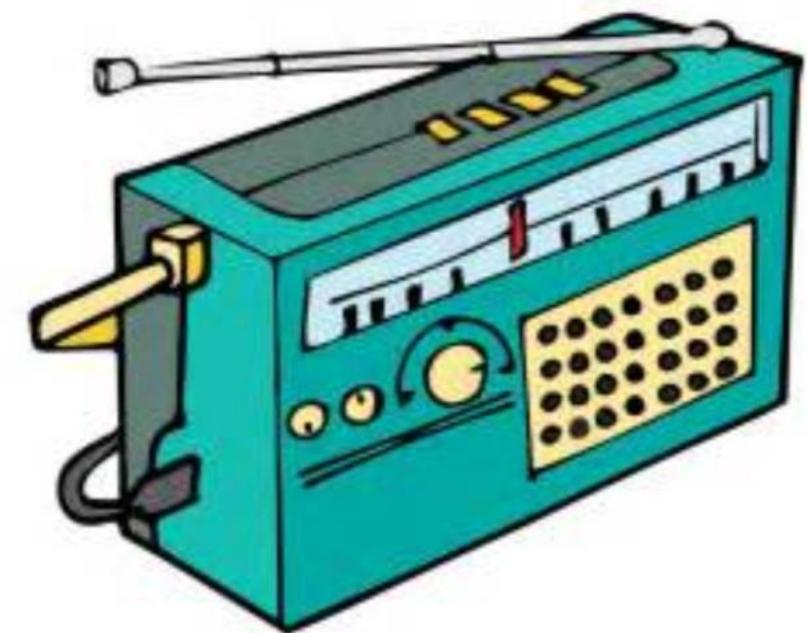
Interface of a Class

We are all familiar with a radio, and, in general, the controls that allow us to perform a limited set of operations on the radio – changing the station, adjusting the volume and choosing between AM and FM.

Some radios use dials, some have push buttons and some support voice commands.

These controls serve as the interface between the radio's users and the internal components.

The interface specifies what operations a radio permits users to perform but does not specify how the operations are implemented inside the radio.



Interface of a Class

The interface of a class describes *what* services a class's clients can use and how to *request* those services, but not *how* the class carries out the services.

A class's public interface consists of the class's public member functions which can also be known as the class's public services.

We can specify a class's interface by writing a class definition that lists only the class's member function prototypes and the class's data members.

Separating the Interface from the Implementation

To separate the class's interface from the implementation, we break up the class into two files – the header in which the class is defined and the source code file in which the class's member functions are defined.

By convention, member function definitions are placed in a source code file of the same base name as the class's header file but with a .cpp filename extension.

Doing so allows the following...

1. the class is still reusable
2. the clients of the class know what member functions the class provides, how to call them and what return types to expect
3. the clients do not know how the class's member functions are implemented

```
// Time.h

#include <string>

class Time
{
public:

private:
    unsigned int hour{0};      // 0 - 23 (24-hour clock format)
    unsigned int minute{0};    // 0 - 59
    unsigned int second{0};    // 0 - 59
};
```

```
// set new Time value using universal time
void setTime(int h, int m, int s)
{
    // validate hour, minute and second
    if ((h >= 0 && h < 24) &&
        (m >= 0 && m < 60) &&
        (s >= 0 && s < 60))
    {
        hour = h;
        minute = m;
        second = s;
    }
    else
    {
        throw invalid_argument("hour, minute and/or second was out of range");
    }
}
```

```
// return Time as a string in universal-time format (HH:MM:SS)
std::string toUniversalString() const
{
    ostringstream output;
    output << setfill('0') << setw(2) << hour << ":"
        << setw(2) << minute << ":" << setw(2) << second;
    return output.str(); // returns the formatted string
}
```

```
// return Time as string in standard-time format (HH:MM:SS AM or PM)
std::string toStandardString() const
{
    ostringstream output;
    output << ((hour == 0 || hour == 12) ? 12 : hour % 12) << ":"
        << setfill('0') << setw(2) << minute << ":" << setw(2)
        << second << (hour < 12 ? " AM" : " PM");
    return output.str(); // returns the formatted string
}
```

```
// Time.h

#include <string>

// include guard
#ifndef TIME_H
#define TIME_H

class Time
{
public:
    void setTime(int, int, int);          // set hour, minute and second
    std::string toUniversalString() const; // 24-hour time format string
    std::string toStandardString() const; // 12-hour time format string
private:
    unsigned int hour{0};    // 0 - 23 (24-hour clock format)
    unsigned int minute{0}; // 0 - 59
    unsigned int second{0}; // 0 - 59
};

#endif
```

Instead of function definitions, the class contains function prototypes that describe the class's public interface without revealing the member function implementation.

We don't have a constructor in this example but it would be included here as well.

This is enough information for the compiler to create an object (reserve enough memory) and ensure that it is called properly.

Include Guard

When we build larger programs, other definitions and declarations will also be placed in the headers.

The include guard prevents the code between the `#ifndef` and `#endif` from being `#included` if the name `TIME_H` has been defined.

When `Time.h` is `#included` the first time, the identifier `TIME_H` is not yet defined. In this case, the `#define` directive defines `TIME_H` and the preprocessor includes the `Time.h` header's contents in the `.cpp` file.

If the header is `#included` again, `TIME_H` is defined already and the code in between `#ifndef` and `#endif` is ignored by the preprocessor.

Time.cpp

```
#include <iomanip> // for setw and setfill stream manipulators
#include <stdexcept> // for invalid_argument exception class
#include <sstream> // for ostringstream class
#include <string>
#include "Time.h" // include definition of class Time from Time.h
```

```
// set new Time value using universal time
void Time::setTime(int h, int m, int s)
{
    // validate hour, minute and second
    if ((h >= 0 && h < 24) &&
        (m >= 0 && m < 60) &&
        (s >= 0 && s < 60))
    {
        hour = h;
        minute = m;
        second = s;
    }
    else
    {
        throw invalid_argument("hour, minute and/or second was out of range");
    }
}
```

throws an exception of type `invalid_argument` (from `<stdexcept>`). The `throw` statement creates a new object of type `invalid_argument`. The custom message in the `"..."` is passed to the argument's constructor.

Time.cpp

Time.cpp

```
// return Time as a string in universal-time format (HH:MM:SS)
std::string Time::toUniversalString() const
{
    ostringstream output;
    output << setfill('0') << setw(2) << hour << ":"
        << setw(2) << minute << ":" << setw(2) << second;
    return output.str(); // returns the formatted string
}
```

```
// return Time as string in standard-time format (HH:MM:SS AM or PM)
std::string Time::toStandardString() const
{
    ostringstream output;
    output << ((hour == 0 || hour == 12) ? 12 : hour % 12) << ":"
        << setfill('0') << setw(2) << minute << ":" << setw(2)
        << second << (hour < 12 ? " AM" : " PM");
    return output.str(); // returns the formatted string
}
```

setfill (n)

sticky manipulator

specifies a fill character that is displayed when an integer is output in a field wider than the number of digits in the value

the fill characters appear to the left of the digits in the number because the number is right aligned by default

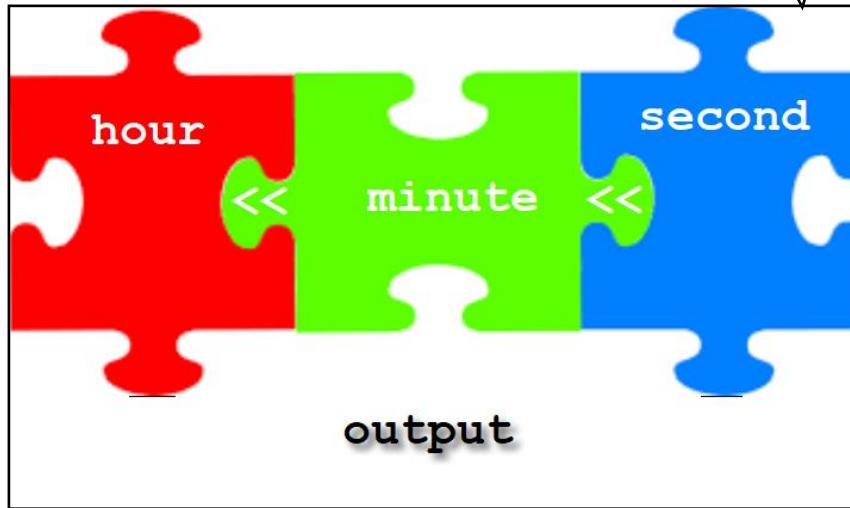
So 2 becomes 02

```
std::string Time::toUniversalString() const
```

```
{
```

```
    ostringstream output;
```

from header
↳sstream



Why const?

```
    output << setfill('0') << setw(2) << hour << ":"  
        << setw(2) << minute << ":" << setw(2) << second;
```

setw() is not sticky and setfill() is

Put the puzzle pieces of hour, minute and second together into 1 string – output.

```
    return output.str(); // returns the formatted string
```

```
}
```

str() is a member function of ostringstream that returns the string

```
// TestTime.cpp
```

```
#include <iostream>
#include <stdexcept> // for invalid_argument exception class
#include "Time.h" // include definition of class Time from Time.h
```

```
using namespace std;
```

```
// displays Time in 24-hour and 12-hour formats
void displayTime(const string& message, const Time& time)
{
    cout << message << "\nUniversal time: " << time;
    cout << "\nStandard time: " << time.toStandard();
}
```

global function

message passed by reference but marked as const. Why?

time passed by reference but marked as const. Why? but

time passed by value but marked as const. Why? but

member function of Time

UniversalString()

n

member function of Time

```
TimeTest.cpp
```

Global vs Member Functions

Member functions `toUniversalString()` and `toStandardString()` member functions take no arguments because they implicitly know that they are to create string representations of the data for a particular `Time` object on which they are invoked.

Using an object-oriented programming approach often requires fewer arguments when calling functions.

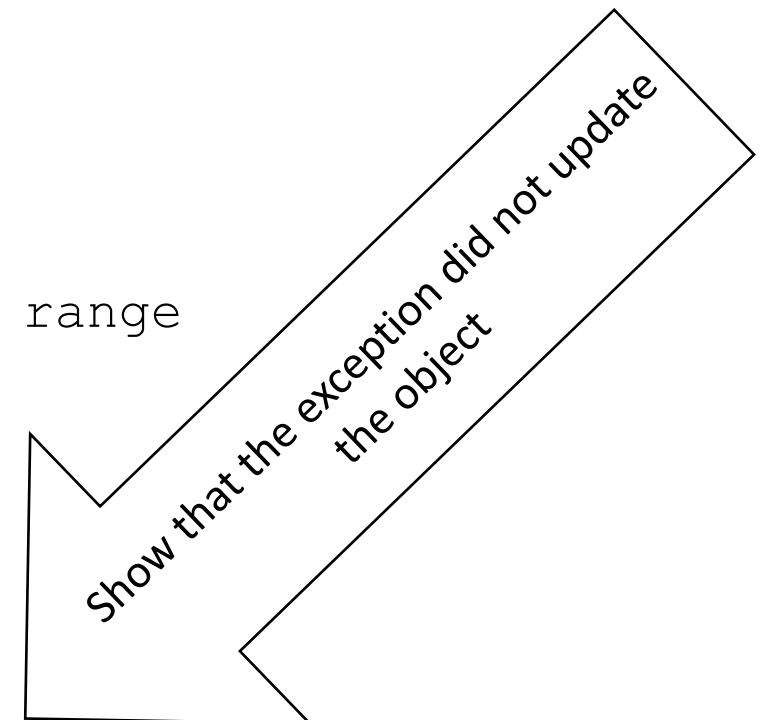
Also reduces the likelihood of passing the wrong arguments, the wrong types of arguments or the wrong number of arguments.

```
int main(void)
{
    Time MyTime;

    displayTime("Initial time:", MyTime);
    MyTime.setTime(13, 27, 6); // change time
    displayTime("After setTime:", MyTime);

    // attempt to set the time with invalid values
    try
    {
        MyTime.setTime(99, 99, 99); // all values out of range
    }
    catch (invalid_argument& say)
    {
        cout << "Exception: " << say.what() << "\n\n";
    }

    // display time value after attempting to set an invalid time
    displayTime("After attempting to set an invalid time:", MyTime);
}
```



Show that the exception did not update
the object

Initial time:

Universal time: 00:00:00

Standard time: 12:00:00 AM

After setTime:

Universal time: 13:27:06

Standard time: 1:27:06 PM

Exception: hour, minute and/or second was out of range

After attempting to set an invalid time:

Universal time: 13:27:06

Standard time: 1:27:06 PM

Compiling and Linking

Now we have three files

`Time.h` – header file with class definition and function prototypes

`Time.cpp` – member function code

`TimeTest.cpp` – a program to create `Time` objects and test them

How do we compile these into one executable?

Compiling and Linking

Seems like objects would be quite large because they contain data member and member functions. Instantiating multiple objects from a class would; therefore, take up a lot of space.

Objects only contain data so objects are much smaller than if they also contained member functions.

The compiler creates only one copy of the member functions separate from all objects of the class.

All objects of the class share this one copy.

makefile for two modules and header

```
SRC1 = TimeTest.cpp
SRC2 = Time.cpp
OBJ1 = $(SRC1:.cpp=.o)
OBJ2 = $(SRC2:.cpp=.o)
EXE = $(SRC1:.cpp=.e)

CFLAGS = -g -std=c++11

all : $(EXE)

$(EXE) : $(OBJ1) $(OBJ2)
        g++ $(CFLAGS) $(OBJ1) $(OBJ2) -o $(EXE)

$(OBJ1) : $(SRC1)
        g++ -c $(CFLAGS) $(SRC1) -o $(OBJ1)

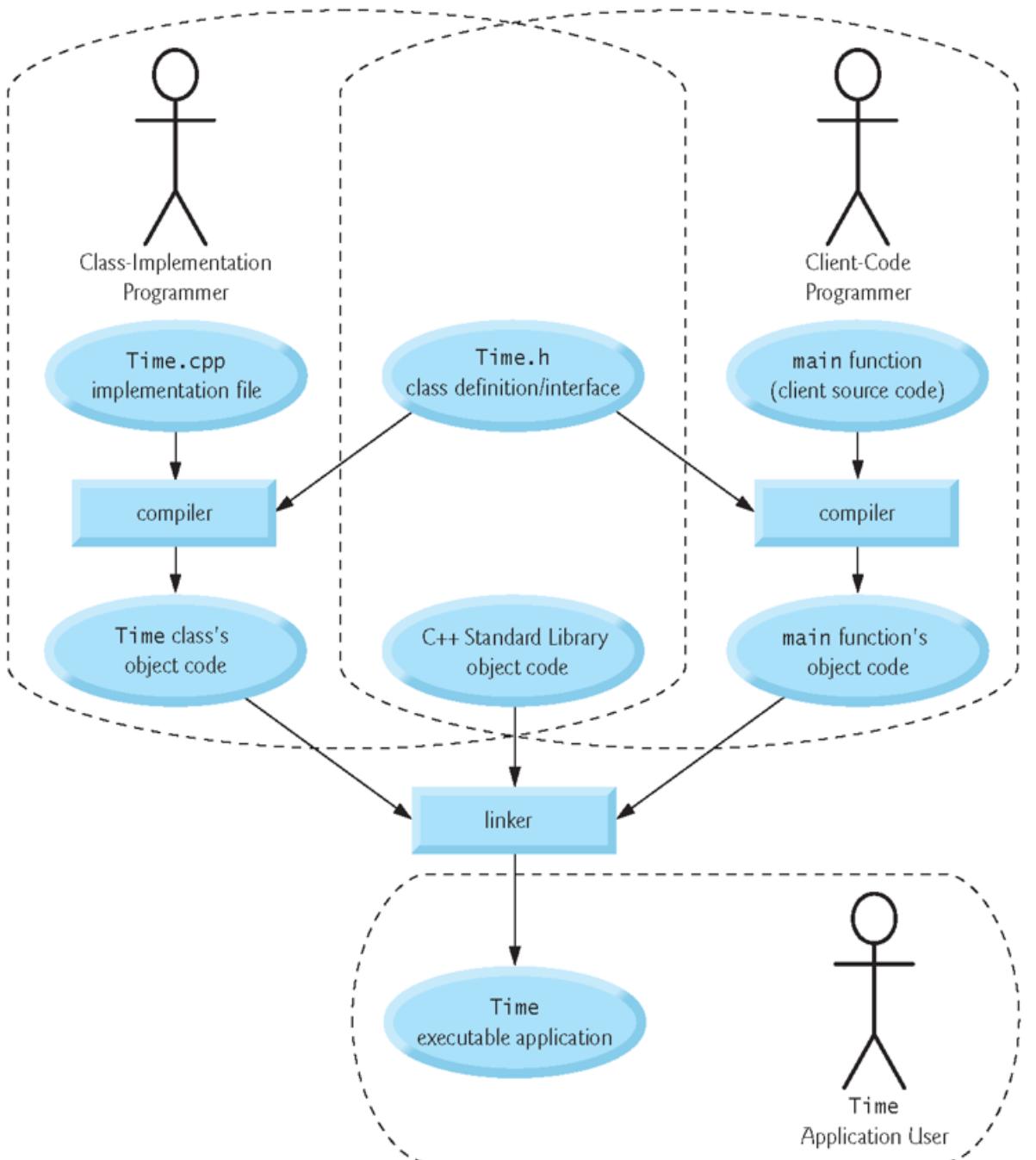
$(OBJ2) : $(SRC2)
        g++ -c $(CFLAGS) $(SRC2) -o $(OBJ2)
```

11

CSE 1325

Week of 11/02/2020

Instructor : Donna French



SRC1 = TimeTest.cpp
 SRC2 = Time.cpp
 OBJ1 = \$(SRC1:.cpp=.o)
 OBJ2 = \$(SRC2:.cpp=.o)
 EXE = \$(SRC1:.cpp=.e)

CFLAGS = -g -std=c++11

all : \$(EXE)

\$(EXE) : \$(OBJ1) \$(OBJ2)
 g++ \$(CFLAGS) \$(OBJ1) \$(OBJ2) -o \$(EXE)

\$(OBJ1) : \$(SRC1)
 g++ -c \$(CFLAGS) \$(SRC1) -o \$(OBJ1)

\$(OBJ2) : \$(SRC2)
 g++ -c \$(CFLAGS) \$(SRC2) -o \$(OBJ2)

```
// operator overload Demo

#include "Widget.h"

#include <iostream>

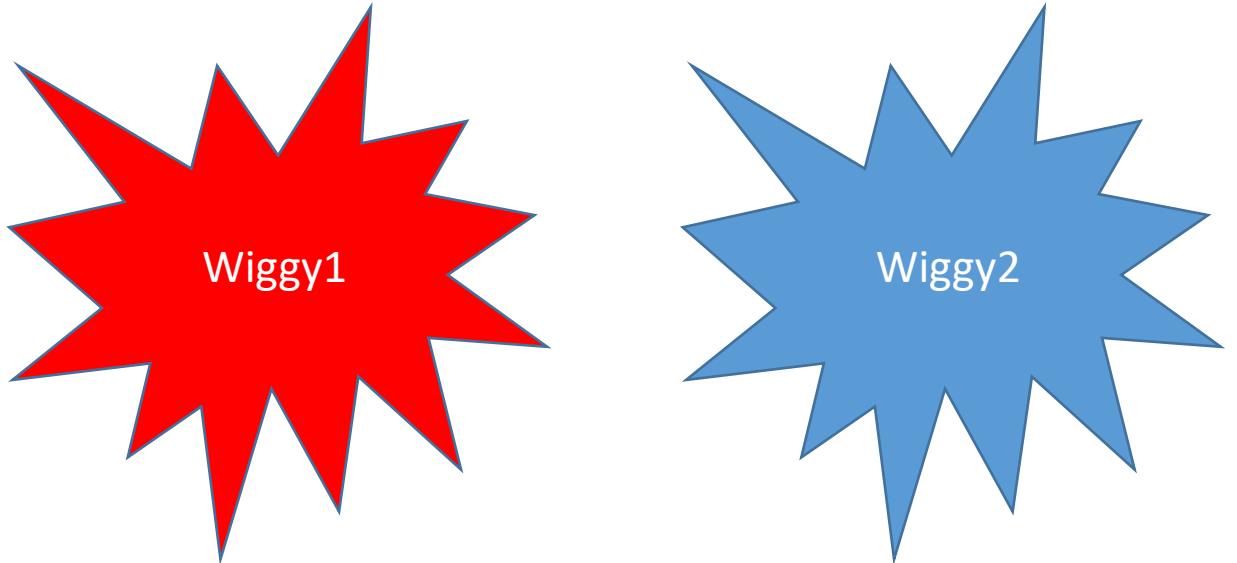
using namespace std;

int main(void)
{
    Widget W1{"Wiggy1", "red"};
    Widget W2{"Wiggy2", "blue"};

    cout << "Widgets are equivalent if they are the same color" << endl;

    if (W1 == W2)
        cout << "equivalent" << endl;
    else
        cout << "not equivalent" << endl;

    return 0;
}
```



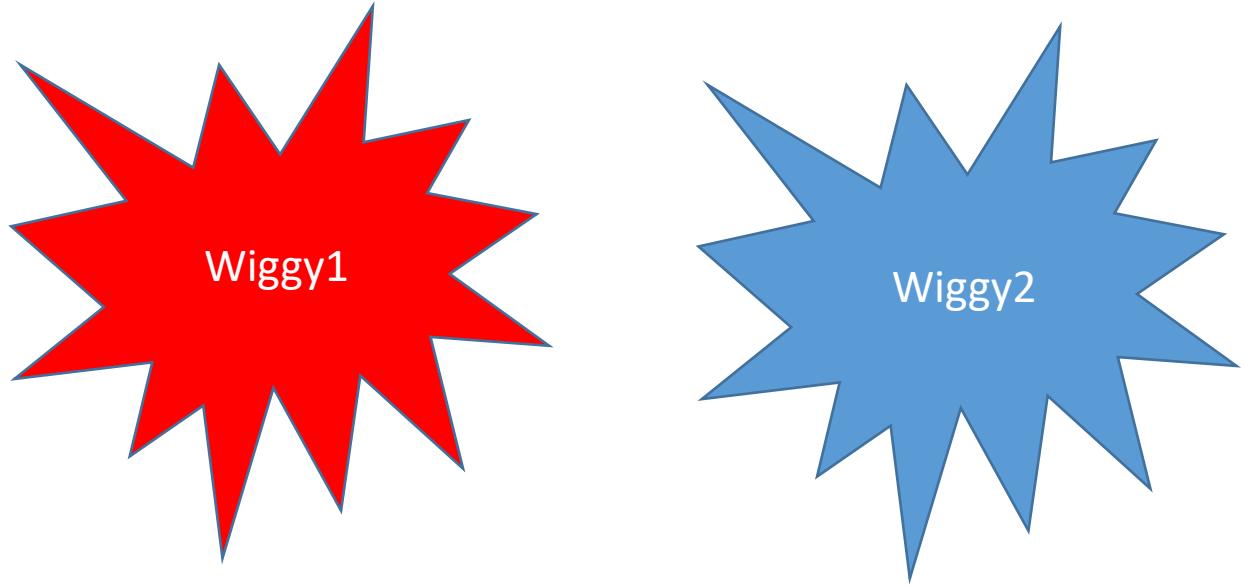
```
#include <iostream>
class Widget
{
public :
    Widget(std::string Name, std::string Coloring) : name{Name}, color{Coloring}
    {
    }

    std::string getName()
    {
        return name;
    }

    std::string getColor()
    {
        return color;
    }

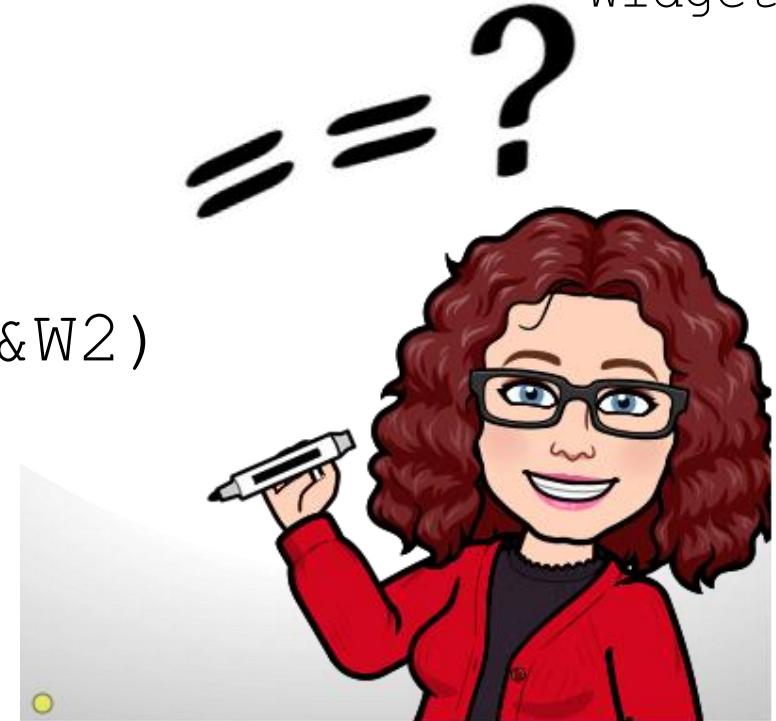
private :
    std::string name;
    std::string color;
};

Widget W1{"Wiggy1", "red"};
Widget W2{"Wiggy2", "blue"};
```



What would the operator overload function look like?

```
bool operator==(Widget &W1, Widget &W2)
{
    if (W1.color == W2.color)
        return true;
    else
        return false;
}
```



```
#include <iostream>

class Widget
{
    friend bool operator==(Widget &, Widget &);

public :
    Widget(std::string Name, std::string Coloring) : name{Name}, color{Coloring}
    {
    }

    std::string getName()
    {
        return name;
    }

    std::string getColor()
    {
        return color;
    }

private :
    std::string name;
    std::string color;
};

bool operator==(Widget &W1, Widget &W2)
{
    if (W1.color == W2.color)
        return true;
}
```

```
// operator overload Demo

#include "Widget.h"

#include <iostream>

using namespace std;

int main(void)
{
    Widget W1{"Wiggy1", "red"};
    Widget W2{"Wiggy2", "blue"};

    cout << "Widgets are equivalent if they are the same color" << endl;

    cout << W1 << W2;

    if (W1 == W2)
        cout << "\n\nnot equivalent" << endl;
    else
        cout << "\n\nnot equivalent" << endl;

    return 0;
}
```

Widgets are equivalent if they are the same color

Widget name : Wiggy1
Widget color : red
Widget name : Wiggy2
Widget color : blue

not equivalent



What would the << overload function look like?



```
std::ostream& operator<<(std::ostream& output, const Widget& Wout)
{
    output << "\nWidget name : " << Wout.name
        << "\nWidget color : " << Wout.color;

    return output;
}
```

```
#include <iostream>

class Widget
{
    friend bool operator==(Widget &, Widget &);

    friend std::ostream& operator<<(std::ostream&, const Widget&);

public :
    Widget(std::string Name, std::string Coloring) : name{Name}, color{Coloring}
    {
    }

    std::string getName()
    {
        return name;
    }

    std::string getColor()
    {
        return color;
    }

private :
    std::string name;
    std::string color;
}

std::ostream& operator<<(std::ostream& output, const Widget& Wout)
{
    output << "\nWidget name : " << Wout.name
        << "\nWidget color : " << Wout.color;

    return output;
}
```

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

std::string Widget::getName()
{
    return name;
}

std::string Widget::getColor()
{
    return color;
}

bool operator==(Widget &W1, Widget &W2)
{
    if (W1.color == W2.color)
        return true;
}

std::ostream& operator<<(std::ostream& output, const Widget& Wout)
{
    output << "\nWidget name : " << Wout.name
        << "\nWidget color : " << Wout.color;

    return output;
}
```



Constructor with Default Arguments

Like other functions, constructors can specify default arguments.

Time.h

```
Time(int = 0, int = 0, int = 0);
```

Time.cpp

```
Time::Time(int hour, int minute, int second)
{
    setTime(hour, minute, second);
}
```

```
class Time
{
public:
    Time(int = 0, int = 0, int = 0); // default constructor

    // set functions
    void setTime(int, int, int); // set hour, minute, second
    void setHour(int); // set hour (after validation)
    void setMinute(int); // set minute (after validation)
    void setSecond(int); // set second (after validation)

    // get functions
    unsigned int getHour() const; // return hour
    unsigned int getMinute() const; // return minute
    unsigned int getSecond() const; // return second

    std::string toUniversalString() const; // 24-hour time format string
    std::string toStandardString() const; // 12-hour time format string
private:
    unsigned int hour{0}; // 0 - 23 (24-hour clock format)
    unsigned int minute{0}; // 0 - 59
    unsigned int second{0}; // 0 - 59
};
```

```
void Time::setTime(int h, int m, int s)
{
    setHour(h); // set private field hour
    setMinute(m); // set private field minute
    setSecond(s); // set private field second
}
```

```
// set hour value
void Time::setHour(int h)
{
    if (h >= 0 && h < 24)
    {
        hour = h;
    }
    else
    {
        throw invalid_argument("hour must be 0-23");
    }
}
```

```
// set minute value
void Time::setMinute(int m)
{
    if (m >= 0 && m < 60)
    {
        minute = m;
    }
    else
    {
        throw invalid_argument("minute must be 0-59");
    }
}
```

```
// set second value
void Time::setSecond(int s)
{
    if (s >= 0 && s < 60)
    {
        second = s;
    }
    else
    {
        throw invalid_argument("second must be 0-59");
    }
}
```

Exception Handling

C++ uses a `throw` statement to signal that an exception or error case has occurred.

To use a `throw` statement, simply use the `throw` keyword, followed by a value of any data type you wish to use to signal that an error has occurred.

Typically, this value will be an error code, a description of the problem, or a custom exception class.

```
throw -1;  
terminate called after throwing an instance of 'int'  
Aborted (core dumped)
```

```
throw "Catch it!!!!";  
terminate called after throwing an instance of 'char const*'  
Aborted (core dumped)
```

Exception Handling

A `throw` statement acts as a signal that some kind of problem that needs to be handled has occurred.

Throwing exceptions is only one part of the exception handling process.

In C++, we use the **try** keyword to define a block of statements (called a **try block**). The try block acts as an observer looking for any exceptions that are thrown by any of the statements within the try block.

```
try
{
    throw "Catch it!!!!";
}
```

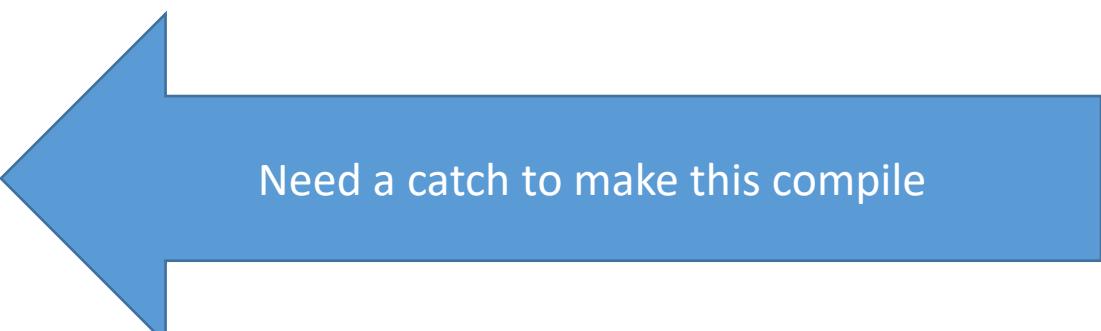
Exception Handling

```
throw "Catch it!!!";
```

```
terminate called after throwing an instance of 'char  
const*'.
```

```
Aborted (core dumped)
```

```
try  
{  
    throw "Catch it!!!";  
}
```



Need a catch to make this compile

Exception Handling

```
try
{
    throw "Catch it!!!";
}
catch(const char *message)
{
    cout << "Throw sent a message - " << message << endl;
}

cout << "Mischief managed" << endl;
```

Exception Handling



```
student@cse1325: /media/sf_VM$ ./trycatchDemo.e
Throw sent a message - Catch it!!!
Mischief managed
student@cse1325: /media/sf_VM$
```

A screenshot of a terminal window titled "student@cse1325: /media/sf_VM\$". The window shows the command "student@cse1325: /media/sf_VM\$./trycatchDemo.e" being run. The output of the program is displayed below the command, showing three lines of text: "Throw sent a message - Catch it!!!", "Mischief managed", and a final prompt "student@cse1325: /media/sf_VM\$".

The program continues to run after the exception is handled and the final message "Mischief managed" is able to print.

Exception Handling

```
try
{
    throw "Catch it!!!";
}

catch(const char *message)
{
    cout << "Throw sent a message - " << message << endl;
}

cout << "Mischief managed" << endl;
```



A screenshot of a terminal window titled 'student@cse1325: /media/sf_VM\$'. The window shows the command 'student@cse1325: /media/sf_VM\$./trycatchDemo.e' being run. The output of the program is displayed below the command: 'Throw sent a message - Catch it!!!' followed by 'Mischief managed'. The terminal prompt 'student@cse1325: /media/sf_VM\$' is shown again at the end.

```
student@cse1325: /media/sf_VM$ ./trycatchDemo.e
Throw sent a message - Catch it!!!
Mischief managed
student@cse1325: /media/sf_VM$
```

```
try
{
    cout << "Before throw..." << endl;
    throw "Catch it!!!";
    cout << "After throw..." << endl;
}

catch(const char *message)
{
    cout << "Throw sent a message - " << message << endl;
}

cout << "Mischief managed" << endl;
```

Anything after `throw` is not executed

```
student@cse1325:/media/sf_VM$ ./trycatchDemo.e
Before throw...
Throw sent a message - Catch it!!!
Mischief managed
student@cse1325:/media/sf_VM$ █
```

```
try
{
    int x = 100;
    throw "Catch it";
}

catch(const char *message)
{
    cout << "Throw sent a message - " << message << endl;
}

cout << "Mischief managed " << x << " times" << endl;
```

Any variables declared in a try block are out of scope and not accessible outside of it.

```
g++ -c -g -std=c++11 trycatchDemo.cpp -o trycatchDemo.o
trycatchDemo.cpp: In function 'int main()':
trycatchDemo.cpp:24:33: error: 'x' was not declared in this scope
    cout << "Mischief managed " << x << " times" << endl;
                                         ^
makefile:14: recipe for target 'trycatchDemo.o' failed
make: *** [trycatchDemo.o] Error 1
```

Constructor with Default Arguments

A constructor that defaults all of its arguments is also a default constructor – a constructor that can be invoked with no arguments.

There can be at most one default constructor per class.

```
Time t1;  
Time t2{2};  
Time t3{21, 34};  
Time t4{12, 25, 42};
```

Default Memberwise Assignment

The assignment operator (=) can be used to assign an object to another object of the same class.

By default, the assignment is performed by memberwise assignment which is also called **copy assignment**.

Each data member of the object on the right of the = is assigned individually to the same data member in the object on the left of the =.

```
Time MyTime;           //Instantiate objects MyTime and YourTime
Time YourTime;

MyTime.setTime(13, 27, 6); // change time

YourTime = MyTime;      // Set YourTime equal to MyTime
```

```
38          YourTime = MyTime;    ← Check value of MyTime before executing assignment operator
(gdb) p MyTime
$1 = {hour = 13, minute = 27, second = 6}

(gdb) p YourTime   ← Check value of YourTime before executing assignment operator
$2 = {hour = 0, minute = 0, second = 0}

(gdb) step        ← Execute the assignment operator

(gdb) p YourTime ← YourTime now has the same values as MyTime
$3 = {hour = 13, minute = 27, second = 6}
```

Copy Constructor

Objects may be passed as function arguments and may be returned from functions.

The default is to pass by value.

Customized copy constructors are needed for classes whose data members contain pointers to dynamically allocated memory

```
48         displayTimeCopy("Displaying YourTime", YourTime);  
(gdb) p &YourTime  
$11 = (Time *) 0x7fffffff0b0  
(gdb) step
```

Address of YourTime in main()

```
displayTimeCopy (message="Displaying YourTime", time=...) at TimeTest.cpp:18  
18 {  
(gdb) p &time  
$12 = (Time *) 0x7fffffff008
```

Address of time in displayTimeCopy()

```
void displayTimeCopy(string message, Time time)
```

Object time will be received by displayTimeCopy() as a copy (pass by value)

Address of YourTime in main() is not the same as the address of time in displayTimeCopy()

The copy constructor made a copy of YourTime

```
47         displayTime("Displaying YourTime", YourTime);
```

```
(gdb) p &YourTime
```

```
$9 = (Time *) 0x7fffffff0b0
```

Address of YourTime in main ()

```
displayTime (message="Displaying YourTime", time=...) at TimeTest.cpp:11
```

```
11 {
```

```
(gdb) p &time
```

```
$10 = (const Time *) 0x7fffffff0b0
```

Address of time in displayTime ()

```
void displayTime(const string& message, const Time& time)
```

Object time will be received by displayTimeCopy () as an address (pass by reference)

Address of YourTime in main () is the same as the address of time in displayTime ()

Copy Constructor

When an object is passed by value, C++ creates a new object and uses a **copy constructor** to copy the original object's values into the new object.

Uses memberwise assignment/copy assignment.

If you do not provide a copy constructor for your classes, C++ will create a public copy constructor for you.

It is possible to create a customized copy constructor.

Copy Constructor

```
Time MyTime;  
MyTime.setTime(13, 27, 6); // change time  
Time YourTime(MyTime);
```

27

Time YourTime (MyTime) ;

Uses default copy constructor

```
(gdb) p YourTime  
$2 = {hour = 13, minute = 27, second = 6}  
  
(gdb) p MyTime  
$3 = {hour = 13, minute = 27, second = 6}
```

Debug Note : debug
does not step into
the default copy
constructor

Time.cpp

```
Time::Time(unsigned int h, unsigned int m, unsigned int s)
    : hour{h}, minute{m}, second{s}
{
}

Time::Time(const Time &timeCopy)
    : hour{timeCopy.hour}, minute{timeCopy.minute}, second{timeCopy.second}
{
}
```

The diagram consists of two blue arrows pointing from the text "constructor" and "copy constructor" to their respective definitions in the code. The first arrow points to the first constructor definition at the top of the code. The second arrow points to the copy constructor definition in the middle of the code.

Time.h

```
Time(unsigned int=0, unsigned int=0, unsigned int=0);
Time(const Time &);
```

```
27      Time YourTime(MyTime);
```

```
(gdb) step
```

```
Time::Time (this=0x7fffffffdf8, timeCopy=...) at Time.cpp:17
```

```
17      : hour{timeCopy.hour}, minute{timeCopy.minute}, second{timeCopy.second}
```

Time::Time(const Time &timeCopy)

Time::Time(const Time timeCopy)
: hour{timeCopy.hour}, minute{timeCopy.minute}, second{timeCopy.second}
{
}

Take out &

**Time.h:13:19: error: invalid constructor; you probably meant
'Time (const Time&)'**

Time::Time(Time &timeCopy)
: hour{timeCopy.hour}, minute{timeCopy.minute}, second{timeCopy.second}
{
}

Take out const

/media/sf_VM/TimeTest.cpp:27: undefined reference to
'Time::Time(Time const&)'
Time YourTime(MyTime);

Debug Note : Debug will show you the class's definition when you use ptype

```
25          Time MyTime;
(gdb) ptype Time
type = class Time {
private:
    unsigned int hour;
    unsigned int minute;
    unsigned int second;

public:
    Time(unsigned int, unsigned int, unsigned int);
    void setTime(int, int, int);
    std::__cxx11::string toUniversalString(void) const;
    std::__cxx11::string toStandardString(void) const;
}
```

Debug Note : Debug will step into the constructor which allows you to see the default parameter values, the "this" pointer and the member initializer list.

```
25     Time MyTime;
```

```
Time::Time (this=0x7fffffffdf8, h=0, m=0,  
s=0) at Time.cpp:12
```

```
12     : hour{h}, minute{m}, second{s} {};
```

Destructors

A destructor is another type of special member function.

The name of the destructor for a class is the tilde character (~) followed by the class name.

```
~Time();
```

The tilde operator is the bitwise complement operator so it is logical that the destructor is the complement of the constructor.

A destructor may not specify parameters or a return type.

Destructors

A class's destructor is called implicitly when an object is destroyed.

An object is destroyed when program execution leaves the scope in which that object was instantiated.

The destructor itself does not actually release the object's memory but it does perform termination housekeeping before the object's memory is reclaimed by the system in order to reused to hold new objects.

Destructors

Receives no parameters and returns no value

May not specify a return type—not even void

A class may have only one destructor

Destructor overloading is not allowed

If the programmer does not explicitly provide a destructor, the compiler creates an “empty” destructor

It is a syntax error to attempt to pass arguments to a destructor, to specify a return type for a destructor (even `void` cannot be specified), to return values from a destructor or to overload a destructor.

When Constructors and Destructors Are Called

Constructors and destructors are called implicitly by the compiler

Order of these function calls depends on the order in which execution enters and leaves the scopes where the objects are instantiated

Destructor calls are made in the reverse order of the corresponding constructor calls

Storage classes of objects can alter the order in which destructors are called

When Constructors and Destructors Are Called Global Scope

Constructors are called before any other function (including `main`)

The corresponding destructors are called when `main` terminates

If the function `exit()` is used

- the program is forced to terminate immediately
- the destructors of local objects are not executed
- used to terminate a program when a fatal unrecoverable error is detected

bad things happened

If the function `abort()` is used

really bad things happened

- performs similarly to function `exit`
- forces the program to terminate immediately without allowing programmer-defined cleanup code of any kind to be called
- usually used to indicate an abnormal termination of the program

When Constructors and Destructors Are Called Non-static Local Objects

Constructor is called when the object is defined.

Corresponding destructor is called when execution leaves the object's scope.

Constructors and destructors are called each time execution enters and leaves the scope of the object.

Destructors are not called if the program terminates with an `exit()` or `abort()`.

When Constructors and Destructors Are Called static Local Objects

Constructor is called only once when execution first reaches where the object is defined.

Corresponding destructor is called when main terminates or when exit () is called.

Destructors are not called if the program terminates with an abort () .

When Constructors and Destructors Are Called

Global	non-static	static
constructors are called before any other function (including <code>main</code>)	constructors are called each time execution enters and leaves the scope of the object	constructor is called only once when execution first reaches where the object is defined
destructors are called when <code>main</code> terminates – not called with <code>exit()</code> or <code>abort()</code>	destructors are called when <code>main</code> terminates or when execution leaves the object's scope – not called with <code>exit()</code> or <code>abort()</code>	destructor is called when <code>main</code> terminates or when <code>exit()</code> is called – not called with <code>abort()</code>
destroyed in the reverse order of their creation	destroyed in the reverse order of the constructor calls	destroyed in the reverse order of their creation

```
#ifndef CREATE_H
#define CREATE_H

class CreateAndDestroy
{
public:
    CreateAndDestroy(int, std::string); // constructor
    ~CreateAndDestroy();               // destructor

private:
    int objectID;                   // ID number for object
    std::string message;            // message describing object
};

#endif
```

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

// constructor sets object's ID number and descriptive message
CreateAndDestroy::CreateAndDestroy(int ID, std::string messageString)
: objectID{ID}, message{messageString}
{
    std::cout << "\tObject " << objectID << "    constructor runs    "
          << message << std::endl;
}

// destructor
CreateAndDestroy::~CreateAndDestroy()
{
    std::cout << "\tObject " << objectID << "    destructor runs    "
          << message << std::endl;
}
```

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

using namespace std;

void CreateObject(); // prototype

// Construct a global object
CreateAndDestroy first{1, "(global before main)"}
```

first
1
(global before main)

```
CreateAndDestroy::CreateAndDestroy(int ID, std::string messageString)
: objectID{ID}, message{messageString}
{
    std::cout << "\tObject " << objectID << "    constructor runs    "
                  << message << std::endl;
}
```

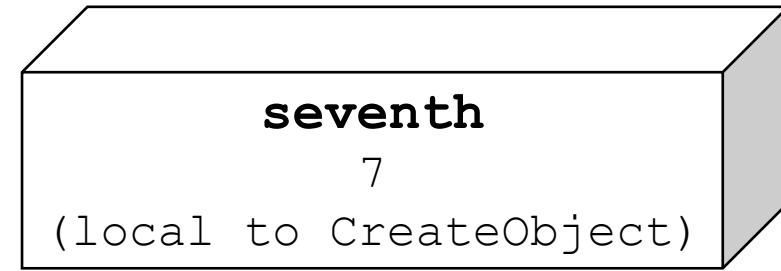
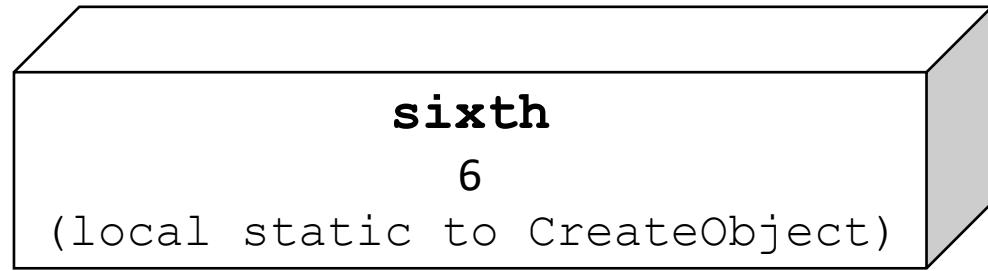
```
// function to create objects
void CreateObject()
{
    cout << "\nCreateObject() start\n" << endl;

    CreateAndDestroy fifth{5, "(local to CreateObject)"};

    static CreateAndDestroy sixth{6, "(local static to CreateObject)"};

    CreateAndDestroy seventh{7, "(local to CreateObject)"};

    cout << "\nCreateObject() finish\n" << endl;
}
```



```
int main(void)
{
    cout << "\nmain() starts\n" << endl;

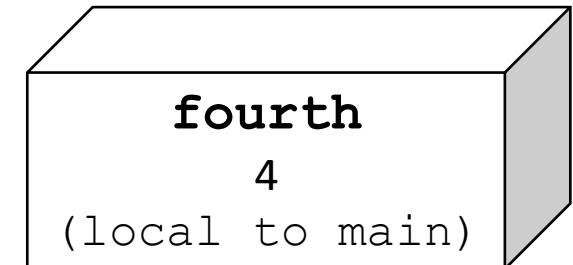
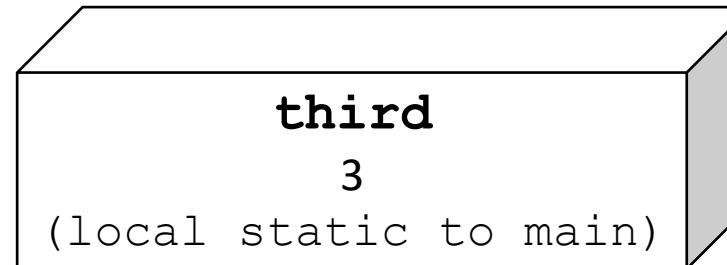
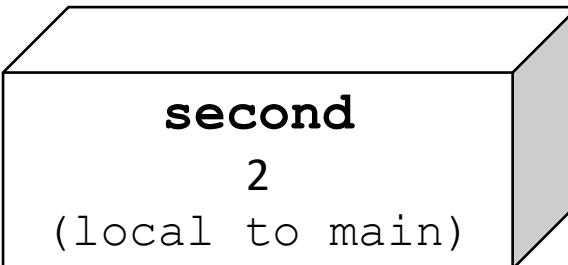
    CreateAndDestroy second{2, "(local to main)"};

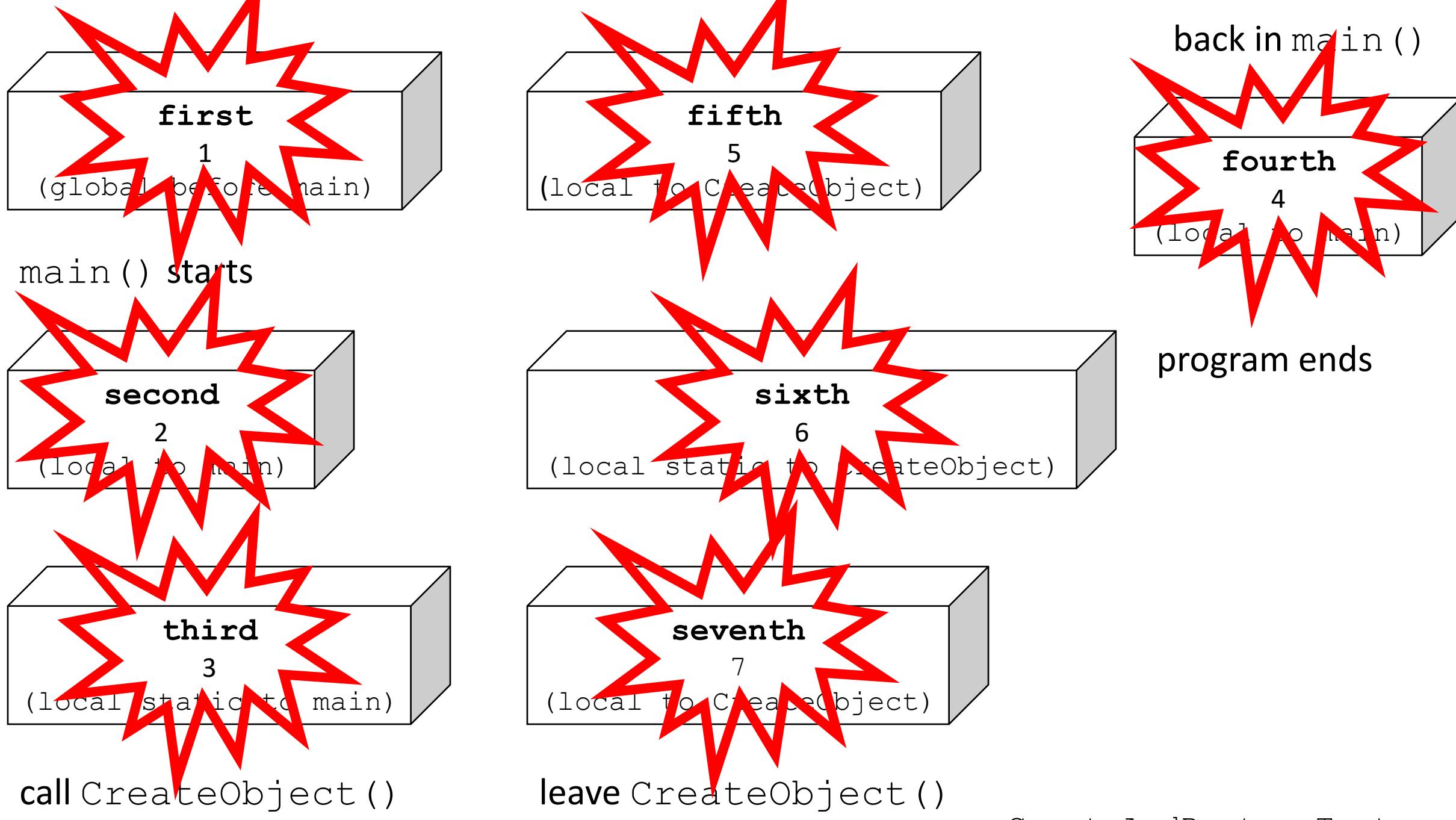
    static CreateAndDestroy third{3, "(local static to main)"};

    cout << "\ncalling CreateObject()\n" << endl;
    CreateObject();
    cout << "\nback from CreateObject()\n" << endl;

    CreateAndDestroy fourth{4, "(local to main)"};

    cout << "\nmain() finish\n" << endl;
}
```





`CreateAndDestroyTest.cpp`

Object 1 constructor runs (global before main)

main() starts

Object 2 constructor runs (local to main)

Object 3 constructor runs (local static to main)

calling CreateObject()

CreateObject() start

Object 5 constructor runs (local to CreateObject)

Object 6 constructor runs (local static to CreateObject)

Object 7 constructor runs (local to CreateObject)

CreateObject() finish

Object 7 destructor runs (local to CreateObject)
Object 5 destructor runs (local to CreateObject)

back from CreateObject()

Object 4 constructor runs (local to main)

main() finish

Object 4 destructor runs (local to main)
Object 2 destructor runs (local to main)
Object 6 destructor runs (local static to CreateObject)
Object 3 destructor runs (local static to main)
Object 1 destructor runs (global before main)

Constructing Machine Bugs Bunny

Destroying Machine Bugs Bunny

Constructing Machine Cecil Turtle

Destroying Machine Bugs Bunny

Destroying Machine Cecil Turtle

Constructing Machine Daffy Duck

Destroying Machine Bugs Bunny

Destroying Machine Cecil Turtle

Destroying Machine Daffy Duck

Constructing Machine Elmer Fudd

Destroying Machine Elmer Fudd

Constructing Machine Fog Horn

Destroying Machine Bugs Bunny

Destroying Machine Cecil Turtle

Destroying Machine Daffy Duck

Destroying Machine Elmer Fudd

Destroying Machine Fog Horn

Pick a Snack Machine

0. Exit

1. Machine Bugs Bunny

2. Machine Cecil Turtle

3. Machine Daffy Duck

4. Machine Elmer Fudd

5. Machine Fog Horn

6. Add a new machine

Enter choice 0

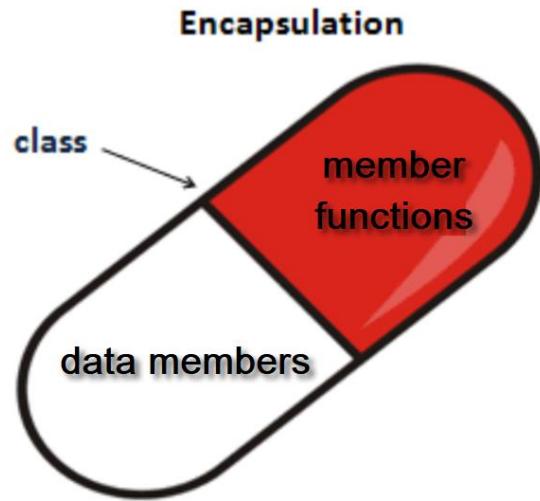
Destroying Machine Bugs Bunny

Destroying Machine Cecil Turtle

Destroying Machine Daffy Duck

Destroying Machine Elmer Fudd

Destroying Machine Fog Horn



Encapsulation

Classes wrap attributes and member functions into objects created from those classes – an object's attributes and member functions are intimately related.

Objects may communicate with one another, but they are not normally allowed to know how other objects are implemented.

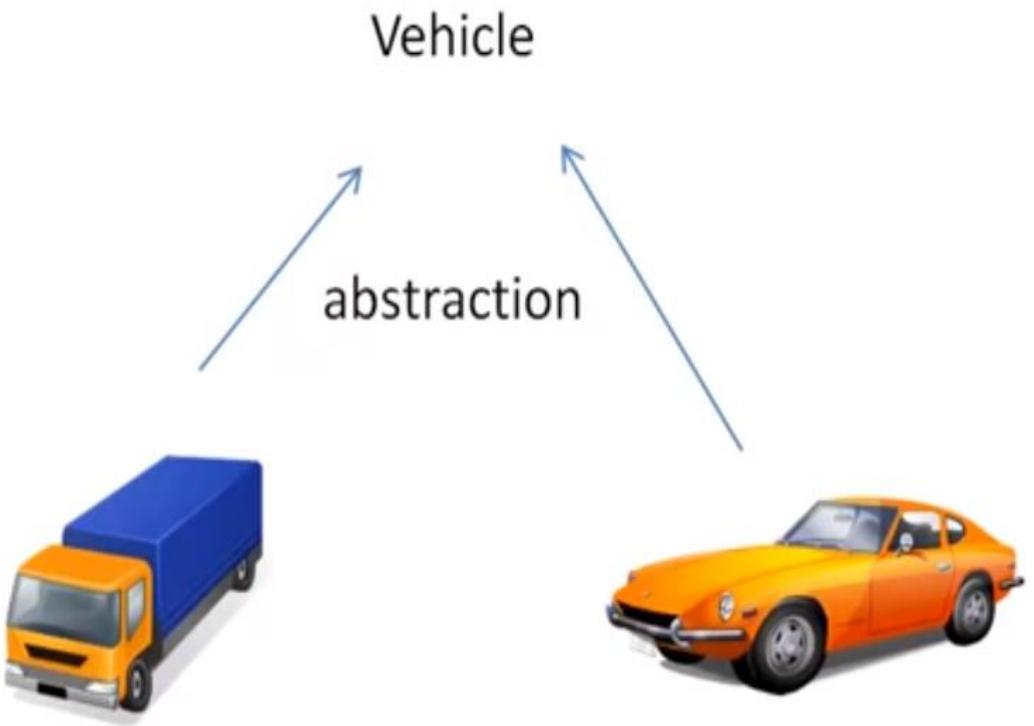
Encapsulation is the technique of information hiding - implementation details are hidden within the objects themselves.

Abstraction

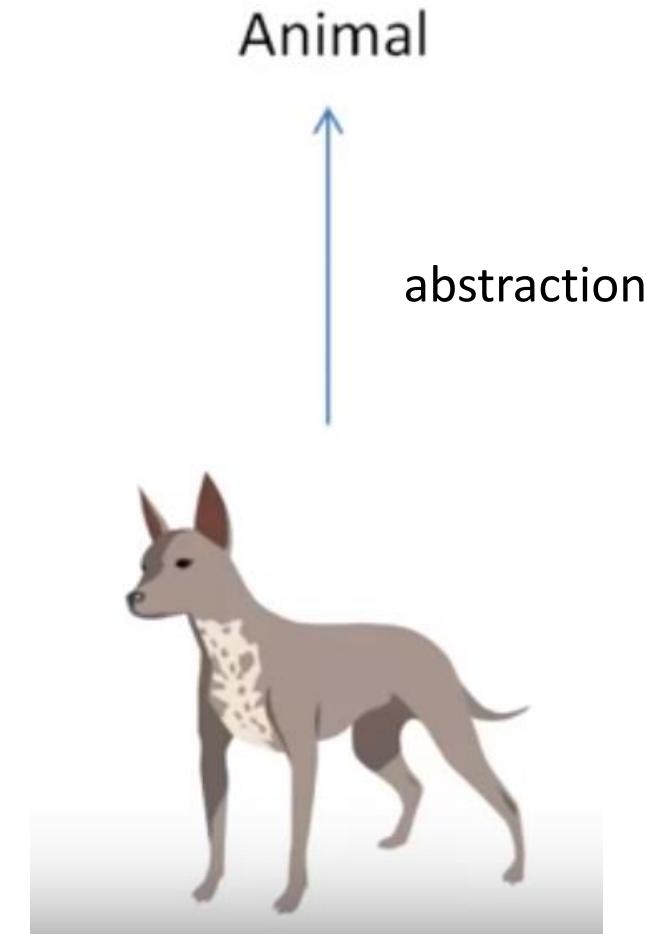
Abstraction is the concept of describing something in simpler terms, i.e abstracting away the details, in order to focus on what is important.

Abstraction is used to reduce complexity and allow efficient design and implementation of complex software systems.

Abstraction is the act of representing essential features without including the background details or explanations.



Vehicle is an abstraction of truck and car.



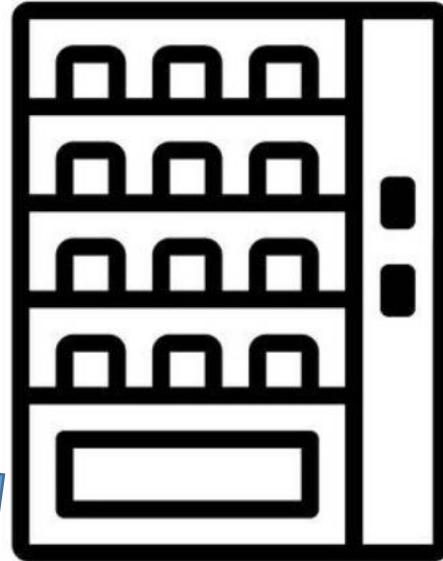
Animal is an abstraction of dog.

Vending Machine

Snack Machine



Coke Machine



Abstraction

Abstraction

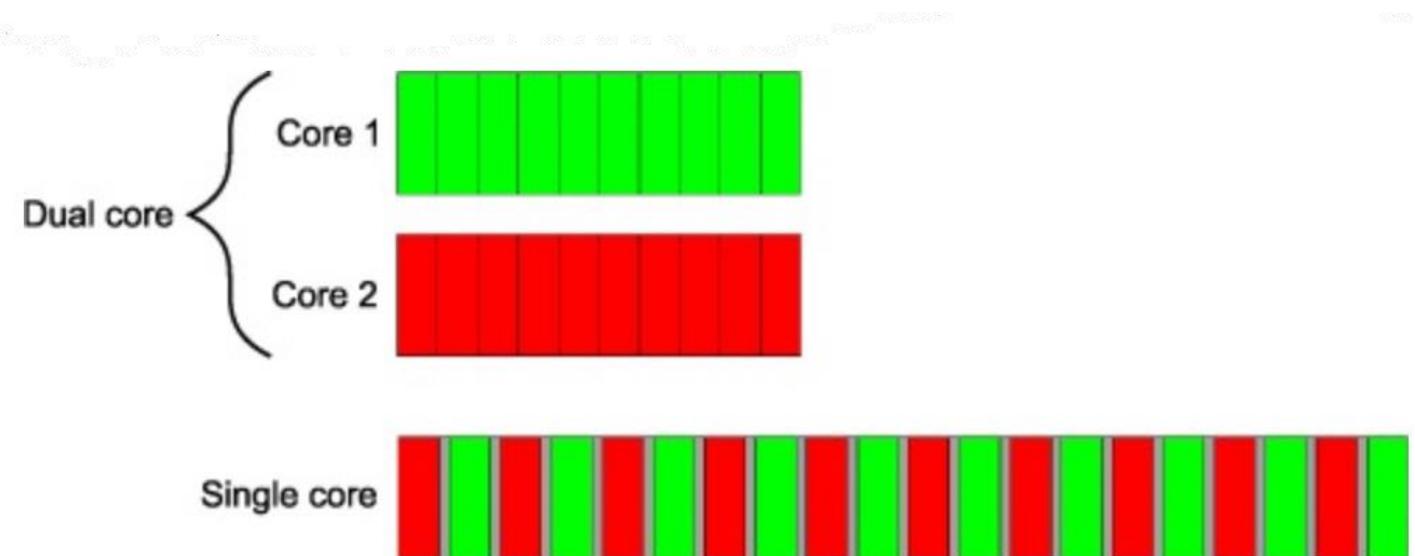
Multithreading

Most of today's computers, smartphones and tablets are typically multicore.

The most common level of multicore processor today

dual core

quad core



In multicore hardware systems, the hardware can put multiple processes to work simultaneously on different parts of your task; thereby, enabling the program to complete faster.

Multithreading

To take full advantage of multicore architecture, we need to write multithreaded applications.

When a program splits tasks into separate threads, a multicore system can run those threads in parallel.

When you run any program on a modern computer system, your program's tasks compete for the attention of the processor(s) with the operating system, other programs and other activities that the operating system is running on your behalf. All kinds of tasks are typically running in the background of your system.

Multithreading

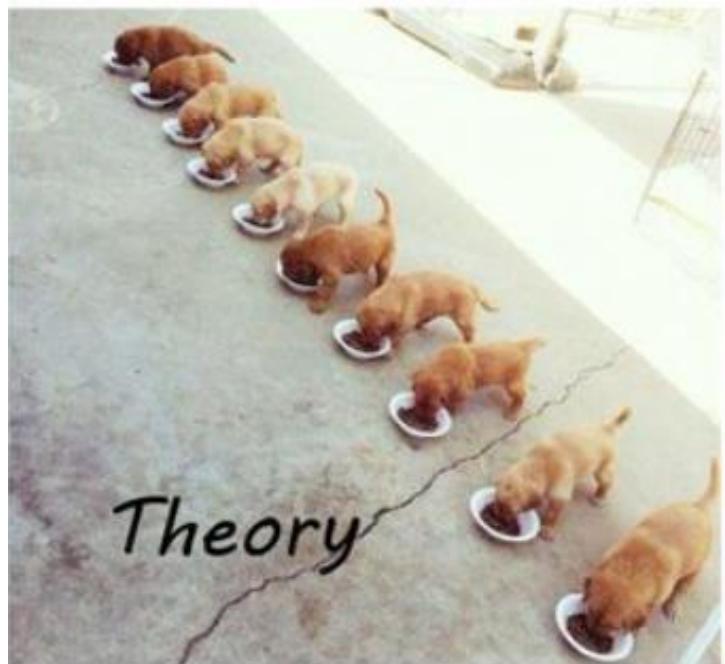
Therefore, it is important to recognize that different runs of the same process may take different amounts of time and the various threads may run in different orders at different speeds.

There's also overhead inherent to multithreading itself. Simply dividing a task into two threads and running it on a dual core system does not guarantee that it will run twice as fast.

Multithreading

There is not guarantee of which threads will execute when and how fast they will execute regardless of how the program is designed or how the processors are laid out.

Multithreaded programming



Concurrency

The ability of different parts or units of a program, algorithm, or problem to be executed out-of-order or in partial order, without affecting the final outcome.

Process

A self-contained execution environment including its own memory space.

Thread

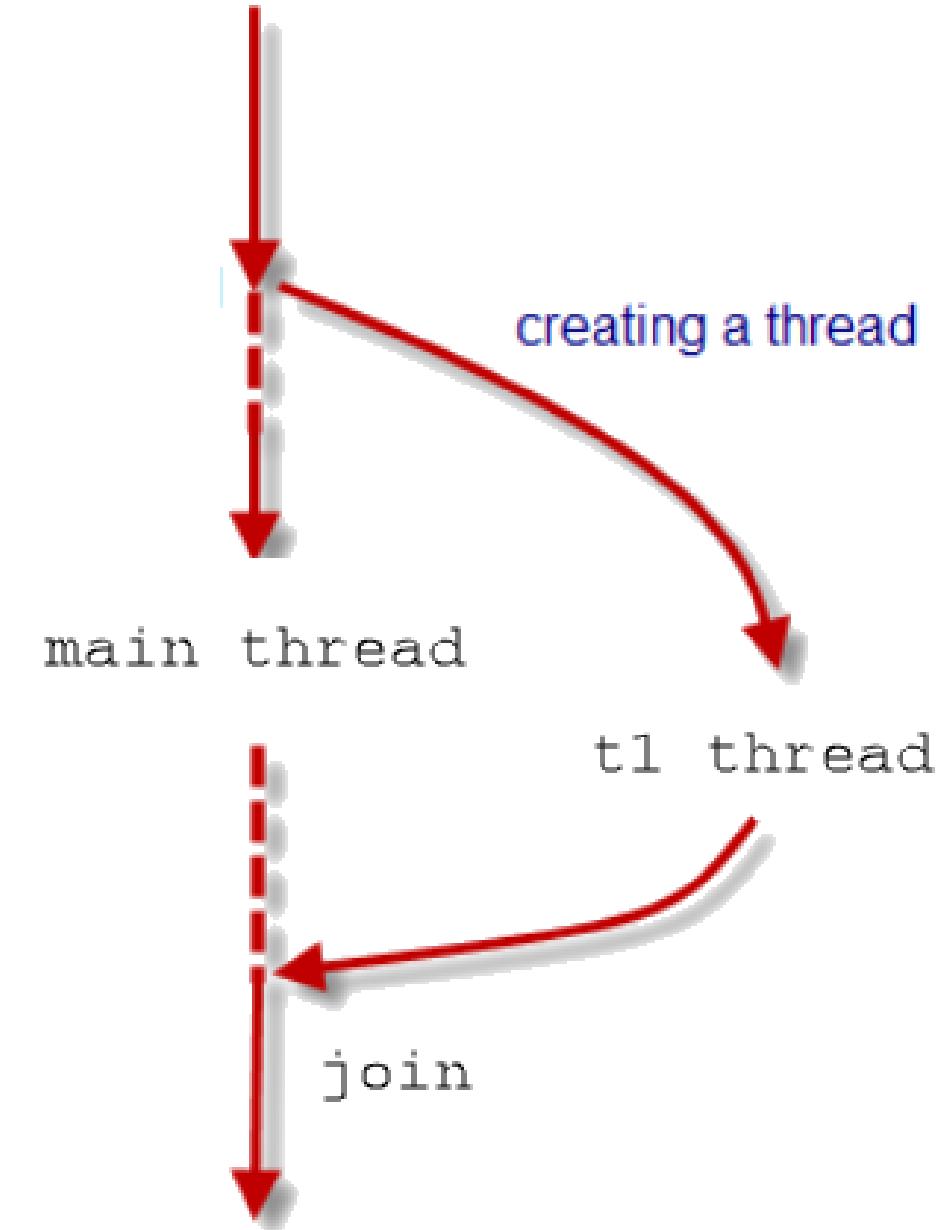
An independent path of execution within a process, running concurrently (as it appears) with other threads within a shared memory space.

Threads

Class to represent individual threads of execution.

A thread of execution is a sequence of instructions that can be executed concurrently with other such sequences in multithreading environments, while sharing a same address space.

main () is a thread



Threads

Real World Examples of Threads

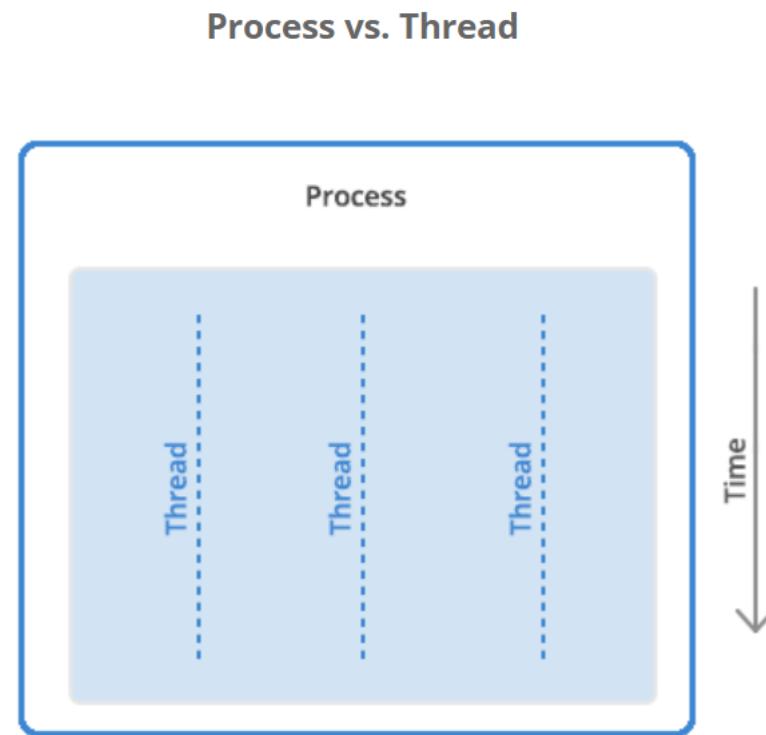
Text editor – one thread is accepting your typing, one thread is checking your spelling, one thread is occasionally saving your document. etc...

Video game – one thread is tracking your health, one thread is tracking your position, one thread is tracking your ammo, etc...

You – one thread is breathing, one thread is keeping your heart beating, one thread is falling asleep, one thread is halfway listening, etc...

Threads

A thread is the unit of execution within a process. A process can have anywhere from just one thread to many threads.



Threads

When a process starts, it is assigned memory and resources. Each thread in the process shares that memory and resources.

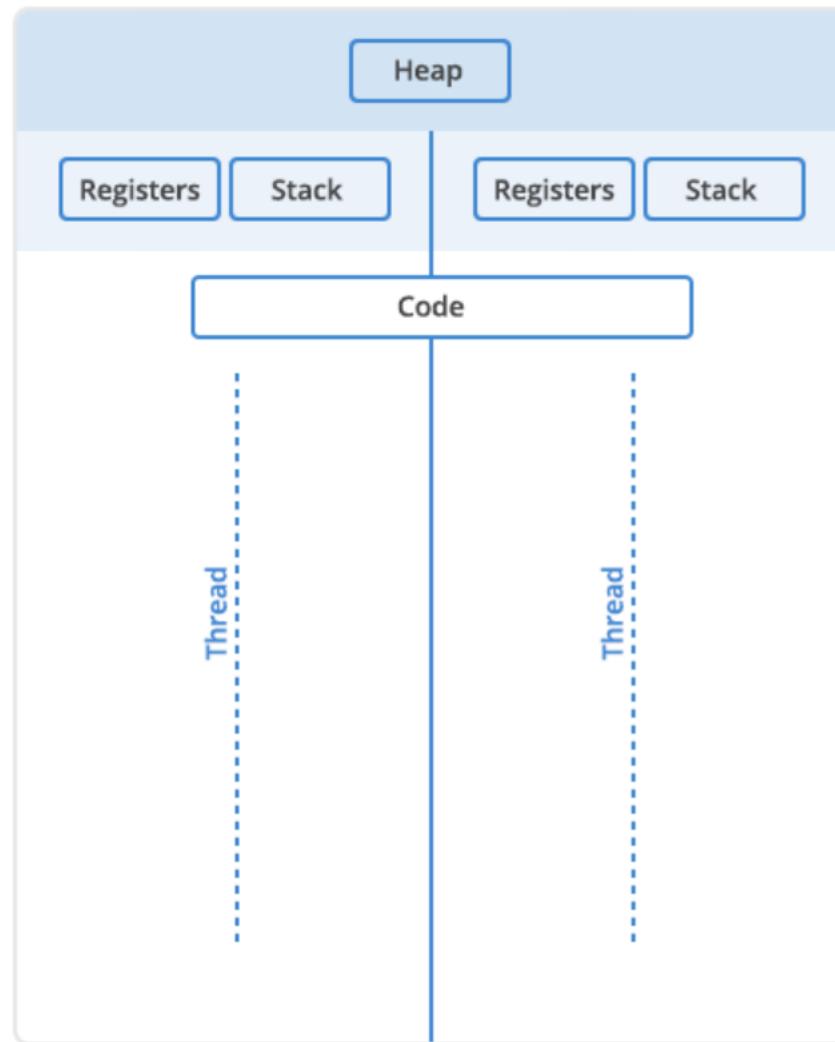
In single-threaded processes, the process contains one thread. The process and the thread are one and the same, and there is only one thing happening.

In multithreaded processes, the process contains more than one thread, and the process is accomplishing a number of things at the same time.

Single Thread



Multi Threaded



Threads

Two types of memory are available to a process or a thread

- the stack

- the heap

It is important to distinguish between these two types of process memory because

- each thread will have its own stack

- all the threads in a process will share the heap

Threads

must include <thread>

```
#makefile for multithreaded C++ program  
SRC = threadDemo.cpp  
OBJ = $(SRC:.cpp=.o)  
EXE = $(SRC:.cpp=.e)
```

```
CFLAGS = -g -std=c++11 -pthread
```

```
all : $(EXE)
```

```
$(EXE) : $(OBJ)
```

```
g++ $ (CFLAGS) $ (OBJ) -o $ (EXE)
```

must be compiled with -pthread

```
$ (OBJ) : $ (SRC)
```

```
g++ -c $ (CFLAGS) $ (SRC) -o $ (OBJ)
```

```
g++ threadDemo.cpp -pthread -g -std=c++11
```

Threads

To construct a thread, we instantiate a thread object by calling the thread initialization constructor.

This will construct a thread object that represents a new joinable thread of execution.

The new thread of execution calls the passed in function with the passed in arguments.

```
thread t1(threadT1, "Hello");
```

```
#include <iostream>
#include <thread>

using namespace std;

void threadFunction(string msg)
{
    cout << "threadFunction says " << msg << endl;
}

int main(void)
{
    //Construct a new thread and run it
    thread t1(threadFunction, "Hello");

    return 0;
}
```

Instantiate a **thread** object named **t1** using the **initialization constructor**.

Pass function "**threadFunction**" to the **constructor** along with parameter string "**Hello**".

The **thread constructor** will call **threadFunction** with parameter "**Hello**"

threadFunction ("Hello") ;

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

void threadFunction(string msg)
{
    cout << "threadFunction says "
         << msg << endl;
}

int main(void)
{
    //Construct a new thread and run it
    thread t1(threadFunction, "Hello");

    return 0;
}
```

```
student@cse1325:/media/sf_VM$ ./a.out
terminate called without an active exception
Aborted (core dumped)
```

```
Breakpoint 1, main () at threadDemo.cpp:13
13      {
(gdb) n
15          thread t1(threadFunction, "Hello");
(gdb)
[New Thread 0x7ffff6f4f700 (LWP 13497)]
threadFunction says Hello
[Thread 0x7ffff6f4f700 (LWP 13497) exited]
18          return 0;
(gdb)
15          thread t1(threadFunction, "Hello");
(gdb)
terminate called without an active exception

Thread 1 "a.out" received signal SIGABRT, Aborted.
0x00007ffff728e428 in __GI_raise (sig=sig@entry=6)
    at ../sysdeps/unix/sysv/linux/raise.c:54
54    ../sysdeps/unix/sysv/linux/raise.c: No such file or directory.
(gdb)

Program terminated with signal SIGABRT, Aborted.
The program no longer exists.
```

Threads

After the new thread has been launched,

```
thread t1(threadFunction, "Hello");
```

the initial thread (`main`) continues execution.

It does not wait for the new thread to finish and ends the program—possibly before the new thread has had a chance to run.

We need to add a call to `thread` member function `join` which will cause the calling thread (`main`) to wait for the thread associated with the `thread` object `t1`

```
#include <iostream>
#include <thread>

using namespace std;

void threadFunction(string msg)
{
    cout << "threadFunction says "
        << msg << endl;
}

int main(void)
{
    //Construct a new thread and run it
    thread t1(threadFunction, "Hello");
t1.join();
    return 0;
}
```

```
student@cse1325:/media/sf_VM$ ./a.out
threadFunction says Hello
student@cse1325:/media/sf_VM$ █
```

```
Breakpoint 1, main () at threadDemo.cpp:13
13      {
(gdb) n
15          thread t1(threadFunction, "Hello");
(gdb) n
[New Thread 0x7ffff6f4f700 (LWP 13520)]
threadFunction says Hello
[Thread 0x7ffff6f4f700 (LWP 13520) exited]
18          t1.join();
(gdb) n
20          return 0;
(gdb) n
15          thread t1(threadFunction, "Hello");
(gdb) n
21      }
(gdb) n
__libc_start_main (main=0x4011d5 <main()>, argc=1, argv=0x7fffffff1f8,
                  init=<optimized out>, fini=<optimized out>, rtld_fini=<optimized out>
                  stack_end=0x7fffffff1e8) at ../csu/libc-start.c:325
325      ../csu/libc-start.c: No such file or directory.
(gdb) n
[Inferior 1 (process 13516) exited normally]
```

Threads

An initialized thread object represents an active thread of execution and is joinable and has a unique thread id which we can obtain by calling thread member function `get_id()`.

```
thread t1(threadT1, "Hello");
thread t2(threadT2, "Hello");
thread t3(threadT3, "Hello");
thread t4(threadT4, "Hello");
thread t5(threadT5, "Hello");

cout << "t1's id is "
     << t1.get_id() << endl;
cout << "t2's id is "
     << t2.get_id() << endl;
cout << "main's id is "
     << this_thread::get_id()
     << endl;
```

t1's id is	140390222829312
t2's id is	140390214436608
main's id is	140390240180032
threadT5 says Hello	T5 i = 1
threadT4 says Hello	T4 i = 2
threadT3 says Hello	T3 i = 3
threadT2 says Hello	T2 i = 4
threadT1 says Hello	T1 i = 5

Threads

We can also ask the thread pointed at by this to give us its id.

```
void threadFunction(int x)
{
    cout << "My id = " << this_thread::get_id() << endl;
}

int main(void)
{
    int x = 0;
    thread::id main_tid = this_thread::get_id();
    thread t1(threadFunction, x);
    cout << "t1's id = " << t1.get_id() << endl;
    cout << "main's id = " << main_tid << endl;
    t1.join();

    return 0;
}
```

```
t1's id = 139717741209344
main's id = 139717759383360
My id = 139717741209344
```

Threads

A default-constructed (non-initialized) thread object is not joinable.

```
thread t6();
```

```
cout << "t6's id is "
    << t6.get_id()
    << endl;
```

```
t6.join();
```

```
student@cse1325:/media/sf_VM$ g++ threadDemo.cpp -g -std=c++11 -pthread
threadDemo.cpp: In function 'int main()':
threadDemo.cpp:53:30: error: request for member 'get_id' in 't6', which is of no
n-class type 'std::thread()'
    cout << "t6's id is " << t6.get_id() << endl;
                                         ^
threadDemo.cpp:62:5: error: request for member 'join' in 't6', which is of non-c
lass type 'std::thread()'
    t6.join();
                                         ^
```

Threads

The act of calling `join()` cleans up any storage associated with the thread.

The `thread` object is no longer associated with the now-finished thread - it isn't associated with any thread.

This means that you can call `join()` only once for a given thread.

Once you've called `join()`, the `thread` object is no longer joinable.

```
int main(void)
{
    //Construct a new thread and run it
    thread t1(threadFunction, "Hello");
    t1.join();
    t1.join();
    return 0;
}
```

terminate called after throwing an instance of
'std::system_error'
what(): Invalid argument
Aborted (core dumped)

Threads

The arguments passed to the thread's function are passed by copy by default.

```
void threadFunction(int x)
{
    x++;
    cout << "x = " << x << endl;
}

int main(void)
{
    int x = 0;

    cout << "x before = " << x << endl;
    thread t1(threadFunction, x);
    cout << "x after = " << x << endl;

    t1.join();

    return 0;
}
```

```
student@csel325:/media/sf_VMS$ ./thread1Demo.e
x before = 0
x after = 0
x = 1
```

Threads

By default, the arguments are *copied* into internal storage where they can be accessed by the newly created thread of execution, even if the corresponding parameter in the function is expecting a reference.

```
void threadFunction(int &x)
{
    x++;
    cout << "x = " << x << endl;
}

int main(void)
{
    int x = 0;

    cout << "x before = " << x << endl;
    thread t1(threadFunction, x);
    cout << "x after = " << x << endl;

    t1.join();

    return 0;
}
```

```
student@cse1325:/media/sf_VMS$ make
g++ -c -g -std=c++11 -pthread thread1Demo.cpp -o thread1Demo.o
In file included from thread1Demo.cpp:3:0:
/usr/include/c++/7/thread: In instantiation of 'struct std::thread::_Invoker<std::tuple<void (*)(int&), int> >':
/usr/include/c++/7/thread:127:22:   required from 'std::thread::thread(_Callable&&, _Args&& ...) [with _Callable = void (&)(int&); _Args = {int&}]'
thread1Demo.cpp:21:29:   required from here
/usr/include/c++/7/thread:240:2: error: no matching function for call to 'std::thread::_Invoker<std::tuple<void (*)(int&), int> >::__M_invoke(std::tuple<void (*)(int&), int> >::__Indices)'
          operator()()
          ~~~~~
/usr/include/c++/7/thread:231:4: note: candidate: template<long unsigned int ..._Ind> decltype (std::__invoke(_S_declval<_Ind>)()...)
std::thread::_Invoker<_Tuple>::__M_invoke(std::__Index_tuple<_Ind ...>) [with long unsigned int ..._Ind = {_Ind ...}; _Tuple = std::tuple<void (*)(int&), int>]
          __M_invoke(__Index_tuple<_Ind...>)
          ~~~~~
/usr/include/c++/7/thread:231:4: note:   template argument deduction/substitution failed:
/usr/include/c++/7/thread: In substitution of 'template<long unsigned int ..._Ind> decltype (std::__invoke(_S_declval<_Ind>)()...) std::thread::_Invoker<std::tuple<void (*)(int&), int> >::__M_invoke<_Ind ...>(std::__Index_tuple<_Indl ...>) [with long unsigned int ..._Ind = {0, 1}]':
/usr/include/c++/7/thread:240:2:   required from 'struct std::thread::_Invoker<std::tuple<void (*)(int&), int> >'
/usr/include/c++/7/thread:127:22:   required from 'std::thread::thread(_Callable&&, _Args&& ...) [with _Callable = void (&)(int&); _Args = {int&}]'
thread1Demo.cpp:21:29:   required from here
/usr/include/c++/7/thread:233:29: error: no matching function for call to '__invoke(std::__tuple_element_t<0, std::tuple<void (*)(int&), int> >, std::__tuple_element_t<1, std::tuple<void (*)(int&), int> >)'
          -> decltype(std::__invoke(_S_declval<_Ind>)())
          ~~~~~
In file included from /usr/include/c++/7/tuple:41:0,
                 from /usr/include/c++/7/bits/unique_ptr.h:37,
                 from /usr/include/c++/7/memory:80,
                 from /usr/include/c++/7/thread:39,
                 from thread1Demo.cpp:3:
/usr/include/c++/7/bits/Invoke.h:89:5: note: candidate: template<class _Callable, class ... _Args> constexpr typename std::__invoke_result<_Functor, _ArgTypes>::type std::__invoke(_Callable&&, _Args&& ...)
          __invoke(_Callable&& __fn, _Args&&... __args)
          ~~~~~
/usr/include/c++/7/bits/Invoke.h:89:5: note:   template argument deduction/substitution failed:
/usr/include/c++/7/bits/Invoke.h: In substitution of 'template<class _Callable, class ... _Args> constexpr typename std::__invoke_result<_Functor, _ArgTypes>::type std::__invoke(_Callable&&, _Args&& ...) [with _Callable = void (*)(int&); _Args = {int}]':
/usr/include/c++/7/thread:233:29:   required by substitution of 'template<long unsigned int ..._Ind> decltype (std::__invoke(_S_declval<_Ind>)()...')
std::thread::_Invoker<std::tuple<void (*)(int&), int> >::__M_invoke<_Ind ...>(std::__Index_tuple<_Indl ...>) [with long unsigned int ..._Ind = {0, 1}]'
/usr/include/c++/7/thread:240:2:   required from 'struct std::thread::_Invoker<std::tuple<void (*)(int&), int> >'
/usr/include/c++/7/thread:127:22:   required from 'std::thread::thread(_Callable&&, _Args&& ...) [with _Callable = void (&)(int&); _Args = {int&}]'
thread1Demo.cpp:21:29:   required from here
/usr/include/c++/7/bits/Invoke.h:89:5: error: no type named 'type' in 'struct std::__invoke_result<void (*)(int&), int>'
makefile:14: recipe for target 'thread1Demo.o' failed
make: *** [thread1Demo.o] Error 1
```

```
int main(void)
{
    string fruit;

    try
    {
        getFruit(fruit);
        cout << "You followed instructions and entered " << fruit << endl;
    }
    catch (const AppleEx &say)
    {
        cout << say.what();
    }
    catch(const OrangeEx &say)
    {
        cout << say.what();
    }

    return 0;
}
```

throw

AppleEx

OrangeEx

12

CSE 1325

Week of 11/09/2020

Instructor : Donna French

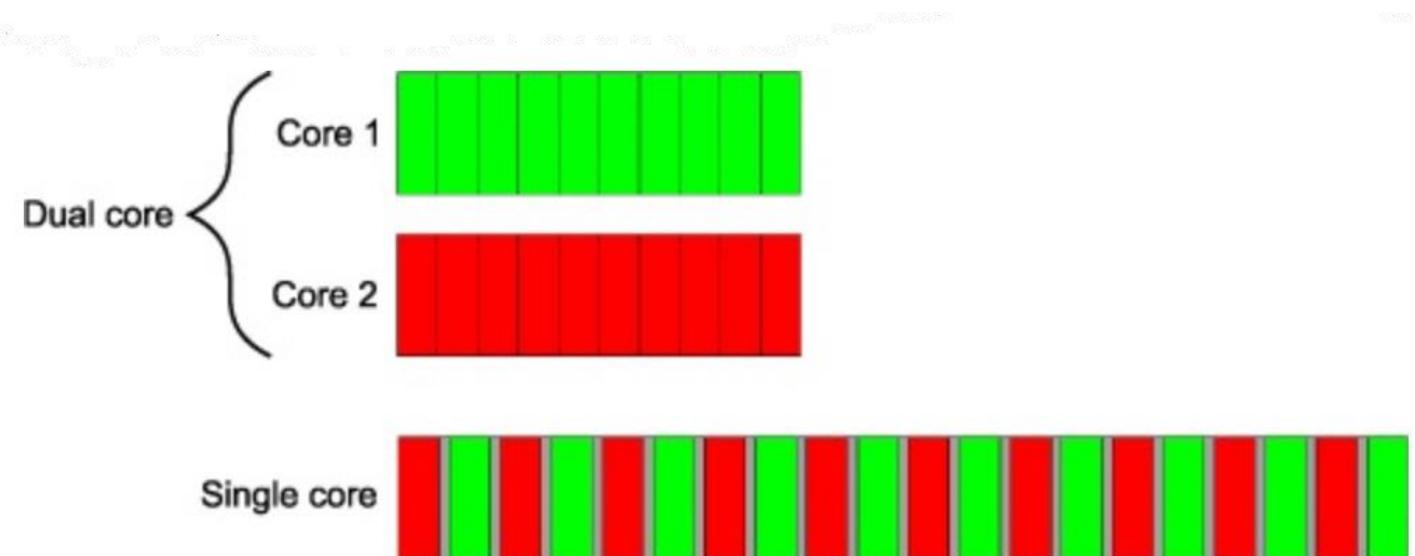
Multithreading

Most of today's computers, smartphones and tablets are typically multicore.

The most common level of multicore processor today

dual core

quad core



In multicore hardware systems, the hardware can put multiple processes to work simultaneously on different parts of your task; thereby, enabling the program to complete faster.

Multithreading

To take full advantage of multicore architecture, we need to write multithreaded applications.

When a program splits tasks into separate threads, a multicore system can run those threads in parallel.

When you run any program on a modern computer system, your program's tasks compete for the attention of the processor(s) with the operating system, other programs and other activities that the operating system is running on your behalf. All kinds of tasks are typically running in the background of your system.

Multithreading

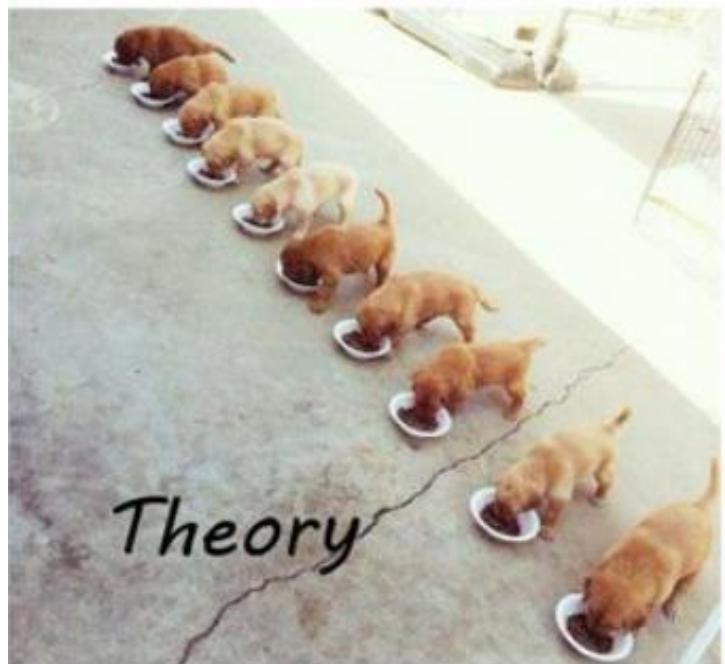
Therefore, it is important to recognize that different runs of the same process may take different amounts of time and the various threads may run in different orders at different speeds.

There's also overhead inherent to multithreading itself. Simply dividing a task into two threads and running it on a dual core system does not guarantee that it will run twice as fast.

Multithreading

There is not guarantee of which threads will execute when and how fast they will execute regardless of how the program is designed or how the processors are laid out.

Multithreaded programming



Concurrency

The ability of different parts or units of a program, algorithm, or problem to be executed out-of-order or in partial order, without affecting the final outcome.

Process

A self-contained execution environment including its own memory space.

Thread

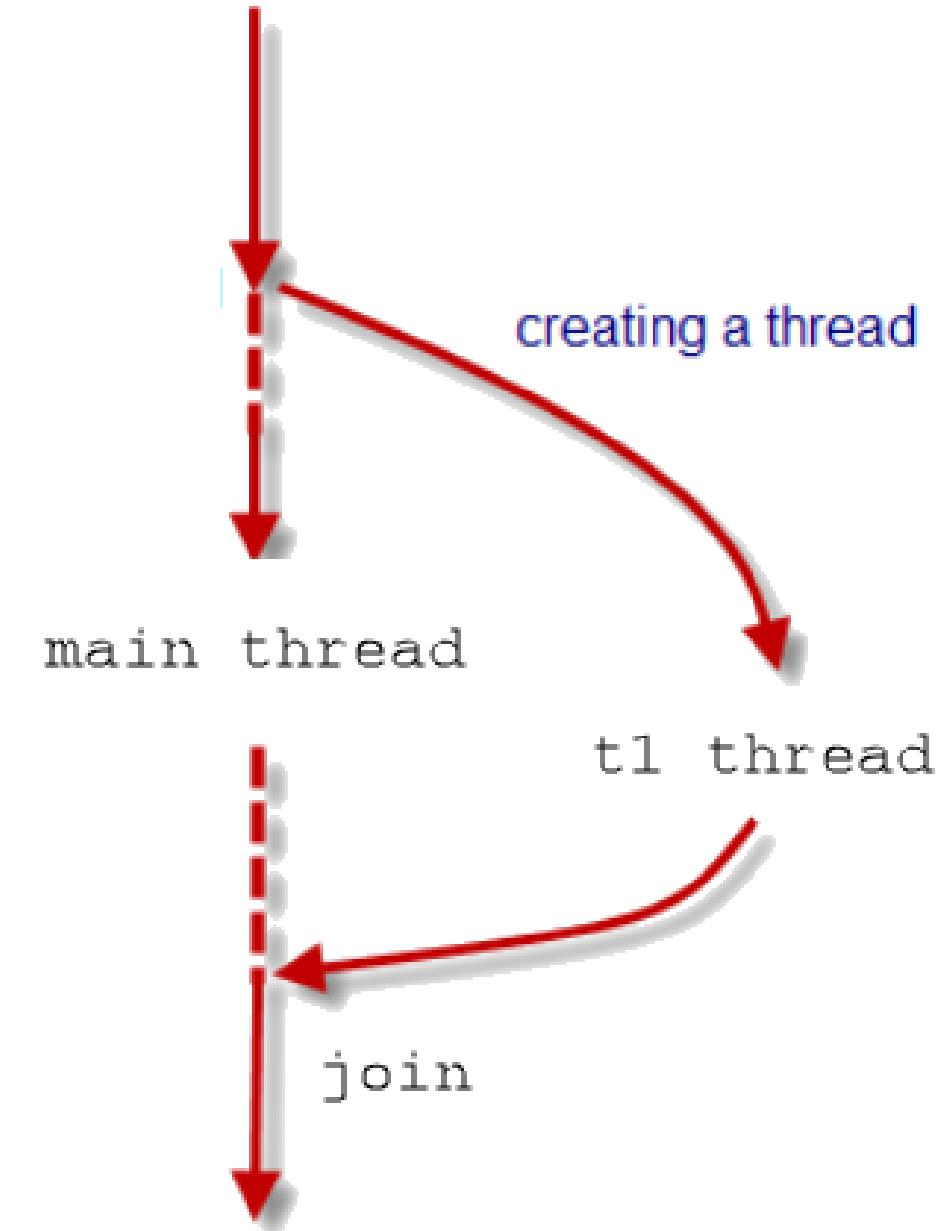
An independent path of execution within a process, running concurrently (as it appears) with other threads within a shared memory space.

Threads

Class to represent individual threads of execution.

A thread of execution is a sequence of instructions that can be executed concurrently with other such sequences in multithreading environments, while sharing a same address space.

main () is a thread



Threads

Real World Examples of Threads

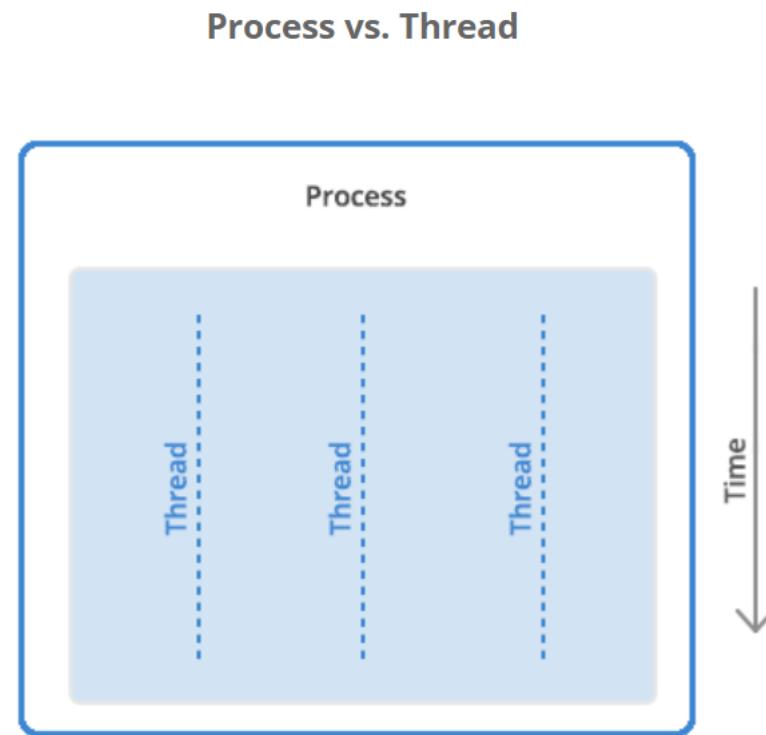
Text editor – one thread is accepting your typing, one thread is checking your spelling, one thread is occasionally saving your document. etc...

Video game – one thread is tracking your health, one thread is tracking your position, one thread is tracking your ammo, etc...

You – one thread is breathing, one thread is keeping your heart beating, one thread is falling asleep, one thread is halfway listening, etc...

Threads

A thread is the unit of execution within a process. A process can have anywhere from just one thread to many threads.



Threads

When a process starts, it is assigned memory and resources. Each thread in the process shares that memory and resources.

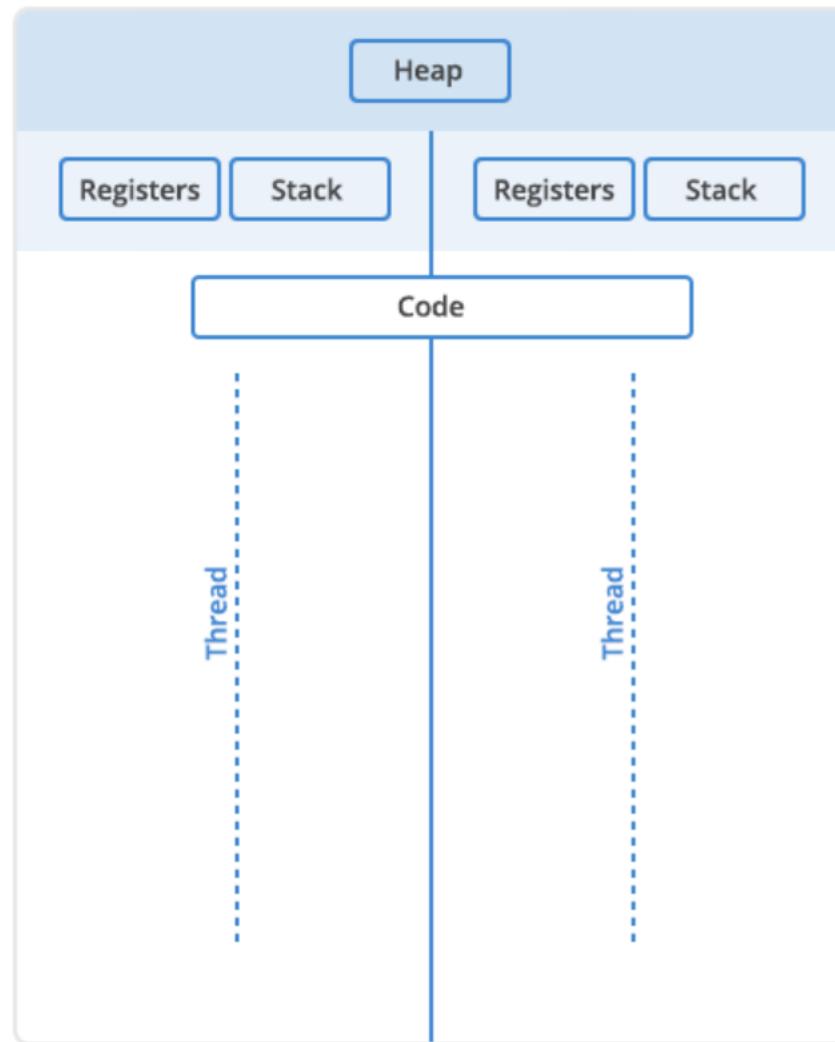
In single-threaded processes, the process contains one thread. The process and the thread are one and the same, and there is only one thing happening.

In multithreaded processes, the process contains more than one thread, and the process is accomplishing a number of things at the same time.

Single Thread



Multi Threaded



Threads

Two types of memory are available to a process or a thread

- the stack

- the heap

It is important to distinguish between these two types of process memory because

- each thread will have its own stack

- all the threads in a process will share the heap

Threads

must include <thread>

```
#makefile for multithreaded C++ program  
SRC = threadDemo.cpp  
OBJ = $(SRC:.cpp=.o)  
EXE = $(SRC:.cpp=.e)
```

```
CFLAGS = -g -std=c++11 -pthread
```

```
all : $(EXE)
```

```
$(EXE) : $(OBJ)
```

```
g++ $ (CFLAGS) $ (OBJ) -o $ (EXE)
```

must be compiled with -pthread

```
$ (OBJ) : $ (SRC)
```

```
g++ -c $ (CFLAGS) $ (SRC) -o $ (OBJ)
```

```
g++ threadDemo.cpp -pthread -g -std=c++11
```

Threads

To construct a thread, we instantiate a thread object by calling the thread initialization constructor.

This will construct a thread object that represents a new joinable thread of execution.

The new thread of execution calls the passed in function with the passed in arguments.

```
thread t1(threadT1, "Hello");
```

```
#include <iostream>
#include <thread>

using namespace std;

void threadFunction(string msg)
{
    cout << "threadFunction says " << msg << endl;
}

int main(void)
{
    //Construct a new thread and run it
    thread t1(threadFunction, "Hello");

    return 0;
}
```

Instantiate a **thread** object named **t1** using the **initialization constructor**.

Pass function "**threadFunction**" to the **constructor** along with parameter string "**Hello**".

The **thread constructor** will call **threadFunction** with parameter "**Hello**"

threadFunction ("Hello") ;

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

void threadFunction(string msg)
{
    cout << "threadFunction says "
        << msg << endl;
}

int main(void)
{
    //Construct a new thread and run it
    thread t1(threadFunction, "Hello");

    return 0;
}
```

```
student@cse1325:/media/sf_VM$ ./a.out
terminate called without an active exception
Aborted (core dumped)
```

```
Breakpoint 1, main () at threadDemo.cpp:13
13      {
(gdb) n
15          thread t1(threadFunction, "Hello");
(gdb)
[New Thread 0x7ffff6f4f700 (LWP 13497)]
threadFunction says Hello
[Thread 0x7ffff6f4f700 (LWP 13497) exited]
18          return 0;
(gdb)
15          thread t1(threadFunction, "Hello");
(gdb)
terminate called without an active exception

Thread 1 "a.out" received signal SIGABRT, Aborted.
0x00007ffff728e428 in __GI_raise (sig=sig@entry=6)
    at ../sysdeps/unix/sysv/linux/raise.c:54
54    ../sysdeps/unix/sysv/linux/raise.c: No such file or directory.
(gdb)

Program terminated with signal SIGABRT, Aborted.
The program no longer exists.
```

Threads

After the new thread has been launched,

```
thread t1(threadFunction, "Hello");
```

the initial thread (`main`) continues execution.

It does not wait for the new thread to finish and ends the program—possibly before the new thread has had a chance to run.

We need to add a call to `thread` member function `join` which will cause the calling thread (`main`) to wait for the thread associated with the `thread` object `t1`

```

#include <iostream>
#include <thread>

using namespace std;

void threadFunction(string msg)
{
    cout << "threadFunction says "
        << msg << endl;
}

int main(void)
{
    //Construct a new thread and run it
    thread t1(threadFunction, "Hello");
t1.join();
    return 0;
}

```

```

student@cse1325:/media/sf_VM$ ./a.out
threadFunction says Hello
student@cse1325:/media/sf_VM$ █

```

```

Breakpoint 1, main () at threadDemo.cpp:13
13      {
(gdb) n
15          thread t1(threadFunction, "Hello");
(gdb) n
[New Thread 0x7ffff6f4f700 (LWP 13520)]
threadFunction says Hello
[Thread 0x7ffff6f4f700 (LWP 13520) exited]
18          t1.join();
(gdb) n
20          return 0;
(gdb) n
15          thread t1(threadFunction, "Hello");
(gdb) n
21      }
(gdb) n
__libc_start_main (main=0x4011d5 <main()>, argc=1, argv=0x7fffffff1f8,
                  init=<optimized out>, fini=<optimized out>, rtld_fini=<optimized out>
                  stack_end=0x7fffffff1e8) at ../csu/libc-start.c:325
325      ../csu/libc-start.c: No such file or directory.
(gdb) n
[Inferior 1 (process 13516) exited normally]

```

Threads

An initialized thread object represents an active thread of execution and is joinable and has a unique thread id which we can obtain by calling thread member function `get_id()`.

```
thread t1(threadT1, "Hello");
thread t2(threadT2, "Hello");
thread t3(threadT3, "Hello");
thread t4(threadT4, "Hello");
thread t5(threadT5, "Hello");

cout << "t1's id is "
     << t1.get_id() << endl;
cout << "t2's id is "
     << t2.get_id() << endl;
cout << "main's id is "
     << this_thread::get_id()
     << endl;
```

t1's id is	140390222829312
t2's id is	140390214436608
main's id is	140390240180032
threadT5 says Hello	T5 i = 1
threadT4 says Hello	T4 i = 2
threadT3 says Hello	T3 i = 3
threadT2 says Hello	T2 i = 4
threadT1 says Hello	T1 i = 5

Threads
We can
also ask the
thread
pointed at
by this
to give us
its id.

```
void threadFunction(int x)
{
    cout << "My id = " << this_thread::get_id() << endl;
}

int main(void)
{
    int x = 0;
    thread::id main_tid = this_thread::get_id();
    thread t1(threadFunction, x);
    cout << "t1's id = " << t1.get_id() << endl;
    cout << "main's id = " << main_tid << endl;
    t1.join();

    return 0;
}
```

```
t1's id = 139717741209344
main's id = 139717759383360
My id = 139717741209344
```

Threads

A default-constructed (non-initialized) thread object is not joinable.

```
thread t6();
```

```
cout << "t6's id is "
    << t6.get_id()
    << endl;
```

```
t6.join();
```

```
student@cse1325:/media/sf_VM$ g++ threadDemo.cpp -g -std=c++11 -pthread
threadDemo.cpp: In function 'int main()':
threadDemo.cpp:53:30: error: request for member 'get_id' in 't6', which is of no
n-class type 'std::thread()'
    cout << "t6's id is " << t6.get_id() << endl;
                                         ^
threadDemo.cpp:62:5: error: request for member 'join' in 't6', which is of non-c
lass type 'std::thread()'
    t6.join();
                                         ^
```

Threads

The act of calling `join()` cleans up any storage associated with the thread.

The `thread` object is no longer associated with the now-finished thread - it isn't associated with any thread.

This means that you can call `join()` only once for a given thread.

Once you've called `join()`, the `thread` object is no longer joinable.

```
int main(void)
{
    //Construct a new thread and run it
    thread t1(threadFunction, "Hello");
    t1.join();
    t1.join();
    return 0;
}
```

terminate called after throwing an instance of
'std::system_error'
what(): Invalid argument
Aborted (core dumped)

Threads

The arguments passed to the thread's function are passed by copy by default.

```
void threadFunction(int x)
{
    x++;
    cout << "x = " << x << endl;
}

int main(void)
{
    int x = 0;

    cout << "x before = " << x << endl;
    thread t1(threadFunction, x);
    cout << "x after = " << x << endl;

    t1.join();

    return 0;
}
```

```
student@csel325:/media/sf_VMS$ ./thread1Demo.e
x before = 0
x after = 0
x = 1
```

Threads

By default, the arguments are *copied* into internal storage where they can be accessed by the newly created thread of execution, even if the corresponding parameter in the function is expecting a reference.

```
void threadFunction(int &x)
{
    x++;
    cout << "x = " << x << endl;
}

int main(void)
{
    int x = 0;

    cout << "x before = " << x << endl;
    thread t1(threadFunction, x);
    cout << "x after = " << x << endl;

    t1.join();

    return 0;
}
```

```
student@cse1325:/media/sf_VMS$ make
g++ -c -g -std=c++11 -pthread thread1Demo.cpp -o thread1Demo.o
In file included from thread1Demo.cpp:3:0:
/usr/include/c++/7/thread: In instantiation of 'struct std::thread::_Invoker<std::tuple<void (*)(int&), int> >':
/usr/include/c++/7/thread:127:22:   required from 'std::thread::thread(_Callable&&, _Args&& ...) [with _Callable = void (&)(int&); _Args = {int&}]'
thread1Demo.cpp:21:29:   required from here
/usr/include/c++/7/thread:240:2: error: no matching function for call to 'std::thread::_Invoker<std::tuple<void (*)(int&), int> >::__M_invoke(std::tuple<void (*)(int&), int> >::__Indices)'
          operator()()
          ~~~~~
/usr/include/c++/7/thread:231:4: note: candidate: template<long unsigned int ..._Ind> decltype (std::__invoke(_S_declval<_Ind>)()...)
std::thread::_Invoker<_Tuple>::__M_invoke(std::__Index_tuple<_Ind ...>) [with long unsigned int ..._Ind = {_Ind ...}; _Tuple = std::tuple<void (*)(int&), int>]
          __M_invoke(__Index_tuple<_Ind...>)
          ~~~~~
/usr/include/c++/7/thread:231:4: note:   template argument deduction/substitution failed:
/usr/include/c++/7/thread: In substitution of 'template<long unsigned int ..._Ind> decltype (std::__invoke(_S_declval<_Ind>)()...) std::thread::_Invoker<std::tuple<void (*)(int&), int> >::__M_invoke<_Ind ...>(std::__Index_tuple<_Indl ...>) [with long unsigned int ..._Ind = {0, 1}]':
/usr/include/c++/7/thread:240:2:   required from 'struct std::thread::_Invoker<std::tuple<void (*)(int&), int> >'
/usr/include/c++/7/thread:127:22:   required from 'std::thread::thread(_Callable&&, _Args&& ...) [with _Callable = void (&)(int&); _Args = {int&}]'
thread1Demo.cpp:21:29:   required from here
/usr/include/c++/7/thread:233:29: error: no matching function for call to '__invoke(std::__tuple_element_t<0, std::tuple<void (*)(int&), int> >, std::__tuple_element_t<1, std::tuple<void (*)(int&), int> >)'
          -> decltype(std::__invoke(_S_declval<_Ind>)())
          ~~~~~
In file included from /usr/include/c++/7/tuple:41:0,
                 from /usr/include/c++/7/bits/unique_ptr.h:37,
                 from /usr/include/c++/7/memory:80,
                 from /usr/include/c++/7/thread:39,
                 from thread1Demo.cpp:3:
/usr/include/c++/7/bits/Invoke.h:89:5: note: candidate: template<class _Callable, class ... _Args> constexpr typename std::__invoke_result<_Functor, _ArgTypes>::type std::__invoke(_Callable&&, _Args&& ...)
          __invoke(_Callable&& __fn, _Args&&... __args)
          ~~~~~
/usr/include/c++/7/bits/Invoke.h:89:5: note:   template argument deduction/substitution failed:
/usr/include/c++/7/bits/Invoke.h: In substitution of 'template<class _Callable, class ... _Args> constexpr typename std::__invoke_result<_Functor, _ArgTypes>::type std::__invoke(_Callable&&, _Args&& ...) [with _Callable = void (*)(int&); _Args = {int}]':
/usr/include/c++/7/thread:233:29:   required by substitution of 'template<long unsigned int ..._Ind> decltype (std::__invoke(_S_declval<_Ind>)()...')
std::thread::_Invoker<std::tuple<void (*)(int&), int> >::__M_invoke<_Ind ...>(std::__Index_tuple<_Indl ...>) [with long unsigned int ..._Ind = {0, 1}]'
/usr/include/c++/7/thread:240:2:   required from 'struct std::thread::_Invoker<std::tuple<void (*)(int&), int> >'
/usr/include/c++/7/thread:127:22:   required from 'std::thread::thread(_Callable&&, _Args&& ...) [with _Callable = void (&)(int&); _Args = {int&}]'
thread1Demo.cpp:21:29:   required from here
/usr/include/c++/7/bits/Invoke.h:89:5: error: no type named 'type' in 'struct std::__invoke_result<void (*)(int&), int>'
makefile:14: recipe for target 'thread1Demo.o' failed
make: *** [thread1Demo.o] Error 1
```

Threads

To "pause" a thread's execution, it can be made to sleep by calling `sleep_for` with a parameter of a typedef from the `chrono` time library.

```
this_thread::sleep_for(chrono::seconds(10));
```

Use `#include <chrono>` in order to have access to `chrono`'s typedefs `hours`, `minutes` and `seconds`.

```
int main(void)
{
    thread t1(threadT1, "I'm thread T1");
    thread t2(threadT2, "I'm thread T2");
    thread t3(threadT3, 5);

    cout << "main() is napping for 5 seconds" << endl;
    this_thread::sleep_for(chrono::minutes(5));
    cout << "main() says I'M AWAKE " << endl;

    // Makes the main thread wait for the new thread to finish execution
    t1.join();
    t2.join();
    t3.join();

    return 0;
}
```

```
void threadT3(int seconds)
{
    cout << "threadT3 is napping for " << seconds
        << "seconds" << endl;
    this_thread::sleep_for(chrono::seconds(seconds));
    cout << "threadT3 says I'M AWAKE " << endl;
}
```

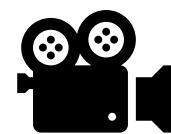


student@cse1325: /media/sf_VM

- + ×

File Edit Tabs Help

student@cse1325:/media/sf_VM\$./threadsleepDemo.eI



student@cse1325:/me...

10:26

Threads

Reentrant Function/Algorithm

A reentrant function/algorithm behaves correctly if called simultaneously by several threads.

Functions that are callable by several threads must be made reentrant. To make a function reentrant might require changes to the function interface or to the implementation.

Functions that access global state, like memory or files, have reentrance problems.

In C++, functions can be made reentrant by using mutex.

Threads

mutex

A mutex is a [*lockable object*](#) that is designed to signal when critical sections of code need exclusive access and prevents other threads with the same protection from executing concurrently and access the same memory locations.

Locking a mutex prevents other threads from locking it (exclusive access) until it is unlocked.

Must

```
#include <mutex>
```



```
int main(void)
{
    srand(time(NULL));

    thread asterisk_thread(print_it, 20, '*');
    thread dollar_thread(print_it, 20, '$');

    asterisk_thread.join();
    dollar_thread.join();

    return 0;
}

void print_it(int NumberOfCharsToPrint, char charToPrint)
{
    for (int i = 0; i < NumberOfCharsToPrint; ++i)
    {
        cerr << charToPrint;

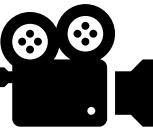
        this_thread::sleep_for(chrono::milliseconds(200+rand()%200));
    }

    cout << '\n';
}
```

student@cse1325: /media/sf_VM

File Edit Tabs Help

student@cse1325:/media/sf_VM\$./mutexDemo.e



student@cse1325:/me...

11:00

```
int main(void)
{
    srand(time(NULL));

    thread asterisk_thread(print_it, 20, '*');
    thread dollar_thread(print_it, 20, '$');

    asterisk_thread.join();
    dollar_thread.join();

    return 0;
}

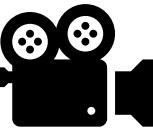
void print_it(int NumberOfCharsToPrint, char charToPrint)
{
    for (int i = 0; i < NumberOfCharsToPrint; ++i)
    {
        cerr << charToPrint; 
        this_thread::sleep_for(chrono::milliseconds(200+rand()%200));
    }

    cout << '\n';
}
```

Why `cerr` and not `cout`?

File Edit View Terminal Tabs Help

student@maverick:/media/sf_VM/CSE1325\$./mutex1Demo.



Threads

Threads don't do a very good job of sharing a resource.

`asterisk_thread` and `dollar_thread` were sharing the standard output stream and were not able to evenly or consistently take turns.

What if those two threads had been sharing a file and were tasked with updating that file?

The data in the file would be in a different order every time the process ran.

mutex

We can use a synchronization primitive called a mutex (*mutual exclusion*) to help alleviate this sharing issue.

Before accessing a shared resource, you lock the mutex associated with that resource and, when you have finished accessing the data structure, you unlock the mutex.

The Thread Library ensures that once one thread has locked a specific mutex, all other threads that try to lock the same mutex have to wait until the thread that successfully locked the mutex unlocks it.

mutex

Create a mutex by constructing an instance of std::mutex

```
mutex mtx;
```

If not using namespace std, then this would need to be std::mutex

lock it with a call to the member function lock()

```
mtx.lock();
```

and unlock it with a call to the member function unlock()

```
mtx.unlock();
```

```
mutex mtx; // construct mutex object

void print_it(int NumberOfCharsToPrint, char charToPrint)
{
    mtx.lock();

    for (int i = 0; i < NumberOfCharsToPrint; ++i)
    {
        cerr << charToPrint;
        this_thread::sleep_for(chrono::milliseconds(200+rand()%200));
    }

    cout << '\n';

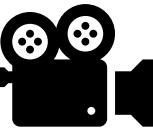
    mtx.unlock();
}
```

student@cse1325: /media/sf_VM

File Edit Tabs Help

student@cse1325:/media/sf_VM\$./mutexDemo.e

\$



```
int main(void)
{
    vector<thread> Alphabet;

    for(int i = 0; i < 26; ++i)
    {
        char Letter = i+65;
        Alphabet.push_back(std::thread(print_it, Letter));
    }

    for (thread& it : Alphabet)
    {
        it.join();
    }

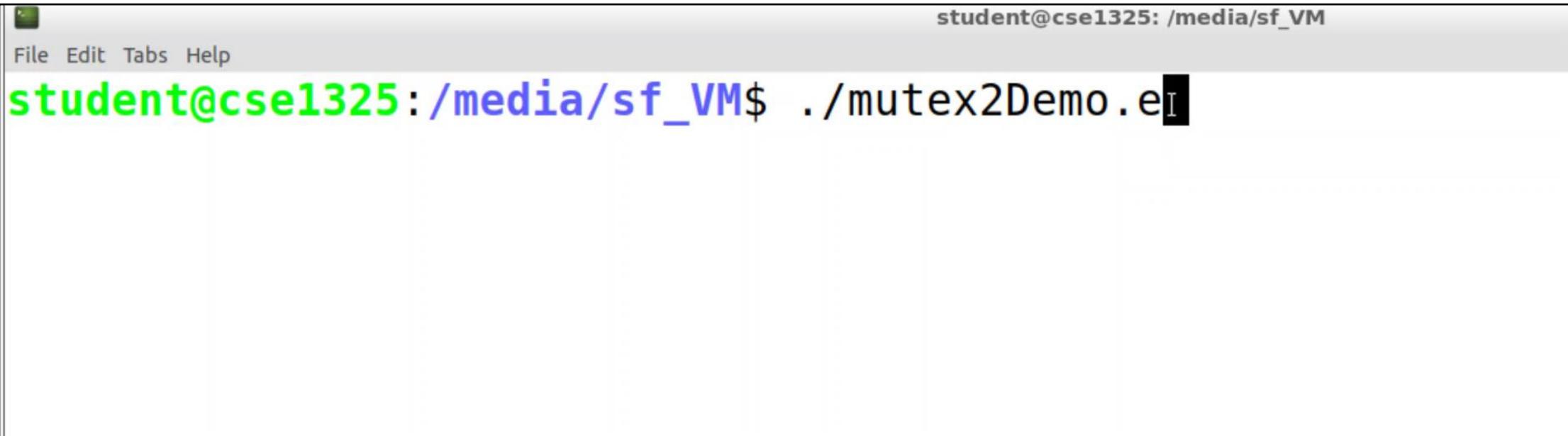
    cout << endl;

    return 0;
}
```

```
void print_it(char charToPrint)
{
    cerr << charToPrint;
    cerr << charToPrint;
}
```

```
student@cse1325:/media/sf_VM$ ./mutex2Demo.e
GGHHIIFFJJKKLLMMEENNOOPPQQRRSSDDTTUVVWWXXYYZZCCBAA
student@cse1325:/media/sf_VM$
```

```
void print_it(char charToPrint)
{
    this_thread::sleep_for(chrono::milliseconds(200+rand()%200));
    cerr << charToPrint;
    this_thread::sleep_for(chrono::milliseconds(200+rand()%200));
    cerr << charToPrint;
}
```



The screenshot shows a terminal window with the following details:

- Title Bar:** student@cse1325: /media/sf_VM
- Menu Bar:** File Edit Tabs Help
- Command Line:** student@cse1325:/media/sf_VM\$./mutex2Demo.e[
The command entered is ./mutex2Demo.e, where the last few characters are obscured by a black box.

```
void print_it(char charToPrint)
{
    mtx.lock();

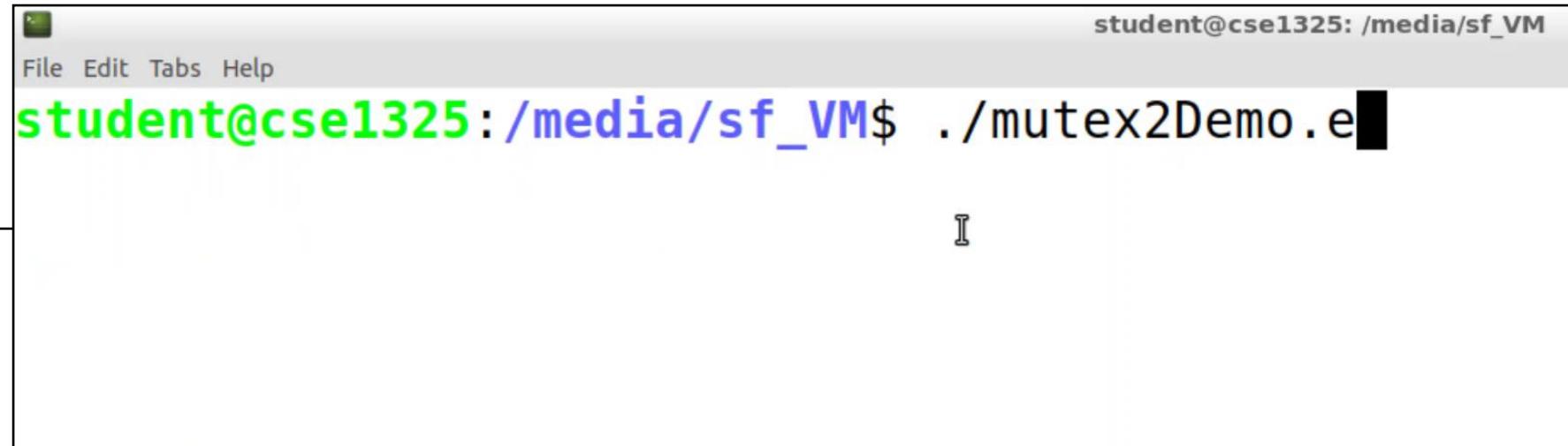
    this_thread::sleep_for(chrono::milliseconds(200+rand()%200));

    cerr << charToPrint;

    this_thread::sleep_for(chrono::milliseconds(200+rand()%200));

    cerr << charToPrint;

    mtx.unlock();
}
```



A screenshot of a terminal window titled "student@cse1325: /media/sf_VM". The window shows the command "student@cse1325:/media/sf_VM\$./mutex2Demo.e" followed by a black rectangular redaction box. The terminal interface includes a menu bar with "File", "Edit", "Tabs", and "Help".

```
int main(void)
{
    vector<thread> Numbers;
    srand(time(NULL));

    for(int i = 10; i < 100; ++i)
    {
        Numbers.push_back(std::thread(print_it, i));
    }

    for(thread& it : Numbers)
    {
        it.join();
    }

    return 0;
}
```

```
void print_it(int NumberToPrint)
{
    static int counter = 1;

    cerr << NumberToPrint << '-';

    this_thread::sleep_for(chrono::milliseconds(200+rand()%200));

    if (!(counter++ %10))
        cout << endl;
}
```

```
student@cse1325:/media/sf_VM$ ./mutex3Demo.e
16-17-15-18-19-14-20-21-22-23-13-24-26-27-28-25-29-30-31-12-32-33-34-35-36-40-37-39-4
2-41-38-59-48-56-43-52-58-55-46-49-60-11-53-54-45-57-62-44-47-50-61-51-63-64-65-66-67
-68-69-70-71-72-73-74-75-76-77-78-79-80-81-82-83-84-85-86-10-87-88-89-90-91-92-93-94-
95-96-97-98-99-
```

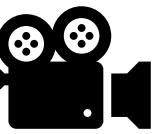
```
student@cse1325:/media/sf_VM$ █
```



student@cse1325: /media/sf_VM

- + x

File Edit Tabs Help



student@cse1325:/media/sf_VM\$./mutex3Demo.e

```
mutex Kitty;

void print_it(int NumberToPrint)
{
    static int counter = 1;

    Kitty.lock();

    cerr << NumberToPrint << '-';

    this_thread::sleep_for(chrono::milliseconds(200+rand()%200));

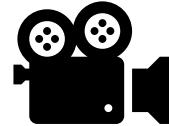
    if (!(counter++ %10))
        cout << endl;

    Kitty.unlock();
}
```



student@cse1325: /

File Edit Tabs Help



student@cse1325:/media/sf_VM\$./mutex3Demo.e

student@cse1325: /media/sf_StudentCode/George

File Edit Tabs Help

```
student@cse1325:/media/sf_StudentCode/George$ ./horserace
```

student@cse1325: /media/sf_StudentCode/George

File Edit Tabs Help

```
student@cse1325:/media/sf_StudentCode/George$ ./horserace
```

Multithreading Vocabulary

Stack – Scratch memory for a thread

Heap – Memory shared by all threads for dynamic allocation

Concurrency – Performing 2 or more algorithms simultaneously

Process – A self-contained execution environment including its own memory space

Thread – An independent path of execution within a process, running concurrently with other threads within a shared memory space

Reentrant – An algorithm that can be paused while executing and then safely executed by a different thread

mutex – A mutual exclusion object that prevents two properly written threads from concurrently accessing a critical resource

Standard Template Library (STL) C++ Standard Library

The Standard Library defines powerful, template-based, reuseable components that implement many common data structures and algorithms used to process those data structures.

Three key components of the Standard Library

Containers (templatized data structures)

Iterators

Algorithms

Standard Template Library (STL)

C++ Standard Library

Containers are data structures capable of storing objects of almost any data type.

Three styles of container classes

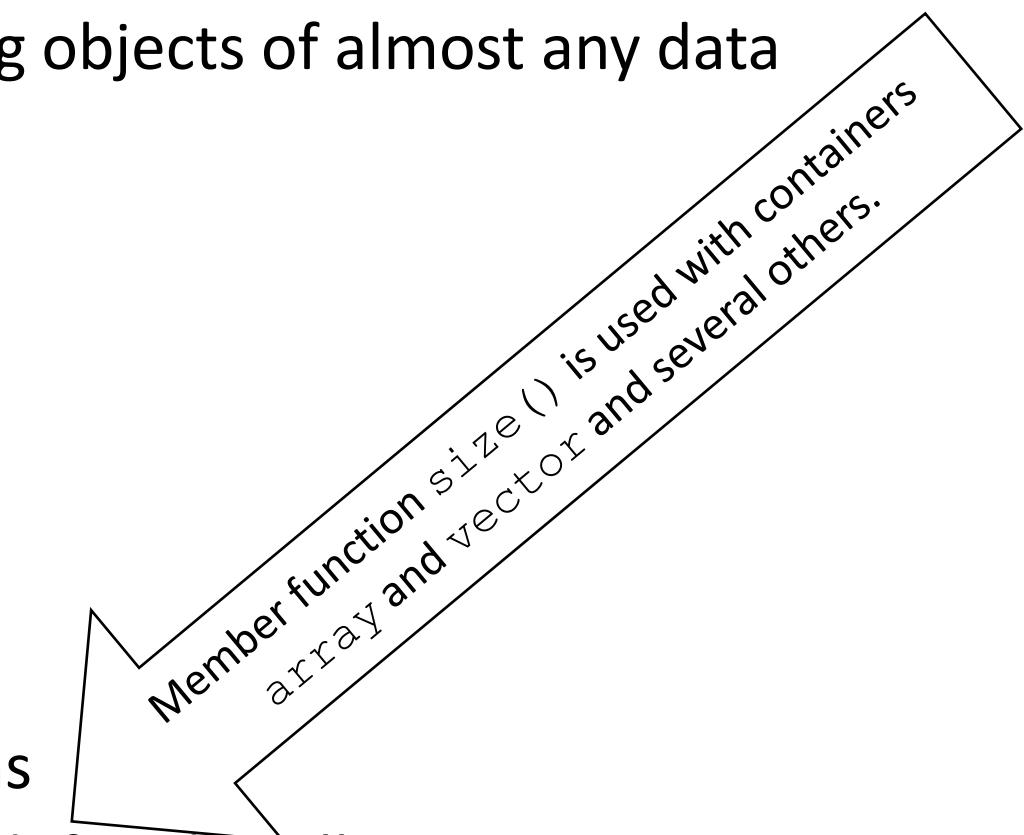
- first-class containers

- container adapters

- near containers

Each container has associated member functions

- a subset of those member functions are defined in all containers



Member function `size()` is used with containers
array and `vector` and several others.

Container Classes

first-class containers

- sequence containers

- associative containers

container adapters

- constrained version of sequence containers

near containers

- containers that exhibit some but not all capabilities of the first-class containers

- used for performing high-speed mathematical vector operations

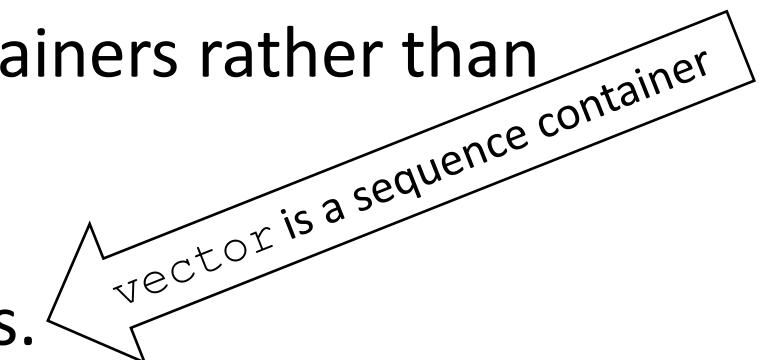
Container Classes

When an object is inserted into a container, a copy of the object is made.

The object should provide a copy constructor and copy assignment operator (custom or default).

It is usually preferable to reuse Standard Library containers rather than developing custom templatized data structures.

`vector` is typically satisfactory for most applications.



Associative Containers

The associative containers provide direct access to store and retrieve elements via keys (often called search keys).

The four ordered associative containers are

multiset,

set,

multimap

map

Each of these maintains its keys in sorted order.

Associative Containers

Class `map` provides operations for manipulating values associated with keys (these values are sometimes referred to as mapped values).

The primary difference between a `multimap` and a `map` is that a `multimap` allows duplicate keys with associated values to be stored and a `map` allows only *unique keys* with associated values.

In addition to the common container member functions, *ordered associative containers* also support several other member functions that are specific to associative containers.

map

The map *associative container* (from header `<map>`) performs fast storage and retrieval of *unique keys* and *associated values*.

The elements' ordering is determined by a comparator function object.

For example, in an integer map, elements can be sorted in ascending order by ordering the keys with comparator function object `less<int>`.

No two elements in the container can have equivalent *keys*.

map

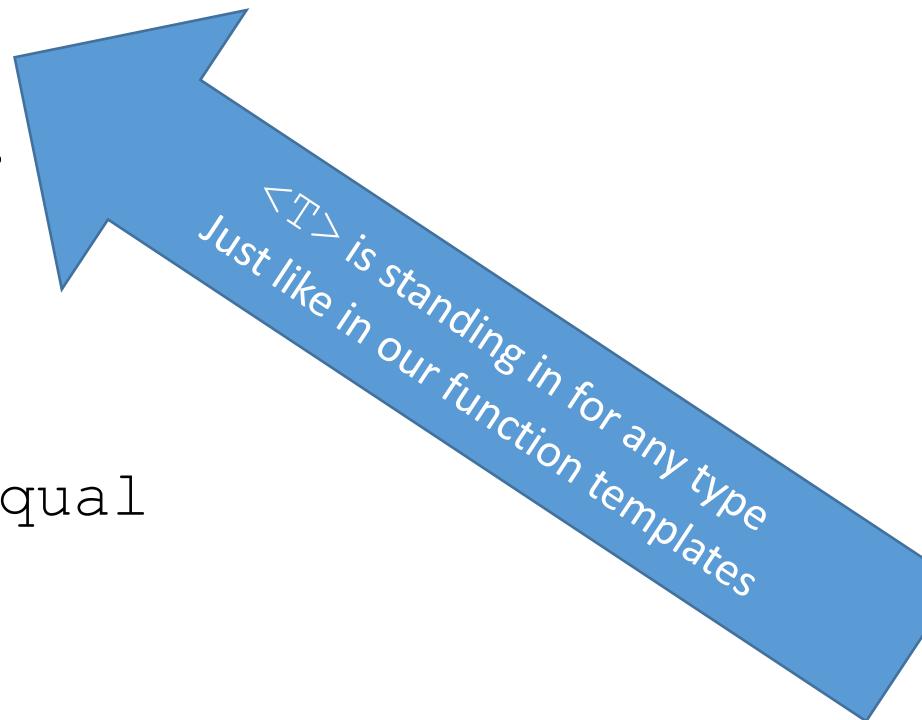
The data type of the keys in all ordered associative containers must support comparison based on the comparator function object—keys sorted with `less<T>` must support comparison with operator `<`.

The default comparator function object is `less<T>` which sorts the key using `<`.

The comparator function object `greater<T>` sorts using `>`.

Other builtin comparator function objects

<code>equal_to</code>	<code>greater</code>	<code>less</code>
<code>not_equal_to</code>	<code>greater_equal</code>	<code>less_equal</code>



`<T>` is standing in for any type
Just like in our function templates

It is possible to create your own comparator function object.

map

A **map** is a set where each element is a pair, called a key/value pair.

The key is used for sorting and indexing the data and must be unique.

The value is the actual data.

Duplicate keys are *not* allowed—a single value can be associated with each key.

This is called a one-to-one mapping.

map

A map of students where **id number** is the key and **name** is the value can be represented graphically as

Notice that keys are arranged in ascending order.

Maps always arrange its keys in sorted order.

Here the keys are of string type; therefore, they are sorted lexicographically.

1120217	Nikhilesh
1120236	Navneet
1120250	Vikas
1120255	Doodrah

A diagram illustrating the mapping between the table columns and the terms 'Keys' and 'values'. Two blue rounded rectangular boxes at the bottom right are labeled 'Keys' and 'values'. Two curved arrows originate from these boxes: one arrow points from 'Keys' to the first column of the table, and another arrow points from 'values' to the second column.

map

For example, a company that uses unique employee numbers, such as 100, 200 and 300, might have a map that associates employee numbers with their telephone extensions—4321, 4115 and 5217, respectively.

With a map you specify the key and get back the associated data quickly.

Providing the key in a map's subscript operator [] locates the value associated with that key in the map.

map

To create a map

```
map<int, double, less<int>> MyMap;
```



MyMap is the name of our map object.

Key type is int

Type of key's associated value is double.

map's elements will be sorted in ascending order using the function object less<int>. Ascending order is the default for a map so less<int> can be omitted.

map

To insert a new key-value pair

```
MyMap.insert(make_pair(15, 2.7));
```

MyMap is the name of the map and insert () is a member function of map

insert () adds a new key-value pair to the map

make_pair () is a member function of map that uses the types specified for the keys and values to create a key-value pair object.

```
map<int, double> MyMap;  
  
MyMap.insert(make_pair(15, 2.7));  
MyMap.insert(make_pair(30, 111.11));  
MyMap.insert(make_pair(5, 1010.1));  
MyMap.insert(make_pair(10, 22.22));  
MyMap.insert(make_pair(25, 33.333));  
MyMap.insert(make_pair(5, 77.54)); ← dup ignored – no update  
MyMap.insert(make_pair(20, 9.345)); ← dup ignored – no update  
MyMap.insert(make_pair(15, 99.3)); ← dup ignored – no update
```

```
(gdb) p MyMap  
$5 = std::map with 6 elements = {  
    [5] = 1010.1,  
    [10] = 22.21999999999999,  
    [15] = 2.700000000000002,  
    [20] = 9.345000000000006,  
    [25] = 33.33299999999998,  
    [30] = 111.11  
}
```

```
map<int, double> MyMap;  
  
MyMap.insert(make_pair(15, 2.7));  
MyMap.insert(make_pair(30, 111.11));  
MyMap.insert(make_pair(5, 1010.1));  
MyMap.insert(make_pair(10, 22.22));  
MyMap.insert(make_pair(25, 33.333));  
MyMap.insert(make_pair(5, 77.54)); // dup ignored  
MyMap.insert(make_pair(20, 9.345));  
MyMap.insert(make_pair(15, 99.3)); // dup ignored
```

```
cout << "MyMap contains:\nKey\tValue\n";
```

```
// walk through elements of MyMap  
for (auto mapItem : MyMap)
```

range-based for statement

for each iteration, assign the next element of MyMap to key-value pair object mapItem, then execute the loop's body
auto uses int because the map's keys are of type int

```
(gdb) p MyMap
$5 = std::map with 6 elements = {
[5] = 1010.1,
[10] = 22.219999999999999,
[15] = 2.7000000000000002,
[20] = 9.3450000000000006,
[25] = 33.332999999999998,
[30] = 111.11
}
cout << "MyMap contains:\nKey\tValue\n";
for (auto mapItem : MyMap)
{
    cout << mapItem.first << '\t' << mapItem.second << '\n';
}
```

mapItem is an iterator object of type key-value MyMap

```
(gdb) p mapItem
$4 = {
    first = 5,
    second = 1010.1
}
```

MyMap contains:
Key Value
5 1010.1

```
// walk through elements of MyMap
for (auto mapItem : MyMap)
{
    cout << mapItem.first << '\t'
        << mapItem.second << '\n';
}

MyMap[25] = 9999.99; // use subscripting to change value for key 25
MyMap[40] = 8765.43; // use subscripting to insert value for key 40

cout << "\nAfter subscript operations, \nMyMap contains:\nKey\tValue\n";

for (auto mapItem : MyMap)
{
    cout << mapItem.first << '\t'
        << mapItem.second << '\n';
}
```

```
(gdb) p MyMap
$1 = std::map with 6 elements = {[5] = 10.101010101010101
22.21999999999999,
[15] = 2.7000000000000002, [20] = 9.345000000000001
[25] = 33.33299999999998, [30] = 111.1111111111111
30 MyMap[25] = 9999.99; // use [] to change
31 MyMap[40] = 8765.43; // use [] to insert
```

MyMap contains:

Key	Value
5	1010.1
10	22.22
15	2.7
20	9.345
25	33.333
30	111.11

```
(gdb) p MyMap
$2 = std::map with 7 elements = {[5] = 1010.1, [10] = 22.22, [15] = 2.7, [20] = 9.345000000000001, [25] = 9999.989999999998, [30] = 111.11, [40] = 8765.43}
```

After [] operations,
MyMap contains:

Key	Value
5	1010.1
10	22.22
15	2.7
20	9.345
25	9999.99
30	111.11
40	8765.43

```
#include <iostream>
#include <map>
#include <sstream>

using namespace std;

string MapToString(map<string, double> & maptoString)
{
    ostringstream MapString;

    for (auto mapItem : maptoString)
    {
        MapString << mapItem.first << "\t" << mapItem.second << endl;
    }

    return MapString.str();
}

int main (void)
{
    map<string,double> GradeMap{ {"A",78.52} , {"B",85.94} , {"C",90.57} };

    string MapString = MapToString(GradeMap);

    cout << MapString; // Print the result
}
```

```
int main (void)
{
    map<string,double> GradeMap{ {"A",78.52} , {"B",85.94} , {"C",90.57} } ;

    string MapString = MapToString(GradeMap) ;

    cout << MapString; // Print the result
}
```

```
(gdb) p GradeMap
$2 = std::map with 3 elements =
  ["A"] = 78.51999999999996,
  ["B"] = 85.93999999999998,
  ["C"] = 90.56999999999993
}
```

```
(gdb) p MapString
$3 = "A\t78.52\nB\t85.94\nC\t90.57\n"
```

A 78.52
B 85.94
C 90.57

map<string, double>

Key is a string – letter grade
Value is a double – numeric grade

```
string MapToString(map<string, double> & maptoString) {  
    ostringstream MapString; #include <sstream>  
    for (auto mapItem : maptoString) range based for  
    {  
        MapString << mapItem.first << "\t" << mapItem.second << endl;  
    }  
    return MapString.str();  
}
```

```
A 78.52  
B 85.94  
C 90.57
```

(gdb) p mapItem
\$6 = {
 first = "A",
 second = 78.51999999999996}

(gdb) p mapItem
\$8 = {
 first = "B",
 second = 85.93999999999998}

(gdb) p mapItem
\$11 = {
 first = "C",
 second = 90.56999999999993}

```
25 string MapString = MapToString(GradeMap);
```

```
(gdb) step
```

```
MapToString (maptoString=std::map with 3  
elements = {...}) at map2Demo.cpp:10
```

map2Demo.cpp

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

using namespace std;

int main(void)
{
    map<string, int> mapOfWords;
    string SearchWord;
    int SearchValue;
```

mapOfWords is a map with string keys and int values.

SearchWord will be used to search for keys in our map

SearchValue will be used to search for values in our map

map function empty ()

```
if (mapOfWords.empty())
    cout << "Our map is empty" << endl;
```

```
if (!mapOfWords.empty())
    cout << "Our map is NOT empty" << endl;
```

returns true if map size is 0, else returns false

Never throws an exception

```
if (mapOfWords.empty())
    cout << "Our map is empty" << endl;

cout << "Inserting 'earth'" << endl;
mapOfWords.insert(make_pair("earth", 2));

cout << "Inserting 'moon'" << endl;
mapOfWords.insert(make_pair("moon", 4));

if (!mapOfWords.empty())
    cout << "Our map is NOT empty" << endl;
```

```
cout << "Inserting 'sun'" << endl;
mapOfWords["sun"] = 3;
```

Our map is empty

Inserting 'earth'

Inserting 'moon'

Our map is NOT empty

Inserting 'sun'

(gdb) p mapOfWords
\$2 = std::map with 3
elements = {
 ["earth"] = 2,
 ["moon"] = 4,
 ["sun"] = 3
}

map function size()

mapOfWords.size()

does not take any parameters

returns the number of elements in the map

Never throws an exception

```
cout << "Key\tValue" << endl;
for (auto mapItem : mapOfWords)
{
    cout << mapItem.first << "\t" << mapItem.second << endl;
}

cout << "Our map has " << mapOfWords.size() << " elements" << endl;
```

Key	Value
earth	2
moon	4
sun	3

Our map has 3 elements

```
(gdb) p mapOfWords
$3 = std::map with 3 elements = {
    ["earth"] = 2,
    ["moon"] = 4,
    ["sun"] = 3
}
```

```
Breakpoint 3, main () at map3Demo.cpp:38
38         mapOfWords["earth"] = 1;
```

```
(gdb) p mapOfWords
$3 = std::map with 3 elements = {
    ["earth"] = 1,
    ["moon"] = 4,
    ["sun"] = 3
}
```

map3Demo.cpp

operator [] vs insert function

If specified key already existed in map then operator [] will silently change its value; whereas, insert will not replace an already added key. Instead, it returns if element was added or not.

```
// Will replace the value of already added key i.e. earth  
mapOfWords["earth"] = 1;
```

Whereas, for insert member function,

```
// fails and returns false in object's second member  
if (mapOfWords.insert(make_pair("earth", 1)).second == false)  
{  
    cout << "Element with key 'earth' not inserted because  
          already existed" << endl;  
}
```

```
(gdb) p mapOfWords
$2 = std::map with 3 elements = { ["earth"] = 2, ["moon"] = 4, ["sun"] = 3 }

38     mapOfWords["earth"] = 1;

(gdb) p mapOfWords
$3 = std::map with 3 elements = { ["earth"] = 1, ["moon"] = 4, ["sun"] = 3 }

41     if (mapOfWords.insert(make_pair("earth", 1)).second == false)
43             cout << "Element with key 'earth' not inserted because already
existed" << endl;
```

Element with key 'earth' not inserted because already existed

```
(gdb) p mapOfWords
$4 = std::map with 3 elements = { ["earth"] = 1, ["moon"] = 4, ["sun"] = 3 }
(gdb)
```

map function at ()

mapOfWords.at(key)

takes in *key*

returns a reference to the key's value

If an exception is thrown, there are no changes in the container.

It throws [out of range](#) if *key* is not the key of an element in the [map](#).

```
// at() will throw an exception when given key out of range
try
{
    cout << "The value for 'earth' is " << mapOfWords.at("earth") << endl;
    cout << "The value for 'venus' is " << mapOfWords.at("venus") << endl;
}
catch (out_of_range& say)
{
    cerr << "An out of range exception was thrown by "
        << say.what() << endl;
}
```

The value for 'earth' is 1

An out of range exception was thrown by map::at

operator [] vs at function

operator [] does not do range checking. If you access a key using the indexing operator [] that is not currently a part of a map, then it automatically adds a key for you

at () member function does range checking and throws an exception when you are trying to access a nonexisting element.

```
// [] operator does not check bounds and does not throw exception
cout << "The value for 'venus' is " << mapOfWords["venus"] << endl;
```

The value for 'venus' is 0

```
(gdb) p mapOfWords
$1 = std::map with 4 elements = { ["earth"] = 1, ["moon"] = 4, ["sun"] = 3,
["venus"] = 0}
```

map function find()

mapOfWords.find(key)

takes in key

returns an iterator to the element if key is found, else returns end

If an exception is thrown, there are no changes in the container.

```
for (auto mapItem : mapOfWords)
{
    cout << mapItem.first << "\t" << mapItem.second << endl;
}

cout << "Enter a word to find " << endl;
cin >> SearchWord;

// Searching element in map by key.
if (mapOfWords.find(SearchWord) != mapOfWords.end())
    cout << "word '" << SearchWord << "' found" << endl;
else
    cout << "word '" << SearchWord << "' not found" << endl;

if (mapOfWords.find("mars") == mapOfWords.end())
    cout << "word 'mars' not found" << endl;
```

earth	1
moon	4
sun	3
venus	0

```
if (mapOfWords.find(SearchWord) != mapOfWords.end())
    cout << "word '" << SearchWord << "' found" << endl;
else
    cout << "word '" << SearchWord << "' not found" <<
endl;

if (mapOfWords.find("mars") == mapOfWords.end())
    cout << "word 'mars' not found" << endl;
```

Enter a word to find
moon

word 'moon' found

word 'mars' not found

map function count ()

mapOfWords.count(key)

takes in key

returns 1 if key is found, else returns 0

If an exception is thrown, there are no changes in the container.

```
cout << "Key 'mars' ";
if (mapOfWords.count("mars"))
    cout << " is part of our map" << endl;
else
    cout << " is not part of our map" << endl;
```

earth	1
moon	4
sun	3
venus	0

```
cout << "Enter a word to search for in our map ";
cin >> SearchWord;
```

```
cout << "Key '" << SearchWord << "'";
```

```
if (mapOfWords.count(SearchWord))
    cout << " is part of our map" << endl;
else
    cout << " is not part of our map" << endl;
```

```
Key 'mars' is not part of our map
Enter a word to search for in our map earth
Key 'earth' is part of our map
```

count () vs find () function

Since a map can only have at most one key, count () will essentially stop after one element has been found. However, in view of more general containers such as multimap and multisets, find is strictly better if you only care whether some element with this key exists, since it can really stop once the first matching element has been found.

In general, both count () and find () will use the container-specific lookup methods (tree traversal or hash table lookup), which are always fairly efficient. It's just that count () has to continue iterating until the end of the equal-range, whereas find () does not.

If you just want to find whether the key exists or not, and don't care about the value, it is better to use count () as it returns only an integer. find () returns an iterator, thus by using count (), you will save the construction of an iterator.

```
map<char, string> MyPets;  
  
MyPets.insert(make_pair('A', "Appa"));  
MyPets.insert(make_pair('S', "Sylvester"));  
MyPets.insert(make_pair('S', "Shade"));   
MyPets.insert(make_pair('J', "Josie")); 
```

```
for (auto Pet : MyPets)  
{  
    cout << Pet.first << '\t' << Pet.second << endl;  
}
```

```
student@cse1325:/media/sf_VMS$ ./map4Demo.e  
A      Appa  
J      Josie  
S      Sylvester  
student@cse1325:/media/sf_VMS$
```

```
map<char, string> MyPets;
```

```
MyPets['A'] = "Appa";
MyPets['S'] = "Sylvester";
MyPets['S'] = "Shade";
MyPets['J'] = "Josie";
```

Duplicate key but [] overwrites!

```
for (auto Pet : MyPets)
{
    cout << Pet.first << '\t' << Pet.second << endl;
}
```

```
student@cse1325:/media/sf_VMS$ ./map4Demo.e
A      Appa
J      Josie
S      Shade
student@cse1325:/media/sf_VMS$
```

```
map<string, string> MyPets;

MyPets.insert(make_pair("AP", "Appa"));
MyPets.insert(make_pair("SY", "Sylvester"));
MyPets.insert(make_pair("SH", "Shade"));
MyPets.insert(make_pair("JO", "Josie"));

for (auto Pet : MyPets)
{
    cout << Pet.first << '\t' << Pet.second << endl;
}
```

```
student@cse1325:/media/sf_VMS$ ./map4Demo.e
AP      Appa
JO      Josie
SH      Shade
SY      Sylvester
student@cse1325:/media/sf_VMS$
```

Standard Template Library (STL)

C++ Standard Library

The Standard Library defines powerful, template-based, reuseable components that implement many common data structures and algorithms used to process those data structures.

Three key components of the Standard Library

Containers (templatized data structures)

Iterators

Algorithms

iterator

An iterator is an object that is pointing to some element in a range of elements (such as a container like `map`) that has the ability to iterate through the elements of that range.

An **iterator** is an object that can traverse (iterate over) a container class without the user having to know how the container is implemented. With many classes (particularly lists and the associative classes), iterators are the primary way elements of these classes are accessed.

Iterators provide an easy way to step through the elements of a container class without having to understand how the container class is implemented.

iterator

An iterator is best visualized as a pointer to a given element in the container, with a set of overloaded operators to provide a set of well-defined functions:

- * Dereferencing the iterator returns the element that the iterator is currently pointing at.
- ++ Moves the iterator to the next element in the container. Most iterators also provide -- to move to the previous element.
- == Basic comparison operators to determine if two iterators point to the same element.
- != To compare the values that two iterators are pointing at, dereference the iterators first, and then use a comparison operator.
- = Assign the iterator to a new position (typically the start or end of the container's elements). To assign the value of the element the iterator is pointing at, dereference the iterator first, then use the assign operator.

iterator

Each container includes four basic member functions for use with = (assignment)

begin() returns an iterator representing the beginning of the elements in the container.

end() returns an iterator representing the element just past the end of the elements.

cbegin() returns a const (read-only) iterator representing the beginning of the elements in the container.

cend() returns a const (read-only) iterator representing the element just past the end of the elements.

iterator

end() returns an iterator representing the element just past the end of the elements. It points to a non-existent element that is used to determine when the end of a container is reached.

cend() returns a const (read-only) iterator representing the element just past the end of the elements.

`end()` / `cend()` do not point to the last element in the list – they point just past the end.

This is done primarily to make looping easy: iterating over the elements can continue until the iterator reaches `end()`.

iterator

All containers provide (at least) two types of iterators:

container::iterator provides a read/write iterator

container::const_iterator provides a read-only iterator

Constant iterators cannot be used the container needs to be changed.

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

int main(void)
{
    std::vector<int> vect{0,1,2,3,4,5};

    std::vector<int>::const_iterator it;

    it = vect.begin();

    while (it != vect.end())
    {
        std::cout << *it << " ";
        ++it; // and iterate to the next element
    }

    std::cout << '\n';
}
```

What if `vect.begin() + 1`?

1 2 3 4 5

What if `it = vect.end() - 1`
and `it != vector.begin()`
and `-it`?

5 4 3 2 1 – don't get 0

What if we don't use `vect.end() - 1`
and we just use `vect.end()`?
Print garbage and then numbers

0	1	2	3	4	5
---	---	---	---	---	---

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

int main()
{
    std::vector<int> vect{0,1,2,3,4,5};

    std::vector<int>::iterator it;

    it = vect.begin();

    while (it != vect.end())
    {
        *it *= 3;
        std::cout << *it << " ";
        ++it; // and iterate to the next element
    }

    std::cout << '\n';
}
```

0	3	6	9	12	15
---	---	---	---	----	----

iterator

map uses a bidirectional iterator which means it supports ++ and –

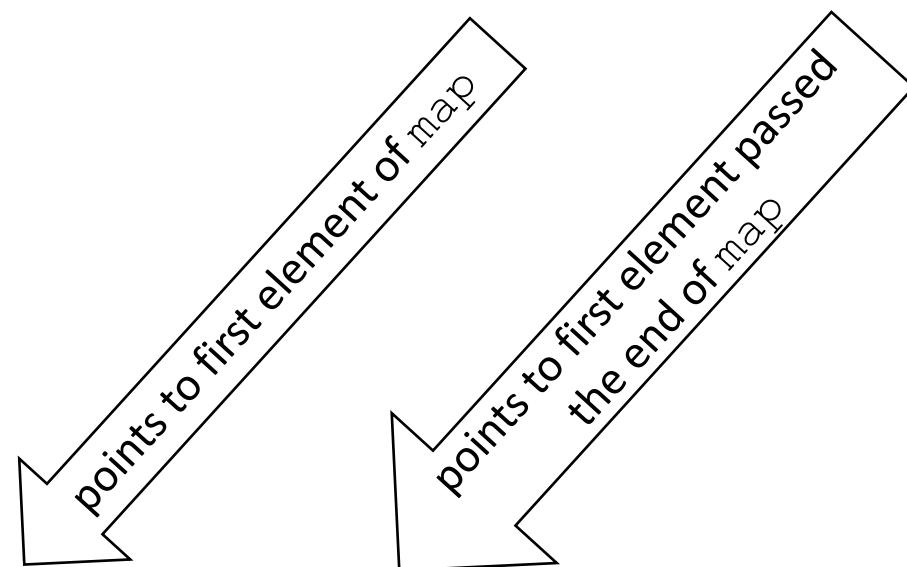
map iterators can be

dereferenced using *

accessed using ->

compared using ==

tested for inequality using !=



map iterators have member functions begin () and end ()

```
cout << "Key\tValue" << endl;
map<string, int>::iterator it = mapOfWords.begin();
while(it != mapOfWords.end())
{
    cout << it->first << "\t"
        << it->second << endl;
    it++;
}
```

The code demonstrates the use of a map's iterator to print its contents. It starts by defining a map of words and their counts. Then, it prints each key-value pair using a while loop that continues until the iterator reaches the map's end. The iterator is advanced after each output. The code ends with a final increment of the iterator.

Key	Value
earth	1
moon	4
sun	3

```
cout << "Enter a value to search for in our map ";
cin >> SearchValue;

it = mapOfWords.begin(); // reset iterator

while(it != mapOfWords.end())
{
    if (it->second == SearchValue)
    {
        cout << "Key " << it->first << " has value " << it->second << endl;
    }
    it++;
}
```

Key	Value
earth	1
moon	4
sun	3

```
Enter a value to search for in our map 4
Key moon has value 4
```

```
void PrintVector(vector<int> PV)
{
    cout << "\nMyVector contains\n" << endl;

    for (auto it : PV)
        cout << it << ' ';
    cout << endl;
}
```

```
void PrintVector(vector<int> PV)
{
    cout << "\nMyVector contains\n" << endl;

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

```
cout << "Key\tValue" << endl;
map<string, int>::iterator it = mapOfWords.begin();
while(it != mapOfWords.end())
{
    if (it->second != false)
    {
        cout << it->first << "\t"
            << it->second << endl;
    }
    it++;
}
```

Diagram annotations:

- An arrow labeled "iterator" points to the variable `it` in the first line of code.
- An arrow labeled "start a beginning of map" points to the `begin()` method call in the second line of code.
- A callout box around the condition `it != mapOfWords.end()` contains the text "end when iterator goes one past the end of the map".
- A callout box around the `if` statement condition `it->second != false` contains the text "eliminates \"added\" venus".

Key	Value
earth	1
moon	4
sun	3

```
cout << "Enter a value to search for in our map ";
cin >> SearchValue;

it = mapOfWords.begin(); // reset iterator

while(it != mapOfWords.end())
{
    if (it->second == SearchValue)
    {
        cout << "Key " << it->first << " has value " << it->second << endl;
    }
    it++;
}
```

Key	Value
earth	1
moon	4
sun	3

```
Enter a value to search for in our map 4
Key moon has value 4
```

CSE 1325

Week of 11/16/2020

Instructor : Donna French

Object Relationships

Inheritance



- Represents the "is a" relationship
- Shows the relationship between a super class/base class/parent and a derived/subclass/child.
- Arrow is on the side of the base class/parent

Inheritance

Inheritance involves creating new objects by directly acquiring the attributes and behaviors of other objects and then extending or specializing them.

Inheritance is everywhere in real life.

Most living things (including you) inherited traits from parents.

Even non living things can inherit traits from their predecessors.

Inheritance

When Apple decides to create the next generation of iPhone, it does not start from scratch when creating a new phone.

They start with what they already know about the current version of the iPhone and build upon that.

Most new version of electronics build upon the previous version. Not only does this lead to less work to create a new version but it also allows for backward compatibility.

Pet
EyeColor: String
Age: Float
Weight: Float
Location: String
eat (foodType)
sleep(timeLength)

<i>Person</i>
Name
Phone Number
Email Address
Purchase Parking Pass

has a

is a

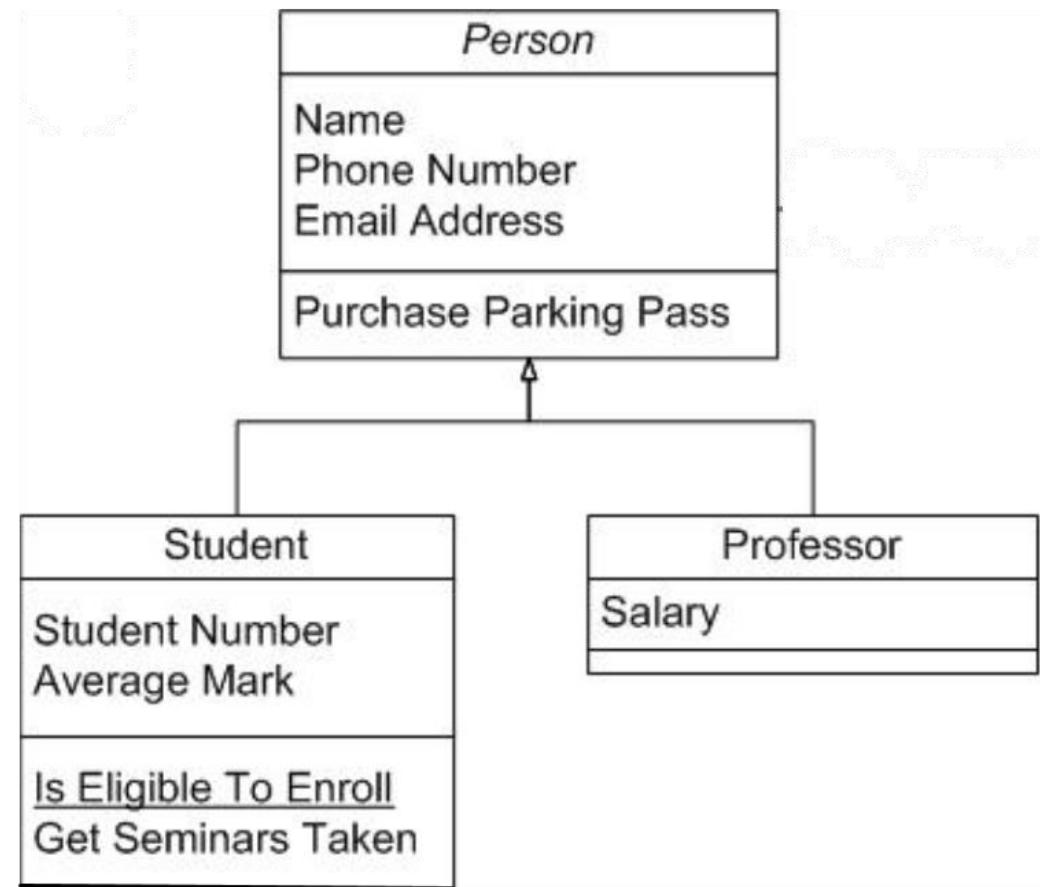
Inheritance

So does a student have a name, phone number and email address? Can a student purchase a parking pass?

Does a professor have a name, phone number and email address? Can a professor purchase a parking pass?

A professor has a salary – does every person have a salary?

A student has a student number – does every person have a student number?

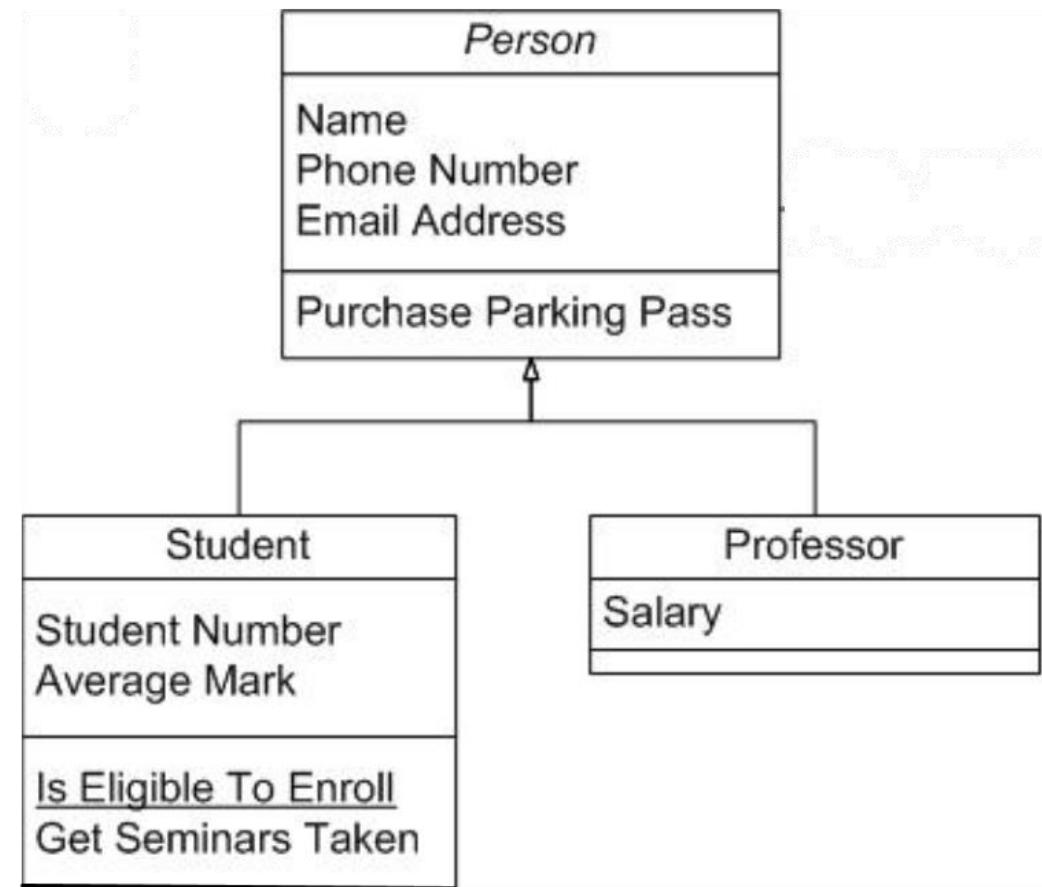


Inheritance

Rather than include name, phone number, email address and the ability to purchase a parking pass in our Student and Professor classes, we allow Student and Professor to inherit those attributes/abilities from Person.

This reduces the complexity of the Student and Professor class by making them contain less.

This also allows us to make changes to the Person class without directly changing Student and Professor.

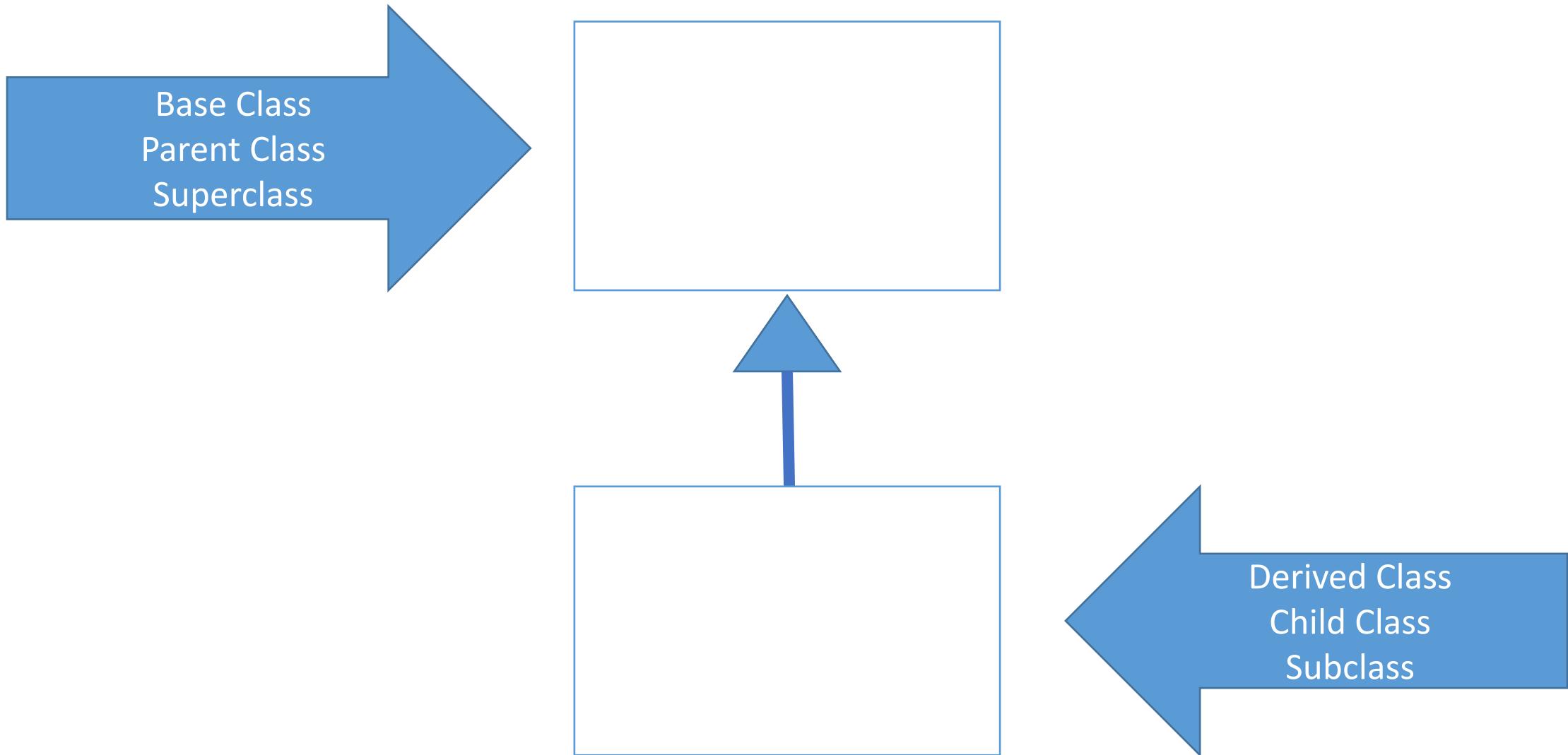


Inheritance

A form of software reuse in which you create a class that absorbs an existing class's data and behaviors and enhances them with new capabilities.

- You can designate that a new class should **inherit** the members of an existing class.
- This existing class is called the **base class**, and the new class is referred to as the **derived class**.

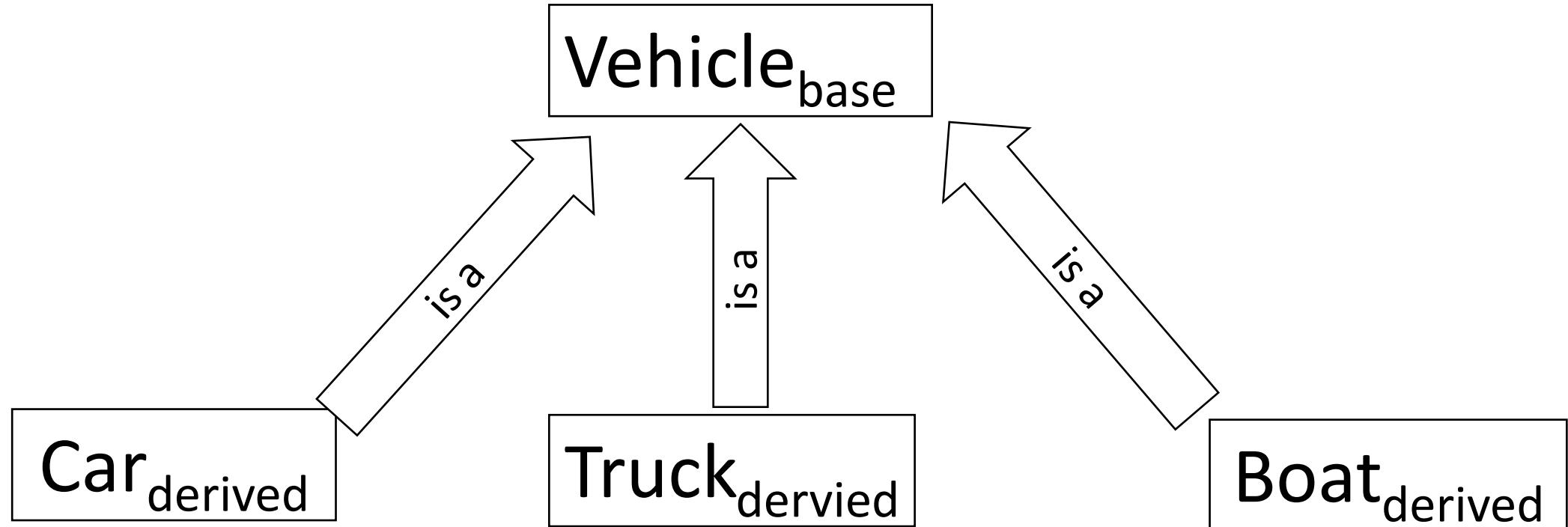
Inheritance



Inheritance

- A derived class represents a *more specialized* group of objects.
- C++ offers public, protected and private inheritance.
- With public inheritance, every object of a derived class is also an object of that derived class's base class.
- However, base-class objects are not objects of their derived classes.

base-class objects are not objects of their derived classes



every object of a derived class is also an object of that derived class's base class.

Inheritance

Base classes tend to be *more general* and derived classes tend to be *more specific*.

Base Class

Student

Shape

Loan

Employee

Account

Derived Class

GraduateStudent, UnderGraduateStudent

Circle, Triangle, Rectangle, Sphere, Cube

CarLoan, HomeImprovementLoan, StudentLoan

Faculty, Staff

CheckingAccount, SavingsAccount

Inheritance

Because every derived-class object *is an* object of its base class and one base class can have *many* derived classes, the set of objects represented by a base class typically is *larger* than the set of objects represented by any of its derived classes.

Base class `Vehicle` represents all vehicles including cars, boats, trucks, airplanes and bicycles.

Derived class `Car` represents a smaller, more specific subset of all vehicles.

Base class `Pet` represents all animals kept as pets whereas as derived class `Cat` is a specific subset of `Pet`.

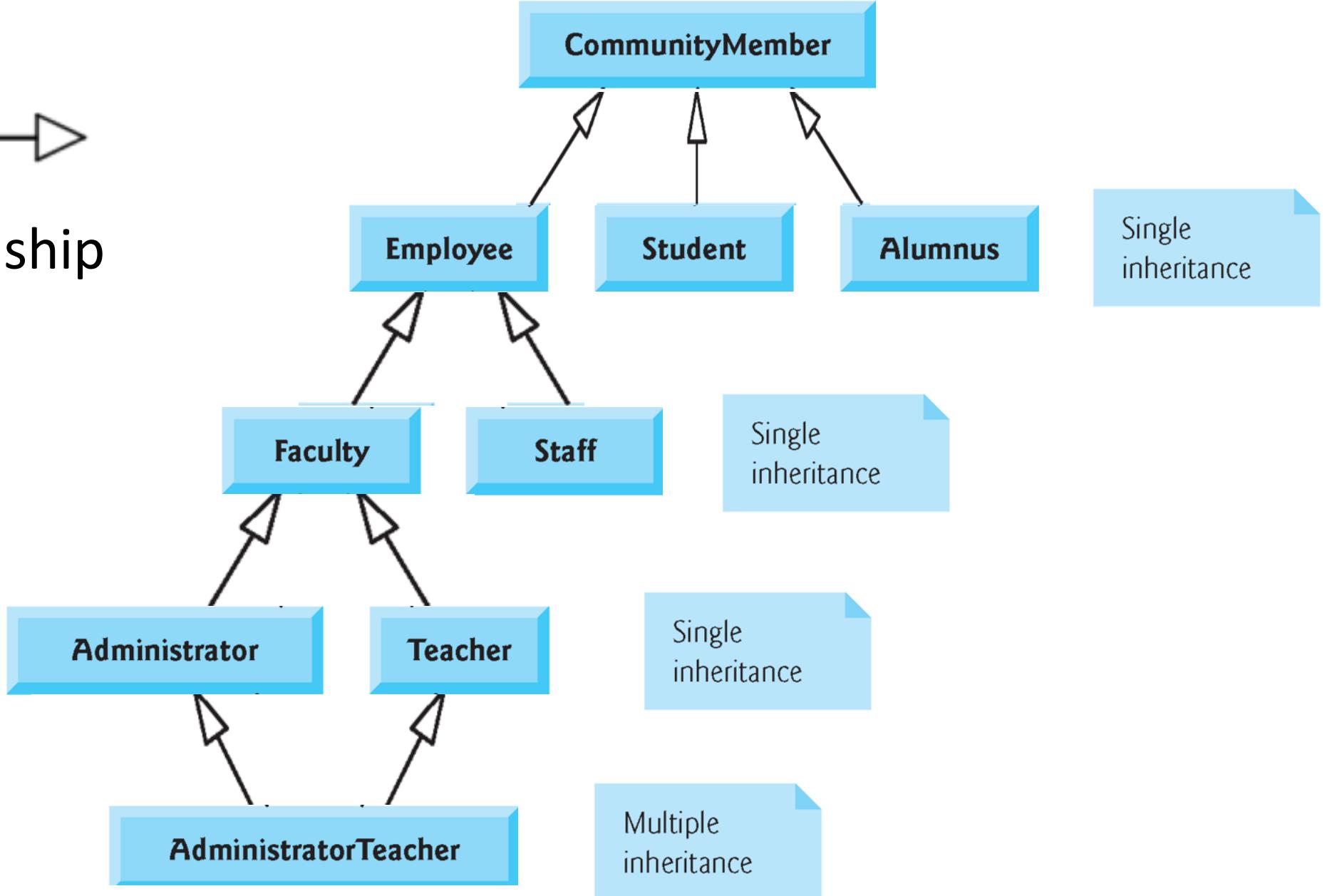
Inheritance

Inheritance relationships form **class hierarchies**.

- A base class exists in a hierarchical relationship with its derived classes.
- Although classes can exist independently, once they are associated with an inheritance relationships, they become related to other classes.
- A class becomes either a base class—supplying members to other classes, a derived class—inheriting its members from other classes, or *both*.

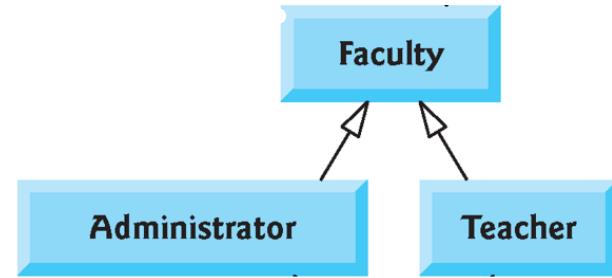


is-a relationship



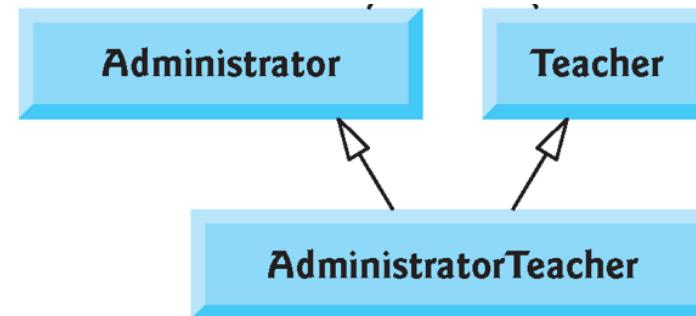
Inheritance

Single Inheritance



A class is derived from one base class

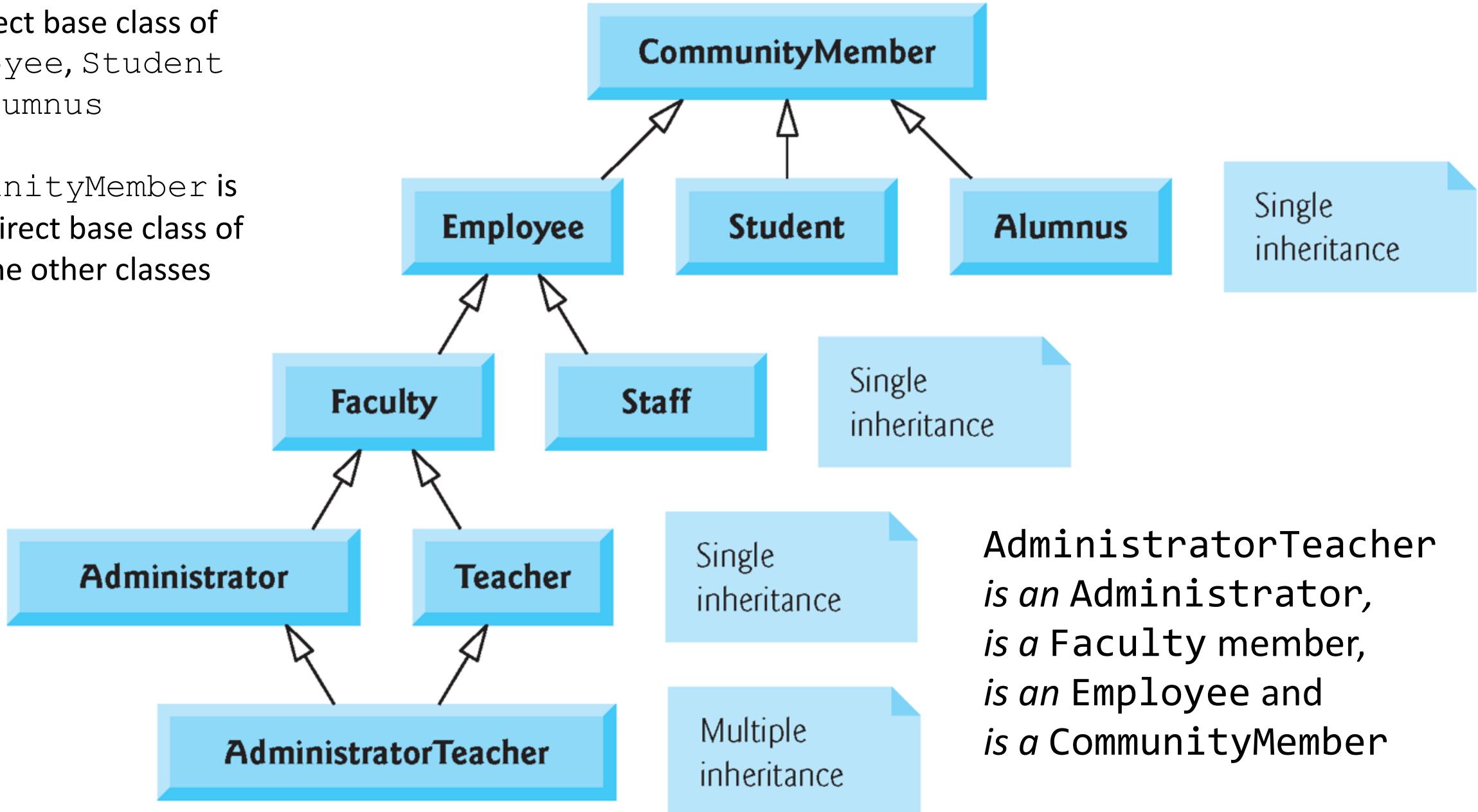
Multiple inheritance



A derived class inherits simultaneously from two or more (possibly unrelated) base classes.

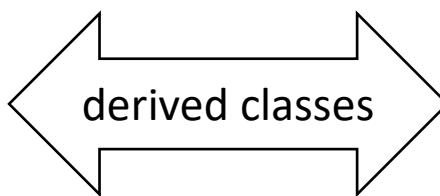
CommunityMember is the direct base class of Employee, Student and Alumnus

CommunityMember is the indirect base class of all of the other classes





base class



derived classes

Inheritance

3 forms of inheritance

public

private

protected

Regardless of the form of inheritance, private data members of the base class are not accessible directly from that class's derived classes.

Private base-class data members are still inherited – still considered parts of the derived class

Inheritance

Public inheritance

all other base-class members retain their original member access then they become members of the derived class

public members of the base class become public members of the derived class

Inheritance

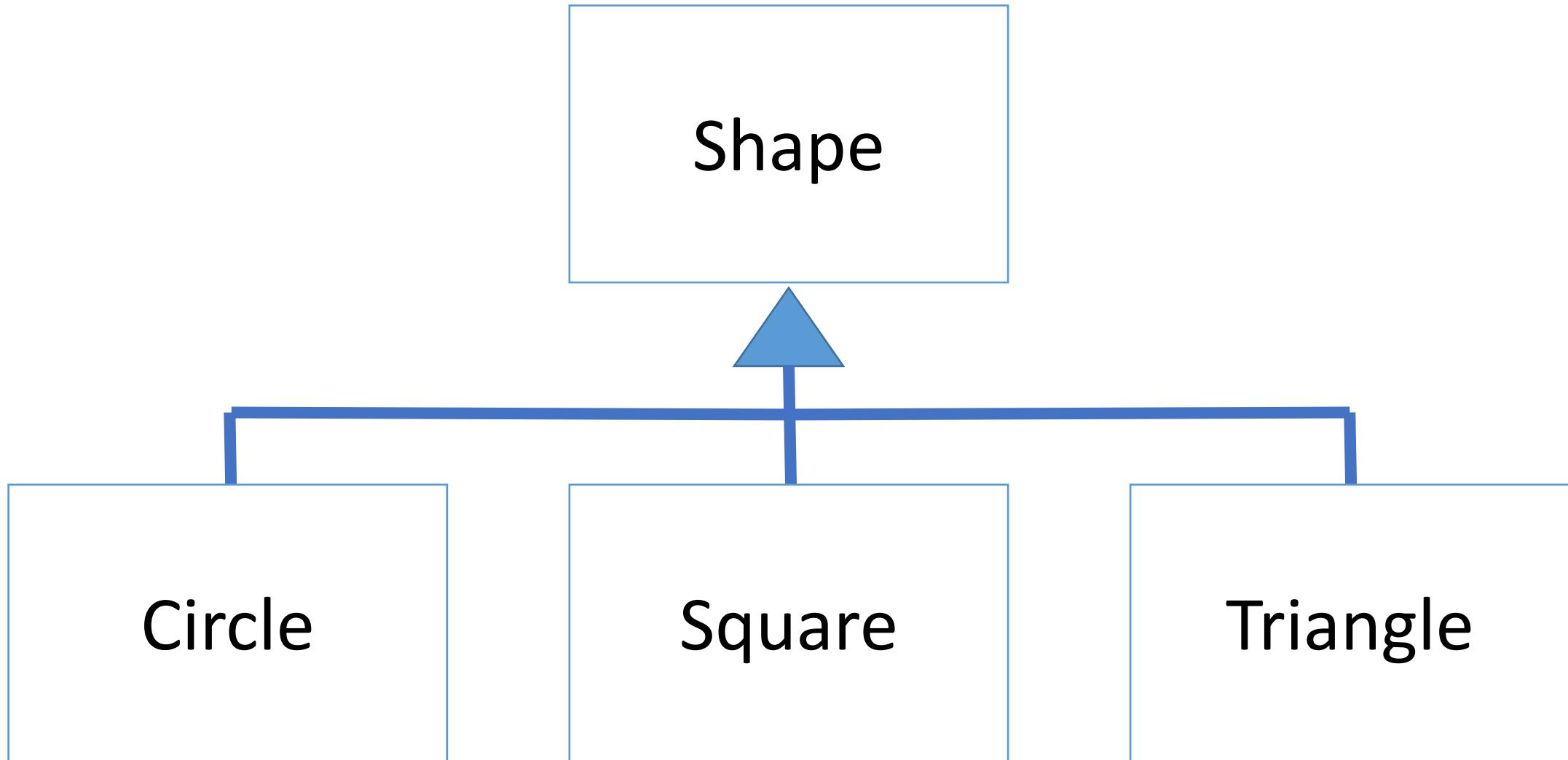
With inheritance, the common data members and member functions of all the classes in the hierarchy are declared in the base class.

When changes are required for these common features, you need to make the changes only the base class.

Derived classes then inherit the changes.

Without inheritance, changes would need to be made to all the source code files that contain a copy of the code in question.

Inheritance



```
class Shape
{
public :
    Shape(std::string name) : ShapeName{name}
    {
    }
    std::string getName()
    {
        return ShapeName;
    }
    float dim1;
    float dim2;
    std::string ShapeName;
};
```

```
int main (void)
{
    Shape A("Poly");

    std::cout << "My name is "
                  << A.getName()
                  << std::endl;

    return 0;
}
```

My name is Poly

Now I want to create a class Circle.

The more abstract version of Circle is Shape.

Shape knows its name and how to get its name. Shape also knows dimensions.

We want Circle to know these same things but we also want Circle to calculate its area.

When we create a Circle object, we want to construct it with its dimension/radius set already.

```
class Circle
{
    public:
        Circle(float radius=0)
    {
        dim1 = dim2 = radius;
    }

    float getarea()
    {
        return dim1 * dim2 * M_PI;
    }
};
```

```
student@cse1325:/media/sf_VM$ make
g++ -c -g -std=c++11 ShapeInherit.cpp -o ShapeInherit.o
ShapeInherit.cpp: In constructor 'Circle::Circle(float)':
ShapeInherit.cpp:28:3: error: no matching function for call to
`Shape::Shape()'
{
^
ShapeInherit.cpp:11:3: note: candidate: Shape::Shape(std::__cxx11::string)
    Shape(std::string name) : ShapeName{name}
^~~~~
ShapeInherit.cpp:11:3: note:    candidate expects 1 argument, 0 provided
ShapeInherit.cpp:8:7: note: candidate: Shape::Shape(const Shape&)
class Shape
^~~~~
ShapeInherit.cpp:8:7: note:    candidate expects 1 argument, 0 provided
ShapeInherit.cpp:8:7: note: candidate: Shape::Shape(Shape&&)
ShapeInherit.cpp:8:7: note:    candidate expects 1 argument, 0 provided
makefile:14: recipe for target 'ShapeInherit.o' failed
make: *** [ShapeInherit.o] Error 1
```

```
ShapeInherit.cpp: In constructor 'Circle::Circle(float)':  
ShapeInherit.cpp:28:3: error: no matching function for call to  
'Shape::Shape()'
```

```
Shape(std::string name) : ShapeName{name}  
{  
}
```

C++ requires that a derived-class constructor (`Circle`) call its base class constructor (`Shape`) to initialize the base class (`Shape`) data members that are inherited into the derived class (`Circle`).

We could simply add default values to `Shape`'s constructor so that `Circle` can call `Shape`'s constructor without any parameters.

```
Shape(std::string name="BaseShape") : ShapeName{name}  
{  
}
```

But this solution does not allow us to name our `Circle` objects – they would all be `BaseShape`.

```
class Circle : public Shape
{
public:
    Circle(float radius=0)
    {
        dim1 = dim2 = radius;
    }

    float getarea()
    {
        return dim1 * dim2 * M_PI;
    }
};
```

We need to alter Circle's constructor – it needs to accept a name that it can then pass on to Shape's constructor.

```
class Circle : public Shape
{
public:
    Circle(std::string name, float radius=0)
        : Shape(name)
    {
        dim1 = dim2 = radius;
    }

    float getarea()
    {
        return dim1 * dim2 * M_PI;
    }
};
```

```
Shape A("Poly");  
  
std::cout << "My name is " << A.getName() << std::endl;  
  
Circle C("Hoop");  
  
std::cout << "My name is " << C.getName() << std::endl;
```

My name is Poly
My name is Hoop

Object C of class Circle is able to use the member function
getName() even though class Circle does not contain getName()
Circle inherited it from Shape

```
69           Shape A("Poly");
```

```
(gdb) ptype A
```

```
type = class Shape {
    public:
        float dim1;
        float dim2;
        std::__cxx11::string ShapeName;
```

```
    Shape(std::__cxx11::string);
    std::__cxx11::string getName(void);
```

```
}
```

```
(gdb) p A
```

```
$2 = {
    dim1 = 1.40129846e-45,
    dim2 = 0,
    ShapeName = "Poly"
}
```

73

```
Circle C("Hoop");
```

```
(gdb) ptype C
```

```
type = class Circle : public Shape {  
public:  
    Circle(std::__cxx11::string, float);  
    float getarea(void);  
}
```

```
(gdb) p C
```

```
$3 = {  
<Shape> = {  
    dim1 = 0,  
    dim2 = 0,  
    ShapeName = "Hoop"  
}, <No data fields>}
```

Why is dim1 and dim2 set to 0?

```
Circle(std::string name, float radius=0)  
: Shape(name)  
{  
    dim1 = dim2 = radius;  
}
```

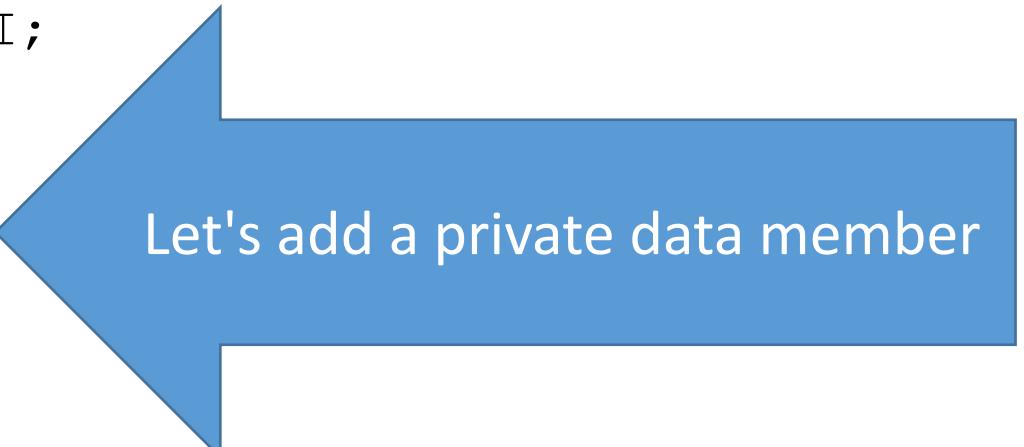
```
class Circle : public Shape
{
public:
    Circle(std::string name, float radius=0)
        : Shape(name)
    {
        dim1 = dim2 = radius;
    }

    float getarea()
    {
        return dim1 * dim2 * M_PI;
    }

private :
    std::string color;
};
```

And instantiate our object with a radius

```
Circle C("Hoop", 3);
```



Let's add a private data member

76

```
Circle C("Hoop", 3);
```

(gdb) p C

```
$2 = {  
    <Shape> = {  
        dim1 = 3,  
        dim2 = 3,  
        ShapeName = "Hoop"  
    },  
    members of Circle:  
    color = ""  
}
```



Inherited from Shape

Defined in Circle

```
class Circle : public Shape
{
public:
    Circle(std::string name, float radius=0)
        : Shape(name)
    {
        dim1 = dim2 = radius;
    }

    float getarea()
    {
        return dim1 * dim2 * M_PI;
    }

private :
    std::string color;
};
```

My name is Hoop and my area is 28.2743

```
Circle C("Hoop", 3);

std::cout << "My name is " << C.getName()
             << " and my area is "
             << C.getarea() << std::endl;
```

Let's add another derived class - Rectangle

```
class Rectangle : public Shape
{
public:
    Rectangle(std::string name, float height=0, float width=0)
        : Shape(name)
    {
        dim1 = height;
        dim2 = width;
    }

    float getarea()
    {
        return dim1 * dim2;
    }
};
```

```
Rectangle R("NotQuiteSquare", 4, 6);

std::cout << "My name is " << R.getName()
             << " and my area is "
             << R.getarea() << std::endl;
```

My name is NotQuiteSquare and my area is 24

```
(gdb) p R  
$2 = {  
    <Shape> = {  
        dim1 = 4,  
        dim2 = 6,  
        ShapeName = "NotQuiteSquare"  
    }, <No data fields>}
```

Now, let's add a new shape – a square.

How is a square different from a rectangle?

A square is a special type of rectangle where all four sides have the same length.

So a square is a shape and a rectangle which means

- the area calculation for a square is the same as a rectangle.
- rectangle's area calculation requires two sides ($l * w$) so square could use the same calculation as long as $length = width$.

```
class Square
{
public:
    Square(float size)
    {
        dim1 = size;
        dim2 = size;
    }
};
```

My name is Quad and my area is 16

```
Square S("Quad", 4);

std::cout << "My name is " << S.getName()
             << " and my area is "
             << S.getarea() << std::endl;
```

```
class Square : public Rectangle
{
public:
    Square(std::string name, float size)
        : Rectangle(name, size)
    {
        dim1 = size;
        dim2 = size;
    }
};
```

85

 Square S ("Quad", 4);

(gdb) p S

\$2 = {

 <Rectangle> = {

 <Shape> = {

 dim1 = 4,

 dim2 = 4,

 ShapeName = "Quad"

 }, <No data fields>}, <No data fields>

```
class Square : public Rectangle
{
public:
    Square(std::string name, float size)
        : Rectangle(name, size)
    {
        dim1 = size;
        dim2 = size;
    }
private:
    std::string location{"Line 68"};
};
```

(gdb) p S
\$2 = {
 <Rectangle> = {
 <Shape> = {
 dim1 = 4,
 dim2 = 4,
 ShapeName = "Quad"
 }, <No data fields>},
 members of Square:
 location = "Line 68"
}

Inheritance

```
class Circle : public Shape
```

```
class Rectangle : public Shape
```

```
class Square : public Rectangle
```

colon (:) in the class definition indicates inheritance

keyword public indicates the type of inheritance

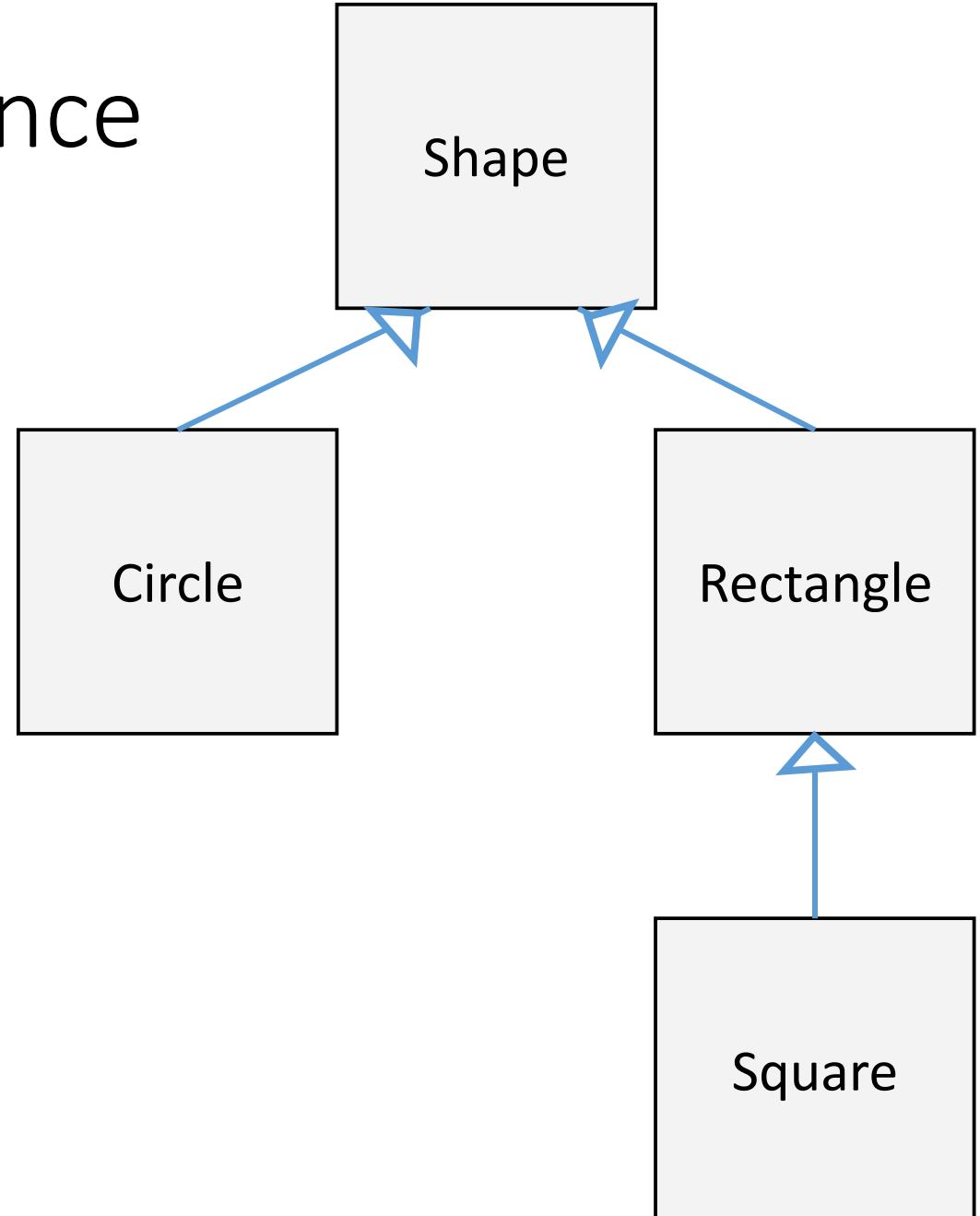
Constructors and destructors are not inherited

Inheritance

When C++ constructs derived objects, it does so in phases.

First, the most-base class (at the top of the inheritance tree) is constructed first.

Then each child class is constructed in order, until the most-child class (at the bottom of the inheritance tree) is constructed last.

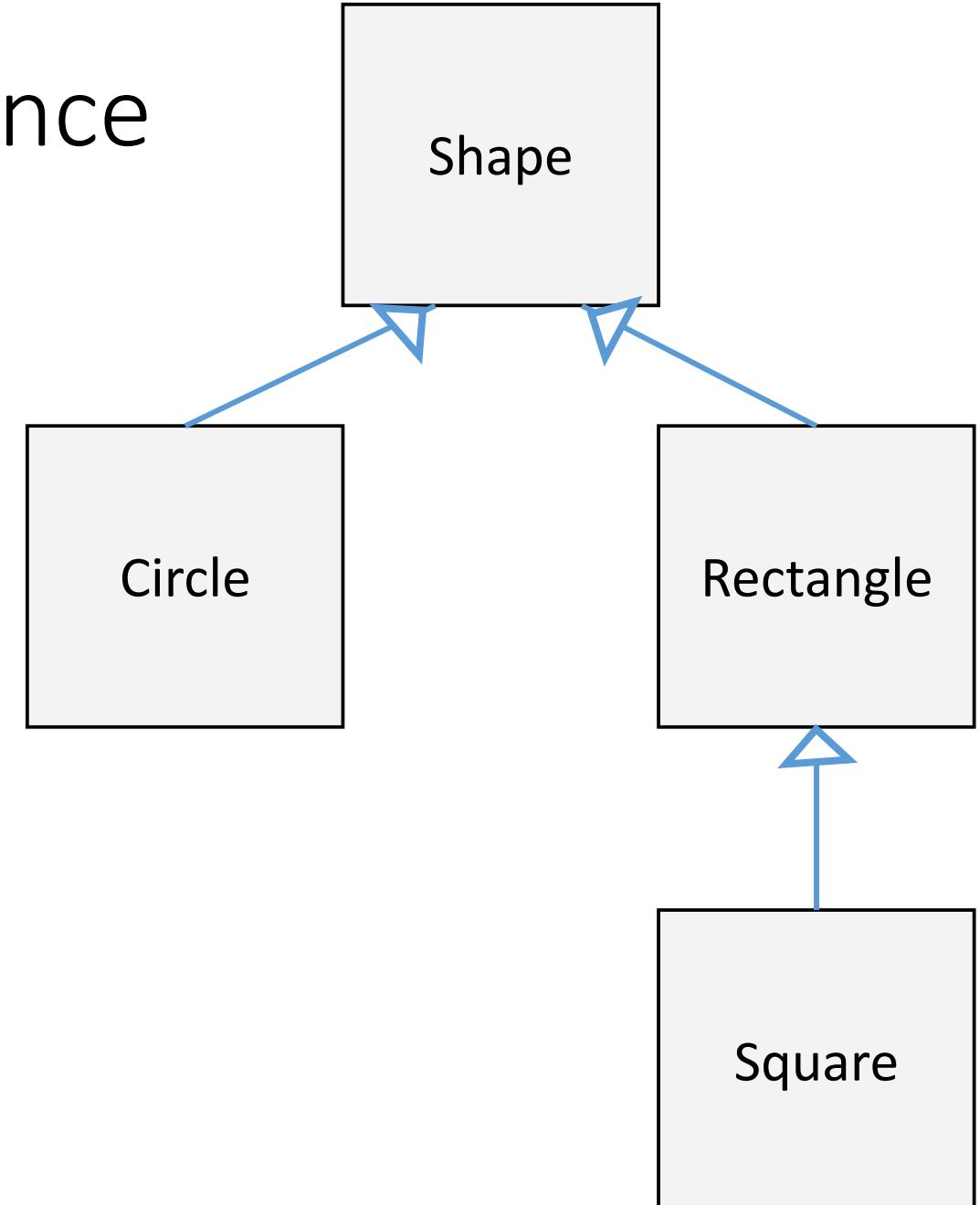


Inheritance

So when we construct a Circle, a Shape is constructed first and then the Circle is constructed.

When we construct a Square, a Shape is constructed and then a Rectangle and then a Square.

A child cannot exist until the parent exists.



Inheritance

```
std::cout << "Let's make a Shape" << std::endl;
```

```
Shape A("Poly");
```

```
std::cout << "My name is " << A.getName() << std::endl;
```

Let's make a Shape
SHAPE!

My name is Poly

```
Shape(std::string name="BaseShape")  
: ShapeName{ name }  
{  
    std::cout << "SHAPE!" << std::endl;  
}
```

Inheritance

```
std::cout << "Let's make a Circle" << std::endl;
```

```
Circle C("Hoop", 3);
```

```
std::cout << "My name is " << C.getName()
             << " and my area is " << C.getArea()
             << std::endl;
```

Let's make a Circle

SHAPE!

CIRCLE!

My name is Hoop and my area is 28.2743

```
Circle(std::string name, float radius=0)
: Shape(name)
{
    dim1 = dim2 = radius;
    std::cout << "CIRCLE!" << std::endl;
}
```

Inheritance

```
std::cout << "Let's make a Rectangle" << std::endl;
```

```
Rectangle R("NotQuiteSquare", 4, 6);
```

```
std::cout << "My name is " << R.getName()
             << " and my area is " << R.getArea()
             << std::endl;
```

Let's make a Rectangle

SHAPE!

RECTANGLE!

My name is NotQuiteSquare and m

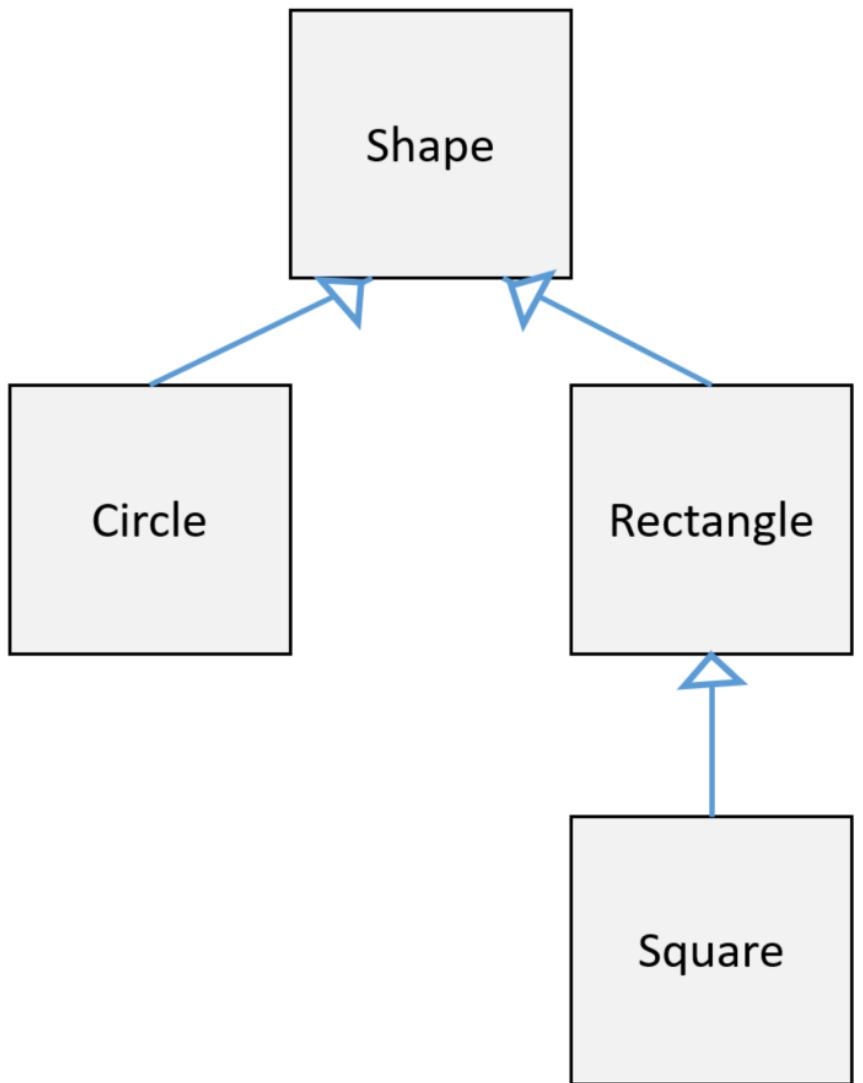
```
Rectangle(std::string name, float
          height=0, float width=0)
: Shape(name)
{
    dim1 = height;
    dim2 = width;
    std::cout << "RECTANGLE!"
              << std::endl;
}
```

```
    Square(std::string name, float size)
: Rectangle(name, size)
{
    dim1 = size;
    dim2 = size;
    std::cout << "SQUARE!" << std::endl;
}
```

```
std::cout << "Let's make a Square" << std::endl;
```

```
Square S("Quad", 4);
```

```
std::cout << "My name is " << S.getName()
             << " and my area is " << S.getarea()
             << std::endl;
```



Shape

Circle

Rectangle

Square

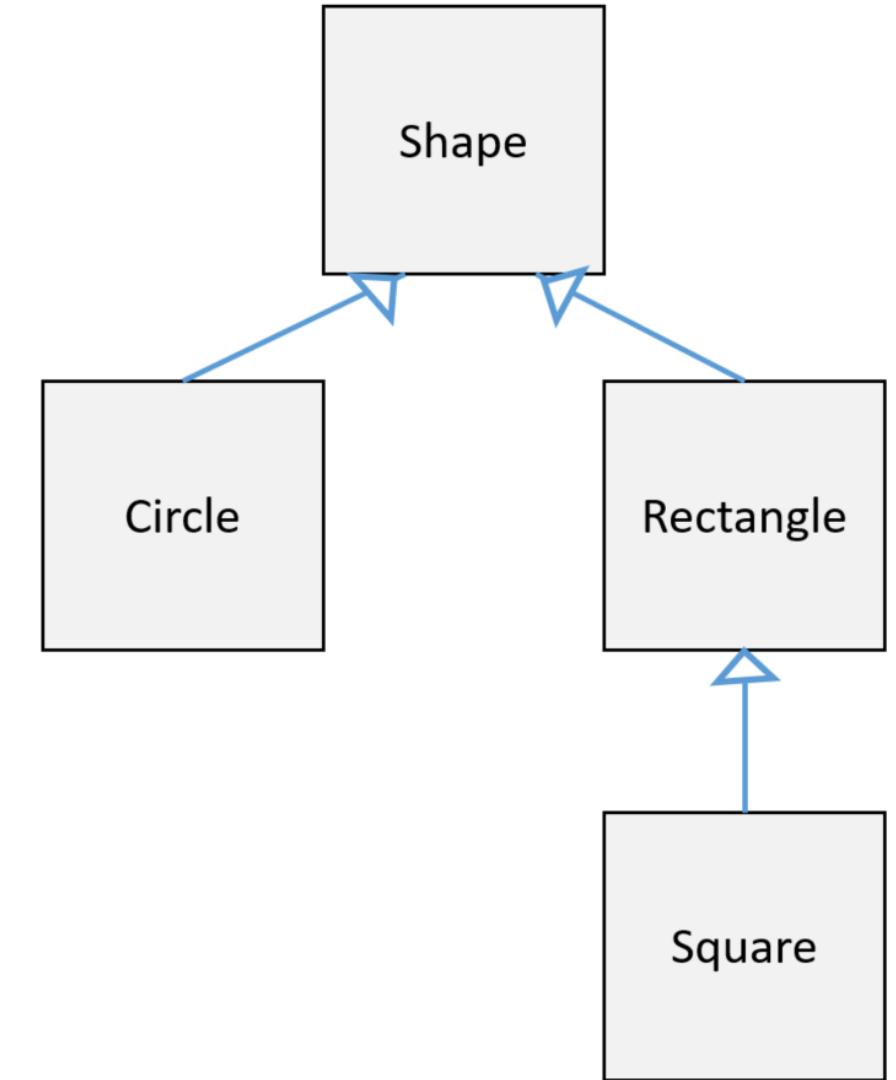
Let's make a Square

SHAPE !

RECTANGLE !

SQUARE !

My name is Quad and my area is 16



Inheritance

The derived class often uses variables and functions from the base class but the base class knows nothing about the derived class.

Instantiating the base class first ensures those variables are already initialized by the time the derived class is created and ready to use them.

Remember that C++ always constructs the “first” or “most base” class first. It then walks through the inheritance tree in order and constructs each successive derived class.

Inheritance

With non-derived classes, constructors only have to worry about their own members.

For example, consider Shape. We can create a Shape object like this:

```
Shape A ("Poly");
```

Here's what actually happens when Shape is instantiated:

- Memory for Shape is set aside
- The appropriate Shape constructor is called
- The initialization list initializes variables
- The body of the constructor executes
- Control is returned to the caller

Inheritance

With derived classes, a few more things happen

For example, consider `Circle`. We can create a `Circle` object like this:

```
Circle A ("Hoop");
```

Here's what actually happens when `Circle` is instantiated:

- Memory for derived is set aside (enough for both the `Shape` and `Circle` portions)
- The appropriate `Circle` constructor is called
- The `Shape` object is constructed first using the appropriate `Shape` constructor. If no base constructor is specified, the default constructor will be used.
- The initialization list initializes variables
- The body of the constructor executes
- Control is returned to the caller

Inheritance

The only real difference between constructing an object that inherits and an object that does not inherit is that before the Derived constructor can do anything substantial, the Base constructor is called first.

The Base constructor sets up the Base portion of the object, control is returned to the Derived constructor, and the Derived constructor is allowed to finish up its job.

```
class Circle : public Shape
{
public:
    Circle(std::string name)
        : Shape(name)
    {
        dim1 = dim2 = radius;
        std::cout << "CIRCLE!" << std::endl;
    }

    float getarea()
    {
        return dim1 * dim2 * M_PI;
    }

private :
    std::string color;
};
```

What happens if we call Shape's constructor as Shape () rather than Shape (name) ?

```
Shape(std::string name="BaseShape") :
    ShapeName{ name }
{
    std::cout << "SHAPE!" << std::endl;
}
```

Let's make a Circle
SHAPE!
CIRCLE!



My name is Hoop and my area is 28.2743

Let's make a Circle
SHAPE!
CIRCLE!

My name is BaseShape and my area is 28.2743

We called `Shape()` and not
`Shape(name)` so the default
parameter value of `BaseShape`
was used to construct `Circle`.

```
Shape(std::string name="BaseShape") :  
    ShapeName{ name }  
{  
    std::cout << "SHAPE!" << std::endl;  
}
```

Constructors and Destructors in Derived Classes

- Instantiating a derived-class object begins a *chain* of constructor calls in which the derived-class constructor, before performing its own tasks, invokes its direct base class's constructor either explicitly (via a base-class member initializer) or implicitly (calling the base class's default constructor).
- If the base class is derived from another class, the base-class constructor is required to invoke the constructor of the next class up in the hierarchy, and so on.
- The last constructor called in this chain is the constructor of the class at the base of the hierarchy, whose body actually finishes executing *first*.
- The most derived-class constructor's body finishes executing *last*.
- Each base-class constructor initializes the base-class data members that the derived-class object inherits.

Constructors and Destructors in Derived Classes

- When a derived-class object is destroyed, the program calls that object's destructor.
- This begins a chain (or cascade) of destructor calls in which the derived-class destructor and the destructors of the direct and indirect base classes and the classes' members execute in *reverse* of the order in which the constructors executed.
- When a derived-class object's destructor is called, the destructor performs its task, then invokes the destructor of the next base class up the hierarchy.
- This process repeats until the destructor of the final base class at the top of the hierarchy is called.
- Then the object is removed from memory.

Constructors and Destructors in Derived Classes

Base-class constructors, destructors and overloaded assignment operators are *not* inherited by derived classes.

Derived-class constructors, destructors and overloaded assignment operators, however, can call base-class versions.

If the derived class does not explicitly define constructors, the compiler still generates a default constructor in the derived class.

```
class Shape
{
public :
    Shape(std::string name="BaseShape") : ShapeName{name}
    {
        std::cout << "SHAPE!" << std::endl;
    }
    std::string getName()
    {
        return ShapeName;
    }

    float dim1;
    float dim2;
    std::string ShapeName;
};
```



Up to now, we kept Shape's data members in the public access. What happens if we change them to private?

```
student@cse1325:/media/sf_VMS make
g++ -c -g -std=c++11 ShapeInheritp.cpp -o ShapeInheritp.o
ShapeInheritp.cpp: In constructor 'Circle::Circle(std::__cxx11::string, float)':
ShapeInheritp.cpp:33:4: error: 'float Shape::dim1' is private within this context
    dim1 = dim2 = radius;
    ^~~~
ShapeInheritp.cpp:22:9: note: declared private here
    float dim1;
    ^~~~
ShapeInheritp.cpp:33:11: error: 'float Shape::dim2' is private within this context
    dim1 = dim2 = radius;
    ^~~~
ShapeInheritp.cpp:23:9: note: declared private here
    float dim2;
    ^~~~
ShapeInheritp.cpp: In member function 'float Circle::getarea()':
ShapeInheritp.cpp:39:11: error: 'float Shape::dim1' is private within this context
    return dim1 * dim2 * M_PI;
    ^~~~
ShapeInheritp.cpp:22:9: note: declared private here
    float dim1;
    ^~~~
ShapeInheritp.cpp:39:18: error: 'float Shape::dim2' is private within this context
    return dim1 * dim2 * M_PI;
    ^~~~
ShapeInheritp.cpp:23:9: note: declared private here
    float dim2;
    ^~~~
ShapeInheritp.cpp: In constructor 'Rectangle::Rectangle(std::__cxx11::string, float, float)':
ShapeInheritp.cpp:52:4: error: 'float Shape::dim1' is private within this context
    dim1 = height;
    ^~~~
ShapeInheritp.cpp:22:9: note: declared private here
    float dim1;
    ^~~~
ShapeInheritp.cpp:53:4: error: 'float Shape::dim2' is private within this context
    dim2 = width;
    ^~~~
ShapeInheritp.cpp:23:9: note: declared private here
    float dim2;
    ^~~~
ShapeInheritp.cpp: In member function 'float Rectangle::getarea()':
ShapeInheritp.cpp:59:11: error: 'float Shape::dim1' is private within this context
    return dim1 * dim2;
    ^~~~
ShapeInheritp.cpp:22:9: note: declared private here
    float dim1;
    ^~~~
ShapeInheritp.cpp:59:18: error: 'float Shape::dim2' is private within this context
    return dim1 * dim2;
    ^~~~
ShapeInheritp.cpp:23:9: note: declared private here
    float dim2;
    ^~~~
ShapeInheritp.cpp: In constructor 'Square::Square(std::__cxx11::string, float)':
ShapeInheritp.cpp:69:4: error: 'float Shape::dim1' is private within this context
    dim1 = size;
    ^~~~
ShapeInheritp.cpp:22:9: note: declared private here
    float dim1;
    ^~~~
ShapeInheritp.cpp:70:4: error: 'float Shape::dim2' is private within this context
    dim2 = size;
    ^~~~
ShapeInheritp.cpp:23:9: note: declared private here
    float dim2;
    ^~~~
makefile:14: recipe for target 'ShapeInheritp.o' failed
make: *** [ShapeInheritp.o] Error 1
```

```
ShapeInheritp.cpp: In constructor `Circle::Circle(std::__cxx11::string,
float)` :
ShapeInheritp.cpp:33:4: error: `float Shape::dim1` is private within this
context
    dim1 = dim2 = radius;
    ^~~~
ShapeInheritp.cpp:22:9: note: declared private here
    float dim1;
    ^~~~
ShapeInheritp.cpp:33:11: error: `float Shape::dim2` is private within this
context
    dim1 = dim2 = radius;
    ^~~~
ShapeInheritp.cpp:23:9: note: declared private here
    float dim2;
```

Inheritance

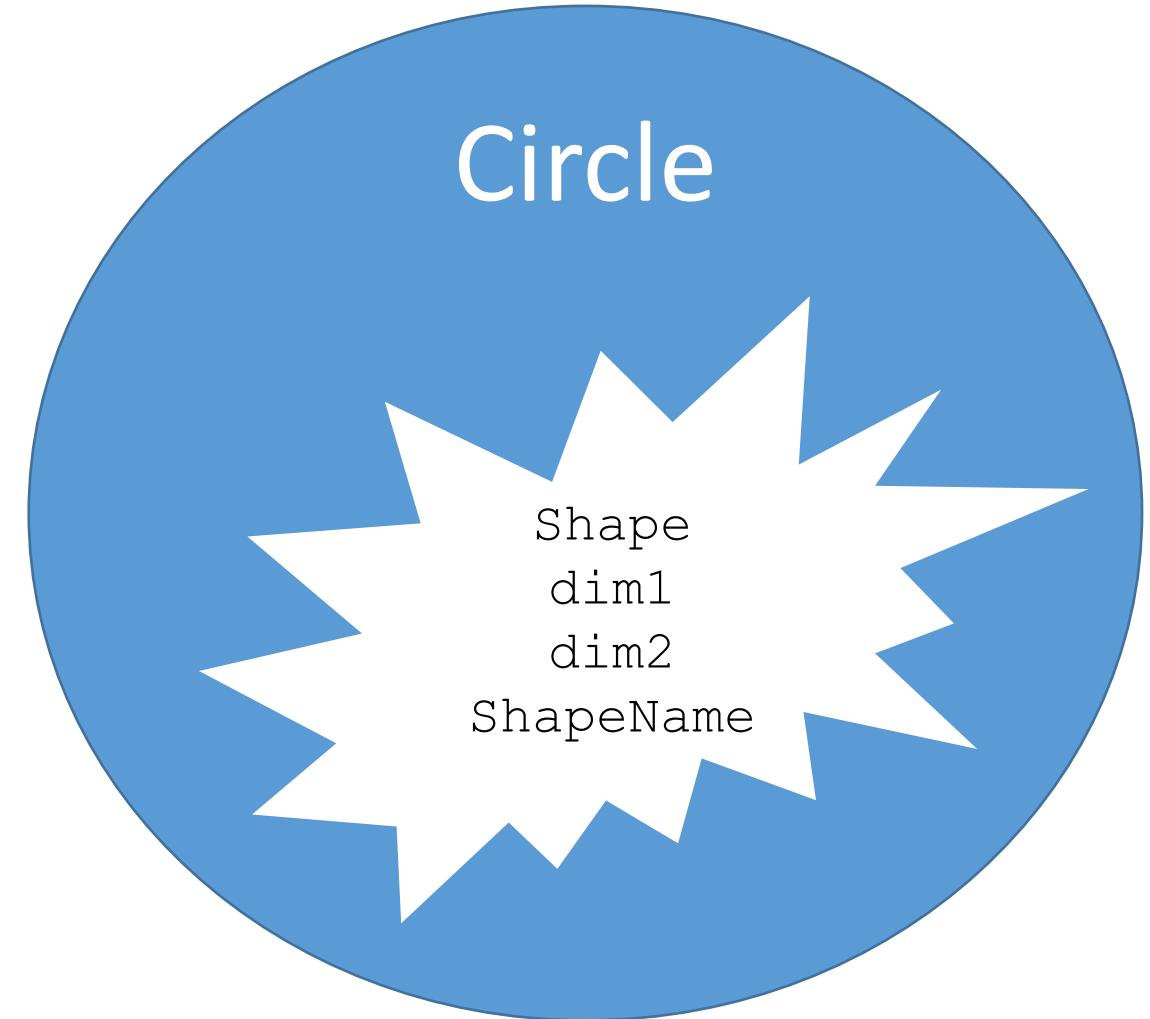
Public members can be accessed by anybody.

Private members can only be accessed by member functions of the same class or friends.

This means derived classes cannot access private members of the base class directly.

Derived class Circle inherits Shape's information but is not allowed to directly access the private data members inherited from Shape.

Derived classes will need to use access functions to access private members of the base class.



We would need to add getters and setters to `Shape` to provide access to private data members.

```
void set_dims(float Dim1, float Dim2)
{
    dim1 = Dim1;
    dim2 = Dim2;
}
float get_dim1()
{
    return dim1;
}
float get_dim2()
{
    return dim2;
}
```

We would then change Circle to use them.

```
Circle(std::string name, float radius=0)
: Shape(name)
{
    set_dims(radius, radius);
    std::cout << "CIRCLE!" << std::endl;
}

float getarea()
{
    return get_dim1() * get_dim2() * M_PI;
}
```

Let's make a Circle
SHAPE!
CIRCLE!
My name is Hoop and my area is 28.2743

Including the Base-Class Header in the Derived-Class Header with #include

We #include the base class's header in the derived class's header. This is necessary for three reasons.

The derived class uses the base class's name, so we must tell the compiler that the base class exists.

The compiler uses a class definition to determine the size of an object of that class. A client program that creates an object of a class #includes the class definition to enable the compiler to reserve the proper amount of memory.

The compiler must determine whether the derived class uses the base class's inherited members properly.

```
class Common
{
public :
    void getInfo(void)
    {
        cout << "Enter your name " << endl;
        cin >> name;
        cout << "Enter your gender " << endl;
        cin >> gender;
        cout << "\nEnter your age " << endl;
        cin >> age;
    }
    void displayInfo(void)
    {
        cout << "Display Data" << endl;
        cout << "Name\t" << name << endl;
        cout << "Gender\t" << gender << endl;
        cout << "Age\t" << age << endl;
    }
private :
    string name;
    string gender;
    int age;
};
```

```
class Principal : public Common
{
public :
void getSalary(void)
{
    cout << "Enter Principal salary ";
    cin >> salary;
}

void showSalary(void)
{
    cout << "Principal Salary : "
        << salary << endl;
}

private :
int salary;
};
```

```
87          Principal Pal;
(gdb) p Pal
$2 = {
<Common> = {
    name = "", 
    gender = "", 
    age = 6299808
},
members of Principal:
salary = 0
}
```

```
class Teacher : public Common
{
public :
    void getSalary(void)
    {
        cout << "Enter Teacher salary ";
        cin >> salary;
    }

    void showSalary(void)
    {
        cout << "Teacher Salary : "
            << salary << endl;
    }

private :
    int salary;
};
```

```
94              Teacher Mr;

(gdb) p Mr
$3 = {
<Common> = {
    name = "", 
    gender = "", 
    age = 6299112
},
members of Teacher:
salary = 0
}
```

```
class Student : public Common
{
public :
    void getGrade(void)
    {
        cout << "Enter Student grade ";
        cin >> grade;
    }

    void showGrade(void)
    {
        cout << "Student Grade : "
            << grade << endl;
    }

private :
    int grade;
};
```

```
101          Student You;

(gdb) p You
$4 = {
<Common> = {
    name = "", 
    gender = "", 
    age = 4198944
},
members of Student:
grade = 0
}
```

```
87 Principal Pal;      89 Pal.getSalary();          (gdb) p Pal
(gdb) n                  (gdb) n
88 Pal.getInfo();        Enter Principal salary 12345 $5 = {
(gdb) n
Enter your name           91 Pal.displayInfo();       <Common> = {
Fred                      (gdb) n
Display Data
Name Fred
Gender Male
Age 34
92 Pal.showSalary();      (gdb) n
Principal Salary : 12345 (gdb) ptype Pal
                                         type = class Principal : public
                                         Common {
                                         private:
                                         int salary;

                                         public:
                                         void getSalary(void);
                                         void showSalary(void);
                                         }
```

```
94 Teacher Mr;
(gdb) n
95 Mr.getInfo();
(gdb) n
Enter your name
Bob

Enter your gender
Male

Enter your age
45

96 Mr.getSalary();
(gdb) n
Enter Teacher salary 23456
98 Mr.displayInfo();
(gdb) n
Display Data
Name Bob
Gender Male
Age 45
99 Mr.showSalary();
(gdb) n
Principal Salary : 23456

(gdb) p Mr
$6 = {
<Common> = {
    name = "Bob",
    gender = "Male",
    age = 45
},
members of Teacher:
salary = 23456
}

(gdb) ptype Mr
type = class Teacher : public
Common {
private:
    int salary;

public:
    void getSalary(void);
    void showSalary(void);
}
```

```
101 Student You;      103 You.displayInfo();          (gdb) p You
(gdb) n                  (gdb) n
102 You.getInfo();      Display Data
(gdb) n
Enter your name
Mary
Enter your gender
Female
Enter your age
12
103 You.displayInfo();  Name Mary
(gdb) n
104 You.getGrade();     Gender Female
(gdb) n
105 You.showGrade();    Age 12
Enter Student grade A
(gdb) n
Student Grade : 0
(gdb) n
(gdb) p You
$2 = {
<Common> = {
    name = "Mary",
    gender = "Female",
    age = 12
},
members of Student:
grade = 0
}
(gdb) ptype You
type = class Student : public
Common {
private:
    int grade;

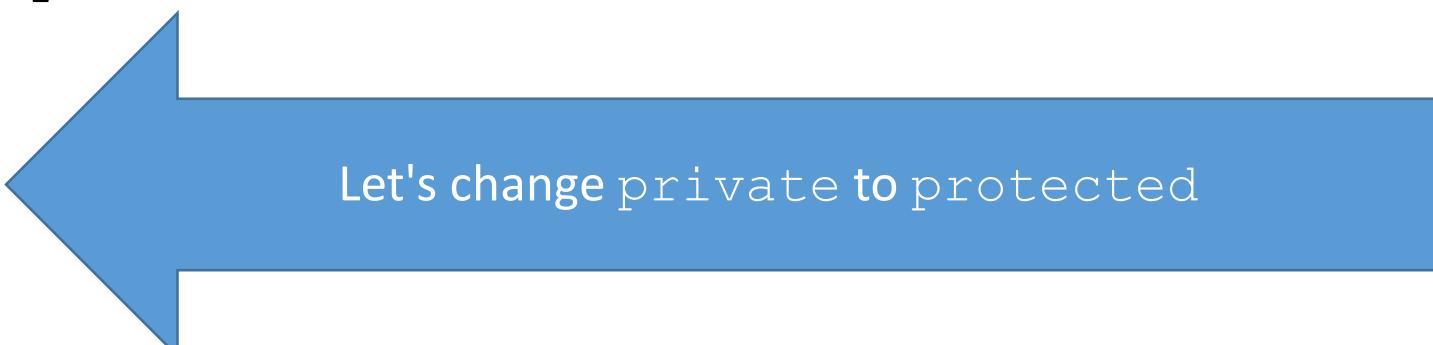
public:
    void getGrade(void);
    void showGrade(void);
}
```

protected Data

To enable class Rectangle to directly access Shape data members dim1 and dim2, we can declare those members as protected in the base class.

A base class's protected members can be accessed within the body of that base class, by members and friends of that base class, and by members and friends of any classes derived from that base class.

```
class Shape
{
public :
    Shape(std::string name="BaseShape") : ShapeName{name}
    {
        std::cout << "SHAPE!" << std::endl;
    }
    std::string getName()
    {
        return ShapeName;
    }
protected :
    float dim1;
    float dim2;
    std::string ShapeName;
};
```



Let's change **private** to **protected**

```
student@cse1325:/media/sf_VM$ make
g++ -c -g -std=c++11 ShapeInheritp.cpp -o
ShapeInheritp.o
g++ -g -std=c++11 ShapeInheritp.o -o ShapeInheritp.e
student@cse1325:/media/sf_VM$ ./ShapeInheritp.e
Let's make a Shape
SHAPE!
My name is Poly
Let's make a Circle
SHAPE!
CIRCLE!
My name is Hoop and my area is 28.2743
Let's make a Rectangle
SHAPE!
RECTANGLE!
My name is NotQuiteSquare and my area is 24
Let's make a Square
SHAPE!
RECTANGLE!
SQUARE!
My name is Quad and my area is 16
```

protected Data

- Rectangle and Circle inherit from class Shape.
- Objects of class Rectangle and Circle can access inherited data members that are declared protected in class Shape.
- Objects of a derived class also can access protected members in any of that derived class's *indirect base* classes.
- Square inherits from Rectangle which inherits from Shape and Square can access Shape's protected members.

Notes on Using protected Data

Inheriting protected data members slightly increases performance because we can directly access the members without incurring the overhead of calls to *set* or *get* member functions.

In most cases, it's better to use private data members to encourage proper software engineering and leave code optimization issues to the compiler. Your code will be easier to maintain, modify and debug.

protected data members are notated in class diagrams with the # mark

Notes on Using protected Data

Using protected data members creates two serious problems.

1. The derived-class object does not have to use a member function to set the value of the base class's protected data member.

Our private data member `color` is set to private and is kept from being set to the value of "green" by the setter for it.

```
void setColor(std::string Color)
{
    if (Color == "green")
    {
        throw std::invalid_argument("No green!!!");
    }
    color = Color;
}

private :
    std::string color;
```



Member function of Rectangle

```
Square S("Quad", 4);
```

```
S.setColor("green");
```

Square inherited color from Rectangle.

Let's make a Square
SHAPE!

RECTANGLE!

SQUARE!

terminate called after throwing an instance of
'std::invalid_argument'

what(): No green!!

Aborted (core dumped)

If we create a function in class Square to set the inherited private data member color...

```
void setSquareColor(std::string Color)
{
    color = Color;
}
```

```
student@cse1325:/media/sf_VM$ make
g++ -c -g -std=c++11 ShapeInheritp.cpp -o ShapeInheritp.o
ShapeInheritp.cpp: In member function 'void
Square::setSquareColor(std::__cxx11::string)':
ShapeInheritp.cpp:101:4: error: 'std::__cxx11::string Rectangle::color' is
private within this context
    color = Color;
    ^~~~~~
```

```
ShapeInheritp.cpp:86:15: note: declared private here
```

```
    std::string color;
        ^~~~~~
```

```
makefile:14: recipe for target 'ShapeInheritp.o' failed
```

```
make: *** [ShapeInheritp.o] Error 1
```

If we change our Rectangle private data member color to protected.

From

```
private :  
    std::string color;
```

To

```
protected :  
    std::string color;
```

```
129         S.setSquareColor("green");
(gdb) s
Square::setSquareColor (this=0x7fffffffdf0, Color="green") at
ShapeInheritp.cpp:101
101             color = Color;

(gdb) p S
$2 = {
<Rectangle> = {
    <Shape> = {
        dim1 = 4,
        dim2 = 4,
        ShapeName = "Quad"
    },
    members of Rectangle:
    color = "green"
},
members of Square:
location = "Line 68"
}
```

Inherited data member `color` was set to "green" even though `Rectangle` has a setter for `color` that explicitly does not allow the value of "green" for `color`.

Notes on Using protected Data

Using protected data members creates two serious problems.

2. Derived-class member functions are more likely to be written so that they depend on the base-class implementation.

Derived classes should depend only on the base-class services (i.e., non-private member functions) and not on the base-class implementation.

- With protected data members in the base class, if the base-class implementation changes, we may need to modify all derived classes of that base class.
- Such software is said to be **fragile** or **brittle**, because a small change in the base class can “break” derived-class implementation.

public, protected and private Inheritance

When deriving a class from a base class, the base class may be inherited through public, protected or private inheritance.

Use of protected and private inheritance is rare.

A base class's private members are *never* accessible directly from a derived class, but can be accessed through calls to the public and protected members of the base class.

public, protected and private Inheritance

If you do not choose an inheritance type, C++ defaults to private inheritance (just like members default to private access if you do not specify otherwise).

There are 9 combinations: 3 member access specifiers (public, private, and protected), and 3 inheritance types (public, private, and protected).

Base-class member-access specifier	Type of inheritance		
	public inheritance	protected inheritance	private inheritance
public	<p>public in derived class.</p> <p>Can be accessed directly by member functions, friend functions and nonmember functions.</p>	<p>protected in derived class.</p> <p>Can be accessed directly by member functions and friend functions.</p>	<p>private in derived class.</p> <p>Can be accessed directly by member functions and friend functions.</p>
protected	<p>protected in derived class.</p> <p>Can be accessed directly by member functions and friend functions.</p>	<p>protected in derived class.</p> <p>Can be accessed directly by member functions and friend functions.</p>	<p>private in derived class.</p> <p>Can be accessed directly by member functions and friend functions.</p>
private	<p>Hidden in derived class.</p> <p>Can be accessed by member functions and friend functions through public or protected member functions of the base class.</p>	<p>Hidden in derived class.</p> <p>Can be accessed by member functions and friend functions through public or protected member functions of the base class.</p>	<p>Hidden in derived class.</p> <p>Can be accessed by member functions and friend functions through public or protected member functions of the base class.</p>

public, protected and private Inheritance

So what's the difference between these?

When members are inherited, the access specifier for an inherited member may be changed (in the derived class only) depending on the type of inheritance used.

Members that were public or protected in the base class may change access specifiers in the derived class.

public, protected and private Inheritance

A class (and friends) can always access its own non-inherited members.

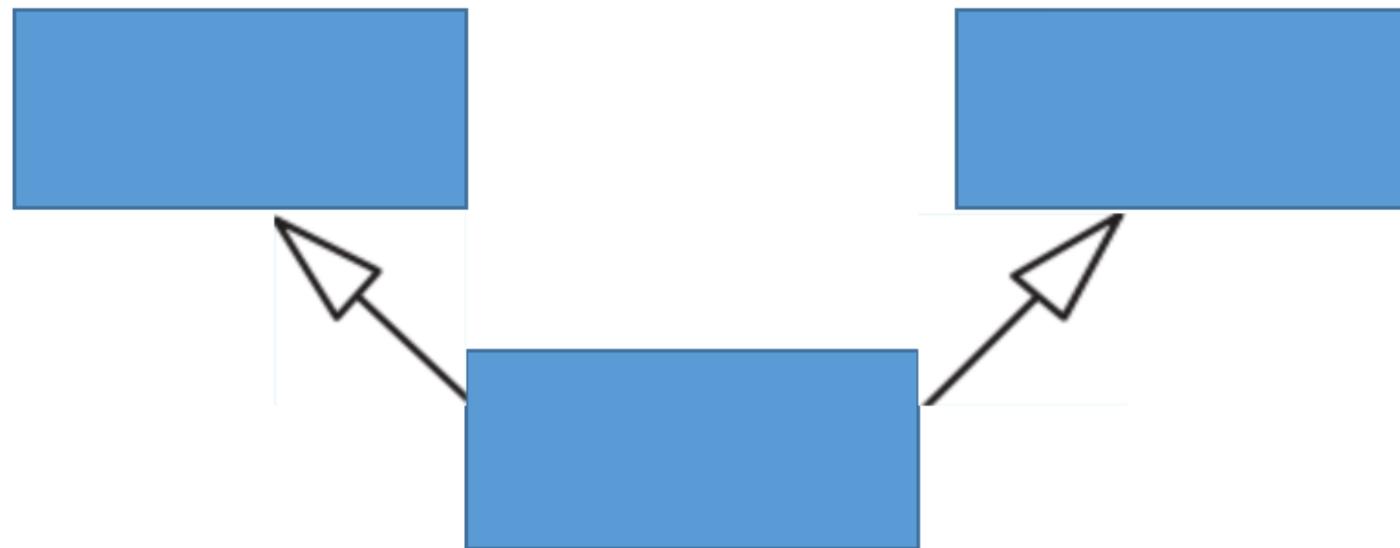
The access specifiers only affect whether outsiders and derived classes can access those members.

When derived classes inherit members, those members may change access specifiers in the derived class.

This does not affect the derived classes' own (non-inherited) members (which have their own access specifiers). It only affects whether outsiders and classes derived from the derived class can access those inherited members.

Multiple Inheritance

Multiple Inheritance occurs when a derived class inherits the members of two or more base classes.



Multiple Inheritance

Multiple Inheritance is a powerful capability that encourages interesting forms of software reuse.

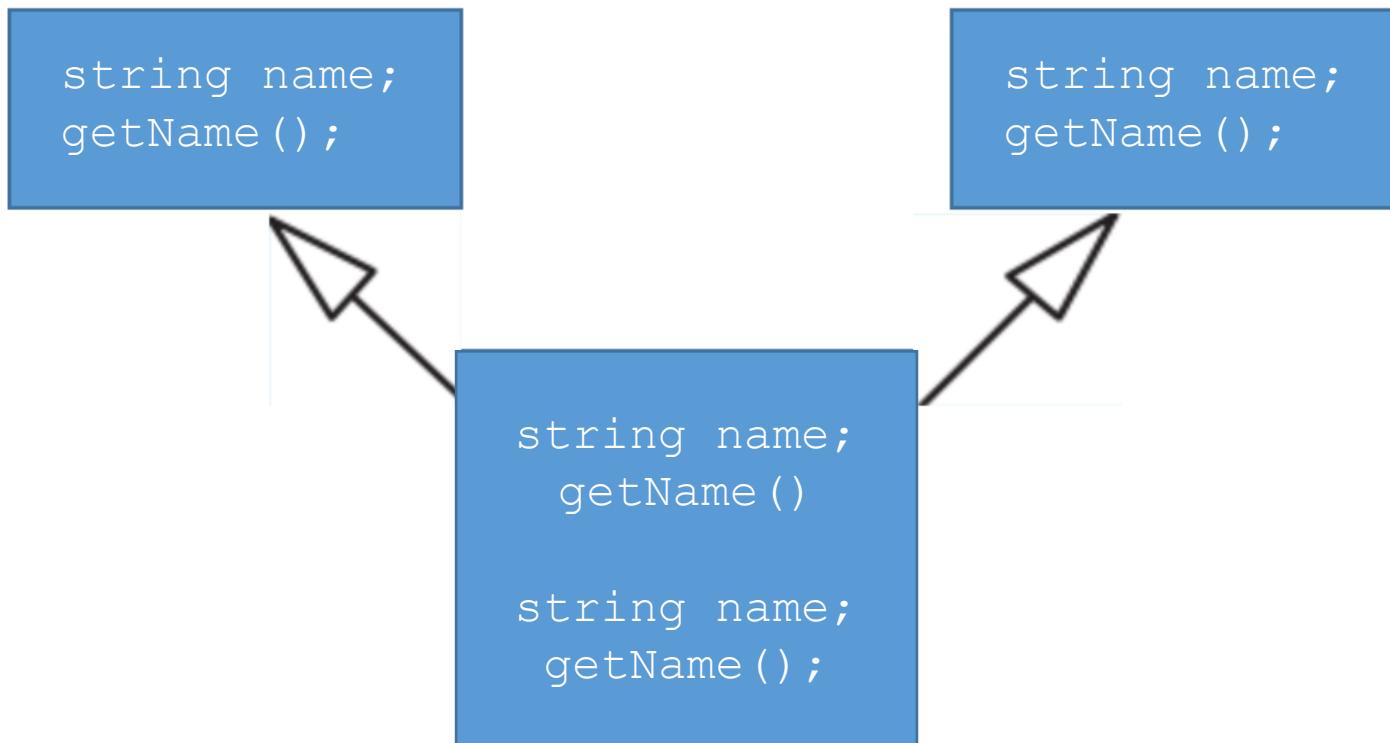
It can also cause a variety of ambiguity problems.

It is a difficult concept that should be used only by experienced programmers.

Some of the problems associated with multiple inheritance are so subtle that newer programming languages such as Java and C# do not enable a class to derive from more than one base class.

Multiple Inheritance

Multiple Inheritance can cause a situation where the derived class inherits data members or member functions from base classes that share names.



```
class A
{
public :
    A(string myname="")
    {
        name = myname;
    }
    string name;
};

class B
{
public :
    B(string myname="")
    {
        name = myname;
    }
    string name;
};
```

```
class AB : public A, public B
{
public :
    string ABVar;
};

int main(void)
{
    A a("IamA");
    B b("IamB");
    AB ab;

    return 0;
}
```

```
35          A a ("IamA");  
(gdb) n  
36          B b ("IamB");  
(gdb) n  
  
(gdb) p a  
$6 = {  
    name = "IamA"  
}  
(gdb) p b  
$7 = {  
    name = "IamB"  
}
```

Object a has been instantiated

Object b has been instantiated

```
37          AB ab;
(gdb) step
AB::AB (this=0x7fffffff090) at
miDemo.cpp:27
27  class AB : public A, public B
A::A (this=0x7fffffff090, myname="") at
miDemo.cpp:11
11
{
(gdb)
12  constructor for A      name = myname;
(gdb)
13 }
```

B::B (this=0x7fffffff0b0, myname="") at
miDemo.cpp:21

```
21
{
(gdb)
22  constructor for B      name = myname;
(gdb)
23 }
```

```
(gdb) p ab
$8 = {
<A> = {
    name = ""
},
<B> = {
    name = ""
},
members of
AB:
ABVar = ""
```

Multiple Inheritance

The base class constructors are called in the order of that the inheritance is specified – not in the order in which their constructors are mentioned.

Constructor for A was called before constructor for B because AB inherited A before B

```
class AB : public A, public B
```

If the base class constructors are not explicitly called in the member initializer list, their default constructors are called implicitly.

```
37          AB ab;  
(gdb) step  
AB::AB (this=0x7fffffff090) at miDemo.cpp:27  
27  class AB : public B, public A  
(gdb)  
B::B (this=0x7fffffff090, myname="") at miDemo.cpp:21  
21  {  
(gdb)  
22          name = myname;  
(gdb)  
23  }  
(gdb)  
A::A (this=0x7fffffff0b0, myname="") at miDemo.cpp:11  
11  {  
(gdb)  
12          name = myname;  
(gdb)  
13  }
```

Multiple Inheritance

```
int main(void)
{
    A a("IamA");
    B b("IamB");
    AB ab;

    cout << "Object A's name is " << a.name << endl;
    cout << "Object B's name is " << b.name << endl;

    return 0;
}
```

Object A's name is IamA
Object B's name is IamB

Multiple Inheritance

```
int main(void)
{
    A a ("IamA");
    B b ("IamB");
    AB ab;

    cout << "Object A's name is " << a.name << endl;
    cout << "Object B's name is " << b.name << endl;
cout << "Object AB's name is " << ab.name << endl;

    return 0;
}
```

So what happens when we try to print
AB's name?

```
(gdb) p ab
$8 = {
    <A> = {
        name = ""
    },
    <B> = {
        name = ""
    },
    members of
AB:
    ABVar = ""
}
```

```
miDemo.cpp: In function 'int main()':
miDemo.cpp:41:39: error: request for member 'name' is ambiguous
    cout << "Object AB's name is " << ab.name << endl;
                                         ^
miDemo.cpp:14:10: note: candidates are: std::__cxx11::string A::name
    string name;
               ^
miDemo.cpp:24:10: note:                     std::__cxx11::string B::name
    string name;
               ^
```

Multiple Inheritance

We run into the same issue if we try to create a constructor for AB.

```
class AB : public B, public A
{
public :
    AB(string myname="")
    {
        name = myname;
    }
    string ABVar;
};

miDemo.cpp: In constructor 'AB::AB(std::__cxx11::string)':
miDemo.cpp:32:4: error: reference to 'name' is ambiguous
    name = myname;
          ^
miDemo.cpp:14:10: note: candidates are: std::__cxx11::string A::name
    string name;
          ^
miDemo.cpp:24:10: note:
    string name;
          ^
std::__cxx11::string B::name
```

```
class AB : public B, public A
{
public :
    AB(string myname="")
    {
        A::name = "A"+myname;
        B::name = "B"+myname;
    }
    string ABVar;
};

int main(void)
{
    A a("IamA");
    B b("IamB");
    AB ab("IamAB");

    cout << "Object A's name is " << a.name << endl;
    cout << "Object B's name is " << b.name << endl;
    cout << "Object AB's A name is " << ab.A::name << endl;
    cout << "Object AB's B name is " << ab.B::name << endl;

    return 0;
}
```

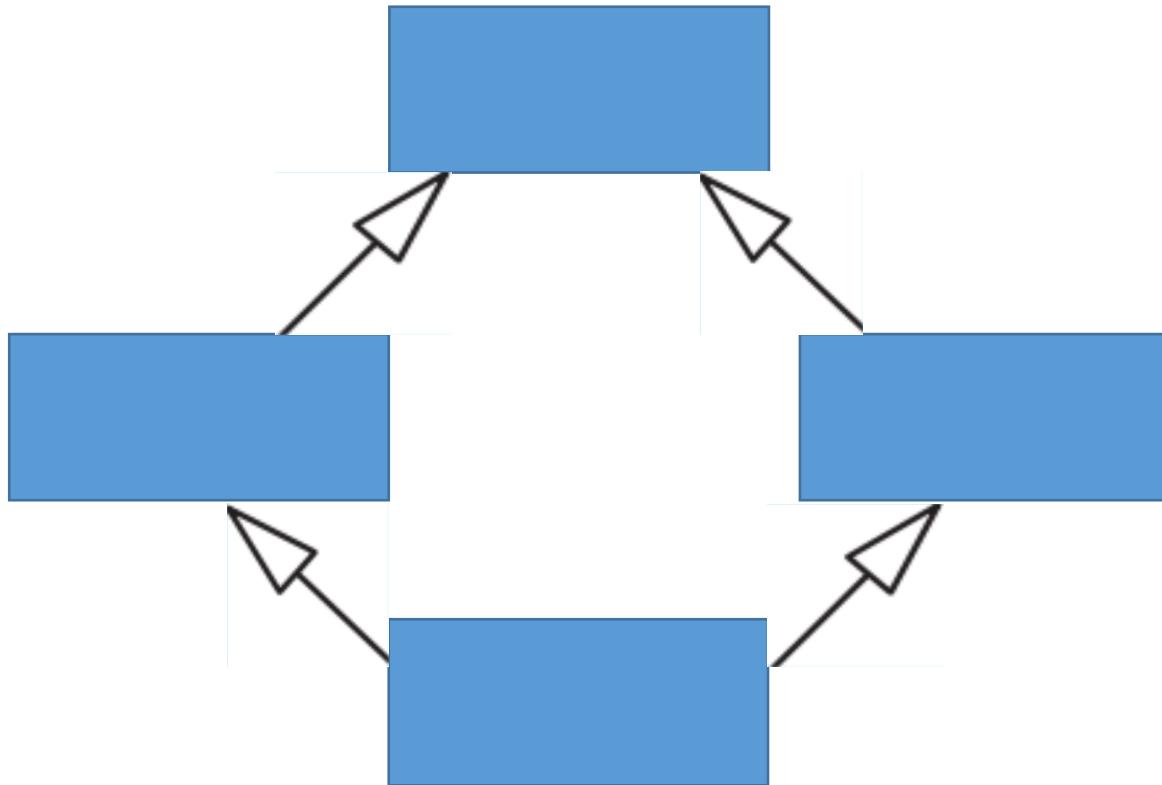
Object A's name is IamA
Object B's name is IamB
Object AB's A name is AIamAB
Object AB's B name is BIamAB

Multiple Inheritance

Diamond Inheritance

Diamond Inheritance occurs when a derived class inherits the members of two or more base classes who themselves inherited from a single base class.

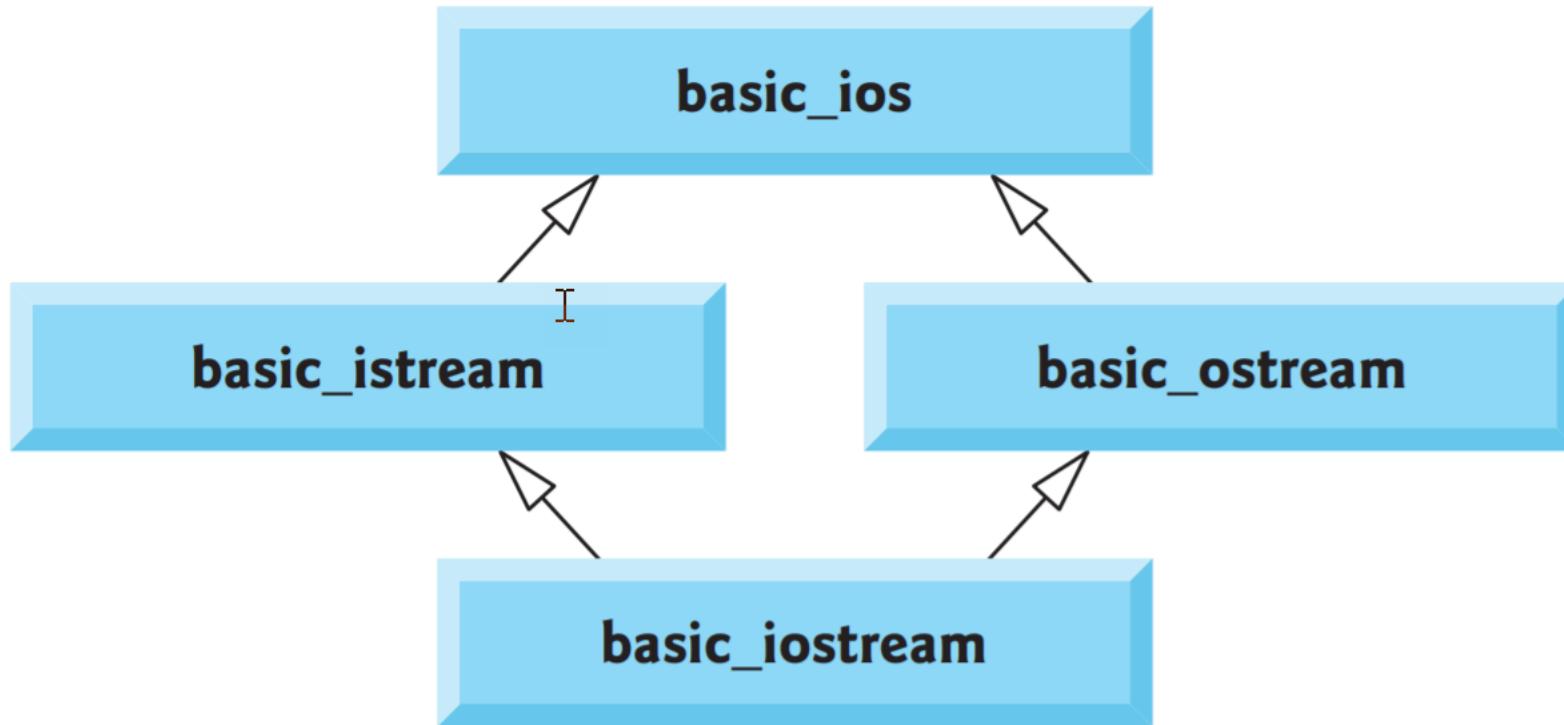
Not limited to 2 in the middle layer



Just need 1 on top and 1 on the bottom

Multiple Inheritance Diamond Inheritance

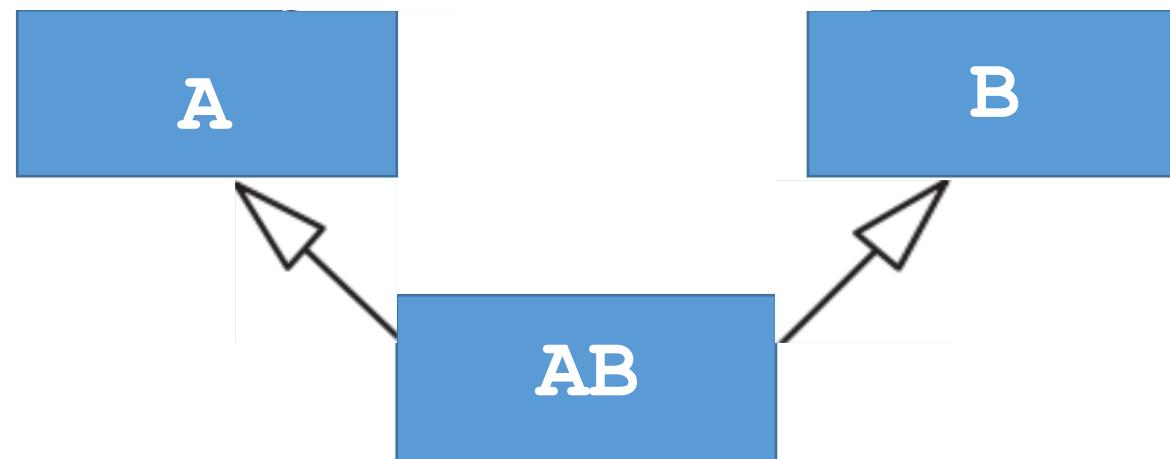
C++ Standard Library use diamond inheritance to form class `base_iostream`.



Multiple Inheritance

Diamond Inheritance

If we add a base class to our example – a new base class above our existing A and B in the hierarchy...



```
class O
{
public :
O(string myname="")
{
    name = myname;
}
string name;
};
```

```
class A : public O
{
public :
A(string myname="")
{
    name = myname;
}
string name;
};
```

```
class B : public O
{
public :
B(string myname="")
{
    name = myname;
}
string name;
};
```

```
class AB : public A, public B
{
public :
AB(string myname="")
{
    A::name = "A"+myname;
    B::name = "B"+myname;
}
string ABVar;
};
```

```
(gdb) p a
$4 = {
    <O> = {
        name = """
    },
members of A:
name = "IamA"
}
(gdb) p b
$5 = {
    <O> = {
        name = """
    },
members of B:
name = "IamB"
}
```

```
(gdb) p ab
$6 = {
    <A> = {
        <O> = {
            name = """
        },
members of A:
name = "AIamAB"
},
<B> = {
    <O> = {
        name = """
    },
members of B:
name = "BIamAB"
},
members of AB:
ABVar = """
}
```

```
(gdb) p ab
$6 = {
<A> = {
<O> = {
    name = ""
},
members of A:
name = "AIamAB"
},
<B> = {
<O> = {
    name = ""
},
members of B:
name = "BIamAB"
},
members of AB:
ABVar = ""}
```

Object ab has two copies of O

So which one will print?

```
cout << "Object A's name is " << a.name << endl;
cout << "Object B's name is " << b.name << endl;
cout << "Object AB's A name is " << ab.A::name << endl;
cout << "Object AB's B name is " << ab.B::name << endl;
cout << "Object AB's O name is " << ab.O::name << endl;
```

diamondDemo.cpp: In function 'int main()':

diamondDemo.cpp:60:44: error: 'O' is an ambiguous base of 'AB'

```
cout << "Object AB's O name is " << ab.O::name << endl;
```

^

```
(gdb) p ab
$6 = {
    <A> = {
        <O> = {
            name = ""
        },
        members of A:
        name = "AIamAB"
    },
    <B> = {
        <O> = {
            name = ""
        },
        members of B:
        name = "BIamAB"
    },
    members of AB:
    ABVar = ""
}

cout << "Object AB's O name is " << ab.O::name << endl;
diamondDemo.cpp: In function 'int main()':
diamondDemo.cpp:60:44: error: 'O' is an ambiguous base of 'AB'
    cout << "Object AB's O name is " << ab.O::name << endl;
                                            ^
cout << "Object AB's O name is " << ab.A::O::name << endl;

student@cse1325:/media/sf_VM@ make
g++ -c -g -std=c++11 diamondDemo.cpp -o diamondDemo.o
diamondDemo.cpp: In function 'int main()':
diamondDemo.cpp:58:47: error: 'O' is an ambiguous base
of 'AB'
    cout << "Object AB's O name is " << ab.A::O::name <<
endl;
                                         ^
makefile:14: recipe for target 'diamondDemo.o' failed
make: *** [diamondDemo.o] Error 1
```

Multiple Inheritance

Diamond Inheritance

How to resolve the ambiguity

```
class A : public O class B : public O
{
public :
A(string myname="")
{
    name = myname;
}
string name;
};

public :
B(string myname="")
{
    name = myname;
}
string name;
};
```

```
(gdb) p a
$1 = {
    <O> = {
        name = ""
    },
members of A:
    _vptr.A = 0x401fc0
<VTT for A>,
    name = "IamA"
}
(gdb) p b
$2 = {
    <O> = {
        name = ""
    },
members of B:
    _vptr.B = 0x401fa0
<VTT for B>,
    name = "IamB"
}
```

```
(gdb) p ab
$3 = {
    <A> = {
        <O> = {
            name = ""
        },
members of A:
    _vptr.A = 0x401f20 <vtable for AB+24>,
        name = "AIamAB"
    },
    <B> = {
        members of B:
            _vptr.B = 0x401f38 <VTT for AB>,
                name = "BIamAB"
    },
members of AB:
    ABVar = ""
}
```

Only one copy of O in object ab now.

Object A's name is IamA
Object B's name is IamB
Object AB's A name is AIamAB
Object AB's B name is BIamAB
Object AB's O name is

CSE 1325

Week of 11/23/2020

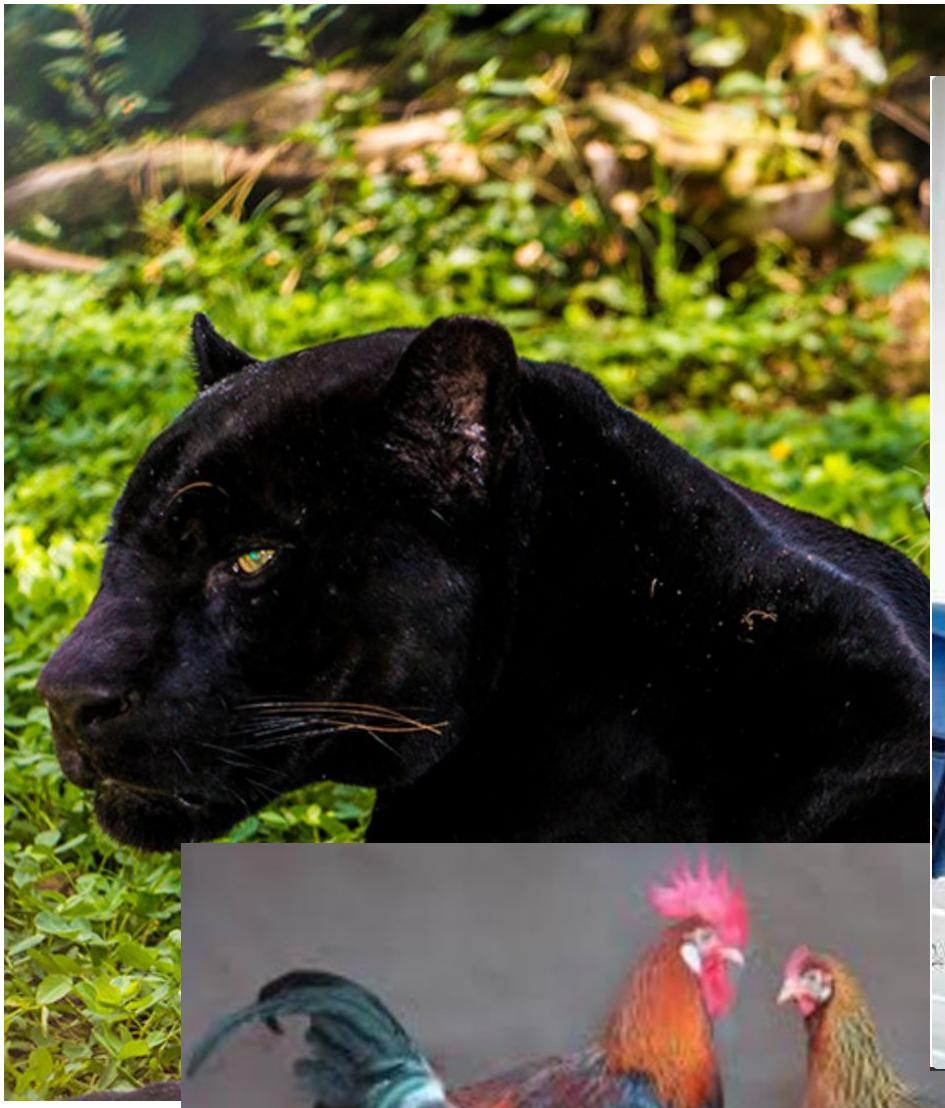
Instructor : Donna French

Polymorphism

Polymorphism occurs in biology.

Polymorphism in biology and zoology is the occurrence of two or more clearly different morphs or forms, also referred to as alternative phenotypes, in the population of a species.

To be classified as such, morphs must occupy the same habitat at the same time and belong to a panmictic population



Polymorphism

The word **polymorphism** means having many forms.

Typically, **polymorphism** occurs when there is a hierarchy of classes and they are related by inheritance.

C++ polymorphism means that a call to a member function will cause a different function to be executed depending on the type of object that invokes the function.

```
class Shape
{
public :
    Shape(std::string name="BaseShape") : ShapeName{name}
    {
        std::cout << "SHAPE!" << std::endl;
    }
    std::string getName()
    {
        return ShapeName;
    }

    float dim1;
    float dim2;
    std::string ShapeName;
};
```

```
class Rectangle : public Shape
{
public:
    Rectangle(std::string name, float height=0, float width=0)
        : Shape(name)
    {
        dim1 = height;
        dim2 = width;
    }

    float getarea()
    {
        return dim1 * dim2;
    }
};
```

```
class Square : public Rectangle
{
public:
    Square(std::string name, float size)
        : Rectangle(name, size)
    {
        dim1 = size;
        dim2 = size;
    }

private:
    std::string location{"Line 68"};
};
```

Seems like it would be a good idea for `getarea()` to live in the base class and be inherited?

```
class Circle : public Shape
{
public:
    Circle(std::string name, float radius=0)
        : Shape()
    {
        dim1 = dim2 = radius;
    }

    float getarea()
    {
        return dim1 * dim2 * M_PI;
    }

private :
    std::string color;
};
```

```
class Shape
{
public :
    Shape(std::string name="BaseShape") : ShapeName{ name }
    {
        std::cout << "SHAPE!" << std::endl;
    }
    std::string getName()
    {
        return ShapeName;
    }
    float getarea()
    {
    }

    float dim1;
    float dim2;
    std::string ShapeName;
};
```

We want all of our derived classes to have/use the version of `getarea()` that does not take in any parameters and returns a `float`.
What would I put in here? "Shape" doesn't have an area!?

We don't want to allow a derived class to create a `getarea()` function that requires a parameter or returns an `int` for example.

Virtual Function

- Declaring a base class function as `virtual` allows derived classes to override that function's behavior which enables polymorphic behavior.
- An overridden function in a derived class has the same signature and return type (prototype) as the function it overrides in its base class.
- With `virtual` functions, the type of the object determines which version of a `virtual` function to invoke.
- If a base class function is not `virtual`, then the derived class could redefine the function.

```
class Shape
{
public :
    Shape(std::string name="BaseShape") : ShapeName{name}
    {
        std::cout << "SHAPE!" << std::endl;
    }
    std::string getName()
    {
        return ShapeName;
    }
virtual float getarea()
{
}
```

float dim1;

float dim2;

std::string ShapeName;

} ;



getarea() is now a virtual function

```
class Circle : public Shape  
{  
public:  
    Circle(float radius=0)  
    {  
        dim1 = dim2 = radius;  
    }  
  
    float getarea()  
    {  
        return dim1 * dim2 * M_PI;  
    }  
};
```

Shape declared `getarea()` to be virtual which allows `Circle` to override the inherited behavior.

Circle inherits publicly from Shape
float dim1
float dim2

dim1 and dim2 are public members of Shape; therefore, Circle can set them directly

`M_PI` is define in `<cmath>`

```
class Circle : public Shape
{
public:
    Circle(float radius=0)
    {
        dim1 = dim2 = radius;
    }

    float getarea()
    {
        return dim1 * dim2 * M_PI;
    }
};
```

Shape.cpp:26:13: error: conflicting return type specified for 'virtual int Circle::getarea()'

 int getarea()

 ^

Shape.cpp:15:17: error: overriding 'virtual float Shape::getarea()'

 virtual float getarea(void) {};

Virtual Function

- Once a function is declared `virtual`, it remains `virtual` all the way down the inheritance hierarchy from that point, even if that function is not explicitly declared `virtual` when a derived class overrides it.
- When a derived class chooses not to override a `virtual` function from its base class, the derived class simply inherits its base class's `virtual` function implementation.

```
class Rectangle : public Shape  
{  
public:  
    Rectangle(float height=0, float width=0)  
    {  
        dim1 = height;  
        dim2 = width;  
    }  
  
    float getarea ()  
    {  
        return dim1 * dim2;  
    }  
};
```

Rectangle inherits publicly from Shape
float dim1
float dim2

Constructor with default parameters

dim1 and dim2 are public members of Shape;
therefore, Rectangle can set them directly

Shape declared getarea () to be virtual which allows Rectangle to
override the inherited behavior.

Note that this version of getarea () is different from Circle's version.

```
class Square : public Rectangle
{
public:
    Square(float size)
    {
        dim1 = size;
        dim2 = size;
    }
};
```

*Square inherits publicly from Rectangle
which inherits publicly from Shape*

*float dim1
float dim2*

*dim1 and dim2 are public members of Shape;
therefore, Square can set them directly*

Square inherited Rectangle's version of `getarea()`. No need to write a new version of `getarea()` since the area of a square is calculated using the same formula as a rectangle.

```
int main (void)
{
    vector<Shape*>MyShapes;

    Circle C1(15.0);
    Rectangle R1(34.0, 2.0);
    Circle C2(2.3);
    Rectangle R2(5.61, 7.92);
    Square S1(3.33);

    MyShapes.push_back(&C1);
    MyShapes.push_back(&R1);
    MyShapes.push_back(&C2);
    MyShapes.push_back(&R2);
    MyShapes.push_back(&S1);

    for (auto it : MyShapes)
    {
        cout << "Area is " << it->getarea() << endl;
    }

    return 0;
}
```

MyShapes is a vector of pointers of class Shape

Instantiating Circle, Rectangle and Square objects.

Filling the vector with pointers to the objects

Use a range based for loop to display the vector contents

The elements of MyShapes are pointers; therefore, use pointer notation to call the member function getarea()

```
Circle C1(15.0);
Rectangle R1(34.0,2.0);
Circle C2(2.3);
Rectangle R2(5.61,7.92);
Square S1(3.33);
```

```
cout << "Area is " << it->getarea() << endl;
```

```
Area is 706.858
Area is 68
Area is 16.619
Area is 44.4312
Area is 11.0889
```

```
Circle C1(15.0);
Rectangle R1(34.0,2.0);
Circle C2(2.3);
Rectangle R2(5.61,7.92);
Square S1(3.33);

MyShapes.push_back(&C1);
MyShapes.push_back(&R1);
MyShapes.push_back(&C2);
MyShapes.push_back(&R2);
MyShapes.push_back(&S1);
```

```
(gdb) p MyShapes
$4 = std::vector of length 5, capacity 8 = {0x7fffffff090,
0x7fffffff0a0, 0x7fffffff0b0, 0x7fffffff0c0, 0x7fffffff0d0}
```

```
(gdb) p &C1
$7 = (Circle *) 0x7fffffff090
```

```
(gdb) p C1
$8 = {
<Shape> = {
    _vptr.Shape = 0x401eb8 <vtable for Circle+16>,
    dim1 = 15,
    dim2 = 15
}, <No data fields>}
```

```
76         for (auto it : MyShapes)

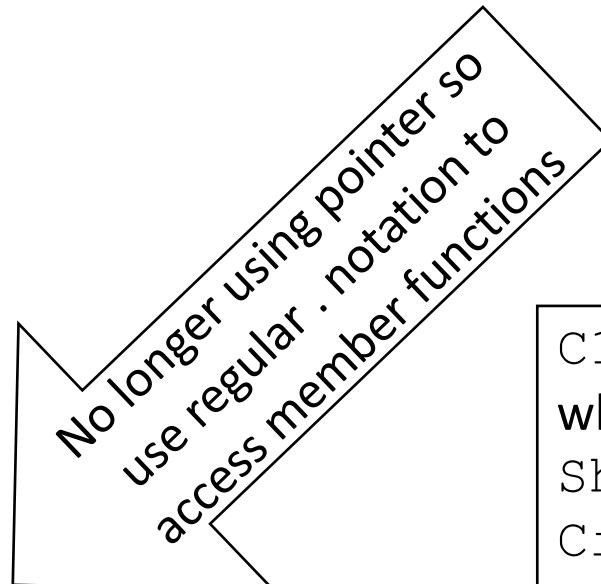
(gdb) p it
$10 = (Shape *) 0x7fffffff090
(gdb) p *it
$11 = {
    _vptr.Shape = 0x401eb8 <vtable for Circle+16>,
    dim1 = 15,
    dim2 = 15
}
```

```
(gdb) p &C1
$7 = (Circle *)
0x7fffffff090
(gdb) p C1
$8 = {
    <Shape> = {
        _vptr.Shape = 0x401eb8
    <vtable for Circle+16>,
        dim1 = 15,
        dim2 = 15
    }, <No data fields>}
```

```
78             cout << "Area is " << it->getarea() << endl;
(gdb) step
Circle::getarea(this=0x7fffffff090) at Shape.cpp:28
28             return dim1 * dim2 * M_PI;
(gdb)
29         }
(gdb)
Area is 706.858
```

What happens if we don't use pointers to the derived class instantiations?

```
vector<Shape>MyShapes;  
  
Circle C1(15.0);  
  
MyShapes.push_back(C1);  
  
for (auto it : MyShapes)  
{  
    cout << "Area is " << it.getarea() << endl;  
}
```



C1 is cast from Circle to Shape when it is pushed into a vector of type Shape. It then loses access to Circle's getarea() and tries to use Shape's getarea() which does not have a calculation in it which causes undefined behavior.

Area is 15

```
81  
(gdb) step  
Shape::getarea (this=0x7fffffff0d0) at Shape.cpp:15  
15  
(gdb)  
Area is 15
```

```
cout << "Area is " << it.getarea() << endl;  
  
virtual float getarea(void) {};
```

How do we add a private data member to Shape? dim1 and dim2 were both public.

Let's give each of our Shape objects a name by constructing each object with a name that matches the instantiation name. For example, object C1's name will be C1.

```
vector<Shape*>MyShapes;
```

No change

```
Circle C1("C1", 15.0);
```

```
Rectangle R1("R1", 34.0, 2.0);
```

```
Circle C2("C2", 2.3);
```

```
Rectangle R2("R2", 5.61, 7.92);
```

```
Square S1("S1", 3.33);
```

Passing a name in the constructor

```
MyShapes.push_back(&C1);
```

```
MyShapes.push_back(&R1);
```

```
MyShapes.push_back(&C2);
```

```
MyShapes.push_back(&R2);
```

```
MyShapes.push_back(&S1);
```

No change

```
class Shape
{
public :
    float dim1;
    float dim2;
    virtual float getarea(void) { } ;

private :
    string name;
};
```

```
Circle C1(15.0);

class Circle : public Shape
{
public:
    Circle(float radius=0)
    {
        dim1 = dim2 = radius;
    }

    float getarea()
    {
        return dim1 * dim2 * M_PI;
    }
};
```

```
Circle C1("C1", 15.0);

class Circle : public Shape
{
public:
    Circle(string shapeName="",
           float radius=0)
    {
        dim1 = dim2 = radius;
        name = shapeName;
    }

    float getarea()
    {
        return dim1 * dim2 * M_PI;
    }
};
```

```
Circle C1("C1", 15.0);

class Circle : public Shape
{
public:
    Circle(string shapeName="",
           float radius=0)
    {
        dim1 = dim2 = radius;
        name = shapeName;
    }

    float getarea()
    {
        return dim1 * dim2 * M_PI;
    }
};
```

C++ rigidly enforces restrictions on accessing private data members, so that *even a derived class* (which is intimately related to its base class) cannot access the base class's private data.

```
Shape1.cpp: In constructor 'Circle::Circle(std::__cxx11::string, float)':
Shape1.cpp:26:10: error: 'std::__cxx11::string Shape::name' is private
    string name;
               ^
Shape1.cpp:36:4: error: within this context
    name = shapeName;
               ^
```

We could make our private data member name protected instead of private.

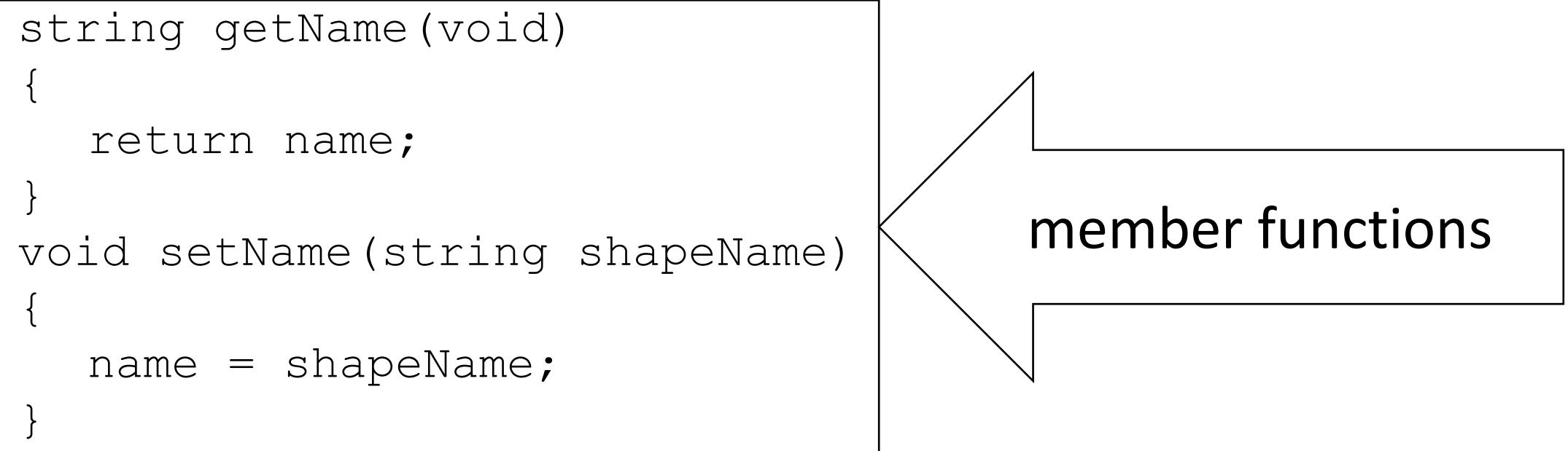
```
class Shape
{
    public :
        float dim1;
        float dim2;
        virtual float getarea(void) { } ;

    protected :
        string name;
};
```

This will compile just fine.

While this does allow direct access, using protected access is not always an option.

```
class Shape
{
public :
    float dim1;
    float dim2;
    virtual float getarea(void) { } ;
    string getName(void)
    {
        return name;
    }
    void setName(string shapeName)
    {
        name = shapeName;
    }
}
```



member functions

```
private :
    string name;
};
```

```
Circle C1("C1", 15.0);

class Circle : public Shape
{
public:
    Circle(string shapeName="", float radius=0)
    {
        dim1 = dim2 = radius;
        setName(shapeName);
    }

    float getarea()
    {
        return dim1 * dim2 * M_PI;
    }
};
```

C1's area is 706.858

```
class Rectangle : public Shape
{
public:
    Rectangle(string shapeName="", float height=0, float width=0)
    {
        dim1 = height;
        dim2 = width;
        setName(shapeName);
    }

    float getarea()
    {
        return dim1 * dim2;
    }
}
```

```
class Square : public Rectangle
{
public:
    Square(string shapeName, float size)
    {
        dim1 = size;
        dim2 = size;
        setName(shapeName);
    }

};
```

```
for (auto it : MyShapes)
{
    cout << it->getName() << "'s area is " << it->getarea() << endl;
}
```

C1's area is 706.858

R1's area is 68

C2's area is 16.619

R2's area is 44.4312

S1's area is 11.0889

```
92         cout << it->getName() << "'s area is " << it->getarea() << endl;
(gdb) p it
$4 = (Shape *) 0x7fffffff000
(gdb) p *it
$5 = {_vptr.Shape = 0x402940 <vtable for Circle+16>, dim1 = 15, dim2 = 15,
name = "C1"}
(gdb) step

Circle::getarea (this=0x7fffffff000) at Shape1.cpp:40
40             return dim1 * dim2 * M_PI;
(gdb)
41
(gdb)
Shape::getName[abi:cxx11] () (this=0x7fffffff000) at Shape1.cpp:18
18             return name;
(gdb)
19
(gdb)
```

C1's area is 706.858

shape1Demo.cpp

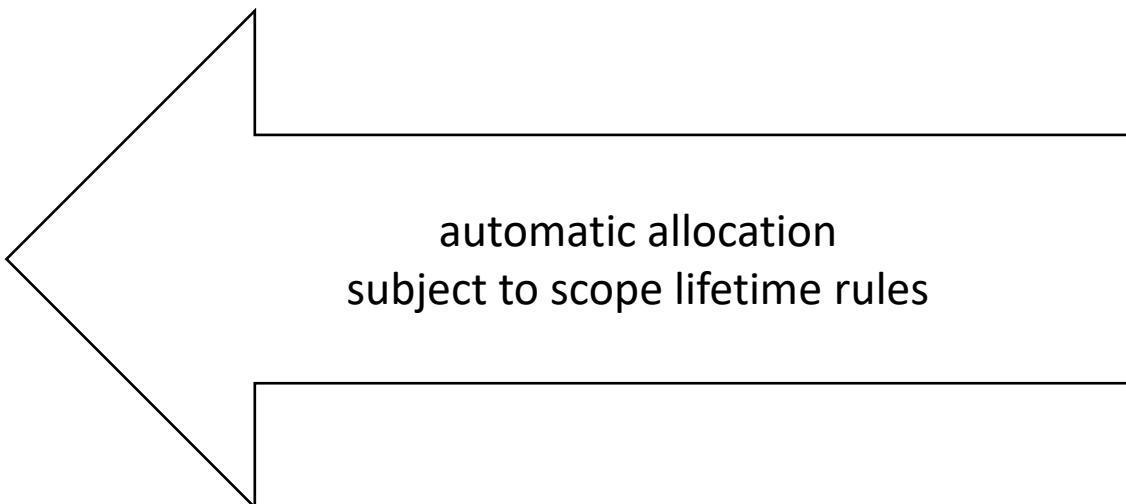
```
vector<Shape*>MyShapes;
```

shape2Demo.cpp

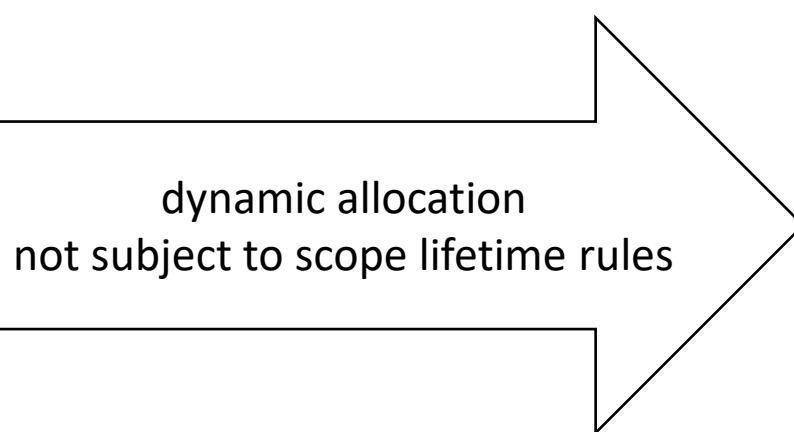
```
Circle C1("C1", 15.0);  
Rectangle R1("R1", 34.0, 2.0);  
Circle C2("C2", 2.3);  
Rectangle R2("R2", 5.61, 7.92);
```

```
Square S1("S1", 3.33);
```

```
MyShapes.push_back(&C1);  
MyShapes.push_back(&R1);  
MyShapes.push_back(&C2);  
MyShapes.push_back(&R2);  
MyShapes.push_back(&S1);
```



automatic allocation
subject to scope lifetime rules



dynamic allocation
not subject to scope lifetime rules

```
vector<Shape*>MyShapes;
```

```
MyShapes.push_back(new Circle("C1", 15.0));  
MyShapes.push_back(new Rectangle("R1", 34.0, 2.0));  
MyShapes.push_back(new Circle("C2", 2.3));  
MyShapes.push_back(new Rectangle("R2", 5.61, 7.92));  
MyShapes.push_back(new Square("S1", 3.33));
```

```
class Shape
{
public :
    Shape(string shapeName="")
    {
        name = shapeName;
    }
    float dim1;
    float dim2;
    virtual float getarea() { };
    string getName(void)
    {
        return name;
    }
    void setName(string shapeName)
    {
        name = shapeName;
    }
void Hello(void)
{
    cout << "Hi! My class is Shape." << endl;
}
private :
    string name;
};
```

Added constructor with default parameters

Added member function

```
class Square : public Rectangle
{
public:
    Square(string shapeName, float size) : Rectangle()
    {
        dim1 = size;
        dim2 = size;
        setName(shapeName);
    }

void Hello(void)
{
    cout << "Hi! My class is Square." << endl;
}
};
```

```
void Hello(void)
{
    cout << "Hi! My class is Rectangle." << endl;
```

```
void Hello(void)
{
    cout << "Hi! My class is Circle." << endl;
}
```

```
vector<Shape*>MyShapes;

MyShapes.push_back(new Shape("Shape1"));
MyShapes.push_back(new Circle("C1", 15.0));
MyShapes.push_back(new Rectangle("R1", 34.0, 2.0));
MyShapes.push_back(new Circle("C2", 2.3));
MyShapes.push_back(new Rectangle("R2", 5.61, 7.92));
MyShapes.push_back(new Square("S1", 3.33));

for (auto it : MyShapes)
{
    it->Hello();
    cout << it->getName() << "'s area is " << it->getarea() << endl;
}
```

Create a Shape

Call Hello() for each object in MyShapes

```
Hi! My class is Shape.  
Shape1's area is 0  
Hi! My class is Shape.  
C1's area is 706.858  
Hi! My class is Shape.  
R1's area is 68  
Hi! My class is Shape.  
C2's area is 16.619  
Hi! My class is Shape.  
R2's area is 44.4312  
Hi! My class is Shape.  
S1's area is 11.0889  
Hi! My class is Shape.  
C3's area is 706.858
```

The version of Hello() in Shape() was encountered first; therefore, was the one that was executed.

The versions of Hello() in the derived classes were unreachable and were not executed.

```
class Shape
{
public :
    Shape(string shapeName="")
    {
        name = shapeName;
    }
    float dim1;
    float dim2;
    virtual float getarea() {};
    string getName(void)
    {
        return name;
    }
    void setName(string shapeName)
    {
        name = shapeName;
    }
virtual void Hello(void)
{
    cout << "Hi! My class is Shape." << endl;
}
private :
    string name;
};
```

Hi! My class is Shape.
Shape1's area is 0
Hi! My class is Circle.
C1's area is 706.858
Hi! My class is Rectangle.
R1's area is 68
Hi! My class is Circle.
C2's area is 16.619
Hi! My class is Rectangle.
R2's area is 44.4312
Hi! My class is Square.
S1's area is 11.0889

Added virtual

```
Hi! My class is Shape.  
Shape1's area is 0  
Hi! My class is Circle.  
C1's area is 706.858  
Hi! My class is Rectangle.  
R1's area is 68  
Hi! My class is Circle.  
C2's area is 16.619  
Hi! My class is Rectangle.  
R2's area is 44.4312  
Hi! My class is Square.  
S1's area is 11.0889
```

No changes were made to the derived class's Hello().

Now that the base class version of Hello() is virtual, the derived classes use their own versions.

Rectangle's Hello() is virtual because the base class's version is virtual; therefore, Square uses its version.

Once declared virtual, any derived class version will be virtual.

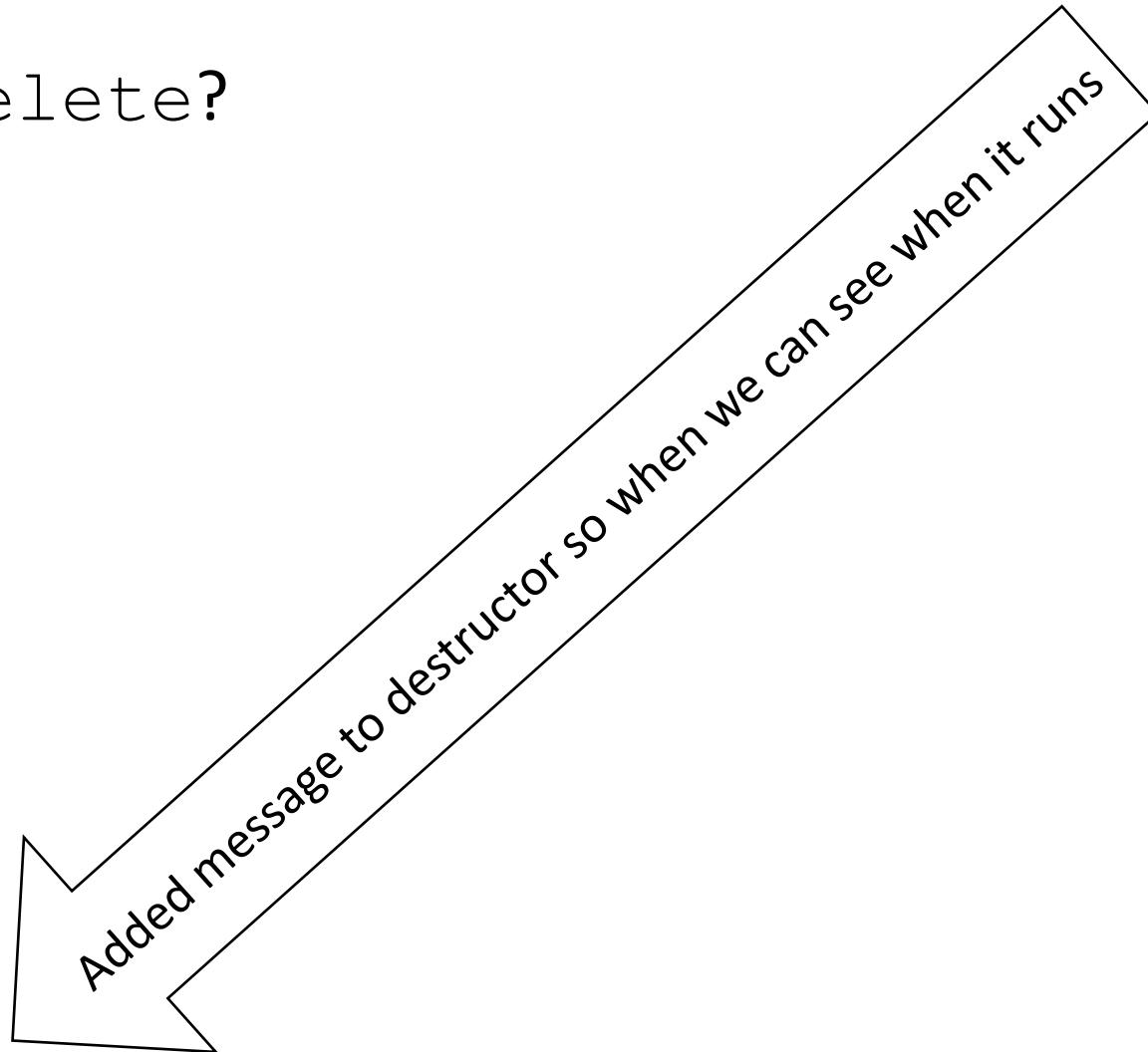
Base class can still use the virtual function.

Derived classes do not inherit destructors and do not have default destructor.

So what happens when we use delete?

```
for (auto it : MyShapes)
{
    delete it;
}

class Shape
{
public :
    Shape(string shapeName="")
    {
        name = shapeName;
    }
    ~Shape ()
    {
        cout << "Destroying Shape" << endl;
    }
}
```



Hi! My class is Shape.

Shape1's area is 0

Hi! My class is Circle.

C1's area is 706.858

Hi! My class is Rectangle.

R1's area is 68

Hi! My class is Circle.

C2's area is 16.619

Hi! My class is Rectangle.

R2's area is 44.4312

Hi! My class is Square.

S1's area is 11.0889

Destroying Shape

Destroying Shape

Destroying Shape

Destroying Shape

Destroying Shape

Destroying Shape

Destructor in Shape is called.

Resources specific to the derived class would not be released

So we need to make the destructor virtual

```
class Shape
{
public :
    Shape(string shapeName="")
    {
        name = shapeName;
    }
    virtual ~Shape()
    {
        cout << "Destroying Shape" << endl;
    }
}
```

Hi! My class is Shape.

Shape1's area is 0

Hi! My class is Circle.

C1's area is 706.858

Hi! My class is Rectangle.

R1's area is 68

Hi! My class is Circle.

C2's area is 16.619

Hi! My class is Rectangle.

R2's area is 44.4312

Hi! My class is Square.

S1's area is 11.0889

Destroying Shape

Destroying Shape

Destroying Shape

Destroying Shape

Destroying Shape

Destroying Shape

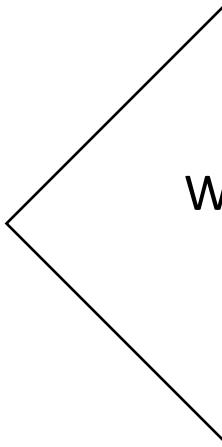
Destructor in Shape is virtual but still called because we did not
create override versions in our derived classes.

```
~Circle()
{
    cout << "Destroying Circle" << endl;
}

~Rectangle()
{
    cout << "Destroying Rectangle" << endl;
}

~Square()
{
    cout << "Destroying Square" << endl;
}
```

```
Hi! My class is Shape.  
Shape1's area is 0  
Hi! My class is Circle.  
C1's area is 706.858  
Hi! My class is Rectangle.  
R1's area is 68  
Hi! My class is Circle.  
C2's area is 16.619  
Hi! My class is Rectangle.  
R2's area is 44.4312  
Hi! My class is Square.  
S1's area is 11.0889  
Destroying Shape  
Destroying Circle  
Destroying Shape  
Destroying Rectangle  
Destroying Shape  
Destroying Circle  
Destroying Shape  
Destroying Rectangle  
Destroying Shape  
Destroying Square  
Destroying Rectangle  
Destroying Shape
```



When a derived class object is destroyed, the base class part of the derived class is also destroyed

What happens if Circle is not a derived class of Shape?

```
class Circle
{
public
    Circle (float xradius=0)
    {
        radius = xradius;
    }

    float getarea()
    {
        return radius * radius * M_PI;
    }
};
```

```
vector<Shape*>MyShapes;
```

```
MyShapes.push_back(new Circle("C1", 15.0));
```

Shape4.cpp: In function 'int main()':

Shape4.cpp:67:37: error: no matching function for call to 'std::vector<Shape*>::push_back(Circle*)'

```
MyShapes.push_back(new Circle(15.0));
```

How to add a new class/shape

```
class Triangle : public Shape
{
public:
    Triangle(string shapeName="", float base=0, float height=0)
    {
        dim1 = base;
        dim2 = height;
        setName(shapeName);
    }
    ~Triangle()
    {
        cout << "Destroying Triangle" << endl;
    }
    float getarea()
    {
        return dim1 * dim2 * 0.5;
    }
    void Hello(void)
    {
        cout << "Hi! My class is Triangle." << endl;
    }
};
```

```
vector<Shape*>MyShapes;

MyShapes.push_back(new Shape("Shape1"));
MyShapes.push_back(new Circle("C1", 15.0));
MyShapes.push_back(new Rectangle("R1", 34.0, 2.0));
MyShapes.push_back(new Circle("C2", 2.3));
MyShapes.push_back(new Rectangle("R2", 5.61, 7.92));
MyShapes.push_back(new Square("S1", 3.33));
MyShapes.push_back(new Triangle("T1", 3.0, 6.0));

for (auto it : MyShapes)
{
    it->Hello();
    cout << it->getName() << "'s area is " << it->getarea() << endl;
}

for (auto it : MyShapes)
{
    delete it;
}
```

```
Hi! My class is Shape.  
Shape1's area is 0  
Hi! My class is Circle.  
C1's area is 706.858  
Hi! My class is Rectangle.  
R1's area is 68  
Hi! My class is Circle.  
C2's area is 16.619  
Hi! My class is Rectangle.  
R2's area is 44.4312  
Hi! My class is Square.  
S1's area is 11.0889  
Hi! My class is Triangle.  
T1's area is 9
```

```
Destroying Shape  
Destroying Circle  
Destroying Shape  
Destroying Rectangle  
Destroying Shape  
Destroying Circle  
Destroying Shape  
Destroying Rectangle  
Destroying Shape  
Destroying Square  
Destroying Rectangle  
Destroying Shape  
Destroying Triangle  
Destroying Shape
```

Abstract Classes

An abstract class is a class from which you never intend to instantiate any objects.

Because these classes normally are used as base classes in inheritance hierarchies, they are referred to as abstract base classes.

These classes cannot be used to instantiate object.

Abstract classes are typically incomplete.

Abstract Classes

An abstract class is a base class from which other classes can inherit.

Classes that can be used to instantiate objects are called concrete classes.

Concrete classes define or inherit implementations for every member function they declare.

Abstract Classes

Shape

abstract class

Circle, Rectangle, Square, Triangle

concrete class

Abstract base classes are too generic to define real objects.

If I asked you to calculate the area of a Shape, how would you do that?

Abstract Classes

A class is made abstract by declaring one or more of its virtual functions to be "pure".

A pure virtual function is specified by placing "= 0" in its declaration.

```
virtual float getarea() = 0;
```

The "= 0" is a pure specifier.

Forces derived classes to override the virtual function. Non pure virtual functions do not force derived classes to implement overrides.

Abstract Classes

virtual function

has an implementation and gives the derived class the option of overriding the function

pure virtual function

does not have an implementation and requires the derived class to override the function

Abstract Classes

When should a function be set as pure virtual?

When the function implementation does not make sense for the base class and you want to force all concrete derived classes to implement the function.

If a derived class does not override the function, then the derived class remains abstract and cannot be concrete. Objects cannot be instantiated from that derived class (compiler errors).

```
class Triangle : public Shape
{
public:
    Triangle(string shapeName="", float base=0, float height=0)
    {
        dim1 = base;
        dim2 = height;
        setName(shapeName);
    }
    ~Triangle()
    {
        cout << "Destroying Triangle" << endl;
    }
    float getarea()
    {
        return dim1 * dim2 * 0.5;
    }
    void Hello(void)
    {
        cout << "Hi! My class is Triangle." << endl;
    }
};
```

```
class Shape
{
public :
    virtual float getarea() { };
};
```

getarea() in Shape is virtual
and Triangle is overriding it

Hi! My class is Triangle.
T1's area is 9
Destroying Triangle
Destroying Shape

```
class Triangle : public Shape
{
public:
    Triangle(string shapeName="", float base=0, float height=0)
    {
        dim1 = base;
        dim2 = height;
        setName(shapeName);
    }
    ~Triangle()
    {
        cout << "Destroying Triangle" << endl;
    }
    void Hello(void)
    {
        cout << "Hi! My class is Triangle." << endl;
    }
};
```

```
class Shape
{
public :
    virtual float getarea() = 0;
};
```

removed the overridden
version of getarea ()

pure is not a keyword
use "= 0"

Shape4.cpp: In function 'int main()':

Shape4.cpp:144:48: error: invalid new-expression of abstract class type 'Triangle'
e'

MyShapes.push_back(new Triangle("T1", 3.0, 6.0));

Shape4.cpp:110:7: note: because the following virtual functions are pure within 'Triangle':

class Triangle : public Shape

Shape4.cpp:21:17: note: virtual float Shape::getarea()

virtual float getarea() = 0;

```
class Triangle : public Shape
{
public:
    Triangle(string shapeName="", float base=0, float height=0)
    {
        dim1 = base;
        dim2 = height;
        setName(shapeName);
    }
    ~Triangle()
    {
        cout << "Destroying Triangle" << endl;
    }
    float getarea()
    {
        return dim1 * dim2 * 0.5;
    }
    void hello(void)
    {
        cout << "Hi! My class is Triangle." << endl;
    }
};
```

```
class Shape
{
public :
    virtual float getarea() = 0;
};
```

Hi! My class is Triangle.
T1's area is 9
Destroying Triangle
Destroying Shape

With `getarea()` now set as a pure virtual function

```
class Shape
{
public :
    virtual float getarea() = 0;
```

We can no longer instantiate objects from `Shape`. `Shape` is now an abstract class and cannot be instantiated.

```
MyShapes.push_back(new Shape("Shape1"));
```

```
Shape4.cpp: In function 'int main()':
Shape4.cpp:138:39: error: invalid new-expression of abstract class type 'Shape'
    MyShapes.push_back(new Shape("Shape1"));
                                         ^
Shape4.cpp:8:7: note: because the following virtual functions are pure within
'Shape':
class Shape
 ^
Shape4.cpp:21:17: note:     virtual float Shape::getarea()
                     virtual float getarea() = 0;
```

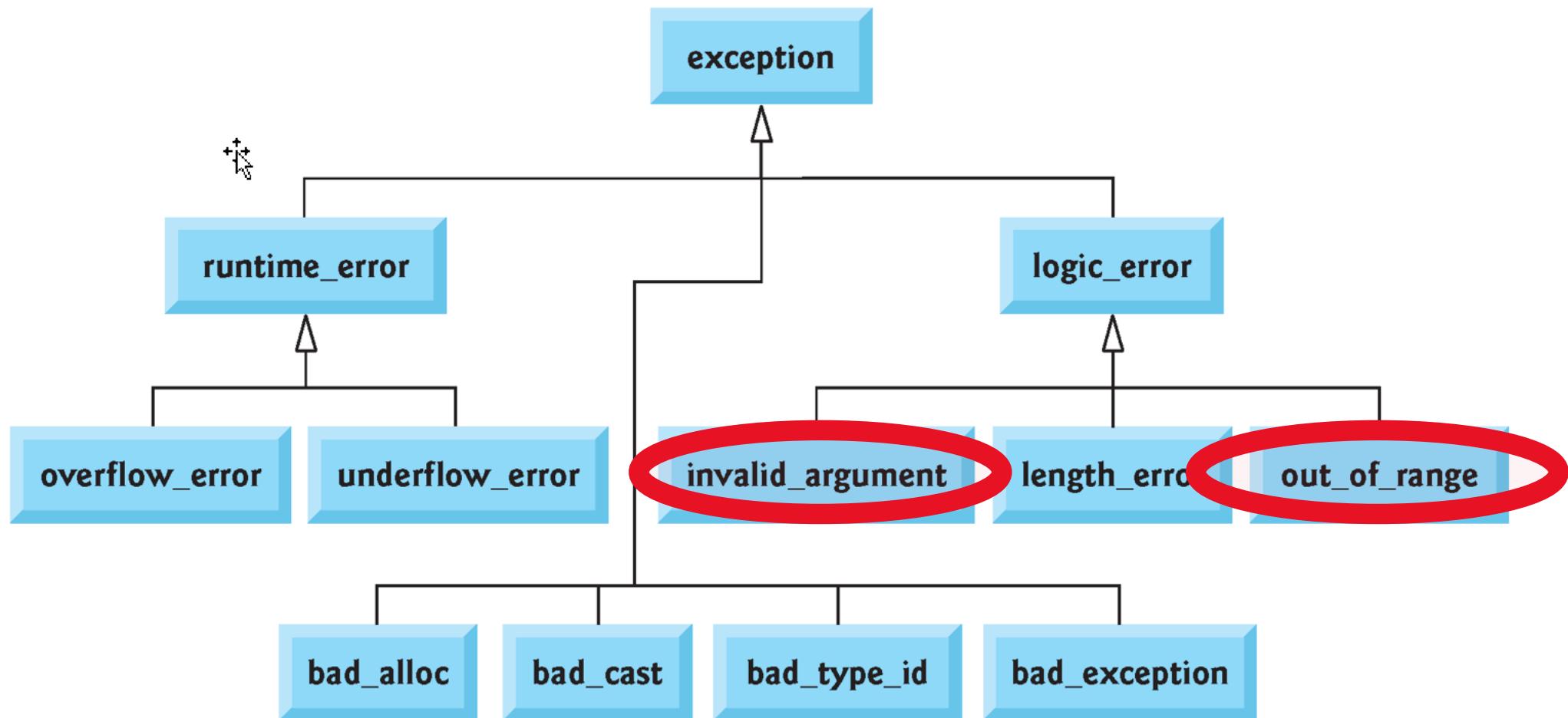
CSE 1325

Week of 11/30/2020

Instructor : Donna French

C++ Standard Exceptions

- Experience has shown that exceptions fall nicely into a number of categories.
- The C++ Standard Library includes a hierarchy of exception classes.



C++ Standard Exceptions

Exceptions thrown by C++ operators

- `bad_alloc`
 - **thrown by** `new`
- `bad_cast`
 - **thrown by** `dynamic_cast`
- `bad_typeid`
 - **thrown by** `typeid`

C++ Standard Exceptions

Exceptions indicating errors in program logic

- `invalid_argument`
 - indicates that a function received an invalid argument
- `length_error`
 - indicates that a length larger than the maximum size allowed for the object being manipulated was used for that object
- `out_of_range`
 - indicates that a value, such as a subscript into an array, exceeded its allowed range of values

C++ Standard Exceptions

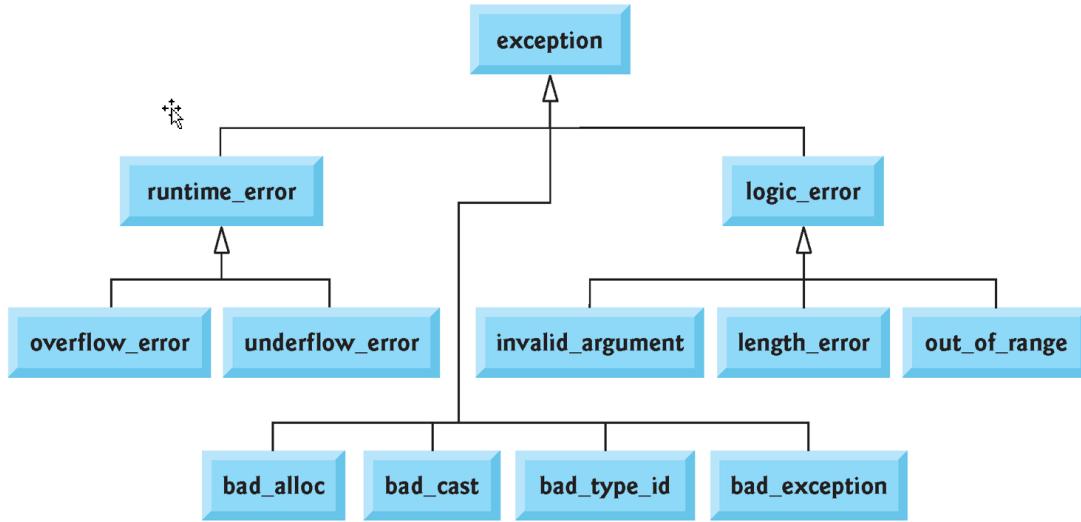
Exceptions indicating errors during run time

- `overflow_error`
 - describes an arithmetic overflow error
 - the result of an arithmetic operation is larger than the largest number that can be stored in the computer
- `underflow_error`
 - describes an arithmetic underflow error
 - the result of an arithmetic operation is smaller than the smallest number that can be stored in the computer

C++ Standard Exceptions

- This hierarchy is headed by base-class exception (defined in header file `<exception>`), which contains virtual function what that derived classes can override to issue appropriate error messages.
- If a catch handler catches a reference to an exception of a base-class type, it also can catch a reference to all objects of classes derived publicly from that base class—this allows for polymorphic processing of related errors

```
try
{
switch (Choice)
{
    case 1 :
        throw runtime_error("Threw runtime_error");
        break;
    case 2 :
        throw overflow_error ("Threw overflow error");
        break;
    case 3 :
        throw underflow_error ("Threw underflow error");
        break;
    case 4 :
        throw bad_alloc();
        break;
    case 5 :
        throw bad_cast("x");
        break;
    case 6 :
        aptr = nullptr;
        cout << typeid(*aptr).name() << endl;
        throw bad_typeid();
        break;
    case 6 :
        aptr = nullptr;
        cout << typeid(*aptr).name() << endl;
        throw bad_typeid();
        break;
    case 7 :
        throw bad_exception();
        break;
    case 8 :
        throw logic_error("Threw logic error");
        break;
    case 9 :
        throw invalid_argument("Threw invalid_argument");
        break;
    case 10 :
        throw length_error("Threw length_error");
        break;
    case 11 :
        throw out_of_range("Threw out_of_range");
        break;
    default :
        cout << "Don't know what to throw" << endl;
}
}
```



```

try
{
    switch...
}

catch (runtime_error &say)
{
    cout << say.what() << endl;
}
catch (logic_error &say)
{
    cout << say.what() << endl;
}
catch (exception &say)
{
    cout << say.what() << endl;
}
  
```

0. Exit
1. runtime_error
2. overflow_error
3. underflow_error
4. bad_alloc
5. bad_cast
6. bad_typeid
7. bad_exception
8. logic_error
9. invalid_argument
10. length_error
11. out_of_range

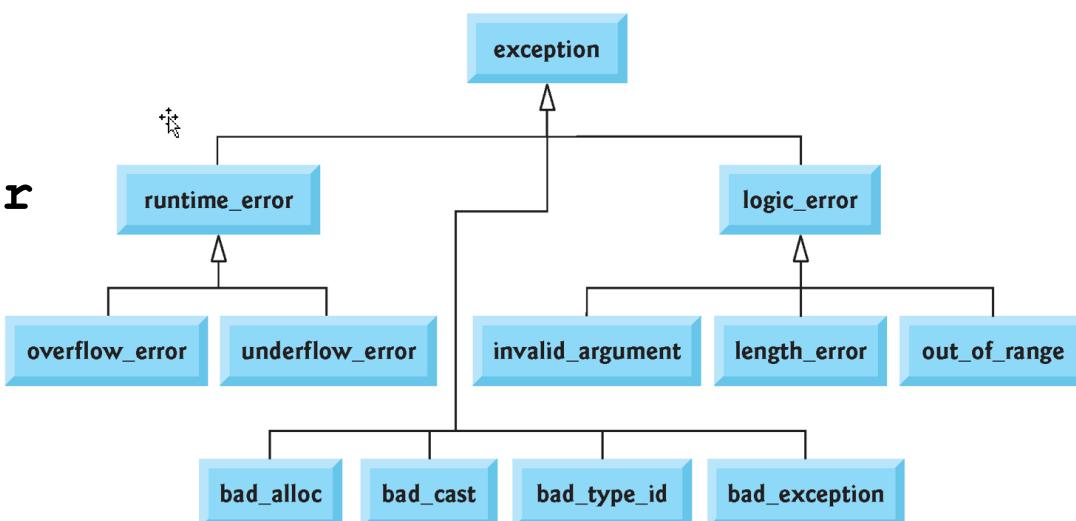
Enter Choice

0. Exit
1. **runtime_error**
2. overflow_error
3. underflow_error
4. bad_alloc
5. bad_cast
6. bad_typeid
7. bad_exception
8. logic_error
9. invalid_argument
10. length_error
11. out_of_range

Enter Choice 1

Threw runtime_error

```
catch (runtime_error &say)
{
    cout << say.what() << endl;
}
catch (logic_error &say)
{
    cout << say.what() << endl;
}
catch (exception &say)
{
    cout << say.what() << endl;
}
```



0. Exit
1. runtime_error
2. **overflow_error**
3. underflow_error
4. bad_alloc
5. bad_cast
6. bad_typeid
7. bad_exception
8. logic_error
9. invalid_argument
10. length_error
11. out_of_range

Enter Choice 2

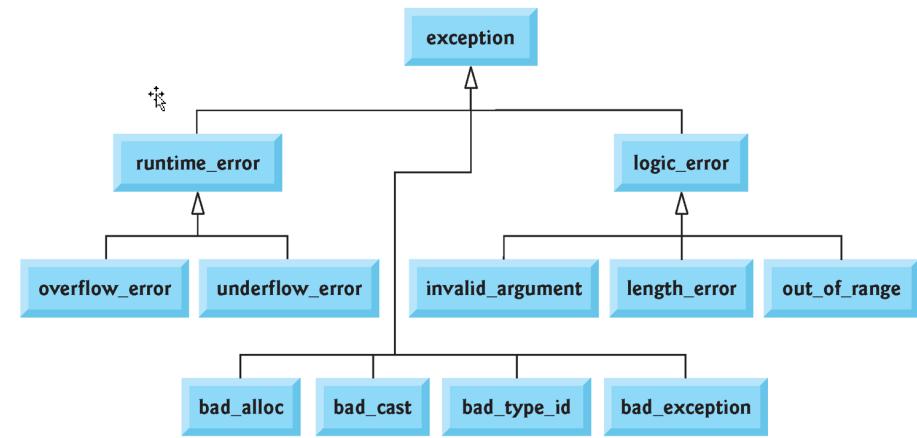
Threw overflow_error

```
0. Exit  
1. runtime_error  
2. overflow_error  
3. underflow_error  
4. bad_alloc  
5. bad_cast  
6. bad_typeid  
7. bad_exception  
8. logic_error  
9. invalid_argument  
10. length_error  
11. out_of_range
```

Enter Choice 3

Threw underflow error

```
0. Exit  
1. runtime_error  
2. overflow_error  
3. underflow_error  
4. bad_alloc  
5. bad_cast  
6. bad_typeid  
7. bad_exception  
8. logic_error  
9. invalid_argument  
10. length_error  
11. out_of_range  
Enter Choice 4  
std::bad_alloc
```



```
catch (runtime_error &say)  
{  
    cout << say.what() << endl;  
}  
catch (logic_error &say)  
{  
    cout << say.what() << endl;  
}  
catch (exception &say)  
{  
    cout << say.what() << endl;  
}
```

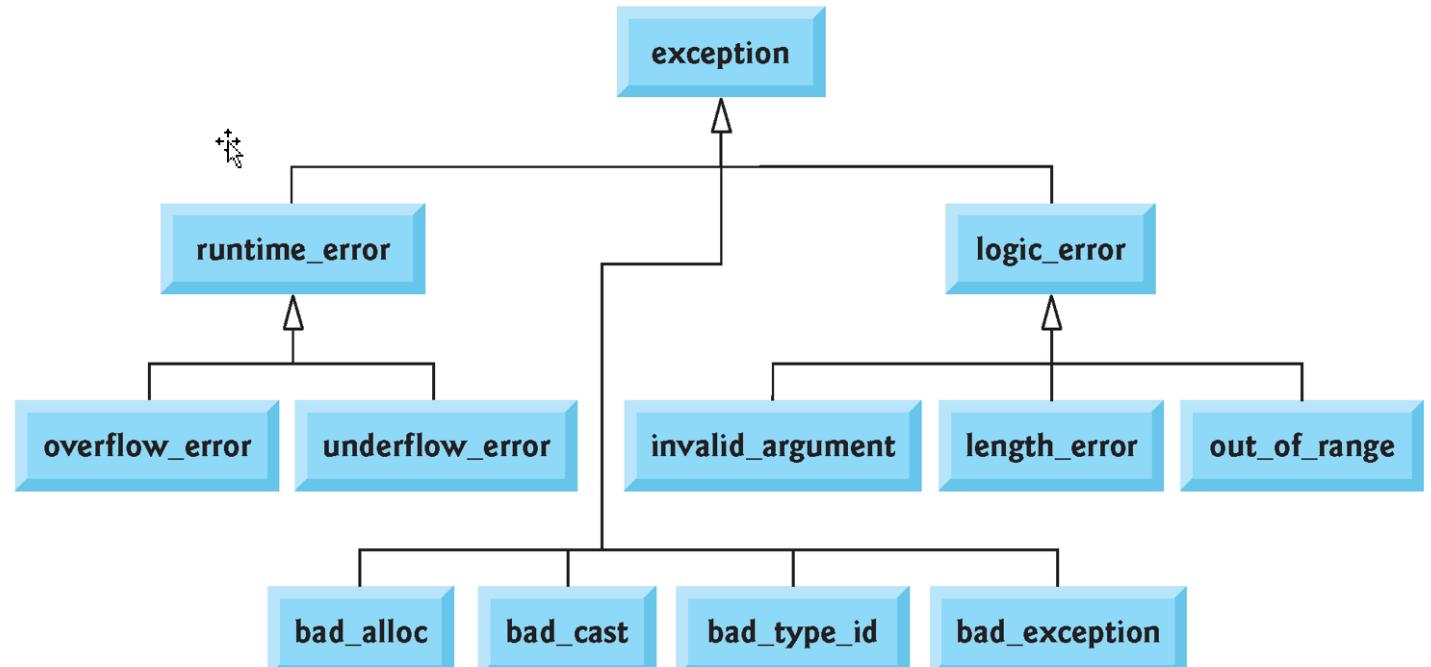
```
student@cse1325:/media/sf_VM/C++
```

```
Exiting program -
```

```
Missing command line parameters:
```

```
CANDYFILENAME
```

```
throw invalid_argument("\n\nMissing command line parameters:\nTOTFILENAME HOUSEFILENAME CANDYFILENAME")
```



```
try
```

```
{
```

```
    get_command_line_params(argc, argv, TOTFN, HFN, CFN);
```

```
}
```

```
catch (invalid_argument& say)
```

```
{
```

```
    cout << "Exiting program - " << say.what() << endl;
```

```
    exit(0);
```

```
}
```

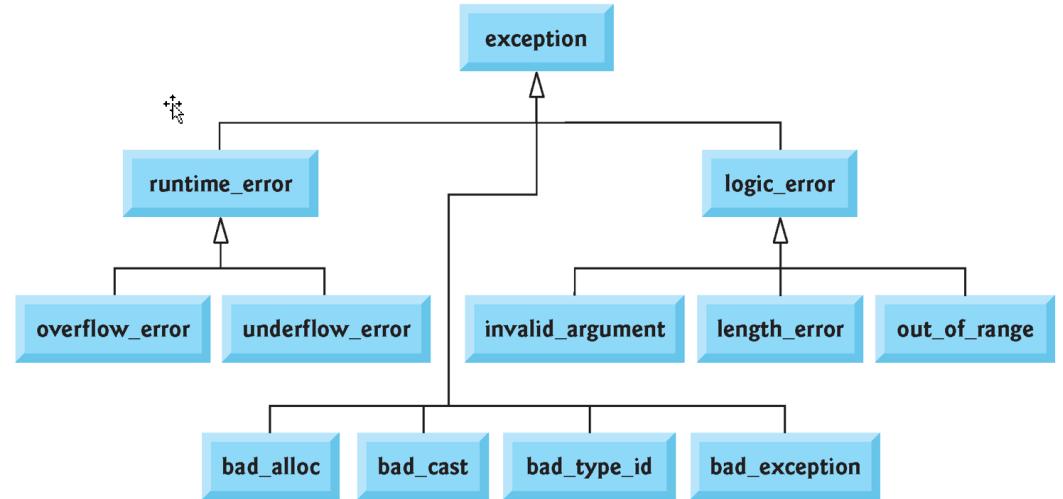
```

catch (invalid_argument& say)
{
    cout << "Exiting program - " << s
    exit(0);
}

catch (logic_error& say)
{
    cout << "Exiting program - " << say.what() << endl;
    exit(0);
}

catch (exception& say)
{
    cout << "Exiting program - " << say.what() << endl;
    exit(0);
}

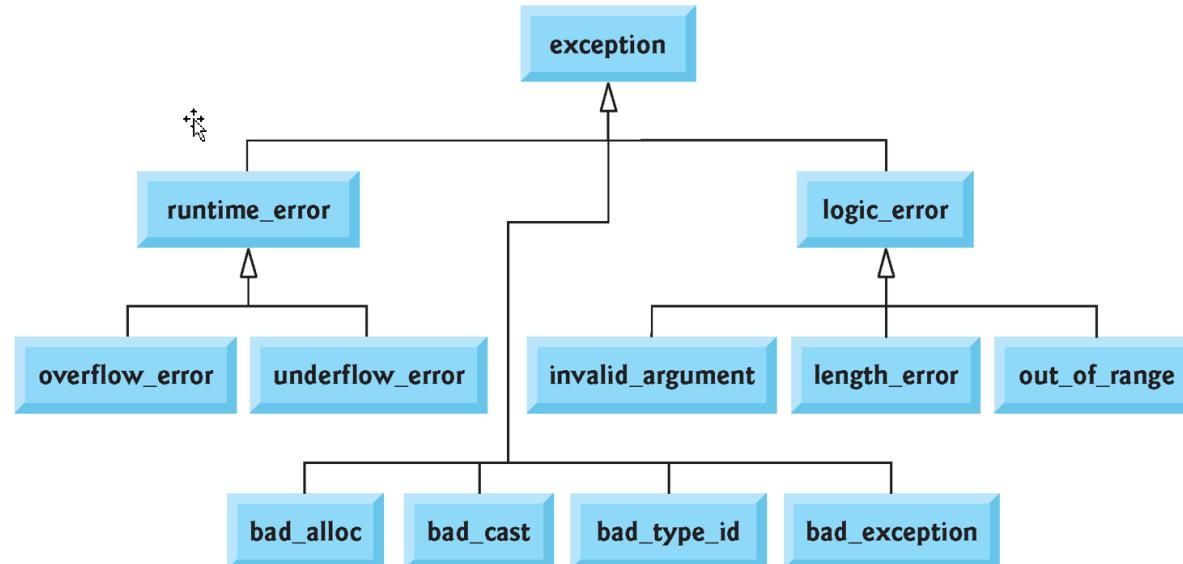
```



Custom Exception Handling

Class `runtime_error` and `logic_error` are derived class of exception (from header file `<exception>`)

Class `exception` is the standard C++ base class for exceptions in the C++ Standard Library.



Custom Exception Handling

A typical exception class that derives from the `runtime_error` class defines only a constructor that passes an error-message string to the base-class `runtime_error` constructor.

Every exception class that derives directly or indirectly from `exception` contains the virtual function `what()`, which returns an exception object's error message.

You are not required to derive a custom exception class from the standard exception classes provided by C++ but doing so allows you to use the virtual function `what()` to obtain an appropriate error message and also allows you to polymorphically process the exceptions by catching a reference to the base-class type.

```
student@cse1325:/media/sf_VMS$ ./custexFruitDemo.e
Enter any fruit except apple or orange
You followed instructions and entered pear
```

```
student@cse1325:/media/sf_VMS$ ./custexFruitDemo.e
Enter any fruit except apple or orange
How dare you enter apple!!
```

```
student@cse1325:/media/sf_VMS$ ./custexFruitDemo.e
Enter any fruit except apple or orange
Ewwww...orange?! - really?
```

```
student@cse1325:/media/sf_VMS$
```

```
int main(void)
{
    string fruit;

    try
    {
        getFruit(fruit);
        cout << "You followed instructions and entered " << fruit << endl;
    }
}
```

```
void getFruit(string &fruit)
{
    cout << "Enter any fruit except apple or orange ";
    cin >> fruit;

    if (fruit == "apple")
    {
        throw AppleEx();
    }
    if (fruit == "orange")
    {
        throw OrangeEx();
    }
}
```

```
int main(void)
{
    string fruit;

    try
    {
        getFruit(fruit);
        cout << "You followed instructions and entered "
            << fruit << endl;
    }
```

```
int main(void)
{
    string fruit;

    try
    {
        getFruit(fruit);
        cout << "You followed instructions and entered " << fruit << endl;
    }
    catch (const AppleEx &say)
    {
        cout << say.what();
    }
    catch(const OrangeEx &say)
    {
        cout << say.what();
    }

    return 0;
}
```

```
class AppleEx : public std::logic_error
{
public:
    AppleEx() : std::logic_error{"How dare you enter apple!!\n"}
{
}
};

class OrangeEx : public std::logic_error
{
public:
    OrangeEx() : std::logic_error{"Ewwww...orange?! - really?\n"}
{
}
};
```

Both classes publicly inherit from `logic_error`; therefore, inherit the ability to store a statement in the object which can later be retrieved using `what()`.

Casting

What if we need a member function that we don't want to inherit from the base class?

What if we add a member function to a derived class?

```
class Circle : public Shape  
{  
public:  
    float getDiameter()  
    {  
        return 2 * (dim1 + dim2);  
    }  
};
```



If this was defined in Shape, then Square, Rectangle and Triangle would inherit it for no reason.

Casting

```
for (auto it : MyShapes)
{
    it->Hello();
    cout << it->getName() << "'s area is " << it->getarea() << endl;
    cout << it->getDiameter() << endl;
}
```

```
student@cse1325:/media/sf_VM$ g++ Shape5.cpp -g -std=c++11
Shape5.cpp: In function 'int main()':
Shape5.cpp:155:15: error: 'class Shape' has no member named 'getDiameter'
    cout << it->getDiameter() << endl;
               ^
```

Dynamic Casting

```
for (auto it : MyShapes)
{
    it->Hello();
    cout << it->getName() << "'s area is " << it->getArea() << endl;

    Circle* IamaCircle = dynamic_cast<Circle*>(it);
    if (IamaCircle != nullptr)
    {
        cout << "My diameter is " << IamaCircle->getDiameter() << endl;
    }
}
```

`dynamic_cast` returns a value of `nullptr` if the input object is not of the requested type.

In this `for` loop, `dynamic_cast` will not be equal to `nullptr` when `it` is a `Circle`.

`IamaCircle` is then a pointer to `Circle` that can call `Circle`'s member function `getDiameter()`.

Dynamic Casting

```
for (auto it : MyShapes)
{
    it->Hello();
    cout << it->getName() << "'s area is " << it->getarea() << endl;

    Circle* IamaCircle = dynamic_cast<Circle*>(it);
    if (IamaCircle != nullptr)
    {
        cout << "My diameter is " << it->getDiameter() << endl;
    }
}
```

Shape5.cpp: In function 'int main()':

Shape5.cpp:164:37: error: 'class Shape' has no member named 'getDiameter'

```
cout << "My diameter is " << it->getDiameter() << endl;
```

Static Casting

```
Square S2("S2", 5);
```

```
cout << S2.getName() << " is a " << S2.MySquareFunction() << endl;
```

S2 is a Square

```
class Square : public Rectangle
{
public:
    string MySquareFunction(void)
    {
        return "Square";
    }
}
```

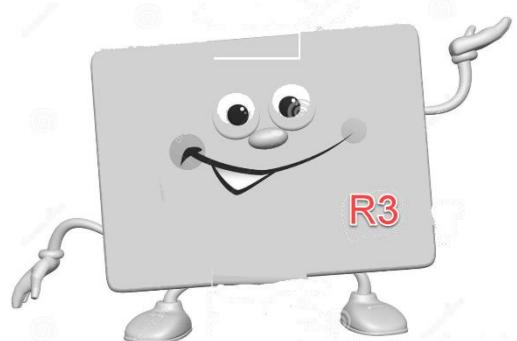
Static Casting

```
Square S2 ("S2", 5);  
cout << S2.getName() << " is a " << S2.MySquareFunction() << endl;
```



```
Rectangle R3 = static_cast<Rectangle>(S2);
```

```
cout << S2.getName() << " is a " << S2.MySquareFunction() <<
```



S2 is a Square

```
cout << R3.getName() << " is a " << R3.MySquareFunction() << endl;
```

```
Shape5.cpp: In function 'int main()':  
Shape5.cpp:173:41: error: 'class Rectangle' has no member named 'MySquareFunction'  
    cout << R3.getName() << " is a " << R3.MySquareFunction() << endl;
```

Static Casting

```
Square S2 ("S2", 5);  
cout << S2.getName() << " is a " << S2.MySquareFunction() << endl;
```

```
Rectangle R3 = static_cast<Rectangle>(S2);
```

```
Square S3 = static_cast<Square>(R3);
```

Can we cast R3 back to a Square since it was a Square before being cast to a Rectangle?

```
Shape5.cpp: In function 'int main()':  
Shape5.cpp:174:36: error: no matching function for call to 'Square::Square(Rectangle&)'  
    Square S3 = static_cast<Square>(R3);  
  
Shape5.cpp:98:3: note: candidate: Square::Square(std::__cxx11::string, float)  
    Square(string shapeName, float size) : Rectangle()  
  
Shape5.cpp:98:3: note:   candidate expects 2 arguments, 1 provided  
Shape5.cpp:95:7: note: candidate: Square::Square(const Square&)  
class Square : public Rectangle  
  
Shape5.cpp:95:7: note:   no known conversion for argument 1 from 'Rectangle' to  
'const Square&'
```

Static Casting

```
Square S2 ("S2", 5);  
cout << S2.getName() << " is a " << S2.MySquareFunction() << endl;  
Rectangle R3 = static_cast<Rectangle>(S2);
```

```
Square S3 = static_cast<Square>(R3);
```

Rectangle is the base class for Square.

Derived class Square can be cast to its base class Rectangle but base class Rectangle cannot be cast to derived class Square.

Any extra/added after inheritance properties of the derived class are lost when cast to the base class.

A base class object cannot be cast to a derived class object because then it would be an object of that type without any of the extra/added after inheritance properties that other objects of that derived class have.

Function templates enable you to conveniently specify a variety of related (overloaded) functions—called function-template specializations.

Class templates enable you to conveniently specify a variety of related classes—called class-template specializations.

Programming with templates is known as generic programming.

Function templates and class templates are like stencils out of which we trace shapes; function-template specializations and class-template specializations are like the separate tracings that all have the same shape, but could, for example, be drawn in different colors , line thicknesses and textures.

Class Templates

Class templates encourage software reusability by enabling a variety of type-specific class-template specializations to be instantiated from a single class template.

Class templates are called parameterized types, because they require one or more type parameters to specify how to customize a generic class template to form a class-template specialization.

When a particular specialization is needed, you use a concise, simple notation, and the compiler writes the specialization source code.

Class Templates

To create a template specialization with a user-defined type, the user-defined type must meet the template's requirements.

For example, the template might compare objects of the user-defined type with `<` to determine sorting order. Or, the template might call a specific member function on an object of the user-defined type.

If the user-defined type does not overload the required operator or provide the required functions, compilation errors occur.

Class Templates

So what if we want to create a class that contains the data and member functions to emulate stack behavior?

We will use a vector but want to be able to call functions like push and pop.

And, we want to be able to use our stack with ints, chars and doubles.

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

using namespace std;

int main()
{
    Stack<double> doubleStack; // create a Stack of double
    Stack<int> intStack;      // create a Stack of int
    Stack<char> charStack;   // create a Stack of int
```

This is instantiating objects doubleStack, intStack and charStack.

They are using template class Stack. The part inside the <> is the type we want Stack to substitute in the template.

```
int StackSize = 0;  
  
double doubleValue{1.1}; // first value to push  
int intValue{1}; // first value to push  
char charValue{'a'};
```

Setting up a variable for the size of our Stacks and initializing the first value going into our Stacks.

```
StackSize = 5;  
cout << "\nPushing " << StackSize << " elements onto doubleStack\n";
```

```
// push 5 doubles onto doubleStack  
for (int i = 0; i < StackSize; ++i)  
{  
    doubleStack.push(doubleValue);  
    cout << doubleValue << "\t";  
    doubleValue += 1.1;  
}
```

```
cout << "\n\nPopping elements from doubleStack\n";
```

```
// pop elements from doubleStack  
while (!doubleStack.isEmpty())  
{  
    cout << doubleStack.top() << "\t"; // display top element  
    doubleStack.pop(); // remove top element  
}
```

push () is a member function of our object doubleStack.

isEmpty () is a member function of our object doubleStack.

top () and pop () are member functions of our object doubleStack

```
StackSize = 10;
cout << "\n\n\nPushing " << StackSize << " elements onto intStack\n";

// push 10 integers onto intStack
for (int i = 0; i < StackSize; ++i)
{
    intValue.push(intValue);
    cout << intValue++ << "\t";
}

cout << "\n\nPopping elements from intStack\n";

// pop elements from intStack
while (!intStack.isEmpty())
{
    cout << intStack.top() << "\t";
    intStack.pop();
}
```

Because intStack was instantiated using class Stack, it also has member functions push(), isEmpty(), top() and pop().

```
StackSize = 3;  
cout << "\n\n\nPushing " << StackSize << " elements onto charStack\n";  
  
// push 3 integers onto charStack  
for (int i = 0; i < StackSize; ++i)  
{  
    charStack.push(charValue);  
    cout << charValue++ << "\t";  
}  
  
cout << "\n\nPopping elements from charStack\n";  
  
// pop elements from charStack  
while (!charStack.isEmpty())  
{  
    cout << charStack.top() << "\t";  
    charStack.pop();  
}
```

Because charStack was instantiated using class Stack, it also has member functions push(), isEmpty(), top() and pop().

Class Templates

`charStack`, `intStack` and `doubleStack` all have access to the same functions because they were instantiated from the same class `Stack`.

So how did class `Stack` create a vector of characters, ints and doubles?

Class `Stack` is a template class.

First, we set up the include guard in our Stack.h file.

We need the include for vector since we are using a vector as our "stack".

We want this to be a template class so we add

template <typename T>

right before class Stack to make our class a template.

```
// Stack class template.
```

```
#ifndef STACK_H
```

```
#define STACK_H
```

```
#include <vector>
```

```
template<typename T>
```

```
class Stack
```

```
{
```

```
// class body on next slide
```

```
};
```

```
#endif
```

```

template<typename T>
class Stack
{
public:
    const T &top()
    {
        return stack.front();
    }

    void push(const T& pushValue)
    {
        stack.insert(stack.begin(), pushValue);
    }

    void pop()
    {
        stack.erase(stack.begin());
    }

    bool isEmpty() const
    {
        return stack.empty();
    }

    int size() const
    {
        return stack.size();
    }

private:
    std::vector<T> stack;
};

#endif

```

Private data member `stack` is a `vector` of type `T` where `T` is our template substitution. So when we instantiated our objects, we passed in the type we wanted substituted for `T`.

```

Stack<double> doubleStack;
Stack<int> intStack;
Stack<char> charStack;

```

The public member functions then take advantage of vector's abilities to emulate stack behavior.

Notice that function `top()` returns the first value of the vector using `front()`. The `T` in

```
const T &top()
```

will be replaced by `double` when `doubleStack` is instantiated and by `int` when `intStack` is instantiated and by `char` when `charStack` is instantiated.

```
template<typename T>
class Stack
{
public:
    const T &top()
    {
        return stack.front();
    }

    void push(const T &pushValue)
    {
        stack.insert(stack.begin(), pushValue);
    }

    void pop()
    {
        stack.erase(stack.begin());
    }

    bool isEmpty() const
    {
        return stack.empty();
    }

    int size() const
    {
        return stack.size();
    }

private:
    std::vector<T> stack;
};

#endif
```

Notice that function `push()` has a parameter `pushValue` of type `T`. The `T` in

`void push(const T &pushValue)`

will be replaced by `double` when `doubleStack` is instantiated and by `int` when `intStack` is instantiated and by `char` when `charStack` is instantiated.

```
template<typename T>
class Stack
{
public:
    const T &top()
    {
        return stack.front();
    }

    void push(const T &pushValue)
    {
        stack.insert(stack.begin(), pushValue);
    }

    void pop()
    {
        stack.erase(stack.begin());
    }

    bool isEmpty() const
    {
        return stack.empty();
    }

    int size() const
    {
        return stack.size();
    }

private:
    std::vector<T> stack;
};

#endif
```

Be careful when using templates – don't try to use a data type when instantiating objects using a class template that the functions in the template cannot handle.

In this example, the data type that T will be replaced with must be a type that you can create a vector of and that those vector functions can handle.