AMS 562 Lecture Notes

Release 2020.F

Qiao Chen

Aug 22, 2020

INTRODUCTION

CHAPTER

ONE

WHY C++?

Table of Contents

1.1 You Should Consider Using C++ If

The chances are that you may heard a lot of good things of C++, and finally decide to give it a shot. Great! There is not doubt with the power of C++. It is one of the most widely used programming languages, found in a large range of applications. Unsurprisingly, *scientific computing* communities use C++ quite often in their projects. For instance, if you are interested in solving *partial differential equations* (PDEs), then open-sourced frameworks such as OpenFOAM and SU2 might attract people who work in *computational fluid dynamics* (CFD), and FEniCS and deal.II for researchers who are interested in *finite element methods* (FEM) that has been used in structural problems for decades. Of course, the *computational geometry algorithms library* (CGAL) is very popular for computer scientists who develop geometry-based algorithms. What about for students who work in much lower level research areas, such as developing numerical algorithms. No problem, Eigen has been around for a while that supports efficient representations of fundamental numerical computational objects, like vectors, matrices, and tensors. OK, I think I can stop here!

Concepts that you learn from C++ can be directly used in other programming languages such as Java, Python, MATLAB, C, Fortran, etc. Personally, I believe writing an algorithm in C++ helps you fully understand the idea and, potentially, can also inspire you for the algorithm design. Let's say, you probably has already learned how to do matrix multiplications in your college linear algebra classes. Then, when your instructors asked you to implement this as homework assignment in MATLAB, what you did is probably something like the following

```
A = rand(3);
B = rand(3);
C = A*B;
```

or in Python

```
import numpy as np
A = np.random.random((3,3))
B = np.random.random((3,3))
C = A.dot(B)
```

Well, I am kidding as this is cheating as homework assignments. You probably implemented the triple for-loops. If you did a timing analysis, you would probably see that the triple for-loop version is more than 100 times slower. Languages like Python or MATLAB hide details from users, for instance, many students don't understand how matrices are stored in MATLAB and *numpy* and the difference between accessing from column `j` to column `j+1` in these two languages, i.e. speed is different when accessing `A[i,j:j+2]` and `A(i,j:j+1)`. On the other side, C++ hides nothing from users.

1.2 You Should NOT Consider Using C++ If

Very well, so I should only code with C++!

Wait, wait... There are many things that can simply be done with in 1 or 2 lines in Python, but may take you hundreds of lines in C++. Let's still use the above matrix multiplication example. First, the concept of matrices does not exist in **native** C++, so you need define it. Once you have the matrix *class*, you, of course, cannot expect that C++ is smart enough to automatically generate those linear algebra operations for you, as a result, you need to implement them. Finally, debugging your implementations will take a fairly amount of time.

In short, there are several situations that I think you should not consider directly using C++

1.2.1 "By the end of the day, all I want is something..."

Well, with C++, *"by the end of the day"*, you are still struggling with software designs. Typically, it's not easy to get something working in C++ as nicely as *high level* programming languages in short terms.

1.2.2 "My project/research is still at infant stage, I wanna test some ideas."

C++ is not a friendly language for developing algorithms... MATLAB, for example, is a great framework for developing numerical algorithms. As a matter of fact, all SBU students can use it "freely" under its educational license.

Note: This statement works for most students who study in typical applied math fields. However, for students majoring in *high performance computing* (HPC), stories are different.

1.3 “I’m confused. . . ”

“I am here to learn C++, but it seems like I should not use it. . . ”

Learning C++ is always beneficial, even though you may not use it right away! Personally, I believe learning C++ can help you understand more and be more professional in programming. For instance, if you program for Python and Java, C++ can help you understand more about their *garbage collections* (GC). Also, *shadow copy* will no longer be an enemy of you, because you know what’s going on under the hood.

If you code for MATLAB, C++ can help you be aware of *pass by value* (PBV), and how to avoid unnecessary data copying so that you can write more efficient MATLAB scripts.

1.4 How To Read The Lecture

By “I”, I mean myself. By “We”, I mean the Department of Allied Mathematics & Statistics (AMS) and(or) Institute for Advanced Computational Science (IACS). Also, it’s worth noting that this lecture note is mainly created as memos for the course, and sharing some experiences in programming C++. So I will use the informal word “you” to address the students. Finally, the term C++ implicitly refer to *C++11* (see *C++ Standard*), for other versions, I will address them explicitly.

Sphinx is really a great tool for writing this note. Here, I leverage several nice directives to do highlighting.

Tips, used for giving some tips/tricks

Tip: *Lecture Zero* is great place to start.

Notes, used for emphasis on technical points

Note: My fundamental assumption is that you all are new to C++ programming but with decent experiences in at least one of Python, MATLAB/Octave, or Java.

Warnings, danger areas!

Warning: Generally speaking, C++ is not for someone who has no programming background.

1.5 Acknowledgments

I would like first to thank AMS and IACS for giving me this great opportunity for teaching this course. Also, my supervisor, Prof. Jim Jiao, helps a lot in sharing his experiences in teaching and setting up the Docker images.

CHAPTER

TWO

LECTURE ZERO

Table of Contents

2.1 Comment! Please Comment...

All programming languages have their way to comment the source codes. In Python, people use hash tag to start a line of comment

```
# This is a comment
# This is another comment
```

In MATLAB, people use percentage sign, i.e.

```
% This is a comment
%{
This
is
a block of
comments
%}
```

In C++, we use double forward slashes // to comment, i.e.

```
// This is a c++ comment
// Compiler will not read me!
```

Of course, the old C style comment is also valid, i.e.

```
/*
compiler will
not read
anything
in this
block
*/
```

Tip: For C++, it's better to stick with `//` even though you have multiple lines of comments.

2.1.1 Making comments make others life better.

Imagine you join a research group and continue on some former member's work. The only thing you get is his/her codes without a single line of comments. Then, let's just hope the codes are robust enough thus having no errors (99.99% chance no!). Otherwise, it will waste you lots of time (sometimes can be a whole semester) to figure out how to make the code run.

Also notice that, eventually, you will become someone's "former" member, so do not make your academic litter brothers/sisters hate you.

2.1.2 "I'm so glad that I commented!"

Actually, you are the one who benefit the most from making comments. While doing research, an important thing that everybody needs to keep in mind is to make the work reproducible. One needs to make sure that his/her results can be reproduced in the future. Therefore, make comments for yourselves in the future.

Of course, there are better tools, e.g. Git see *here*, to help you manage your work. But making comments are the most fundamental requirement that.

Let's take a look at the following examples:

```
// This function does blah blah ...
// This is the core component in the algorithm of step 1 in my paper...
// Essentially, this function is an extension of blah blah ...
//
// The following inputs arguments are needed:
//  ...
// This function returns an integer flag that indicates ...
//
// See Also
//  compute_next, compute_final
int compute_core(...) {
    // The first step is to ...
    ...
    // The second step is to ...
    ...
    // WARNING! The following codes assume ...
    ...
    // Finally clean up everything with ...
```

(continues on next page)

(continued from previous page)

```
    ...
}
```

vs.

```
// compute core function
int compute_core(...) {
    ...
}
```

2.2 Naming Conventions

Defining variable names is not a trivial task. There are some standard rules, but for this class, I will briefly share some of my **personal** styles.

Warning: The following information is based on personal experience, thus it is subjective.

2.2.1 Rule I: Make the variable names meaningful.

Historically, people like to use `foo`, `bar`, or `spam` to demonstrate ideas. Usually, they don't assume or know the backgrounds of the readers, so they choose some *placeholder* names that can be replaced in real codes. Hence, avoid using these names in your projects; no one wants to see placeholders in the final products.

Another commonly used word is `temp` or `tmp` that stands for temporary variables. Personally, I use this word only for some variables that have short lifetime.

A rule of thumb is to ask your workmates and check whether they can understand the meaning of your variables.

2.2.2 Rule II: Make the variable names compact.

Making things meaningful doesn't mean you have to be verbose. Also, don't forget we have just learned how and why to make comments! Commonly, people use abbreviations for their variable names, and this is what we shall follow.

Tip: Choose an abbreviation that will appear in your project documentation or your papers/reports.

2.2.3 Rule III: Most functions should start with verbs.

This idea is commonly agreed by most of people, especially in scientific programming.

Tip: Some commonly used verbs: `set`, `get`, `fetch`, `extract`, `is`, `assign`, `compute`, `determine`, `swap`, `move`, `init`, `destroy`, `create`, `remove`, `reset`, `reserve`, `put`, `resize`, etc.

Note: C++ encourages people to hide *member variables* (future) and access them through *member functions* (future), e.g. `size`, `length`, `data`, `capacity`, etc.

2.3 Format Your Files

Besides comment your codes, another important aspect is to have a nice layout of your codes, so that it makes others who read your work enjoyable.

```
for(int a=0;a <10;   ++a)  {
        foo[a] =1.0; bar[a]=2.0;
    spam[a]   =   3.0;}
```

vs.

```
for (int a = 0; a < 10; ++a) {
    foo[a]  = 1.0;
    bar[a]  = 2.0;
    spam[a] = 3.0;
}
```

Well, I vote for the second one!

However, manually doing this is a pain. We shall leverage the automatic tool such as clang-format, which can be used through our editor (vscode) through the Docker image for this course.

2.4 C++ Standard

The first ISO C++ standard was ratified in 1998, so that version is referred as C++98. Later on, we got C++03. **The game changer is C++11**, which introduced many unique features. Personally, I think C++11 is totally a different language compared to its previous versions. For this course, all materials are based on C++11, and the term C++ is implicitly referred to version 11, for instance, the following statement is correct under this assumption: *In C++, you can use lambdas instead of functors*.

The current standard is C++17. However, in scientific computing, the truth is that there usually exists a gap between current standard in practice and current ISO standard. The good news is that most of open-sourced projects have moved or are moving to C++11, it's safe to assume C++11 for audiences who are interested in *scientific programming*. As a matter of fact, the first feature-complete GCC was version 4.8.1[1], which was released on May 31, 2013[2]. Also, our Seawulf cluster's system GCC has

[1] see GCC C++ standard supports

[2] see GCC release

version 4.8.5, which means it fully supports C++11.

CHAPTER

THREE

SIMPLE C++

Table of Contents

3.1 The `main` functions

Each C++ program is written in a `main` function in C++, where a `main` function must return an integer to indicate your system the exit status of the program.

Note: 0 is used for indicating exiting successfully.

Here is the simplest C++ program, `int main() { return 0; }`, which does nothing but just returns `EXIT_SUCCESS` to your system.

3.2 Compile the one-line program

Unlike Python and MATLAB, where you can directly invoke the scripts, all C++ programs must be first **compiled** into executable binaries.

Note: The compilation stage is one of the key reason why static languages are faster than dynamic languages.

Now, copy the one line program into a file with name `simple.cpp`, inside a terminal, invoke:

```
$ g++ simple.cpp
```

This by default will compile the program into an executable file called `a.out` that lies in the same directory. To run the program, type:

```
$ ./a.out
```

Tip: `./` in front of the executable means the file is under *current working directory* (cwd). You can type `pwd` in the terminal to check *cwd*. In general, `.` means current, `..` means previous, and `/` is the path separator on Linux. To navigate to a directory through terminal, you need to use the built-in command `cd`, which stands for *change directory*. For instance, if I want to go to previous directory, I can simply type `cd ...` `cd ./foo` will bring me to the `foo` folder that locates at *cwd* (you can omit `./` in this case). Absolute path can also be used. For example, `cd /path/to/my-homework` will navigate to `/path/to/my-homework`, and the leading `/` on Linux means the root directory.

Of course, this program seemingly does nothing. To check the returned code, type `echo $?`, which will gives you the exit-code of most recent program. You should see `0` on the screen.

3.3 "Hello World!"

Unfortunately, you cannot write one line code for "Hello World!" in C++. In Python, you can write a `hello_world.py` script with:

```
print('Hello World!')
```

And simply type:

```
$ python3 hello_world.py
```

You should see "Hello World!" on your screen. Or even something like:

```
$ python3 -c "print('Hello World!')"
```

However, there is no built-in `print` method in C++, we have to include the standard input and output library, i.e. `iostream`.

```
1 #include <iostream>
2 
3 int main() {
4     std::cout << "Hello World!" << std::endl;
5     return 0;
6 }
```

Once we include the IO library (line 1), we can use the standard output streamer, i.e. `std::out`, to write out messages (line 4).

Now, copy the program into `hello_world.cpp`, and compile and run it.

Note: This section is basically to demonstrate some simple codes. There will be a specific section talking about IO.

CHAPTER

FOUR

SOFTWARE REQUIREMENTS

Table of Contents

Note: In this section, we will briefly introduce the software skills needed for this course. We will provide useful links for you to further train yourself.

4.1 Using Our Docker Container

Why not take the advantage of cloud computing for teaching? The answer we come up with is Docker containers, which allow you run a collection of software packages (including OS) regardless your host machine systems. This class is cluster-free, i.e. you don't need to worry about dealing with our clusters and running everything through command lines.

First, read the description of our Docker image, i.e. AMS 562 container. Once you have installed Docker and Python, download these two scripts that will ease the process for using our container.

- Desktop driver: ams562_desktop.py
- Jupyter driver: ams562_jupyter,py

For the first time, open a terminal/console/powershell session:

```
$ python ams562_desktop
```

This will automatically pull the container and create the desktop environment in your default web browser. To see all options:

```
$ python ams562_desktop -h
usage: ams562_desktop.py [-h] [-i IMAGE] [-t TAG] [-v VOLUME] [-w WORKDIR]
                         [-p] [-r] [-c] [-d] [-s SIZE] [-n] [-N] [-V] [-q]
                         [-A ARGS]

Launch a Docker image with Ubuntu and LXDE window manager, and␣
↪automatically
```

(continues on next page)

(continued from previous page)

```
open up the URL in the default web browser. It also sets up port forwarding
for ssh.

optional arguments:
-h, --help            show this help message and exit
-i IMAGE, --image IMAGE
                      The Docker image to use. The default is
                      ams562/desktop.
-t TAG, --tag TAG     Tag of the image. The default is latest. If the image
                      already has a tag, its tag prevails.
-v VOLUME, --volume VOLUME
                      A data volume to be mounted at ~/" + projdir + ". The
                      default is ams562_project.
-w WORKDIR, --workdir WORKDIR
                      The starting work directory in container. The default
                      is ~/project.
-p, --pull            Pull the latest Docker image. The default is not to
                      pull.
-r, --reset           Reset configurations to default.
-c, --clear           Clear the project data volume (please use with
                      caution).
-d, --detach          Run in background and print container id
-s SIZE, --size SIZE  Size of the screen. The default is to use the current
                      screen size.
-n, --no-browser      Do not start web browser
-N, --nvidia          Mount the Nvidia card for GPU computation. (Linux
                      only, experimental, sudo required).
-V, --verbose         Enable verbose mode and print debug info to stderr.
-q, --quiet           Disable screen output (some Docker output cannot be
                      disabled).
-A ARGS, --args ARGS  Additional arguments for the "docker run" command.
                      Useful for specifying additional resources or
                      environment variables.
```

Tip: Always run with `$ python ams562_desktop -p` to get the newest container since our Docker image is automatically rebuilt weekly.

The following directories are mirrored to your local machine:

Docker directories	Host directories
`$DOCKER_HOME/shared`	Current working directory
`$DOCKER_HOME/project`	Data volume
`$DOCKER_HOME/.ssh`	`$HOME/.ssh`
`$DOCKER_HOME/.config`	`$HOME/.config`

The most commonly used is the `shared` directory, which allows you rapidly exchange data between the container and your host machine.

Warning: Except the above four directories, any changes to the container will not be persistent.

4.2 Using Git with SSH Keys

4.2.1 What is Git?

Version control system definitely helps your work. I found this online material is interesting and helpful, please take a look at it.

SmartGit is a nice GUI system for Git.

Tip: You may also find using Git in Visual Studio Code is handy, see *here*.

4.2.2 Set up your private repository on Bitbucket

We will use one of the popular online git network, Bitbucket, to collect your homework assignments. Register an account with your **SBU** email address, then create a **private** repository following this description.

Name your repository with `ams562_<your name>`. And initialize your repository with a `README.md` that at least includes your SBU ID and name, something like the following is fine:

```
# Welcome to my repository for AMS 562

* name: <your name>
* SBU ID: <your id>
```

4.2.3 Set up an SSH Key (optional)

Using Secure Shell is the preferred way (but not required for this class) for using Git. Follow this description to setup an SSH key for your Bitbucket account.

Note: Keep your private key and **passphrase** secure!

4.3 Code with Visual Studio Code

Using a decent editor is necessary, and develop with IDE-like environment is extremely helpful. Among all existing popular editors, we have decided to provide the Visual Studio Code that is developed by Microsoft. This editor has been installed and properly configured in our Docker image.

4.3.1 Using git Inside VScode

Using Git trough terminal might be confusing for people who first work. To make things more consistent with our Docker setting, we find that using Git through Visual Studio Code is extremely convenient. See this description.

Notice that if you use Git through *SSH*, then you need to run the following command inside the terminal of the container:

```
$ ssh-add
Enter passphrase for /path/to/.ssh/<your private key>:
```

Then enter your passphrase that your created for the key. This is done once only.

4.3.2 Using Terminal Inside VScode

Using the integrated terminal of vscode is recommended, you can create/return to a terminal session by type `CTRL-``. By default, it will put you at the *current working directory*, then you can invoke any commands inside this integrated terminal.

Tip: You can directly open a file by clicking on its absolute path inside the integrated terminal. Therefore, it's convenient to embed abs path of a file inside the integrated terminal. One easy way to do so is to add `` `pwd`/ `` in front of your file.

```
$ g++ `pwd`/main.cpp
```

CHAPTER

FIVE

LECTURE 1: TYPES & I/O

Table of Contents

5.1 C++ is all about TYPES!

Unlike Python (or any other dynamic languages), all C++ variables must be initialized with their types explicitly given. And the variable names cannot be reused within the same *cope*. Consider the following Python code

```
In [1]: a = 1

In [2]: type(a)
Out[2]: int

In [3]: a = 1.0

In [4]: type(a)
Out[4]: float

In [5]: a = 'a'

In [6]: type(a)
Out[6]: str
```

In the program, variable `a` first is initialized as an integer, but later on it switches to floating point number and string. **This, however, is not allowed in C++.**

5.1.1 The Built-in Types

A built-in is a component that comes with the programming language; using built-in components does not require you import any external interfaces (even including official ones). In C++, we have built-in data types that define the foundation of the language (or even other languages). The built-in types can be mainly divided into three groups:

1. integral types,
2. floating number types, and
3. the valueless type, i.e. `void`.

The Integral Types

Let's put item 3 apart now. The integral types can be further categorized into:

- integers
- characters
- boolean

Integers

Integers types are used to store the whole numbers in programming. The most commonly used one is probably `int`. For integers, both **signed** and **unsigned** versions are provided, where the former allows negative values.

Table 1: Integer Types Table

Types	Size (bytes)	Range
`short`	2	-32,768 to 32,767
`unsigned short`	2	0 to 65,535
`int`	4	-2,147,483,648 to 2,147,483,647
`unsigned int`	4	0 to 4,294,967,295
`long`	8	-2^63 to 2^63-1
`unsigned long`	8	0 to 2^64-1

Warning: In general, the sizes of integer types are platforms and compilers depended. The above information is for GCC with 64bit machines (as our docker image). On some old machines, you may find that the sizes of `int` and `long` are 2 and 4 byte, respectively. In order to have 64bit (8-byte) integer, you need `long long`.

Note: `signed` is also a keyword in C++, but using it is optional, i.e. `signed int` is identical to `int`.

Characters

The keyword to store character data is `char`, whose size must be 1 byte. `char` ranges from $[-128, 127]$ and $[0, 255]$ for `unsigned char`.

Boolean

A `bool` is used to store logical values, i.e. either `true` or `false`. Typically, the size of `bool` is 1 byte (in theory, we only need 1 bit).

Warning: `signed` and `unsigned` are not applicable to `bool`

Floating Numbers

Clearly, integers are not enough! Especially in scientific computing, we need real numbers that can store data from our models. In C++, the concept of *floating numbers* is used to represent real numbers. Like the *integer* table, the following is the table of floating numbers

Table 2: Floating Numbers Table

Types	Size (bytes)	Range
`float`	4	$1.18e^{-38}$ to $3.4e^{38}$
`double`	8	$3.36e^{-308}$ to $1.8e^{308}$
`long double`	ID	ID

where *ID* stands for *implementation depended*.

Note: The size and range of `long double` is implementation depended. The size may be 8, 12, or 16 bytes depending on different compilers. In general, `long double` is not a commonly used type.

Precision

The fact is that floating numbers, in general, cannot represent real numbers **exactly**. This is particularly true for irrational numbers, i.e. $\sqrt{2}$, π, etc. We refer `float` as *single-precision format*[1] while *double-precision format*[2] for `double`.

Table 3: Floating Numbers Precision Table

Precision	*Significant digits*[3]
single-precision	typically 7
double-precision	typically 15

For instance, given two real numbers 1.1 and 1.100000004, which are, of course, different numbers in the exact arithmetic setting. However, under single-precision format, they are equal. What about double-precision? Checkout notebook precision.

Double-precision format is about twice accurate than single-precision, and has a much wider range. **Question:** what are the points of using single-precision numbers?

[1] please read Wikipedia page for more
[2] please read Wikipedia page for more
[3] please read Wikipedia page for more

5.1.2 The `string` Type

`string` is also an important type in all programming language. With standard C++, `string` is not a built-in type, it's defined in the standard library `<string>`. Therefore, including the interface is needed for using strings.

```cpp
#include <string>

std::string name = "John";
std::string age = "32";
```

Note: If you are familiar with C, there is so-called C-string type, which is an *array* of characters, i.e. `char`.

5.1.3 Literals

Literals are constant values of any programs. In C++ (almost all other languages), there are five types:

1. integer literals,
2. floating point literals,
3. character literals,
4. boolean literals, and
5. string literals.

Each literal has its own **form** and **type**. Notice that literals are commonly used in *initializing* variables.

Integer Literals

Examples of integer literals are:

```
32, 1, -2, -100, 30001, ...
```

Their **types** are `int`. Then, how to specify literals of other integer types? You need suffix `u` and `l` (ell), where the former represents `unsigned` and the latter is for `long` types:

```
32l     // long literal
32u     // unsigned int literal
32ul    // unsigned long literal
```

Note: There does not exist numeric literals for `short` and `unsigned short`.

Floating Point Literals

The following forms:

```
1.0, 2.0, 3.0, 4.0, ...
```

are all double-precision floating number literals. Suffix `f` is used to denote single-precision, i.e. `float`.

```
1.0f    // float 1
-2.0f   // float -1
-5.21f  // float -5.21
```

You can also use scientific notations:

```
1.0e0   // double, 1x10^0, i.e. 1
2.0e-3  // double, 2x10^-3, i.e. 0.002, or
2e-3

1.e10f      // float, 1x10^10
5.32001e3f  // float, 5320.01
```

Character Literals

For character literals, use the single quotations:

```
'a'     // character a
'A'     // character A
'7'     // character 7
```

Boolean Literals

C++ uses `true` and `false` for Logical literals.

String Literals

For strings, C++ uses double quotations, for instance:

```
"Hello World!"      // string value of Hello World!
"AMS 562"           // string of AMS 562
```

A string is a sequence of characters.

Warning: In Python, single quotations can be used for strings, i.e. see the *Hello World* example. However, this rule cannot be applied to C++, i.e. `'abc'` is referred as multi-character literal that has type of `int` instead of `char` and the value is ID.

Escape Sequences

Questions: How to use literals to represent string `"A"` and character `'` with the quotation marks?

Such special characters are so-called *escape sequences* and start with backslash. Commonly used ones are:

Table 4: Commonly Used Escape Sequences (ES)[4]

ES	Description
`\'`	single quote
`\"`	double quote
`\\`	backslash
`\n`	**new line**
`\t`	horizontal tab
`\v`	vertical tab

Now let's consider the following string literals with escape sequences:

```
"Hello\nWorld!"          // Hello<new line>World!
"Hello\tWorld!"          // Hello<tab>World!
"\"Hello World\""        // "Hello World!"
```

5.1.4 Define & Initialize Variables

At the beginning of this lecture, we have showed an *Python program* to demonstrate one of the major differences between C++ and dynamic language. In C++, you must to explicitly construct variables with their types given. The format is `[type] var;`, where `[type]` is legal types, e.g. `int`, `double`, `std::string`, etc.

Define Variables

Here are some examples of defining variables:

```
int a;              // define an integer with var name a
double tol;         // define a double with var name tol
std::string addr;   // define a string with var name addr, req <string>
```

Warning: Once a variable name is been occupied, you cannot reuse it for anything else (within the same *scope*).

[4] Check cppreference page for more.

Initialize Variable Values

It's good practice to initialize a variable while defining it.

```
unsigned long size = 100000000000ul;     // a huge size
float error = 0.0f;                      // initialize to 0
std::string filename = "input.txt";      // a string of filename
```

Checkout notebook types and run it.

Note: One should not expect any default behaviors of uninitialized variables, e.g. when you write `int a;`, `a` might be zero but you should not assume this!

Type Conversions

Let's take a look at the following seemingly trivial code:

```
double two = 2;
```

It defines a double-precision floating point number `two` and initializes it to 2. However, recall that each literal has its own *type*, which means the above code assigns a double precision number with an integer. This is called type conversion in C++.

Type Conversions Between Integers

In general, type conversions between integers are simply copying the values. However, keep in mind that all integer types have their ranges. Converting from larger size types to smaller ones may potentially cause troubles, i.e. integer *overflow* and *underflow*.

```
unsigned int wha = -1; // What the value of wha??
```

Typically, the issues come from converting between signed and unsigned integers. Let's take a look at the *code* above. It tries to convert `int` value -1 to `unsigned int` variable `wha`.

Note: You can consider each integer type form a cyclic list that `MAX+1` is its `MIN` and `MIN-1` is the `MAX`.

As a result, the actual value of `wha` is 4,294,967,295. Checkout and run notebook conv.

Warning: Unless you 100% know what you are doing, converting between signed and unsigned integers should be avoid!

Converting Floating Point Numbers to Integers

The rule for converting floating numbers to integers is to truncate them into whole numbers.

```
int a = 12.03;     // a is 12
int b = -1.234e2; // b is -123
```

5.1.5 The `const` Specifier

`const` is a keyword in C++ that indicates an variable is immutable. Once a variable is defined as constant, you cannot modify its value, **so initialization is must for defining constant variables!**

```
const int a = 4;     // define a constant integer of value 4
// a = 2;            // ERROR! you cannot modify constant vars
// const double b;   // ERROR! const var must be initialized
```

Tip: Use `const` whenever possible!

5.1.6 Array

Array is one of the most basic data structures in programming. As a matter of facet, it is also the most commonly used data structure in scientific computing. An array is a sequence of objects that have the same size and type. In C++, an array can be constructed with square bracket `[N]`, where `N` is the size of array.

```
double arr[3];    // create an array of 3 doubles
int pos[5];       // an array of 5 integers
```

To initialize an array, the curly brackets `{}` are needed, e.g.

```
1 double tols[2] = {1e-4, 2e-5};  // array of two with values 1e-4 and 2e-5
2 int mappings[] = {2, 0, 1};     // array of three integers, size 3
3 // short a[];                   // ERROR! size must be provided
4 // const int b[2];              // ERROR! b must be initialized
5 int z[3] = {1};                 // partial initialization is ok
6 // int m[2] = {1,2,3};          // ERROR! exceeded the size
```

It is allowed to implicitly provide the size if you initialize the array, as shown in line 2. Also, partially initialize an array is allowed, but the right-hand side must be no larger than the actual array size. Checkout and run notebook array.

Accessing Array Elements

To access an specific element of an array, we need to use operator `[index]`.

> **Warning:** Un like Fortran and MATLAB, C++ is zero-based indexing, i.e. the first element index starts from 0 instead of 1.

```
double stdv[3];      // array of 3 doubles
stdv[0] = 1.0;       // first element 1
stdv[1] = 2.0;       // second element 2
stdv[2] = 3.0;       // last element 3
// stdv[3];          // out of bound!
```

Multidimensional Array

Multidimensional arrays are useful, for instance, a matrix can be represented as a 2D array, where the first dimension is the row size and column size for the second dimension. The concept of multidimensional array can be interpreted as *an array of arrays*.

```
1   double mat[2][2];        // array of two arrays of size 2
2   mat[0][0] = 1;
3   mat[0][1] = 2;
4   mat[1][0] = 3;
5   mat[1][1] = 4;
6   /*
7       a matrix of 2x2
8
9       | 1  2 |
10      | 3  4 |
11  */
```

What is the type of `mat`? It is an array but its elements also have type of array, i.e. `double[2]`. In line 2, `mat[0][0]` first accesses to the first element of `mat`, which is an array, say `mat0`, the accesses the first element of `mat0`, i.e. `mat0[0]`.

The `char[]`

As we already learned in section *string*, a string is just a sequence of `char`, i.e. `char[]`. Therefore, `char[]` is also called *C string*, or the native string type of C++. You can initialize a C string either using the array fashion or the *string literal* way.

```
// The follow two are identical
char str1[] = "AMS 562";
char str2[] = {'A', 'M', 'S', ' ', '5', '6', '2'};
```

5.1.7 Scope & Lifetime of Variables

We have already learned that reusing a variable name is not allowed in C++, but this rule applies to the variables with the same scope. Scope operators must appear as a pair of scope opener and scope closer, where are `{` and `}`, respectively.

```cpp
int a = 1;    // define integer a
// double a; // ERROR! reusing a within the same scope

// start a new scope
{
    double a; // OK!
}
// scope ends
```

Note: The *main* function or any other functions all have local scope.

The lifetime of a variable is associated with its scope. When it reaches the end of scope, it will become inaccessible and be popped out from the program stack.

```cpp
// start with a child scope
{
    int a;
} // a becomes invalid!

// a = 1; // ERROR! a does not exist!!

int b = 1; // define b, and its life begins

{
    double b = 2.0; // define local b
    b = 3;
    // which b?
}

b = 2;
// which b?
```

Note: Child scope overwrites its parent scopes.

5.2 Standard Input & Output

5.2.1 The `std::cout` Stream

By default, most program environments' standard output is screen. In C++, the global object `std::cout` is defined and guaranteed to be initialized at the beginning of any programs. The object itself is defined in the standard I/O library `<iostream>`, which must be included in order to perform I/O tasks.

std::cout stands for standard C output, which is a `FILE*` object in C (`sys.stdout` in Python). C++ uses the abstraction called *streams* to perform I/O operations.

Output operator << (bitwise left shift, or double less-than signs) is used to indicate "write to a streamer".

```
1  #include <iostream>  // bring in std::cout
2
3  std::cout << "Hello World!" << std::endl;
4  std::cout << "1+1=" << 2 << std::endl;
5  std::cout << "size of double is: " << sizeof(double) << std::endl;
```

Notice that `std::endl` is *manipulator* to produce a newline. Line 4 and 5 show that you can recursively write to the `cout` streamer and the output contents can be different types, e.g. in line 4, `"1+1="` is string but `2` is integer.

Note: In stead of using manipulator `std::endl`, you can also use the newline *escape sequence*—`\n`.Therefore, the outputs are identical between `std::cout << "Hello World!" << std::endl;` and `std::cout << "Hello World!\n";`.

Checkout and run notebook cout.

5.2.2 The `std::cin` Stream

Use Input Operator >>

The default standard input for most program environments is through keyboard. In C, the `FILE*` object is `stdin` (`sys.stdin` in Python). This input streamer is able to read the user inputs from keyboard. `std::cin` stands for standard C input.

Similar to output operator, the input operator is bitwise shift right (or double greater-than signs) >>. The basic syntax is `std::cin>>var;` and the user inputs will be stored in `var`.

```
#include <iostream> // cout and cin

// best practice, always indicate the user what to enter
std::cout << "Please enter your first name: ";
std::string name;
// the program will hang here til receive the user input
std::cin >> name;
std::cout << "Hello! " << name << std::endl;
```

`std::cin` read the user inputs into its buffer, and the input operator `>>` searches the user keyboard inputs and treated them as a sequence of white space separated arguments. Therefore, if you enter your full name, e.g. "Qiao Chen", only the first value will be printed because the program only asks for one input arguments, i.e. `name`.

It's also possible to handle multiple input arguments:

```
#include <iostream> // cout and cin

// best practice, always indicate the user what to enter
std::cout << "Please enter your first and last names: ";
std::string fname, lname;
// the program will hang here til receive the user input
std::cin >> fname >> lname;
std::cout << "Hello! " << fname << ' ' << lname << std::endl;
```

Compile and run program cin_demo.

Read An Entire Line

It's also convenient to read an entire line at once. To do this, you need to use the `std::getline` function. Like input operator `>>`, *getline* treats the user inputs as a sequence of `\n` separated arguments. The syntax is: `std::getline([input streamer], string)`.

```
#include <iostream>

std::string buffer;          // create buffer
std::cout << "Please enter a sentence...\n";
std::getline(std::cin, buffer);
std::cout << "You just entered: " << buffer << '\n';
```

Warning: Special care must be taken into consideration if you want to mix the use of input operator `>>` and `std::getline`. When the user types "Hello" and press `ENTER`, the actual input value is "Hello\n".

```
std::string word, sent;
std::cout << "Enter a word:";
std::cin >> word;       // read in a word from cin
std::cout << "The word you just entered is:" << word << std::endl;
std::cout << "Enter a sentence:\n";
std::getline(std::cin, sent);
std::cout << "The sentence you entered is:\n" << sent << std::endl;
```

The above code will not work as what you expect, because after reading a word, there is still a **newline**, `\n`, character left, and this confuses the following *getline* operation. In order to skip the character `\n`, you need to call `std::cin.ignore();`. The correct program is:

```
std::string word, sent;
std::cout << "Enter a word:";
std::cin >> word;       // read in a word from cin
std::cout << "The word you just entered is:" << word << std::endl;
std::cin.ignore();      // ignore '\n'
```

(continues on next page)

(continued from previous page)

```
std::cout << "Enter a sentence:\n";
std::getline(std::cin, sent);
std::cout << "The sentence you entered is:\n" << sent << std::endl;
```

5.2.3 The `std::cerr` Stream

`std::cerr` stands for standard C error output. Its associated C `FILE*` object is `stderr` (`sys.stderr` in Python). It, like *cout*, is an output streamer. It also writes outputs on the screen.

```
#include <iostream>

std::cerr << "This is an error message!\n";
std::err << "WARNING! Converting from signed to unsigned is dangerous!\n";
```

5.2.4 `std::cout` vs. `std::cerr`

Conceptually, we understand that `std::cout` should be used for writing normal messages, e.g. logging information; `std::cerr`, on the other hand, should be used for indicating error messages. However, in terms of programming, what is the difference between these two streamers, since they seemingly both just output messages on the screen.

Before we dig into this question, we need to understand the concept *file descriptor* (FD). FD is a handle (usually non-negative integer) that uniquely indicates an open **file** object. On Linux, a file object can also be other input/output resources such as pipe and network sockets. Recall that both *cout* and *cerr* stand for *standard outputs*, the latter is specifically for error output. All these two are the default output FDs, to which a program can write outputs. On Linus, there should be three standard FDs:

Table 5: Standard File Descriptors

Streams	File Types	FD handles
`std::cin`	standard input	0
`std::cout`	standard output	1
`std::cerr`	standard error	2

Consider the following program:

```
#include <iostream>

int main() {
  // write a message to stdout
  std::cout << "This is from cout\n";
  // write something to stderr
  std::cerr << "This is from cerr\n";
  return 0;
}
```

and you can download it program cout_vs_cerr.

Inside a terminal, compile the program:

```
$ g++ cout_vs_cerr.cpp
```

Let's first run the program normally

```
$ ./a.out
This is from cout
This is from cerr
```

Both of "This is from cout" and "This is from cerr" are printed on the screen. Bash allows you to *redirect* standard outputs by using > when you run any programs. Now rerun the program:

```
$ ./a.out >cout.txt
This is from cerr
```

Only "This is from cerr" is shown on the screen, the output that was written to `cout` had been redirected to the file "cout.txt". Invoke the built-in commands `ls` and `cat` to list *cwd* and print out the content of a file.

```
$ ls
a.out  cout.txt  cout_vs_cerr.cpp
$ cat cout.txt
This is from cout
```

In addition, you redirect a specific file descriptor by adding the FD handle in front of >. Finally rerun the program:

```
$ ./a.out 1>cout.txt 2>cerr.txt
$ ls
a.out  cerr.txt  cout.txt  cout_vs_cerr.cpp
$ cat cout.txt cerr.txt
This is from cout
This is from cerr
```

This time, both *cout* and *cerr* wrote to files "cout.txt" and "cerr.txt", respectively. In practice, this allows you easily to group the program outputs into normal progress logging information and error/warning information. For instance:

```
$ ./my_prog 1>prog.log 2>error.log
```

If something goes wrong, the user can always trace back in "error.log" given the assumption that your program writes error/warning messages to `std::cerr`.

Tip: `&>file.txt` can redirect both *stdout* and *stderr* to "file.txt". For instance, `./a.out &>output.log`.

CHAPTER

SIX

LECTURE 2: REFERENCES, POINTERS & DYNAMIC ARRAY

Table of Contents

6.1 Understanding References in C++

A reference is an **alternative name** of another object in C++. The syntax is adding symbol `&` after the type identifier (declarator), i.e. `[type] &var`.

6.1.1 Initializing Variables vs. Initializing References

In lecture 1 *Define & Initialize Variables*, we have learned how to initialize a variable, e.g.

```
int a = 1;  // define and initialize integer a with value 1
```

When we initialize a variable, the initializer (right-hand side) will be copied to the variable. The above code is to copy the right-hand side, which is 1, to variable `a`. Intuitively, recopying values is allowed for variables, so that you can write:

```
int a;  // define a
a = 2;  // recopy value 2
```

However, for references, we bind, instead of copy, them to their initializers. Each reference is bound to its initial object, and rebinding it is not allowed.

Warning: All references must be defined with initializers!

Note: A reference is just an alias of an object!

```
int obj = 1;         // integer of value 1
int &ref = obj;      // bind reference ref to obj
```

For the code above, if we modify the value of `obj`, say `obj=2;`, what will be the value of `ref`?

Checkout notebook ref and run it.

Note: Modifying a reference will affect the object that it bind to. Essentially, the behavior of an object and its references is synchronized.

Due to the fact that a reference can be used to modify the values of its original object, **binding a (normal) reference to a constant object is not allowed!**

```
const double tol = 1e-2;
double &tol_ref = tol; // error!
// similarly, a literal is considered to be constant object
char &A = 'A'; // error!
```

6.1.2 References with `const`-Qualifier

You can bind a constant object with **constant reference**. Because a reference itself already has the property of `const`-ness, i.e. you cannot rebind a reference, **constant reference** can also be used to bind temporary variables (future).

```
const float alpha = 1.0f;
const float &alpha_ref = alpha;  // ok!
const std::string &str_ref = "ams562"; // ok!
```

Note: You can bind constant references to normal objects.

Checkout and run notebook const_ref.

6.2 Understanding Pointers in C++

Pointer is probably one of the most difficult concept to understand in C and C++. Before we jump into pointers, we need to first understand the memory addresses in C++.

An **object** has its unique memory address. A pointer is a special type of **objects** that can hold memory addresses as its values.

To define a pointer, we need to add symbol `*` after the type identifier, i.e. `[type] *ptr`.

```
int *ptr;    // define a pointer
```

6.2.1 The `address-of` Operator

To extract the memory address of an object, we need to use the `address-of` operator, i.e. `&`.

Note: Do not get confused with the **symbol** `&` used in defining *references*, since it appears after the **type** identifier. The `address-of` operator is used in front of **variables**.

```
1  int a = 1;           // define an integer
2  int *a_ptr = &a;     // define a pointer that points to a's address
```

In line 2, the `address-of` operator is used in order to extract the memory address of `a`, and the value (of the memory address) is assigned to the pointer `a_ptr`. Moreover, `a_ptr` is defined as a pointer that points an integer object, typically, you cannot mix the pointer type and object type.

Warning: `double *ptr = &a` is not allowed with the code above!

6.2.2 The `dereference` Operator

Accessing objects is a typical usage of pointers. To do so, we need the `dereference` operator—`*`.

Note: Similarly, do not confuse with `dereference` operator and the symbol `*` for defining pointers.

```
int a = 1;
int *a_ptr = &a;  // copy a's address
std::cout << a_ptr << '\n'; // print a's address
std::cout << a << "==" << *a_ptr << '\n';
*a_ptr = 2;
// what is a?
```

Checkout and run notebook ptr.

6.2.3 Initialize Pointers

A **null pointer** points to a special memory address that indicates empty object. Typically, you should initialize a pointer with **null** if you don't know what addresses it needs to take. In C++, we can use (and should use) `nullptr` for **null pointer**. Some programs prefer to use `0` and the traditional `NULL` (from C, requiring `<cstdlib>` interface).

```
double *ptr1 = nullptr;  // C++ preferred
double *ptr2 = 0;        // equiv way
double *ptr3 = NULL;     // require <cstdlib>
```

Warning: It is legal to **initialize** a pointer with value `0` (null). However, you cannot reset a pointer with integer object with value zero.

```
double *ptr = 0;     // fine!
ptr = 0;             // ok!
int zero = 0;
ptr = zero;          // ERROR! cannot assign double * with int
```

Tip: Always use `nullptr` in C++.

Warning: Uninitialized pointers are extremely dangerous and using them is one of the typical error sources.

```
int *p1;             // uninitialized
int *p2 =nullptr;    // initialized but points to null
int a;
int *p3 = &a;
```

Danger: You cannot deference `nullptr`, since it will cause `segmentation fault`, which is a critical memory bug that will immediately abort your programs.

```
int *a = nullptr;
*a = 1;   // Seg fault! program crushes!!
```

6.2.4 References vs. Pointers

There are some similarities between *references* and pointers. An obvious one is that you can use both to access and modify an object.

```
double tol = 1e-6;
double &tol_ref = tol;
double *tol_ptr = &tol;
```

(continues on next page)

(continued from previous page)

```
tol_ref = 1e-2;
std::cout << tol << std::endl; // what is the output?
std::cout << *tol_ptr << std::endl; // what about this?
*tol_ptr = 0.0;
std::cout << tol_ref << std::endl; // this?
```

However, there is one fundamental difference: **references are not an object!** This means that you cannot get the memory addresses of references (well, technically the memory address of a reference is that of the object it refers to, since a reference is just an alias of an object.)

A pointer, on the other hand, is an object in C++ thus having its own memory address. You can also refer to pointers through the reference semantic.

```
int a;
int b;
int *p = &a;     // p holds a's addr
int *&p_ref = p; // a reference of pointer
p_ref = &b;      // what is p now?

// similarly, since p is object, we can create pointers
// that point to it
int **pp = &p;   // pp holds p's address
std::cout << *pp << '\n'; // what is the deref of pp
*pp = &a;
// what is p_ref now?
```

Try the notebook ptr_ref.

6.2.5 Pointers with const-Qualifier

Unlike *references*, pointers are normal objects, so there is a difference between *pointers to constants* and *constant pointers*.

A pointer to constant is that the pointer itself is a normal object but points to a constant object, i.e. you cannot modify the underlying object. However, you can modify the pointer, e.g. assign another memory address.

```
const int a = 1;
const int *p2c = &a;
*p2c = 2;    // ERROR! a is constant
p2c = nullptr; // fine
```

A constant pointer is that the pointer itself is constant object, i.e. you cannot modify the memory address. But you can still dereference it to modify the object that it points to.

```
int a;
int *const p = &a; // you must initialize p, why?
*p = 1; // fine
p = nullptr; // ERROR!
```

Of course, you can have *constant pointers to constant*, i.e.

```
int a;
const int *const p = &a;
```

(continues on next page)

(continued from previous page)

```
*p = 2; // ERROR!
p = nullptr; // ERROR!
```

Tip: Read a declaration from right to left.

Note: Pointers to constant are widely used in practice.

6.3 Dynamic Memory Allocation/Deallocation

6.3.1 Stack Memory vs. Heap Memory

The stack memory is fast but limited. In general, the stack is fixed. Ideally, we want to use the stack memory, because it is directly accessible to the CPU stack registers and the operating system can even directly allocate the stack in the cache. The fact is that all variables are constructed in the stack memory.

Note: All arrays are stored in the stack.

However, due to the limited size, we can easily get ourselves into overflow.

```
double huge_data[extremely_large_size]; // this will break!
```

An easier way to understand this limitation is to look at the following *recursive function*:

```
void never_call(int i) {
    int j = 1;
    int k = i;
    std::cout << k << std::endl;
    never_call(j+k); // recursively call "myself"
}
```

Each time, when the function `never_call` is invoked, three variables will be created—`i` (local copy), `j` and `k`. Due to the recursive mechanism, all the variables will live in the stack thus resulting `segmentation fault` eventually because of stack size overflow.

The **heap memory**, on the other side, is, loosely speaking, the left-over memory in the RAM after 1) your program is loaded and 2) the memory is allocated. Unlike the stack memory, where all variables are stored, the space we request in the heap is not directly appeared in the program, we need to create a *pointer* that points to the leading (usually) memory address of that chuck of memory space.

Note: We typically refer variables that created in the stack as the *meta data* that describes the actual data, which is typically large and stored in the heap.

The heap memory is used for *dynamic memory allocation* (a.k.a runtime memory allocation), which is usually used in the following two situations:

1. the data size is large, and
2. the data size is unknown and dynamically changing.

For the second one, a typical situation is that we you write a word processing program, you don't know how many characters the user may use, so a memory space that can grow dynamically is must.

Tip: For some applications, even though the data size cannot be determined beforehand, but the upper bound can be precisely estimated. In this case, if the upper bound is small, then you should use the stack memory! One example would be create a *C-string* to store the user input filename, i.e. `char filename_buffer[200]`.

6.3.2 Dynamic Memory Allocation

Since we already learned that we need to use *pointers* to point to the memory locations in the head, you should not be surprised that dynamic memory allocations involve using pointers. The syntax is `[type] *ptr = new [type]`, where the operator `new` is to allocate memory dynamically.

```
int *bad_ptr = 3; // ERROR!
int *ptr = new ptr;  // request a valid place first
*ptr = 3;    // dereferencing a valid pointer is fine
// or
int *ptr_init = new int (3);
```

6.3.3 Dynamic Memory Deallocation

As we already learned in the *scope*, all variables have the lifetime that is bounded by the scope, i.e.

```
{   // scope begins
    int a; // push a into the stack
}   // scope ends, a is popped
```

Does this rule applied for dynamic memory in the heap? Recall that a *Understanding Pointers in C++* is also a variable.

```
1 {   // scope begins
2     int *ptr = new int;  // allocate a dynamic chuck
3 }   // scope ends, ptr is popped
```

In line 3, once the scope ends, `ptr` will be popped out and no long visible, what about the dynamic memory it used to point at?

Warning: Dynamic memory will not be automatically cleaned up!

Actually, the code above is extremely dangerous, because once `ptr` is popped out, it's impossible for you to access to the dynamic memory space that `ptr` used to point at. People refer this as *memory leaks*.

To relax a dynamic allocation, we need the deallocation operator `delete`, the syntax is `delete [ptr];`.

```
{
    int * ptr = new int;
    delete ptr;  // freed here!
}   // no leak!
```

Important: There must be exactly a `delete` that matches to a `new`.

```
double *p = new double;  // allocate here
p = new double;      // reallocate here, previous allocation leaked!
delete p;   // the first double is leaked.
// delete p; // delete twice won't work and dangerous!
```

6.3.4 Dynamic Array Allocation/Deallocation

A common use of dynamic memory allocation is to request a large chunk of contiguous memory space, i.e. an array, during runtime. For this task, should use operator `new[]` and relax the dynamic array with operator `delete[]`.

```
double *data;        // create the meta data
unsigned long HUGE = ...;
data = new double [HUGE]; // request the dynamic array of size HUGE
// do work with data
// don't forget to relax the memory
delete[] data;
```

Note: A *pointer* can be accessed like an *array* using operator `[]`. For instance, given the example above, you can do `data[0]` to access the first element in the dynamic array, and `data[n]` for the n-th one.

A common mistake is mixing `new`/`new[]` with `delete`/`delete[]`.

Warning: `new` must be coupled with `delete`, and the same rule applies for `new[]` and `delete[]`.

```
int *p1 = new int;
delete [] p1; // WRONG!
double *p2 = new double [2];
delete p2; // WRONG!
```

Try the notebook dyn.

Note: With modern C++, you really don't need to worry that much about managing dynamic memory. In the future lectures, we will learn using `vector` as well as the so-called *smart pointers*.

6.4 Defining Multiple Variables

Now, with the knowledge of compound types, i.e. *pointers* and *references*, I think it's a good time for you to understand a tricky part in C++—defining multiple variables.

You can define multiple variables like:

```
int i, j;  // define two variables without initialization
int k=0, t; // define another two, initialize k
float x, y=1.0f; // only initialize the second one
std::string depart("ams"), course("562"); // initialize both
```

This is pretty intuitive. Now let's say I want create two points:

```
int *ptr1, ptr2;
ptr2 = nullptr; // ERROR!!
```

This is because compound types have the local property. Therefore, `ptr2` is actually just an integer.

Now, try to figure out the types of the following variables.

```
int a=0,*b=0,&c=a,**d=&b,&e=c,**&f=d,g=e; // read from right to left
```

CHAPTER

SEVEN

LECTURE 3: EXPRESSIONS & STATEMENTS

Table of Contents

7.1 Operators & Operations

7.1.1 Elementary Arithmetic Operators

For most *built-in types*, we have elementary operations such as *addition*, *subtraction*, *multiplication*, *division*, and *modulus*.

1. `+`: the addition operator
2. `-`: the subtraction operator
3. `*`: the multiplication operator
4. `/`: the division operator
5. `%`: the modulus operator

Note: 1 and 2 can be also used as unary operators, i.e. represent *positive* and *negative*, respectively.

Be aware that the modulus operator only works for integers in C++, this is unlike Python, where the operator is also applicable to floating numbers.

```
int a = 5;
std::cout << "mod(5,2)=" << 5%2;

// std::cout << 5.0%2; // ERROR! % is not defined for floating numbers
```

Also, for the division operator, the behaviors for integers and floating numbers are different.

```
std::cout << "5/2=" << 5/2; // this is 2
std::cout << "5.0/2=" << 5./2; // this is 2.5
```

The resulting integers from dividing integers will be truncated. This is called *integer division*.

Note: The division is treated as *integer division* iff both left- and right-hand sides are integer types.

7.1.2 Assignment Operators

So far, we have frequently been using the assignment operator, i.e. =. There are other types of assignment operators in C++.

1. +=: plus assignment
2. -=: minus assignment
3. *=: product assignment
4. /=: quotient assignment
5. %=: remainder assignment

There are so-called *compound assignment operators*, and you under these by the following expression: A?=B for ? ∈ {+,-,*,/,%} is equivalent to A=A?B.

7.1.3 Increment & Decrement Operators

In C++, we also have increment and decrement operators for integers to increase or decrease their values by 1.

1. ++: increment operator
2. --: decrement operator

Use as Suffix

These operators can be used as suffixes, e.g.

```
int a = 0;
a++;
std::cout << a; // print out 1
```

Post- increment/decrement operators modify the value of the target object after processing the current statement.

```
int a = 0;
int b = a++;
std::cout << "a=" << a;
std::cout << "b=" << b;
// what is a? what is b? try this!
```

Use as Prefix

These operators, of course, can be used as prefixes, e.g.

```
int a = 0;
--a;
std::cout << a; // print out -1
```

Pre- increment/decrement operators modify the value of the target object before processing the current statement.

```
int a = 0;
int b = --a;
std::cout << "a=" << a;
std::cout << "b=" << b;
// what is a? what is b? try this!
```

An Exercise

Take a look at the following program:

```
int i1 = 1;
int i2 = ++i1;
int i3 = ++ ++i1;
int i4 = i1++;
// we cannot do i1++ ++
std::cout << "i1 = " << i1 << "\n"
          << "i2 = " << i2 << "\n"
          << "i3 = " << i3 << "\n"
          << "i4 = " << i4 << "\n";
```

Run this example in notebook inc_dec.

7.1.4 Comparison & Logical Operations

To compare two values, you need to use comparison operators in C++:

1. <: strictly less than
2. >: strictly greater than
3. <=: less than or equal to
4. >=: greater than or equal to
5. ==: equal to
6. `!=`: not equal to

The resulting object of the comparison operators are boolean flags, i.e. either `true` or `false`.

For logical operators, we have:

1. `&&`: logical and
2. `||`: logical or
3. `!`: logical not (use as unary operator)
4. ^: logical xor

Note: Technically speaking, ^ is a bitwise operator not a logical operator. However, since `true` can be converted into integer 1 and 0 for `false`, and `1^0=1`, `1^1=0`, and `0^0=0`, which behave exactly like an xor operation.

```
std::cout << "1<2 and 2<3 is: " << (1<2 && 2<3);
std::cout << "3==4 || 3==6/2 is: " << (3==4 || 3==6/2);
std::cout << "not 1.0<0.0 is: " << (!(1.0<0.0));
std::cout << "either 10>2 or 5>2 but not both: " << ((10>2)^(5>2));
```

Try out comparison and logical operators in notebook logi_comp.

7.1.5 Common Mathematical Operations

As applied scientists who do programming, there is not doubt that using common mathematical functions is necessary. Unlike MATLAB, in which the common mathematical functions are defined as built-ins, C++ doesn't know how to do math by default (sigh…). Fortunately, the *standard math library* provides most common mathematical operations that can potentially become very handy.

```
#include <cmath>  // standard math interface

...

std::cout << "sin(pi) is: " << std::sin(M_PI);
std::cout << "cos(0) is: " << std::cos(0.0);
std::cout << "arcsin(1) is: " << std::asin(1.0);
std::cout << "log 2 of base 2 is: " << std::log2(2);
std::cout << "|-1| is: " << std::abs(-1);
```

For a complete list and the usage, please refer to Standard library header <cmath>.

Try out this notebook math.

7.1.6 Pointer Arithmetic

Again, pointers… C++ allows you to perform arithmetic operations on pointers. Of course, the means are different from *the elementary arithmetic operations*. You can add or subtract an integer from a pointer, e.g.

```
1 int obj[2];
2 int *ptr = obj; // equiv to ptr = &obj[0]
3 int *ptr_next = ptr+1;
4 int *ptr_ori = ptr_next-1;
```

Here, adding one to `ptr` meaning that advance `ptr` to next memory address thus resulting the next adjacent pointer, i.e. `ptr_next`. Similarly, in line 4, subtracting one from a pointer meaning that move the pointer to its previous adjacent position, i.e. `ptr_ori`.

Note: Adding/subtracting integers from a pointer result another pointer.

You can also subtract two pointers, e.g. `ptr1-ptr2`. The result is an integer that represents the *signed distance* between the two pointers.

```
float a[2];
float *ptr_a = a, *ptr_b = ptr_a+1;
std::cout << "distance in memory between a[0] and a[1] is: " << (ptr_a-ptr_
↪b); // -1
```

Increment and decrement operators are also applicable to pointers, i.e.

```
1 int arr[5] = {1,2,3,4,5};  // an array of length 5
2 int *ptr = arr;
3 std::cout << *ptr; // 1
4 std::cout << *ptr++; // 1, why??
5 std::cout << *++ptr; // 3
6 std::cout << *ptr--; // 3, where is ptr now?
```

Let's take a look at the code above. Line 1 and 2 define an array of size 5 and initialize its value to `{1,2,3,4,5}`, then define a pointer `ptr` that points to the array.

Line 3 simply shows the value of `arr[0]` by deferencing `ptr`. Line 4 will first prints the value of `*ptr` then advances `ptr` to its nest position, i.e. `arr[1]`. Line 5 first advances `ptr` to `arr[2]` and displays its value.

For more, take a look at this notebook ptr_arith.

7.2 Control Statements

A statement in C++, roughly speaking, is a single line of code that ends with semicolon, i.e. `;`, which can be executed by the program.

Control statements are special statements in C++ (or any other programming languages) that control how/whether other statements will be executed. With *my fundamental assumption*, I will not go detail in the concept of control statements. I will mainly focus on introducing the syntax and giving examples.

Note: It's probably a good idea to review the concept of *scope* in C++.

In general, control statements in C++ can be put into three families:

1. Conditional statements
2. Loop statements
3. Jump statements

7.2.1 Conditional Statements

C++ provides two control statements to perform conditional executions.

The `if` Statement

An `if` statement conditionally execute another statement based on whether or not a specified condition is `true`.

```
if (<condition>) {
    // do things if <condition> is true
}
```

For example:

```
1  std::string dep;
2  std::cout << "enter the department:";
3  std::cin >> dep;
4  if (dep == "ams") {
5      std::string course;
6      std::cout << "enter the class number:";
7      std::cin >> course;
8      std::cout << "welcome to ams" << course << std::endl;
9  }
```

Line 4-9 will only be executed if the input `dep` is `"ams"`. Compile and run this program.

Of course, you can add `else` so that the condition is complete, the syntax is:

```
if (<condition>) {
    // do things if <condition> is true
} else {
    // do things if not <condition>
}
```

For example:

```
unsigned n;
std::cout << "enter a non-negative whole number:";
std::cin >> n;
std::string odd_or_even;
if (n%2) {
    odd_or_even = "odd";
} else {
    odd_or_even = "even";
}
std::cout << "you just entered an " << odd_or_even << " number\n";
```

Multiple (more than 2) condition branches are supported with `else if` statement, the syntax is:

```
if (<condition1>) {
    // do things if <condition1> is true
} else if (<condition2>) {
    // do things if <condition2> is true
} else if (<condition3>) {
    // do things if <condition3> is true
} else {
    // ow
}
```

Note: Multiple condition branches are executed in sequential order, and the statement will terminate til reach the first `true` case of the end of the `if` statement.

Note: A `if-else if-else if-...-else` is considered as a **single** statement!

Let's take a look at the following two different programs:

```
const int n = 2;
int a;
if (n == 2) {
    a = 100;
} else if (n % 2 == 0) {
    a = 200;
} else {
    a = 300;
}
std::cout << "a=" << a << std::endl;
```

What is the value of `a`?

```
const int n = 2;
int a;
if (n == 2) {
    a = 100;
}
if (n%2 == 0) {
    a = 200;
}
if (n%2) {
    a = 300;
}
std::cout << "a=" << a << std::endl;
```

What about this one?

Command Line Inputs

So far, our *main* functions are defined without any input arguments. However, it's common to have `argc` and `argv` as the function input parameters, i.e.

```
// main.cpp
int main(int argc, char *argv[]) {
    return 0;
}
```

Where `argc` is an integer and `argv` is an *array* of *C-strings*. So what are the meanings of these variables? `argc` indicates the number of input arguments from command line when the program is executed, and `argv` stores their values in raw strings.

For instance, you can *compile* the program into an executable binary `a.out` by:

```
$ g++ -std=c++11 main.cpp
```

Then, you can run the program by:

```
$ ./a.out
```

Which means the command line arguments is `./a.out`. In the program, `argc` is 1 and the first element (C-string) in `argv`, i.e. `argv[0]`, stores the name of the executable—`./a.out`.

Note: All programs have at least one command line argument, which stores their names.

If you type:

```
$ ./a.out 1 2 3 abc
```

The program's `argc` is 5 with `argv={"./a.out","1","2","3","abc"}`.

The functionality of command line inputs is important, because it enables *batch processing* with user inputs.

Note: In most cases, interactive inputs, e.g. through `std::cin` and keyboards, are not possible, especially for scientific computing where programs usually run on clusters.

Now, combining this information with `if` statement, you can parse the user command line inputs. In addition, it's common that you want the user to pass some numerical values, so converting from C-strings to integral/floating numbers is necessary. This can be done with the functions `std::atoi` and `std::atof` that are defined in official `<cstdlib>` library.

```
# include <cstdlib>

...

const char * i_str = "4";
const char * f_str = "1e-1";
const int i = std::atoi(i_str);
const double f = std::atof(f_str);
// i is 4 and f is 0.1
```

Hint: Check the number of input arguments by `if (argc < 3) ...`

Compile and run program cmd_inputs.

Writing Efficient `if` Statement

Make the condition branches that have large probability taking higher priority.

```
unsigned n;
std::cin >> n;
const unsigned rem = n%100;

// prefer

if (rem != 0) {
    // do work I
} else {
    // do work II
}

// over

if (rem == 0) {
    // do work II
} else {
    // do work I
}
```

The `switch` Statement

Another conditional statement is `switch`, which can be used to choose one of the several integral expressions. Let's take a look at the following example with a `char` as our integral expression.

```
char c;
std::cin >> c;
bool is_vowel = false;
switch (c) {
case 'a':
    is_vowel = true;
    break;
case 'e':
    is_vowel = true;
    break;
case 'i':
    is_vowel = true;
    break;
case 'o':
    is_vowel = true;
    break;
case 'u':
    is_vowel = true;
    break;
}
if (is_vowel) {
    // do something
}
```

Each `case` is an entry point of the corresponding `switch` statement, and **the statement will not terminate until it reaches the first break or the end of the statement**. Therefore, the code above is equivalent to:

```cpp
char c;
std::cin >> c;
bool is_vowel = false;
switch (c) {
case 'a':
case 'e':
case 'i':
case 'o':
case 'u':
    is_vowel = true;
    break;
}
if (is_vowel) {
    // do something
}
```

Warning: Missing `break` is a common bug in one's programs.

To make a complete condition, you need to use `default`, which indicates the default behavior. The `is_vowel` example can also be written as:

```cpp
char c;
std::cin >> c;
bool is_vowel;
switch (c) {
case 'a':
case 'e':
case 'i':
case 'o':
case 'u':
    is_vowel = true;
    break; // missing this will make is_vowel always false
default:
    is_vowel = false;
    break;
}
if (is_vowel) {
    // do something
}
```

Play around with this notebook switch.

7.2.2 Loop Statements

Loop statements allow you to repeatedly execute some statements that follow same/similar structure.

The `for` Loop

The `for` loop in C++ has the syntactic form:

```
for (<init statement>; <condition statement>; <express>) {
    // do work
}
```

Each of the three blocks is separated by semicolon. The `init statement` will be invoked once. Here is what happens under the hood:

```
for (<init statement>; <condition statement>; <express>) {
    // if <init statement> has not been invoked, do it
    // if <condition statement> fails, stop

    // do work

    // invoke <express>
}
```

With for loop, we can perform some simple operations with arrays. For instance, you can initialize an array given the size is unknown.

```
int N;
std::cin >> N;
double *data = new double [N];
for (int i = 0; i < N; ++i) {
    data[i] = 1.0;
}
delete [] data;
```

Or accumulate the value:

```
double sum(0.0);
for (int i = 0; i < N; ++i) {
    sum += data[i];
}
std::cout << "sum of data is: " << sum;
std::cout << "average is: " << sum/N; // assume N>0
```

Note: The counter `i` in the examples above has **local scope**.

You can, of course, define the counter out of the `for` loop:

```
int i;
for (i = 0; i < N; ++i) {
    // do work
}

```

(continues on next page)

(continued from previous page)

```
6  // or
7
8  int j = 0;
9  for (; j < N; ++j) {
10     // do work
11 }
12
13 // or
14 int k = 0;
15 for (; k < N; ) {
16     // do work
17     ++k;
18 }
```

In fact, all three blocks can be empty (like line 9 and 15), even at the same time (non-stopping loop), i.e.

```
for (;;) {}  // non-stopping...
```

C++ doesn't restrict that the loop counter must be integers. Pointers are another common counter.

```
for (double *data_ptr = data; data_ptr < data+N; ++data_ptr) {
    *data_ptr = 1.0;
}
```

is equivalent to the first example in `for` loop section.

Warning: Looping over floating numbers is not recommended and should be avoided, because it's expansive and problematic due to rounding errors.

```
// this is not preferred

const double h = 1./3;
double acc = 0.0;
for (; acc < 100.0; acc += h) {
    // do work with acc
}

// do this instead

for (int i = 0; i < 300; ++i) {
    acc += h;
    // do work with acc
}
```

Implement a Forward Linked List with `struct`

In C++, it's common that you want to group some data into a structure, in this case, you can use `struct`. The syntactic form is:

```
struct StructName {
    // any attributes
}; // <-- don't forget the semicolon
```

For instance:

```
// you can represent a complex number by the following structure
struct ComplexNumber {
    double real;
    double imag;
};
```

Now, `ComplexNumber` is a customized type that is defined by the user. To access an element/member in the structure, you need to use the accessing operator `.`, i.e.

```
ComplexNumber a, b; // two complex number
a.real = 1.0;
a.imag = -1.0;
b.real = 2.0;
b.imag = 1.0;
```

Warning: `ComplexNumber` is not a built-in type, so you should not expect those arithmetic operations can be applied to it without any additional efforts. However, both `real` and `imag` are just `double`.

You can, of course, define pointers that have base type `ComplexNumber`.

```
ComplexNumber *ptr_a = &a, *ptr_b = &b;
```

To access the elements/members of a structure through its pointers, the accessing operator `->` is needed, i.e.

```
std::cout << "complex a=(" << ptr_a->real << ',' << ptr_a->imag << ")\n";
// this prints out a=(1.0,-1.0)
```

Dynamic memory allocation is also applicable.

```
ComplexNumber *ptr = new ComplexNumber;
ptr->real = 1.0;
ptr->imag = 1.0;
delete ptr;

ComplexNumber *ptr_arr = new ComplexNumber[10];
for (int i = 0; i < 10; ++i) {
    ptr_arr[i].real = 1.0;
    ptr_arr[i].imag = 2.0;
}
delete [] ptr_arr;
```

A linked list, like array, is another fundamental data structure. Unlike the array, which has contiguous memory layout, a linked list can stay in arbitrary locations in the memory, and each of its element (usually called node) points to each other through pointers. A common structure for a node of linked list:

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

Notice that `next` points to the next node in the linked list. Let's implement a forward linked list with initialization, node insertion, and finalization.

Now, open this notebook for_fll.

The `while` Loop

The `while` loop, like `for`, is another iterative statement in C++. The syntax is:

```
while (<condition>) {
    <evaluate the condition>
    // do work
    <update conditional statement>
}
```

You can easily translate a `for` loop into a `while` loop:

```
// for version
for (int i = 0; i < n; ++i) {
    std::cout << "i=" << i << ' ';
}

// while version
int i = 0;
while (i<n) {
    std::cout << "i=" << i << ' ';
    ++i; // what will happen if this statement is missing?
}
```

Forward Linked List with `while`

We can also implement the *forward linked list* with the `while` loop.

Open this notebook while_fll.

The `do-while` Loop

The last loop statement is so-called `do-while` in C++. The syntactic form is:

```
do {
    // do work
    <evaluate condition statement>
} while (<condition>); // <- semicolon
```

`while` vs. `do-while`

do-while guarantees that at least one statement will be evaluated, e.g.

```
// while version
while (false) {
    std::cout << "never executed\n";
}

// do-while version
do {
    std::cout << "executed!\n";
} while (false);
```

Warning: As a result, it's easy to run into infinite loops with `do-while` mechanism!

Pick the logic bugs in the following code:

```
// version 1
unsigned n = 100u;
do {
    n--;
} while (n>=0u);

// version 2
unsigned n;
std::cout << "enter an non-negative integer:";
std::cin >> n;
do {
    n--;
} while (n>0u);
```

7.2.3 Jump Statements

The laster family of control statement is the *jump* statement. This is mainly used to `continue` a loop statement and/or `break` it given the fact that its conditional expression cannot be easily determined beforehand.

```
// skip statements if loop counter is odd
for (int i = 0; i < n; ++i) {
    if (i%2) {
        continue;
    }
    // do work
}
```

In the code above, if `i` is odd, then the statement will be skipped.

```
// break a loop if the loop counter is 5
for (int i = 0; i < n; ++i) {
    if (i == 5) {
        break;
    }
    // do work
}
```

In the code above, if `i` is 5, then the for loop will be terminated.

A practical example would be interactively talk with the user through `std::cin`.

```
std::string input;
std::string buffer;
while (true) {
    std::cout << "enter something:";
    std::cin >> input;
    if (input != "break") {
        std::cout << "you entered: "
                  << input
                  << ", the loop will continue\n";
        // clear the buffer in cin in case the user may enter more
        // than a word
        std::getline(std::cin, buffer);
        continue;
    } else {
        std::cout << "bye!\n";
        break;
    }
}
```

Play around with program ita_cin.

CHAPTER

EIGHT

LECTURE 4: FUNCTIONS

Table of Contents

8.1 The Basis

Function is an important concept in programming, because it allows us to easily modularize our programs and reuse the codes in the future. In this lecture, I will show you how to write functions in C++.

8.1.1 Fundamental Concepts of Functions in C++

Currently, we write everything inside the *main* function. This is fine for small projects, say homework assignments. However, for larger projects, this can be very limited. Consider the following situation.

```cpp
int flag;
std::string method;
// do something with flag
flag = ...;
switch (flag) {
case 0:
    method = "method1";
    break; // don't forget break the switch
case 1:
    method = "method2";
    break;
case 2:
    method = "method3";
    break;
default:
    method = "default";
    break;
}

// then run different statements with different "methods"
```

This code looks fine, but assume you need to determine the “method” twice in your program, then the intuitive solution would be: “okay, let’s just copy and paste this part.” This, of course, works, but what about later on, you need to add another method, say “method4”. You need to add “method4” twice, otherwise your program may run into troubles.

At this moment, you probably already notice the limitation of this approach, i.e. not extendable and easy to introduce bugs. A right way is to use a function:

```cpp
std::string chooseMethods(int flag) {
    std::method;
    switch (flag) {
    case 0:
        method = "method1";
        break; // don't forget break the switch
    case 1:
        method = "method2";
        break;
    case 2:
        method = "method3";
        break;
    default:
        method = "default";
        break;
    }
    return method;
}
```

Now, in your program, whenever you need to determine the “method”, you can simply **call** the function `chooseMethods`:

```
int flag;
std::cout << "enter method flag integer: ";
std::cin >> flag;
std::string method = chooseMethods(flag);
```

In this way, everytimes when you need to add a new method, there is only once place you need to worry about, i.e. the function `chooseMethods`.

Now, let's take a closer look at the function above, in which `chooseMethods` is its *name*, `std::string` is the *return type*, and `int flag` is the *parameter list*.

Tip: Functions are objects.

Essentially, every function has:

1. a name,
2. a return type, and
3. a parameter list.

Be aware that both 2 and 3 can be empty argument, i.e. `void` for empty return and empty parameter list for the latter.

```
// an empty param list with empty return
void printHelloWorld() {
    std::cout << "Hello World\n";
}
```

Be aware that *empty value return* doesn't mean there is no return in the function. The code above is equivalent to:

```
void printHelloWorld() {
    std::cout << "Hello World\n";
    return;
}
```

The emphasized line demonstrates that this function returns *empty value*. Returning empty value is not necessary, but sometimes it can be useful.

```
void doWork(double *data) {
    *data = 1.0;
}
```

For the `doWork` function, you can simply use it like:

```
double a;
doWork(&a); // recall address-of operator to get the pointer
std::cout << "a=" << a << ".\n"; // print 1
```

However, we know that if `data` is `nullptr`, then we have a big problem, i.e. recall dereferencing `nullptr` will cause `seg fault`.

In this case, you can do a *quick return* by checking `data`, and this requires returning empty stage.

```
void doWork(double *data) {
    if (!data) {
        return;
    }
    *data = 1.0;
}
```

The Return Type

The return type of a function must be listed explicitly and uniquely, i.e. you cannot have a function that has multiple return types.

```
int myFunc() { return 1; } // Ok
// int, int want2Return2Ints() { return 1, 2; } // ERROR!
```

However, there are always workarounds to mimic multiple returns that appear in dynamic languages, e.g. Python. One of them is to use a *struct*.

```
struct ComplexNumber {
    double real;
    double imag;
}; // don't forget the semicolon

ComplexNumber getDefaultComplexNumber() {
    ComplexNumber a;
    a.real = 0.0;
    a.imag = 0.0;
    return a;
}
```

Nonempty return types can be handles or omitted.

```
bool assign(const int len, double *array, const double value) {
    if (!array || len < 0) {
        return false;
    }
    for (int i = 0; i < len; ++i) {
        array[i] = value;
    }
    return true;
}
```

You can use the function above either

```
if (!assign(len, array, value)) {
    // recall that you should use cerr for error streaming
    // if the inputs are not acceptable, then we know that array is not
    // been touched
    std::cerr << "invalid inputs\n";
}
```

or, simply do

```
assign(len, array, value);
```

The Parameter List

The `(int flag)` in `chooseMethods` and `(double *data)` in `doWork` are called parameter lists. In general, the parameter list of functions is a *comma-separated declaration-like* of parameters. Therefore, for a parameter list, multiple arguments are, of course, supported.

```
void demoParList(int a, double b, std::string method, double *data) {
    // do work
}
```

Note: The C++ function parameter lists have strict order, i.e. there is not *key value pairs* in C++ regarding function inputs.

Return Pointers & References

There is nothing to stop you from returning *pointers* and *references*. However, whenever you directly access the memory, special care must be taken.

Let's first look at the following function that returns a pointer:

```
int *getPointer() {
    int a = 1;
    int *ptr = &a;
    return ptr;
}
```

On return, `getPointer` will return a **copy** of `ptr` and this behavior is will defined. Now, let's look at the the *function body* (content inside the function scope). A **local** integer `a` is created as well as a pointer `ptr` that points to it. The memory address of `a` is copied whiling returning `ptr`, but `a` is popped out from the stack right after the return statement. **Therefore, the dereference of the returned pointer is undefined.**

Typically, returning pointers is used for *dynamic memory allocation*:

```
double *allocArray(unsigned int size) {
    return new double [size];
}
```

Note: Don't forget to relax the memory.

Tip: If you need to write a function that returns a pointer pointing to the head memory, it's general practice that the function name should start with `create`, `alloc`, etc.

Returning references, on the other hand, is even tricker.

```
int &getRef() {
    int a;
    return a;
}
```

The code above has the legal C++ syntax. However, the returned reference refers to some local variable that is gone once the function scope ends. As a result, the refer you get from this function is undefined.

Note: Typically, compilers will warn you for returning local references.

Note: For new C++ programmers, avoid returning references.

Forward Linked List, again, with Functions

Now, a more structured implementation of our *forward linked list example* can be found in notebook func_fll.

8.1.2 Passing Arguments

A very good example to understand argument passing in C++ is the following `swap` function. A swapping operation is to exchange the contents between two objects. Let's take a look at the following pseudo code of swapping:

```
Inputs: obj1, obj2
Outputs: obj1 w/ value of obj2, and obj2 w/ value of obj1

function swap(obj1, obj2)
do
    obj1 <-> obj2
end do
```

Following the pseudo code and with the knowledge in high level programming languages, you probably simply come with a C++ implementation:

```
void Swap1(int a, int b) {
    const int temp = a;
    a = b;
    b = temp;
}
```

However, this code will not work. The reason is simple, because both `a` and `b` are copied locally inside the function thus having no effects to the actual inputted parameters.

The Copy Property

By default, all arguments are copied by their values thus resulting locally scoped variables. For instance, let's take a look at the following usage of our "wrong" swapping function.

```
int lhs = 1, rhs = 2;
Swap1(lhs, rhs);
std::cout << "after swapping, lhs="<< lhs << ", rhs=" << rhs << ".\n";
```

During the calling of `Swap1`, `lhs` and `rhs` are copied as local variables `a` and `b`, so they have totally different memory addresses comparing to the original `lhs` and `rhs`. Therefore, any operations performed on `a` and `b` have no effects to `lhs` and `rhs`.

Since we have just mentioned memory addresses, you probably can simply come up a proper implementation like:

```
void Swap2(int *a, int *b) {
    const int temp = *a;
    *a = *b;
    *b = temp;
}
```

Now, the local copies of `a` and `b` are the pointers that still point to the input arguments. Therefore, the code above can successfully swap the contents.

```
int lhs = 1, rhs = 2;
Swap2(&lhs, &rhs); // be aware that we pass in memory addr
std::cout << "after swapping, lhs="<< lhs << ", rhs=" << rhs << ".\n";
```

In programming, this copy property is referred as *pass by values* (PBV). Notice that with PBV, you duplicate each of the parameters thus doubling the memory usage.

Note: MATLAB, by default, has PBV property.

Download and play around with notebook swap.

Pass by References (PBR)

C++ allows you pass parameters as their references. This is a convenient feature that allows one to write efficient code.

```
void Swap3(int &a, int &b) {
    const int temp = a;
    a = b;
    b = a;
}
```

In the code above, instead of creating copies, two references that bind to the input arguments are created. Recall that references are just alternative names of their corresponding objects, therefore, any modification will affect the original variables.

Tip: Use `const` reference with `std::string` whenever possible.

In general, the creation of an `std::string` requires a dynamic memory allocation (because the size of the string is unknown) and a memory copying. As a result, passing `std::string` by value can be very inefficient.

```
// first printing
void print1(std::string msg) {
    std::cout << msg << '\n';
```

(continues on next page)

(continued from previous page)

```
}

// second printing
void print2(const std::string &msg) {
    std::cout << msg << '\n';
}

std::string msg = "hello world!";

for (int i = 0; i < 10000; ++i) {
    print1(msg);
    print2(msg);
}
```

For the example above, without additional efforts, the first version requires more resources than the second one does due to 10000 additional times of dynamic memory allocation and data copying.

Tip: Link we have learned in the *reference* section, the same rule applies for using references with functions, i.e. use `const` whenever possible.

PBV vs. PBR

In general, if the data is too large so that copying it becomes the bottleneck of your programs, then you should switch to use PBR.

Considering `Swap2` and `Swap3`, both of them perform the swapping operation. The former requires copying the pointers, while references are passed for the latter. For this case, it's hard so say which one is preferred.

Tip: People come converted from C programming usually prefer passing objects by their memory addresses (pointers), because it's clear to them that the objects will be modified. For native C++ programmers, the latter is usually used. But the drawback is that sometime people get confused about the function interface, e.g. the interface is identical for `Swap1` and `Swap3`.

8.1.3 Function Prototypes & Implementations

Declarations & Definitions of Variables

In *defining and initialing variables*, we have learned how to define a variable, say `int a;`. With this simple piece of code, two steps actually happen: 1) *declaring* `a` as an `int`, and 2) *defining* it in the stack memory.

We can, of course, explicitly separate these two steps by using the keyword `extern` in C++.

```
extern int a;  // declaration

int main() {
    std::cout << "a=" << a;
```

(continues on next page)

(continued from previous page)

```
    return 0;
}

// define a
int a = 1;
```

The separation of declarations and their corresponding definitions is significant in C++ (as well as in C), because this allows us to structure libraries (packages) whose declarations go to the *header files* and definitions stay in the *source files*. These concepts will be taught in the future lecture.

Note: Declarations can appear as many times as you want, but definitions must be unique!

```
extern int a;  // declaration
extern int a;  // Ok
extern int a;  // No problem

int main() {
    std::cout << "a=" << a;
    return 0;
}

// define a
int a = 1;
// int a = 2; // ERROR, we already learned
```

Constant variables can also be declared first.

```
// declaration
extern const int b;

int main() {
    std::cout << "b=" << b;
}

const int b = 2; // define it, must be initialized
```

"Declarations" of Functions

In C++, a function's declaration is called *prototype*.

```
void myFunc(); // note that the semicolon indicates prototype
void myFunc(); // you can declare as many times as you want...
```

Notice that `extern` is optional.

“Definitions” of Functions

A function’s definition is called *implementation*.

```
void myFunc() {
    std::cout << "calling myFunc\n";
}
```

Notice that the implementation of a function is indicated by the scope opener (`{`) and closer (`}`).

8.2 Advanced Topics

8.2.1 Function Types & Function Pointers

Recall that C++ is a static language, which all the variables must have their unique types. Unsurprisingly, functions are variables thus having their types.

At the beginning the *the basis section*, we have learned that any functions are associated with a return type and a list of parameter arguments. Therefore, the type of a function can be determined by the return type and the parameter list.

For instance, let’s look at the simplest function, which takes nothing and returns empty.

```
void Empty() {}
```

Its return type is `void` and input argument list is empty, or `void`. So we can say that function `Empty` is determined by a function type that returns `void` and takes `void`.

Syntactically, the type of `Empty` is `void(void)` (C/C++) or simply just `void()` (C++). In general, the type of a function is given by the following syntax: `return_type(type1, type2, ...)`, for example, the `Swap2` function has type of `void(int*, int*)`.

Note: Function declaration (prototyping) is actually like any other declarations thus having type and variable name. The different is that the variable name is in the middle of the type, i.e. `void Empty()`. In this case, it is very similar to defining arrays, i.e. `int array2[2]`, where `array2` is the variable that has type of `int[2]`.

With a type precisely defined, we expect, of course, its pointer and reference defined as well. The syntax is as following:

```
void (*)(); // function pointer pointing to void()
void (&)(); // function reference referring to void()
```

Note: Dereferencing function pointers will evaluate the function pointers themselves.

```
int Identity(int a) {
    return a;
}

// define the function pointer
int (*fun_ptr)(int);

int main() {
    // explicit
    fun_ptr = &Identity;
    // implicit
    fun_ptr = Identity;
    // dereference evaluates back to pointer
    fun_ptr = *Identity;
    // or...
    fun_ptr = ***Identity;
}
```

The existence of function pointers is useful. because it allows us to pass functions to other functions as their parameter list's arguments.

```
void call(void (*func)()) {
    func(); // call the function
}
int call2(int (*func)(int)) {
    return func(2); // type is int(int)
}
```

Tip: You should always use function pointers.

Note: Function pointers are very old-school. We will learn using *lambda calculus* and/or `<functional>` in C++ in the future.

8.2.2 Default Arguments

One of the important features in high level programming languages is to use the so-called *default arguments*.

Default parameters are not allowed in the middle of a parameter list.

```
void doWork1(int a, int b, double tol = 1e-12, double sigma=2.0); // ok
void doWork2(int a, int b = 1, int c); // ERROR! b must appear after c
```

With default parameters, you can use `doWork1` in the following ways:

```
doWork1(1, 2); // Ok, equiv to doWork1(1,2,1e-12,2.)
doWork1(1, 2, 1e-4); // Ok, overwrite tol
doWork1(1, 2, 1e-2,3.0); // Ok
```

Warning: Default parameters can only be used for function prototyping, they are not allowed in function definitions that are separated from the declarations.

```
int f1(int a = 1); // declaration
// int f1(int a = 1) { return a;} // ERROR!
int f1(int a) { return a; } // OK

int f2(int a = 1) { return a; } // OK, declaration and definition together
```

The notebook default is a good example of showing using default argument with function as input argument for computing derivatives.

8.2.3 Function Overloading

Recall that when you first learned about *types* in C++, I told you that the names of variables are unique. However, C++ allows you to have multiple functions with same name under certain situations. This exception is, roughly speaking, called *function overloading*.

To be more precise, function overloading is: *functions that have the same name but different parameter lists and that appears in the same scope.*

```
// consider the following two interface for computing
// the mean of an array
double dmean (const int n, const double *array);
float smean (const int n, const float *array);

// C++ function overloading allows you to have a unified interface
// for them, i.g.
double mean(const int n, const double *array);
float mean(const int n, const float *array);
```

Note: Overloaded functions cannot differ only in the return types!

```
// The following "overloading" of fun is not allowed!!
double fun();
float fun(); // This will throw error!
```

The Beauty

C programming does not allow function overloading, this is very inconvenient. I like to use the absolute function as an example and this function is provided in both C and C++ standard libraries.

C++ defines integer absolute value functions in `<cstdlib>` and floating siblings under `<cmath>`. For plain old C, they are defined in `<stdlib.h>` and `<math.h>`.

	float	double	int	long
C	fabsf	fabs	abs	labs
C++	abs	abs	abs	abs

As you can see, in C, there is a uniquely defined `abs` function for each of the built-in type. With the power of function overloading, a unified interface `abs` is defined for all built-in types.

Take a look at the notebook overload.

The Traps

Function overloading is powerful, but you need to use it with special care. Now consider the following example.

```
void f(int a);
void f(int a, int b = 1);
```

The functions above have different parameter list, so they are valid overloaded functions. But now, in the program we you try to call `f`, we will run into problem.

```
int main() {
    f(1); // which one????
}
```

This is called *interface ambiguous error* in C++ and the compiler will abort. However, the tricky part is that when you build the library with the two `f`, the compiler will not complain.

Also, let's take a look at the example below.

```
void f(int a);
void f(const int a);
```

It seems to you that `int` and `const int` are "two" types, but with regards of the function parameter, they are same. Therefore, the `f` above is not considered as function overloading, and if you try to define them separately, you will have multiple definitions error.

8.2.4 Function Matching (optional section)

Todo: Need to convert this part from my old lecture slides.

CHAPTER NINE

LECTURE 5: PACKAGES & MAKEFILES

Table of Contents

9.1 Toward C++ Packages/Libraries

We can't just live with a single `main` function, well, technically you can, but the obvious shortage is that your `main` script will become extremely large and eventually become unmaintainable.

During the development of your main programs, you main find some of the functions (in general, interface methods) are very valuable and you want to reuse them in the future or share with others (great!).

In such situations, you want to implement a *library*.

9.1.1 Separate Interfaces and Implementations

As we already learned, for a program to use an object (variable, function, or class), you just need to make sure its declaration can be seen prior to the execution statement. The actual implementation/definition can go after the declaration, e.g. having the function prototype before the `main` function and its implementation of the `main`.

```
extern const int a;  // declaration
int main() {
    std::cout << "a=" << a <<std::endl;
}
// define a
const int a = 100;
```

Header Files

Roughly speaking, *interfaces* are declarations and usually lie in *header files*, i.e. `*.hpp`, `*.hxx`, `*.H`, `*.h++`, `*.h`, etc. And you use `#include` to bring in their declarations.

```
#include <iostream>

// now, std::cout, std::cin, std::cerr are seen

#include <cmath>

// now std::sin, std::cos, std::log ... are seen

#include <string>

// now std::string, std::to_string ... are seen
```

Tip: The `#include` can be used with angle brackets, i.e. `<>`, or double quotation marks, i.e. `""`. Typically, the former is for system libraries, e.g. standard libraries such as `iostream`, `cstdlib`, `cmath`, etc. The latter is used for local and/or user-defined interfaces.

Take the example of computing derivatives, we can put the `diff` in a header file called `diff.hpp`

```
// in diff.hpp

// prototype of diff
// f is input function
// x is evaluated point
// h is differential spacing
double diff(double (*f)(double), const double x, double h = 1e-5);
```

Now, in the main function, you can `#include` the `diff` interface.

```
// main.cpp

#include <cmath>
#include <iostream>
```

(continues on next page)

(continued from previous page)

```
// bring in diff
#include "diff.hpp" // double quotation marks

int main() {
    std::cout << "sin\'(1)=" << diff(std::sin, 1)  << std::endl;
    return 0;
}
```

Tip: Personally, I think the best way to under stand `#include` is *copy/paste*, i.e. copy the contents in the file that is included and paste them at where `#include` appears.

Now, open the terminal and compile the program.

```
$ g++ main.cpp
/tmp/cced5cZh.o: In function 'main':
main.cpp:(.text+0x46): undefined reference to 'diff(double (*)(double),␣
↪double, double)'
collect2: error: ld returned 1 exit status
```

We got an error, which is expected, because we have not yet define the function `diff`.

Note: `undefined reference to` is a common error message that indicates missing definitions to certain interfaces.

Source Files

Source files are files that hold the implementations of the interfaces that are defined in their corresponding header files.

```
// in diff1.cpp
#include "diff.hpp"  // first include the interface

// define diff, notice that default argument is dropped
// first order scheme
double diff(double (*f)(double), const double x, double h) {
    if (h <= 0.0) {
        h = 1e-5;
    }
    return (f(x + h) - f(x)) / h;
}
```

With the source code implementation, we can now compile our program as:

```
$ g++ main.cpp diff1.cpp -o main
$ ./main
sin'(1)=0.540298
```

Notice that we have included the source file in our compiled file list, this is called *implicit linking*.

Warning: You can have as many file as you want in the compiled file list, but **only** one `main` function can exist. Moreover, each implementation must be unique!

Take a look at the archive diff.

Compilation & Linking

Previously, I told you that when you compile a program, say `demo.cpp`, you just simply type:

```
$ g++ demo.cpp
$ ./a.out
```

What happens under the hood is that `g++` first compile `demo.cpp` into *machine code*, then link the *object file* with standard C++ libraries.

```
$ g++ -c demo.cpp
```

`-c` indicates compilation and will yields so-called *object files*, i.e. `*.o` files. The command above will generate an object file `demo.o`.

```
$ g++ demo.o -o demo
$ ./demo
```

The command above is linking that will link the machine code with C++ libraries and produce an executable `demo`.

The separation of compilation and linking is sometimes referred as *explicit linking* (the first style is *implicit*).

If we have multiple files, we can still do implicit linking (as we have already shown in previous section).

```
$ g++ demo.cpp src1.cpp src2.cpp ... -o demo
$ ./demo
```

The drawback is that it's not portable and will generate a huge executable.

The preferred way is to do explicit linking.

```
$ g++ -c src1.cpp
$ g++ -c src2.cpp
...
$ g++ demo.cpp src1.o src2.o ... -o demo
$ ./demo
```

Libraries

We have been keep talking about libraries in C++, in order to create a (static) library, we need to use GNU archive tool `ar` on Linux.

The archive tool is able to create, modify, and extract files from an archive. On Linux, a (static) library ends with `.a` and starts with `lib`. For instance, if you have a library that has name of `demo`, then the archive name is `libdemo.a`.

To create an archive/library, you need to invoke:

```
$ ar [options] library-file object-files...
```

Commonly used options are:

1. `r`: Insert the files member... into archive (with replacement).
2. `s`: Add an index to the archive, or update it if it already exists.
3. `v`: Be verbose printing.

Therefore, to create a library (using the demo example above), you can:

```
$ ar rsv libdemo.a src1.o src2.o ...
$ g++ demo.cpp libdemo.a
$ ./a.out
```

For a complete list of options and how they work, please refer to the manual page of `ar`, i.e. type `man ar` in the terminal. Or look at the online Linux manual page of ar.

Additional Compilation/Linking Arguments

During compilation, the headers that are included in the source files are assumed can be located in either the standard C++ include paths or paths that exist relative to `cwd`.

Note: On Linux, there are some standard system-wise C++ include paths besides the C++ header files themselves. Typically they are `/usr/include` and `/usr/local/include`. Adding files to these two paths requires root privilege and is not recommended for libraries that are still under development. In addition, for different Linux distributions, there might be some different "standard" include paths.

In this case, we need a way that to tell compiler to where to find the header files included in the source code. GCC allows you to use `-I` argument followed by path argument.

Now, let's consider the following demo source code `demo_src.cpp` example:

```
// demo_src.cpp
#include "demo_header.hpp"

// impls ...
```

We we invoke the compilation command, i.e.

```
$ g++ -c demo_src.cpp
```

It assumes that the header file `demo_header.hpp` can be found either in standard include paths. However, this is, in practice, typically not the case. In order to tell GCC where to find `demo_header.hpp`, we need to add `-I` switch followed by either relative or absolute path that points to `demo_header.hpp`.

```
$ g++ -c -I/path/to/demo_header.hpp demo_src.cpp
```

If there are multiple includes, you can add additional `-I` switches.

On Linux, linking can be done by explicitly adding the library file with its relative or absolute path. However, a more common way is to use `-l` (ell) followed by the library name. For instance, `-ldemo` assumes a library file with name `libdemo`.

The problem, link `#include`, `-ldemo` assumes that `libdemo` can be found under "standard" library-paths that usually do **NOT** include `cwd`. Therefore, if you have `libdemo.a` that stays in `cwd`, then

```
$ g++ demo.cpp libdemo.a
```

works but

```
$ g++ demo.cpp -ldemo
```

does not work because the linker cannot find `libdemo`.

To overcome this, using the `-L` switch to indicate the library search paths, i.e.

```
$ g++ demo.cpp -L/path/to/libdemo -ldemo
```

A Complete Example

First download the archive linking example, in which a `hello-world` project is implemented.

Here is the project structure:

```
linking/
    include1/
        hello.hpp   # hello header file
    include2/
        world.hpp   # world header file
    lib/
    main.cpp        # main program
    hello.cpp       # hello source code
    world.cpp       # world source code
```

Listing 1: include1/hello.hpp

```
void hello();
```

Listing 2: hello.cpp

```
#include <iostream>
#include "hello.hpp"  // under include1
void hello() {
    std::cout << "Hello ";
}
```

Listing 3: include2/world.hpp

```
void world();
```

Listing 4: world.cpp

```
#include "world.hpp"  // under include2
#include <iostream>

void world() {
    std::cout << "World\n";
}
```

The main program is:

Listing 5: main.cpp

```
#include "hello.hpp"
#include "world.hpp"
int main() {
    hello();
    world();
    return 0;
}
```

Clearly, we want our main program to print out `Hello World` in the terminal. To do that, we need first compile the source files `hello.cpp` and `world.cpp`.

```
$ g++ -c -Iinclude1 hello.cpp
$ g++ -c -Iinclude2 world.cpp
```

Note: Relative paths are used here.

Now to compile the main program into object file, we need to add both include paths since both `hello.hpp` and `world.hpp` are used in the `main.cpp`.

```
$ g++ -c -Iinclude1 -Iinclude2 main.cpp
$ ls
include1/  include2/  lib/  hello.cpp  hello.o  main.cpp  main.o  world.
↪cpp  world.o
```

Next step is to user `ar` to create a library called `hello-world`.

```
$ ar rsv lib/libhello-world.a hello.o world.o
ar: creating lib/libhello-world.a
a - hello.o
a - world.o
$ ls lib
libhello-world.a
```

Warning: Do **NOT** include the `main.o` or, generally speaking, definition of the `main` function into an archive!!

To do the final linking, first try w/o specifying the `-L` switch.

```
$ g++ main.o -lhello-world
/usr/bin/ld: cannot find -lhello-world
collect2: error: ld returned 1 exit status
```

We need to specify the path to `libhello-world`.

```
$ g++ main.o -L lib -lhello-world
$ ./a.out
Hello World
```

Tip: Space is allowed between paths and `-I`/`-L`.

9.2 Makefiles

In practice, for relatively large projects, manually doing compilation and/or linking is infeasible. Therefore, we need a *building* tool that can automatically do the tedious work for us. Luckily, we may have a ton of compilations, but they are very well structured thus having common procedures. Therefore, writing the *descriptions* of compilation/linking is feasible and should be easy.

An essential tool for building C++ projects on Linux is `make` or *makefiles*. It's hard to lecture everything of `make` since it is a very large building framework. In addition, man-made makefiles become rare and rare.

For this section, we will learn through examples and try to understand some essential components of `make`, which will be used in future homework assignments. If you are interested in `make`, it's good to look at the official website of make.

Take a look at the archive make_eg.

CHAPTER

TEN

LECTURE 6: CLASSES

Table of Contents

10.1 Introducing *Object-Oriented Programming* (OOP), `class`

One of the main different between C and C++ is that the latter is an *object-oriented programming* (OOP) language, which means models are organized around *objects* instead of *actions*. Traditionally, programs are done in a way such that data and actions are separated and allocating data is easy but defining the complex actions on top of it is hard.

The fact is that objects are what we care about, instead of arbitrarily data. For instance, each of the student has a unique school ID, which can be represented as an integer or a string. However, such a raw data type has no meaning on its own thus useless; the "ID" data only becomes meaningful when it couples with the corresponding student, which can be viewed as an `object`.

In OOP, an `object` is a user-defined model that can contain both data (a.k.a. fields/attributes/members) and code (a.k.a. methods/member functions).

Note: One of the nice feature of OOP is that it allows people to fit the programming language into their problems.

The data and code usually are only applicable to the object itself and this makes the programming cleaner and nicer.

10.1.1 The Basis

Personally, the first step of doing OOP in C++ is to think through your problems and structure the design of objects. (This is the most challenging part in OOP!)

For instance, let's keep thinking of the example of `Student`. Each student has:

1. a name: first/last and (or) middle names,
2. sex: male, female, or other,
3. type: undergraduate, master, or doctoral,
4. status: in what year is this student in, and
5. ID number.

These will be the members of `Student`, and their corresponding data types are:

1. name: `std::string`,
2. sex: `int` or `std::string`,
3. type: `int` or `std::string`,
4. status: `int` or `std::string`, and
5. ID number: `int` or `std::string`.

Now, let's group everything into a `class`, which is conceptually and syntactically close to *struct*.

```cpp
class Student {
    std::string name;
    std::string sex;
    std::string type;
    std::string status;
```

(continues on next page)

(continued from previous page)

```
    long id;
}; // semi-colon
```

Now, we want to create a student named Joe Chang.

```
Student joe;
joe.name = "Joe Chang"; // ERROR!
```

However, this will not work, the following error will be thrown:

> error: 'std::__cxx11::string Student::name' is private within this context

The reason is that we don't have "permission" to access the `name` member in `Student` due to "incorrect" *accessing specifiers*.

In C++, each content of a `class` object is associated with one of the following accessing specifiers: `public`, `private`, and `protected` (future).

`public` Access

A public access of a member or a member method is to allow it to be used from outside of a class by any other functions/classes. To make the above example work, we can simply do:

```
class Student {
public:
    std::string name;
    std::string sex;
    std::string type;
    std::string status;
    long id;
}; // semi-colon
```

Then, all of `name`, `sex`, `type`, `status`, and `id` are exposed to outside thus modifying their values is valid.

```
Student joe;
joe.name = "Joe Chang"; // Okay
```

Note: `struct`, by default, grants all its contents with `public` accessing permission.

`private` Access

In contract to `public` accessing, contents with `private` accessing specifier only allows to be accessed inside the class. By "inside", it means that a `private` member can be only modified by other member functions.

Note: By default, all contents in a `class` are `private`.

Remarks Regarding `public` & `private`

Typically, C++ prefers to have all data attributes under `private` specifier and modify them through `public` member functions (later). To distinguish between `public` and `private` contents, usually, people define some consistent naming schemes of the variables.

Tip: Personally, I like to either append an "_" either in front of or at the end of `private` members.

```cpp
class Student {
private:
    std::string _name;   // append front _ to indicate private
};
```

Constructors & Destructors

When we create an object, the object's *constructors* will be called. And when the object reaches its end of life, C++ provides a mechanism called *destructor* that will be invoked. C++ generates a *default constructor*, a *copy constructor* and a *destructor* for every class been defined.

Note: The reason of the existence of the former two is that, by default, C++ must guarantee every user-defined object can be constructed and copied.

You can, of course, overwrite the three class "built-ins".

```cpp
class Student {
public:
    // default constructor
    Student() {
        // if you don't define this, then all the
        // members will have default values of string
        // and long
        name = "NA";
        id = -1;
    }

    // copy constructor
    Student(const Student &other) {
        // by default, it this is not given, then
        // all attributes will copy the corresponding entries
        // in "other".
        id = other.id;
        name = other.name;
        sex = other.sex;
        type = other.type;
        status = other.status;
    }

    // destructor
    ~Student() {
        std::cout << "destructor called!\n"
    }
```

(continues on next page)

(continued from previous page)

```
    std::string name;
    std::string sex;
    std::string type;
    std::string status;
    long id;
};
```

Some remarks:

1. Constructors are "member function" like except that the function name must be the object name and there is no return type.
2. Destructors are also function-like w/o return types, in addition, a prefix ~ must be written.

```
{
    Student joe;
    // what is the following output
    std::cout << joe.name << ':' << joe.id << '\n';

    joe.id = 123;
    joe.sex = "male";
    joe.type = "master";
    joe.status= "G1";

    Student joe_copy(joe); // invoke copy constructor

    // what about this?
    std::cout << joe_copy.type << ':' << joe_copy.status << '\n';
} // destructor called here!
```

To better understanding constructors and destructors, I **strongly** encourage you look at the notebook constr-destr.

More on Constructors

Of course, you are not limited on only have the default and copy constructors, to write your own constructors of an object, the following two rules, of which you need to keep in mind:

1. Defining constructors is like defining functions **without** return types.
2. The "function" name must be the name of the object.

The rest are just parameter list that you want to enable to the object construction interface.

```
class Student {
public:
    // constructor with student name
    Student(const std::string &nm) {
        name = nm;
    }

    // constructor with student name and ID
    Student(const std::string &nm, long ID) {
        name = nm;
```

(continues on next page)

(continued from previous page)

```
        id = ID;
    }

    // constructor with all information
    Student(const std::string &nm,
            const std::string &s,
            const std::string &tp,
            const std::string &st,
            long ID) {
        name = nm;
        sex = s;
        type = tp;
        status = st;
        id = ID;
    }

    std::string name;
    std::string sex;
    std::string type;
    std::string status;
    long id;
};
```

Then, in the main program, you can use these constructors as:

```
Student s1; // default constructor
Student s2("Jane"); // constructor with name
Student s3("Chow", 1220033); // constructor with name and ID
Student s3_cp(s3); // copy constructor
Student s4("Mike",
           "male",
           "master",
           "G1",
           1234567890); // the "complete" version
```

Implicit Construction

For **single** parameter constructors, by default, C++ assumes the constructors are *implicit* meaning that the object can be constructed by converting from the single input parameter. In the `Student` example, the constructor with only name parameter is implicit.

```
Student se("Foo Bar"); // explicit construction
Student si = "Spam Eggs"; // implicit construction
```

If we use parentheses, the construction is explicit, the assignment is for implicit construction. Well, you may feel like this is not important, since it makes you feel like it's just a way how people prefer to construct objects. However, the critical difference is *function argument passing*.

```
// an demo interface function that takes student as input
void interface(const Student &stdt) {
    std::cout << stdt.name << std::endl;
}
```

As you can see, the function takes a `Student` as input, but because of the conversion between `std::string` and `Student` is given by the implicit construction process through `Student::Student(const std::string &nm)`, this function can also takes `std:string` as input!

```
Student s("Foo Bar");
interface(s);

// This is also allowed! The reason is that the input string
// "Spam Eggs" will be implicitly converted to a Student object.
interface("Spam Eggs");
```

Explicit Construction

There is not really an answer to whether allowing implicit constructor is good or not. In certain cases, it's nice and provide users a more flexible constructor interfaces. However, at the same time, it's hard to debug the code.

C++, with this consideration, provides a keyword `explicit` so that the implicit-ness of a constructor can be disabled.

```
class Student {
public:
    // explicit constructor with student name
    explicit Student(const std::string &nm) {
        name = nm;
    }
    ... // data members
};
```

Now, if you try to call the `interface` function with a string input, you will get a compilation error.

```
// interface("Spam Eggs"); // ERROR! string cannot be converted to Student
```

To have a better understanding of explicit/implicit constructors, please look at notebook imp_ex.

Member Functions

As we have already mentioned, an object in OOP language is not limited to only hold data attributes, it can also contain code or, in other words, member functions, which are a group of functions that only work for this type of objects.

The `this` Pointer

We have already seen functions like:

```
struct MyStruct {...};
void method(const MyStruct &obj, ...); // method for accessing obj
void method(MyStruct &obj, ...); // method for modifying obj
```

Note: The function `method` above is overloaded.

Conceptually, this kind of functions are considered to be "member functions" of `MyStruct` because it can only work for `MyStruct`. With C++ classes, you can define member functions inside the class definition (just like data attributes, which we have seen previously). Let's look at the following example:

```
class MyClass {
public:
void method();
};
```

To call it, just access the member functions like data attributes:

```
MyClass obj;
obj.method();
```

Question: how many input argument(s) does method have?

Well, intuitively, you may say zero, and this is what we have learned in the *function* lecture. However, the correct answer is **one**.

If the input is zero, how does `method` know that it belongs to `MyClass` ;). What happens under the hood is that for each of the member functions, a `this` pointer is passed in as the actual first input argument. The `this` pointer has type of the object, in this case `MyClass`.

Note: For those who come from Python, `this` == `self`. The only difference is that `this` is transparent.

Personally, I suggest that to better understanding member functions, let's Introduce a concept called *generic member function interface*. The `method` function has a generic interface of:

```
// Notice that no constant qualifier
void MyClass::method(MyClass *this);
```

Warning: The generic interfaces are just for demonstrations purpose, they are not legal C++ syntax!

Now, let's convert the following member functions to their generic interfaces.

```
class MyClass {
public:
void set_value() {
    // we can explicitly use this to access a
    this->_a = 0;
}
// overload!
void set_value(int a) {
    // we can omit this, which is typically
    // what ppl do in practice
    _a = a;
}
void copy(const MyClass &other) {
```

(continues on next page)

(continued from previous page)

```
    _a = other._a; // within class definition, we can access _a!
}

private:
int _a; // give some data for demo
};
```

Remarks:

1. Now, you should understand that the meaning of `private` accessing. `_a` can only be accessed by `MyClass` during defining the interfaces.
2. Because of using `this` is optional, we have a "convention" that to use special pattern to indicate private data attributes.

The corresponding generic interfaces (not legal C++ syntax!) are:

```
void MyClass::set_value(MyClass *this) {
    this->_a = 0;
}
void MyClass::set_value(MyClass *this, int a) {
    this->_a = a;
}
void MyClass::copy(MyClass *this, const MyClass &other) {
    this->_a = other._a;
}
```

Note: Like `private` data attributes, you can also have `private` member functions.

`const` Member Functions

All the member functions we have defined so far can be only used without `const` qualifier.

```
void call_myClass(const MyClass &obj) {
    // ERROR! because set_value passes MyClass * in
    // not const MyClass *
    // and the const qualifier applied to obj will
    // automatically affect all its member functions/attributes
    obj.set_value();
}
```

Intuitively, we know that the generic interface of member function with `const this` should be:

```
void MyClass::method(const MyClass *this, ...);
```

This is correct, but how to indicate that `this` has `const` qualifier? C++ requires you to append `const` to the end of any member functions to indicate that these member functions use `const this`; member functions with `const this` are referred as *constant member functions*.

```
class MyClass {
public:
    ... // set methods
```

(continues on next page)

(continued from previous page)

```
 4      // typically, get method should have constant qualifier
 5      int get_value() const {
 6          // or this->_a
 7          return _a;
 8      }
 9  private:
10      int _a;
11  };
```

The `const` qualifier in line 4 indicates that the first input argument of `get_value` is `const MyClass *this`.

Tip: The general rules can be also applied here, i.e. use `const` whenever possible for member functions.

```
void print(const MyClass &obj) {
    std::cout << obj.get_value();
}
MyClass obj;
obj.set_value(100);
print(obj);
```

If `get_value` is not constant member function, the we cannot call it inside the `print` *free function*.

Note: *free functions* are simply not member functions, i.e. they belong to no specific classes.

Defining Member Functions

To separate the interfaces and implementations, we need to know how to define constructors, destructors, and member functions outside of the class.

```
// in MyClass.hpp

// we only declare the member functions
// inside the class definition
class MyClass {
public:
    // explicit constructor
    explicit MyClass(int a);

    // set value
    void set_value(int a);

    // get value
    int get_value() const;
private:
    int _a;
};
```

To define these functions and the constructor, we need to indicate C++ that these methods are not *free*, they belong to `MyClass`. We need to use the scope accessor operator `::` (we have seen this in `namespace`).

```
// in MyClass.cpp

#include "MyClass.hpp"

// explicit can only be used inside class!
MyClass::MyClass(int a) {
    _a = a;
    // or this->_a = a;
}

void MyClass::set_value(int a) {
    _a = a;
}

void MyClass::get_value() const {
    return _a;
}
```

Warning: `explicit` can only be specified inside class definition!

The `ComplexNumber` Class

Let's define a complex number class by using all the techniques we have learned so far. First a complex number associates with a real and an imaginary components thus the class data attributes should only have two floating numbers.

```
class ComplexNumber {
private:
    double _real, _imag; // real and imaginary parts
};
```

Now, we want to overwrite the *default constructor* so that we can control the default behaviors of `_real` and `_imag` when users create default complex numbers.

```
class ComplexNumber {
public:
    ComplexNumber() {
        _real = 0.0;
        _imag = 0.0;
    }
private:
    double _real, _imag; // real and imaginary parts
};
```

We want to customize a constructor so that it can take input doubles. In addition, we want to make the imaginary has default value 0. In addition, we want to implement a helper function for printing the values.

```
class ComplexNumber {
public:
```

(continues on next page)

(continued from previous page)

```cpp
    ...
    // constructor with two doubles, note that we don't
    // mark explicit, so that double can be converted to ComplexNumber
    ComplexNumber(double real, double imag = 0.0) {
        _real = real;
        _imag = imag;
    }
    // implement a helper function for printing
    void print() const {
        std::cout << "real=" << _real << ", imag=" << _imag << '\n';
    }
private:
    double _real, _imag; // real and imaginary parts
};
```

Note: The constructor above an be used to implicitly convert a `double` into a `ComplexNumber` by assigning the `double` as the real part and make the imaginary component zero.

Now, let's think about some member functions. We want to provide the following functionalities for `ComplexNumber`:

1. setting/getting real & imaginary parts,
2. copying other `ComplexNumber` numbers,
3. making a copy of "myself", i.e. `this`,
4. computing the conjugate/modulus, and
5. adding/subtracting complex numbers.

Set/Get Values

Of course, we can do the traditional set/get for bullet 1, like what we did for `MyClass`, but a neat way is to utilize *function overloading*.

```cpp
class ComplexNumber {
public:
    ...
    // real returns the normal reference to _real
    double &real() { return _real; }
    // this one is overloaded to return a copy of real
    double real() const { return _real; }

    // similarly for imag
    double &imag() { return _imag; }
    double imag() const { return _imag; }
private:
    double _real, _imag; // real and imaginary parts
};
```

Why `real` and `imag` are considered as function overloading? Well, let's take a look at their generic interfaces:

```
double &ComplexNumber::real(ComplexNumber *this);
double ComplexNumber::real(const ComplexNumber *this);
double &ComplexNumber::imag(ComplexNumber *this);
double ComplexNumber::imag(const ComplexNumber *this);
```

Since they are differ **not only in return type**, they are valid overloaded functions.

Question: what if we don't have constant qualifier in the second versions?

With this interfaces, you can do:

```
ComplexNumber a(1.0, 1.0);
// which real/imag are called?
a.real() = 2.0;
a.imag() = -1.0;
// what about the following two?
std::cout << "real=" << a.real();
std::cout << "imag=" << a.imag();
```

Note: `std::cout` only accepts objects with constant qualifier.

Copy & Make Copies

Similarly, we utilize *function overloading* for implementing these two functionalities.

```
class ComplexNumber {
public:
    ...
    // make a copy
    // because we won't modify this, make it constant
    ComplexNumber copy() const {
        // what is the type of *this? and what constructor is called?
        ComplexNumber tmp(*this);
        return tmp;
        // or return *this; // do you get it?
    }
private:
    double _real, _imag; // real and imaginary parts
};
```

And copy other complex numbers:

```
class ComplexNumber {
public:
    ...
    // copy other values, no constant qualifier
    void copy(const ComplexNumber &other) {
        _real = other._real;
        _imag = other._imag;
    }
private:
    double _real, _imag; // real and imaginary parts
};
```

Compute Conjugate and Modulus

Given a complex number with *Euler form*, i.e. $z = x + yi$, where i is the *imaginary unit*, i.e. $i = \sqrt{-1}$. Then, the modulus of z is:

$$|z| = \sqrt{x^2 + y^2}$$

and the conjugate is:

$$\bar{z} = x - yi$$

```cpp
class ComplexNumber {
public:
    ...
    // modulus, require <cmath>
    double modulus() const {
        return std::sqrt(_real*_real+_imag*_imag);
    }
    // conjugate
    ComplexNumber conj() const {
        return ComplexNumber(_real, -_imag);
    }
private:
    double _real, _imag; // real and imaginary parts
};
```

Add/Subtract `ComplexNumber`

Given complex numbers z_1 and z_2, their addition and subtraction are:

$$z_1 + z_2 = (x_1 + x_2) + (y_1 + y_2)\,i$$
$$z_1 - z_2 = (x_1 - x_2) + (y_1 - y_2)\,i$$

```cpp
class ComplexNumber {
public:
    ...
    // add, i.e. this+rhs
    ComplexNumber add(const ComplexNumber &rhs) const {
        return ComplexNumber(_real+rhs.real(), _imag+rhs.imag());
    }
    // subtract, i.e. this-rhs
    ComplexNumber sub(const ComplexNumber &rhs) const {
        return ComplexNumber(_real-rhs.real(), _imag-rhs.imag());
    }
private:
    double _real, _imag; // real and imaginary parts
};
```

Usages:

```
ComplexNumber z1(1.0); // what is z1.imag()?
ComplexNumber z2(2.0, -1.0);

// tmp vars can call constant member functions

// what is the value?
z1.add(z2).print();

// can we do this? answer: yes, but why?
z1.add(3.0).print();

z2.sub(z1).print(); // what about this?
```

Put Everything Together

The overall implement can be found in notebook complex including some usage codes.

In addition, the archive complex contains the header `ComplexNumber.hpp`, the source `ComplexNumber.cpp` and a main program for testing. This is a good exercise for you to understand how to separate declarations and implementations with classes.

The `Array` Class

Another good starting example for understanding classes in C++ is `Array`, which involves a great usage of the destructor.

```
class Array {
private:
    double *_data;   // memory
    unsigned _size;  // length of the array
};
```

You probably hate *dynamic memory management* (same do I)! One obvious advantage of using the constructors and destructor of a class is that you just need to implement `new/delete` and/or `new[]/delete[]` once and C++ will, then, automatically handle dynamic memory for you.

Therefore, for the `Array` class, you need to allocate memory inside the constructors and relax it in the destructor, i.e. `~Array::Array()`. Another important point is the copy constructor, be default, if you don't explicitly provide a copy constructor, then C++ will assume the default behavior, i.e. copy `_data` and `_size` accordingly. However, what you may want is to implement a *deep copy*, so that whenever the copy constructor is invoked, it's guaranteed that a new `_data` is allocated and the values are copied.

Hint: When you implement the copy constructor, `new` the `_data` based on the size of `other`. Then copying the values by using an *iterative statement*.

Here you can find the interface I have defined for `Array`.

```
class Array {
public:
    // default constructor
```

(continues on next page)

(continued from previous page)

```
    Array();

    // copy constructor
    Array(const Array &other);

    // constructor with array size and
    explicit Array(unsigned n, double v = 0.0);

    // we want to overwrite the implicit destructor
    ~Array();

    // get the length of array
    unsigned size() const;

    // get the data pointer of array
    double *data();

    // get read-only pointer of array
    const double *data() const;

    // given a rhs Array, copy its value
    void copy(const Array &rhs);

    // reference entry "index" of the data
    double &at(unsigned index);

    // read-only reference entry "index" of the data
    const double &at(unsigned index) const;

    // resize the array
    void resize(unsigned new_size, bool prsv = true);

    // some math functions
    // L-2 norm, sum, maximum/minimum, and dot (inner) products
    double norm() const;
    double sum() const;
    double max() const;
    double min() const;
    double dot(const Array &rhs) const;

    // addition: this+rhs, return a new array
    Array add(const Array &rhs) const;

    // subtraction: this-rhs, return a new array
    Array sub(const Array &rhs) const;

    // do some terminal printing
    void print() const;

private:
    double *_data;    ///< data pointer
    unsigned _size;   ///< length of the array
};
```

Implement `Array` based on this interface and test your work. You can find a more detailed version in archive array and the corresponding notebook array.

10.2 Advanced Topics of `class`

10.2.1 Operator Overloading

`class` in C++ helps us to cluster the data and well organize the relation between our problems and the actual implementations. However, there are still techniques that can make our life even nicer.

Recall the *complex numer* example, while you read through the lecture notes, you probably have some of the following thinkings:

1. "It would be really nice to have `a = b` instead of `a.copy(b)`."
2. "It would be great to write `z3 = z1+z2` instead of `z3.copy(z1+z2)`."
3. "I like `std::cout << z1;` instead of doing `z1.print()`."

"But can I do this??"

Yes! C++ allows you to overload certain existing *operators* for your own classes.

Unary Operators & Binary Operators

We have learned the concepts of *unary* and *binary* operators in *lecture 3*. Here, let's do a quick review.

Unary operators are those applied as *prefix* of objects, e.g. not, negative, *address-of, deference*, etc.

Binary operators are those applied between *left-hand sides* and *right-hand side*, e.g. plus, minus, etc.

Overload-able Operators

In C++, most commonly used operators can be overloaded, such as `+`, `-`, `*`, `/`, `%`, `=`, `+=`, `-=`, `*=`, `/=`, `<<`, `>>`, `&&`, `||`, `!`, `++`, `--`, `()`, `[]`, etc.

For a complete list, please read the cppreference web page.

Overloading Operators as Free and Member Functions

Note: Operators in C++ are nothing but just functions. Therefore, overloading operators is nothing but *overloading functions*.

Declaring an operator overloading is just like declaring free/member functions.

For Unary Operators

Given a unary operator `@`, i.e. `@obj` where `obj` is some object. To declare operator overloading of it as **free** function:

Obj operator @ (obj);

as **member** function:

Obj operator @ ();

For the latter case, recall the concept of *the generic interface*, you know that `this` is passed in thus, conceptually, equivalent to the free function version.

For example, we want to declare the negative operator for class `Obj`:

```
class Obj {
public:
    // as member function, this is implicitly pass in
    Obj operator-() const;
};
// as free function
Obj operator-(const Obj &obj);

// in main.cpp
Obj a;
// you can do -a
-a;
// or use as a function style
a.operator-(); // if defined as member method
operator-(a); // if defined as free method
```

For Binary Operators

Given a binary operator `@`, i.e. `lhs@rhs` where `lhs` and `rhs` are some objects (with potentially different types). To declare operator overloading of it as a **free** function:

> Obj operator @ (lhs, rhs)

as a **member** function:

> Obj operator @ (rhs)

For the latter case, `this` is treated as the left-hand side.

For instance, to add addition operator + to `Obj`, we need:

```
class Obj {
public:
    // as member function
    Obj operator+(const Obj &rhs) const; // const member function!
};
// as free function
Obj operator+(const Obj &lhs, const &Obj &rhs);

// in main function
Obj lhs, rhs;
// neat way
lhs+rhs;
// or as a function fashion
lhs.operator+(rhs); // member
operator+(lhs, rhs); // free
```

See the program binary_vs_unary for a better understanding.

Assignment & Compound Assignments

For *assignment and compound assignment operators*, they are considered as binary operators and **MUST** be overloaded as member methods. There are, in general, two ways to implement it:

return void:

```
class Obj {
public:
    // assignment
    void operator=(const Obj &obj) {
        // typically, copy the data of obj
    }
};
```

vs. **return reference**:

```
class Obj {
public:
    // assignment
    Obj &operator=(const Obj &obj) {
        // typically, copy the data of obj
        return *this; // and return the reference of "myself"
```

(continues on next page)

(continued from previous page)

```
    }
};
```

For most assignment operations, there is no difference between this two implementations, i.e.

```
Obj a, b;
a = b;
// or
a.operator=(b);
```

However, the latter enables so-called *chain reaction* of assignment operators:

```
Obj a, b, c;

// we want to copy c to both a and b
// for return void version, we have to do it one by one
a = c;
b = c;
// the reason is that
a.operator=(c); // void is returned

// for reference return, the chain reaction is
a = b = c;
// neat right? What happens under the hood is that
a.operator=(b.operator=(c));
// b.operator=(c) also returns an Obj that can be kept passing in to
// a's operator=
```

Note: The chain reaction is enabled for built-in types, e.g. `a=b=c=d=1` given integers `a`, `b`, `c` and `d`.

See the notebook chain_assign for more.

Output/Input Operators Overloading

In *this first lecture*, we have learned the how to deal with *standard I/O*. Good news is that we can overload the input operator >> and output operator <<. **But these operators must be overloaded as free functions.**

The syntax to overload output operator is:

```
#include <iostream>

std::ostream & operator<<(std::ostream &out, const Obj &obj);
```

As you can see, the interface takes a *generic* output streamer, i.e. `out` with type `std::ostream`, and outputs its reference. So that you can enable the chain reaction of the output operator.

To implement it, you basically need to write contents to the output streamer and return it at the end.

```
std::ostream & operator<<(std::ostream &out, const Obj &obj) {
    out << "obj";
    return out; // must return!
```

(continues on next page)

(continued from previous page)

```
}

// in main program
Obj obj1, obj2;
std::cout << obj1 << ' ' << obj2;
// this prints "obj obj" w/o quotations
```

The exact same logic can be applied on the input operator `>>`. The syntax is

```
std::istream & operator>>(std::istream &in, Obj &obj) {
    // write things to input streamer
    // in >> ...
    return in;
}
```

Where `std::istream` is the *generic* input stream type.

Take a look at program ovld_io.

Overloading `[]` & `()`

In practice, these two operators are commonly overloaded. The former can be considered as the "accessing" operator and the latter can be used as "callable" operator.

Note: These two operators must be overloaded as member methods.

Essentially, these two operators enable your classes by `obj[...]` and `obj(...)`. One major difference is that the former can only accept a **single** input parameter, while unlimited number of parameters can be used in overloading `()`.

Note: The reason I like to reference `()` as "callable" operator is that it allows you to make your objects behave like functions. This is so-called *functors* in C++, which we will cover in the future.

```
// create a 2by2 matrix
class Matrix22 {
public:
Matrix22() {
    for (int i = 0; i < 4; ++i)
        _data[i] = 0.0;
}

// overload []
double &operator[](int i) { return _data[i]; }
const double &operator[](int i) const { return _data[i]; }

// overload ()
double &operator()(int row, int col) { return _data[2*row+col]; }
const double &operator()(int row, int col) const {
    return _data[2*row+col];
}
```

(continues on next page)

(continued from previous page)

```
private:
// store in single array
// 0:->(0,0)
// 1:->(0,1)
// 2:->(1,0)
// 3:->(1,1)
double _data[4];
};
```

Now, to use the `Matrix22` object, we can write:

```
Matrix22 mat;
mat[0] = 1.0; //  assign the first element
std::cout << "mat(0,0)=" << mat[0] << '\n';
// the following style is extremely friendly to mathematicians
// to under what we are doing.
mat(1,1) = 2.0; // assign the last element
std::cout << "mat(1,1)=" << mat(1,1) << '\n';

// it's equiv to write
mat.operator[](0) = 1;
// and
mat.operator()(1,1) = 2;
```

Reworked Version of `ComplexNumber`

With the power of operator overloading, let's redo our `ComplexNumber` example. As what we did before, let's first structure out what functionalities we need and decide what operators we want.

1. all arithmetic operations,
2. some assignment functionalities,
3. equal and not equal comparisons, and
4. enabling standard I/O operators.

We will extend these functionalities to `ComplexNumber` thus reviewing *the interface* if necessary.

Arithmetic Operations

Given complex numbers z_1 and z_2, and their corresponding *Euler formers*, their product is defined as

$$z_1z_2 = (x_1x_2 - y_1y_2) + (x_1y_2 + y_1x_2)i$$

And their division, i.e. $\frac{z_1}{z_2}$, is given by:

$$\frac{z_1}{z_2} = \frac{z_1\bar{z}_2}{|z_2|^2}$$

Where $\bar{z}$ and $|z|$ are conjugate and modulus, resp, of z.

```
1   // overload as free functions
2
3   // addition and subtraction, directly calling the existing member
4   // functions add/sub
5   ComplexNumber operator+(const ComplexNumber &lhs, const ComplexNumber &
    ↪rhs) {
6       return lhs.add(rhs);
7   }
8   ComplexNumber operator-(const ComplexNumber &lhs, const ComplexNumber &
    ↪rhs) {
9       return lhs.sub(rhs);
10  }
11
12  // product
13  ComplexNumber operator*(const ComplexNumber &lhs, const ComplexNumber &
    ↪rhs) {
14      // compute real part
15      const double real = lhs.real()*rhs.real()-lhs.imag()*rhs.imag();
16      // compute imag part
17      const double imag = lhs.real()*rhs.imag()+lhs.imag()*rhs.real();
18      return ComplexNumber(real, imag);
19  }
20
21  // division, this is important
22  ComplexNumber operator/(const ComplexNumber &lhs, const ComplexNumber &
    ↪rhs) {
23      // compute lhs*conj(rhs) by calling * operator defined above
24      const ComplexNumber top = lhs*rhs.conj();
25      // compute bottom by calling modulus and square it
26      double bot = rhs.modulus();
27      // note, calling rhs.modulus() twice is expensive to do
28      bot = bot*bot;
29      // inverse it
30      const double ibot = 1./bot;
31      return top*ibot; // question, this is valid, but why?
32  }
```

Remark: as you can see, the division is implemented by fully utilizing what we have already implemented. Of course, you can expand the formula and manually compute the real and imaginary parts (like the multiplication). In line 24, the operator `*`, which is defined in line 13) is called, and it is called again in line 31 because implicitly casting from `double` to `ComplexNumber` is allowed!

Assignments

Notice that assignments operators, including compound types, must be overloaded as member functions.

```
class ComplexNumber {
public:
    // assignment
    ComplexNumber &operator=(const ComplexNumber &rhs) {
        copy(rhs);
        return *this; // don't forget this!
    }
    // plus assign
    ComplexNumber &operator+=(const ComplexNumber &rhs) {
```

(continues on next page)

(continued from previous page)

```
        _real += rhs._real;
        _imag += rhs._imag;
        return *this;
    }
    // minus assign
    ComplexNumber &operator-=(const ComplexNumber &rhs) {
        _real -= rhs._real;
        _imag -= rhs._imag;
        return *this;
    }
};
```

Remark: by `return *this`, the chain reaction is enabled.

Comparisons

We will overload equal, i.e. ==, and not equal to, i.e. `!=`, as free functions.

```
1 bool operator==(const ComplexNumber &lhs, const ComplexNumber &rhs) {
2     return lhs.real() == rhs.real() && lhs.imag() == rhs.imag();
3 }
4 bool operator!=(const ComplexNumber &lhs, const ComplexNumber &rhs) {
5     // return lhs.real() != rhs.real() || lhs.imag() != rhs.imag();
6     // neat way
7     return !(lhs == rhs);
8 }
```

Remark: in line 7 above, the equal to operator that is defined in line 1 is first called and returns a boolean value. We, then, apply the not operator to the logical value in order to get its opposite, i.e. not equal to.

I/O Overloading

Note that I/O must be overloaded as free functions. In general, `<iostream>` or equivalent headers should be included.

```
std::ostream & operator<<(std::ostream & out, const ComplexNumber &z) {
    // format is "real imag " w/o quotations
    out << z.real() << ' ' << z.imag() << ' ';
    return out;
}
std::istream & operator>>(std::istream & in, ComplexNumber &z) {
    // load real then imag
    in >> z.real() >> z.imag();
    return in;
}
```

Summary

Putting everything together, please take a look at our new `ComplexNumber` by downloading the archive complex_new file.

10.2.2 Class Inheritance

One of the important concept of class-based OOP language is the so-called *class inheritance* relationships. This technique is typically used in the case where two (or more) class types have a natural *child* and *parent* relation.

One of the main purpose of class inheritance is *sharing common implementations*. Note that a parent can have multiple children. To declare a class inheritance, we need to user inheritance symbol **:**.

```cpp
class Base {
public:
    void base_method() const {
        std::cout << "base method\n";
    }
};

class Derived1 : public Base {};
class Derived2 : public Base {};
```

Guess what, `base_method` is automatically enabled in both `Derived1` and `Derived2`.

The `public` inheritance enables all the public members in `Base` and treat them as `public` members in the derived classes.

Note: Of course, there are other types of inheritances, but for this class, we only focus on `public` inheritance, which is the most commonly used one.

```cpp
Derived1 d1;
d1.base_method();
Derived2 d2;
d2.base_method();
```

Also, derived types can be passed as function arguments where the base type is needed. i.e.

```cpp
void call_base(const Base &obj) {
    // because base_method is constant member function
    // we can call it here.
    obj.base_method();
}

// in main.cpp
Derived1 d1;
Derived2 d2;
call_base(d1);
call_base(d2);
```

Accessing `Base`'s Non-public Members

Let's take a look at the following example:

```cpp
class Parent {
private:
    int _bank_acc;
};
```

We define a `Parent` class that holds a `_bank_acc` member variable, now you may think in order to get the information of the bank account, let's define (or . . . ;) pretend) to be a child of it. You probably think something like this:

```cpp
class Child : public Parent {
public:
    int get_bank_acc() const {
        return _bank_acc;
    }
};
```

However, this does not work! **Derived classes cannot access private members of the base class.**

Important: Unlike Java, C#, and Python, the OO inheritance in C++ is very restricted and this, sometimes, can be a nice thing to have.

Clearly, `private` is not a right permission to have in order to allow child classes to access non-public members. In this case, as we *already mentioned* before, you need to use `protected` access specifier.

```cpp
class Parent {
protected:
    int _bank_acc;
};
```

Now, your "parent" allows you to his/her bank account ;)

Initializing Member Attributes

```cpp
1 class MyInt {
2 public:
3     MyInt () { _v = 0; }
4 private:
5     int _v;
6 };
```

Now, let's look at the simply class example above, especially line 3 and 5. From line 3, we know that `_v` is already defined, otherwise, we cannot use it. The question is **whether line 5 defines it?** The answer is **NO**, line 5 just simply **declares** that `MyInt` will hold an `int`. So where is `_v` actually got defined?

It's got defined inside the constructor and this behavior is implicit. To explicitly define the data attributes, we need to use the *initializer list*.

```cpp
class MyInt {
public:
    MyInt() : _v(0) {}
private:
    int _v;
};
```

In this case, `_v` is calling the constructor `int(int)` in order to define it with value 0.

Let's look at a more informative example, since `int` is too simple.

```cpp
class Object {
public:
    Object() {
        std::cout << "default constructor\n";
    }
    Object(int a) {
        std::cout << "constructor with a=" << a << '\n';
    }
    Object(const std::string &str) {
        std::cout << "constructor with str: " << str << '\n';
    }
};
```

Now, another family of objects `Holder?`; each of them holds an `Object` instance.

```cpp
// holder1 simply do nothing
class Holder1 {
public:
    Holder1() {}
private:
    Object _obj;
};

// holder2 defines _obj with another constructor
class Holder2 {
public:
    Holder2() : _obj(1) {}
private:
    Object _obj;
};

// holder3 defines _obj with the string constructor
class Holder3 {
public:
    Holder3() : _obj("ams562") {}
private:
    Object _obj;
};
```

To see the results, check this notebook init_list.

The `Guest` Example

Let's consider the following example, a hotel wants to implement a simple program that is able to computes the bill for customers given how many nights they stay. The bill contains two parts 1) cost per night and 2) cost per breakfast. By default, the prices are $150 and $20.

```
class Guest {
public:
    Guest(const std::string &name,
          const int nights) :
        _name(name),
        _nights(nights),
        _pri1(150.0f),
        _pri2(20.0f) {
        _type = "normal";
    }

    // get name
    const std::string &name() const { return _name; }

    // get how many nights this person has stayed
    int nights() const { return _nights; }

    // compute bill
    float compute_bill() const { return (_pri1+_pri2)*_nights; }

    const std::string &type() const { return _type; }
private:
    std::string _name;
protected:
    std::string _type; // customer type
    int _nights; // how many nights
    float _pri1; // cost per day
    float _pri2; // cost per meal
};
```

Now, we can use the following example:

```
Guest g1("Foo Bar", 10);
std::cout << "cost for g1 is $" << g1.compute_bill();
```

Of course, you may want to offer discounts for membership owners, say VIPs, who are still considered as `Guest`. For VIPs, your policy is $120 per day and $10 for meals.

```
class VIP : public Guest {
public:
    // call guest constructor in initializer list
    VIP(const std::string &name, const int nights) :
        Guest(name, nights) {
        // now pri1 = 150 and pri2 = 20
        // change them to the right ones for vip
        _pri1 = 120.0f;
        _pri2 = 10.0f;
        _type = "vip";
    }
};
```

(continues on next page)

(continued from previous page)

```
// in main program
VIP g2("Egg Spam", 20);
std::cout << "cost for g2 is $" << g2.compute_bill();
```

Try this notebook guest.

10.2.3 Polymorphism

Note: This is an important concept for class-based OOP languages, but I will **not** test you explicitly on this part in homework and project assignments.

Let's keep considering the `Guest` example, assume that we now want to create a special group of customers who take the advantage of the time limited online offer. For these customers, the rule is that the total cost per day is $100 including everything; moreover, an additional 10% discount will be applied. Let's name this customer group to be `SpecialGuest`.

Of course, the key difference is the `compute_bill` function, and you may play tricks like assigning `_pri2` to 0 and `_pri1` to 90%x$100. But the problem is that the code will become unreadable and to specialized.

A clean solution would be rewrite `compute_bill`.

```
class SpecialGuest : public Guest {
public:
// call guest constructor in initializer list
SpecialGuest(const std::string &name, const int nights) :
    Guest(name, nights) {
    // we don't care of _pri1 and _pri2 because we
    // will rewrite the function compute_bill
    _type = "special";
}

// rewrite compute bill with the special rule
float compute_bill() const { return 90.0*_nights; }
};
```

Now, let's *overload the output operator* in order to have a clean interface:

```
std::ostream & operator << (std::ostream &out, const Guest &g) {
    std::cout <<
        g.name() << ", " <<
        g.type() << ", " <<
        g.nights() << ", total: $" <<
        g.compute_bill() << '\n';
}
```

In the main program, you have a `SpecialGuest` named Joe,

```
SpecialGuest g("Joe", 10);
std::cout << g;
```

What you expected is probably:

```
Jeo, special, 10, total: $900.0
```

But what you get is actually:

```
Jeo, special, 10, total: $1700.0
```

So Joe is not a special guest! What is the problem?

The issue is that the output operator (or in general any functions) takes `Guest` as input type. Therefore, when you call `compute_bill` inside the function body, i.e. the code block above, the actual version you invoke is the one of `Guest`.

To overcome this limitation, you need to use the so-called *virtual functions* or utilize the *polymorphism* technique in C++. This requires us to rework the `Guest`.

```
 1  class Guest {
 2  public:
 3      Guest(const std::string &name,
 4            const int nights) :
 5          _name(name),
 6          _nights(nights),
 7          _pri1(150.0f),
 8          _pri2(20.0f) {
 9          _type = "normal";
10      }
11
12      // get name
13      const std::string &name() const { return _name; }
14
15      // get how many nights this person has stayed
16      int nights() const { return _nights; }
17
18      // compute bill
19      virtual float compute_bill() const { return (_pri1+_pri2)*_nights; }
20
21      const std::string &type() const { return _type; }
22  private:
23      std::string _name;
24  protected:
25      std::string _type; // customer type
26      int _nights; // how many nights
27      float _pri1; // cost per day
28      float _pri2; // cost per meal
29  };
```

In line 19, the keyword `virtual` is to indicate that member function `compute_bill` can, potentially, be *ovrride* in its child classes.

Now, in the `SpecialGuest`, we can have:

```
class SpecialGuest : public Guest {
public:
// call guest constructor in initializer list
SpecialGuest(const std::string &name, const int nights) :
    Guest(name, nights) {
    // we don't care of _pri1 and _pri2 because we
    // will rewrite the function compute_bill
```

(continues on next page)

(continued from previous page)

```
    _type = "special";
}

// rewrite compute bill with the special rule
virtual float compute_bill() const override {
    return 90.0*_nights;
}
};
```

Note: `virtual` specifier in derived classes is not needed, but it provides a clearer interface. `override` can only be used in derived classes and it's also optional, but having it makes the code clearer.

If you `cout` Joe this time, you will get the desired output. In practice, polymorphism is an extremely useful technique. But since it involves many advanced understanding of C++, I will not go into details for this class.

CHAPTER

ELEVEN

LECTURE 7: INTRODUCTION TO TEMPLATE AND STL

Table of Contents

11.1 Introduction to `template` & *STL*

11.1.1 What are Templates?

template is a unique concept of C++; it allows to define certain templates for the compiler so that the latter can generate the code for you on the fly. Using template enables the ability of interacting with compiler in order to fully optimize the code.

A *template* argument can be either a **type** or an *integral object*. It can be applied to both functions and classes.

Using templates is just like defining functions, you need to provide a list of *template arguments* with in angle brackets, i.e. <>.

```
template<class T>
void print(const T& obj) {
    std::cout << obj;
}
```

The function above defines a template function that simply writes the unique input argument, `obj`, into `cout` streamer.

The "type" `T` is a template or placeholder that will be deduced by the compiler during compilation. For instance, we can directly plugin any *build-in types* into the `print` function, because we know the output operator is well-defined for them.

```
print(1);         // int
print(2.0);       // double
print(3.0f);      // float
print(1L);        // long
```

During instantiation of the template types, the compiler *must* compile the code on the fly, which guarantees a better performance.

Question, do the print statements above refer to the same function?

NO, each of them is derived from the template function `print` and then customized with the specific type `T`. In fact, the actual function types are:

```
print<int>(1);
print<double>(2.0);
print<float>(3.0f);
print<long>(1L);
```

So, the actual *function type* of the first print is `void(int)` with function name `print<int>`.

Now, for user defined types, i.e. classes, as long as the *output operator is overloaded*, we can directly plug them in without any addition efforts. For instance, our *complex number* is a perfect example.

```
ComplexNumber a(1.0, 2.0);
print(a);
```

because the output operator is overloaded for `ComplexNumber`, the function `print<ComplexNumber>` just works out-of-box.

Note: Templates are one of the most useful and powerful technique in practice. However, due to its complexity and the time limitation of this class, it's impossible to cover it in detail. For further reading, I recommend the book C++ Templates - The Complete Guide, 2nd Edition as a wonderful offline learning material.

Tip: Essentially, the existence of `template` can be viewed as two kinds of restaurants; the first one provides a menu of dishes, whereas the second one provides a list of template recipes with the ingredients and seasonings offered by the customers.

11.1.2 The *Standard Template Libraries* (STL)

One of the great success of using template is the *standard template libraries* (STL). By using templates, STL allows you to plug your application data into the standard programming framework, which provides you convenient yet robust basic data structure *containers* and *algorithms* that are associated with them.

In summary, STL contains three main components:

1. data containers,
2. object iterators, and
3. algorithms

Containers

The idea of containers are derived from fundamental data structures in computer science. For instance, an array is a collection of elements that are stored contiguously in the memory. Each of the elements is associated with an unique key that is the index corresponding to the array; the concept of array is pretty much static, but what is unknown is the **elements**. They can be anything, from as simple as numbers (e.g. `int`, `double`, etc.) or as complicated as some user-defined `struct` or `class`. With this consideration and the power of templates, what we can have is to implement an "abstract" interface that holds the concepts of the array, including the data management and data manipulation, then have the user elements as template arguments, so that whenever the user gives us his/her application data, we can simply generate an array data structure that is optimized for the application.

All containers share some common operations, such as `pop`, `insert`, `push`, `erase`, range-for-loop, etc. The actual uses may vary. STL containers give you a standard yet powerful way to represent and manipulate the data. Two most important features of why we should use STL containers are:

1. STL is very reliable, i.e. people all over the world use them and keep testing them on different systems.
2. STL is guaranteed to be portable, i.e. there is an obvious reason that is because of the tag—*standard*! All vendor implementations (GNU, Microsoft, Clang, Intel, etc) must stay with standard so that their implementations can be considered as standard. So if you use STL containers for your data, your code will run on Windows, Linux, UNIX, etc.

Tip: With modern C++ (after C++11), you should always consider using STL.

There are mainly two types of containers:

Sequential Containers

- **`std::vector` in `<vector>`**
  - dynamic array like container
  - most efficient and widely used container
  - insertion/deletion in the middle is expensive
  - popping/pushing from back is efficient (if you do it right)
  - constant data accessing

- **std::list in <list>**
  - doubly linked list like container
  - data accessing is slow
  - efficient element insertion and deletion
- **std::deque in <deque>**
  - doubly ended queue like container
  - roughly speaking, "vector" but supports adding/removing elements from both front and end
- **std::forward_list in <forward_list>**
  - singly linked list like container
  - you can only walk through the list in one direction
  - slightly efficient memory usage compared to std::list
  - recall *the list example*
- **std::array in <array>**
  - static array like container
  - does not support data adding and removing
  - ideal for small data on stack manipulation

Note: The difference of `std::vector` and `std::array` essentially narrows down to the comparison between *dynamic arrays* and *static arrays*.

All sequential containers have the default interface:

```
std::vector<double> v;           // vector of doubles
std::list<double> dl;            // list of doubles
std::deque<double> dq;           // deque of doubles
std::forward_list<double> fl;    // forward list of doubles
```

One exception is the `std::array`, since it's static array internally, we need to give the size as the template argument, which means the size must be known during compilation.

```
std::array<double, 3> a3;    // array of double with 3 elements
// conceptually, it's equiv to
double a3_raw[3];
```

Associative Containers

Some sorted data structures are implemented as well.

- `std::set` collection of unique keys
- `std::map` collection of unique key-value pairs
- `std::multiset` non-unique version of `std::set`
- `std::multimap` non-unique version of `std::map`

Tip: For Python programmers, you can consider `std::set` is the Pythonic `set` and `std::map` for Pythonic `dict`.

Note: However, do note that the dictionary in Python is hashmap, whereas `std::map` is ordered by keys. The hashmap in C++ is a container called std::unordered_map.

We will mainly focus on std::vector, since its the commonly used one in scientific programming.

Iterators

One of the major drawback of *pointers* is that it exposes the raw memory to you, and the standard *pointer arithmetic* may become invalid in certain situations.

For example, for an array, the difference between its two pointers gives the number of elements in between. However, the difference between two list's pointers is meaningless.

With this in mind, C++ defines a special concept—*iterators*, which are associated with different containers.

Note: *Iterators*, which are very much like pointers, behave almost the same as pointers do.

Conceptually, the main difference between pointers and iterators is that the former directly point to the system memory, while the latter point to the objects in the containers.

```
// for std::vector and and array
itr++;  // move the iterator to next element in std::vector
ptr++;  // move the address to the next one

// for std::forward_list and our list
itr++;  // move to next node in the std::forward_list
ptr = ptr->next; // move to next node in memory
// note that ptr++ give non sense output
```

All containers have the following two member functions implemented and overloaded with constant member functions as well.

- `begin`, get the iterator that points to the first element.
- `end`, get the ending iterator indicator (one after the last element).

```
auto first = v.begin(); // assume some vector (or anything else)
auto ending = v.end();
// you can simply loop through it by
for (auto itr = first; itr != ending; ++itr) {
    // do things with itr
    *itr; // dereference
    itr-> ...; access some contents.
    ...
}
```

Tip: Use == and != for iterators, just like pointers.

C++ also provides unified interfaces for all iterators:

- `std::advance`, advance an iterator **inplace** to certain positions.

```
// iter will points to the 4-th element from the current one
std::advance(iter, 4);
```

- `std::next`, get next iterator

```
// iter1 points to iter's next, iter unchanged
auto iter1 = std::next(iter);
// iter2 points to the second element from iter
auto iter2 = std::next(iter, 2);
```

- `std::prev`, the opposite of `std::next`
- `std::distance`, get the **signed number** of elements between two iterators

```
// n is the number of element between iter1 and iter2, w/o iter2
int n = std::distance(iter1, iter2);
```

Important: For the iterator of `std::vector`, it literally behaves exactly the same as the underlying memory pointers. Therefore, **for what we are focusing on**, you can consider iterators and pointers are basically interchangeable.

Algorithms

Based on standardizing how to traverse containers, STL also implements a collection of algorithms such as `copy`, `fill`, search, `for_each`, `max/min`, `partition`, `sort`, binary search, etc. Again, use these whenever possible to reduce your developing time and increase the portability and robustness of your code.

11.1.3 How To Get Start With STL

With what we have learned so far, it's great to check the online references for the APIs of STL. The one I use is cppreference.

CHAPTER

TWELVE

LECTURE 8: USING <VECTOR>

Table of Contents

12.1 Using `<vector>` in STL

`std::vector` is the most widely used container in STL. The reason is simple, i.e. because arrays can always give you better performance due to the compact memory layout.

Tip: In our Jupyter notebook, type `?std::vector` will bring you to the cppreference page of STL vector.

Unlike the "vector" we have seen in previous homework and project assignments, STL vector is a *template implementation*, thus it is not restricting to `double` (what we have). Generally speaking, the contained data type usually comes from different applications, which are, typically, not as simple as *the built-in types*.

```
namespace std {
    template<class T>
    vector;
}
```

You can consider the STL vector has the interface above.

For the commonly used member function interfaces, please look at the cppreference page referred above.

```
#include <vector>

int main() {
    std::vector<int> a(10);
    for (int i = 0; i < 10; ++i) {
        a[i] = i+1;
    }
}
```

With STL containers, no more dynamic memory management!

12.1.1 Traversing `vector`

Looping through `std::vector` is flexible, you can stick with the old school fashion, i.e. indices and operator `[]`. In addition, you have the following options:

Using Iterator

One of the standard way nowadays to loop through vectors is to use *iterators*.

Note: For `vector`, you can consider that its iterators are nothing but just pointers. More detailed speaking, the normal iterators, i.e. `std::vector::iterator` is the normal pointer, while the `std::vector::const_iterator` maps to the *pointer to constant*.

```
// given a vector v of integers
for (std::vector<int>::iterator itr = v.begin(); itr != v.end(); ++itr)
    *itr = 1;
// "dereference" the iterator and assign the value to 1
for (std::vector<int>::const_iterator itr=v.begin(); itr!=v.end(); ++itr)
    std::cout << *itr;
// cout the content with constant iterator
```

However, it's sometimes really annoying to write the iterator types, we can use `auto` and let the compiler deduce the type for use automatically.

```
for (auto itr = v.begin(); itr != v.end(); ++itr)
    *itr = 1; // itr is normal iterator
for (auto itr = v.begin(); itr != v.end(); ++itr)
    std::cout << *itr;
```

Note: C++ has introduced `auto` since C++11. It is convenient to use especially when the type names are wordy. However, at the same time, you may make your codes hard to read, given the fact that C++ codes are hard to understand already.

Using The *Range-for* Loop

The *range-for* loop comes with iterators (explained briefly later). It mimics the Python range-based loops.

```
a = [10+i for i in some_list]
for val in some_list:
    print(val)
```

Range-for loop hides the indices (since sometimes we really don't need them) and directly accesses the contents in sequential order.

In C++ (after C++11), the range-based for loop reads:

```
for ([type] val: [container])
    // do work with val
```

Where `type` is the the type of elements that are contained in `container`.

```
// again, assume we have an integer vector v
for (int val:v) {
    std::cout << val;
}
```

The code above loops through `v` by its values. You can, of course, loop through it with reference.

```
for (int &val:v) {
    val = 1;
}
```

Now, all elements in `v` are assigned with value 1.

Tip: Using *ranged-for* loop sometimes make your code more readable to people who come from high level programming languages.

The Bottom Line

Use whatever you are comfortable with! In addition, choose one over another one based on difference situations; for instance, if you have too many vectors that need to be looped at once (with potentially difference indices), then using traditional way may be messy. However, for simple tasks, using indices makes the codes straightforward and easy to read.

Understanding iterators is important since iterators are not only used in looping.

Checkout the notebook vector_intro.

12.1.2 Adding/Removing Elements

One of the major advantage of using containers is to management data dynamically; this includes the following operations:

1. `insert`, add new elements in random positions,
2. `erase`, remove elements in random positions,
3. `clear`, destroy all elements at once,
4. `push_back`, add a new element to the end, and
5. `pop_back`, remove the last element.

To insert or remove elements, you need to pass in the iterators that indicate the positions.

See this notebook vector_elemmani.

12.1.3 Using Vector Efficiently

Warning: Although `std::vector` provides you the flexibility in managing the contents, you should not always use them without proper understanding of what actually happens under the hood.

The first concern is data movement due to adding and inserting elements in the middle of arrays. These options require $O(n)$ complexity thus should be avoided. If you find out that `insert` and `remove` are highly needed for your application, then `vector` is not the right container for you.

Nest, we need to under stand how `std::vector` manages the internal database.

Each vector has two sizes that associated with the data size: 1) capacity, i.e. the storage allocated and currently available to use, and 2) size, i.e. the actual storage is in-used.

Note: `size` is no larger than `capacity`.

The capacity information can be queried by the member function `std::vector::capacity`.

Please look at the notebook vector_size, which explains the size and capacity in detail.

For problem sizes are known, then allocating enough storage at the beginning is the way to go. If the problem sizes are not known beforehand, then try to estimate a reasonable upper bound and using *push_back* to construct the vectors incrementally.

Note: Call `std::vector::reserve` to reserve capacity.

Note: STL guarantees that even starting with a **real** empty vector, pushing back always results *amortized* constant time complexity. But it's good exercise to pre-allocate the vector, especially for relatively small problems.

CHAPTER

THIRTEEN

LECTURE 9: `<ALGORITHM>`

Table of Contents

13.1 Using `<algorithm>`

The `<algorithm>` library implements many fundamental algorithms in computer science; they can be used as the building bricks for different applications.

Note: All algorithms require iterators and return (if any) iterators. For *vectors*, iterators are just pointers.

Tip: I highly recommend you all look at the cppreference page for algorithm, it is clear and easy to read. Most importantly, it provides you with the "possible implementations" and complexity analysis so that it's easy to know what happens under the hood. It's worth noting that "possible implementations" are, of course, not actual implementations in, say, GCC, but they are logically identical to each other.

13.1.1 Getting Start with `copy` & `fill`

Let's start with some easy examples, e.g. copy two containers and fill a uniform value to the elements in a container.

```
#include <algorithm>
#include <vector>

// in main
std::vector<int> src = {1, 2, 3};
std::vector<int> tgt(3);
std::copy(src.begin(), src.end(), tgt.begin());

// copy values from src begin to src end to tgt with starting
// iterator tgt.begin()
// tgt = {1, 2, 3}
```

The interface is generic, which means it is applicable to not only `vector` but other containers.

```
#include <list>
#include <vector>
#include <algorithm>

// in main
std::list<int> src = {1,2,3};
// since we have using a list, we know that the memory layout is
// not compact
std::vector<int> tgt(3);
std::copy(src.begin(), src.end(), tgt.begin());
// tgt = {1,2,3}
```

Similarly, we can use `std::fill` to fill a container with a uniform value.

```
std::vector<double> data(10);
std::fill(data.begin(), data.end(), 1.0);
// data = 1
```

Note: Most algorithms return the destination iterators, thus you can use it recursively.

```
// concentrate two vectors into one
std::vector<int> v1(2), v2(4);
std::vector<int> v(6);
std::copy(
    v2.begin(), v2.end(),  // the second part
    std::copy(v1.begin(), v1.end(), v.begin());

// on return v = {v1 v2}
```

Tip: `std::copy_n` and `fill_n` become handy if you know how many elements you need to deal with.

See this notebook alg_simple.

13.1.2 `sort`

STL has a standard, i.e. $O(n \log n)$, sorting implemented in `<algorithm>`. Using it is straightforward:

```
std::vector<int> v = {...}; // fill in the values
std::sort(v.begin(), v.end());
// done!
// now v is sorted in ascending order
```

Now, let's take a closer look and see what happens under the hood. First, we need to understand what are needed in sorting algorithms: 1) comparison, i.e. `operator<` and 2) swap, exchange two elements. `int`, as a built-in type, natively supports these requirements.

If the contained type, i.e. `T`, is not as simple as `int`, then you need to ensure these two requirements.

Note: The requirement of `swap` is called ValueSwappable.

It's always a good practice to have a `swap` member function and a free version **under the same namespace** for your data type (they may not be needed, but will save you a lot when you need them, especially for fulfilling some implicit requirements). For instance,

```
struct Student {
    int id;
    std::string name;
    void swap(Student &other) {
        std::swap(id, other.id);
        std::swap(name, other.name);
        // NOTE that swap is implemented under algorithm
    }
};
// free version
void swap(Student &l, Student &r) { l.swap(r); }

// Now, in order to sort Students, we need <
bool operator<(const Student &l, const Student &r) {
    return l.id<r.id;
}
```

See this notebook sort for running examples.

Question: How to sort the contents in descending order?

13.1.3 Algorithms with Customized Operations

STL allows you to pass in callable objects into certain algorithm routines so that you can easily extend the functionality of STL algorithms.

Commonly used operations are:

1. unary boolean operations
2. binary boolean operations
3. processing unary operations

Note: C++ only requires the user-defined "operations" are *callable*!

Their corresponding interfaces:

```
bool unary_boolean(const T &);
bool binary_boolean(const T &, const  T&);
void unary_proc(T &); // NOTE that T here can be constant types
```

A perfect example would be using `std::count_if`. Here, assuming I have a students homework scores and I want to count the total number of students whose scores are above 50.

```
// define unary operation, in this case, we can use function
bool above50(const float score) { return score > 50.0f; }

std::vector<float> scores = {...}; // fill in the scores
const int counts = std::count_if(scores.begin(), scores.end(), above50);
std::cout << "total # > 50 is " << counts;
```

The binary boolean operations can be used in `std::sort`. The default behavior is that given element `a` and `b`, `a` goes before `b` if `a` is less (operator <) than `b`. This can be overwrite by providing a customized binary boolean operation with left-hand and right-hand sides, say `lhs` and `rhs`, resp. Then `lhs` goes before `rhs` in the sorted order if the binary operation returns `true`.

```
// define the function
bool greater(const int lhs, const int rhs) { return lhs > rhs; }

// under main function
// generate a random array
std::vector<int> v = {...};

std::sort(v.begin(), v.end(), greater);
// now v is in descending order
```

Note: The binary boolean operation must meet the requirement that if `comp(lhs,rhs)` returns `true`, then `comp(rhs,lhs)` must be `false`.

13.1.4 Functors

There are many drawbacks of using traditional functions as operations inside algorithm routines. One of the obvious one is that it's not easy to capture external data into the function.

For example, for the counting averaging example, what if we want to count students store that is greater than α, where α can't be determined beforehand?

```
bool greater2(const float score) {
// we cannot add alpha into the function interface, because that will
// change the API thus resulting a non-unary operation
    return score > alpha; // how to plug in alpha???
}
```

The problem can be easily solved by using the so-called *functors* in C++. *Functors* are function objects that overload the `operator()`, so that it mimics the behavior of functions.

```
class AboveAvg {
public: // must be public!
    // don't allow default construction
    AboveAvg() = delete;
    // constructor to plug in the default score
    explicit AboveAvg(float avg) : _avg(avg) {}

    // operator() overloaded as "unary" operation
    bool operator()(const float score) const {
        return score > _avg;
    }
private:
    float _avg;
};
```

Note: The `operator()` must be overloaded as **constant** member functions.

```
// inside main function
float avg;
std::cout << "Enter average score: ";
std::cin >> avg;
AboveAvg abv_avg(avg);
std::vector<float> scores = {...}; // fill in the scores
const int counts = std::count_if(scores.begin(), scores.end(), abv_avg);
std::cout << "total # above " << avg << " is " << counts;
```

13.1.5 Lambdas

One of the drawback of using functors is that you need to explicitly create a functor whenever you use them. This can be inconvenient and make you codes messy due to too many classes.

As an alternative, modern C++ encourages people using *lambdas* instead. A lambda is an unnamed function object capable of capturing other objects in scope. **Define lambdas whenever you need them.**

```
// lambda syntax
[captures](parameter list){function body}
```

Here, we show a simple example that how to convert a “hello world” function into a lambda.

```
// function version

#include <iostream>
void hello_world() {
    std::cout << "Hello World!\n";
}
int main() {
    hello_world();
}

// lambda version
```

(continues on next page)

(continued from previous page)

```
#include <iostream>

int main() {
    auto hello_world = [](){
        std::cout << "Hello World!\n";
    }; // semi-colon
    hello_world();
}
```

As you can see, the lambda version doesn't require you to explicitly define the function. Everything is local with lambda, which is a very nice feature to have.

Captures

Lambdas are just a modern look of traditional functors, with this been said, it's important to know how to deal with captures. You have already learned that captures can be done with functors be declare member objects inside the functors. For lambdas, captures are handles within `[]`.

1. =, capture by values
2. `&`, capture by references

```
1 const int a = 1;
2 const auto print_xpa = [=](const int x) {
3     std::cout << "x+a=" << x+a;
4 };
5
6 print_xpa(2); // 3
7 print_xpa(3); // 4
```

In line 2, the = inside `[]` indicates that everything inside this lambda that doesn't appear in its parameter list will be captured (copied) by value. In line 3, the variable `a`, which is not a parameter of lambda `print_xpa`, is **copied** inside the lambda. The equivalent version of functor is:

```
class XPA {
public:
    XPA() = delete;
    XPA(const int a) : _a(a) {}
    // callable
    void operator()(const int x) const {
        std::cout << "x+a=" << x+_a;
    }
private:
    const int _a; // not reference, copy the value
};

// inside main
XPA print_xpa(1);
print_xpa(2);
print_xpa(3);
```

To capture objects by their references, we need to use `&`.

```
1 int counter = 0;
2 auto increment_counter = [&]() {++counter;};
3
4 increment_counter();
5 increment_counter();
6
7 std::cout << "counter = " << counter; // 2
```

In line 2, we can see that `counter` is indicated to be passed in with its reference inside `increment_counter`, thus it can be modified. As a result, after calling the lambda twice (line 4-5), the counter has been set to 2.

Note: Since lambdas have no explicit types, we need to use `auto` and let the compiler deduce the types for us. There are difference between constant lambdas, i.e. defined with `const auto`, and normal lambdas. The former cannot modify objects that are captured by their references.

Examples of using lambdas with algorithms:

```
std::vector<int> rand_vals = {...};
// Note that C++ can automatically deduce the return type if you
// have return inside lambdas
std::sort(rand_vals.begin(), rand_vals.end(),
    [](const int lhs, const int rhs) {return lhs>rhs;});

float avg;
std::cout << "Enter average score: ";
std::cin >> avg;
std::vector<float> scores = {...}; // fill in the scores
const int counts = std::count_if(scores.begin(), scores.end(),
    [=](const float score){return score > avg;});
std::cout << "total # above " << avg << " is " << counts;
```

13.1.6 `for_each`

For each is a fully black-box algorithm routine, which loop through a container and apply a unary operation on each of its elements.

Basically, a `std::for_each` can be converted into a for-loop with iterators:

```
for (auto itr = v.begin(); itr != v.end(); ++itr)
    unary_op(*itr);
```

Now, the following loop is for printing all values of a vector:

```
std::vector<int> v = {...};

for (auto itr = v.begin(); itr != v.end(); ++itr) {
    std::cout << *itr << '\n';
}
```

can be converted into a black-box `for_each`:

```
std::for_each(v.begin(), v.end(), [](const int &val){
    std::cout << v << '\n';
});
```

Now, let's see a more complicated example, adding two vectors. This task is easy to do with traditional for loop:

```
std::vector<int> a = {1,2,3}, b = {3,2,1}, c(3);
// c=a+b

// traditional for loop
for (int i = 0; i < 3; ++i)
    c[i] = a[i]+b[i];
```

The difficulties for converting this for loop into an `std::for_each` call are:

1. how to pass `a` and `b` into the lambda, and
2. how to handle the index `i`.

For 1, we already know the solution—capture by references; for 2, we also need to capture a reference of some index counter that is defined outside of the lambda, and keep increment it inside the lambda.

```
int index = 0;
// capture index, a, b into lambda
auto add_ab = [&](int &current_c){
    current_c = a[index]+b[index];
    ++index; // increment index here
};
std::for_each(c.begin(), c.end(), add_ab);
// Note that each of the element (int) is passed in as normal reference
// and the key is that index goes with the same speed of for_each, thus
// c and a+b are sync
```

CHAPTER

FOURTEEN

LECTURE 10: STORING MATRIX IN SCIENTIFIC COMPUTING

Table of Contents

14.1 Efficient Matrix Storage

In this lecture, we try to address a problem we have already “learned” how to deal with—*how to store matrices?*

Previously, we learned that we can use a “double pointers” to store matrices, since the double-pointer data structure can be natively mapped to any matrices.

```
double **mat;
std::cout << "enter the number of row and columns [space separated]: ";
unsigned m, n;
std::cin >> m >> n;
// first allocate m "rows"
mat = new double *[m];
// for each of the row, allocate n columns
for (unsigned i = 0u; i < m; ++i)
    mat[i] = new double [n];
// to access, using nested []
mat[0][0] = 1.0;
// we need to relax the memory
for (unsigned i = 0u; i < m; ++i)
    delete [] mat[i];
// delete the mat
delete [] mat;
```

This is ugly and we need to manage the dynamic allocation explicitly. With *vector* from STL, we can use a nested `std::vector`.

```
std::cout << "enter the number of row and columns [space separated]: ";
unsigned m, n;
std::cin >> m >> n;
std::vector<std::vector<double>> mat(m, std::vector<double>(n));
// explaination:
//  we first allocate the outer vector with size m and unifrom
//  value (std::vector<double>) of std::vector<double>(n)
//  the value is another vector that is initialized with size n
```

But, is this enough? Obviously, the performance won't be good due to the non-contiguous memory layout, i.e. the last element in row i and the first element in row i+1 are not adjacent with each other. For relatively small matrices, the penalty can be amplified.

14.1.1 How to Store Matrice Efficiently?

Since we want to use compact storage, 1D arrays should be used instead of "2D" ones. Also, each matrix has a row size and a column size, which can be considered as shape descriptor; the actual matrix lies on top of a 1D array that is virtually splitted into m rows and n columns by the shape descriptor.

```
struct Matrix {
std::vector<double> data;
int rows, cols;
};
```

Therefore, the array `data` has size of `rows*cols`. Now, the question is how to stretch the matrix into a 1D array, particularly speaking, given the following matrix $\boldsymbol{A}$, in what order we put the values into `data`?

$$\boldsymbol{A} = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

`data` can be either `{1, 2, 3, 4}` or `{1, 3, 2, 4}`.

C Order/Orientation

In computational linear algebra, C order means that a matrix is stored with with **row** orientation, i.e. `data` is `[row0 row1 row2 ...]`. For instance, given the matrix $\boldsymbol{A}$ above, the C order is `{1,2,3,4}`.

Fortran Order/Orientation

In contrast to C order, Fortran order stores matrices with column orientation, e.g. `data={1,3,2,4}`. That is, `data` is `[col0 col1 col2 ...]`.

Note: Column orientation is pretty standard in scientific programming due to the use of Fortran programming language. But for this class, we will focus on C order.

Note: MATLAB stores matrices in Fortran order. NumPy allows you to choose between C and Fortran, but the default is in C order.

Leading Dimension and Submatrices

With `data`, row size (`m`) and column size (`n`), let's see how to loop through all entries in the matrix (assuming C order).

```
// the following code print the matrix
// in the terminal with 2D shape
for (int i = 0; i < m; ++i) {
    for (int j = 0; j < n; ++j)
        std::cout << mat[i*n+j] << ' ';
    std::cout << '\n'; // cut new line
}
```

The key is `i*n`, which gives you the starting index of row i.

Now, let's see how to deal with submatrices. One way is to create a whole copy of the submatrices, but this approach is inefficient, because you need to first allocate buffer, then copy the values, and most likely you need to push the values back the original matrices once you are done with manipulating the submatrices.

The problem can be solved by introducing the so-called *leading dimensions*.

Note: A *leading dimension* of a matrix **memory** storage (assuming row orientation) is the stride size that equals to the difference between two adjacent entries in any column vectors.

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

Given the 2 by 3 matrix above, the leading dimension is 3 (column size).

Tip: For **complete** row major (rwo orientated) matrices, their leading dimensions are just the corresponding column sizes.

With the leading dimension introduced, you can create submatrix descriptors instead of memory copies. For instance, I want manipulate the submatrix:

$$\begin{pmatrix} 2 & 3 \\ 5 & 6 \end{pmatrix}$$

i.e. the tailing 2 by 2 block, one way is:

```
std::vector<double> submat(4);
submat[0] = mat[1]; // copy (0,1)
submat[1] = mat[2]; // copy (0,2)
submat[2] = mat[4]; // copy(1,1) 1*3+1
submat[3] = mat[5]; // copy (1,2) 1*3+2
for (int i = 0; i < 2; ++i)
    for (int j = 0; j < 2; ++j)
        submat[i*2+j] *= (i+j);
// copy back
mat[1] = submat[0];
```

(continues on next page)

(continued from previous page)

```
mat[2] = submat[1];
mat[4] = submat[2];
mat[5] = submat[3];
```

Of course, this is inefficient, in stead, we can do:

```
double *submat = &mat[0*3+1]; // point to the (0,1)
int ld = 3; // leading dimension
for (int i = 0; i < 2; ++i)
    for (int j = 0; j < 2; ++j)
        submat[i*ld+j] *= (i+j);
// done!
```

Notice that `submat` points at the value 2 in the original matrix, which has the leading dimension of 3. Then `submat[ld]` is nothing but the original position below 2, which is 5.

Note: BLAS (Basic Linear Algebra Subroutines and LAPACK (Linear Algebra PACKage) are next step you should look into. Particularly speaking, their C interfaces, *cblas* and *LAPACKE*.

CHAPTER FIFTEEN

CASE STUDIES