

Lecture 01

Introduction to the course. The UNIX shell. The build process.

Advanced Programming - SISSA, UniTS, 2023-2024

Pasquale Claudio Africa

26 Sep 2023

About me

- B.Sc. and M.Sc. in *Mathematical Engineering*, Politecnico di Milano, 2010-2015.
- Ph.D. in *Mathematical Models and Methods in Engineering*, Politecnico di Milano, 2019.
- Assistant Professor (RTDa) in Numerical Analysis at SISSA since March 2023.

Teaching experience

Advanced programming, numerical analysis, mathematical modeling and scientific computing.

Main languages: C++, MPI, OpenMP, Python, MATLAB/Octave.

Research interests

High-Performance Computing (HPC), mathematical models and computational methods to solve problems in science and engineering, numerical methods for Partial Differential Equations (PDEs).

Course overview

Practical info

- **Instructor:** Pasquale Claudio Africa pafrica@sissa.it
- **Tutor:** Marco Feder mfeder@sissa.it

Course material

- **MOODLE2** : please **subscribe** to the "Advanced Programming" course.
- **GitHub** : timetable, lecture notes and slides, exercise sessions.
- **Microsoft Teams** : live streaming, recordings.

Other resources

- Books (see course syllabus on MOODLE2/GitHub).
- Internet (plenty of free or paid resources).

Practical info

Lectures at SISSA. Check out [GitHub](#) regularly for up-to-date timetable, lecture topics, rooms, and course material.

Course balance (approximate):

- C++: **50%**, Python: **50%**.
- Frontal lectures: **50%**, live programming sessions: **50%**.

For live programming sessions please **bring your own laptop**.

Questions?

- Use Forum on MOODLE2 or Discussions on GitHub.
- Engage with each other!
- **Office hours:** send me or the tutor an email to book a session.

Objectives and expectations

- UNIX shell and the software build process.
- Advanced programming concepts, specifically in C++ and Python.
- Object-oriented and generic programming paradigms.
- Common data structures, algorithms, libraries for scientific computing.
- Software development tools in UNIX/Linux (software documentation, version control, testing, and project management).

Required skills

- Former knowledge of programming fundamentals (syntax, data types, variables, control structures, functions).
- Prior experience with C, C++, Java, or Python, is **recommended**, not mandatory.

Exams

1. Homework assignments:

- Throughout the course, you will be assigned 2-4 homework projects to complete either individually or in groups.
- Your submission should include solution code and a concise presentation with supporting slides, outlining your proposed solution and design choices.

2. Computer-based written exam:

- Programming questions and exercises, to be submitted digitally.

3. (Optional) Oral discussion, upon request (by either students or the instructor). It can increase or decrease your grade by up to 3 points.

Maximum achievable grade: **30 + honors** (granted in exceptional cases).

Some advice

1. Study day by day

- Create a study schedule and set clear goals.
- Prioritize consistency.

2. Practice regularly

- Solve exercises on your own.
- Programming is (mostly) learnt by doing, not by reading.

3. Interact:

- Engage in discussions on MOODLE2 forums and GitHub.
- Seek help when needed.
- **Share your knowledge with others.**

Laptop configuration

Please **bring your own laptop** with a working UNIX/Linux environment, whether standalone, dual boot, or virtualized.

For beginners: <https://ubuntu.com/tutorials/install-ubuntu-desktop>.

You can write code using any **text editor** (such as Emacs, Vim, or Nano), or an Integrated Development Environment (**IDE**) (such as VSCode, Eclipse, or Code::Blocks).

Requirements

- C++ compiler with full support for C++17, such as $\text{GCC} \geq 10$, or $\text{Clang} \geq 11$.
- Python 3. [Jupyter](#) is recommended, but not mandatory.

Any recent Linux distribution, such as Ubuntu ≥ 22.04 , or Debian ≥ 11 , or macOS system that meets these requirements should be suitable for the course.

Windows users

- Windows Subsystem for Linux (WSL2) . Ubuntu version recommended, then follow Ubuntu-specific instructions.
- Virtual machine (such as [VirtualBox](#)).
- **(Expert users)** [Dual boot](#) .

macOS users

- [Xcode](#) : provides Clang.
- [Homebrew](#) : provides GCC, Clang, Python 3.

Linux users

- Install GCC and Python 3 using your package manager (such as apt, yum, pacman).

Industry demands for advanced programming skills

- Rapid technological advancements.
- Need for scalable and efficient software solutions.
- Increased reliance on data-driven decision-making.
- Job market competitiveness and higher earning potential for skilled programmers.

Career opportunities:

- **Software developer:** Design and develop software applications.
- **Systems architect:** Plan and design complex software systems.
- **Data scientist/engineer:** Analyze and process data using advanced algorithms.
- **DevOps engineer:** Automate software development and deployment processes.
- **Cybersecurity analyst:** Secure software systems and networks.
- **Research & Development:** Contribute to scientific research with innovative solutions.

Importance of advanced programming skills (1/2)

- **Data manipulation:** C++ and Python enable efficient handling of large datasets and complex data structures.
- **Algorithm development:** Proficiency in these languages is crucial for coding and optimizing complex mathematical and scientific algorithms.
- **Tool integration:** Advanced programming skills facilitate the integration of various tools and libraries for data analysis and scientific computing.
- **Customization:** C++ and Python allow customization of software tools to meet specific project requirements.
- **Performance optimization:** These languages enable the writing of high-performance code, vital for resource-intensive calculations.
- **Machine learning:** Python is a primary language for machine learning, and C++ can be used for performance-critical components.

Importance of advanced programming skills (2/2)

- **Visualization:** Python's libraries like Matplotlib and C++'s graphics capabilities aid in creating informative visualizations.
- **Reproducibility:** Well-structured code in C++ and Python ensures research reproducibility and collaboration.
- **Prototyping:** Python facilitates rapid prototyping and experimentation for hypothesis testing and model development.
- **Automation:** Automation of repetitive tasks in data preprocessing and analysis streamlines workflows.
- **Security:** C++ can be employed for secure and efficient data handling, important in data science and scientific computing.
- **Scalability:** Both languages are suitable for building scalable systems and algorithms to handle growing data and computing needs.

Why should I learn programming in the ChatGPT era?

1. You can't really understand/modify/improve a text written in English, unless you are proficient in English!
2. Career opportunities:
 - Coding opens doors to a diverse array of high-demand careers in technology and data-driven sectors.
 - It cultivates critical thinking and problem-solving skills.
3. Artificial Intelligence (AI) and chatbots **lack** creativity!
 - They derive knowledge from historical data.
 - Innovation, idea generation, implementation of novel concepts through software and technology remain a **human** prerogative (at least for now 😅).
4. Understand how AI and chatbots work under the hood.

Time for a poll!

<http://etc.ch/TVQS>



Reminder for DSAI/SDIC students: AI2S Welcome Day

- When: 27 Sep (**tomorrow**), 19:00 - 01:00
- Where: Loft (via Giovanni e Demetrio Economo 12)
- Form: <http://bit.ly/AI2SWelcomeDay>

Welcome to the Advanced Programming!

Outline

1. History of C++.
2. History of Python.
3. The build process.
4. Introduction to the UNIX shell.
5. Introduction to `git`.

History of C++

History of C++

Introduction

The history of the C++ programming language is a remarkable journey through the evolution of computer science and software development. C++, often considered a superset of the C programming language, was designed with the goal of combining the low-level power of C with high-level features for structured programming. This chapter explores the origins, key milestones, and influential figures that shaped C++ into the versatile and widely-used language it is today.

The birth of C++

Early roots: the C language

C++ owes its existence to the C programming language, which was developed at Bell Labs in the early 1970s by Dennis Ritchie. C quickly gained popularity due to its efficiency, portability, and the flexibility it offered to system programmers. Its simplicity, combined with powerful features for memory manipulation, made it a go-to language for developing operating systems and other system software.

The birth of C++

Bjarne Stroustrup's vision

The story of C++ begins in 1979 when Bjarne Stroustrup, a Danish computer scientist, started working on what he initially called "C with Classes." Stroustrup, then at Bell Labs, aimed to enhance C by adding support for object-oriented programming (OOP) concepts. His motivation was to create a language that could address the growing complexity of software systems while maintaining the performance and control of C.

The birth of C++

From C with classes to C++

Stroustrup's work on extending C led to the development of a preprocessor called "Cfront" in 1983. Cfront allowed C++ code to be translated into C code, which could then be compiled using standard C compilers. This approach eased the transition to C++ for existing C programmers.

In 1983, the name "C++" was coined, signifying the evolution of C with Classes. The term "C++" represents the incremental increase operator in C, suggesting that C++ is an improved version of C. In 1985, Stroustrup published the first edition of "The C++ Programming Language," a seminal book that introduced programmers to the language's concepts and features.

Early growth and ANSI C++ standardization

As C++ gained popularity, there was a need for standardization to ensure compatibility across different compilers and platforms. In 1989, the American National Standards Institute (ANSI) began working on a standard for the C++ language. This effort culminated in the release of the ANSI C++ standard in 1998, which provided a formal specification of the language.

See <https://isocpp.org/> for language references, guidelines, and much more.

Object-Oriented paradigm

C++ introduced essential features of the OOP paradigm, including classes, objects, inheritance, and polymorphism. These features allowed developers to build more modular and maintainable software by encapsulating data and behavior within objects.

Template metaprogramming

Another significant innovation in C++ was the introduction of templates. Templates allowed for generic programming, enabling the creation of data structures and algorithms that could work with different data types. Template metaprogramming, a technique that uses templates for compile-time computations, further expanded C++'s capabilities.

C++ in the modern era

Standardization efforts

The C++ language has continued to evolve through a series of standards, each introducing new features and improvements. Notable standards include C++98, C++11, C++14, C++17, and C++20 (C++23 expected soon). These standards have added features like smart pointers, lambda expressions, range-based for loops, and modules, enhancing the language's expressiveness and safety.

Open source and the C++ community

C++'s success can be attributed in part to the vibrant open-source community that has formed around it. Open-source libraries and frameworks, such as the Boost C++ Libraries, have extended C++'s functionality and encouraged collaborative development.

The future of C++

C++ has found applications in a wide range of fields, including game development, finance, embedded systems, and scientific computing. Its combination of performance, portability, and expressive power makes it a versatile choice for building software across various domains.

C++ continues to evolve, with ongoing work on future language standards. Features like concepts (a type of compile-time constraint) and modules (for better code organization and encapsulation) are expected to play a significant role in shaping the language's future.

The history of C++ is a testament to the enduring power of a well-designed programming language. From its humble beginnings as an extension of C to its status as a modern, versatile language, C++ has left an indelible mark on the world of software development. Its rich history, coupled with ongoing innovations, ensures that C++ will remain a vital tool for programmers for years to come.

History of Python

History of Python

Introduction

Python is a versatile and widely-used programming language known for its simplicity, readability, and the ease with which it allows developers to write clean and maintainable code. This chapter delves into the rich history of Python, tracing its origins, key milestones, and the individuals who played pivotal roles in its development.

The Genesis of Python

Python's name and design philosophy

Python was created by Guido van Rossum, a Dutch programmer, in the late 1980s. Guido started working on Python in December 1989 during his time at the Centrum Wiskunde & Informatica (CWI) in the Netherlands. His motivation was to develop a language that combined the simplicity of ABC (a programming language he had worked on previously) with the extensibility of the Amoeba operating system.

Guido named the language after his love for the British comedy group Monty Python. Python's design philosophy, often referred to as the "*Zen of Python*" emphasizes readability, simplicity, and elegance. This philosophy is encapsulated in the [PEP 20](#) document, which includes guiding aphorisms like "*Readability counts*" and "*There should be one - and preferably only one - obvious way to do it*".

Python's early years

Python 0.9.0

Python's first public release, Python 0.9.0, occurred in February 1991. This release introduced essential features like exception handling, functions, and modules, which laid the foundation for the language's future growth.

The Python Software Foundation

In 2001, the Python Software Foundation (PSF) was established as a non-profit organization to promote and support Python. The PSF plays a crucial role in managing Python's development, organizing conferences (e.g., PyCon), and providing grants and resources to the Python community.

Python 2.x and 3.x

The transition to Python 3

Python 2.x and Python 3.x marked a significant phase in Python's history. Python 3, released in December 2008, introduced backward-incompatible changes to address shortcomings in the language. While Python 2 continued to be used for some time, the Python community has actively encouraged the transition to Python 3.

Python's popularity and versatility

Python's readability, simplicity, and extensive standard library contributed to its widespread adoption. It became a go-to language for web development, scientific computing, data analysis, and automation. Popular web frameworks like Django and Flask further fueled Python's growth.

Python in Data Science and Machine Learning

Python gained prominence in data science and machine learning due to libraries like NumPy, pandas, scikit-learn, and TensorFlow. Its ease of use and rich ecosystem made it a favorite among data scientists and engineers.

Python in education

Python's readability and simplicity have made it an excellent choice for teaching programming. It is widely used in educational settings to introduce programming concepts to beginners.

The future of Python

Python continues to evolve through a series of releases, with each version bringing new features and enhancements. Python's community-driven development process ensures that the language remains vibrant and relevant.

Python is well-positioned to thrive in emerging areas like artificial intelligence, web development, and cloud computing. Its adaptability and large community ensure it can address a wide range of challenges.

The history of Python is a testament to the enduring impact of a well-designed programming language. From its inception in the late 1980s to its current status as a versatile and ubiquitous language, Python has empowered developers to create a diverse array of software solutions. Its readability, simplicity, and thriving community ensure that Python will remain a cornerstone of the programming landscape for years to come.

Popularity of programming languages

Source: <https://pypl.github.io/PYPL.html>

Curated lists of awesome C++ and Python frameworks, libraries, resources, and shiny things.

- [awesome-cpp](#)
- [awesome-python](#)
- [awesome-scientific-python](#)
- [awesome-scientific-computing](#)

The build process: Preprocessor, Compiler, Linker, Loader

Chapter overview

- Understand the difference between compiled and interpreted languages.
- Understand the build process.
- Explore the roles of the preprocessor, compiler, linker, and loader.

Compiled vs. interpreted languages

Preprocessor

- Handles directives and macros before compilation.
- Originated for code reusability and organization.

Preprocessor directives

- `#include` : Includes header files.
- `#define` : Defines macros for code replacement.
- `#ifdef` , `#ifndef` , `#else` , `#endif` : Conditional compilation.
- `#pragma` : Compiler-specific directives.

Macros

- Example: `#define SQUARE(x) ((x) * (x))`
- Usage: `int result = SQUARE(5); // Expands to: ((5) * (5))`

Compiler

- Translates source code into assembly/machine code.
- Evolved with programming languages and instructions.

Compilation process

1. Lexical analysis: Tokenization.
2. Syntax analysis (parsing): Syntax tree.
3. Semantic analysis: Checking.
4. Code generation: Assembly/machine code.
5. Optimization: Efficiency improvement.
6. Output: Object files.

Common compiler options

-O : Optimization levels; -g : Debugging info; -std : C++ standard.

Linker

- Combines object files into an executable.
- Supports modular code.

Linking process

1. Symbol resolution: Match symbols.
2. Relocation: Adjust addresses.
3. Output: Executable.
4. Linker errors/warnings.
5. Example: `g++ main.o helper.o -o my_program`

Static vs. dynamic linking

- Static: Larger binary, library inclusion.
- Dynamic: Smaller binary, runtime library reference.

Loader

- Loads executables for execution.
- Tied to memory management evolution.

Loading process

1. Memory allocation: Reserve memory.
2. Relocation: Adjust addresses.
3. Initialization: Set up environment.
4. Execution: Start execution.

Dynamic linking at runtime

- Inclusion of external libraries during execution.
- Enhances flexibility.

Introduction to the UNIX shell

What is a shell?

From <http://www.linfo.org/shell.html> :

A shell is a program that provides the traditional, text-only user interface for Linux and other UNIX-like operating systems. Its primary function is to read commands that are typed into a console [...] and then execute (i.e., run) them. The term shell derives its name from the fact that it is an outer layer of an operating system. A shell is an interface between the user and the internal parts of the OS (at the very core of which is the kernel).

What is Bash?

Bash stands for: Bourne Again Shell, a homage to its creator Stephen Bourne. It is the default shell for most UNIX systems and Linux distributions. It is both a command interpreter and a scripting language. The shell might be changed by simply typing its name and even the default shell might be changed for all sessions.

macOS has replaced it with zsh, which is mostly compatible with Bash, since v10.15 Catalina.

Other shells available: tsh, ksh, csh, Dash, Fish, Windows PowerShell, ...

Variables and environmental variables

As shell is a program, it has its variables. You can assign a value to a variable with the equal sign (**no spaces!**), for instance type `A=1`. You can then retrieve its value using the dollar sign and curly braces, for instance to display it the user may type `echo ${A}`. Some variables can affect the way running processes will behave on a computer, these are called **environmental variables**. For this reason, some variables are set by default, for instance to display the user home directory type `echo ${HOME}`. To set an environmental variable just prepend `export`, for instance `export PATH="/usr/sbin:$PATH"` adds the folder `/usr/sbin` to the `PATH` environment variable. `PATH` specifies a set of directories where executable programs are located.

Types of shell (login vs. non-login)

- A **login** shell logs you into the system as a specific user (it requires username and password). When you hit `Ctrl+Alt+F1` to login into a virtual terminal you get after successful login: a login shell (that is interactive).
- A **non-login** shell is executed without logging in (it requires a current logged in user). When you open a graphic terminal it is a non-login (interactive) shell.

Types of shell (interactive vs. non-interactive)

- In an **interactive** shell (login or non-login) you can interactively type or interrupt commands. For example a graphic terminal (non-login) or a virtual terminal (login). In an interactive shell the prompt variable must be set (`$PS1`).
- A **non-interactive** shell is usually run from an automated process. Input and output are not exposed (unless explicitly handled by the calling process). This is normally a non-login shell, because the calling user has logged in already. A shell running a script is always a non-interactive shell (but the script can emulate an interactive shell by prompting the user to input values).

Bash as a command line interpreter

When launching a terminal a UNIX system first launches the shell interpreter specified in the **SHELL environment variable**. If `SHELL` is unset it uses the system default.

After having sourced the initialization files, the interpreter shows the **prompt** (defined by the environment variable `$PS1`).

Initialization files are hidden files stored in the user's home directory, executed as soon as an **interactive** shell is run.

Initialization files

Initialization files in a shell are scripts or configuration files that are executed or sourced when the shell starts. These files are used to set up the shell environment, customize its behavior, and define various settings that affect how the shell operates.

- **login:**

- `/etc/profile` , `/etc/profile.d/*` , `~/.profile` for Bourne-compatible shells
- `~/.bash_profile` (or `~/.bash_login`) for Bash
- `/etc/zprofile` , `~/.zprofile` for zsh
- `/etc/csh.login` , `~/.login` for csh

- **non-login:** `/etc/bash.bashrc` , `~/.bashrc` for Bash

Initialization files

- **interactive:**

- `/etc/profile` , `/etc/profile.d/*` and `~/.profile`
- `/etc/bash.bashrc` , `~/.bashrc` for Bash

- **non-interactive:**

- `/etc/bash.bashrc` for Bash (but most of the times the script begins with: `[-z "$PS1"] && return` , i.e. don't do anything if it's a non-interactive shell).
- depending on the shell, the file specified in `$ENV` (or `$BASH_ENV`) might be read.

Getting started

To get a little hang of the bash, let's try a few simple commands:

- `echo` : prints whatever you type at the shell prompt.
- `date` : displays the current time and date.
- `clear` : clean the terminal.

Basic Bash commands

- `pwd` stands for **Print working directory** and it points to the current working directory, that is, the directory that the shell is currently looking at. It's also the default place where the shell commands will look for data files.
- `ls` stands for a **List** and it lists the contents of a directory. ls usually starts out looking at our home directory. This means if we print ls by itself, it will always print the contents of the current directory.
- `cd` stands for **Change directory** and changes the active directory to the path specified.

Basic Bash commands

- `cp` stands for **Copy** and it moves one or more files or directories from one place to another. We need to specify what we want to move, i.e., the source and where we want to move them, i.e., the destination.
- `mv` stands for **Move** and it moves one or more files or directories from one place to another. We need to specify what we want to move, i.e., the source and where we want to move them, i.e., the destination.
- `touch` command is used to create new, empty files. It is also used to change the timestamps on existing files and directories.
- `mkdir` stands for **Make directory** and is used to make a new directory or a folder.
- `rm` stands for **Remove** and it removes files or directories. By default, it does not remove directories, unless you provide the flag `rm -r` (`-r` means recursively).
⚠ Warning: Files removed via `rm` are lost forever, please be careful!

Not all commands are equals

When executing a command, like `ls` a subprocess is created. A subprocess inherits all the environment variables from the parent process, executes the command and returns the control to the calling process.

A subprocess cannot change the state of the calling process.

The command `source script_file` executes the commands contained in `script_file` as if they were typed directly on the terminal. It is only used on scripts that have to change some environmental variables or define aliases or functions. Typing `. script_file` does the same.

If the environment should not be altered, use `./script_file`, instead.

Run a script

To run your brand new script you may need to change the access permissions of the file. To make a file executable run

```
chmod +x script_file
```

Finally, remember that the **first line of the script** (the so-called *shebang*) tells the shell which interpreter to use while executing the file. So, for example, if your script starts with `#!/bin/bash` it will be run by `Bash`, if it starts with `#!/usr/bin/env python` it will be run by `Python`.

Built-in commands

Some commands, like `cd` are executed directly by the shell, without creating a subprocess.

Indeed it would be impossible to have `cd` as a regular command!

The reason is: a subprocess cannot change the state of the calling process, whereas `cd` needs to change the value of the environmental variable `PWD` (that contains the name of the current working directory).

Other commands

In general a **command** can refer to:

- A builtin command.
- An executable.
- A function.

The shell looks for executables with a given name within directories specified in the environment variable `PATH`, whereas aliases and functions are usually sourced by the `.bashrc` file (or equivalent).

- To check what `command_name` is: `type command_name`.
- To check its location: `which command_name`.

A warning about filenames

⚠️ In order to live happily and without worries, **don't** use spaces nor accented characters in filenames!

Space characters in file names should be forbidden by law! The space is used as separation character, having it in a file name makes things a lot more complicated in any script (not just Bash scripts).

Use underscores (snake case): `my_wonderful_file_name`, or uppercase characters (camel case): `myWonderfulFileName`, or hyphens: `my-wonderful-file-name`, or a mixture: `myWonderful_file-name`, instead.

But **not** `my wonderful file name`. It is not wonderful at all if it has to be parsed in a script.

More commands

- `cat` stands for **Concatenate** and it reads a file and outputs its content. It can read any number of files, and hence the name concatenate.
- `wc` is short for **Word count**. It reads a list of files and generates one or more of the following statistics: newline count, word count, and byte count.
- `grep` stands for **Global regular expression print**. It searches for lines with a given string or looks for a pattern in a given input stream.
- `head` shows the first line(s) of a file.
- `tail` shows the last line(s) of a file.
- `file` reads the files specified and performs a series of tests in attempt to classify them by type.

Redirection, pipelines and filters

We can add operators between commands in order to chain them together.

- The pipe operator `|`, forwards the output of one command to another. E.g., `cat /etc/passwd | grep my_username` checks system information about "my_username".
- The redirect operator `>` sends the standard output of one command to a file. E.g., `ls > files-in-this-folder.txt` saves a file with the list of files.
- The append operator `>>` appends the output of one command to a file.
- The operator `&>` sends the standard output and the standard error to file.
- `&&` pipe is activated only if the return status of the first command is 0. It is used to chain commands together: e.g., `sudo apt update && sudo apt upgrade`
- `||` pipe is activated only if the return status of first command is different from 0.
- `;` is a way to execute two commands regardless of the output status.
- `$?` is a variable containing the output status of the last command.

Advanced commands

- `tr` stands for **translate**. It supports a range of transformations including uppercase to lowercase, squeezing repeating characters, deleting specific characters, and basic find and replace. For instance:
 - `echo "Welcome to Advanced Programming!" | tr [a-z] [A-Z]` converts all characters to upper case.
 - `echo -e "A;B;c\n1,2;1,4;1,8" | tr "," "." | tr ";" ","` replaces commas with dots and semi-colons with commas.
 - `echo "My ID is 73535" | tr -d [:digit:]` deletes all the digits from the string.

Advanced commands

- `sed` stands for **stream editor** and it can perform lots of functions on file like searching, find and replace, insertion or deletion. We give just an hint of its true power
 - `echo "UNIX is great OS. UNIX is open source." | sed "s/UNIX/Linux/"` replaces the first occurrence of "UNIX" with "Linux".
 - `echo "UNIX is great OS. UNIX is open source." | sed "s/UNIX/Linux/2"` replaces the second occurrence of "UNIX" with "Linux".
 - `echo "UNIX is great OS. UNIX is open source." | sed "s/UNIX/Linux/g"` replaces all occurrences of "UNIX" with "Linux".
 - `echo -e "ABC\nabc" | sed "/abc/d"` delete lines matching "abc".
 - `echo -e "1\n2\n3\n4\n5\n6\n7\n8" | sed "3,6d"` delete lines from 3 to 6.

Advanced commands

- `cut` is a command for cutting out the sections from each line of files and writing the result to standard output.
 - `cut -b 1-3,7- state.txt` cut bytes (`-b`) from 1 to 3 and from 7 to end of the line
 - `echo -e "A,B,C\n1.22,1.2,3\n5,6,7\n9.9999,0,0" | cut -d "," -f 1` get the first column of a CSV (`-d` specifies the column delimiter, `-f n` specifies to pick the *n*-th column from each line)
- `find` is used to find files in specified directories that meet certain conditions. For example:
`find . -type d -name "*lib*"` find all directories (not files) starting from the current one (`.`) whose name contain "lib".
- `locate` is less powerful than `find` but much faster since it relies on a database that is updated on a daily base or manually using the command `updatedb` . For example: `locate -i foo` finds all files or directories whose name contains `foo` ignoring case.

Quotes

Double quotes may be used to identify a string where the variables are interpreted. Single quotes identify a string where variables are not interpreted. Check the output of the following commands

```
a=yes  
echo "$a"  
echo '$a'
```

The output of a command can be converted into a string and assigned to a variable for later reuse:

```
list=`ls -l` # Or, equivalently:  
list=$(ls -l)
```

Processes

- Run a command in background: `./my_command &`
- `ctrl-Z` suspends the current subprocess.
- `jobs` lists all subprocesses running in the background in the terminal.
- `bg %n` reactivates the *n*-th subprocess and sends it to the background.
- `fg %n` brings the *n*-th subprocess back to the foreground.
- `ctrl-C` terminates the subprocess in the foreground (when not trapped).
- `kill pid` sends termination signal to the subprocess with id `pid`. You can get a list of the most computationally expensive processes with `top` and a complete list with `ps aux` (usually `ps aux` is filtered through a pipe with `grep`)

All subprocesses in the background of the terminal are terminated when the terminal is closed (unless launched with `nohup`, but that is another story...)

How to get help

Most commands provide a `-h` or `--help` flag to print a short help information:

```
find -h
```

`man command` prints the documentation manual for command.

There is also an info facility that sometimes provides more information: `info command`.

Introduction to git

Version control

Version control, also known as source control, is the practice of tracking and managing changes to software code. Version control systems are software tools that help software teams manage changes to source code over time.

`git` is a free and open-source version control system, originally created by Linus Torvalds in 2005. Unlike older centralized version control systems such as SVN and CVS, Git is distributed: every developer has the full history of their code repository locally. This makes the initial clone of the repository slower, but subsequent operations dramatically faster.

How does git work?

1. Create (or find) a repository with a git hosting tool (an online platform that hosts your project, like GitHub or Gitlab).
2. `git clone` (download) the repository.
3. `git add` a file to your local repo.
4. `git commit` (save) the changes, this is a local action, the remote repository (the one in the cloud) is still unchanged.
5. `git push` your changes, this action synchronizes your version with the one in the hosting platform.

How does git works? (Collaborative)

If you and your teammates work on different files the workflow is the same as before, you just have to remember to `pull` the changes that your colleagues made.

If you have to work on the same files, the best practice is to create a new `branch`, which is a particular version of the code that branches from the main one. After you have finished working on your feature you `merge` the branch into the main.

Other useful `git` commands

- `git diff` shows the differences between your code and the last commit.
- `git status` lists the status of all the files (e.g. which files have been changed, which are new, which are deleted and which have been added).
- `git log` shows the history of commits.
- `git checkout` switches to a specific commit or branch.
- `git stash` temporarily hides all the modified tracked files.

SSH authentication

1. Sign up for a [GitHub](#) account.
2. [Create a SSH key](#).
3. [Add it to your account](#).
4. Configure your machine:

```
git config --global user.name "Name Surname"  
git config --global user.email "name.surname@email.com"
```

See [here](#) for more details on SSH authentication.

The course repository

Clone the course repository:

```
git clone git@github.com:pcafrica/advanced_programming_2023-2024.git
```

Before every lecture, download the latest updates by running:

```
git pull origin main
```

from inside the cloned folder.

Summary

1. Historical background on C++, Python.
2. The build process:
 - Compiled vs. interpreted languages.
 - Preprocessor, compiler, linker, loader.
3. Introduction to the UNIX shell:
 - What is a shell.
 - Variables.
 - Basic commands and scripting.
4. Introduction to `git`:
 - Local vs. remote.
 - Branching and collaborative working.
 - Sync the course material with your computer.



Warning

Please get your laptop ready by Thursday!



Have a great semester!

Laboratory 01

The UNIX shell.

Advanced Programming - SISSA, UniTS, 2023-2024

Pasquale Claudio Africa

28 Sep 2023

Exercise 1: basic Bash commands

Perform the following tasks in your command-line terminal.

1. Navigate to your home folder.
2. Create a folder named `test1`.
3. Navigate to `test1` and create a new directory `test2`.
4. Navigate to `test2` and go up one directory.
5. Create the following files: `f1.txt`, `f2.txt`, `f3.dat`, `f4.md`, `README.md`, `.hidden`.
6. List all files (including hidden ones).
7. List only `.txt` files.
8. Move `README.md` to folder `test2`.
9. Move all `.txt` files to `test2` in one command.
10. Remove `f3.dat`.
11. Remove all contents of `test1` and the folder itself in one command.

Exercise 2: dataset exploration

You can access an open dataset of logs collected from a high-performance computing cluster at the Los Alamos National Laboratories. The dataset is available on [this webpage](#).

To download the dataset using `wget`, run the following command:

```
 wget https://raw.githubusercontent.com/logpai/loghub/master/HPC/HPC_2k.log_structured.csv
```

After downloading the dataset, perform the following analyses using only Bash commands.

1. Find out how many unique node names are present in the dataset.
2. Export the list from the previous point to a file named `nodes.log`
3. Determine the number of times the "unavailable" event (E13) has been reported.
4. Identify the number of unique nodes that have reported either event E32 or event E33.
5. Calculate how many times the node "gige7" has reported a critical event (E15).
6. Find out how many times the "node-2" node has been reported in the logs.

Exercise 3: creating a backup script

In this exercise, you'll create a Bash script that automates the process of creating a backup of a specified directory. The script should accomplish the following tasks:

1. Receive the directory to backup as an input argument.
2. Create a timestamped backup folder inside a specified backup directory.
3. Copy all files and directories from the user-specified directory to the backup folder.
4. Compress the backup folder into a single archive file.

Note: You can use basic commands like `read` , `mkdir` , `cp` , `tar` , and `echo` .

Hint: Generate a timestamp in the format `YYYYMMDD_hhmmss` with `date +%Y%m%d_%H%M%S` .

Exercise 3: creating a backup script. Instructions

1. Create a new Bash script file named `backup.sh`.
2. Inside the script, use basic Bash commands to implement the following steps:
 - i. Prompt the user to enter the directory they want to back up.
 - ii. Create a timestamped backup folder (e.g., `backup_<timestamp>`) inside a specified backup directory (you can define this directory at the beginning of your script).
 - iii. Copy all files and directories from the user-specified directory to the backup folder.
 - iv. Compress the backup folder into a single archive file `backup_<timestamp>.tar.gz`.
 - v. Display a message indicating the successful completion of the backup process.
3. Test your script by running it in your terminal. Ensure it performs all the specified tasks correctly.
4. **(Bonus)** Implement error handling in your script. For example, check if the specified input directory exists.

Exercise 4: hands on git . Collaborative file management (1/3)

1. Form groups of 2-3 members.
2. Designate one member to create a new repository (visit <https://github.com/> and click the + button in the top right corner), and ensure everyone clones it.
3. In a sequential manner, each group member should create a file with a distinct name and push it to the online repository while the remaining members pull the changes.
4. Repeat step 3, but this time, each participant should modify a different file than the ones modified by the other members of the group.

Exercise 4: hands on git . Collaborative file management (2/3)

Now, let's work on the same file, `main.cpp` . Each person should create a hello world `main.cpp` that includes a personalized greeting with your name. To prevent conflicts, follow these steps:

1. Create a unique branch using the command: `git checkout -b [new_branch]` .
2. Develop your code and push your branch to the online repository.
3. Once everyone has finished their work, merge your branch into the `main` branch using the following commands:

```
git checkout main  
git pull origin main  
git merge [new_branch]  
git push origin main
```

Exercise 4: hands on git . Collaborative file management (3/3)

How to deal with git conflicts

The first person to complete this process will experience no issues. However, subsequent participants may encounter merge conflicts.

Git will mark the conflicting sections in the file. You'll see these sections surrounded by <<<<<< , ====== , and >>>>> markers.

Carefully review the conflicting sections and decide which changes to keep. Remove the conflict markers (<<<<<< , ====== , >>>>>) and make the necessary adjustments to the code to integrate both sets of changes correctly.

After resolving the conflict, commit your changes and push your resolution to the repository.

Lecture 02

Introduction to C++. Built-in data types. Variables, pointers and references. Control structures. Functions.

Advanced Programming - SISSA, UniTS, 2023-2024

Pasquale Claudio Africa, Marco Feder

03 Oct 2023

Why C++?

C++ is:

- Reasonably efficient in terms of CPU time and memory handling, being a **compiled language**.
- High demand in industry.
- A (sometimes exceedingly) complex language: if you know C++ you will learn other languages quickly.
- A **strongly typed**¹ language: safer code, less ambiguous semantic, more efficient memory handling.
- Supporting functional, object-oriented, and generic programming.
- Backward compatible (unlike Python... 😞). Old code compiles (almost) seamlessly.
- 🌲 It is **green!**

¹ Not everybody agrees on the definition of *strongly typed*.

Outline

1. Structure of a basic C++ program
2. Fundamental types
3. Memory management: variables, pointers, references, arrays
4. Conditional statements
5. Functions and operators
6. User-defined types: `enum` , `union` , `struct` , POD structs
7. Declarations and definitions
8. Code organization
9. The build toolchain in practice

Structure of a basic C++ program

Structure of a basic C++ program

- C++ program structure includes a collection of functions.
- Every C++ program must contain one `main()` function, which serves as the entry point.
- Other functions can be defined as needed.
- Statements within functions are enclosed in curly braces `{}`.
- Statements are executed sequentially unless control structures (e.g., loops, conditionals) are used.

Hello world!

```
#include <iostream>

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

- `#include <iostream>` : Includes the Input/Output stream library.
- `int main()` : Entry point of the program.
- `std::cout` : Standard output stream.
- `<<` : Stream insertion operator.
- `"Hello, world!"` : Text to be printed.
- `<< std::endl` : Newline character.
- `return 0;` : Indicates successful program execution.

How to compile and run

To compile the program:

```
g++ hello_world.cpp -o hello_world
```

To run the compiled program:

```
./hello_world
```

To check exit code:

```
echo $?
```

C++ as a strongly typed language

- C++ enforces strict type checking at compile time.
- Variables must be declared with a specific type.
- Type errors are detected and reported at compile time.
- This helps prevent runtime errors and enhances code reliability.

Example

```
int x = 5;
char ch = 'A';
float f = 3.14;

x = 1.6;           // Legal, but truncated to the 'int' 1.
f = "a string"; // Illegal.

unsigned int y{3.0}; // Uniform initialization: illegal.
```

Fundamental types

Fundamental types

Data Type	Size (Bytes)
bool	1
(unsigned) char	1
(unsigned) short	2
(unsigned) int	4
(unsigned) long	4 or 8
(unsigned) long long	8
float	4
double	8
long double	8, 12, or 16

Integer numbers

- C++ provides several integer types with varying sizes.
- Common integer types include `int`, `short`, `long`, and `long long`.
- The range of values that can be stored depends on the type.

Example

```
int age = 30;  
  
short population = 32000;  
  
long long largeNumber = 123456789012345;
```

Floating-point numbers

- C++ supports floating-point types for representing real numbers.
- Common floating-point types include `float`, `double`, and `long double`.
- These types can represent decimal fractions.

Example

```
float pi = 3.14;  
  
double gravity = 9.81;
```

Floating-point arithmetic

Floating-point arithmetic is a method for representing and performing operations on real numbers $\pm f \cdot b^e$ in a binary format (i.e., $b = 2$).

- **Representation:** Floating-point numbers consist of three components: sign s (0: positive, 1: negative), significand f , and exponent e .
- **Normalized numbers:** In normalized form, the most significant bit of the significand is always 1, allowing for a wide range of values to be represented efficiently.
- **IEEE 754 standard:** The most commonly adopted standard for floating-point arithmetic is the [IEEE 754 Standard for Floating-Point Arithmetic](#). This standard specifies the formats, precision, rounding rules, and handling of special values like NaN (Not-a-Number) and infinity.

Floating-point arithmetic limitations

```
double epsilon = 1.0; // Machine epsilon.
```

```
while (1.0 + epsilon != 1.0) {
    epsilon /= 2.0;
}
```

```
double a = 0.1, b = 0.2, c = 0.3;
```

```
if (a + b == c) { // Unsafe comparison.
    // This may not always be true due to precision limitations.
}
```

```
double x = 1.0, y = 1.0 / 3.0;
double sum = y + y + y;
```

```
if (std::abs(x - sum) < tolerance) { // Safer comparison.
    // Use tolerance to handle potential rounding errors.
}
```

Characters and strings

- Characters are represented using the `char` type.
- Strings are sequences of characters and are represented using the `std::string` type.

Example

```
char comma = ',';

std::string name = "John";

std::string greeting = "Hello";

// Concatenate strings.
std::string message = greeting + comma + ' ' + name;
```

Boolean types

- C++ has a built-in Boolean type called `bool`.
- It can have two values: `true` or `false`.
- Useful for conditional statements and logical operations.
- Numbers can be converted to Boolean.

Example

```
bool is_true = true;  
  
bool is_false = false;  
  
if (-1.5) // true.  
    // ...  
  
if (0) // false.  
    // ...
```

Initialization

- Initialization sets the initial value of a variable at the time of declaration.
- C++ supports various forms of initialization, including direct, copy, and list initialization.

Example

```
int x = 5; // Direct initialization.  
int y(10); // Constructor-style initialization.  
int z{15}; // Uniform initialization (preferred).
```

auto

In many situations, the compiler can determine the correct type of an object using the initialization value.

```
auto a{42};           // int.  
auto b{12L};          // long.  
auto c{5.0F};          // float.  
auto d{10.0};          // double.  
auto e{false};         // bool.  
auto f{"string"}; // char[7].  
  
// C++11.  
auto fun1(const int i) -> int { return 2 * i; }  
  
// C++14.  
auto fun2(const int i) { return 2 * i; }
```

Memory management: variables, pointers, references, arrays

Heap vs. stack

- Programs use memory to store data and variables.
- Memory is divided into two main areas: the stack and the heap.

Stack memory

- **Stack:** A region of memory for function call frames.
- Variables stored on the stack have a fixed size and scope.
- Memory is allocated and deallocated automatically.
- Well-suited for small, short-lived variables.

Heap memory

- **Heap:** A region of dynamic memory for data with varying lifetimes.
- Variables on the heap have a dynamic size and longer lifetimes.
- Memory allocation and deallocation are explicit (manual).

Variables and pointers

- Variables stored on the stack are typically accessed directly.
- Pointers to stack variables can be used safely within their scope.
- Heap-allocated variables require pointers for access.
- Pointers to heap variables must be managed carefully.

```
int stack_var = 42; // Stack variable.  
int* stack_ptr = &stack_var; // Pointer to stack variable.  
  
int* heap_ptr = new int(42); // Pointer to heap variable.  
// ...  
delete heap_ptr;  
heap_ptr = nullptr;
```

Lifetime and scope

- Stack variables have a limited lifetime within their scope.
- Heap variables can have a longer lifetime beyond their defining scope.
- Deallocating heap memory is the programmer's responsibility.

Best practices

- Use the stack for small, short-lived variables.
- Use the heap for dynamic data with extended lifetimes.
- Always deallocate heap memory to prevent memory leaks.

Variables

- Variables are named memory locations used to store data.
- They must be declared with a specific type before use.
- Variables can be modified and accessed in your program.

Example

```
int x = 5; // Declaration and initialization.  
x = 10;    // Variable modification.
```

```
int y; // Declaration with default initialization.  
y = 20; // Initialization after declaration.
```

```
const double a = 3.7;  
a = 5; // Error!
```

Pointers

- Pointers are variables that store memory addresses.
- They allow you to work with memory directly.
- Declared using `*` symbol.

Example

```
int number = 42;

int* pointer = &number; // Pointer to 'number'.

// Create a dynamic integer with new.
int* dynamic_variable = new int;
*dynamic_variable = 5;
// Deallocate it.
delete dynamic_variable;
dynamic_variable = nullptr;
```

Pointers: common problems

```
int* arr = new int[5]; // Dynamically allocate an integer array.  
  
// Access and use the array beyond its allocated size.  
for (int i = 0; i <= 5; i++) {  
    arr[i] = i;  
}  
  
// Forgot to delete the dynamically allocated array, causing a memory leak.  
// delete[] arr;  
  
// Attempt to access memory beyond the allocated array's bounds, causing undefined behavior.  
std::cout << arr[10] << std::endl;
```

References

- References provide an alias for an existing variable.
- Declared using `&` symbol.
- Provide an alternative way to access a variable.

Example

```
int a = 10;
int& ref = a; // Reference to 'a'.

ref = 20; // Modifies 'a'.

int b = 10;
ref = b;
ref = 5; // What's now the value of 'a' and 'b'?
```

Arrays

- Arrays are collections of elements of the same type.
- Elements are accessed by their index (position).
- C++ provides the much safer `std::array<type>`, `std::vector<type>`.

Example

```
int numbers[5]; // Array declaration.  
numbers[0] = 1; // Assigning values to elements.  
  
int* dynamic_array = new int[5];  
  
for (int i = 0; i < 5; ++i) {  
    dynamic_array[i] = i * 2;  
}  
  
delete[] dynamic_array;
```

Conditional statements

if ... else if ... else

- Conditional statements allow you to execute different code based on conditions.
- In C++, we use `if`, `else if`, and `else` statements for conditional execution.

Example

```
int x = 10;

if (x > 5) {
    std::cout << "x is greater than 5." << std::endl;
} else if (x > 3) {
    std::cout << "x is greater than 3 but not greater than 5." << std::endl;
} else {
    std::cout << "x is not greater than 5." << std::endl;
}
```

switch ... case

- The `switch` statement is a control flow structure alternative to using multiple `if ... else` statements based on the value of an expression.

Example

```
switch (expression) {  
    case constant1:  
        // Code to execute if expression == constant1.  
        break;  
    case constant2:  
        // Code to execute if expression == constant2.  
        break;  
    // ... more cases ...  
    default:  
        // Code to execute if expression doesn't match any case.  
}
```

Functions and operators

Functions

- Functions are blocks of code that perform a specific task.
- Functions are defined with a return type, name, and parameters.
- They can be called to execute their code.

Example

```
int add(int a, int b) {  
    return a + b;  
}  
  
int result = add(3, 4); // Calling the 'add' function.
```

void

- `void` is a data type that represents the absence of a specific type.
- It indicates that a function does not return any value or that a pointer does not have a defined type.
- Dangerous to use.

Example

```
void greet() {  
    std::cout << "Hello, world!" << std::endl;  
}  
  
void* generic_ptr;  
int x = 10;  
  
generic_ptr = &x; // Can point to any data type.
```

Pass by value vs. pointer vs. reference (1/2)

```
void modify_by_copy(int x) {
    // Creates a copy of 'x' inside the function.
    x = 20; // Changes the copy 'x', not the original value.
}

void modify_by_ptr(int* ptr) {
    *ptr = 30; // Modifies the original value via the pointer.
}

void modify_by_ref(int& ref) {
    ref = 40; // Modifies the original value through the reference.
}
```

Pass by value vs. pointer vs. reference (2/2)

```
int value = 10;

modify_by_copy(value); // Pass by value.
std::cout << value << std::endl; // Output: 10.

modify_by_ptr(&value); // Pass by pointer
std::cout << value << std::endl; // Output: 30.

modify_by_ref(value); // Pass by reference
std::cout << value << std::endl; // Output: 40.
```

Best practices

- Pass by value for small, non-mutable data.
- Pass by pointer for modifying values or working with arrays.
- Pass by reference for efficiency and direct modification of values.

Return by value vs. pointer vs. reference (1/2)

```
int get_copy() {
    return 42; // Return a copy of the value.
}

int* get_ptr() {
    int* arr = new int[5];
    // ...
    return arr; // Return a pointer to the array.
}

int& get_ref() {
    static int value = 10; // Beware: if not static, undefined behavior.
    return value; // Return a reference to 'value'.
}
```

Return by value vs. pointer vs. reference (2/2)

```
int result1 = get_copy(); // Return by value.  
  
int* result2 = get_ptr(); // Return by pointer.  
result2[2] = 5;  
delete[] result2; // Beware: memory leaks.  
  
int& result3 = get_ref(); // Return by reference.  
result3 = 20;
```

Best practices

- Return by value for small, non-mutable data.
- Return by pointer for dynamically allocated data.
- Return by reference for efficiency and direct modification of data.

const correctness (1/2)

```
void print_value(const int x) {
    // x = 42; // Error: Cannot modify 'x'.
}

const int get_copy() {
    const int x = 42;
    return x;
}

int result = get_copy();
result = 10; // Safe, it's a copy!

const int age = 30; // Immutable variable.
const int* ptr_to_const = &age; // Pointer to an integer which is constant.

ptr_to_const = &result; // Now pointing to another variable.
*ptr_to_const = 42; // Error: cannot modify pointed object.
```

Question: how to declare a constant pointer to a non-constant `int` ?

`const` correctness (2/2)

Benefits

- Prevents unintended modifications: Helps avoid accidental data modifications, enhancing code safety.
- Self-documenting code: Makes code more self-documenting by indicating the intent of data usage.
- Compiler optimizations: Allows the compiler to perform certain optimizations, as it knows that `const` data won't change.

Best practices

- Const correctness is a valuable practice for writing safe and maintainable C++ code.
- Use `const` to indicate read-only data and functions.
- Incorrectly using `const` can lead to compiler errors or unexpected behavior.

Operators

- Operators are symbols used to perform operations on variables and values.
- Arithmetic operators: +, -, *, /, %
- Arithmetic and assignment operators: +=, -=, *=, /=, %=
- Comparison operators: ==, !=, <, >, <=, >=, <=> (C++20)
- Logical operators: &&, ||, !

Example

```
int x = 5, y = 3;
bool is_true = (x > y) && (x != 0); // Logical expression.
int z = (x > y) ? 2 : 1; // Ternary operator.

x += 2; // 7.
y *= 4; // 12.
z /= 2; // 1.
```

Increment operators

1. Pre-increment (`++var`):

- Increases the variable's value before using it.
- The updated value is immediately reflected. No temporary needed: **more efficient**.

2. Post-increment (`var++`):

- Uses the current value of the variable before incrementing.
- The variable's value is increased after its current value is used.

```
int a = 5;  
int b = ++a; // Pre-increment.  
// a is now 6, b is also 6.
```

```
int c = a++; // Post-increment.  
// a is now 7, but c is 6.
```

Function overloading

- Function overloading is a feature in C++ that allows you to define multiple functions with the same name but different parameters.
- The compiler selects the appropriate function based on the number or types of arguments during the function call.

```
void print(int x) {
    std::cout << "Integer value: " << x << std::endl;
}

double print(double x) {
    std::cout << "Double value: " << x << std::endl;
}

print(3); // Calls the int version.
print(2.5); // Calls the double version.
```

User-defined types

enum

- Enumerations (enums) allow you to define a set of named values.
- Enums provide a way to create user-defined data types.

Example

```
enum Color : unsigned int {  
    Red = 0,  
    Green,  
    Blue  
};  
  
Color my_color = Green;
```

union

- Unions allow you to define a type that can hold different data types.
- Only one member of a union can be accessed at a time.
- Useful for optimizing memory usage.

Example

```
union Duration {  
    int seconds;  
    short hours;  
};  
  
Duration d;  
d.seconds = 259200;  
  
short h = d.hours; // Contains garbage: undefined behavior.
```

struct

- Structs (structures) allow you to group related data members into a single unit.
- Members can have different data types.
- Structs provide a way to create custom data structures.

Example

```
struct Point {  
    int x;  
    int y;  
};
```

```
Point p;  
p.x = 3;  
p.y = 5;
```

Actually, in C++ `struct` is just a special type of `class`. When Referring to C-style structs, a more proper name would be **Plain Old Data (POD) structs**.

Plain Old Data (POD) structs

- POD structs are classes with simple data members and no user-defined constructors or destructors.
- They have C-like semantics and can be used in low-level operations.

Example

```
struct Rectangle {  
    double width;  
    double height;  
};  
  
Rectangle r;  
r.width = 10;  
r.height = 20;  
  
Rectangle s{5, 10};  
Rectangle t = s; // POD structs are trivially copyable.
```

Looking towards classes

- Object-oriented programming (OOP) is a programming paradigm that uses classes and objects.
- C++ is an object-oriented language that supports OOP principles.
- Classes are user-defined data types that encapsulate data and behavior.
- Classes extend structs by including **member functions** other than data.
- OOP promotes code reusability, modularity, and organization.

Declarations and definitions

Declaration

- Declarations inform the compiler about the existence of variables or functions.
- They provide type information but do not allocate memory or provide implementation.

```
int x; // Declaration of 'x'.
extern int y; // Declaration of 'y'.
struct X; // Forward-declaration.
```

Definition

- Definitions provide the actual implementation of variables or functions.
- They allocate memory for variables or specify the behavior of functions.

```
int x = 5; // Definition of 'x'.
```

Declaring functions

- Function declarations provide enough information for the compiler to use the function.
- They specify the return type, name, and parameter types.
- Function declarations are typically placed in header files.

Example

```
int add(int a, int b); // Declaration of 'add' function.
```

Defining functions

- Function definitions specify the implementation of a function.
- They include the function's return type, name, parameters, and code block.
- They are typically placed in source files.

Example

```
int add(int a, int b) { // Definition of 'add' function.  
    return a + b;  
}
```

Code organization

Modular programming

- Modular programming divides code into separate modules or units.
- Each module focuses on a specific task or functionality.
- Benefits:
 - Improved code organization and readability
 - Easier maintenance and debugging
 - Code reusability
 - Encapsulation of functionality

Building blocks of C++ code modules

- C++ code modules consist of:
- Header files (`.h` or `.hpp`) for declarations
- Source files (`.cpp`) for definitions
- Implementation files (`.cpp`) for non-template classes
- Header files contain function prototypes and class declarations.
- Source files contain function and class definitions.

Header files

- Header files (`.h` or `.hpp`) contain declarations and prototypes.
- They define the interface to a module or class.
- Header files are included in source files to access declarations.

```
// my_module.hpp
int add(int a, int b); // Function prototype.
```

Best practices

- Use include guards or `#pragma once` to prevent multiple inclusions.
- Include only necessary headers to reduce compilation time.
- Keep header files concise and focused on declarations.
- Use descriptive and unique names for header files.
- Document complex or non-obvious declarations.

Source files

- Source files (.cpp) contain the definitions of functions and classes.
- They implement the functionality declared in header files.
- Source files include header files for access to declarations.

```
// my_module.cpp
#include "my_module.hpp" // Include the corresponding header.

int add(int a, int b) {
    return a + b;
}
```

The need for header guards

- Header guards (or include guards) prevent multiple inclusions of the same header file.
- They ensure that a header file is included only once during compilation.
- Header guards are essential to avoid redefinition errors.

Without header guards, if a header file is included multiple times in a source file or across multiple source files, it can lead to redefinition errors.

How to implement header guards

- Place `#ifndef`, `#define`, and `#endif` or `#pragma once` directives in the header file.
- Use a unique identifier (usually based on the filename) as the guard symbol.

Example (file `my_module.h`):

```
#ifndef MY_MODULE_H_
#define MY_MODULE_H_

// ...

#endif
```

Modern compilers also support:

```
#pragma once

// ...
```

Preventing header file inclusion issues

To avoid issues with header file inclusions:

- Include necessary headers in your source files.
- Avoid circular dependencies (A includes B, and B includes A).
- Use forward declarations when possible to minimize dependencies.
- Follow a consistent naming convention for header guards.

Managing scope in C++

- Scope determines the visibility and lifetime of variables and functions.
- C++ uses blocks, functions, and namespaces to manage scope.
- Variables declared inside a block have block scope.
- Variables declared outside of any function or class have global scope.
- Namespaces help organize code and avoid naming conflicts.

```
int x = 10;

{ // Manually define a scope.
    int y = 20;
    //
}
// Destroy all variables local to the scope.
// Beware: dynamically allocated variables must be deleted manually.

std::cout << y << std::endl; // Error: 'y' is undefined here.
```

Using namespaces for organization

- Namespaces group related declarations to avoid naming collisions.
- They provide a way to organize code into logical units.
- Namespace members are accessed using the `::` operator.
- Example of using a namespace:

```
namespace Math {  
    int add(int a, int b) {  
        return a + b;  
    }  
}  
  
int result1 = Math::add(3, 4); // Accessing a namespace member.  
  
using namespace Math; // Useful, but dangerous due to possible name clashes.  
int result2 = add(3, 4);
```

The build toolchain in practice

Preprocessor and compiler

- The preprocessor (`cpp`) handles preprocessing directives.
- It includes headers, performs macro substitution, and removes comments.
- The compiler (`g++` , `clang++`) translates source code into object files.
- Preprocessor and compiler commands are combined when you run `g++` or `clang++`.

Example (project with three files: `module.hpp`, `module.cpp`, `main.cpp`):

```
# Preprocessor.  
g++ -E module.cpp -I/path/to/include/dir -o module_preprocessed.cpp  
g++ -E main.cpp -I/path/to/include/dir -o main_preprocessed.cpp  
  
# Compiler.  
g++ -c module_preprocessed.cpp -o module.o  
g++ -c main_preprocessed.cpp -o main.o
```

Linker

- The linker (`ld`) combines object files and resolves external references.
- It creates an executable program from multiple object files.
- Linker errors occur if functions or variables are not defined.

Example

```
g++ module.o main.o -o my_program
```

Link against an external library:

```
g++ module.o main.o -o my_program -lmy_lib -L/path/to/my/lib
```

In this example, the `-lmy_lib` flag is used to link against the library `libmy_lib.so`. The `-l` flag is followed by the library name without the `lib` prefix and without the file extension `.so` (dynamic) or `.a` (static).

Loader

- The loader loads the executable program into memory for execution.
- It allocates memory for the program's data and code sections.
- The operating system's loader handles this task.

Example

```
./my_program
```

If linked against an external dynamic library, the loader has to know where it is located. The list of directories where to find dynamic libraries is contained in the colon-separated environment variable `LD_LIBRARY_PATH`.

```
export LD_LIBRARY_PATH+=:/path/to/my/lib  
./my_program
```



Classes and object-oriented programming

Laboratory 02

Introduction to C++.

Advanced Programming - SISSA, UniTS, 2023-2024

Pasquale Claudio Africa

05 Oct 2023

Exercise 1: control structures

Write a C++ program `temperature_converter` that converts a given temperature from Celsius to Fahrenheit or viceversa.

- The program takes an input number from the user and the input unit as a string (or character).
- Use conditional statements to check if the user has provided a temperature in Fahrenheit or Celsius and print the corresponding output.
- The conversion formulas are:

$$T_{\text{Fahrenheit}} = \frac{9}{5}T_{\text{Celsius}} + 32.$$

$$T_{\text{Celsius}} = \frac{5}{9}(T_{\text{Fahrenheit}} - 32).$$

Exercise 2: memory management

Create a C++ program that dynamically allocates memory for an array of integers.

- Allow the user to specify the size of the array.
- Fill the array with random integers.
- Write a function to find and display the maximum and minimum values in the array. **Hint:** the maximum and minimum value are stored in two variables passed as references to this function.

Exercise 3: complete the missing statistics calculator

You are provided with a partially implemented C++ program for calculating and displaying statistics for a set of numbers. Your task is to fill in the missing parts to make the program functional.

- Use the provided function prototypes in `statistics.hpp` to guide your implementation in `statistics.cpp`.
- You may need to sort the input numbers to calculate the median. You can use standard C++ functions like `std::sort` for this purpose.
- Implement a loop in `main.cpp` to allow the user to calculate statistics for multiple sets of numbers without exiting the program.

Exercise 4: struct

- Define a `struct` called `Student` that represents information about a student, including their name, age, and grade average.
- Create a `std::vector` of 5 `Student` objects and initialize them with sample data.
- Write a function to display the information of all students in the array.
- Organize your code by separating the struct definition, data initialization, and display function into different files or modules.

Exercise 5: code organization

Write a C++ program that simulates a simple calculator. Define functions for addition, subtraction, multiplication, and division. Allow the user to enter two numbers and choose an operation. Perform the chosen operation and display the result.

Here's a breakdown of the project structure:

- `calculator/`
 - `src/`
 - `main.cpp`
 - `calculator.cpp`
 - `include/`
 - `calculator.hpp`
 - `build/` (build artifacts, such as object files and executables).
 - `build.sh` (a Bash script that compiles and properly links the code)

Lecture 03

Object oriented programming. Classes and access control in C++.

Advanced Programming - SISSA, UniTS, 2023-2024

Pasquale Claudio Africa

10 Oct 2023

Outline

1. Introduction to Object-Oriented Programming (OOP)
2. Classes and objects in C++
3. Notes on code organization
4. Encapsulation and access control
5. Operator overloading
6. Class collaborations

Introduction to Object-Oriented Programming (OOP)

What is OOP?

- OOP is a programming paradigm that revolves around objects.
- Objects represent instances of classes, encapsulating data **and behavior**.
- Key principles include encapsulation, inheritance, and polymorphism.

OOP in C++

- C++ is not (only) a OOP language.
- C++ is a multi-paradigm programming language that supports procedural, object-oriented, and generic programming.
- C++ allows developers to combine these paradigms effectively for various programming tasks.

Key principles of OOP

Object-Oriented Programming (OOP) is a programming paradigm that emphasizes the use of objects to represent real-world entities and concepts. It is based on several key principles:

- **Encapsulation:** Encapsulation bundles data (attributes) and the functions (methods) that operate on the data into a single unit called an *object*. This promotes data hiding and reduces the complexity of the code.
- **Inheritance:** Inheritance allows you to create new classes (derived or child classes) based on existing classes (base or parent classes). It enables code reuse and the creation of class hierarchies.
- **Polymorphism:** Polymorphism allows objects of different classes to be treated as objects of a common base class. It promotes code flexibility and the ability to work with objects at a higher level of abstraction.

RAII idiom (Resource Acquisition Is Initialization)

Holding a resource is a class invariant, tightly bound to the object's lifetime

RAII idiom:

1. Encapsulate a resource within a class (constructor).
2. Utilize the resource through a local instance of the class.
3. Automatically release the resource when the object goes out of scope (destructor).

Implications

1. C++ does not rely on a garbage collector.
2. Resource management becomes the programmer's responsibility.

Advantages of OOP

OOP offers numerous advantages, including:

- **Modularity:** OOP encourages the division of a complex system into smaller, manageable objects, promoting code modularity and reusability.
- **Maintenance:** Objects are self-contained, making it easier to maintain and update specific parts of the code without affecting other parts.
- **Flexibility:** Inheritance and polymorphism provide flexibility, allowing you to extend and modify the behavior of classes without altering their existing code.
- **Readability:** OOP promotes code readability by organizing data and functions related to a specific object within a class.

Classes and objects in C++

Creating objects (1/2)

In C++, objects are instances of classes. Here's an example of creating and using a `Car` object:

```
class Car {  
public:  
    std::string manufacturer;  
    std::string model;  
    unsigned int year;  
  
    void start_engine() {  
        std::cout << "Engine started!" << std::endl;  
    }  
};
```

Creating objects (2/2)

```
// Access by direct instance.  
Car my_car; // Creating an object of class Car.  
  
my_car.manufacturer = "Toyota";  
my_car.model = "Camry";  
my_car.year = 2023;  
  
my_car.start_engine(); // Invoking a method.  
  
// Access by pointer.  
// This works also for dynamically allocated objects.  
Car* my_car_ptr;  
  
my_car_ptr->manufacturer = "Alfa Romeo";  
my_car_ptr->model = "Giulietta";  
my_car_ptr->year = 2010;  
  
my_car_ptr->start_engine(); // Invoking a method.
```

Members

- Member variables, also known as *attributes* or instance variables, store data within a class. In the `Car` class, `manufacter`, `model`, and `year` are member variables that hold information about the car. These variables encapsulate the car's characteristics within the class.
- Member functions, or *methods*, define the behavior of a class. The `start_engine` method in the `Car` class initiates the car's engine. Methods encapsulate the actions or operations that can be performed on the object's data.

Static members

Static members in a class are shared among all instances of that class. They are declared using the `static` keyword and can be accessed using the class name rather than an object. Static members are useful for maintaining shared data or functionality across objects.

static members

```
class Circle {
public:
    static const double PI = 3.14159265359; // Static constant shared by all Circle objects.
    double radius;

    double calculate_area() {
        return PI * radius * radius;
    }

    static void print_shape_name() {
        std::cout << "This is a circle." << std::endl;
    }
};

Circle circle;
circle.radius = 5.0;

const double area = circle.calculate_area(); // Accessing a non-static member.
const double pi_value = Circle::PI; // Accessing a static member.
Circle::print_shape_name();
```

const members (1/2)

When used in the context of classes, `const` can be applied to member variables, member functions, and even to the class itself.

```
class MyClass {  
public:  
    MyClass(int x) : value(x) {} // Constructor initializes the const member.  
  
    void print_value() const {  
        // value *= 2; // Illegal!  
        std::cout << "Const version: " << value << std::endl;  
    }  
  
    const int value;  
};
```

⚠ If you have a `const` member function but need to modify a member variable, you can declare that variable as `mutable`.

const members (2/2)

```
class MyClass {  
public:  
    void print() {  
        std::cout << "Non-const version" << std::endl;  
    }  
  
    void print() const {  
        std::cout << "Const version" << std::endl;  
    }  
};  
  
MyClass obj1;          // Create a non-const object.  
const MyClass obj2; // Create a const object.  
  
obj1.print(); // Calls the non-const version.  
obj2.print(); // Calls the const version.
```

The `this` pointer

The `this` pointer is a special keyword in C++ that represents a pointer to the current instance of a class. It is a hidden argument to all non-static member functions and is automatically passed to those functions by the compiler.

It allows to **access members** of an object from within its member functions. It helps **resolve ambiguity** and allows you to access the class's members within its member functions, by allowing to distinguish between the local variables and member variables of a class when they have the same name.

```
class MyClass {  
public:  
    int x;  
  
    void print_x() const {  
        std::cout << "Value of x: " << this->x << std::endl; // Using this pointer with the arrow operator.  
    }  
};
```

Constructors

Constructors are special member functions that initialize objects when they are created. They have the same name as the class and can take arguments to set initial values for member variables.

Types of constructor

- **Default constructor:** It takes no arguments. If you don't provide any constructors for a class, C++ will generate a default constructor automatically using default values (e.g., zero for numbers, empty for strings).
- **Parameterized constructor:** It takes one or more parameters to initialize member variables based on the provided values. It creates objects with specific initial states.
- **Copy constructor:** It creates a new object as a copy of an existing object of the same class. It takes a reference to an object of the same class as a parameter. It is invoked when objects are copied, passed by value, or initialized with other objects.

Default constructor

```
class MyClass {  
public:  
    // Default constructor  
    MyClass() {  
        // Initialization code (if needed).  
    }  
  
    // Or:  
    // MyClass() = default;  
  
    std::string name;  
    unsigned int length;  
};  
  
MyClass obj;      // Direct initialization.  
MyClass obj2{};  // Uniform initialization (preferred).  
  
MyClass obj3();  // Illegal: the compiler believes we are declaring a function.
```

Parametrized constructors

```
class Student {  
public:  
    Student(std::string name, unsigned int age) {  
        this->name = name;  
        this->age = age;  
    }  
  
    void display_info() const {  
        std::cout << "Name: " << name << ", Age: " << age << std::endl;  
    }  
  
    std::string name;  
    unsigned int age;  
};  
  
Student student1("Alice", 20); // Creating an object and initializing it using a constructor.  
student1.display_info();  
  
Student student2{"Bob", 23}; // Uniform initialization.  
student2.display_info();
```

Initializer list

An initializer list is used within a constructor to initialize member variables before entering the constructor body. It is a recommended practice, especially for initializing member objects or constants, as it can improve performance.

```
class Rectangle {  
public:  
    Rectangle(double length, double width) : length(length), width(width) {  
        // Constructor body (if needed).  
    }  
  
    double calculate_area() const {  
        return length * width;  
    }  
  
    double length;  
    double width;  
};  
  
Rectangle rectangle{5.0, 3.0}; // Creating an object and initializing it using an initializer list.  
const double area = rectangle.calculate_area();
```

Copy constructor and copy assignment

```
class Book {
public:
    Book(std::string title, std::string author) : title(title), author(author) {}

    // Copy constructor.
    Book(const Book& other) : title(other.title), author(other.author) {}

    // Copy assignment operator.
    Book& operator=(const Book& other) {
        if (this != &other) {
            title = other.title;
            author = other.author;
        }
        return *this;
    }

    void display_info() const {
        std::cout << "Title: " << title << ", Author: " << author << std::endl;
    }

    std::string title;
    std::string author;
};

Book book1{"The catcher in the rye", "J.D. Salinger"};
Book book2 = book1; // Copying using the copy constructor.
Book book3{"Marcovaldo", "I. Calvino"};
book3 = book1; // Copying using the copy assignment operator.
```

Default constructor and default initialization

- If you don't provide any constructors for a class, C++ will automatically generate a default constructor. However, if you provide any custom constructors, the default constructor won't be generated unless you explicitly define it.
- Default initialization of primitive types (e.g., `int`, `double`) sets them to zero, while non-primitive types (e.g., objects, strings) may have default constructors that initialize them to appropriate default values.

When constructors are called (1/2)

- Object creation:** When you create an object of a class using its constructor, the constructor is called.

```
MyClass obj1;    // Calls the default constructor.  
MyClass obj2{}; // Calls the default constructor.  
Student student{"Alice", 20}; // Calls the parameterized constructor.
```

- Copy initialization:** When you initialize one object with another, the copy constructor is called.

```
MyClass obj1 = obj2; // Calls the copy constructor.
```

When constructors are called (2/2)

3. **Pass and return by value:** When you pass an object by value to a function or return an object by value from a function, the copy constructor is called.

```
void some_function(Student s) {
    // Calls the copy constructor when s is passed.
}

Student create_student() {
    Student s{"Bob", 22};
    return s; // Calls the copy constructor when s is returned.
}
```

4. **Dynamic object creation:** When you create objects dynamically using new, the constructor is called.

```
MyClass* ptr = new MyClass{}; // Calls the default constructor.
```

Destructor (1/2)

A destructor is another special member function that is used to clean up resources held by an object before it goes out of scope or is explicitly deleted. Destructors have the same name as the class but preceded by a tilde (`~`). They are called automatically when an object's lifetime ends.

Rule of three

If a class defines (or deletes) one of the three special member functions:

- destructor
- copy constructor
- copy assignment operator

then it should probably provide all three of them.

Destructor (2/2)

```
class FileHandler {  
public:  
    FileHandler(std::string filename) : filename(filename) {  
        file.open(filename);  
    }  
  
    ~FileHandler() {  
        if (file.is_open()) {  
            file.close();  
        }  
    }  
  
    std::string filename;  
    std::ofstream file;  
};  
  
{  
    FileHandler file{"data.txt"}; // Automatically destroyed when going out of scope.  
} // When going out of scope, destructor is called, and the file is closed.
```

Constructors and destructor implicitly declared by the compilers

Source: <https://howardhinnant.github.io/classdecl.html>

Notes on code organization

The `inline` directive

In C++, the `inline` keyword can be applied to free functions (functions that are not members of any class) to suggest that the function should be inlined by the compiler. This means that the compiler replaces function calls with the actual function code at the call site, potentially leading to better performance, especially for small, frequently used functions.

```
// Inline function declaration for a free function.  
inline int add(int a, int b) {  
    return a + b;  
}  
  
const int result = add(5, 7); // Calls the inline function.  
std::cout << "Result: " << result << std::endl;
```

Best practices (1/2)

- **Function size:** Inlining is most effective for small functions. For larger functions, inlining can lead to code bloat and may not improve performance.
- **Compiler's discretion:** The `inline` keyword is a *suggestion* to the compiler, and the compiler can choose whether or not to inline the function based on optimization settings and other factors.
- **Header files:** If you define `inline` functions in header files, be cautious about including the same header in multiple source files. It can lead to multiple definitions if not managed properly. Using header guards (see Lecture 02) helps prevent this issue.
- **Balancing readability:** While inlining can improve performance, it should be used judiciously. Overusing inline for functions that don't provide significant performance benefits can lead to less readable code due to code duplication.

Best practices (2/2)

Pros of using `inline`

- **Potential performance improvement:** Inlining small functions can eliminate the function call overhead and improve runtime performance.
- **Avoiding multiple definitions:** When the same inline function is defined in multiple translation units (source files), the One Definition Rule (ODR) allows the multiple definitions to be treated as equivalent, which avoids linker errors.

In summary, you can use the `inline` keyword to suggest to the compiler that it should consider inlining the function for potential performance improvement. However, it's essential to balance performance considerations with code readability and maintainability.

Where to define class member functions?

In C++, member functions of a class can be defined either in-class (inline) or out of class. Each approach has its use cases and implications.

In-class (inline) definition (1/3)

Member functions are defined within the class declaration itself, typically in the header file. This is common for short, simple functions that are typically one-liners or very concise.

```
// my_class.hpp
class MyClass {
public:
    int add(int a, int b) // inline keyword is implicit here.
    {
        return a + b;
    }
};
```

In-class definition (2/3)

The previous code is equivalent to the following:

```
// my_class.hpp

class MyClass {
public:
    int add(int a, int b);
};

int MyClass::add(int a, int b) // inline keyword is implicit here.
{
    return a + b;
}
```

In-class definition (3/3)

Pros

- Compact and concise code.
- Compiler may choose to inline the function for performance.

Cons

- May lead to code bloat if used extensively with large functions.
- Changes to the function may necessitate recompilation of all translation units that include the header.

Out of class definition (1/2)

Member functions are declared in the class declaration (in the header file) and defined separately in the source file (.cpp file). Typically used for functions with larger implementations or when you want to separate interface from implementation.

```
// my_class.hpp

class MyClass {
public:
    int add(int a, int b);
};

// my_class.cpp

#include "my_class.h"

int MyClass::add(int a, int b) {
    return a + b;
}
```

Out of class definition (2/2)

Pros

- Separation of interface from implementation for cleaner code organization.
- Changes to the function implementation do not require recompilation of all translation units that include the header.

Cons

- Slightly more verbose in terms of code.
- Requires separate source file for function definitions.

Best practices

1. Use in-class (`inline`) definitions for very short and simple functions (e.g., accessors, mutators) to potentially benefit from inlining.
2. Use out of class definitions for larger or more complex functions to keep the header files clean and to separate interface from implementation.
3. Consider code readability and maintainability when making a choice.

In practice, a combination of both in-class and out-of-class definitions is often used, with the goal of keeping the code organized, maintainable, and efficient.

Encapsulation and access control

Data encapsulation

Data encapsulation is a fundamental concept in OOP that involves bundling data (attributes) and methods (functions) that operate on that data into a single unit called an object. Encapsulation helps hide the internal details of an object and exposes only the necessary functionality through well-defined interfaces.

```
class BankAccount {  
public:  
    BankAccount(std::string account_holder, double balance) : account_holder(account_holder), balance(balance) {}  
  
    void deposit(double amount) {  
        balance += amount;  
    }  
  
    double get_balance() const {  
        return balance;  
    }  
  
private:  
    std::string account_holder;  
    double balance;  
};
```

Access specifiers (1/2)

C++ provides access specifiers to control the visibility and accessibility of class members (variables and methods). These access specifiers enforce encapsulation and access control within the class.

- `public` : Members declared as public are accessible from any part of the program. They form the class's public interface.
- `private` : Members declared as private are not accessible from outside the class. They are used for internal implementation details.
- `protected` : Members declared as protected are accessible within the class and by derived classes (in **inheritance** scenarios).

 **Inheritance will be covered in the next lecture!**

Access specifiers (2/2)

```
class MyClass {  
public:  
    int public_var;      // Public member variable.  
    void public_func() // Public member function.  
    {  
        // ...  
    }  
  
private:  
    int private_var;    // Private member variable.  
    void private_func() // Private member function.  
    {  
        // ...  
    }  
};
```

class vs. struct

In C++, both `class` and `struct` are used to define classes. The only difference between them is the default access specifier:

- In a `class`, members are private by default.
- In a `struct`, members are public by default.

```
class MyClass {  
    int x; // Private by default.  
public:  
    int y; // Public.  
};  
  
struct MyStruct {  
    int a; // Public by default.  
private:  
    int b; // Private.  
};
```

Getter and setter methods (1/2)

Getter and setter methods, also known as accessors and mutators, are used to control access to private member variables.

- **Getter** methods allow reading the values of private variables.
- **Setter** methods enable modifying those values in a controlled manner.

They are commonly used for encapsulation and access control.

Getter and setter methods (2/2)

```
class TemperatureSensor {  
public:  
    double get_temperature() const {  
        return temperature;  
    }  
  
    void set_temperature(double newTemperature) {  
        if (newTemperature >= -50.0 && newTemperature <= 150.0) {  
            temperature = newTemperature;  
        } else {  
            std::cout << "Invalid temperature value!" << std::endl;  
        }  
    }  
  
private:  
    double temperature;  
};
```

friend classes (1/2)

A `friend` class is a class that is granted access to the private members of another class. This access allows the `friend` class to operate on the private members of the class it is friends with.

```
class Circle {  
public:  
    friend class Cylinder; // Cylinder class is a friend of Circle.  
  
    Circle(double r) : radius(r) {}  
  
    double get_area() const {  
        return 3.14159265359 * radius * radius;  
    }  
  
private:  
    double radius;  
};
```

friend classes (2/2)

```
class Cylinder {  
public:  
    double get_volume(const Circle& circle) const {  
        // Accessing the private member 'radius' of the Circle class.  
        return circle.radius * circle.radius * height;  
    }  
  
private:  
    double height;  
};  
  
Circle circle{5.0};  
Cylinder cylinder;  
const double volume = cylinder.get_volume(circle); // Cylinder accesses Circle's private member 'radius'.
```

Operator overloading

Operator overloading (1/2)

Operator overloading is a feature in C++ that allows you to define custom behaviors for operators when used with objects of your own class. In essence, it enables you to extend the functionality of operators beyond their predefined meanings, making objects of your class work with operators in a way that makes sense for your class's context.

Why use operator overloading?

Operator overloading can improve code readability and maintainability by allowing you to write more natural and expressive code. It lets you use operators like `+`, `-`, `*`, `/`, and others to perform operations specific to your class, just as you would with built-in data types.

Operator overloading (2/2)

```
class Complex {  
public:  
    double real;  
    double imag;  
  
    Complex operator+(const Complex& other) {  
        Complex result;  
        result.real = this->real + other.real;  
        result.imag = this->imag + other.imag;  
        return result;  
    }  
};  
  
Complex a{2.0, 3.0};  
Complex b{1.0, 2.0};  
Complex c = a + b; // Using the overloaded '+' operator.
```

Commonly overloaded operators

While you can overload many C++ operators, here are some of the most commonly overloaded operators:

- Arithmetic operators: `+`, `-`, `*`, `/`, `%`, etc.
- Comparison operators: `==`, `!=`, `<`, `>`, `<=`, `>=`, `<=>` (since C++20), etc.
- Assignment operators: `=`, `+=`, `-=`, etc.
- Increment/decrement operators: `++`, `--`.
- Stream insertion/extraction operators: `<<`, `>>` (used for input and output).
- Function call operator: `()` (used to create objects that act like functions).
- Subscript operator: `[]` (used to access elements of an array-like class).
- Member access operator: `->` (used to access members of an object through a pointer).

Overloading as a member vs. non-member function

You can overload operators as member functions or non-member functions.

- When overloaded as a **member function**, the left operand is an object of the class, and the right operand is passed as a parameter.
- When overloaded as a **non-member function**, both operands are passed as parameters. This is often preferred when the left operand is not an object of the class you're overloading the operator for. Sometimes, you may need to access private members of a class when overloading an operator. In such cases, you can declare the overloaded operator function as a friend of the class. This allows the operator function to access the private members of the class.

friend functions

```
class MyClass {  
public:  
    MyClass(int v) : value(v) {}  
  
    // Declaring the '<<' operator as a friend function.  
    friend std::ostream& operator<<(std::ostream& os, const MyClass& obj);  
  
private:  
    int value;  
};  
  
// Overloading the '<<' operator as a non-member function (outside the class).  
std::ostream& operator<<(std::ostream& os, const MyClass& obj) {  
    os << obj.value;  
    return os;  
}  
  
MyClass obj;  
std::cout << obj << std::endl;
```

Operator overloading: best practices

- 1. Operators that cannot be overloaded:** Some operators, like `::`, `.*`, and `? :`, cannot be overloaded.
- 2. Don't change the basic meaning of an operator:** Overloading should make sense in the context of your class. For example, overloading `+` for string concatenation is intuitive, but overloading it for subtraction is not.
- 3. Be mindful of operator precedence and associativity:** Overloaded operators should follow the same precedence and associativity rules as their built-in counterparts (such as in expressions like `2 * 3 + 1`).
- 4. Avoid excessive overloading:** Overloading too many operators can make your code less readable and harder to maintain. Focus on overloading the operators that provide significant benefits.

Class collaborations

Relationships between classes (1/2)

Classes can have various relationships, including:

- 1. Association:** A loose relationship where classes are related, but one does not necessarily contain the other. For example, a `Student` class may be associated with a `Course` or a `Teacher` class.
- 2. Aggregation:** A "has-a" relationship where one class contains another as a part, but the contained object can exist independently. For example, a `University` class may aggregate `Department` classes.

Relationships between classes (2/2)

3. **Composition:** A stronger "*whole-part*" relationship where one class contains another as a part, and the part cannot exist independently. For example, an `Apartment` class composes a `Room` class.
4. **Inheritance:** Inheritance represents an "*is-a*" relationship, where one class (the derived or subclass) inherits properties and behaviors from another class (the base or superclass). This is a fundamental form of collaboration in object-oriented programming.

Association (1/3)

Association is a relationship between two or more classes that defines how they are related to each other. It represents a general form of relationship between classes.

```
class Student; // Forward declaration.

class Course {
public:
    Course(const std::string& name) : course_name(name) {}

    const std::string& get_course_name() const {
        return course_name;
    }

private:
    std::string course_name;
};
```

Association (2/3)

```
class Student {
public:
    Student(const std::string& name) : student_name(name) {}

    void enroll_course(Course* course) {
        // Append new element to the vector.
        enrolled_courses.push_back(course);
    }

    void list_enrolled_courses() const {
        std::cout << student_name << " is enrolled in the following courses:" << std::endl;
        for (const auto& course : enrolled_courses) {
            std::cout << "- " << course->get_course_name() << std::endl;
        }
    }
private:
    std::string student_name;
    std::vector<Course*> enrolled_courses;
};
```

Association (3/3)

```
// Creating Course objects.  
Course math{"Mathematics"};  
Course physics{"Physics"};  
Course chemistry{"Chemistry"};  
  
// Creating Student objects.  
Student alice{"Alice"};  
Student bob{"Bob"};  
  
// Associating students with courses.  
alice.enroll_course(&math);  
alice.enroll_course(&physics);  
bob.enroll_course(&chemistry);  
  
// Listing enrolled courses for each student.  
alice.list_enrolled_courses();  
bob.list_enrolled_courses();
```

Aggregation

Aggregation represents a relationship where one class (the whole) contains another class (the part), but the part can exist independently. It is represented by a "has-a" relationship.

```
class Department {  
    // Department implementation...  
};  
  
class University {  
private:  
    std::vector<Department> departments; // Aggregation: University has Departments.  
    // Other member variables...  
};
```

Composition

Composition is a stronger relationship where one class (the whole) contains another class (the part), and the part cannot exist independently. It is represented by a "*whole-part*" relationship.

```
class Room {  
    // Room implementation...  
};  
  
class Apartment {  
private:  
    Room living; // Composition: Apartment are composed by Room objects.  
    Room kitchen;  
    Room bedroom;  
    // Other member variables...  
};
```



Inheritance + polymorphism

Exercise session 03

Object oriented programming. Classes and access control in C++.

Advanced Programming - SISSA, UniTS, 2023-2024

Pasquale Claudio Africa

12 Oct 2023

Exercise 1: a class for data processing

1. Create a class named `DataProcessor` with private data members for a data array and its size. The data array should be represented as a `double *data`.
2. Implement a constructor that takes an array of floating-point numbers and its size as input and initializes the class data members.
3. Implement a copy constructor, a copy assignment operator and the destructor.
4. Add a method `n_elements()` that returns the number of elements in the array.
5. Test all these functionalities in the `main` function by creating proper instances of `DataProcessor` and displaying the results.

Exercise 2: computing statistics

1. Add methods to compute minimum and maximum values.
2. Add a method to compute the mean (average) of the data.
3. Add a method to compute the standard deviation of the data.
4. Add tests to validate these new functionalities.

Exercise 3: code organization

1. Organize the `DataProcessor` class by separating declarations and definition into separate header (`data_processor.hpp`) and source (`data_processor.cpp`) files.
2. Create a main program file that includes the header and demonstrates the use of the `DataProcessor` class for data analysis.
3. Compile the program using the following command:

```
g++ -Wall -Wpedantic -O3 data_processor.cpp main.cpp -o data_processor
```

Exercise 4: operator overloading and friend functions

1. Overload the output stream operator `<<` as a `friend` function to allow printing the list of values in the stored data, separated by a comma.
2. Overload the `[]` operator to allow indexing and accessing individual data elements. This operator will be used for both read and write access.
3. Overload the `+` operator in the `DataProcessor` class to allow adding two `DataProcessor` objects. The result should be a new `DataProcessor` object containing the element-wise sum of the data arrays. The operator should also print an error if the two operands do not have the same size.
4. Add tests to validate these new functionalities.

Exercise 5: `const`, `static`

1. Ensure the `const`-correctness of all member variables and methods by adding proper `const` qualifiers.
2. Add a `static` member function `get_n_instances()` that returns how many instances of `DataProcessor` objects are currently active.
3. Implement a free function

```
double compute_correlation(const DataProcessor &dp1, const DataProcessor &dp2);
```

that computes the Pearson correlation coefficient between two datasets with the same size.

4. Add tests to validate these new functionalities.

Lecture 04

Inheritance and polymorphism in C++.

Advanced Programming - SISSA, UniTS, 2023-2024

Pasquale Claudio Africa

17 Oct 2023

Outline

1. Class collaborations
2. Inheritance
3. Dynamic (runtime) polymorphism
4. Abstract classes

Class collaborations

Relationships between classes (1/2)

Classes can have various relationships, including:

- 1. Association:** A loose relationship where classes are related, but one does not necessarily contain the other. For example, a `Student` class may be associated with a `Course` or a `Teacher` class.
- 2. Dependency:** Dependency represents a situation where one class depends on another class. If class `A` uses or is dependent on class `B` (e.g., by invoking methods or using objects of class `B`), there is a dependency relationship between them. For example, a `car` class might depend on a, if you use a `FuelTank` class to store fuel.
- 3. Aggregation:** A "has-a" relationship where one class contains another as a part, but the contained object can exist independently. For example, a `University` class may aggregate `Department` classes.

Relationships between classes (2/2)

3. **Composition:** A stronger "*part-of*" relationship where one class contains another as a part, and the part cannot exist independently. For example, an `Apartment` class composes a `Room` class.
4. **Inheritance:** Inheritance represents an "*is-a*" relationship, where one class (the derived or subclass) inherits properties and behaviors from another class (the base or superclass). This is a fundamental form of collaboration in object-oriented programming.

Association (1/3)

Association is a relationship between two or more classes that defines how they are related to each other. It represents a general form of relationship between classes.

```
class Student; // Forward declaration.

class Course {
public:
    Course(const std::string& name) : course_name(name) {}

    const std::string& get_course_name() const {
        return course_name;
    }

private:
    std::string course_name;
};
```

Association (2/3)

```
class Student {  
public:  
    Student(const std::string& name) : student_name(name) {}  
  
    void enroll_course(Course *course) {  
        // Append new element to the vector.  
        enrolled_courses.push_back(course);  
    }  
  
    void list_enrolled_courses() const {  
        std::cout << student_name << " is enrolled in the following courses:" << std::endl;  
        for (const auto& course : enrolled_courses) {  
            std::cout << "- " << course->get_course_name() << std::endl;  
        }  
    }  
  
private:  
    std::string student_name;  
    std::vector<Course*> enrolled_courses;  
};
```

Association (3/3)

```
// Creating Course objects.  
Course math{"Mathematics"};  
Course physics{"Physics"};  
Course chemistry{"Chemistry"};  
  
// Creating Student objects.  
Student alice{"Alice"};  
Student bob{"Bob"};  
  
// Associating students with courses.  
alice.enroll_course(&math);  
alice.enroll_course(&physics);  
bob.enroll_course(&chemistry);  
  
// Listing enrolled courses for each student.  
alice.list_enrolled_courses();  
bob.list_enrolled_courses();
```

Dependency

Dependency: We say that class A depends on class B if A uses B. This is a very general term that implies **the declaration of class B being visible when you declare class A**. This visibility may be necessary because a method of class A takes an object of type B as a parameter:

```
class B; // Forward declaration.

class A {
public:
    void fun(const B& b);
    // ...
}
```

Aggregation

Aggregation represents a relationship where one class (the whole) contains another class (the part), but the part can exist independently. It is represented by a "has-a" relationship.

```
class Department {  
    // Department implementation...  
};  
  
class University {  
private:  
    std::vector<Department*> departments; // Aggregation: University has Departments.  
    // Other member variables...  
};
```

Composition

Composition is a stronger relationship where one class (the whole) contains another class (the part), and the part cannot exist independently. It is represented by a "*part-of*" relationship.

```
class Room {  
    // Room implementation...  
};  
  
class Apartment {  
private:  
    Room living; // Composition: Apartment are composed by Room objects.  
    Room kitchen;  
    Room bedroom;  
    // Other member variables...  
};
```

⚠ For improved resource management, composition can be implemented using pointers or references. See below for an example.

Views (proxies)

View (or **proxy**) is another type of aggregation that enables access to the members of the aggregating object using a different, often more specialized, interface. For example, you can access a general matrix as a diagonal matrix using a view:

```
class Matrix {  
public:  
    double & operator()(int i, int j);  
};  
  
class DiagonalView {  
public:  
    DiagonalView(Matrix &mat) : mat(mat) {}  
    double & operator()(int i, int j) {  
        return (i == j) ? mat(i, i) : 0.0;  
    }  
private:  
    Matrix &mat;  
}
```

Inheritance

Inheritance

Inheritance is a mechanism in object-oriented programming that allows a new class (the derived or subclass) to inherit properties and behaviors from an existing class (the base or superclass).

Inheritance establishes an “*is-a*” relationship between classes, where the derived class is a specialized form of the base class.

Example

You may have a base class `Shape` and derived classes like `Circle`, `Rectangle`, and `Triangle`. Each derived class “*is-a*” type of shape.

Inheritance in C++ (1/2)

```
class Shape { // Base class.  
public:  
    void f() { std::cout << "f (base class)." << std::endl; }  
  
    void draw() { std::cout << "Drawing a shape." << std::endl; }  
};  
  
class Circle : public Shape { // Derived class.  
public:  
    void g() { std::cout << "g (derived class)." << std::endl; }  
  
    void draw() { std::cout << "Drawing a circle." << std::endl; }  
};  
  
Circle circle; // Creating an object of the derived class.  
circle.f(); // Calls the f() method of the base class.  
circle.g(); // Calls the g() method of the derived class.  
circle.draw(); // Calls the draw() method of the derived class.
```

Inheritance in C++ (2/2)

In C++, inheritance is implemented using the `class` or `struct` keyword followed by a colon and the access specifier (`public` , `protected` , or `private`) followed by the base class name. For example:

```
class DerivedClass : access-specifier BaseClass {  
    // Derived class members and methods...  
};
```

Inheritance and access control (1/3)

```
class Base {  
public:  
    int public_data;  
  
protected:  
    int protected_data;  
  
private:  
    int private_data;  
};
```

Inheritance and access control (2/3)

```
class DerivedPublic : public Base {  
    // public_data remains public.  
    // protected_data remains protected.  
    // private_data is inaccessible.  
};
```

```
class DerivedProtected : protected Base {  
    // public_data becomes protected.  
    // protected_data remains protected.  
    // private_data is inaccessible.  
};
```

```
class DerivedPrivate : private Base { // 'private' is the default, if omitted.  
    // public_data becomes private.  
    // protected_data becomes private.  
    // private_data is inaccessible.  
};
```

Inheritance and access control (3/3)

- **Public** inheritance maintains the "is-a" relationship and allows the derived class to access and modify the public members of the base class.
- **Protected** inheritance restricts access to the base class's members in the derived class, making them protected.
- **Private** inheritance encapsulates the base class's members within the derived class, making them private and not accessible outside the derived class.

Construction of a derived class

When constructing an object of a derived class, the process follows a simple rule:

1. First, variables inherited from the base class are constructed using either the default constructor or the rule specified in the constructor of the derived class.
2. Any member variables added by the derived class are then constructed according to the usual rule.

As a result, the members of the base class are available for building members of the derived class.

```
class DerivedClass : public BaseClass {  
public:  
    DerivedClass(int derived_param, int base_param) : BaseClass(base_param) {  
        // Initialize derived members.  
    }  
};
```

Delegating constructor

In the constructor of a derived class, you can call the constructor of the base class, which is useful if you need to pass arguments. If no arguments are passed, the default constructor of the base class is used (in this case, the base class must be default constructible).

```
class B {  
public:  
    B(double x) : x(x) { /* ... */ }  
    // ...  
};  
  
class D : public B {  
public:  
    D(int i, double x) : B(x), my_i(i) { }  
private:  
    int my_i;  
};
```

In this example, an instance like `D d(4, 12.0)` sets `d.x` to `12.0` and `d.my_i` to `4`.

Inheriting constructors

It's important to note that constructors are not inherited by default, but they can be explicitly recalled with `using`.

```
class B {  
public:  
    B(double x) : x(x) { /* ... */ }  
    // ...  
};  
  
class D : public B {  
    using B::B; // Inherits B constructor.  
private:  
    int my_i = 10;  
};
```

In this example, when you create an instance like `D d(12.0)`, it calls the `B::B(double)` constructor, setting `d.x` to `12.0`, and `d.my_i` takes the default value of `10`.

Destruction of a derived class

The destruction of an object of the derived class involves the following steps:

1. First, the member variables defined by the derived class are destroyed in the reverse order of their declaration.
2. Then, the member variables of the base class are destroyed using the usual rule.

Destructors are called in reverse order.

```
DerivedClass::~DerivedClass() {  
    // Clean up derived resources.  
  
    // ~BaseClass() is automatically called here.  
}
```

Multiple inheritance

In C++, it is possible to derive from more than one base class. The derivation rules apply to each base class. Possible naming ambiguity issues can be resolved using the fully qualified name:

```
class D : public B, public C {  
public:  
    void fun() {  
        // If both B and C define f(), you can manually resolve the ambiguity.  
        const double x = B::foo();  
        // ...  
    }  
};
```

Multiple inheritance can lead to the *diamond problem*, where a class indirectly inherits from the same base class through multiple paths. C++ provides ways to mitigate these issues through `virtual` inheritance, ensuring that only one instance of a shared base class is created.

Dynamic (runtime) polymorphism

Polymorphism

Public inheritance is the mechanism through which we implement polymorphism, which allows objects belonging to different classes within a hierarchy to operate according to an appropriate type-specific behavior.

1. A pointer or reference to `D` is implicitly converted to a pointer (reference) to `B` (upcasting). A pointer or reference to `B` can be explicitly converted to a pointer (reference) to `D` using `static_cast` (statically) or `dynamic_cast` (dynamically).
2. Methods declared `virtual` in `B` are overridden by methods with the same signature in `D`.
3. If `B *b = new D` is a pointer to the base class converted from a `D*`, calling a `virtual` method will, in fact, invoke the method defined in `D` (this applies to references as well).

Note: Overridden virtual methods must have the same return type, with one exception: a method returning a pointer (reference) to a base class may be overridden by a method returning a pointer (reference) to a derived class.

"is-a" relationship

Polymorphism should be used only when the relationship between the base and derived class is an "is-a" relationship. In this context, the public interface of the derived class is a superset of that of the base class.

This means that one should be able to safely use any member from the public interface of the base class with an object of the derived class.

Therefore, the base class must define the public interface common to all members of the hierarchy.

Function overriding (1/4)

Function overriding is a key feature of polymorphism. It allows a derived class to provide its own implementation for a function that is already defined in the base class.

In C++, you override a base class function in a derived class by using the same function signature and the `virtual` keyword in the base class and the `override` keyword in the derived class.

⚠ This is not to be confused with function (or operator) overloading!

Dynamic binding is the mechanism that determines at runtime which method to call based on the actual type of the object.

Function overriding (2/4)

```
class Base {  
public:  
    virtual void display() {  
        std::cout << "Base class." << std::endl;  
    }  
};  
  
class Derived : public Base {  
public:  
    void display() override {  
        std::cout << "Derived class." << std::endl;  
    }  
};  
  
Base *ptr = new Derived();  
ptr->display(); // Calls the display() method of the Derived class.  
// Without 'virtual', Base::display() would be invoked, instead.
```

Function overriding (3/4)

```
class Polygon {  
public:  
    virtual double area() { /* ??? */ }  
};  
  
class Square : public Polygon {  
public:  
    double area() override { return side * side; }  
private:  
    double side;  
};  
  
void f(const Polygon &p) {  
    const double a = p.area();  
    // ...  
}  
  
Square s;  
f(s); // Legal! Polymorphism converts 'const Square &' to 'const Polygon &'.
```

Function overriding (3/4)

⚠ Polymorphism applies only when working with pointers or references.

Passing by copy leads to compilation errors:

```
void f(Polygon p) {  
    const double a = p.area();  
    // ...  
}  
  
Square s;  
f(s); // Illegal! A Square is not convertible into a Polygon.
```

A *factory* of polygons

```
unsigned int n_sides;

std::cout << "Number of sides: ";
std::cin > n_sides;

Polygon *p;
if (n_sides == 3)
    p = new Triangle{...};
else if (n_sides == 4)
    p = new Square{...};
else {
    // ...
}

std::cout << "Area: " << p->area() << std::endl;

delete s;
```

Virtual destructors

When applying polymorphism, the destructor of the base class should be defined as `virtual`. This is **compulsory** when the derived class introduces new member variables.

The reason for this necessity can be illustrated with the following code:

```
Polygon *p = new Square();  
// ...  
delete p;
```

In the last line, one should call the `Square` destructor. If you forget to mark the destructor in `Polygon` as `virtual`, that of `Polygon` is called instead. If `Square` has added new data members, this can lead to a memory leak.

Note: If you add the flag `-Wnon-virtual-dtor` at compilation time, the compiler issues a warning if you have forgotten a virtual destructor.

Is a virtual destructor in a derived class necessary?

It is not necessary to have a virtual destructor in the derived class if:

1. You are using inheritance but not polymorphism. This is the case when inheritance is only used to add additional functionalities, and you are not planning to address derived objects through pointers or references to the base. In this case, you have no virtual member functions (and no virtual destructors).
2. You have a hierarchy of classes where all data members are handled by the base class. The scope of the derived class is only to change the behavior of the public interface, with no need for new data members.

Protected and private polymorphism

Protected and private polymorphism uses the other types of inheritance: `protected` and `private`. Private inheritance is the default for classes (hence the need for the `public` keyword to indicate public inheritance), while for `structs`, the default is public.

- `class D: protected B`: Public and protected members of `B` become protected in `D`. Only methods and friends of `D` and classes derived from `D` can convert a `D*` into a `B*` (applies to references as well).
- `class D: private B`: Public and protected members of `B` become private in `D`. Only methods and friends of `D` can convert a `D*` into a `B*` (applies to references as well).

Why protected and private inheritance?

The use of protected and private inheritance is quite special. Typically, you use protected polymorphism when you want to use polymorphism but limit its availability to methods of the derived classes. The object is not polymorphic for the *general public* but only within the class hierarchy.

The use of private polymorphism is less common.

Remember that protected and private inheritance does not implement an "is-a" relationship.

Selective inheritance

In some cases, you may want only a part of the public interface of the base class to be exposed to the general public. You can achieve this through selective inheritance. Here's an example:

```
class Base {  
public:  
    double fun(int i);  
    // ...  
};  
  
class Derived : private Base {  
public:  
    using Base::fun; // fun() is made available.  
    // ...  
};
```

Abstract classes

Abstract classes

In some cases, the base class represents merely an abstract concept, and it does not make sense to create concrete objects of that type. In other words, the base class is meant to define the common public interface of the hierarchy but not to implement it fully.

For this purpose, C++ introduces the concept of an **abstract class**, which is a class where at least one virtual method is defined as **pure virtual**.

```
class Polygon {  
public:  
    virtual double area() = 0; // Pure virtual method.  
};  
  
Polygon p; // Illegal! Cannot instantiate an abstract class.
```

What is an abstract class?

- An abstract class is a class that cannot be instantiated. It serves as a blueprint for other classes and enforces a common interface for its derived classes.
- Abstract classes are defined by declaring at least one *pure virtual* function. These pure virtual functions have no implementation in the base class and are marked with the `virtual` keyword followed by `= 0`.
- Abstract classes can have regular member functions with implementations and data members, just like any other class.
- Derived classes that inherit from an abstract class **must** provide implementations for **all** of the pure virtual methods to become concrete (instantiable) classes.
- Pure virtual functions act as placeholders for functionality that must be provided by derived classes. They enforce a specific method signature that derived classes must adhere to.

A working example

```
class Triangle : public Polygon {  
public:  
    double area() override { return 0.5 * base * height; }  
private:  
    double base;  
    double height;  
};  
  
class Square : public Polygon {  
public:  
    double area() override { return side * side; }  
private:  
    double side;  
};  
  
Triangle t{1.5, 3.0}; // Legal.  
std::cout << t.area() << std::endl;  
  
Square s{0.5}; // Legal.  
std::cout << s.area() << std::endl;
```

The `final` and `override` specifier

Two specifiers in C++ help prevent errors: `final` and `override`.

- `final` for a method means that the method cannot be overridden.
- `final` for a class means that you cannot inherit from that class.
- `override` specifies that a method is overriding one from the base class.

The `override` keyword is not mandatory but strongly recommended, as it can trigger the compiler about possible errors.

Note: The option `-Wsuggest-override` can be used to make the compiler warn you if an `override` appears to be missing.

Examples of `final`

```
class A {  
public:  
    virtual void foo() final;  
    virtual double foo2(double);  
    // ...  
};  
  
class B final : A {  
public:  
    void foo(); // Error: foo cannot be overridden as it's final in A.  
    // ...  
};  
  
class C : B // Error: B is final.  
{  
    // ...  
};
```

Examples of override

```
class A {
    virtual void foo();
    void bar();
    // ...
};

class B : A {
    void foo() const override; // Error: Has a different signature from A::foo.

    void foo() override; // OK: Base class contains a virtual function with the same signature.

    void bar() override; // Error: B::bar doesn't override because A::bar is not virtual.
}
```

⚠ Although not mandatory, the `override` specification when overriding virtual member functions makes your code safer. It is **strongly recommended** to use it.

RTTI and typeid

Run-Time Type Information (RTTI) in C++ allows you to determine the actual type of an object at runtime. RTTI is typically implemented using the `typeid` operator or dynamic casting.

```
#include <typeinfo>

class Base {
public:
    virtual void print() { std::cout << "Base class." << std::endl; }
};

class Derived : public Base {
public:
    void print() override { std::cout << "Derived class." << std::endl; }
};

Base base; Derived derived;
std::cout << "Type of base: " << typeid(base).name() << std::endl;
std::cout << "Type of derived: " << typeid(derived).name() << std::endl;
```

Type checking with `dynamic_cast`

`dynamic_cast<D*>(B*)` tries to convert a `B*` to a `D*` (**downcasting**). If the condition fails, it returns the null pointer; otherwise, it returns the pointer to the derived class. This can be used to determine to which derived class a pointer to a base class refers.

It also works with references, but if the condition is not satisfied, it throws an exception.

```
double fun(const Polygon &p) {
    auto ptr = dynamic_cast<const Square *>(&p);
    if (ptr != nullptr) {
        // It is a square.
        ptr->get_side(); // This is not a member of the abstract Polygon class.
    } else {
        // It is not a square.
    }
}
```

Aggregation vs. composition with polymorphic objects

What happens if you want to aggregate your class with a polymorphic object? Should `Prism` be responsible for `poly_ptr`'s lifetime?

```
class Prism {  
public:  
    // 1) Take a pointer to an already existing object.  
    Prism(Polygon *p) : poly_ptr{p} {} // const vs. non-const?  
  
    // 2) Alternatively, MyClass handles both creation and destruction.  
    void init_as_square(const std::array<Point, 4> &vertices) {  
        poly_ptr = new Square{vertices};  
    }  
    ~Prism() { delete poly_ptr; }  
  
private:  
    Polygon *poly_ptr;  
    double height;  
}
```

Pointer or reference?

Some guidelines about aggregation/composition with pointers vs. references.

Reference

- Use a (const) reference when the aggregated object doesn't change, as is often the case in a "View."
- If you use a reference, the aggregated object must be passed through the constructor, making your class non-default-constructible.

Pointer

- Use (const) pointers if the aggregated object may change at runtime.
- If you use a pointer, **always initialize the pointer to `nullptr`** and create a method to test whether it has been assigned to an object. Initializing pointers to `nullptr` is a good practice.

Some advice

- The general design of code typically follows a top-down approach. You start from the final objective and identify the tasks required to achieve that objective.
- However, the actual programming process follows a bottom-up approach. Each basic task of your code or a set of closely related tasks should be encapsulated in a class, and you should **test these components separately**.
- After verifying the individual components, you can then compose them into a class or set of classes that implement your final objective. Whenever possible, aim to create components that can be reused and avoid code duplication.



Lambda functions, functors. Templates and generic programming

Exercise session 04

Inheritance and polymorphism in C++.

Advanced Programming - SISSA, UniTS, 2023-2024

Pasquale Claudio Africa

19 Oct 2023

Exercise 1: a framework for automatic differentiation

Implement a C++ framework for computing derivatives of arbitrary functions using a polymorphic approach. The goal is to create a structure that can handle common arithmetic operations, such as addition, subtraction, multiplication, division, and exponentiation, on values and their derivatives.

Test the program to compute the value and the derivative of the polynomial
 $f(x) = 2x^3 - 3x^2 + 4x - 5$ and of the function $g(x) = \frac{1}{x^2}$ at $x = 2$.

Exercise 1

1. Define an abstract base class `ADExpression` with two pure virtual functions:
 - `double evaluate()` : This function returns the value of the variable.
 - `double evaluate_derivative()` : This function returns the derivative of the variable.
2. Implement a concrete class `Scalar` that inherits from `ADExpression`. This class represents a scalar variable and its derivative.
3. Implement the following operation classes that also inherit from `ADExpression` :
 - `Sum` : Represents the addition of two `ADExpression` objects.
 - `Difference` : Represents the subtraction of two `ADExpression` objects.
 - `Product` : Represents the multiplication of two `ADExpression` objects.
 - `Division` : Represents the division of two `ADExpression` objects.
 - `Power` : Represents raising an `ADExpression` object to a constant exponent.
4. In the `main` function, demonstrate the usage of these classes.

Exercise 2: inheritance and polymorphism for data analysis

In the context of data analysis, create a C++ program that models different types of data sources and transformation objects.

1. Define an abstract class `DataSource` with attributes and methods that represent common properties of data sources. Create derived classes such as `FileDataSource` and `ConsoleDataSource`.
2. Define an abstract class `DataTransformer` with a virtual method for data transformation. Create derived classes such as `LinearScaler`, `LogTransformer`, and `StandardScaler` that implement specific data transformation methods.
3. Test all functionalities by prompting the user with proper questions.

Exercise 2 (1/5)

1. Define an abstract class `DataSource` with a string attribute `name`, a vector `data`, a method `display_info()` and a pure virtual method `read_data()`.
2. Implement a constructor and a virtual destructor in the `DataSource` class with.
3. Create derived classes `FileDataSource` and `VectorDataSource` that inherit from `DataSource`.
4. Implement constructors for all classes to model different data source types. For example, `FileDataSource` should initialize a `filename` and an input file, `ConsoleDataSource` should have a default constructor.
5. Implement destructors for the derived classes. For example, the `FileDataSource` constructor should open the file, and its destructor should close it.

Exercise 2 (2/5)

1. Implement the `read_data()` methods. `FileDataSource::read_data()` should import values from a file (see `data.txt` as an example), whereas `ConsoleDataSource::read_data()` should read a list of values from the standard input.
2. Create objects of these classes and demonstrate inheritance. For example, create a `FileDataSource` object and call `display_info()` and `read_data()` methods to read and display data from a file.
3. Ensure that resources associated with data sources are properly managed during construction and destruction.

Exercise 2 (3/5)

1. Define a base class `DataTransformer`, bound to `DataSource` by polymorphic composition. `DataTransformer` should have a pure virtual method `transform()` that transforms the `data` vector in the corresponding `DataSource`.
2. Create derived classes `LinearScaler`, `LogTransformer`, and `StandardScaler` that inherit from `DataTransformer`.
3. Override the `transform()` method in each derived class to provide specific data transformation. For example, `LinearScaler` scales the data by multiplying them by a given scaling factor, `LogTransformer` applies a logarithmic transformation and sets negative entries to `0`, and `StandardScaler` performs standardization to the $[0, 1]$ interval.
4. Create objects of these classes and demonstrate their use to transform the previously defined `DataSource` object.

Exercise 2 (4/5)

1. Use the `DataSource` and the `DataTransformer` hierarchy polymorphically. In particular, the program should prompt the user to import data either from a file or from console, and to select the transformation method.
2. Display the original and the transformed data.

Exercise 2 (5/5)

Possibilities for extensions

1. In case a `FileDataSource` is selected, prompt the user to specify the filename from the console.
2. In case a `LinearScaler` is selected, prompt the user to specify the scaling factor from the console.
3. Add a new class to the `DataSource` hierarchy to import a given field from an input `csv` file (see `data.csv` as an example).
4. Write a class or method to overwrite the original data from the text or `csv` file with the transformed ones.

Lecture 05

Functions. Templates and generic programming in C++.

Advanced Programming - SISSA, UniTS, 2023-2024

Pasquale Claudio Africa

24 Oct 2023

Outline

1. Advanced topics on functions

- Function pointers
- Functors (function objects)
- Lambda functions
- Function wrappers (`std::function`)

2. Generic programming and templates

- Generic programming
- Function templates
- Class templates
- Notes on code organization
- Advanced template techniques and concepts

Advanced topics on functions

Functions in C++

In C++, functions are categorized as free functions or member functions (methods) of a class. Subcategories exist for normal functions and template functions, including automatic return functions.

For a free, non-template function, the typical declaration syntax is as follows:

```
return_type function_name(...);  
  
auto function_name(...) -> return_type;  
  
auto function_name(...);
```

The first two forms are essentially equivalent, so choose the one that suits your preference. In the third scenario, the compiler deduces the return type, which will be explained in more detail later.

Function names and function identifiers

A function is **identified** by:

- Its **name**, `function_name` in the previous examples. More precisely, its **fully qualified name**, which includes the namespace, for instance `std::transform`.
- The number and type of its **parameters**.
- The presence of the `const` qualifier (for methods).
- The type of the enclosing class (for methods).

Two functions with different identifiers are eventually treated as **different functions**, which is the key for **function overloading**.

 **The return type is NOT part of the function identifier!**

Why use free functions

A function represents a mapping from input data (via its arguments) to an output provided through the returned value or, in some cases, through an argument if the corresponding parameter is a non-const reference.

As a result, a free function is typically **stateless**, meaning that two different calls to a function with the same input will yield the same output. The only exception to this rule is when a local variable is declared `static`.

Hence, you typically implement a function when your requirement is indeed a straightforward input/output mapping, akin to the standard mathematical definition of a function, $f : U \rightarrow V$.

What type does a function return

In rare instances, a function returns a reference to an object passed by reference. This is usually done when concatenation is desired, with the streaming operator being a typical example.

```
std::ostream &operator<<(std::ostream &os, const MyClass &obj)
{
    // ...
    return os; // Return the stream.
}
```

This enables concatenation like so:

```
std::cout << x << " concatenated with " << y;
```

Default parameters

In the **declaration** of a function, you can provide default values for the rightmost parameters.

```
std::vector<double> cross_prod(const std::vector<double> &a,  
                                const std::vector<double> &b,  
                                const unsigned int ndim = 2);
```

For instance:

```
a = cross_prod(c, d); // This sets ndim to 2.
```

A recall of function overloading

```
int fun(int i);
double fun(const double &z);
// double fun(double y); // Error: ambiguous!

auto x = fun(1);    // Calls fun(int).
auto y = fun(1.0); // Calls fun(const double &).
```

The function that gives the best match of the argument types is chosen. Beware of possible ambiguities and implicit conversions!

Callable objects

A callable object refers to an object that can be called as if it were a function, i.e., using the function call operator `operator()`. Callable objects include:

- **Functions** (free functions or member functions).
- **Function pointers**.
- **Member function pointers**: These allow you to call member functions of a class.
- **Functors (function objects)**: Instances of classes that overload `operator()`.
- **Lambda functions**: Introduced in C++11, they are useful for short, local functions.
- **Function wrappers**: `std::function` generalizes the concept of a function pointer.

Function pointers

Pointers to functions

```
double integrand(double x);

// Pointer to a function taking a double as an input and returning a double.
using f_ptr = double (*)(double);
// Or: typedef double (*f_ptr)(double);
double integrate(double a, double b, const f_ptr fun);

double I = integrate(0, 3.1415, integrand); // Passing function as a pointer.

f_ptr my_sin = std::sin; // Assigning a function pointer.

I = integrate(0, 3.1415, my_sin);
```

The name of the function is interpreted as a pointer to that function. However, you may precede it by `&`: `f_ptr f = &integrand`.

We will see in a while a safer and more general alternative to function pointers, the function wrapper `std::function` of the STL.

Dynamic function selection

You can use function pointers to select and call functions at runtime based on user input or other conditions.

```
int add(int a, int b) { return a + b; }
int subtract(int a, int b) { return a - b; }

int main() {
    int (*operation)(int, int); // 'operation' is a function pointer.

    if (user_input == "add") {
        operation = add;
    } else {
        operation = subtract;
    }

    const int result = operation(10, 5); // Calls either Add or Subtract based on user input.

    return 0;
}
```

Member function pointers

```
std::vector<Shape*> shapes;
shapes.push_back(new Circle(3.0));
shapes.push_back(new Rectangle(2.0, 4.0));
shapes.push_back(new Circle(2.5));

// Define a member function pointer for the area function.
double (Shape::*area_fun)() const = &Shape::area;

for (const auto shape : shapes) {
    const double area = (shape->*area_fun)();
    std::cout << "Area: " << area << std::endl;
}

// Cleanup allocated objects.
for (auto shape : shapes) {
    delete shape;
}
```

Functors (function objects)

Functors (function objects)

A **function object** or **functor** is a class object which overloads the **call operator** (`operator()`). It has semantics very similar to that of a function:

```
class Cube {  
public:  
    double operator()(const double &x) const { return x * x * x; }  
};  
  
Cube cube{}; // A function object.  
auto y = cube(3.4); // Calls Cube::operator()(3.4).  
auto z = Cube{}(8.0); // I create the functor on the fly.
```

If the call operator returns a `bool`, the function object is a **predicate**. If a call to the call operator does not change the data members of the object, you should declare `operator()` as `const` (as with any other method).

Why functors?

A characteristic of a functor is that it may have a state, so it can interact with other objects and store additional information to be used to calculate the result.

```
class Calculator {  
public:  
    int result = 0;  
  
    class Add {  
        public:  
            Calculator& calc;  
            Add(Calculator& c) : calc(c) {}  
            void operator()(int x, int y) { calc.result = x + y; }  
    };  
};  
  
calculator calc;  
calculator::Add add(calc);  
add(5, 3); // Result is stored in calc.result;
```

Predefined functors in the Standard Template Library

Under the header `<functional>`, you find a lot of predefined functors.

```
std::vector<int> in = {1, 2, 3, 4, 5};  
  
std::vector<int> out;  
  
std::transform(in.begin(), in.end(), // Source.  
              std::back_inserter(out), // Destination.  
              std::negate<int>());  
  
const double prod = std::accumulate(in.begin(), in.end(), 1.0, std::multiplies<int>());
```

Now `out = {-1, -2, -3, -4, -5}.`

`std::negate<type>` is a **unary functor** provided by the standard library.

`std::back_inserter<type>` inserts the transformed elements at the end (back) of vector `j`.

Some predefined functors in the STL

Functor	Description
<code>plus<T></code> , <code>minus<T></code>	Addition/Subtraction (Binary)
<code>multiplies<T></code> , <code>divides<T></code>	Multiplication/Division (Binary)
<code>modulus<T></code>	Modulus (Unary)
<code>negate<T></code>	Negative (Unary)
<code>equal_to<T></code> , <code>not_equal_to<T></code>	(Non-)Equality Comparison (Binary)
<code>greater</code> , <code>less</code> , <code>greater_equal</code> , <code>less_equal</code>	Comparison (Binary)
<code>logical_and<T></code> , <code>logical_or<T></code> , <code>logical_not<T></code>	Logical AND/OR/NOT (Binary)

For a full list, have a look at [this web page](#).

Lambda functions

Lambda expressions

We have a very powerful syntax to create short (and inlined) functions quickly: the lambda expressions (also called lambda functions or simply lambdas). They are similar to Matlab anonymous functions, like `f = @(x) x^2`.

```
auto f = [] (double x) { return 3 * x; }; // f is a lambda function.  
auto y = f(9.0); // y is equal to 27.0.
```

Note that I did not need to specify the return type in this case, the compiler deduces it as `decltype(3 * x)`, which returns `double`.

Capture specification

The capture specification allows you to use variables in the enclosing scope inside the lambda, either by value (a local copy is made) or by reference.

- [] : Captures nothing.
- [&] : Captures all variables by reference.
- [=] : Captures all variables by making a copy.
- [y] : Captures only y by making a copy.
- [&y] : Captures only y by reference.
- [=, &x] : Captures any referenced variable by making a copy, but capture variable x by reference.
- [this] : Captures the this pointer of the enclosing class object.
- [*this] : Captures a copy of the enclosing class object.

An example of use of [this]

With [this], we get the this pointer to the calling object:

```
class MyClass {  
public:  
    void compute() const {  
        auto prod = [this](double a) { x *= a; };  
        std::for_each(v.begin(), v.end(), prod);  
    }  
private:  
    double x = 1.0;  
    std::vector<double> v;  
};  
  
MyClass c;  
double res = c.compute();
```

Here, compute() uses the lambda prod that **changes** the member x. To be more explicit, you can write this->x *= a; .

Function wrappers

Function wrappers

And now the **catch all function wrapper**. The class `std::function<>` declared in `<functional>` provides polymorphic wrappers that generalize the notion of a function pointer. It allows you to use any **callable object** as **first-class objects**.

```
int add(int a, int b) {
    return a + b;
}

std::function<int(int, int)> func = add;

const int result = func(2, 3);
```

Function wrappers are **very useful** when you want to have a common interface to callable objects.

Function wrappers introduce a little overhead, since the callable object is stored internally as a pointer, but they are extremely flexible, and often the overhead is negligible.

Function wrappers and polymorphism

```
class Shape {  
public:  
    virtual double area() const = 0;  
};  
  
class Circle : public Shape {  
public:  
    Circle(double radius) : radius(radius) {}  
    double area() const override { return 3.14159265359 * radius * radius; }  
private:  
    double radius;  
};  
  
auto compute_area = [](const Shape& s) { return s.area(); };  
// 'auto' here resolves to std::function<double(const Shape&)>.  
  
Circle circle(5.0);  
std::cout << "Circle area: " << compute_area(circle) << std::endl;
```

A vector of functions

`std::function` can wrap any kind of **callable** object.

```
int func(int, int); // A function.

class F2 { // A functor.
public:
    int operator()(int, int) const;
};

// A vector of functions.
std::vector<std::function<int(int, int)>> tasks;
tasks.push_back(func); // Wraps a function.
tasks.push_back(F2{});
tasks.push_back([](int x, int y){ return x * y; }); // Wraps a lambda.

for (auto i : tasks)
    std::cout << i(3, 4) << endl;
```

It prints the result of `func(3, 4)`, `F2{}(3, 4)`, and `12` (3×4).

`std::bind` and function adapters

`std::bind` provides flexibility and reusability in code by decoupling function logic from its arguments and context, making it easier to work with functions as first-class objects.

```
int add(int a, int b) {
    return a + b;
}

// Create a new function based on 'add' where argument 1 is set equal to 5.
std::function<int(int)> add5 = std::bind(add, 5, std::placeholders::_1);
const int result = add5(3);
```

In modern C++, lambda functions often offer a more concise and readable alternative to `std::bind`, which still remains valuable for complex binding scenarios or when you need to reuse a set of bound arguments.

Generic programming and templates

Generic programming

What is generic programming?

- Generic programming is a programming paradigm that aims to write code in a way that's independent of data types.
- It focuses on creating reusable and versatile code by using templates or type abstractions.
- The goal is to develop algorithms and data structures that work with various data types.

Why use generic programming?

- **Reusability:** Write code once, use it with multiple data types.
- **Type safety:** Ensures that type-related errors are caught at compile-time.
- **Performance:** Optimized code for specific data types without code duplication.
- **Expressiveness:** Code that's concise and easier to read and maintain.

Examples of generic programming

- **STL (Standard Template Library)**: Offers a collection of generic data structures and algorithms.
- **Function templates**: Write functions that work with various data types.
- **Class templates**: Create versatile, type-safe data structures.

Use cases of generic programming

1. **Containers**: Generic data structures like vectors, stacks, and queues.
2. **Algorithms**: Generic sorting, searching, and transformation algorithms.
3. **Math operations**: Functions for arithmetic operations that work with multiple numeric types.
4. **Custom data structures**: Building generic trees, graphs, or linked lists.

Pros of generic programming

- **Code reusability:** Reduced need to write similar code for different data types.
- **Type safety:** Compile-time checks to ensure type correctness.
- **Efficiency:** Optimized code for specific data types.
- **Readability:** Cleaner and more expressive code.

Cons of generic programming

- **Complexity:** Generic code may be more complex due to abstraction.
- **Compile-time overhead:** Template instantiation can lead to longer compile times.
- **Debugging:** Template error messages can be challenging to decipher.

Function templates

Motivation

Consider the following example:

```
bool less_than(char x1, char x2) {  
    return (x1 < x2);  
}  
  
bool less_than(double x1, double x2) {  
    return (x1 < x2);  
}  
  
// ...
```

You could end up with multiple overloads of the same function: they all have the same implementation, but only differ by few details (such as argument types).

What are function templates?

- Function templates are a feature in C++ that allows you to define generic functions.
- They enable you to write a function once and use it with different data types.
- Function templates are defined using the `template` keyword, followed by type parameters enclosed in angle brackets.

```
template <typename T>
bool less_than(T x1, T x2) {
    return (x1 < x2);
}
```

Creating and using function templates

Function template declaration (and definition)

```
template <typename T>
T add(T a, T b) {
    return a + b;
}
```

Function template instantiation

```
const int result1 = add<int>(5, 3);
const int result2 = add(5, 3); // T automatically deduced as int.
```

```
const double result3 = add<double>(2.5, 3.7);
const double result4 = add(2.5, 3.7); // T automatically deduced as double.
```

Default template parameters

You can give defaults to the rightmost parameters.

```
template <typename T, typename U = double>
multiply_and_add(T a, U b, T c) {
    return a * b + c;
}

// Uses default type double for the second parameter.
const int result1 = multiply_and_add(5, 2.5, 3);

// Uses double for both 'T' and 'U'.
const double result2 = multiply_and_add(2.5, 3.7, 1.2);

// Uses float for the first parameter, int for the second.
const float result3 = multiply_and_add<float, int>(2.5, 3, 1.0);
```

Function template specialization

Template specialization allows you to define different behavior for specific template arguments.

```
template <typename T>
T max(T a, T b) {
    return (a > b) ? a : b;
}

// When 'T' is 'char', this version is invoked.
template <>
char max(char a, char b) {
    return std::toupper(a) > std::toupper(b) ? a : b; // max('A', 'b') is 'b'.
}
```

Example: vector sum using function template

The following function works for any type `T` for which `operator+` is defined.

For instance, this function can concatenate a vector of strings.

```
template <typename T>
T vector_sum(const std::vector<T>& vec) {
    // Initialize sum using T's default constructor
    // (e.g., 0 for numbers, empty for strings.

    T sum{};

    for (const T& elem : vec) {
        sum += elem;
    }

    return sum;
}
```

Important facts about templates

Templates serve as models for generating functions (or classes) once the template parameters are associated with actual types or values at the instance of the template.

This has two important implications:

1. Actual compilation occurs **only** when the template is instantiated (i.e., when it is actually used in your code). Only then can the compiler deduce the template arguments and have the necessary information to produce the machine code.
2. Thus, some compilation errors may only appear when the template is used!

Constant values as template parameters

You can give defaults to the rightmost parameters (this applies also).

A template parameter may also be an integral value.

```
template <int N, int M = 3>
constexpr int Multiply() {
    return N * M;
}

constexpr int result1 = Multiply<5>();    // Calculates 5 * 3 at compile-time.
constexpr int result2 = Multiply<2, 7>(); // Calculates 2 * 7 at compile-time.
```

Only integral types can be used (e.g., integers, enumerations, pointers, ...).

`constexpr` can be applied to variables, functions, and constructors, to ensure that they are evaluated at **compile time**.

Class templates

Introduction to class templates

- Class templates allow you to define generic classes.
- Syntax:

```
template <typename T>
class ClassName { /* ... */ };
```

- Type parameterization enables the class to work with different data types.

Advantages of class templates

- **Encapsulation** of data and behavior for a specific data type.
- **Code reusability**: Define a class structure once and use it with various types.
- **Type safety**: Prevents mixing incompatible types.

Creating and using class templates

Class template declaration (and definition)

```
template <typename T>
class List {
public:
    // ...
private:
    T value;
    List *next;
};
```

Class template instantiation

```
List<int> list_int;          // T is int.
List<double> list_double; // T is double.
// ...
```

Class template specialization

Template specialization allows you to define different behavior for specific template arguments.

```
template <typename T>
class Vector {
    // Implementation of a dynamic array for type T.
};

// Partial specialization for 'std::string'.
template <>
class Vector<std::string> {
    // Specialized behavior for strings.
};
```

Partial specialization

Partial specialization refines specialized behavior for specific subsets of template arguments.

```
// Generic template
template <typename T, int N>
class Array {
private:
    T data[N];
};

// Partial specialization for arrays of size 1.
template <typename T>
class Array<T, 1> {
private:
    T element; // No need to store an array for a single variable!
};

Array<int, 3> arr1; // Uses the generic template for arrays of size 3
Array<char, 1> arr2; // Uses the partially specialized template for arrays of size 1
```

Template alias

Template aliases are a versatile feature that simplifies code by allowing you to create more concise and expressive names for complex template types.

```
template <typename T, int N>
class Array {
private:
    T data[N];
};

// Template alias to create an array of integers.
template <int N>
using IntArray = Array<int, N>;

IntArray<5> arr; // Creates an array of integers with 5 elements.
```

Example: templated stack class (LIFO)

```
template <typename T>
class Stack {
public:
    void push(const T& value) { elements.push_back(value); }

    T pop() {
        if (elements.empty()) {
            std::cerr << "Stack is empty" << std::endl;
            std::exit(1);
        }

        T top = elements.back();
        elements.pop_back();
        return top;
    }

private:
    std::vector<T> elements;
};
```

Best practices

- Use descriptive type parameter names.
- Avoid unnecessary template code duplication.
- Templatize functions for mathematical operations.

For class templates

- Use class templates for generic data structures.
- Define meaningful class names and method names.
- Leverage template specialization for customized behavior.

Notes on code organization

Template instantiation and linkage

- The compiler produces the code corresponding to function templates and class template members that are instantiated in each translation unit.
- It means that all translation units that contain, for instance, an instruction of the type `std::vector<double> a;` produce the machine code corresponding to the default constructor of a `std::vector<double>`. If we then have `a.push_back(5.0)`, the code for the `push_back` method is produced, and so on.
- If the same is true in other compilation units, the **same machine code** is produced several times. It is the **linker** that eventually produces the executable by selecting only one instance.

⚠ There is an (almost inevitable) waste of compilation time.

The problem

- **Template definitions need to be available at the point of instantiation.** When a template is used with specific type arguments, the compiler needs to see the template definition to generate the code for that particular instantiation. Placing the template definition in a source file would make it unavailable for instantiation in other source files.
- If you place template definitions in source files and use the template in multiple source files, you may encounter linker errors due to multiple definitions of the same template. **Placing the template definitions in a header file** ensures that the definition is available for all source files that include it, and the linker can consolidate the definitions as needed.

Possible file organizations with templates

1. Leave everything in a **header file**. However, if the functions/methods are long, it may be worthwhile, for the sake of clarity, to separate definitions from declarations. You can put declarations at the beginning of the file and only short definitions. Then, at the end of the file, add the long definitions for readability.
2. **Separate** declarations (`module.hpp`) and definitions (`module.tpl.hpp`) when templates are long and complex. Then add `#include "module.tpl.hpp"` at the end of `module.hpp`.
3. **Explicitly instantiation** for a specific list of types. Only in this case, definitions can go to a source file. But if you instantiate a template for other types not explicitly instantiated, the compiler will not have access to the definition, leading to linker errors.

Explicit instantiation

We can tell the compiler to produce the code corresponding to a template function or class using **explicit instantiation**. If a source file contains, for instance:

```
template double func(const double &);  
template class MyClass<double>;  
template class MyClass<int>;
```

then the corresponding object file will contain the code corresponding to the template function
`double func<T>(const T &)` with `T=double` and that of **all methods** of the template class
`MyClass<T>` with `T=double` and `T=int`.

This can be useful to save compile time when debugging template classes (since the code for all class methods is generated).

Advanced template techniques and concepts

Type deduction and the `auto` keyword

- Type deduction allows the compiler to determine the data type of variables and return values automatically.
- The `auto` keyword simplifies code and improves readability.

```
auto sum1 = add(5, 3);      // int
auto sum2 = add(2.5, 3.7);  // double
auto sum3 = add(1.0f, 2.0f); // float
```

Type deduction with `auto` is particularly useful when you want to write more generic code that adapts to different data types without explicitly specifying them.

 **Don't overuse `auto`!**

The use of `this` in templates (1/2)

In a class template, derived names are resolved only when a template class is instantiated (only then the compiler knows the actual template argument). Other names are resolved immediately.

```
void my_fun() { ... }

template <typename T> class Base {
public:
    void my_fun(); // Not a good idea!
};

template <typename T>
class Derived : Base<T> {
public:
    void foo() { my_fun(); ... } // Which myfun()???
};
```

In this case, the free function `my_fun()` is used.

The use of `this` in templates (2/2)

Solution: use `this`:

```
template <typename T>
class Derived : Base<T> {
public:
    void foo() { this->my_fun(); ... }
}
```

or the fully-qualified name:

```
template <typename T>
class Derived : Base<T> {
public:
    void foo() { Base<T>::my_fun(); ... }
}
```

In this case, the compiler understands that `my_fun()` depends on the template parameter `T` and will resolve it only at the instance of the template class.

Template template parameters

- Template template parameters allow you to define templates as template arguments.
- Used for defining higher-order templates that accept template classes.

```
template <typename T, template <typename> class C = std::complex>
class MyClass {
private:
    C<T> a;
};

MyClass<double, std::vector> x; // 'a' is a std::vector<double>.

MyClass<float> x; // 'a' is a std::complex<float>.
```

This feature allows to write expressions like `std::vector<std::complex<double>>`.

Template metaprogramming

- Template metaprogramming is a way to perform computations at compile time.
- It involves using template features to generate code during compilation.
- Common use cases: constant expressions, type traits, and generic algorithms.

SFINAE (Substitution Failure Is Not An Error)

- SFINAE is a C++ rule that allows you to enable or disable function templates based on the validity of their substitutions.
- Used for type traits and selective function overloading.

Example: Fibonacci with template metaprogramming

```
template <int N>
class Fibonacci {
public:
    static constexpr int value = Fibonacci<N - 1>::value + Fibonacci<N - 2>::value;
};

template <>
class Fibonacci<0> {
public:
    static constexpr int value = 0;
};

template <>
class Fibonacci<1> {
public:
    static constexpr int value = 1;
};

constexpr int n = Fibonacci<10>::value; // calculated at compile-time.
```

Example: type traits with SFINAE

```
template <typename T>
class has_print {
public:
    template <typename U>
    static std::true_type test decltype(U::print)*);

    template <typename U>
    static std::false_type test(...);

    static constexpr bool value = decltype(test<T>())::value;
};

class MyType {
public:
    void print() {}
};

std::cout << std::boolalpha;
std::cout << has_print<MyType>::value << std::endl; // true
std::cout << has_print<int>::value << std::endl; // false
```

Variadic templates

- Variadic templates allow functions and classes to accept a variable number of arguments. The `...` syntax is used to define them.
- Useful for handling functions with multiple arguments of varying types.

Example: recursive sum function

```
template <typename T>
T sum(T value) {
    return value;
}

template <typename T, typename... Args>
T sum(T first, Args... rest) {
    // Consume the first argument, then recurse over remaining arguments.
    return first + sum(rest...);
}
```

CRTP (Curiously Recurring Template Pattern)

- CRTP is a design pattern where a derived class inherits from a base class template with itself as the template argument.
- Used to achieve static polymorphism and code reuse.

Traits classes and policy-based design

- Traits classes are used to encapsulate properties and behaviors of types.
- Policy-based design involves creating classes or functions with interchangeable policies to customize behavior.

Example: CRTP for static polymorphism

```
template <typename Derived>
class Shape {
public:
    double area() {
        return static_cast<Derived*>(this)->area();
    }
};

class Circle : public Shape<Circle> {
public:
    double area() {
        // Compute area of a circle.
    }
};

Circle c; c.area();
```

CRTP allows the `Shape` class to know the interface of its derived class at compile time, enabling **static (compile-time) polymorphism** and avoiding runtime overhead.

Example: traits for type information

```
#include <type_traits>

template <typename T>
void process_type(T value) {
    if constexpr (std::is_integral_v<T>) {
        // Process integral types.
    } else if constexpr (std::is_floating_point_v<T>) {
        // Process floating-point types.
    } else {
        // Default behavior.
    }
}
```

⚠️ **if constexpr** available since C++17. It evaluates conditions at compile-time.



The Standard Template Library. New features of C++14/17/20.

Exercise session 05

Functions. Templates and generic programming in C++.

Advanced Programming - SISSA, UniTS, 2023-2024

Pasquale Claudio Africa

27 Oct 2023

Exercise 1: lambda functions

Starting from the `hints/ex1.cpp` source file, create a C++ program that calculates the total cost of a given a list of products.

1. Use `std::accumulate` to calculate the total cost of the products, passing a custom lambda function.
2. Display the results after each step to check for correctness.

Exercise 2: function pointers, functors, and lambda functions

Starting from the `hints/ex2.cpp` source file, develop a library management system with search capabilities using lambdas and functors.

1. Using `std::sort`, sort the books:

- In ascending order based on year, using a **function pointer** as a comparator.
- In descending order based on year, using a **lambda function** as a comparator.
- In ascending order based on the author name, using a **functor** as a comparator.

2. Using `std::copy_if`, fill the `filtered_books` variable by extracting from `books` only the books written by a specific author. Implement the search functional using lambdas.

3. Display the results after each step to check for correctness.

Exercise 3: function wrappers, templates (1/2)

The `hints/ex3/` folder provides a partial C++ implementation of the Newton method to approximate the root(s) of a function f , i.e., to solve the problem $f(x) = 0$.

Newton's method in a nutshell

Starting with an initial guess for the root(s) of the function, denoted as $x^{(0)}$, repeatedly refine the estimate using the formula

$$x^{(k+1)} = x^{(k)} - \frac{f(x^{(k)})}{f'(x^{(k)})},$$

where $f'(x)$ is the derivative of $f(x)$.

The iterations continue until the difference between two consecutive estimates, $\|x^{(k+1)} - x^{(k)}\|$ is smaller than a predefined tolerance. If the condition is not met within a maximum number of iterations, the algorithm failed to reach converge and the solver returns `Nan`.

Exercise 3: function wrappers, templates (2/2)

1. Fill in the missing parts to make the program work with real-valued scalar functions.
2. Use the program to solve $f(x) = x^2 - 1 = 0$, starting from $x^{(0)} = 0.5$.
3. Templatize the solver to be able to deal with more general functions, such as complex-valued functions.
Use the program to solve $f(x) = x^2 + 1 = 0$, starting from $x^{(0)} = 0.5 + 0.5i$.
4. How would organize the project files? Is it better to keep everything in header files, or splitting declarations and definitions in header and source files by providing explicit instantiations? Try both alternatives.

A note on the use of template arguments as policies

⚠ In C++, template arguments can be used as *policies*.

This allows you to customize the behavior of a class or function without changing its core implementation. As templates provide a way to write generic code that can work with different data types, policies provide a way to write generic code that can behave according to different algorithms or strategies.

When you use a template argument as a policy, you are essentially saying, "Here is a piece of code, and I want to let users decide certain aspects of its behavior." This can include things like how elements are compared, how data is stored, which specific algorithm to apply, or how certain operations are performed.

See the example included in the `examples` folder.

Exercise 4: template metaprogramming

Use template metaprogramming to calculate the factorial of an integer at compile time.

1. Define a template class for calculating the factorial of an integer.
2. Instantiate the template for the integers 5, 7, and 10.
3. Use `static_assert` to ensure that values are computed at compile time rather than runtime.
4. Print the results of the factorials.

Lecture 06

The Standard Template Library.

Advanced Programming - SISSA, UniTS, 2023-2024

Pasquale Claudio Africa

31 Oct 2023

Outline

1. Containers

- Sequence containers
- Container adaptors
- Associative containers
- Special containers

2. Iterators

3. Algorithms

4. Evolution of the STL

An overview of the Standard Template Library (STL) in C++

History

In November 1993, Alexander Stepanov introduced the Standard Template Library (STL) to the ANSI/ISO committee for C++ standardization. This library was founded on the principles of generic programming and featured generic containers, iterators, and a comprehensive set of algorithms designed to operate on them.

The proposal was not only accepted but also paved the way for what is now known as the Standard Library. The Standard Library has evolved into a vast collection of tools that play an indispensable role in modern C++ development.

Notably, all functionalities provided by the Standard Library are encapsulated under the `std::` namespace or, in some cases, within sub-namespaces also contained within `std::`.

The Boost libraries

Many components added to the Standard Library over the years were originally proposed in the [Boost C++ libraries](#), a collection of libraries designed to complement the Standard Library and cater to a wide range of applications.

Notably, one outstanding component within the Boost libraries is the [Boost Graph Library](#), widely used by professionals working with graphs and networks.

All Boost libraries are open-source and can be installed individually or as a whole on various Linux platforms using package managers. For instance, on Ubuntu, you can use the command `sudo apt-get install libboost-all-dev`. Alternatively, you can download the source code and compile them manually.

An overview of the STL (1/2)

- **Porting of C libraries:** Several C libraries have been adapted to the `std` namespace. As an example, `<math.h>` becomes `<cmath>`, and they all begin with a 'c'.
- **Containers:** Generic containers and iterators.
- **Utilities:** Smart pointers, fixed-size collections of heterogeneous values like `pair` and `tuple`, clocks and timers, function wrappers, and predefined functors. Also, the `ratio` class for constant rationals.
- **Algorithms:** These operate on ranges of values, usually stored in standard containers, to perform specific actions like sorting, transformations, and copying. Some of them even support parallel execution.
- **Strings and text processing:** The `string` class and its derived classes, regular expressions, and efficient string manipulation tools.
- **Support for I/O:** I/O streams and related utilities.
- **Utilities for diagnostics:** Standard exception classes and exception handlers.

An overview of the STL (2/2)

- **Numerics:** `complex<T>`, numeric limits, random numbers and distributions, and basic mathematical operators.
- **Support for generic programming:** Type traits, `declval`, and `as_const`.
- **Support for reference and move semantics:** Reference wrappers, `move()`, and `forward<T>`.
- **Support for multithreading and concurrency:** Threads, mutexes, locks, and parallel algorithms.
- **Support for internationalization:** `locale` and `wide_char`.
- **Filesystem:** Tools for examining the file system.
- **Allocators:** Utilities that allow you to change how objects are allocated within containers.
- **Utilities for hybrid data:** `optional`, `variant`, and `any`.

A milestone: C++11

- **Standardization process:** C++11 marked the successful completion of a rigorous standardization process.
- **Modern features:** C++11 introduced modern language features like lambda expressions and range-based for loops.
- **Enhanced Standard Library:** New containers, algorithms, and utility classes.
- **Improved memory management:** Smart pointers were introduced in C++11.
- **Initializer lists:** C++11 introduced initializer lists, simplifying data structure initialization.
- **Simpler and safer code:** The addition of lambda expressions improved code readability and maintainability.
- **Standardized threads:** It brought a standardized threading library, enabling concurrent and parallel programming.
- **Improved performance:** Move semantics and rvalue references boosted resource management and program performance.

Containers

Sequence containers

Containers can be categorized based on how data is stored and handled internally. The categories include:

- **Sequence containers:** `std::vector<T>` , `std::array<T, N>` , `std::deque<T>` , `std::list<T>` , `std::forward_list<T>` .
 - Ordered collections of elements with their position independent of the element value.
 - In `std::vector` and `std::array` , elements are guaranteed to be contiguous in memory and can be accessed directly using the `[]` operator.
- **Adaptors:** These are built on top of other containers and provide special operations:
 - `std::stack<T>` , `std::queue<T>` , and `std::priority_queue<T>` .

Example: std::vector

```
std::vector<int> v{2, 4, 5}; // 2, 4, 5.  
v.push_back(6);           // 2, 4, 5, 6.  
v.pop_back();             // 2, 4, 5.  
v[1] = 3;                 // 3, 4, 5.  
std::cout << v[2];        // 5  
for (int x : v)  
    std::cout << x << ' '; // Prints: 2 3 5  
std::cout << std::endl;  
  
v.reserve(8);  
v.resize(5, 0);  
std::cout << v.capacity() << std::endl;  
std::cout << v.size() << std::endl;
```

Example: std::array

```
std::array<int, 6> a{4,8,15,16,23,42};  
std::cout << a.size() << std::endl;      // 6  
std::cout << a[0] << std::endl;          // 4  
std::cout << a[3] << std::endl;          // 16  
std::cout << a.front() << std::endl;     // 4  
std::cout << a.back() << std::endl;       // 42  
  
std::array<int, 3> b{7,8,9};  
// a = b; // Compiler error: types don't match!
```

Source

Associative containers (1/3)

- **Associative containers:** These collections have elements whose position depends on their content. They are divided into:
 - **Maps:** Elements are key-value pairs.
 - **Sets:** Elements are just values (in sets, keys and values are considered the same).
 - Furthermore, they can be divided into **ordered** and **unordered**, depending on how the elements are stored, imposing different requirements on element types.

⚠ In a set, the terms "value" and "key" are used interchangeably since they are equivalent.

Associative containers (2/3)

- Ordered associative containers:
 - `std::set<K>` (no repetition) and `std::multiset<K>` (repetition allowed): They store single values, and the value is the key.
 - `std::map<K, V>` (no repetition of keys) and `std::multimap<K, V>` (repetition of keys allowed): They store pairs of (key, value) and act as **dictionaries**.
- An ordering relation must be defined for the key `K`. It can be done using a specific callable object, a specialization of the functor `std::less<K>`, or by defining `operator<()`.
- Keys can be accessed read-only; modifications of keys require special operations.

Associative containers (3/3)

- **Unordered associative containers:**
 - `std::unordered_set<K>` and `std::unordered_multiset<K>`.
 - `std::unordered_map<K, V>` and `std::unordered_multimap<K, V>`.
- Their general behavior is similar to that of the ordered counterparts.
- A hashing function, mapping keys to positive integers in a range [0, max), should be provided along with a proper equivalence relation among keys.
- For standard types, a default hash function is provided by the library, as well as relational operators.

Source

Example: std::map

```
std::map<std::string, int> age; // Creating a std::map with string keys (names) and integer values (ages).  
  
// Inserting key-value pairs into the map. Elements are automatically sorted by key.  
age["Alice"] = 25;  
age["Charlie"] = 22;  
age["Charlie"] = 23; // Overwrite the previous value.  
age["Bob"] = 30;  
  
// Accessing elements by key.  
const std::string name = "Charlie";  
if (age.find(name) != age.end()) {  
    std::cout << name << " is " << age[name] << " years old." << std::endl;  
} else {  
    std::cout << "Information about " << name << " not found." << std::endl;  
}  
  
const int age_david = age.at("David"); // Throw an exception if "David" is not present.  
const int age_david2 = age["David"]; // WARNING: this will allocate "David" if not present!  
  
// Iterating through the map.  
std::cout << "Name - Age map:" << std::endl;  
for (const auto& entry : age) {  
    std::cout << entry.first << " is " << entry.second << " years old." << std::endl;  
}
```

Example: std::set

```
std::set<int> numbers; // Creating a std::set of integers.

// Inserting elements into the set. Elements are automatically sorted.
numbers.insert(10);
numbers.insert(30);
numbers.insert(20);
numbers.insert(10); // Duplicate, won't be added.
numbers.insert(20); // Duplicate, won't be added.

// Checking if an element is in the set.
const int search_value = 20;
if (numbers.find(search_value) != numbers.end()) { // Or, since C++20: if (numbers.contains(search_value))
    std::cout << search_value << " is in the set." << std::endl;
} else {
    std::cout << search_value << " is not in the set." << std::endl;
}

// Iterating through the set.
for (const int& num : numbers) {
    std::cout << num << " ";
}
std::cout << std::endl;
```

Special containers: `std::byte`

`std::byte` is a relatively low-level data type introduced in C++17, and its primary use is to represent individual bytes in memory, often used for bitwise operations and when dealing with raw memory. `std::byte` can be used for encoding and decoding data:

Example

```
std::byte flags = std::byte(0b11001010);
std::byte mask = std::byte(0b11110000);
std::byte result = flags & mask; // Bitwise AND operation
```

Special containers: `std::pair`

`std::pair` represents a pair of values. It's commonly used to combine two values into a single entity.

Example

```
std::pair<double, double> min_max(const std::vector<double> &vec) {
    // Compute min_val and max_val.
    return std::make_pair(min_val, max_val);
}

std::vector<double> data;
// ...
const std::pair<double, double> result = min_max(data);

std::cout << "Minimum value: " << result.first << std::endl;
std::cout << "Maximum value: " << result.second << std::endl;
```

Special containers: `std::tuple`

`std::tuple` is a generalization of `std::pair` representing a heterogeneous collection of values. It can hold elements of different types.

Example

```
std::tuple<std::string, int, std::string> get_person_info() {
    return std::make_tuple("Alice", 28, "Engineer");
}

std::tuple<std::string, int, std::string> person = get_person_info();

// Access and display the individual elements of the tuple.
const std::string name = std::get<0>(person);
const int age = std::get<1>(person);
const std::string occupation = std::get<2>(person);
```

Special containers: `std::variant`

`std::variant` represents a type-safe union of types, allowing you to hold one value from a set of specified types.

Example

```
std::variant<double, std::string> var;

var = "Hello"; // Hold a string.
var = 10.5;    // Hold a double.

const double c = std::get<double>(var); // c is now 10.5.

std::string s = std::get<std::string>(var); // Runtime error: not currently holding a string!!

// But I can check.
if (var.holds_alternative<std::string>()) {
    // It's a string.
}
```

Special containers: `std::optional`

`std::optional` is a special wrapper introduced in C++17 for a type that behaves partially similarly to a pointer but is convertible to `bool`, with `false` indicating that the value is missing or unset. It also contains other methods to interrogate its content.

Example

```
// A vector of optionals storing a double.  
std::vector<std::optional<double>> data(100); // All elements are unset.  
data[10] = 45.27; // You set the optional just by assigning the value.  
auto d = data[7]; // This is unset: you can interrogate it.  
  
if (d.has_value()) // Or: if (d)  
    std::cout << d.value() << std::endl;  
else  
    std::cout << "Value unset";  
  
const double value_or_default = data[20].value_or(1.5);
```

Special containers: `std::any`

`std::any` is a class introduced in C++17 that provides a dynamic, type-safe container for holding values of any type. It allows you to store and retrieve objects of different types in a type-safe manner.

```
std::any data;

data = 42; // Store an integer.

if (data.type() == typeid(int)) {
    const int value = std::any_cast<int>(data);
}

data = std::string("Hello, world!"); // Store a string.

if (data.type() == typeid(std::string)) {
    const std::string value = std::any_cast<std::string>(data);
}
```

Source

Iterators

Iterators

Iterators are a generalization of **pointers** that allow a C++ program to work with different data structures (for example, **containers** and ranges (since C++20)) in a uniform manner. The iterator library provides definitions for iterators, as well as iterator traits, adaptors, and utility functions.

Since iterators are an abstraction of pointers, their semantics are a generalization of most of the semantics of pointers in C++. This ensures that every function template that takes iterators works as well with regular pointers.

Basic functionality

An iterator is **any object** that allows iterating over a succession of elements, typically stored in a standard container. It can be **dereferenced** with the `*` operator, returning an element of the range, and incremented (moving to the next element) with the `++` operator.

Source

⚠️ C++20 has redefined the categories with `ranges`. Old ones are now referred to as Legacy.

Source

Containers iterators

All main containers have iterators that belong to the **Forward** category. `std::array` and `std::vector` have **Random access** iterators.

All containers have the methods `begin()` and `end()` (`cbegin()` and `cend()`) that return the (`const`) iterator to the first and the *last + 1* element in the container. You may also use the corresponding free functions `std::begin()` and `std::end()`, which can be overloaded for any type.

All containers define the types `Container::iterator`, `Container::reverse_iterator`, and the corresponding `const` versions (`Container::const_iterator`, etc.).

! In a `const` iterator, it is the pointed element that is `const`, not the iterator itself! More precisely, it is an *iterator to const*.

! `auto` simplifies the use of iterators!

Methods and types in containers (1/2)

- Default, copy, and move constructors
- `Container c(beg, end)` : Constructor from the range `[beg, end]`
- `size()` : Number of stored elements
- `empty()` : `true` if empty
- `max_size()` : Max number of elements that can be stored
- Comparison operators
- `c1 = c2` : Copy assignment, `c1` may be a container of a different type from `c2`
- `c1.swap(c2)` : Swaps data (`c2` may be a container of different type)
- `std::swap(c1, c2)` : As above (as a free function)

Methods and types in containers (2/2)

- `begin()` : Iterator to the first element
- `end()` : Iterator to the position after the last element
- `rbegin()` : Reverse iterator for reverse iteration (initial position)
- `rend()` : Reverse iterator (position after the last element)
- `cbegin()`, `cend()`, `crbegin()`, `crend()` : Same as above, but iterating over `const` elements
- `insert(pos, elem)` : Inserts a copy of `elem` (return value may differ)
- `emplace(pos, args...)` : Inserts an element by constructing it in place
- `erase(beg, end)` : Removes all elements in the range `[beg, end)`
- `clear()` : Removes all elements (makes the container empty)

Types defined by containers

- `C::value_type` : The type of the object stored in a container. `value_type` must be assignable and copy constructible, but need not be default constructible.
- `C::iterator` : The type of the iterator used to iterate through a container's elements.
- `C::const_iterator` : A type of iterator that may be used to examine but not modify a container's elements.
- `C::reference` : A type that behaves as a reference to the container's value type.
- `C::const_reference` : A type that behaves as a const reference to the container's value type.
- `C::pointer` : A type that behaves as a pointer to the container's value type.
- `C::difference_type` : A signed integral type used to represent the distance between two of the container's iterators.
- `C::size_type` : An unsigned integral type that can represent any nonnegative value of the container's distance type.

Why types in a container?

Having the type of the contained elements defined in the container may seem peculiar. After all, the type of elements in a `vector<T>` is just `T`! However, this technique is useful in generic programming:

```
template <typename Container>
void my_fun(Container &c) {
    using ValueType = typename Container::value_type;
    // ...
    valueType a;
}
```

The `auto` specifier and `decltype()` reduce this need. For instance, you could have written:

```
using ValueType = decltype(*c.begin());
```

But being explicit is often better! Having traits to specify type members gives a lot of flexibility (and indeed the standard library uses traits...).

Distance between iterators

The distance between iterators is equal to the number of elements in the range defined by them.

```
{  
    const std::set<int> my_set = {10, 20, 30, 40, 50};  
  
    auto first = my_set.lower_bound(20); // Iterator to the first element >= 20.  
    auto second = my_set.lower_bound(40); // Iterator to the first element >= 40.  
    const int distance = std::distance(first, second); // Calculate the distance.  
}  
  
{  
    const std::vector<int> my_vector = {1, 2, 3, 4, 5};  
  
    auto first = my_vector.begin();  
    auto second = std::find(my_vector.begin(), my_vector.end(), 4); // Return iterator to element 4.  
    const int distance = std::distance(first, second); // Calculate the distance.  
}
```

Distance may be negative if iterators are **random access**.

size_type and **std::size_t**

`Container::size_type` in a sequence container is the type used as an argument in `operator[]`, defined for these containers.

It is guaranteed to be an unsigned integral type. Use it instead of `int` or `unsigned int` if you anticipate problems with implicit conversions. `size_type` is implementation-dependent (it may vary between 32-bit and 64-bit architectures).

By default, it is set equal to `std::size_t`, defined in `<cstddef>`, which is the type used to address ordinary arrays.

If you want to be safe, use `std::size_t` or `Container::size_type` to address sequential containers.

```
for (std::size_t i = 0; i < a.size(); ++i)
    a[i] = ...;
```

Algorithms

Ranges (sequences)

The term **range** (or **sequence**) refers to a pair of iterators that define an interval of elements that are "logically adjacent" within a container.

We provide a working definition. Two iterators `b` and `e` define a valid range $[b, e)$ if the instruction:

```
for (iterator p = b; p != e; ++p) {  
    *p;  
}
```

is valid, and `*p` returns the value of an element of the container.

The algorithms of the standard library typically operate on sequences.

Algorithms

The STL provides an extensive set of algorithms to operate on containers, or more precisely on **ranges**.

For a full list, you may look [here](#) for generic algorithms and [here](#) for numeric algorithms.

⚠ C++20 has revised the concept of range and provides a new set of algorithms in the namespace `std::ranges`, with the same name as the old ones, but simpler to use and sometimes more powerful.

Why using a standard algorithm?

Many standard algorithms can be implemented using a for loop. So what's the advantage? I start by saying that there is nothing wrong with the for-loop version. If you are happy with it, use it. Yet with standard algorithms:

- You are more uniform with respect to different container types.
- The algorithm of the standard library may do certain optimizations if the contained elements have some characteristics.
- You have a parallel version for free (see next slides).

Types of algorithms

Non-modifying algorithms

They Do not modify the value of the range. They work also on constant ranges.

Example

```
It std::find(ForwardIt first, ForwardIt last, const T & value)
```

Finds the first occurrence of `value` in the range `[first, last)`.

Types of algorithms

Modifying algorithms

They either modify the given range, like

```
void std::fill(ForwardIt first, ForwardIt last, const T& value);
```

assigns the given `value` to the elements in the range `[first, last]`.

Or, they copy the result of an operation into another (existing) range. For instance

```
OutIt std::copy(InIt first, InIt last, OutIt result);
```

copies `[first, last)` into the range that starts at `result`.

Inverters

Inverters are special iterators used to insert values into a container. Three main types:

- `std::back_inserter(Container& x)` : Inserts at the back (only for sequential containers).
- `std::front_inserter(Container& x)` : Inserts in the front (only for sequential containers).
- `std::inserter(Container& x, It position)` : Inserts after the indicated position.

Example

```
std::copy(a.begin(), a.end(), std::front_inserter(c));
```

The computational cost depends on the type of container!

Example: `std::inserter`

Several algorithms require writing the output to a non-const range indicated by the iterator to its beginning. Without inserters, it would be impossible to use them on a non-sequential container or on a sequential container of insufficient size.

```
std::vector<double> a;
std::set<double> b;

std::copy(a.begin(), a.end(), b.begin()); // ERROR: b is not large enough.
```

You need an inserter:

```
std::copy(a.begin(), a.end(), std::inserter(b, b.begin())); // Ok.
```

For an associative container, the second argument of `inserter` is taken only as a suggestion.

Types of algorithms

Sorting

- Particular modifying algorithms operating on a range to order it according to an ordering relation (default: `std::less<T>`):

```
std::vector<double> a;  
  
// Descending order: a[i+1] <= a[i].  
std::sort(a.begin(), a.end(), std::greater<double>());  
  
// Ascending order: a[i+1] >= a[i].  
std::sort(a.begin(), a.end());
```

Operating on sorted ranges

- Search algorithms:

```
bool std::binary_search(It first, It last, const T& value);
```

returns true if the `value` is present.

- Set union, intersection, and difference (they do not need to be a `std::set<T>`, it is sufficient that the range is ordered):

```
std::set<int> a;
std::set<int> b;
// ...
set<int> c;
std::set_union(a.begin(), a.end(), b.begin(), b.end(), std::inserter(c, c.begin()));
```

Now $c = a \cup b$.

⚠ Remember that a `std::set` is already ordered!

Types of algorithms

Min and Max

- A series of algorithms to find the minimum and maximum element in a range:

```
template <class T>
const T& max(const T& a, const T& b);

template <class T>
const T& min(const T& a, const T& b);

template <class T, class Compare>
const T& max(const T& a, const T& b, Compare comp);

template <class T>
std::pair<const T&, const T&> minmax(const T& a, const T& b);

template <class InputIt1, class InputIt2>
bool lexicographical_compare(InputIt1 first1, InputIt1 last1, InputIt2 first2, InputIt2 last2);
```

Types of algorithms

Numeric operations

- Numeric operations are available in `<numeric>`.
- Examples:

```
std::vector<double> v;
std::vector<double> w;

// Sum of a range.
auto sum = std::accumulate(v.begin(), v.end(), 0);

// Product of a range.
auto product = std::accumulate(v.begin(), v.end(), 1, std::multiplies<double>());

// The same with lambdas.
auto product = std::accumulate(v.begin(), v.end(), 1, [](double a, double b) { return a * b; });

auto r1 = std::inner_product(v.begin(), v.end(), w.begin(), 0);
```

std::transform

- Another very flexible algorithm is `std::transform`, present in two forms:

```
OutIt transform(Init first1, Init last1, OutIt result, UnaryOperator op);
OutIt transform(Init1 first1, Init1 last1, Init2 first2, OutIt result, BinaryOperator binary_op);
```

- You can apply unary or binary functions to elements in a range.
- The length of the ranges must be consistent (no check is made).

Example

```
std::set<double> a;
std::list<double> l;
// ...

std::vector<double> b(a.size());
std::transform(a.begin(), a.end(), l.begin(), b.begin(), std::plus<double>());
```

b now contains *a + l*.

A list of other interesting algorithms (1/2)

- `std::for_each` : Apply a function to a range.
- `std::find_if` : Find the first element satisfying a predicate.
- `std::count` : Count appearances of a value in a range.
- `std::count_if` : Return the number of elements in a range satisfying a predicate.
- `std::replace` : Replace a value.
- `std::replace_if` : Replace values in a range satisfying a predicate.
- `std::replace_copy` : Copy a range while replacing values.
- `std::replace_copy_if` : Copy a range, replacing values satisfying a predicate.
- `std::fill` : Fill a range with a value.
- `std::fill_n` : Fill `n` elements with a value.
- `std::generate` : Generate values according to a given unary function.

A list of other interesting algorithms (2/2)

- `std::remove_if` : Remove elements satisfying a predicate.
- `std::remove_copy` : Remove values and copy them to another range.
- `std::remove_copy_if` : Remove elements satisfying a predicate and copy.
- `std::unique` : Remove consecutive duplicates.
- `std::random_shuffle` : Rearrange elements in a range randomly.
- `std::partition` : Partition a range into two.
- Operations on sorted ranges, such as union, intersection, etc.

Full list [here](#) and [here](#) for numerical functions and algorithms.

Parallel algorithms

- Since C++17, most STL algorithms now support parallel execution via multi-threading.
- Execution policies:
 - `std::execution::seq` : Sequential execution (no parallelization).
 - `std::execution::par` : Parallel sequenced execution.
 - `std::execution::par_unseq` : Parallel unsequenced execution (vectorization).
 - The last execution policy is activated only if the hardware supports it.
- Be careful with data races; ensure your procedure is parallelizable.
- C++ provides tools to control parallel execution finely (mutexes, etc.), but their use is complex and beyond the scope of this course.

Example: parallel algorithms

```
std::vector<int> v;

// Find element using parallel execution policy.
auto result1 = std::find(std::execution::par, std::begin(v), std::end(v), 2);

// Sort elements using sequential execution policy.
auto result2 = std::sort(std::execution::seq, std::begin(v), std::end(v));
```

Evolution of the STL

Evolution of the STL

The C++ Standard Template Library (STL) has seen several enhancements and improvements in each major C++ standard release, including C++11, C++14, C++17, C++20, and C++23. Here's a summary of the main introductions to the STL in each of these versions.

References and further reading

- [C++ reference](#)
- [**Modern C++ for Absolute Beginners**](#) : A Friendly Introduction to the C++ Programming Language and C++11 to C++23 Standards, Slobodan Dmitrović, Apress, March 2023.
- [**Evolution since C++11**](#)
- [**Learn modern C++**](#)

C++11

1. Move semantics
2. Variadic templates
3. Rvalue references
4. Lambda expressions
5. auto
6. nullptr
7. Range-based for loops
8. Smart pointers
9. Type traits
10. ...

C++14

1. Binary literals
2. Generic lambdas
3. Return type deduction
4. decltype(auto)
5. Variable templates
6. User-defined literals for standard library types

C++17

1. Template argument deduction for class templates
2. Fold expressions
3. Lambda capture `this` by value
4. Structured bindings
5. `constexpr if`
6. UTF-8 character literals
7. New library features like `std::variant`, `std::optional`, and `std::any`

C++20

1. Coroutines
2. Concepts
3. Ranges
4. Modules
5. Designated initializers
6. Template syntax for lambdas
7. `constexpr` virtual functions
8. New library features, including `std::span` and math constants

C++23 (still subject to variation)

1. **Concepts in STL**: Further adoption of concepts in STL algorithms and containers.
2. **Improved parallelism**: Expanding parallel algorithms and enhancing support for parallel execution.
3. **Reflection**: Potential support for reflection, making it easier to inspect and manipulate types at runtime.
4. **Networking library**: The Networking library might become part of the standard, adding networking capabilities.
5. **Enhanced ranges**: Expanding and refining the ranges library with new utilities.

[Source](#)

A final recommendation

C++ is continuously evolving, and to maintain backward compatibility, new features are added while very few, if any, are eliminated. However, if you adopt a specific programming style, you'll find yourself using only a subset of what C++ has to offer.

The more outdated and cumbersome features that make programming more complex and less elegant will gradually be used less and less.

It's advisable to start incorporating the new features that genuinely assist you in writing cleaner, simpler code. Most of the features illustrated here move in that direction.

But always remember: the most important aspect of your code is whether it accomplishes the right task. An elegant code that yields incorrect results is of no use.



Smart pointers, utilities, move semantics.

Lecture 07

Smart pointers, move semantics, STL utilities.

Advanced Programming - SISSA, UniTS, 2023-2024

Pasquale Claudio Africa

07 Nov 2023

Outline

1. Smart pointers
2. Move semantics
3. Exceptions
4. STL utilities
 - I/O streams
 - Random numbers
 - Time measuring
 - Filesystem

Smart pointers

RAII: Resource Acquisition Is Initialization

RAII, short for Resource Acquisition Is Initialization, plays a significant role in C++. It essentially means that an object should be responsible for both the creation and destruction of the resources it owns.

Not RAII compliant:

```
double *p = new double[10];
```

Who is responsible for destroying the resources pointed to by `p`?

RAII compliant:

```
std::array<double, 10> p;
```

The variable `p` takes care of creating 10 doubles and destroying them.

In C++, smart pointers are important tools to implement RAII.

Pointers in modern C++

In modern C++, we use different types of pointers:

- **Standard pointers:** Use them only to watch (and operate on) an object (resource) whose lifespan is independent of that of the pointer (but not shorter).
- **Owning pointers (Smart pointers):** They control the lifespan of the resource they point to.

There are three kinds:

- `std::unique_ptr` : With unique ownership of the resource. The owned resource is destroyed when the pointer goes out of scope.
- `std::shared_ptr` : With shared ownership of a resource. The resource is destroyed when the **last** pointer owning it is destroyed.
- `std::weak_ptr<T>` : A non-owning pointer to a shared resource, reserved for special use cases.

Smart pointers implement the RAI concept. For simply addressing a resource, possibly polymorphically, use ordinary pointers.

Example: the need for `std::unique_ptr` (1/4)

```
class MyClass {  
public:  
    void set_polygon(Polygon *p) {  
        polygon = p;  
    }  
private:  
    Polygon *polygon; // Polymorphic object.  
}  
  
Polygon *create_polygon(std::string t) {  
    switch (t) {  
        case "Triangle":  
            return new Triangle{...};  
        case "Square":  
            return new Square{...};  
        default:  
            return nullptr;  
    }  
}
```

Example: the need for `std::unique_ptr` (2/4)

```
MyClass a;  
a.set_polygon(create_polygon("Triangle"));
```

⚠ This design is error-prone, requiring careful handling of resources, leading to potential memory leaks and dangling pointers.

Example: the need for `std::unique_ptr` (3/4)

```
class MyClass {  
public:  
    set_polygon(std::unique_ptr<Polygon> p) {  
        polygon = std::move(p);  
    }  
private:  
    std::unique_ptr<Polygon> polygon;  
}  
  
std::unique_ptr<Polygon> create_polygon(std::string t) {  
    switch (t) {  
        case "Triangle":  
            return std::make_unique<Triangle>(...); // 'make_unique' available since C++14.  
        case "Square":  
            return std::make_unique<Square>(...);  
        default:  
            return std::unique_ptr<Polygon>(); // null pointer.  
    }  
}
```

Example: the need for `std::unique_ptr` (4/4)

```
MyClass a;  
a.set_polygon(create_polygon("Triangle"));
```

⚠ This version with `std::unique_ptr` is RAII-compliant, improving resource management.

How a `std::unique_ptr` works

A `std::unique_ptr<T>` serves as a unique owner of the object of type `T` it refers to. The object is destroyed automatically when the `std::unique_ptr` gets destroyed.

It implements the `*` and `->` dereferencing operators, so it can be used as a normal pointer. However, it can be initialized to a pointer only through the constructor.

The default constructor produces an empty (null) unique pointer, and you can check if a `std::unique_ptr` is empty by testing `if (ptr)` or using it in a boolean context.

Main methods and utilities of `std::unique_ptr`

- `std::swap(ptr1, ptr2)` : Swaps ownership.
- `ptr1 = std::move(ptr2)` : By definition, **unique** pointers cannot be copied, but their ownership can be transferred using the `std::move` utility. Moves resources from `ptr2` to `ptr1`. The previous resource of `ptr1` is deleted, and `ptr2` remains empty.
- `ptr.reset()` : Deletes the resource, making `ptr` empty.
- `ptr1.reset(ptr2)` : Equivalent to `ptr1 = std::move(ptr2)`.
- `ptr.get()` : Returns a standard pointer to the handled resource.
- `ptr.release()` : Returns a standard pointer, releasing the resource without deleting it. `ptr` becomes empty.

`std::unique_ptr` instances can be stored in standard containers, such as vectors.

Shared pointers

For instance you have several objects that **refer** to a resource (e.g., a matrix, a shape, ...) that is build dynamically (and maybe is a polymorphic object). You want to keep track of all the references in such a way that when (and only when) the last one gets destroyed the resource is also destroyed.

To this purpose you need a `std::shared_ptr<T>`. It implements the semantic of *clean it up when the resource is no longer used*.

While `std::unique_ptr` do not cause any computational overhead they are just a light wrapper around an ordinary pointer), shared pointers do, so use them only if it is really necessary.

Example: the need for `std::shared_ptr` (1/2)

```
class Data { ... };

class Preprocessor {
public:
    Preprocessor(const std::shared_ptr<Data> &data, ...) : data(data) {}
private:
    std::shared_ptr<Data> data;
};

class NumericalSolver {
public:
    NumericalSolver(const std::shared_ptr<Data> &data, ...) : data(data) {}
private:
    std::shared_ptr<Data> data;
};
```

Example: the need for `std::shared_ptr` (2/2)

```
std::shared_ptr<Data> shared_data = std::make_shared<Data>(...);  
  
Preprocessor preprocessor(shared_data, ...);  
preprocessor.preprocess();  
  
// shared_data will still be used by other resources, hence it cannot be destroyed here.  
  
NumericalSolver solver(shared_data, ...);  
solver.solve();
```

How a `std::shared_ptr` works

`std::shared_ptr` allows shared ownership of dynamically allocated objects. It keeps track of the number of shared references to an object through reference counting. When the reference count reaches zero, the object is automatically deallocated, preventing memory leaks.

`std::shared_ptr` is thread-safe, making it suitable for concurrent access. It can also be used for managing resources beyond memory and can be equipped with custom deleters.

It implements the `*` and `->` dereferencing operators as well, so it can be used as a normal pointer. Moreover, it provides copy constructors and assignment operators.

The default constructor produces an empty (null) unique pointer, and you can check if a `std::shared_ptr` is empty by testing `if (ptr)` or using it in a boolean context.

We can swap, move, get, and release a `std::shared_ptr` just as we do with `std::unique_ptr`.

Example: shared pointers

```
// Create a shared_ptr to a dynamically allocated object.  
std::shared_ptr<MyClass> shared_ptr = std::make_shared<MyClass>(42);  
  
// Access the object through the shared_ptr.  
shared_ptr->print();  
  
// Create another shared_ptr that shares ownership  
std::shared_ptr<MyClass> another_shared_ptr = shared_ptr;  
  
// Check the use count (number of shared_ptrs owning the object).  
std::cout << "Use count: " << shared_ptr.use_count() << std::endl;  
  
// Create a new shared_ptr.  
std::shared_ptr<MyClass> new_shared_ptr = std::make_shared<MyClass>(55);  
  
// The old one goes out of scope, but is still referenced by 'another_shared_ptr'.  
shared_ptr = new_shared_ptr;  
  
// Check the use count again.  
std::cout << "Use count: " << shared_ptr.use_count() << std::endl;
```

std::weak_ptr

The `std::weak_ptr` is a smart pointer that holds a non-owning (*weak*) reference to an object managed by a `std::shared_ptr`. It must be converted to `std::shared_ptr` to access the referenced object.

```
std::shared_ptr<int> ptr = std::make_shared(10);
std::weak_ptr<int> weak1 = ptr; // Get pointer to data without taking ownership.

ptr = std::make_shared(5); // Delete managed object, acquires new pointer.
std::weak_ptr<int> weak2 = sptr; // Get pointer to new data without taking ownership.

auto tmp1 = weak1.lock() // tmp1 is nullptr, as weak1 is expired!
auto tmp2 = weak2.lock() // tmp2 is a shared_ptr to new data (5).
std::cout << "weak2 value is " << *tmp2 << std::endl;
```

Reference wrappers

References create aliases to existing objects and must be initialized. It's crucial to be cautious with references to temporary objects. A const reference prolongs the life of a temporary object.

Standard containers can hold only "first-class" objects, but not references. However, you can use `std::reference_wrapper` from the `<functional>` header to store objects with reference-like semantics in a container.

```
int a = 10, b = 20, c = 30;

std::vector<std::reference_wrapper<int>> ref_vector = {a, b, c};

// Modify the original values through the reference wrappers.
for (std::reference_wrapper<int> ref : ref_vector) {
    ref.get() += 5;
}
```

Move semantics

The problem: swap may be costly

Let's consider this function that swaps the arguments:

```
void swap(Matrix& a, Matrix& b) {  
    Matrix tmp{a}; // Make a copy of a.  
    a = b;          // Copy assign b to a.  
    b = tmp;        // Copy assign tmp to b.  
}
```

If `a` and `b` are of big size, this function is very inefficient.

- **Memory inefficient:** we have to store `tmp`.
- **Computationally inefficient:** copy operations imply copying all matrix elements.

In this code, an unintended copy of the matrix occurs during the swap operation.

We need to find a way to prevent these unnecessary copies.

A better swap (before C++11)

Let's assume that `Matrix` stores dynamic data for its elements as a `double*` `data` (maybe it is better to use a standard vector, but it is not relevant here). Before the introduction of move semantics, I could have solved the problem by writing a special method or a **friend** function. For instance:

```
void swap_with_move(Matrix& a, Matrix& b) {
    // Swap number of rows and columns.

    double* tmp = a.data; // Save the pointer.
    a.data = b.data;      // Copy the pointer.
    b.data = tmp;         // Copy the saved pointer.
}
```

This way I just swap the pointers, saving memory and operations, but **only for this specific situation**. It is not generalizable: I cannot write a function template `swap_with_move<T>` because I need to know how data is stored in `T`, for each case.

Questions to be addressed

To implement move semantics, three questions have to be addressed:

1. How can we identify objects that can be safely *moved* instead of copied, so that the compiler may perform the move automatically, whenever possible?
2. How can I actually implement the move in a uniform and general way?
3. How can I specify that I want to *move*, instead of copying?

Let's give the answer one question at a time. For the first one, we need to introduce value categories.

Categories of values

In C++, a value is characterized by its *type* and its *category*, which expresses how the value can be used.

In C++, we have 4 categories for values: *glvalue*, *prvalue*, *xvalue*, and *lvalue*. Moreover, they can be const or non-const.

To simplify matters (without losing important information), we will only use 2 categories: *lvalue* (which also includes *glvalue*) and *rvalue* (which includes *prvalue* and *xvalue*).

Ivalues and rvalues in C

The original definition of lvalues and rvalues from the earliest days of C is as follows:

- | An **lvalue** is an expression that may appear on the left and on the right-hand side of an assignment.
- | An **rvalue** is an expression that *can only appear on the right hand side of an assignment.*

Example

```
double fun(); // A function returning a double.  
  
3.14 = a; // Wrong: a literal expression is an rvalue!  
fun() = 5; // Wrong: returning an object generates an rvalue!
```

Ivalues and rvalues in C++

User-defined types, `const`, and operator overloading make the definition of rvalues/lvalues rather complicated in C++. We avoid the formal definition contained in the standard (very technical). We give a simple definition, correct in most cases:

- An **lvalue** is an expression that refers to a memory location and allows us to take its address via the `&` operator.
- An **rvalue** is an expression that is not an lvalue.

For this reason, lvalue is nowadays interpreted as *locator-value* and no more left-value. It is still true that *a (non-const) rvalue can only be at the right-hand side of an assignment.*

Examples of lvalues

The value held in a variable (i.e., a value with a name) is *always* an lvalue. Even if it is `const` or a `constexpr`, since we can take its address.

```
double a;
int const b = 10;
double* pa = &a; // Address of a.
int const* pb = &b; // Address of b.
```

If a function returns an *lvalue* reference (`&`), the returned value is an lvalue.

```
double& f(double & x) { x *= 3; return x; }

double y = 8.0;
f(y) = 3.0;
double* px = &(f(y)); // Address of y.
```

Examples of rvalues

The value returned by a function is an rvalue.

```
double fun(double x) { ... }
```

Here, `&fun` is a pointer to the function, *not* to the returned value. I cannot take the address of the returned value; it's a temporary object.

Non-string literals are rvalues.

```
double* pd = &(10.5); // Error (taking the address of a temporary doesn't make sense).
```

Compilers are free not to store them in memory, so no address may be taken (and it does not make sense to take it).

Strings, however, are lvalues.

How can we identify objects that can be safely moved instead of copied?

Non-const rvalues are eligible for "automatic moving". Indeed, if we cannot take the address, it means that they exist only to be stored somewhere.

So we have the answer to the first question: *rvalues are movable*. In particular, values returned by a function are movable.

How can I actually implement the move in a uniform and general way?

To answer the second question, let's look at how *references* bind according to the category of the bound values.

We consider ordinary references first, from now on called *lvalue references*. A non-const lvalue reference *cannot bind to rvalues*, while both lvalues and rvalues can be bound to const lvalue references.

Reference binding

```
double & pi = 3.14; // Wrong: A literal expression is an rvalue.  
double const & another_pi = 3.14; // Ok!
```

```
int foo(); // Return an rvalue.  
int & foo(int & a); // Return a reference, thus an lvalue.  
int goo(const int & a); // Returns an rvalue.
```

```
auto p = foo(); // Ok: p is an int.  
int & c = foo(p); // Ok: the function returns an lvalue here!  
int & d = foo(3); // NO! 3 is an rvalue and cannot be bound to an (lvalue) reference.  
auto & x = goo(foo()); // NO! as above.  
const int & a = goo(foo()); // Ok, an rvalue binds to a const lvalue reference.
```

Reference binding in overloaded functions

The interplay between reference types and binding is clear (and important) when looking at function overloading.

```
void foo(int & a);
void foo(const int & a);
void goo(const int & a);
void zoo(int & a);

int g;
const int b = 10;

foo(5); // Calls foo(const int &)
foo(g); // Calls foo(int &)
goo(g); // Calls goo(const int &);
foo(b); // Calls foo(const int &)
goo(b); // Calls goo(const int &);
zoo(b); // Error: a const lvalue can bind only to a const lvalue reference.
```

Conclusion on lvalue reference binding

- A non-const lvalue reference can bind only, and preferably, to non-const lvalues.
- A const lvalue reference binds both to lvalues and rvalues, const and non-const alike.

Here, "preferably" means that it will be chosen in case there is a choice.

This is **before** C++11. In fact, it is still true if we just use lvalue references.

The consequence is that with just lvalue references, we cannot distinguish lvalues from rvalues.

Relation with moving

Let's examine the following code

```
Matrix foo(); // A function returning a large object.  
  
Matrix a;  
a = foo();
```

The return value of `foo` could be moved into `a` safely! (Indeed, the Return Value Optimization already does that for constructors).

It would be beneficial to have an "adornment" that acts like a reference, while ensuring that **it binds exclusively to rvalues and preferably to rvalues**. This way, we can overload the assignment operator as follows:

```
Matrix & operator=(const Matrix & a); // Ordinary copy.  
Matrix & operator=(Matrix "new adorn" a); // Move!
```

rvalue reference

Indeed, C++11 has introduced a new kind of adornment, called **rvalue reference**, indicated by
`&&`.

It **exclusively and preferably binds to rvalues**. Preferably means that, if given the choice, an rvalue binds to an rvalue reference.

An important thing to remember is that **rvalue references bind rvalues and only rvalues**.

Categories of values

We resume some rules:

- If a function returns a value, that value is considered an **rvalue**.
- If a function returns an lvalue reference (const or non-const), that value is considered an lvalue.
- If a function returns an rvalue reference, that value is an rvalue.
- A (named) variable is **always an lvalue**.

This is fundamental for move semantics.

How is move semantics implemented?

We are now able to answer the second question. The key is **the move constructor and the move assignment operators**.

This is the standard signature of move operations for a class named `Matrix` :

```
Matrix(Matrix&&); // Move constructor.  
Matrix & operator=(Matrix&&); // Move assignment operator.
```

Remember that unless you have defined some other constructors or the copy assignment, the compiler provides a synthetic move constructor and move assignment operator automatically, which apply the corresponding moving operation on the non-static data members of the class.

Move semantics for Matrix (1/2)

Let's go back to `Matrix`. Assume that `Matrix` stores the data as a pointer to `double`. A possible copy-constructor and copy-assignment take the form:

```
Matrix(const Matrix & rhs) : nr(rhs.nr), nc(rhs.nc), data(new double[nr * nc]) {  
    // Make a deep copy.  
    for (i = 0; i < rhs.nr * rhs.nc; ++i)  
        data[i] = rhs.data[i];  
}  
  
Matrix & operator=(const Matrix & rhs) {  
    // Release current resource.  
    delete[] this->data;  
    // Get a new data buffer.  
    data = new double[rhs.nr * rhs.nc];  
    // Make a deep copy.  
    for (the i = 0; i < rhs.nr * rhs.nc; ++i)  
        data[i] = rhs.data[i];  
}
```

Move semantics for Matrix (2/2)

The corresponding move operator could be:

```
Matrix(Matrix&& rhs) : nr(rhs.nr), nc(rhs.nc), data(rhs.data) {  
    // Fix rhs so it is a valid empty matrix.  
    rhs.data = nullptr;  
    rhs.nr = rhs.nc = 0;  
}  
  
Matrix & operator=(Matrix&& rhs) {  
    delete[] this->data; // Release the resource.  
    data = rhs.data; // Shallow copy.  
    // Fix rhs so it is a valid empty matrix.  
    rhs.data = nullptr;  
    rhs.nr = rhs.nc = 0;  
}
```

I just grab the resource and leave an empty matrix! **It is important to ensure that the moved object can be deleted correctly!**

The consequence

```
Matrix foo();  
// ...  
Matrix a;  
a = foo(); // A move assignment is called.
```

We can say that a class implements move semantics when the move operators are defined, even if they are automatically by the compiler.

Move semantics and perfect forwarding

Now, let's address the third question: **How can I explicitly instruct the compiler to perform a move instead of a copy operation when move semantics are implemented (possibly with the synthesized move operators)?** This question can be divided into two parts:

Move: How to explicitly tell the compiler to replace a copying operation with a move if move semantics are implemented (perhaps with the synthesized move operators).

Perfect forwarding: How to write function templates that accept arbitrary arguments and forward them to other functions in a way that the target functions receive the values with the same category they were passed to the forwarding function. *This topic will not be covered in this course but you can find a good explanation [here](#).*

Forcing a move: `std::move`

Well, first of all, `std::move` doesn't move anything. They have chosen a wrong name; they should have called it `std::movable` instead. But we have to live with it.

`std::move(expr)` unconditionally casts `expr` to an rvalue. So it makes it available to be moved.

You use it to indicate to the compiler that you want something to be moved, even if it is an lvalue. It is actually moved if move semantics has been implemented for that type. If not, it will be copied.

A new (generic) version of swap

Now we are able to write our `swap`, and in a generic way!

```
template<class T>
void swap(T& a, T& b) {
    T tmp{std::move(a)}; // Move constructor.
    a = std::move(b);   // Move assignment operator.
    b = std::move(tmp); // Move assignment operator.
}
// Or, even simpler:
std::swap(a, b);
```

⚠ If your class stores its dynamic and potentially large data in standard containers, you just need the synthetic move operators (which means that you have move semantics for free!). Another good reason to use standard containers.

⚠ If type `T` implements move semantic, the swap is made using the move operators, and, if implemented correctly, with less memory requirement. If not, we have the usual copy.

Once more: variables are *always* lvalues

Named variables are **always lvalues**! Even if they are declared as rvalue references. In fact, you can take their address!

In particular, **function parameters (of any function, including constructors) are lvalues**, even if their type is an rvalue reference.

Inside the scope of this function:

```
void f(Matrix&& m) {  
    // ...  
}
```

m is an **lvalue**.

The solution

You have to force the move:

```
class Foo {  
public:  
    Foo(Matrix&& m) : my_m{std::move(m)} {}  
    // ...  
private:  
    Matrix my_m;  
}
```

Now, `my_m{std::move(m)}` calls the **move constructor**, and `m` is moved into `my_m`.

What does move semantics have to do with the STL?

All standard containers support move semantic, and all standard algorithms are written so that if the contained type implements move semantics, the creation of unnecessary temporaries can be avoided. All containers also have a `swap()` method that performs swaps intelligently.

Smart pointers supports move (but `std::unique_ptr` disallows copy).

For instance, `std::sort()` (which does a lot of swaps) is much more efficient on dynamically sized objects if move semantics are implemented.

Move semantics also make a few (but not all) template metaprogramming techniques now used in some libraries, like [Eigen](#), to avoid unnecessary large size temporaries.

Exceptions

Preconditions, postconditions, and invariants

In software development, a function (or method) can be seen as a mapping from input data to output data. The software developer specifies the conditions under which the input data is considered valid; this specification is called a **precondition**. The developer also guarantees that the expected output, called a **postcondition**, is provided when the input adheres to the precondition. Failure to meet these conditions is considered a **fault** or **bug** in the code.

An **invariant** of a class is a condition that must be satisfied by the state of an object at any point in time, except for transient situations like the object's construction process. An object is considered to be in an **inconsistent state** if the invariants are not met.

The verification of preconditions, postconditions, and invariants is an integral part of **code verification** during the development phase.

An example

Consider a function in C++:

```
Matrix cholesky(const Matrix& m);
```

- This function has a **precondition** that requires the input matrix `m` to be symmetric positive definite.
- The **postcondition** is that the output matrix is a lower triangular matrix representing the Cholesky factorization of `m`.
- An invariant of a symmetric matrix `m` is that `m(i, j) = m(j, i)` for all matrix elements.

Run-time assertions

Example

```
double calculate(double operand1, double operand2) {  
    assert(operand2 != 0 && "Operand2 cannot be zero.");  
  
    const double result = // ...  
  
    assert(result >= 0 && "Negative result!");  
  
    return result;  
}
```

For improved efficiency, all assertions can be disabled (i.e. the argument to `assert()` will be ignored) by defining the `NDEBUG` preprocessor macro, for instance:

```
g++ -DNDEBUG main.cpp -o main
```

Compile-time assertions

Example

```
template <typename T, int N>
class MyClass {
public:
    MyClass() {
        // Here goes a condition that can be evaluated at compile-time, such as constexpr.
        static_assert(std::is_arithmetic_v<T> && N > 0, "Invalid template arguments.");
        // ...
    }
};
```

If the condition is met, the error message is printed to the standard error and compilation will fail.

Exceptions

An **exception** is an anomalous condition that disrupts the normal flow of a program's execution when left unhandled. It is not the result of incorrect coding but rather arises from challenging or unpredictable circumstances.

Examples of exceptions include running out of memory after a `new` operation, failing to open a file due to insufficient privileges, or encountering an invalid floating-point operation (floating-point exception or FPE) that cannot be easily predicted.

It's essential to note that an **incorrect behavior** (e.g., failure to meet a postcondition for correct input data) stemming from incorrect coding is **not** an exception; it is a bug that should be debugged.

Why handling exceptions

Historically, in scientific computing, exceptions were often not handled at all or led to program termination with an error message. However, the rise of graphical interfaces and more complex software systems has made exception handling more critical. An algorithm's failure should not lead to the termination of the entire program.

There is a growing need to perform *recovery* operations when exceptions occur.

Exception handling in C++

C++ provides an effective mechanism to handle exceptions. The basic structure consists of:

- Using the `throw` command to indicate that an exception has occurred. You can throw an object containing information about the exception.
- Employing the `try-catch` blocks to catch and handle exceptions. If an exception is not caught, it will propagate up the call stack and might lead to program termination.

The `try` block contains the code that might `throw` an exception, while the `catch` block handles the exception.

Example

```
int divide(int dividend, int divisor) {
    if (divisor == 0) {
        throw std::runtime_error("Division by zero is not allowed");
    }
    return dividend / divisor;
}

try {
    const int result = divide(10, 0); // Attempt to divide by zero.
    std::cout << "Result: " << result << std::endl;
} catch (const std::exception& e) {
    std::cerr << "Exception caught: " << e.what() << std::endl;
}
```

Standard exceptions

The Standard Library in C++ provides predefined **exception classes** for common exceptions. They are accessible through the `<exception>` header. These classes derive from `std::exception`, which defines a method `what()` to return an exception message.

```
virtual char const * what() const noexcept;
```

These standard exceptions are designed to be used or derived from when creating your own exceptions. This promotes consistency and helps others understand your error handling approach.

An overview of standard exceptions

- **std::exception**: The base class for all standard exceptions. It provides a `what()` method to retrieve an error message.
- **std::runtime_error**: Represents runtime errors.
- **std::logic_error**: Represents logical errors in the program. It includes exceptions like `std::invalid_argument` and `std::domain_error`.
- **std::overflow_error**: Indicates arithmetic overflow errors.
- **std::underflow_error**: Indicates arithmetic underflow errors.
- **std::range_error**: Indicates errors related to out-of-range values.
- **std::bad_alloc**: Used to indicate memory allocation errors.
- **std::bad_cast**: Indicates casting errors during runtime type identification (RTTI).
- **std::bad_typeid**: Used for errors related to the type identification of objects.
- **std::bad_exception**: A placeholder for all unhandled exceptions.

Example: custom exception handling in C++ (1/3)

```
class InsufficientFundsException : public std::exception {
public:
    InsufficientFundsException(double balance, double withdrawal_amount)
        : balance(balance), withdrawal_amount(withdrawal_amount) {}

    const char * what() const noexcept override {
        return "Insufficient Funds: Cannot complete the withdrawal.";
    }
};

    double get_balance() const { return balance; }

    double get_withdrawal_amount() const { return withdrawal_amount; }

private:
    double balance;
    double withdrawal_amount;
};
```

Example: custom exception handling in C++ (2/3)

```
class BankAccount {  
public:  
    BankAccount(double initial_balance) : balance(initial_balance) {}  
  
    void withdraw(double amount) {  
        if (amount <= 0) {  
            throw std::range_error("The requested amount is negative.");  
        }  
  
        if (amount > balance) {  
            throw InsufficientFundsException(balance, amount);  
        }  
        balance -= amount;  
    }  
  
    double get_balance() const {  
        return balance;  
    }  
  
private:  
    double balance;  
};
```

Example: custom exception handling in C++ (3/3)

```
Bank_account account(1000.0);

try {
    account.withdraw(1500.0);
    // Or: account.withdraw(-500.0);
} catch (const InsufficientFundsException& e) {
    std::cerr << "Exception caught: " << e.what() << std::endl;
    std::cerr << "Balance: " << e.get_balance()
        << ", Withdrawal amount: " << e.get_withdrawal_amount() << std::endl;
} catch (const std::range_error& e) {
    std::cerr << "Exception caught: " << e.what() << std::endl;
} catch (...) {
    std::cerr << "Unknown exception caught." << std::endl;
}
```

Old-style error control

In situations where an algorithm's failure is one of its expected outcomes (e.g., the failure of convergence in an iterative method), returning a **status** rather than throwing an exception may be more suitable. Instead of terminating the program, a status variable is used to indicate the outcome, which can be checked by the caller. See also `std::terminate`, `std::abort`, and, `std::exit`.

Exception handling is increasingly important in code that must be integrated into a broader workflow or graphical interface. However, it's worth noting that the `try-catch` mechanism introduces some inefficiencies since it checks for exceptions every time a function is called. High-performance code often minimizes the use of exception handling.

In practical contexts where exception handling is necessary, the `noexcept` declaration can help optimize efficiency by indicating functions and methods that do not throw exceptions.

Floating point exceptions

It's important to note that **floating point exceptions** (FPE) are a special type of exception. In IEEE-compliant architectures, invalid arithmetic operations on floating-point numbers do not result in program failure. Instead, they produce special numerical values like `inf` (infinity) or `nan` (not-a-number), and the operations continue.

This unique behavior distinguishes floating point exceptions from traditional exceptions.

There are ways, not covered in this course, to properly handle FPEs.

STL utilities

STL utilities: I/O streams

I/O streams

Input/Output (I/O) streams in C++ provide a convenient way to perform input and output operations, allowing you to work with various data sources and destinations, such as files, standard input/output, strings, and more. C++ I/O streams are part of the Standard Library (STL) and are based on the concept of streams. The key components of C++ I/O streams are

`iostream` , `ifstream` , `ofstream` , and `stringstream` .

- `iostream` : The base class for input and output streams. It is derived from `istream` (for input) and `ostream` (for output). It is used for interacting with the standard input and output streams.

```
int number;
std::cout << "Enter a number: ";
std::cin >> number;
std::cout << "You entered: " << number << std::endl;
```

File streams: open modes

The `std::ios_base` namespace defines the following options to deal with files.

Option	Description
<code>in</code>	File open for reading: the internal stream buffer supports input operations.
<code>out</code>	File open for writing: the internal stream buffer supports output operations.
<code>binary</code>	Operations are performed in binary mode rather than text.
<code>ate</code>	The output position starts at the end of the file.
<code>app</code>	All output operations happen at the end of the file, <code>app</code> ending to its existing contents.
<code>trunc</code>	Any contents that existed in the file before it is open are truncated/discarded.

std::ifstream

std::ifstream : This class is used for reading data from files. You can open a file for input and read data from it.

```
std::ifstream file("example.txt", open_mode);

if (file.is_open()) {
    std::string line;
    while (std::getline(file, line)) {
        std::cout << line << std::endl;
    }
    file.close();
} else {
    std::cerr << "Failed to open the file." << std::endl;
}
```

std::ofstream

std::ofstream : This class is used for writing data to files. You can open a file for output and write data to it.

```
std::ofstream file("output.txt", open_mode);

if (file.is_open()) {
    file << "Hello, World!" << std::endl;
    file.close();
} else {
    std::cerr << "Failed to open the file." << std::endl;
}
```

std::stringstream

std::stringstream : This class allows you to work with strings as if they were input and output streams. You can use stringstream for parsing and formatting strings.

```
// Using std::stringstream to format data into a string.  
std::stringstream ss;  
const int num = 42;  
const double pi = 3.14159265359;  
  
ss << "The answer is: " << num << ", and Pi is approximately " << pi;  
  
std::cout << ss.str() << std::endl;  
  
// Parsing data from a string using std::stringstream.  
std::string input = "123 45.67";  
int parsed_int;  
double parsed_double;  
  
std::stringstream(input) >> parsed_int >> parsed_double;
```

I/O formatting

Formatting: I/O streams provide various formatting options to control the appearance of output.

For instance, `std::setw`, `std::setprecision`, `std::setfill`, etc., from the `<iomanip>` header, allow setting field width, precision, and fill characters in the output.

```
const double pi = 3.14159265359;
std::cout << "Default: " << pi << std::endl;
std::cout << "Fixed with 2 decimal places: " << std::fixed << std::setprecision(2) << pi << std::endl;
std::cout << "Scientific notation: " << std::scientific << pi << std::endl;
std::cout.setprecision(6);
std::cout << "Width 10 with left alignment: " << std::left << std::setw(10) << pi << ";" << std::endl;
std::cout << "Width 10 with right alignment: " << std::right << std::setw(10) << std::setfill('*') << pi << std::endl;
```

Output:

```
Default: 3.14159
Fixed with 2 decimal places: 3.14
Scientific notation: 3.141593e+00
Width 10 with left alignment: 3.14e+00 ;
Width 10 with right alignment: **3.14e+00
```

STL utilities: random numbers

Random numbers

The capability of generating random numbers is essential not only for statistical purposes but also for internet communications. But an algorithm is deterministic. However, several techniques have been developed to generate pseudo-random numbers. They are not really random, but they show a low level of auto-correlation.

C++ support for statistical distributions

C++ provides extensive support for (pseudo) random number generators and univariate statistical distributions. You need the header `<random>`. The chosen design is based on two types of objects:

1. **Engines**: They serve as a stateful source of randomness, providing random unsigned integer values uniformly distributed in a range. They are normally used with distributions.
2. **Distributions**: They specify how values generated by the engine have to be transformed to generate a sequence with prescribed statistical properties. The design separates the (pseudo) random number generators from their use to generate a specific distribution.

Engines

Random number engines generate pseudo-random numbers using seed data as an entropy source. Several different classes of pseudo-random number generation algorithms are implemented as templates that can be customized. Some basic engines include:

- `linear_congruential_engine` : Linear congruential algorithm
- `mersenne_twister_engine` : Mersenne twister algorithm
- `subtract_with_carry_engine` : Subtract-with-carry algorithm (a lagged Fibonacci)
- Many more available in the `<random>` header

For simplicity, the library provides predefined engines, such as `std::default_random_engine`, which balances efficiency and quality. There are also non-deterministic engines, like `std::random_device`, which generate non-deterministic random numbers based on hardware data.

Engines

You can generate an object of the chosen class either with the default constructor or by providing a seed (an unsigned integer). If you use the same seed, the sequence of pseudo-random numbers will be the same every time you execute the program.

```
std::default_random_engine rd1;           // With a default-provided seed.  
std::default_random_engine rd2{1566770}; // With a user-provided seed.
```

How to use the `random_device`

The `random_device` provides non-deterministic random numbers based on hardware data. However, it is slower than other engines and is often used to generate the seed for another random engine. Here's how to use it:

```
std::random_device rd;  
std::default_random_engine rd3{rd()}; // With a random generated seed.
```

Default distributions in the STL

- `std::uniform_int_distribution`, `std::uniform_real_distribution`
- `std::normal_distribution`, `std::lognormal_distribution`,
`std::exponential_distribution`
- `std::binomial_distribution`, `std::poisson_distribution`,
- `std::geometric_distribution`, `std::bernoulli_distribution`
- `std::discrete_distribution`
- `std::piecewise_constant_distribution`, `std::piecewise_linear_distribution`
- You can create custom distributions by subclassing the `std::random_distribution` class and providing your own probability distribution function.

Distributions

Distributions are template classes that implement a call operator `()` to transform a random sequence into a specific distribution. You need to pass a random engine to the distribution to generate numbers according to the desired distribution. For example:

```
std::random_device rd;
std::default_random_engine gen{rd()};
std::uniform_int_distribution<> dice{1, 6};

for (unsigned int n = 0; n < 10; ++n)
    std::cout << dice(gen) << ' ';

std::cout << std::endl;
```

Here, `uniform_int_distribution` provides an integer uniform distribution in the range (1, 6).

seed_seq

The utility `std::seed_seq` consumes a sequence of integer-valued data and produces a requested number of unsigned integer values. It provides a way to seed multiple random engines or generators that require a lot of entropy.

For example, the internal state of the `mt19937` generator is represented by 624 integers, hence the best way to seed it is to fill it with 624 numbers based on a high-entropy source (e.g., the `random_device` provided by the operating system):

```
std::random_device rd{};
std::array<std::uint32_t, 624> seed_data;
std::generate(seed_data.begin(), seed_data.end(), std::ref(rd));
std::seed_seq seq(seed_data.begin(), seed_data.end());

std::mt19937 gen{seq};
```

You can use the generated seeds to feed different random engines.

Shuffling

In C++, you can shuffle a range of elements using the `std::shuffle` utility from the `<algorithm>` header. It shuffles the elements randomly so that each possible permutation has the same probability of appearance. Here's an example:

```
std::vector<int> v = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};  
std::random_device rd;  
std::default_random_engine g{rd()};  
std::shuffle(v.begin(), v.end(), g);
```

Every time you run this code, the vector `v` will be shuffled differently.

Sampling

Another useful utility in `<algorithm>` is `std::sample`, which extracts `n` elements from a range without repetition and inserts them into another range. Here's an example:

```
int n = 10;
std::vector<double> p;
// Fill p with more than n values to sample.
std::vector<double> res;
auto seed = std::random_device{}();
std::sample(p.begin(), p.end(), std::back_inserter(res), n, std::mt19937{seed});
```

This code generates a different realization of the sample every time you run it.

STL utilities: Time measuring

Time measuring

C++ provides three common clocks:

- `std::chrono::system_clock` : Represents the system-wide real-time clock. It's suitable for measuring absolute time (can change if the user changes the time on the host machine).
- `std::chrono::steady_clock` : Represents a steady clock that never goes backward. It's suitable for measuring time intervals and performance measurements.
- `std::chrono::high_resolution_clock` : Represents a high-resolution clock with the smallest possible tick duration. It's often used for precise timing.

Example: time measuring

```
void my_function() {
    // Code to measure.
}

auto start = std::chrono::high_resolution_clock::now();
my_function();
auto end = std::chrono::high_resolution_clock::now();

auto duration =
    std::chrono::duration_cast<std::chrono::microseconds>(end - start);

std::cout << "Time taken by function: "
    << duration.count() << " microseconds" << std::endl;
```

Example: benchmarking

```
void my_function() {
    // Code to measure.
}

const int num_iterations = 1000;

auto start = std::chrono::high_resolution_clock::now();
for (int i = 0; i < num_iterations; ++i) {
    my_function();
}
auto end = std::chrono::high_resolution_clock::now();

auto duration =
    std::chrono::duration_cast<std::chrono::microseconds>(end - start);

std::cout << "Average time taken by function: "
    << duration.count() / num_iterations << " microseconds" << std::endl;
```

STL utilities: Filesystem

Filesystem

Since C++17, a full set of utilities to manipulate files, directories, etc. in a filesystem is available.

```
const auto big_file_path{"big/file/to/copy"};

if (std::filesystem::exists(big_file_path)) {
    const auto big_file_size{std::filesystem::file_size(big_file_path)};

    std::filesystem::path tmp_path{/tmp};

    if (std::filesystem::space(tmp_path).available > big_file_size) {
        std::filesystem::create_directory(tmp_path.append("example"));
        std::filesystem::copy_file(big_file_path, tmp_path.append("new_file"));
    }
}
```

A final recommendation

C++ is continuously evolving, and to maintain backward compatibility, new features are added while very few, if any, are eliminated. However, if you adopt a specific programming style, you'll find yourself using only a subset of what C++ has to offer.

The more outdated and cumbersome features that make programming more complex and less elegant will gradually be used less and less.

It's advisable to start incorporating the new features that genuinely assist you in writing cleaner, simpler code. Most of the features illustrated here move in that direction.

But always remember: the most important aspect of your code is whether it accomplishes the right task. An elegant code that yields incorrect results is of no use.



Static and shared libraries.

Exercise session 06

The Standard Template Library, smart pointers and move semantics.

Advanced Programming - SISSA, UniTS, 2023-2024

Pasquale Claudio Africa

09 Nov 2023

Exercise 1: Monte Carlo estimate of π

In this exercise, you will perform a Monte Carlo simulation to estimate π .

1. Consider the square $[0, 1]^2$ and the quarter-circle centered at $(0, 0)$ with radius 1.
2. Generate random points within the square.
3. Count how many of these random points fall within the quarter-circle.
4. After generating a sufficient number of random points, you can estimate

$$\pi \approx \frac{4 \cdot \text{Number of points inside the quarter-circle}}{\text{Total number of generated points}}.$$

To improve estimation accuracy, try to increase the number of random points in your simulation.

Exercise 2: std::set (1/2)

In a building security system, door locks are opened by entering a four-digit access code into a keypad. The access code's validation process is handled through an Access object with the following interface:

```
class Access
{
public:
    void activate(unsigned int code);
    void deactivate(unsigned int code);
    bool is_active(unsigned int code) const;
};
```

Each employee is assigned a unique access code, which can be activated using the `activate()` function. When an employee leaves the company, their access code can be deactivated using the `deactivate()` function.

Exercise 2: `std::set` (2/2)

Your task is to implement the `Access` class as described above. Write a test program that accomplishes the following tasks:

1. Create an instance of the `Access` object.
2. Activate the access codes 1234, 9999, and 9876.
3. Prompt the user to enter an access code, and read the code from the console.
4. Inform the user whether the door opens successfully.
5. Repeat the last two steps until the door successfully opens.
6. Deactivate the code that worked. Also, deactivate the code 9999 and activate the code 1111.
7. Repeat steps 3 and 4 until the door successfully opens.

Exercise 3: `std::map` (1/2)

In the previous exercise, the customer using the security system wants to associate an access level with each access code. Users with higher access levels should be able to open doors to more security-sensitive areas of the building compared to users with lower access levels. Start with your solution from the previous exercise and make the following modifications to the `Access` class:

```
class Access
{
public:
    void activate(unsigned int code, unsigned int level);
    void deactivate(unsigned int code);
    bool is_active(unsigned int code, unsigned int level) const;
};
```

The `is_active()` function should return `true` if the specified access code has an access level greater than or equal to the specified access level. If the access code is not active at all, it should return `false`.

Exercise 3: `std::map` (2/2)

Now, update the main program to perform the following tasks:

1. After creating an instance of the `Access` object, activate code 1234 with access level 1, code 9999 with access level 5, and code 9876 with access level 9.
2. Prompt the user to enter an access code, and read the code from the console.
3. Assuming a door requires access level 5 for entry, print whether it opened successfully.
4. Repeat the last two steps until the door opens.
5. Deactivate the code that worked, change the access level of code 9999 to 8, and activate code 1111 with access level 7.
6. Prompt the user for an access code, read the code from the console.
7. Assuming a door requires access level 7 for entry, print whether it opened successfully.
8. Repeat the last two steps until the door opens.

Exercise 4: STL containers and algorithms

- 1. Generate a vector:** Create a vector named `random_numbers` that contains 100 random integers between 0 and 9.
- 2. Sort the vector:** Create a new vector named `sorted_numbers` by sorting the `random_numbers` vector in ascending order, with repetitions.
- 3. Remove duplicates while sorting:** Create a new vector named `sorted_unique_numbers` by sorting the `random_numbers` vector and removing duplicate entries.
- 4. Remove duplicates without sorting:** Create a new vector named `unsorted_unique_numbers` by printing unique entries from the `random_numbers` in the same order they appear, without repetitions.

Exercise 5: Word frequency analysis

The file `input.txt` contains a list of random complete sentences in English. Develop a C++ program that reads the file, calculates the frequency of each word in the text, and outputs the word-frequency pairs to a new file in a dictionary format.

Write a C++ program to process the input text file by splitting it into words and counting the occurrences of each unique word. Spaces and punctuation should be discarded.

The program should generate a new file (named `output.txt`) containing the word-frequency pairs in a dictionary format. Each line in the output file should consist of a word followed by its frequency, separated by a colon or any other suitable delimiter.

(Bonus): sort the output by frequency, in descending order. If two words have the same frequency, then sort them alphabetically.

Exercise 6: Move semantics for efficient data transfers

Define a class `Vector` that represents a one-dimensional vector of double values, stored as a raw pointer `double *data`.

1. Implement a **move constructor** for the `Vector` class that transfers ownership of the underlying data from the source vector to the destination vector. The move constructor should ensure that the source vector's data is no longer accessible after the transfer.
2. Define a **copy** and a **move assignment operator** for the `Vector` class that allows for the efficient transfer of ownership of the underlying data from one `Vector` object to another. Similar to the move constructor, the move assignment operator should ensure that the source vector's data is no longer accessible after the transfer.
3. Compare the performance of copying and moving large vectors using both copy semantics and move semantics. Measure the time taken to copy and move vectors by increasing the input size from 2^{20} to 2^{30} elements. Analyze the results and observe the performance gain achieved by using move semantics.

Lecture 08

Libraries: principles, generation and use.

Advanced Programming - SISSA, UniTS, 2023-2024

Pasquale Claudio Africa

14 Nov 2023

Outline

1. What is a library?
2. Types of library
 - Header-only
 - Static
 - Shared (dynamic)
 - How to link against libraries
3. Static libraries
 - How to build
3. Shared libraries
 - The linking phase
 - The loading phase
 - How to build
 - Dynamic loading
4. How to compile (and use) third-party libraries?

What is a library?

What is a library?

A library is a collection of pre-written code or routines that can be reused by computer programs. These libraries typically contain functions, variables, classes, and procedures that perform common tasks, allowing developers to save time and effort by leveraging existing code rather than writing everything from scratch.

Libraries can be specific to a programming language or more general-purpose, applicable across different languages. Examples include the Standard Template Library (STL) in C++, the Java Class Library, and the Python Standard Library. Additionally, there are third-party libraries created by developers and organizations to extend the capabilities of programming languages or frameworks.

Why libraries are useful?

- **Code reusability:** Libraries provide a set of functionalities that can be used across multiple projects, reducing the need to write the same code over and over again.
- **Modularity:** Libraries promote modular programming by encapsulating specific functionalities into separate modules or components.
- **Abstraction:** Libraries abstract the underlying implementation details, allowing developers to use high-level interfaces without needing to understand the inner workings of the functions provided by the library.
- **Collaboration:** Communities of developers can share and collaborate on libraries, accelerating the development process. Many programming languages have centralized repositories or package managers to facilitate the distribution and installation of libraries.
- **Efficiency:** Libraries are often optimized and well-tested, providing efficient and reliable solutions for common programming tasks.

Components of a C++ library (1/2)

A library provides utilities that may be used to produce executable code. In C or C++, it is usually formed by:

- A set of **header files** that provide the *public interface* of the library, necessary for those who develop software using the library.
- One or more **library files** that contain, in the form of machine code, the *implementation* of the library. They may be *static* and *shared* (also called *dynamic*).

As an exception, there are libraries whose implementation is only contained in header files (thanks to *inline* functions and templates).

These are called *header-only* libraries, and are the easiest to use.

An example of such is [Eigen](#), a powerful library for linear algebra.

Components of a C++ library (2/2)

Header files are only used in the development phase. In production, only **library files** are needed.

Precompiled executables that just use **shared libraries** do not need header files to work. This is why certain software packages are divided into standard and *development* versions; only the latter contains the full set of header files.

For example:

```
sudo apt install python3
```

will install the `python3` executable and the shared libraries it requires to be run, whereas

```
sudo apt install libpython3-dev
```

will download header files, libraries and tools required for building applications **based on** the source code of Python3 (called `cPython`, written in C).

Curated lists of awesome C++ and Python frameworks, libraries, resources, and shiny things.

- Popular GitHub repositories using C++ (≥ 4500)!
- [awesome-cpp](#)
- [awesome-python](#)
- [awesome-scientific-python](#)
- [awesome-scientific-computing](#)

Types of library

The build process

Header-only libraries

A library formed only by class templates and function templates contains only header files. One example is [Eigen](#), but many others are available.

Using a header-only library is very simple: you have to store the header files in a directory later searched by the preprocessor.

So either you store them in a system include directory, like `/usr/include` or `/usr/local/include` (you must have administrator privileges), or in a directory of your choice that you will then indicate using the `-I` option of the compiler (actually, of the **preprocessor**).

```
g++ -I/path/to/library/include/ ...
```

Header-only libraries: example

```
# Download Eigen 3.4.0.  
wget https://gitlab.com/libeigen/eigen/-/archive/3.4.0/eigen-3.4.0.tar.gz  
  
# Extract the archive to your Desktop.  
tar xzvf eigen-3.4.0.tar.gz -C ${HOME}/Desktop  
  
# Compile and run 'example/eigen.cpp'.  
g++ -I${HOME}/Desktop/eigen-3.4.0 eigen.cpp -o main_eigen && ./main_eigen
```

As simple as that.

From now on, however, we will deal with libraries that contain machine code, not header-only libraries.

Static vs. shared libraries

- **Static library:** A static library, often denoted by a `.lib` (on Windows) or `.a` (on Unix-like systems) file extension, contains compiled code that is linked directly into an executable at compile time. When you build a program using a static library, a copy of the library's code is included in the final executable. This means that the resulting executable is independent of the original library file; it contains all the necessary code to run without relying on external library files.
- **Shared library (Dynamic Link Library `.dll` on Windows, Shared Object `.so` on Unix-like systems, Dynamic Library `.dylib` on macOS):** A shared library contains code that is *loaded at run-time* when the program starts or during execution. Instead of being included in the executable, the program *references* the shared library, and the operating system *loads the library into memory* when needed. Multiple programs can use the same shared library, which can result in more efficient use of system resources.

A guided example

main.cpp (developed by me)

```
#include "mylib.hpp"
...
myfun();
...
```

mylib (developed by somebody else)

```
// mylib.hpp
void myfun();

// mylib.cpp
#include "mylib.hpp"

void myfun() {}
```

The build process: preprocessing + compilation

The preprocessing and compilation steps

```
g++ -Imylib/ -c main.cpp
```

produce the object file `main.o`. What does it contain?

```
$ nm -C main.o
0000000000000000 T main
                      U myfun()
```

The `T` in the second column indicates that the function `main()` is actually defined (resolved) by the library. While `myfun()` is referenced but undefined. So, to produce a working executable, you have to specify to the linker another library or object file where it is defined.

Indeed, the linking phase fails...

```
$ g++ main.o -o main  
/usr/bin/ld: main.o: in function `main':  
main.cpp:(.text+0x9): undefined reference to `myfun()'  
collect2: error: ld returned 1 exit status
```

Case 1: I have access to the implementation of myfun()

Step 1: compile the object file implementing myfun()

```
g++ -c mylib.cpp
```

Step 2: link my application against that object file

```
g++ main.o mylib/mylib.o -o main
```

Now both main and myfun are resolved:

```
$ nm -C main
00000000000011a9 T main
00000000000011bd T myfun()
...
```

Case 2: the reality

Real-case scenarios are typically much more complex because:

1. **Compilation takes time!**
2. One may need to use symbols defined in **multiple object files**, and compiling all of them and/or carrying out the whole list of object file names can be tedious.
3. If a change is made in `mylib` or it is updated, one has to *recompile* `mylib` and *relink* all their applications using `mylib`.
4. Developers of `mylib` may not be so nice: they want to **hide the actual implementation**. They are ok with providing users with `mylib.hpp` and the corresponding *machine code* (which is not human-readable), but not `mylib.cpp`.
5. **Dealing with multiple dependencies** makes the complexity increase.

This is why, typically, developers of a library provide users with *header files* and a *library file*.

The build process: linking against an external library

Option 1: indicate its full path during linking:

```
g++ main.o /path/to/mylib/libmylib.a -o main
```

Option 2: use the **-L<dir> -l<libname>** options.

```
g++ main.o -L/path/to/mylib -lmylib -o main
```

`-L<dir>` is not needed if the library is stored in a standard directory (typically `/usr/lib` or `/usr/local/lib`).

⚠ Note that `libxx.a` becomes `-lxx`.

⚠ If the linker finds a shared library with the same name available in the system and/or in the specified directories, it is given the precedence. If you want to override this behavior, use the `-static` flag.

Inspecting the content of a library

The command `nm` works not only with object files and executables, but also with libraries:

```
$ nm -C libmylib.a  
...  
0000000000000000 T myfun()  
...
```

Besides `T` and `U`, the command may use other letters. The most important ones are:

- `D` or `G`: The symbol refers to initialized data.
- `V` or `W`: The symbol is a weak symbol. It basically means that the (ODR) One Definition Rule will not be applied by the linker on those symbols.

A note: If a function declared `inline` has been actually inlined, the corresponding symbol is not present, since `inline` in this case really means `inline`. The same happens for a `constexpr` function. If the compiler instead decides to treat them as normal functions, the symbol is marked `W`.

Static libraries: how to use and how to build

Static libraries

Static libraries are the *oldest* and most basic way of integrating third-party code. They are basically a collection of object files stored in a single archive.

At the linking stage of the compilation processes, the symbols (which identify objects used in the code) that are still unresolved (i.e., they have not been defined in that translation unit) are searched into the other object files indicated to the linker and in the indicated libraries, and eventually the corresponding code is inserted in the executable.

How to build a static library?

In practice, libraries result themselves from preprocessing and compiling their corresponding source codes. In our example:

```
g++ -c mylib.cpp  
ar rs libmylib.a mylib.o
```

More in general, a **static library** is just an archive collecting object files:

```
g++ -c a.cpp b.cpp c.cpp d.cpp // Create object files.  
ar rs libxx.a a.o b.o c.o  
ar rs libxx.a d.o // You can add one more.
```

Option `r` adds/replaces an object in the library. Option `s` adds an index to the archive, making it a searchable library.

The command `ar -t libxx.a` lists all object files contained in the archive.

Order matters!

When linking with static libraries, the order matters. Libraries should be listed in reverse order of dependency. Libraries that depend on symbols from other libraries should come first in the list.

So, for example, if `myprogram` depends on `mylibrary1` which on turn depends on `mylibrary2`, then `mylibrary2` should come first:

```
g++ myprogram.o -lmylibrary2 -lmylibrary1 -o myprogram
```

And these are both wrong:

```
g++ myprogram.o -lmylibrary1 -lmylibrary2 -o myprogram  
g++ -lmylibrary2 -lmylibrary1 main.o -o myprogram
```

Undefined symbols in `main.o` are not searched in the given libraries.

Pros and cons of static libraries

Pros

- The resulting executable is self-contained, i.e., it contains all the instructions required for its execution.

Cons

- If an external library receives an update (such as improvements or bugfixes), the user has to relink its code against the new version.
- We cannot load symbols dynamically, on the base of decisions taken at run-time (it's an advanced stuff, we will deal with it later).
- The executable might become large.

Shared libraries

Shared libraries

With shared libraries, the mechanism by which code from the library is integrated into your own is very different than the static case.

- The **linker** ensures that symbols that are still unresolved are provided by the library.
- However, the corresponding code is not inserted, and the symbols remain unresolved.
- Instead, a reference to the library is stored in the executable for later use by the **loader** (or dynamic loader). This special program looks for the libraries and loads the code corresponding to the symbols that are still unresolved *at run-time*.

 **The linker and the loader are two different programs.**

Shared libraries: the linking phase

Versioning and naming schemes

Version vs. release

The *version* is an identifier typically represented by a sequence of numbers, indicating instances of a library with a common public interface and functionality. I recommend you to stick with the [Semantic Versioning](#) convention.

Naming scheme

- **Link name:** Used in the linking stage with the `-lmylib` option, of the form `libmylib.so`.
- **soname (Shared Object Name):** Looked after by the loader, typically formed by the link name followed by the major version number, e.g., `libmylib.so.3`.
- **Real name:** The actual file storing the library with the full version number, e.g.,
`libmylib.so.3.2.4`.

Library versions in action

The `ldd` command lists all shared libraries used by an executable (or another shared library):

```
ldd /usr/bin/octave-cli | grep fftw3.so  
libfftw3.so.3 => /lib/x86_64-linux-gnu/libfftw3.so.3 (...)
```

The loader searches for the library in special directories and finds `/lib/x86_64-linux-gnu/libfftw3.so.3`. This library is used when launching Octave.

If there's a new release, placing the corresponding file in the `/lib/x86_64-linux-gnu` directory, and resetting symbolic links, will make Octave use the new release without recompiling (and this is what happens when, for example, you upgrade a package via `apt` or similar).

Dependency management

```
$ ls -l /lib/x86_64-linux-gnu/libfftw3.so  
... /lib/x86_64-linux-gnu/libfftw3.so -> libfftw3.so.3.5.8
```

This means that `libfftw3.so.3` is a symbolic link to `libfftw3.so.3.5.8`. Hence, we are actually using version 3.5.8 of `libfftw3`.

Another nice thing about shared libraries is that they may depend on another shared library. This information can be encoded when creating the library. For instance:

```
ldd /usr/x86_64-linux-gnu/libumfpack.so  
...  
libblas.so.3 => /usr/lib/libblas.so.3
```

The UMFPACK library is linked against version 3 of the BLAS library. This helps to avoid using an incorrect version of dependent libraries.

Shared libraries: the linking phase (1/2)

You then proceed as usual:

```
g++ -I/path/to/mylib -c main.cpp  
g++ main.o -L/path/to/mylib -lmylib -o main
```

The linker looks for `libmylib.so` in system and/or in the specified directories, controls the symbols it provides, and verifies if the library contains a `soname`. If it doesn't, the link name `libmylib.so` is assumed to be also the `soname`.

For example, `libumfpack.so` provides a `soname` (of course, this has been taken care of by the library developers). If you wish, you can check it:

```
$ objdump -p /lib/x86_64-linux-gnu/libumfpack.so | grep SONAME  
SONAME          libumfpack.so.5
```

Shared libraries: the linking phase (2/2)

Being `libmylib.so` a shared library, the linker does not integrate the code of the resolved symbols into the executable. Instead, it just controls that the library provides the symbols and inserts the information about the soname of the library in the executable:

```
ldd main
libmylib.so.2 => /path/to/libmylib.so.2 (....)
```

In conclusion, linking a shared library is not more complicated than linking a static one. However, knowing what happens "under the hood" may be useful to tackle unexpected situations.

⚠ Even if the linker has found the library, it does not mean that the loader will find it as well!

Shared libraries: the loading phase

Where does the loader search for shared libraries?

The loader has a different search strategy with respect to the linker. It looks in `/lib`, `/usr/lib`, and in all the directories contained in `/etc/ld.conf` or in files with the extension `conf` contained in the `/etc/ld.conf.d/` directory.

If you want to permanently add a directory in the search path of the loader, you need to add it to `/etc/ld.conf` or add a conf file in the `/etc/ld.conf.d/` directory with the name of the directory and then launch `ldconfig`. This command rebuilds the database of the shared libraries and should be called every time one adds a new library (for example, `apt` does it for you, and moreover, `ldconfig` is launched at every boot of the computer).

Launching the command `sudo ldconfig -n directory` has the same effect, but in this case modifications will remain valid until the next restart of the computer.

⚠ All these operations require you to act as an administrator, for instance using the `sudo` command. Safer alternatives are in the next slide.

Alternative ways of directing the loader

1. Setting the environment variable `LD_LIBRARY_PATH` : It contains a colon-separated list of directory names where the loader will first look for libraries.

```
# Permanently, for the current terminal session:  
export LD_LIBRARY_PATH+=:dir1:dir2  
./main  
  
# Or, temporarily valid for a single command:  
LD_LIBRARY_PATH+=:dir1 ./main
```

2. With the special linker option, `-wl, -rpath, directory` : During the compilation (linking stage) of the executable, for instance

```
g++ main.cpp -wl, -rpath, /path/to/mylib -L/path/to/mylib -lmylib
```

The loader will look in `/path/to/mylib` before the standard directories. You can use also relative paths.

Shared libraries: how to build

How to build a shared library

1. Compile the source files:

```
g++ -fPIC -c mylib.cpp
```

PIC stands for Position-Independent Code.

2. Create the library:

```
g++ -shared -Wl,-soname,libmylib.so.1 -o libmylib.so.1.0 smalllib.o
```

Note: The library's real name is libmylib.so.1.0 .

3. Create symbolic links for version control:

```
ln -s libmylib.so.1.0 libmylib.so.1  
ln -s libmylib.so.1 libmylib.so
```

Linking the executable against the shared library

Compile the executable, linking the library:

```
g++ -I/path/to/mylib -c main.cpp  
g++ main.o -L/path/to/mylib -lmylib -o main
```

However, running the executable may result in an error:

```
./main error while loading shared libraries:  
  libmylib.so.1: cannot open shared object file: No such file or directory
```

To fix this, direct the loader as explained in the previous section, for instance by modifying `LD_LIBRARY_PATH` or changing the `rpath`:

```
g++ main.o -Wl,-rpath,/path/to/mylib -L/path/to/mylib -lmylib -o main
```

Now, the executable works as expected!

Releasing a new version

Assuming a new release (e.g., version 1.1), compile and link the new library without recompiling the executable:

```
g++ -c -fPIC mylib.cpp # mylib.cpp has some new features!
g++ -shared mylib.o -Wl,-soname,libmylib.so.1 -o libmylib.so.1.1
ln -s libmylib.so.1.1 libmylib.so.1
ln -s libmylib.so.1 libmylib.so
```

Now, running the executable uses the updated library without recompilation or relinking.

Note

For smaller projects without versioning, you can use the same name for link name, `soname`, and real name (e.g., `libmylib.so`). In this case, the `-Wl,soname` option can be omitted and the symbolic links are not needed.

Summary

- Object files should be compiled with the `-fPIC` option.
- The link name is used by the linker for matching symbols.
- The `soname` is used by the loader and is specified during library creation.
- Symbolic links can direct the loader to the desired library (useful for versioning).
- Use `-wl, -rpath` during linking or set `LD_LIBRARY_PATH` for directory search during development.

Shared libraries: dynamic loading

Dynamic loading and plugins

Shared libraries offer two intriguing features:

1. Dynamic loading of the library.
2. Dynamic loading of symbols from the library.

These features form the foundation for implementing *plugins* (and are also employed in Python modules).

Dynamic loading is a fundamental aspect of a plugin architecture, allowing an application to load parts of its implementation dynamically based on user requests.

⚠ This is a very advanced topic. For more information, have a look at [this interesting post](#) (source code [here](#)).

Pros and cons of shared libraries

Pros

1. Updating a library has an immediate effect on all codes linking against it. No recompilation or relinking is needed.
2. Executable is smaller since the code in the library is not duplicated.
3. We can load libraries and symbols runtime (*plugins*).

Cons

1. Executables depend on the library. If you delete the library, all codes using it won't run anymore.
2. Both the linking phase and **the loading phase** need careful management, especially when dealing with different library versions installed.

How to compile (and use) third-party libraries?

A (very) general guide

1. Obtain the library.
2. Read the documentation.
3. Compile the library.
4. Install the library, i.e., store header files and the generated (static or shared) library files into a convenient folder.
5. Integrate it in your project, i.e., include the folder containing header files and add proper link flags. In the case of shared libraries, don't forget to redirect the loader.

Looks easy, doesn't it? 😎

Actually, the crux lies in **step 3**:

- The library may have dependencies on other libraries. 😷
- Fortunately, some libraries use automatic build systems, simplifying the compilation process
... but forcing us to learn how to use these tools! 😅



Introduction to Makefile and CMake.

Exercise session 07

Introduction to GNU Make. Libraries: building and use.

Advanced Programming - SISSA, UniTS, 2023-2024

Pasquale Claudio Africa

16 Nov 2023

Introduction to GNU Make

Introduction

Prerequisites

Ensure the `make` program is installed by checking `make -version`. If not installed, use package managers such as `apt` on Unix or `Homebrew` on macOS. Additionally, have a C++ compiler like `g++` or `clang++` ready for compilation.

Getting Started

Let's start with a simple C++ program consisting of three files:

- `math.hpp`
- `math.cpp`
- `main.cpp`

Manual compilation

```
g++ -c -I. -std=c++17 -Wall -Wpedantic -Werror main.cpp math.cpp  
g++ -Wall -Wpedantic -Werror main.o math.o -o main
```

Alternatively, in a single line:

```
g++ -I. -Wall -Wpedantic -Werror main.cpp math.cpp -o main
```

This process involves creating object files and linking them to generate the executable. Now, let's simplify this with a Makefile.

Definitions

- In a Makefile, a **target** represents the desired output or action. It can be an executable, an object file, or a specific action like "clean."
- **Prerequisites** are files or conditions that a target depends on. If any of the prerequisites have been modified more recently than the target, or if the target does not exist, the associated recipe is executed.
- A **recipe** is a set of shell commands that are executed to build or update the target. Recipes follow the prerequisites and are indented with a **<TAB> character**. Each line in the recipe typically represents a separate command.

Creating a basic Makefile for C++

Putting it all together:

```
main: main.cpp math.cpp  
    g++ -I. -Wall -Wpedantic -Werror main.cpp math.cpp -o main
```

- **Target (`main`)**: The executable we want to create.
- **Prerequisites (`main.cpp math.cpp`)**: The source files required to build the target.
- **Recipe (`g++ [...] main.cpp math.cpp -o main`)**: The shell command to compile (`g++`) and link (`-o main`) the source files into the executable (`main`).

Variables for clarity

Enhance readability and maintainability by using variables:

```
CXX=g++
CPPFLAGS=-I.
CXXFLAGS=-std=c++17 -Wall -Wpedantic -Werror
```

```
main: main.cpp math.cpp
      $(CXX) $(CPPFLAGS) $(CXXFLAGS) main.cpp math.cpp -o main
```

- **CXX**: Compiler variable.
- **CPPFLAGS**: Preprocessor flags variable.
- **CXXFLAGS**: Compiler flags variable.

Automatic dependency generation

```
CXX=g++
CPPFLAGS=-I.
CXXFLAGS=-std=c++17 -Wall -Wpedantic -Werror
DEPS=math.hpp

%.o: %.cpp $(DEPS)
    $(CXX) -c $(CPPFLAGS) $(CXXFLAGS) $< -o $@

main: main.o math.o
    $(CXX) $(CXXFLAGS) $^ -o $@

clean:
    rm *.o main
```

- `%.o:` A generic rule for creating files with `.o` extension.
- `$@`, `$<`, `$^`: Automatic variables representing the target, the first prerequisite, and all prerequisites, respectively.

Phony targets

Define phony targets for non-file related tasks:

```
.PHONY: all clean
```

```
all: main
```

```
main: main.o math.o
```

```
    $(CXX) $(CXXFLAGS) $^ -o $<
```

```
clean:
```

```
    rm *.o main
```

- **.PHONY:** Marks targets that don't represent files.
- **all:** Default target.

Variables for source files

```
CXX=g++
CPPFLAGS=-I.
CXXFLAGS=-std=c++17 -Wall -Wpedantic -Werror
DEPS=math.hpp
SRC=$(wildcard *.cpp)
OBJ=$(SRC:.cpp=.o)
```

```
.PHONY: all clean
```

```
all: main
```

```
main: $(OBJ)
      $(CXX) $(CXXFLAGS) $^ -o $@
```

```
%.o: %.cpp $(DEPS)
      $(CXX) -c $(CXXFLAGS) $< -o $@
```

```
clean:
      rm *.o main
```

Building a library and linking against it

Suppose we have a simple C++ library with two files

- `math.hpp`
- `math.cpp`

Additionally, we have a program, `main.cpp`, that uses functions from this library.

Now, let's create a Makefile to build the library and another one to link our program against it.

Makefile to build a library (1/2)

```
CXX=g++
CPPFLAGS=-I.
CXXFLAGS=-std=c++17 -Wall -Wpedantic -Werror

SRC=math.cpp
OBJ=$(SRC:.cpp=.o)
OBJ_fPIC=$(SRC:.cpp=.shared.o)
DEPS=math.hpp

LIB_NAME_STATIC=libmath.a
LIB_NAME_SHARED=libmath.so

all: static shared

static: $(LIB_NAME_STATIC)
shared: $(LIB_NAME_SHARED)
```

Makefile to build a library (2/2)

```
$(LIB_NAME_STATIC): $(OBJ)
    ar rcs $@ $^

$(LIB_NAME_SHARED): $(OBJ_fPIC)
    g++ $(CXXFLAGS) -shared $^ -o $@

%.shared.o: %.cpp $(DEPS)
    $(CXX) -c -fPIC $(CPPFLAGS) $(CXXFLAGS) $< -o $@

%.o: %.cpp $(DEPS)
    $(CXX) -c $(CPPFLAGS) $(CXXFLAGS) $< -o $@

clean:
    rm -f *.o $(LIB_NAME_STATIC) $(LIB_NAME_SHARED)
```

Makefile to link against a library

```
CXX=g++
CPPFLAGS=-Imath/
CXXFLAGS=-std=c++17 -Wall -Wpedantic -Werror
LDFLAGS=-Wl,-rpath,math/ -Lmath/ # For dynamic linking.
# LDFLAGS=-Lmath/ -static          # For static linking.
LDLIBS=-lmath

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

all: main

main: $(OBJ)
    $(CXX) $(CXXFLAGS) $^ $(LDFLAGS) $(LDLIBS) -o $@

%.o: %.cpp
    $(CXX) -c $< $(CPPFLAGS) $(CXXFLAGS) -o $@

clean:
    rm -f *.o main
```

Summary (1/2)

`make` efficiently determines the need to regenerate a target by checking its existence and the up-to-dateness of prerequisite files. This feature enables it to avoid unnecessary target regeneration.

Make simplifies the installation of numerous libraries through a concise set of commands. A typical sequence for installing an open-source library involves using the following commands:

```
make  
make install
```

Typically, the `make` command builds the library, while `make install` copies the library's headers, the libraries and the binaries to a user-specified folder, which defaults to the `/usr` or `/usr/local` directory. This streamlined process facilitates the integration of the installed library into your source code.

Summary (2/2)

In some circumstances, the build process can be optimized by employing the `make -jN` command, where `N` represents the number of parallel jobs or commands executed concurrently.

Despite its advantages, Makefiles are platform-dependent, necessitating adaptation to different operating systems. To address this issue, we will explore **CMake** as a potential solution, providing a platform-independent alternative for managing and generating build systems.

References

- [A simple makefile tutorial](#) : Essential tutorial on `make` and Makefile.
- [Makefile tutorial](#) : A GitHub repository with numerous makefile examples.
- [GNU make](#) : Official documentation for `make` and Makefile.

Exercise 1: building and using muParserX

- Download and extract `muParserX`:

```
wget https://github.com/beltoforion/muparserx/archive/refs/tags/v4.0.12.tar.gz
```

- The source files of `muParserX` are located inside the `muparserx-4.0.12/parser/` folder.
- In that folder, write a `Makefile` to compile `muParserX` into a shared library.
- Write a Makefile that compiles and links the program in `hints/ex1.cpp` with `muParserX`.

Exercise 2: shared libraries

The `hints/ex2/` directory contains a library that implements a **gradient descent algorithm for linear regression**, accompanied by a source file `ex2.cpp` utilizing this library.

Unfortunately, the gradient descent code within the library contains a bug.

Your tasks are:

1. Compile the library and test file, using the provided Makefiles.
2. Inspect the code to locate the bug within the gradient descent algorithm.
3. Once the bug is identified, fix it within the code. Then, compile an updated version of the library, incorporating the bug fix.
4. Execute the test case to verify that the bug fix successfully addresses the issue. Please note that, since we are dealing with a shared library, this verification should be conducted **without** the need for recompilation or relinking of the test file.

Exercise 3: order matters

The `hints/ex3/` directory contains a source file `ex3.cpp` that uses a library `graphics_lib`, which depends on another library `math_lib`.

1. Generate a static library `libmath.a`.
2. Generate a static library `libgraphics.a`.
3. Compile `ex3.cpp` into an object file `ex3.o`.
4. Link `main.o` against `libmath.a` and `libgraphics.a` to produce the final executable.

What is the correct order for passing `ex3.o`, `libmath.a`, and `libgraphics.a` to the linker to successfully resolve all the symbols?

Would the same considerations apply if dynamic linking (shared libraries) were used instead of static linking?

Exercise 4: dynamic loading

This exercise showcases dynamic loading, the building block for implementing a *plugin* system.

The `hints/ex4/` contains a module `functions` containing the definition of three mathematical functions. The source file `functions.cpp` gets compiled into a shared library `libfunctions.so`, using C *linkage* to prevent *name mangling*.

Notably, when compiling the source file `ex4.cpp` into an executable, there is no need to link against `libfunctions.so`.

1. Fill in the missing parts in `ex4.cpp` to dynamically load the library.
2. Prompt the user for the function name to evaluate at a given point, selecting from the ones available in the library.
2. Perform the evaluation and print the result.
3. Release the library.

Lecture 09

**Introduction to CMake.
Optimization, debugging, profiling, testing.**

Advanced Programming - SISSA, UniTS, 2023-2024

Pasquale Claudio Africa

21 Nov 2023

Outline

1. Introduction to CMake
2. Optimization
3. Debugging
4. Profiling
5. Testing

Introduction to CMake

Build systems

Purposes

Build systems are a way to deploy software.

They are used to:

1. Provide others a way to configure **your** own project.
2. Configure and install third-party software on your system.

Configure means:

- Meet dependencies
- Build
- Test

Build systems available

- **CMake**
 - **Pros:** Easy to learn, great support for multiple IDEs, cross-platform
 - **Cons:** Does not perform automatic compilation test for met dependencies.
- **GNU Autotools**
 - **Pros:** Excellent support for legacy Unix platforms, large selection of existing modules.
 - **Cons:** Slow, hard to use correctly, painful to debug, poor support for non-Unix platforms.
- **Meson** , **Bazel** , **SCons** , ...

Package managers:

- **Conan** , **vcpkg**

Let's try

Install dependencies, then compile and install.

Doxygen (CMake)

```
cd /path/to/doxygen/src/  
mkdir build && cd build  
cmake -DCMAKE_INSTALL_PREFIX=/opt/doxygen ../  
make -j<N>  
(sudo) make install
```

GNU Scientific Library (autotools)

```
cd /path/to/gsl/src/  
./configure --prefix=/opt/gsl --enable-shared --disable-static  
make -j<N>  
(sudo) make install
```

Why CMake?

- More packages use CMake than any other system
- Almost every IDE supports CMake (or vice-versa)
- Really cross-platform, no better choices for Windows
- Extensible, modular design

Who else is using CMake?

- Netflix
- HDF Group, ITK, VTK, Paraview (visualization tools)
- Armadillo, CGAL, LAPACK, Trilinos (linear algebra and algorithms)
- deal.II, Gmsh (FEM analysis)
- KDE, Qt, ReactOS (user interfaces and operating systems)
- ...

CMake basics

The root of a project using CMake must contain a **CMakeLists.txt** file.

```
cmake_minimum_required(VERSION 3.12)

# This is a comment.
project(MyProject VERSION 1.0
        DESCRIPTION "A very nice project"
        LANGUAGES CXX)
```

Please use a CMake version more recent than your compiler (at least ≥ 3.0).

Command names are **case-insensitive**.

CMake 101

Configure

```
cmake -S /path/to/src/ -B build [options...]  
# Or:  
# mkdir build && cd build  
# cmake /path/to/src/ [options...]
```

Compile

```
cd /path/to/build/  
make -j<N>
```

List variable values

```
cd /path/to/build  
cmake /path/to/src/ -L
```

Targets

CMake is all about targets and properties. An executable is a target, a library is a target. Your application is built as a collection of targets depending on each other.

```
# Header files are optional.  
add_executable(my_exec my_main.cpp my_header.h)  
  
# Options are STATIC, SHARED (dynamic) or MODULE (plugins).  
add_library(my_lib STATIC my_class.cpp my_class.h)
```

Target properties

Targets can be associated with various **properties** :

```
add_library(my_lib STATIC my_class.cpp my_class.h)
target_include_directories(my_lib PUBLIC include_dir)
# "PUBLIC" propagates the property to
# other targets depending on "my_lib".
target_link_libraries(my_lib PUBLIC another_lib)

add_executable(my_exec my_main.cpp my_header.h)
target_link_libraries(my_exec my_lib)
target_compile_features(my_exec cxx_std_20)
# Last command is equivalent to
# set_target_properties(my_exec PROPERTIES CXX_STANDARD 20)
```

Interacting with the outside world: local variables

```
set(LIB_NAME "my_lib")

# List items are space- or semicolon-separated.
set(SRCS "my_class.cpp;my_main.cpp")
set(INCLUDE_DIRS "include_one;include_two")

add_library(${LIB_NAME} STATIC ${SRCS} my_class.h)
target_include_directories(${LIB_NAME} PUBLIC ${INCLUDE_DIRS})

add_executable(my_exec my_main.cpp my_header.h)
target_link_libraries(my_exec ${LIB_NAME})
```

Interacting with the outside world: cache variables

Cache variables are used to interact with the command line:

```
# "VALUE" is just the default value.  
set(MY_CACHE_VARIABLE "VALUE" CACHE STRING "Description")  
  
# Boolean specialization.  
option(MY_OPTION "This is settable from the command line" OFF)
```

Then:

```
cmake /path/to/src/ \  
-DMY_CACHE_VARIABLE="SOME_CUSTOM_VALUE" \  
-DMY_OPTION=OFF
```

Useful variables

- **CMAKE_SOURCE_DIR**: top-level source directory
- **CMAKE_BINARY_DIR**: top-level build directory

If the project is organized in sub-folders:

- **CMAKE_CURRENT_SOURCE_DIR**: current source directory being processed
- **CMAKE_CURRENT_BINARY_DIR**: current build directory

```
# Options are "Release", "Debug",
# "RelWithDebInfo", "MinSizeRel"
set(CMAKE_BUILD_TYPE Release)

set(CMAKE_CXX_COMPILER "/path/to/c++/compiler")
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -Wall")
set(CMAKE_LIBRARY_OUTPUT_DIRECTORY lib)
```

Environment variables

```
# Read.  
message("PATH is set to: $ENV{PATH}")  
  
# Write.  
set(ENV{variable_name} value)
```

(although it is generally a good idea to avoid them).

Control flow

```
if("${variable}")
    # True if variable is not false-like
else()
    # Note that undefined variables would be `""` thus false
endif()
```

The following operators can be used.

Unary: NOT , TARGET , EXISTS (file), DEFINED , etc.

Binary: STREQUAL , AND , OR , MATCHES (regular expression), ...

Parentheses can be used to group.

Branch selection

Useful for switching among different implementations or versions of any third-party library.

```
#ifdef USE_ARRAY
    std::array<double, 100> my_array;
#else
    std::vector<double> my_array;
#endif
```

How to select the correct branch?

Pre-processor flags

```
target_compile_definitions(my_exec PRIVATE USE_ARRAY=1)
```

Or let the user set the desired flag:

```
option(WITH_ARRAY "Use std::array instead of std::vector" ON)  
  
if(WITH_ARRAY)  
    target_compile_definitions(my_exec PRIVATE USE_ARRAY=1)  
endif()
```

Modify files depending on variables

`print_version.hpp.in`:

```
void print_version() {
    std::cout << "Version number: " << @MY_PROJECT_VERSION@
                  << std::endl;
}
```

`CMakeLists.txt`:

```
set(MY_PROJECT_VERSION 1.2.0)

configure_file(
    "${CMAKE_CURRENT_SOURCE_DIR}/print_version.hpp.in"
    "${CMAKE_CURRENT_BINARY_DIR}/print_version.hpp")
```

See also: `#cmakedefine`.

Print messages and debug

Content of variables is printed with

```
message("MY_VAR is: ${MY_VAR}")
```

Error messages can be printed with

```
message(FATAL_ERROR "MY_VAR has the wrong value: ${MY_VAR}")
```

Commands being executed are printed with

```
cmake /path/to/src/ -B build --trace-source=CMakeLists.txt  
make VERBOSE=1
```

Looking for third-party libraries

CMake looks for **module files** `FindPackage.cmake` in the directories specified in `CMAKE_PREFIX_PATH`.

```
set(CMAKE_PREFIX_PATH "${CMAKE_PREFIX_PATH} /path/to/module/")

# Specify "REQUIRED" if the library is mandatory.
find_package(Boost 1.50 COMPONENTS filesystem graph)
```

If the library is not located in a system folder, often a hint can be provided:

```
cmake /path/to/src/ -DBOOST_ROOT=/path/to/boost
```

Using third-party libraries

Once the library is found, proper variables are populated.

```
if(${Boost_FOUND})
    target_include_directories(my_lib PUBLIC
                                ${Boost_INCLUDE_DIRS})

    target_link_directories(my_lib PUBLIC
                                ${Boost_LIBRARY_DIRS})
    # Old CMake versions:
    # link_directories(${Boost_LIBRARY_DIRS})

    target_link_libraries(my_lib ${Boost_LIBRARIES})
endif()
```

Compilation test

CMake can try to compile a source and save the exit status in a local variable.

```
try_compile(  
    HAVE_ZIP  
    "${CMAKE_BINARY_DIR}/temp"  
    "${CMAKE_SOURCE_DIR}/tests/test_zip.cpp"  
    LINK_LIBRARIES ${ZIP_LIBRARY}  
    CMAKE_FLAGS  
        "-DINCLUDE_DIRECTORIES=${ZIP_INCLUDE_PATH}"  
        "-DLINK_DIRECTORIES=${ZIP_LIB_PATH}")
```

See also: [try_run](#).

Execution test

CMake can run specific executables and check their exit status to determine (un)successful runs.

```
include(CTest)
enable_testing()
add_test(NAME MyTest COMMAND my_test_executable)
```

Organize a large project

```
cmake_minimum_required(VERSION 3.12)
project(ExampleProject VERSION 1.0 LANGUAGES CXX)

find_package(...)

add_subdirectory(src)
add_subdirectory(apps)
add_subdirectory(tests)
```

Tip: how to organize a large project

```
project/
├── apps/
│   ├── CMakeLists.txt
│   └── my_app.cpp
├── cmake/
│   └── FindSomeLib.cmake
├── doc/
│   └── Doxyfile.in
├── scripts/
│   └── do_something.sh
└── src/
    ├── CMakeLists.txt
    └── my_lib.{hpp,cpp}
└── tests/
    ├── CMakeLists.txt
    └── my_test.cpp
├── .gitignore
├── CMakeLists.txt
└── {LICENSE.md, README.md}
```

Further readings

- Official documentation
- Modern CMake
- It's time to do CMake right
- Effective modern CMake
- More modern CMake

Optimization

Code optimization

Code optimization is the process of enhancing a program's performance, efficiency, and resource utilization without changing its functionality. It involves improving execution speed, reducing memory usage, and enhancing overall system responsiveness.

Optimization techniques

- **Compiler optimizations:** Utilize compiler features to automatically enhance code during compilation.
- **Algorithmic optimization:** Improve the efficiency of algorithms and data structures.
- **Manual refactoring:** Restructure code for better readability, maintainability, and performance.
- **Profiling and analysis:** Use profiling tools to identify and optimize performance bottlenecks.

Optimization options

The compiler enhances performance by optimizing CPU register usage, expression refactoring, and pre-computing constants.

- Disable optimization during debugging.
- Pass the `-O{n}` (`{n=0, 1, 2, s, or 3}`) flag to the compiler to control optimization level, with `-Os` for space optimization and `-O3` for maximum optimization. [Here](#) a detailed list of optimizations enabled with each flag.
- Defining the `-DNDEBUG` preprocessor variable, standard assertion are ignored, resulting in faster code.

Loop unrolling

It is beneficial to unroll small loops. For example, transform:

```
for (int i = 0; i < n; ++i)
    for(int k = 0; k < 3; ++k)
        a[k] += b[k] * c[i];
```

to:

```
for (int i = 0; i < n; ++i) {
    a[0] += b[0] * c[i];
    a[1] += b[1] * c[i];
    a[2] += b[2] * c[i];
}
```

Compiler may unroll loops with `-funroll-loops`, but better performance isn't guaranteed.

Prefetching constant values

Prefetch constant values inside the loop for further optimization:

```
for (int i = 0; i < n; ++i) {  
    auto x = c[i];  
    a[0] += b[0] * x;  
    a[1] += b[1] * x;  
    a[2] += b[2] * x;  
}
```

Avoid `if` inside nested loops

`if` statements, especially in nested loops, can be costly. Consider these improvements:

```
for(int i = 0; i < 10000; ++i) {
    for (int j = 1; j < 10; ++j) {
        if(c[i] > 0)
            a[i][j] = 0;
        else
            a[i][j] = 1;
    }
}
// Better:
for(int i = 0; i < 10000; ++i)
    if(c[i] > 0)
        for(int j = 0; j < 10; ++j) {
            a[i][j] = 0;
        else
            for(int j = 0; j < 10; ++j) {
                a[i][j] = 1;
    }
```

Sum of a vector: two strategies compared

```
double test1(double *data, const size_t count) {
    double sum(0);
    for (size_t j = 0; j < count; ++j)
        sum += data[j];
    return sum;
}

double test2(double *data, const size_t count) {
    double sum(0), sum1(0), sum2(0), sum3(0);
    size_t j;
    for (j = 0; j < (count - 3); j += 4) {
        sum += data[j + 0];
        sum1 += data[j + 1];
        sum2 += data[j + 2];
        sum3 += data[j + 3];
    }
    for (; j < count; ++j)
        sum += data[j];
    sum += sum1 + sum2 + sum3;
    return sum;
}
```

Compiled without optimization, which is faster? `test1` or `test2`?

The answer is not straightforward: it depends on the computer's architecture.

On my laptop (8th Gen Intel(R) Core(TM) i7-8565U CPU @ 1.80GHz), **test2 is approximately 5 times faster than test1** with `count = 1e9`!

Why? The Streaming SIMD Extensions (SSE2) instruction set of the CPU allows for parallelization at the microcode level. It's a super-scalar architecture with multiple instruction pipelines to execute several instructions concurrently during a clock cycle. The code of `test2` better exploits this capability.

Lessons learned: Counting operations doesn't necessarily reflect performance. Compiler optimizers can transform `test1` into `test2` automatically. Sometimes, giving it a hand is beneficial.

Cache friendliness

Efficiency often depends on how variables are accessed in memory. Access variables contiguously for cache pre-fetching effectiveness. For example, if `mat` is a dynamic matrix organized **row-wise**:

```
for (i = 0; i < n_rows; ++i)
    for (j = 0; j < n_cols; ++j)
        a += mat(i, j);
```

is cache-friendly. While

```
for (j = 0; j < n_cols; ++j)
    for (i = 0; i < n_rows; ++i)
        a += mat(i, j);
```

is not, and thus less efficient.

Debugging

Static analysis

Static analysis tools analyze source code by inspecting it for potential issues, vulnerabilities, or adherence to coding standards. Common ones include:

- `cppcheck`
- `cpplint`
- `clang-tidy`

Some of the checks they perform:

- Automatic variable checking.
- Bounds checking for array overruns.
- Unused functions, variable initialization and memory duplication.
- Invalid usage of Standard Template Library functions and idioms.
- Missing `#include`s.
- Memory or resource leaks, performance errors.

C++ interpreters and explorers

- [Cling](#) is an interactive C++ interpreter, built on LLVM and Clang. It's part of the ROOT project at CERN and can be integrated into a Jupyter workspace ([see here](#)). While experimental, an interpreter aids in code prototyping.
- [Compiler Explorer](#) to check how code translates into assembly language.
- [C++ Insights](#) allows viewing source code through a compiler's eyes.

Debuggers

Debuggers are software tools that enable developers to inspect, analyze, and troubleshoot code during the development process. They provide a set of features for identifying and fixing errors in programs.

Key features

- **Breakpoints:** Pauses program execution at specified points to inspect variables and code.
- **Variable inspection:** Allows developers to examine the values of variables during runtime.
- **Step-through execution:** Enables line-by-line execution for precise debugging.
- **Call stack analysis:** Displays the sequence of function calls leading to the current point in code.

Debugging

During code development, debugging allows step-by-step execution. To use a debugger, compile with `-g` (which implies no optimization). `-g` adds information for locating source lines in machine code.

Two debugging types:

- **Static debugging:** Analyze core dump if code aborts.
- **Dynamic debugging:** Execute through a debugger, breaking at points to examine variables.

Two common debuggers are `gdb` and `lldb`.

Debugging levels

Debugging can be at different levels, and using `-g -O` together is allowed. `-g` tells the compiler to provide extra information for the debugger. However, line-by-line debugging reliability decreases with optimization. `-g` implies `-O0` by default.

Debugging levels and special optimization options linked to debugging:

- `-g0` : No debugging information.
- `-g1` : Minimal information for backtraces.
- **The default debugging level is 2.**
- `-g3` : Extra information, including macro definitions.
- `-Og` : Special optimization option. Enables optimizations without interfering with debugging.

Main commands of the debugger

- `run` : Run the program.
- `break` : Set a breakpoint at a line/function.
- `where` : Show location and backtrace.
- `print` : Display variable/expression value.
- `list n` : Show lines around line n.
- `next` : Go to the next instruction, proceeding through subroutines.
- `step` : Go to the next instruction, entering called functions.
- `cont` : Continue executing.
- `backtrace` : Print memory stack after program aborts.
- `quit` : Exit the debugger.
- `help` : Displays help information.

Other debugging tools

`valgrind`, a suite of tools for debugging and profiling. It can find memory leaks, unassigned variables, or check memory usage:

Find memory leaks:

```
valgrind --tool=memcheck --leak-check=yes --log-file=file.log executable
```

Check memory usage:

```
valgrind --tool=massif --massif-out-file=massif.out --demangle=yes executable  
ms_print massif.out > massif.txt
```

`massif.txt` indicates memory usage during the program execution.

Profiling

Profilers

A **profiler** in software development is a tool or set of tools designed to analyze the runtime behavior and performance of a computer program. It provides detailed information about resource utilization, execution times, and function calls during the program's execution.

Key objectives

- **Performance analysis:** Profilers offer insights into how much time a program spends in different functions, helping identify performance bottlenecks.
- **Resource usage:** They measure memory consumption, CPU utilization, and other system resources, aiding in optimizing resource-intensive operations.
- **Function call tracing:** Profilers track the sequence of function calls, enabling developers to understand the flow of execution.

gprof

`gprof` is the GCC simple profiler. In order to use it, compile the code with the `-pg` option at both the compilation and linking stages.

When executing the code, it generates a file called `gmon.out`, which is then utilized by the profiler:

```
gprof --demangle executable > file.txt
```

Then `file.txt` will contain valuable information about the program execution.

Main options of gprof

`gprof` offers a range of options. The main ones are:

- `--annotated-source[=symspec]` : Prints annotated source code. If `symspec` is specified, print output only for matching symbols.
- `-I dirs` : List of directories to search for source files.
- `--graph[=symspec]` : Prints the call graph analysis.
- `--demangle` : Demangles mangled names (essential for C++ programs).
- `--display-unused-functions` : As the name says.
- `--line` : Line-by-line profiling (but maybe better use gcov).

callgrind

`callgrind` is a tool of `valgrind` that you may call, for instance, as:

```
valgrind --tool=callgrind --callgrind-out-file=grind.out --dump-line=yes ./myprog
```

Compile the program with `-g` and **optimization activated**. The option `--dump-lines` is used for line-by-line profiling.

Afterward, post-process the binary file `grind.out`, e.g., using `kcacheGrind`:

```
kcacheGrind grind.out
```

It opens a graphical interface.

Other profilers

There are alternative profilers, some useful in a parallel environment:

- `perf` : Lightweight CPU profiling.
- `gperf-tools` : Formerly Google Performance Tools.
- `The TAU performance system` : Profiling and tracing toolkit for parallel programs.
- `Scalasca` : Performance analysis for parallel applications on distributed memory systems.

Testing

Verification vs. validation

Verification: Ensuring correct implementation

Validation: Confirming desired behavior

- **Verification:** Conducted during development, tests **individual components** separately. Specific tests demonstrate correct functionality, covering the code and checking for memory leaks.
- **Validation:** Performed on the **final code**. Assesses if the code produces the intended outcome—convergence, reasonable results, and expected computational complexity.

Types of testing

- **Unit testing:** Testing individual components (functions, methods, or classes) to ensure each behaves as expected. It focuses on a specific piece of code in isolation.
- **Integration testing:** Verifying that different components/modules of the software work together as intended. It deals with interactions between different parts of the system.
- **Regression testing:** Ensuring recent code changes do not adversely affect existing functionalities. It involves re-running previous tests on the modified codebase to catch unintended side effects.

Importance of testing

- **Early detection of bugs:** Testing allows early detection and fixing of bugs, reducing the cost and time required for debugging later in the development process.
- **Code reliability:** Testing ensures the code behaves as expected and provides reliable results under different conditions.
- **Documentation:** Test cases serve as documentation for how different parts of the code are expected to work. They help other developers understand the intended behavior of functions and classes.

Unit testing in C++

In C++, unit testing often uses frameworks like [Google Test](#), [Catch2](#), or [CTest](#) from the CMake ecosystem.

Here's a simple example using `gtest`:

```
#include "mylibrary.h"
#include "gtest/gtest.h"

TEST(MyLibrary, AddTwoNumbers) {
    EXPECT_EQ(add(2, 3), 5);
}

int main(int argc, char **argv) {
    ::testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

In this example, we test the `add` function from the `mylibrary` module.

Test-Driven Development (TDD)

TDD is a software development approach where tests are written before the actual code. The cycle is writing a test, implementing the code to pass the test, and refactoring.

Advantages

TDD encourages modular and testable code, ensuring all parts of the codebase are covered by tests. It also starts by thinking at how code should be used (bottom-up strategy), possibly guiding the design of the interface exposed.

Process

1. Write a test defining a function or improvements succinctly.
2. Run the test to ensure it fails, showing it doesn't pass.
3. Write the simplest code to make the test pass.
4. Run the test and verify it passes.
5. Refactor the code for better structure or performance.

Continuous Integration (CI) and testing

- **CI:** Frequently integrating code changes into a shared repository. Automated builds and tests ensure new changes don't break existing functionalities.
- **Benefits:**
 - Early detection of integration issues.
 - Regular validation of code against the test suite.
 - Confidence in the stability of the codebase.
- **Popular CI Tools:**
 - Jenkins
 - Travis CI
 - GitHub Actions
 - GitLab CI

Coverage

Code coverage is a metric used in software testing to measure the extent to which source code is executed during the testing process. It provides insights into which parts of the codebase have been exercised by the test suite and which parts remain untested.

Key concepts

- **Lines of code:** Code coverage is often expressed as a percentage of lines of code that have been executed by tests. The goal is to have as close to 100% coverage as possible.
- **Branches and paths:** In addition to lines, code coverage can also consider branches and execution paths within the code. This provides a more detailed analysis of the code's behavior.

Coverage with gcov

gcc supports program coverage with gcov . Compile with -g -fprofile-arcs -ftest-coverage and no optimization. For shared objects with dlopen , add the option -Wl,--dynamic-list-data .

Run the code, producing gcda and gcno files. Use gcov utility:

```
gcov [options] source_file_to_examine [or executable]
```

Text files with code and execution counts for each line are created.

Main Options of gcov

gcov offers various options:

- --demangled-names : Demangle names, useful for C++.
- --function-summaries : Output summaries for each function.
- --branch-probabilities : Write branch frequencies to the output file.

lcov and genhtml: nice graphical tools for gcov

The `gcov` output is verbose. With `lcov` and `genhtml`, you get a graphical view:

Compile with `gcov` rules, then:

```
lcov --capture --directory project_dir --output-file cov.info  
genhtml cov.info --output-directory html
```

`project_dir` is the directory with `gcda` and `gcno` files. In the `html` directory, open `index.html` in your browser.



Introduction to Python.

Exercise session 08

**Introduction to CMake.
Optimization, debugging, profiling, testing.**

Advanced Programming - SISSA, UniTS, 2023-2024

Pasquale Claudio Africa

23 Nov 2023

CMake

Exercise 1: CMake

1. Following `exercises/07/solutions/ex1`, compile `muParserX` using CMake and write a `CMake` script to compile and link the test code `ex1.cpp` against it.
2. Re-do `exercises/07/solutions/ex3` with the help of CMake.

Optimization and profiling

Source

Memory layout

Data structure alignment

```
class MyClass
{
    char a;      // 1 byte.
    short int b; // 2 bytes.
    int c;       // 4 bytes.
    char d;      // 1 byte.
};
```

How data is *not* stored

How data is actually stored

Access patterns and loop tiling (for a row-major matrix)

Examples

The folder `examples/optimization` contains three examples:

1. `data_alignment` compares the memory occupation of two objects containing the same data members but with different data alignment/padding.
2. `loop_unrolling` implements a function that multiplies all elements in a `std::vector` by looping over all its elements and returns the result. The executable compares the performance with those obtained exploiting loop unrolling.
3. `static` implements a function that allocates a `std::vector` and, taking an index as input, returns the corresponding value. The executable compares the performance with those obtained by declaring the vector `static`.

Exercise 2: Optimization

The `hints/ex2/` directory contains the implementation of a class for dense matrices organized as **column-major**.

- Implement `Matrix::transpose()`, a method to compute $A = A^T$.
- Implement `operator*`, a function to compute matrix-matrix multiplication.
- Optimize the matrix-matrix multiplication by transposing the first factor before the computation. Compare the execution speed with the previous implementation.
- Use `valgrind --tool=callgrind` to generate a profiler report.
- Generate a coverage report using `lcov` and `genhtml`.

Debugging

Examples

The content of `examples/debug` was inspired by [this repository](#) and shows basic techniques for debugging as well as an introduction to `gdb`.

Further readings

- [Defensive programming and debugging](#) .
- [Cpp Undefined Behaviour 101](#)
- [Shocking Undefined Behaviour In Action](#)

Exercise 3: Debugging

The `hints/ex3/` directory contains an implementation of a double-linked list class. The class stores a pointer to the head, and each node (except for the head and the tail, obviously) contains a pointer to the previous and to the next node.

The implementation contains a lot of errors, namely:

1. Compilation and syntax errors.
2. Runtime errors, including a segmentation fault and a problem in printing the list.
3. Memory leaks.
4. Two possible *segmentation faults*, not captured by the `main`.

With the help of `gdb` and `valgrind`, solve all these issues and make the code working!

Testing

Exercise 4: Testing

The `hints/ex4/` contains a static function to compute the mean of a `std::vector`.

Following the given directory structure and using [Google Test](#), fill in the missing parts in `tests/mean.cpp` to check that the function behaves as expected in all the listed cases.

To run the testsuite type

```
make test
```

or

```
ctest
```

from the CMake build folder.

Exercise session 09

Introduction to Python. Built-in data types. Variables, lists, tuples, dictionaries, sets. Control structures. Functions. Docstrings.

Advanced Programming - SISSA, UniTS, 2023-2024

Pasquale Claudio Africa

30 Nov 2023

Exercise 1: numerical types, strings

1. What is 5 to the power of 5?
2. What is the remainder from dividing 73 by 6?
3. How many times does the whole number 3 go into 123? What is the remainder of dividing 123 by 3?
4. Split the following string into a list by splitting on the `#` character:

```
s = "apple#banana#cherry#orange"
```

5. Use string methods to extract the website domain from an email, e.g., from the string `"pafrica@fakemail.com"`, you should extract `"fakemail"`.
6. Given the following variables:

```
thing = "light"
speed = 299792458 # m/s
```

Use f-strings to print `"The speed of light is 2.997925e+08 m/s."`

Exercise 1: lists, dictionaries, tuples

7. Given this nested list, use indexing to grab the word "AdvProg" :

```
l = [10, [3, 4], [5, [100, 200, ["AdvProg"]]], 23, 11], 1, 7]
```

8. Given this nested dictionary grab the word "AdvProg" :

```
d = {  
    "outer": [  
        1,  
        2,  
        3,  
        {"inner": ["this", "is", "inception", {"inner_inner": [1, 2, 3, "AdvProg"]}]},  
    ]  
}
```

9. Why does the following cell return an error?

```
t = (1, 2, 3, 4, 5)  
t[-1] = 6
```

Exercise 1: if-else

10. Given the variable `language` which contains a string, use `if-elif-else` to write a program that prints

- "I love snakes!" if `language` is "Python" (any kind of capitalization).
- "Are you a pirate?" if `language` is "C++" (any kind of capitalization).
- "What is `language`?" if `language` is anything else.

Exercise 2: functions, comprehension

1. Create a function `website()` that grabs the website domain from a URL string. For example, if your function is passed `"www.google.com"`, it should return `"google"`.
2. Create a function `divisible(a, b)` that accepts two integers (`a` and `b`) and returns `True` if `a` is divisible by `b` without a remainder. For example, `divisible(10, 3)` should return `False`, while `divisible(6, 3)` should return `True`.
3. Use list comprehension to square every number in the following list of numbers:
`l = [1, 2, 3, 4, 5, 6, 7, 8, 9].`
4. For the following list of names, write a list comprehension that creates a list of *only* words that start with a capital letter. Hint: `str.isupper()`.

```
names = ['Steve Irwin', 'koala', 'kangaroo', 'Australia', 'Sydney', 'desert']
```

Exercise 2: comprehension, exceptions

5. For the following list of `keys` and `vals` use dictionary comprehension to create a dictionary of the form `{'key-0': 0, 'key-1': 1, etc}`. Hint: `zip()` can help you combine two lists into one object to be used for comprehension/looping).

```
keys = [f"key-{k}" for k in range(10)]  
vals = range(10)
```

6. Write a `try / except` to catch the error generated from the following code and print "I caught you!". Make sure you catch the specific error being caused, this is typically better practice than just catching all errors!

```
5 / 0
```

Exercise 2: generators, loops, comprehension

7. This question is a little harder. Create a generator function called `listgen(n)` that yields numbers from 0 to n, in batches of lists of maximum 10 numbers at a time. For example, your function should behave as follows:

```
g = listgen(100)
next(g)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
next(g)
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
next(g)
[20, 21, 22, 23, 24, 25, 26, 27, 28, 29]
```

8. Given this 3x4 matrix implemented as a list of lists, write a list comprehension that creates its transpose:

```
matrix = [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
```

Exercise 2: functions

9. Create a function `lucky_sum(*args)` that takes all the integers a user enters and returns their sum. However, if one of the values is 13 then it does not count towards the sum, nor do any values to its right. For example, your function should behave as follows:

```
lucky_sum(1, 2, 3, 4)  
10
```

```
lucky_sum(1, 13, 3, 4)  
1
```

```
lucky_sum(13)  
0
```

This example is inspired by the related [CodingBat challenge](#).

Exercise 2: loops, comprehension

10. See below code. Print only the EEG signal and events separately from this nested list using
(1) for loop, (2) list comprehension.

```
two_channel_eeg_signal1 = [8, 9]
event1 = 1

two_channel_eeg_signal2 = [3, 3]
event2 = 2

two_channel_eeg_signal3 = [2, 3]
event3 = 2

nested_list = []
nested_list.append([two_channel_eeg_signal1, event1])
nested_list.append([two_channel_eeg_signal2, event2])
nested_list.append([two_channel_eeg_signal3, event3])
print("EEG signal: ", some_nested_list)
```

Exercise 3: sets, functions

Let A and B be sets. The set $(A \setminus B) \cup (B \setminus A)$ is called the symmetric difference of the two sets.

1. Write a function that performs this operation.
2. Use docstrings to document the function and check the documentation invoking the `help()` function.
3. Compare your results to the output of the command:

A.`symmetric_difference`(B)

Lecture 10

Introduction to Python. Built-in data types. Variables, lists, tuples, dictionaries, sets. Control structures. Functions. Docstrings.

Advanced Programming - SISSA, UniTS, 2023-2024

Pasquale Claudio Africa

30 Nov 2023

Outline

- 1. Introduction
- 2. Built-in data types
 - Numeric, Boolean, strings
 - Lists and tuples
 - Sets and dictionaries
- 3. Control structures
- 4. Comprehensions
- 5. Exceptions
- 6. Functions
- 7. Docstrings

Introduction

Why Python?

- **Readability and simplicity:** Python's syntax is designed for readability, making it easy for beginners and promoting clean code in large projects.
- **Versatility:** Supports various programming paradigms (procedural, OO, functional).
- **Extensive standard library:** Comes with a comprehensive standard library, reducing the need for writing code from scratch and promoting code reusability.
- **Large and active community:** Boasts a vibrant community, fostering collaboration, and providing a wealth of third-party libraries and online resources.
- **Data Science and Machine Learning:** Preferred language for data science and machine learning with powerful libraries like NumPy, SciPy, Pandas, PyTorch, and TensorFlow.
- **Cross-platform compatibility:** Code written in Python can run on different operating systems without modification.
- **Industry adoption:** Widely adopted in countless startups and enterprises.

Setting up a Python environment

To work with Python, you need to set up a development environment.

Here are the basic steps:

- **Install Python:** Download and install Python (version ≥ 3) from the official [Python website](#). Advanced users may want to have a look at [PyPy](#).
- **Integrated Development Environment (IDE):** Choose an IDE such as PyCharm, VSCode, or Jupyter Notebook for a more interactive development experience. You can even use online platforms like [Google Colab](#) and [JupyterLab](#).
- **Package management:** Utilize tools like `pip` to install and manage third-party packages.
- **(Advanced users) Virtual environments:** Use virtual environments, such as [conda](#) to isolate project dependencies and avoid conflicts between different projects.

Structure of a basic Python program

A Python program typically consists of the following components:

- **Comments:** Lines starting with `#` are comments. They are ignored by the Python interpreter and serve as notes for developers.
- **Statements:** Python code is composed of statements, which are instructions that the interpreter can execute.
- **Indentation:** Python uses indentation to define code blocks. Consistent indentation is crucial for proper program structure.

Hello, world!

```
# This is a comment.  
print("Hello, World!") # This is also a comment.  
  
x = 3  
  
# Code block with proper indentation.  
if x > 0:  
    print("Positive")
```

Definitions

A **value** is a piece of data that a computer program works with such as a number or text. There are different **types** of values: 42 is an integer and "Hello!" is a string. A **variable** is a name that refers to a value. In mathematics and statistics, we usually use variable names like x and y . In Python, we can use any word as a variable name as long as it starts with a letter or an underscore. However, it should not be a **reserved word** in Python such as for, while, class, lambda, etc. as these words encode special functionality in Python that we don't want to overwrite!

It can be helpful to think of a variable as a box that holds some information (a single number, a vector, a string, etc). We use the **assignment operator** = to assign a value to a variable.

Built-in data types

Python as a strongly, dynamically typed language

Python typing is **dynamic** so you can change a string variable to an `int` (in a **static** language you can't):

```
x = 'somestring'  
x = 50
```

Python typing is **strong** so you can't merge types:

```
'foo' + 3 # TypeError: cannot concatenate 'str' and 'int' objects
```

In weakly-typed languages (such as Javascript) this happens:

```
'foo' + 3 = 'foo3'
```

Built-in data types (1/2)

See the [Python 3 documentation](#) for a summary of the standard built-in Python datatypes.

Type name	Type Category	Description	Example
int	Numeric Type	integer number	42
float	Numeric Type	real number	3.14159
complex	Numeric Type	complex number	1.0 + 2.0j
bool	Boolean Values	true or false	True
str	Sequence Type	text	"I Can Has Cheezburger?"

Built-in data types (2/2)

Type name	Type Category	Description	Example
list	Sequence Type	a collection of objects - mutable & ordered	['Ali', 'Xinyi', 'Miriam']
tuple	Sequence Type	a collection of objects - immutable & ordered	('Thursday', 6, 9, 2018)
set	Set Type	a collection of unique objects - mutable & unordered	{'jack', 'sjoerd'}
dict	Mapping Type	mapping of key-value pairs	{'name': 'DSAI', 'code': 123, 'credits': 6}
NoneType	Null Object	represents no value	None

Numeric, Boolean, strings

Numeric data types

There are three distinct numeric types: integers , floating point numbers , and complex numbers . We can determine the type of an object in Python using type() . We can print the value of the object using print() .

```
x = 42
type(x) # int
print(x) # 42

pi = 3.14159
type(pi) # float

z = 1 + 3j # complex
print(abs(z))
import cmath
cmath.phase(z)
```

Arithmetic operators

Below is a table of the syntax for common arithmetic operations in Python:

Operator	Description
+	addition
-	subtraction
*	multiplication
/	division
**	exponentiation
//	integer division / floor division
%	modulo

Arithmetic operators: examples

```
1 + 2 + 3 + 4 + 5 # 15  
2 * 3.14159 # 6.28318  
2 ** 10 # 1024
```

Division may produce a different `dtype` than expected, it will change `int` to `float`.

```
int_2 = 2  
type(int_2) # int  
int_2 / int_2 # 1.0  
type(int_2 / int_2) # float
```

But the syntax `//` allows us to do *integer division* (aka *floor division*) and retain the `int` data type, it always rounds down.

```
101 / 2 # 50.5  
101 // 2 # 50 (floor division: always rounds down).
```

Boolean

The Boolean (`bool`) type has two values: `True` and `False`.

We can compare objects using comparison operators, and we'll get back a Boolean result:

Operator	Description
<code>x == y</code>	<code>x</code> is equal to <code>y</code>
<code>x != y</code>	<code>x</code> is not equal to <code>y</code>
<code>x > y</code>	<code>x</code> is greater than <code>y</code>
<code>x >= y</code>	<code>x</code> is greater than or equal to <code>y</code>
<code>x < y</code>	<code>x</code> is less than <code>y</code>
<code>x <= y</code>	<code>x</code> is less than or equal to <code>y</code>
<code>x is y</code>	<code>x</code> is the same object as <code>y</code>

Boolean operators

We also have so-called *boolean operators* which also evaluate to either `True` or `False`:

Operator	Description: <code>True</code> if
<code>x and y</code>	both <code>x</code> and <code>y</code> are <code>True</code>
<code>x or y</code>	at least one of <code>x</code> and <code>y</code> is <code>True</code>
<code>not x</code>	<code>x</code> is <code>False</code>

Python also has **bitwise operators** like AND (`&`), OR (`|`), XOR (`^`), NOT (`~`), shift (`<<` and `>>`).

Strings

Strings represent sequences of characters and are widely used in Python.

- **String creation:** Strings can be created using single (') or double (") quotes.
- **String operations:** Concatenation (+), repetition (*), and indexing.

Example

```
# String creation.  
message = "Hello, Python!"  
  
message_twice = message * 2 # "Hello, Python!Hello, Python!"  
  
# String operations.  
greeting = "Hello, "  
name = "Alice"  
full_greeting = greeting + name # Concatenation.
```

String manipulation

```
# String methods.  
message = "Hello, Python!"  
  
# Length of a string.  
length = len(message)  
  
# Upper and lower case.  
upper_case = message.upper() # Return a new string.  
lower_case = message.lower() # Return a new string.  
  
# String formatting.  
formatted_message = f"Message: {message}"
```

There are *many* string methods. Check out the [documentation](#).

String manipulation

```
all_caps = "HOW ARE YOU TODAY?"  
all_caps.split() # ['HOW', 'ARE', 'YOU', 'TODAY?']  
all_caps.count("O") # 3  
  
caps_list = list(all_caps)  
"".join(caps_list) # 'HOW ARE YOU TODAY?'  
"-".join(caps_list) # 'H-O-W- -A-R-E- -Y-O-U- -T-O-D-A-Y-?'  
"".join(caps_list).lower().split(" ") # ['how', 'are', 'you', 'today?']
```

String formatting

Python has ways of creating strings by *filling in the blanks* and formatting them nicely. This is helpful for when you want to print statements that include variables or statements. There are a few ways of doing this but I use and recommend [f-strings](#) which were introduced in Python 3.6. All you need to do is put the letter `f` out the front of your string and then you can include variables with curly-bracket notation `{}`.

```
name = "Newborn Baby"  
age = 4 / 12  
day = 30  
month = 7  
year = 2023  
template_new = f"Hello, my name is {name}. I am {age:.2f} years old. I was born {day}/{month:02}/{year}."  
template_new
```

'Hello, my name is Newborn Baby. I am 0.33 years old. I was born 30/07/2023.'

See format code options [here](#).

Lists and tuples

Lists and tuples

Lists and tuples are versatile data structures in Python.

- **Lists:** Mutable, ordered collections of items. Elements can be added, removed, or modified.
- **Tuples:** Immutable, ordered collections. Once defined, elements cannot be changed.

Example

```
# List.  
numbers = [1, 2, 3, 4, 5]  
  
# Tuple.  
coordinates = (2, 3)
```

List operations

```
# List creation.  
my_list = [1, 2, "THREE", 4, 0.5]  
another_list = [1, "two", [3, 4, "five"], True, None, {"key": "value"}]  
numbers = [1, 2, 3, 4, 5]  
  
len(numbers) # 5  
  
# Adding elements.  
numbers.append([6, 7]) # [1, 2, "THREE", 4, 0.5, [6, 7]]  
numbers.extend([6, 7]) # [1, 2, "THREE", 4, 0.5, 6, 7]  
  
# Removing elements.  
numbers.remove(3)  
popped_value = numbers.pop()
```

You can see the documentation for more [list methods](#).

Tuple operations

```
# Tuple creation.  
today = (1, 2, "THREE", 4, 0.5)  
coordinates = (2, 3)  
  
# Unpacking.  
x, y = coordinates  
  
# Concatenation.  
combined = coordinates + (4, 5) # (2, 3, 4, 5)  
  
# Tuple repetition.  
repeated = coordinates * 3
```

Indexing

We can access values inside a list, tuple, or string using square bracket syntax. Python uses *zero-based indexing*, which means the first element of the list is in position 0, not position 1.

```
my_list = [1, 2, 'THREE', 4, 0.5]
my_list[0] # 1
my_list[2] # 'THREE'
len(my_list) # 5

my_list[5] # IndexError: list index out of range
```

We can use negative indices to count backwards from the end of the list.

```
my_list = [1, 2, 'THREE', 4, 0.5]
my_list[-1] # 0.5
my_list[-2] # 4
```

Slicing

We can use the colon `:` to access a sub-sequence. This is called *slicing*.

```
my_list[1:3] # [2, 'THREE']
```

Note from the above that the **start** of the slice is **inclusive** and the **end** is **exclusive**. So

`my_list[1:3]` fetches elements 1 and 2, but not 3.

Strings behave the same as lists and tuples when it comes to indexing and slicing. Remember, we think of them as a sequence of characters.

```
alphabet = "abcdefghijklmnopqrstuvwxyz"  
alphabet[0] # 'a'  
alphabet[-1] # 'z'  
alphabet[-3] # 'x'  
alphabet[:5] # 'abcde'  
alphabet[12:20] # 'mnopqrst'
```

Sets and dictionaries

Sets (1/2)

Another built-in Python data type is the `set`, which stores an **unordered** list of **unique** items. Being unordered, sets do not record element position or order of insertion and so do not support indexing.

```
s = {2, 3, 5, 11}  
{1, 2, 3} == {3, 2, 1} # True  
[1, 2, 3] == [3, 2, 1] # False  
s.add(2) # Does nothing.
```

Since elements are stored unordered, sets can't be indexed:

```
s[0]
```

| TypeError: 'set' object is not subscriptable

Dictionaries

Dictionaries are key-value pairs, allowing efficient data retrieval.

- **Creating dictionaries:** Define key-value pairs using curly braces `{}`.
- **Accessing values:** Retrieve values using keys.

Example

```
# Dictionary creation.  
student = {"name": "Alice", "age": 20, "grade": "A"}  
  
# Accessing values.  
student_name = student["name"]  
  
# Adding and updating values.  
student["city"] = "New York"  
student["age"] = 21  
  
# Removing key-value pairs.  
del student["grade"]
```

Casting

Sometimes we need to explicitly **cast** a value from one type to another. Python tries to do the conversion, or throws an error if it can't.

```
x = 5.0 # float
x = int(5.0) # int
x = str(5.0) # string '5.0'
str(5.0) == 5.0 # False
int(5.3) # 5

original_set = {1, 2, 3, 4, 5}
converted_tuple = tuple(original_set)

original_dict = {'a': 1, 'b': 2, 'c': 3, 'd': 4}
converted_list = list(original_dict.items())
```

```
float("hello")
```

| ValueError: could not convert string to float: 'hello'

Empties

Sometimes you'll want to create empty objects that will be filled later on.

```
lst = list() # Or:  
lst = []
```

There's no real difference between the two methods above, [] is apparently marginally faster .

```
tup = tuple() # Or:  
tup = ()
```

```
dic = dict() # Or:  
dic = {}
```

```
st = set()
```

None

`NoneType` is its own type in Python. It only has one possible value, `None` - it represents an object with no value.

```
x = None  
print(x)
```

None

```
type(x)
```

NoneType

Control structures

Conditionals (1/2)

Conditional statements allow us to write programs where only certain blocks of code are executed depending on the state of the program. Let's look at some examples and take note of the keywords, syntax and indentation.

```
name = "Tom"

if name.lower() == "tom":
    print("That's my name too!")
elif name.lower() == "santa":
    print("That's a funny name.")
else:
    print(f"Hello {name}! That's a cool name!")
print("Nice to meet you!")
```

That's my name too!

Nice to meet you!

Conditionals (2/2)

The main points to notice:

- Use keywords `if`, `elif` and `else`.
- The colon `:` ends each conditional expression.
- Indentation (by 4 empty space) defines code blocks.
- In an `if` statement, the first block whose conditional statement returns `True` is executed and the program exits the `if` block.
- `if` statements don't necessarily need `elif` or `else`.
- `elif` lets us check several conditions.
- `else` lets us evaluate a default block if all other conditions are `False`.
- the end of the entire `if` statement is where the indentation returns to the same level as the first `if` keyword.

Conditionals: nesting

If statements can also be **nested** inside of one another:

```
name = "Super Tom"

if name.lower() == "tom":
    print("That's my name too!")
elif name.lower() == "santa":
    print("That's a funny name.")
else:
    print(f"Hello {name}! That's a cool name.")
    if name.lower().startswith("super"):
        print("Do you really have superpowers?")

print("Nice to meet you!")
```

Hello Super Tom! That's a cool name.

Do you really have superpowers?

Nice to meet you!

Inline if/else

We can write simple `if` statements *inline*, i.e., in a single line, for simplicity (similar to the ternary operator `(condition) ? if_true : if_false` in C++).

```
words = ["the", "list", "of", "words"]

x = "long list" if len(words) > 10 else "short list"

# This is equivalent to:
if len(words) > 10:
    x = "long list"
else:
    x = "short list"
```

Truth value testing (1/2)

Any object can be tested for *truth* in Python, for use in `if` and `while` statements.

- `True` values: all objects return `True` unless they are a `bool` object with value `False` or have `len() == 0`
- `False` values: `None`, `False`, `0`, empty sequences and collections: `''`, `()`, `[]`, `{}`, `set()`

Read more in the documentation [here](#).

```
x = 1

if x:
    print("I'm truthy!")
else:
    print("I'm falsey!")
```

I'm truthy!

Truth value testing (2/2)

```
x = False

if x:
    print("I'm truthy!")
else:
    print("I'm falsey!")
```

I'm falsey!

```
x = []

if x:
    print("I'm truthy!")
else:
    print("I'm falsey!")
```

I'm falsey!

for loops (1/2)

for loops allow us to execute code a specific number of times.

```
for n in [2, 7, -1, 5]:  
    print(f"The number is {n} and its square is {n**2}")  
print("I'm outside the loop!")
```

The number is 2 and its square is 4

The number is 7 and its square is 49

The number is -1 and its square is 1

The number is 5 and its square is 25

I'm outside the loop!

for loops (2/2)

The main points to notice:

- Keyword `for` begins the loop. Colon `:` ends the first line of the loop.
- Block of code indented is executed for each value in the list (hence the name *for loops*).
- The loop ends after the variable `n` has taken all the values in the list.
- We can iterate over any kind of *iterable*: `range` , `string` , `list` , `tuple` , `set` , `dict` .
- An iterable is really just any object with a sequence of values that can be looped over. In this case, we are iterating over the values in a list.

```
word = "Python"
for letter in word:
    print("Gimme a " + letter + "!")
print(f"What's that spell?! {word}!")
```

range (1/2)

A very common pattern is to use `for` with the `range()`. `range()` gives you a sequence of integers up to some value (non-inclusive of the end-value) and is typically used for looping.

```
range(10)
```

```
range(0, 10)
```

```
list(range(10))
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

range (2/2)

```
for i in range(10):  
    print(i)
```

We can also specify a start value and a skip-by value with `range` :

```
for i in range(1, 101, 10):  
    print(i)
```

1
11
21
...
91

Nested `for` loops (1/2)

We can write a loop inside another loop to iterate over multiple dimensions of data:

```
for x in [1, 2, 3]:  
    for y in ["a", "b", "c"]:  
        print((x, y))
```

(1, 'a')

(1, 'b')

(1, 'c')

(2, 'a')

(2, 'b')

(2, 'c')

(3, 'a')

(3, 'b')

(3, 'c')

Nested `for` loops (2/2)

Or, almost equivalently:

```
list_1 = [0, 1, 2]
list_2 = ["a", "b", "c"]
for i in range(3):
    print(list_1[i], list_2[i])
```

```
0 a
1 b
2 c
```

zip

`zip` returns a zip object which is an iterable of tuples.

```
for i in zip(list_1, list_2):
    print(i)
```

```
(0, 'a')
(1, 'b')
(2, 'c')
```

We can even *unpack* these tuples directly in the `for` loop:

```
for i, j in zip(list_1, list_2):
    print(i, j)
```

```
0 a
1 b
2 c
```

enumerate

`enumerate` adds a counter to an iterable which we can use within the loop.

```
for i in enumerate(list_2):
    print(i)
```

```
(0, 'a')
(1, 'b')
(2, 'c')
```

```
for n, i in enumerate(list_2):
    print(f"index {n}, value {i}")
```

```
index 0, value a
index 1, value b
index 2, value c
```

Looping over dictionaries

We can loop through key-value pairs of a dictionary using `.items()`. The general syntax is `for key, value in dictionary.items()`.

```
courses = {"Programming": "awesome!",
           "Statistics": "naptme!"}

for course, description in courses.items():
    print(f"{course} is {description}")
```

Programming is awesome!
Statistics is naptme!

```
for n, (course, description) in enumerate(courses.items()):
    print(f"Item {n}: {course} is {description}")
```

Item 0: Programming is awesome!
Item 1: Statistics is naptme!

while loops

We can also use a `while loop` to execute a block of code until a condition is met.

```
n = 10
while n > 0:
    print(n)
    n -= 1
```

```
10
9
8
...
1
```

break

Hence, in some cases, you may want to force a loop to stop based on some criteria, using the `break` keyword.

```
n = 123
i = 0
while n != 1:
    print(int(n))
    if n % 2 == 0: # n is even.
        n = n / 2
    else: # n is odd
        n = n * 3 + 1
    i += 1
    if i == 10:
        print(f"Ugh, too many iterations!")
        break
```

123

...

Ugh, too many iterations!

continue

The `continue` keyword is similar to `break` but won't stop the loop. Instead, it just restarts the loop from the next iteration.

```
n = 10
while n > 0:
    if n % 2 != 0: # n is odd
        n = n - 1
        continue
    print(n)
    n = n - 1
```

10
8
6
4
2

Comprehensions

Comprehensions

Comprehensions allow us to build lists/tuples/sets/dictionaries in one convenient, compact line of code. I use these quite a bit! Below is a standard `for` loop you might use to iterate over an iterable and create a list:

```
subliminal = ['Tom', 'ingests', 'many', 'eggs', 'to', 'outrun', 'large', 'eagles', 'after', 'running', 'near', '!']
first_letters = []
for word in subliminal:
    first_letters.append(word[0])
print(first_letters)
```

```
['T', 'i', 'm', 'e', 't', 'o', 'l', 'e', 'a', 'r', 'n', '!']
```

List comprehension allows us to do this in one compact line:

```
letters = [word[0] for word in subliminal] # List comprehension.
letters
```

```
['T', 'i', 'm', 'e', 't', 'o', 'l', 'e', 'a', 'r', 'n', '!']
```

Multiple comprehensions

We can make things more complicated by doing multiple iteration or conditional iteration:

```
[(i, j) for i in range(3) for j in range(4)]
```

```
[(0, 0), (0, 1), (0, 2), ..., (2, 3)]
```

```
[i for i in range(11) if i % 2 == 0]
```

```
[0, 2, 4, 6, 8, 10]
```

```
[-i if i % 2 else i for i in range(11)]
```

```
[0, -1, 2, -3, 4, -5, 6, -7, 8, -9, 10]
```

Set and dict comprehensions

Set comprehension:

```
words = ['hello', 'goodbye', 'the', 'antidisestablishmentarianism']
y = {word[-1] for word in words} # {'e', 'o', 'm'}
```

Dictionary comprehension:

```
word_lengths = {word:len(word) for word in words}
# {'hello': 5, 'goodbye': 7, 'the': 3, 'antidisestablishmentarianism': 28}
```

Tuple comprehension doesn't work as you might expect: we get a **generator** instead (explained below).

Exceptions

Exceptions

If something goes wrong, we don't want our code to crash - we want it to **fail gracefully**. In Python, this can be accomplished using `try / except`. Here is a basic example:

```
this_variable_does_not_exist
print("Another line") # Code fails before getting to this line.
```

NameError: name 'this_variable_does_not_exist' is not defined

try-catch

```
try:  
    this_variable_does_not_exist  
except:  
    pass # Do nothing.  
    print("You did something bad! But I won't raise an error.")  
print("Another line")
```

```
You did something bad! But I won't raise an error.  
Another line
```

Python tries to execute the code in the `try` block. If an error is encountered, we *catch* this in the `except` block (also called `try / catch` in other languages). There are many different error types, or **exceptions** - we saw `NameError` above.

5 / 0

More exception types

```
my_list = [1, 2, 3]
my_list[5]
```

| IndexError: list index out of range

```
my_tuple = (1, 2, 3)
my_tuple[0] = 0
```

| TypeError: 'tuple' object does not support item assignment

Raise exceptions

We can also write code that raises an exception on purpose, using `raise`:

```
def add_one(x):
    if not isinstance(x, float) and not isinstance(x, int):
        raise TypeError(f"Sorry, x must be numeric, you entered a {type(x)}.")

    return x + 1

add_one("blah")
```

| `TypeError: Sorry, x must be numeric, you entered a <class 'str'>.`

This is useful when your function is complicated and would fail in a complicated way, with a weird error message. You can make the cause of the error much clearer to the *user* of the function.

Finally, we can even define our own exception types by inheriting from the `Exception` class - we'll explore classes and inheritance in the next lecture!

Functions

Functions

A **function** is a reusable piece of code that can accept input parameters, also known as *arguments*. For example, let's define a function called `square` which takes one input parameter `n` and returns the square `n**2`:

```
def square(n):
    n_squared = n**2
    return n_squared
```

```
square(2) # 4
square(100) # 10000
square(12345) # 152399025
```

Functions begin with the `def` keyword, then the function name, arguments in parentheses, and then a colon (`:`). The code executed by the function is defined by indentation. The output or *return* value of the function is specified using the `return` keyword.

Local variables

When you create a variable inside a function, it is local, which means that it only exists inside the function. For example:

```
def cat_string(str1, str2):
    string = str1 + str2
    return string

cat_string('My name is ', 'Tom')
```

'My name is Tom'

string

NameError: name 'string' is not defined

Mutable vs. immutable input arguments (1/2)

Strings and tuples are immutable types which means they can't be modified. Lists are mutable and we can assign new values for its various entries. This is the main difference between lists and tuples.

```
names_list = ["Indiana", "Fang", "Linsey"]
names_list[0] = "Cool guy" # Ok.

names_tuple = ("Indiana", "Fang", "Linsey")
names_tuple[0] = "Not cool guy"
```

| TypeError: 'tuple' object does not support item assignment

Same goes for strings. Once defined we cannot modify the characters of the string.

```
my_name = "Tom"
my_name[-1] = "q"
```

| TypeError: 'str' object does not support item assignment

Mutable vs. immutable input arguments (2/2)

```
x = ([1, 2, 3], 5)  
x[1] = 7
```

| TypeError: 'tuple' object does not support item assignment

```
x[0][1] = 4 # Ok. We are modifying a list here.
```

⚠ Warning

In Python, input arguments are passed by **reference**.

- When **mutable** objects are passed to a function, changes made to the object inside the function affect the original object outside the function.
- When **immutable** objects are passed to a function, they cannot be modified inside the function, hence the original object remains unchanged.

Side effects

If a function changes the variables passed into it, then it is said to have **side effects**. For example:

```
def silly_sum(my_list):  
    my_list.append(0)  
    return sum(my_list)
```

```
l = [1, 2, 3, 4]  
out = silly_sum(l) # 10
```

The above looks like what we wanted? But it changed our `l` object.

```
l
```

```
[1, 2, 3, 4, 0]
```

If your function has side effects like this, you must mention it in the documentation.

Null return type

If you do not specify a return value, the function returns `None` when it terminates:

```
def f(x):
    x + 1 # No return.
    if x == 999:
        return
print(f(0))
```

None

Default arguments (1 / 2)

Sometimes it is convenient to have *default values* for some arguments in a function. Because they have default values, these arguments are optional, and are hence called *optional arguments*. For example:

```
def repeat_string(s, n = 2):  
    return s * n
```

```
repeat_string("abc", 5)
```

```
'abcabcaabcabc'
```

```
repeat_string("abc")
```

```
'abcabc'
```

You can have any number of required arguments and any number of optional arguments.

Default arguments (2 / 2)

All the optional arguments must come after the required arguments. The required arguments are mapped by the order they appear. The optional arguments can be specified out of order when using the function.

```
def example(a, b, c = "DEFAULT", d = "DEFAULT"):
    print(a, b, c, d)

example(1, 2, 3, 4) # 1 2 3 4
```

- Using the defaults for `c` and `d`:

```
example(1, 2) # 1 2 DEFAULT DEFAULT
```

- Specifying `c` and `d` as **keyword arguments** (i.e. by name):

```
example(1, 2, c=3, d=4) # 1 2 3 4
```

Type hints

Type hinting is exactly what it sounds like, it hints at the data type of function arguments. You can indicate the type of an argument in a function using the syntax `argument : dtype`, and the type of the return value using `def func() -> dtype`. Let's see an example:

```
def repeat_string(s: str, n: int = 2) -> str:  
    return s * n
```

Type hinting just helps your users and IDE identify dtypes and possible bugs. It's just another level of documentation. They do not force users to use that date type, for example, I can still pass an `dict` to `repeat_string` if I want to:

```
repeat_string({'key_1': 1, 'key_2': 2})
```

| TypeError: unsupported operand type(s) for *: 'dict' and 'int'

Multiple return values (1/2)

In many programming languages, functions can only return one object. That is technically true in Python too, but there is a *workaround*, which is to return a tuple.

```
def sum_and_product(x, y):
    return (x + y, x * y)

sum_and_product(5, 6) # (11, 30)
```

The parentheses can be omitted (and often are), and a `tuple` is implicitly returned as defined by the use of the comma:

```
def sum_and_product(x, y):
    return x + y, x * y

sum_and_product(5, 6) # (11, 30)
```

Multiple return values (2/2)

It is common to immediately unpack a returned tuple into separate variables, so it really feels like the function is returning multiple values:

```
s, p = sum_and_product(5, 6)
```

As an aside, it is conventional in Python to use `_` for values to be discarded:

```
s, _ = sum_and_product(5, 6)
```

 **Warning:** `_` becomes an actual variable name!

Unpacking (1/3)

In Python, the asterisk (`*`) is used for unpacking iterable objects. It allows you to extract the elements from an iterable (e.g., a list, tuple) or the key-value pairs from a dictionary.

Here are a few common use cases for the asterisk for unpacking:

- 1. Unpacking in function calls:** When calling a function, you can use the asterisk to unpack the elements of a list, tuple, or any iterable as individual arguments to the function.

```
def add_numbers(a, b, c):
    return a + b + c

numbers = [1, 2, 3]
result = add_numbers(*numbers)
print(result) # 6
```

Unpacking (2/3)

2. **Unpacking in iterables:** You can use the asterisk to unpack elements from one iterable into another.

```
first_list = [1, 2, 3]
second_list = [4, 5, 6]
combined_list = [*first_list, *second_list]
print(combined_list) # [1, 2, 3, 4, 5, 6]
```

3. **Unpacking in tuple assignment:** The asterisk can be used in tuple assignment to capture multiple elements at once.

```
first, *rest = [1, 2, 3, 4, 5]
print(first) # 1
print(rest) # [2, 3, 4, 5]
```

Unpacking (3/3)

4. **Unpacking in dictionary merging:** When merging dictionaries, the double asterisk (`**`) is used to unpack the key-value pairs from one dictionary into another.

```
dict1 = {'a': 1, 'b': 2}
dict2 = {'c': 3, 'd': 4}
merged_dict = {**dict1, **dict2}
print(merged_dict) # {'a': 1, 'b': 2, 'c': 3, 'd': 4}
```

5. **Extended unpacking in function definitions:** In function definitions, you can use the asterisk to collect variable positional arguments (`*args`) and variable keyword arguments (`**kwargs`).

```
def example_function(a, b, *args, **kwargs):
    # a and b are regular arguments
    # args is a tuple of positional arguments
    # kwargs is a dictionary of keyword arguments
    pass
```

Functions with arbitrary number of arguments: *args

In Python, `*args` and `**kwargs` are used in function definitions to allow a variable number of arguments.

- `*args` (Arbitrary positional arguments): It allows a function to accept a variable number of positional arguments. The `*args` parameter is used to collect any number of positional arguments into a tuple.

```
def print_args(*args):
    for arg in args:
        print(arg)

print_args(1, 2, 3, "four")
```

Functions with arbitrary number of arguments:

**kwargs

- **kwargs (Arbitrary keyword arguments): It allows a function to accept a variable number of keyword arguments. The **kwargs parameter is used to collect any number of keyword arguments into a dictionary.

```
def print_kwargs(**kwargs):  
    for key, value in kwargs.items():  
        print(f"{key}: {value}")  
  
print_kwargs(name="John", age=25, city="New York")
```

Combining `*args` and `**kwargs`

You can use both `*args` and `**kwargs` in the same function definition to accept any combination of positional and keyword arguments.

```
def print_args_and_kwargs(*args, **kwargs):
    for arg in args:
        print(arg)
    for key, value in kwargs.items():
        print(f"{key}: {value}")

print_args_and_kwargs(1, 2, 3, name="John", age=25)
```

1

2

3

name: John

age: 25

Functions as a data type

In Python, functions are actually a data type:

```
def do_nothing(x):
    return x

type(do_nothing) # function
```

This means you can pass functions as arguments into other functions.

```
def square(y):
    return y ** 2

def evaluate_function_on_x_plus_1(fun, x):
    return fun(x + 1)

evaluate_function_on_x_plus_1(square, 5) # 36
```

Anonymous functions

There are two ways to define functions in Python. The way we've been using up until now:

```
def add_one(x):
    return x + 1

add_one(7.2) # 8.2
```

Or by using the `lambda` keyword:

```
add_one = lambda x: x + 1
type(add_one) # function
```

The two approaches above are identical. The one with `lambda` is called an **anonymous function**. Anonymous functions can only take up one line of code, so they aren't appropriate in most cases, but can be useful for smaller things.

Generators (1/3)

Recall list comprehension:

```
[n for n in range(10)]
```

Comprehensions evaluate the entire expression at once, and then returns the full data product. Sometimes, we want to work with just one part of our data at a time, for example, when we can't fit all of our data in memory. For this, we can use *generators*.

```
(n for n in range(10))
```

```
<generator object at 0x7f06c9b9ba70>
```

Notice that we just created a `generator object`. Generator objects are like a *recipe* for generating values. They don't actually do any computation until they are asked to.

Generators (2/3)

We can get values from a generator in three main ways:

- Using `next()`
- Using `list()`
- Looping

```
gen = (n for n in range(10))
next(gen) # 0
next(gen) # 1
```

Once the generator is exhausted, it will raise a `StopIteration` exception.

We can see all the values of a generator using `list()` but this defeats the purpose of using a generator in the first place:

```
gen = (n for n in range(10))
list(gen) # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Generators (3/3)

Finally, we can iterate over generator objects too:

```
gen = (n for n in range(10))
for i in gen:
    print(i)
```

Above, we saw how to create a generator object using comprehension syntax but with parentheses. We can also create a generator using functions and the `yield` keyword:

```
def gen():
    for n in range(10):
        yield (n, n ** 2)

g = gen()
next(g) # (0, 0)
next(g) # (1, 1)
next(g) # (2, 4)
```

Docstrings

Docstrings

Writing good functions brings up the idea of function documentation, called "docstrings". The **docstring** goes right after the `def` line and is wrapped in **triple quotes** `"""`.

```
def make_palindrome(string):
    """Turns the string into a palindrome by concatenating itself with a reversed version of itself."""

    return string + string[::-1]
```

In Python we can use the `help()` function to view another function's documentation. In Jupyter, we can use `?` to view the documentation string of any function in our environment.

```
make_palindrome?
```

Docstring: structure

General docstring convention in Python is described in [PEP 257 - Docstring Conventions](#). There are many different docstring style conventions used in Python. The exact style you use can be important for helping you to render your documentation, or for helping your IDE parse your documentation. Common styles include:

1. **Single-line**: If it's short, then just a single line describing the function will do (as above).
2. **reST style**: see [here](#).
3. **NumPy style**: see [here](#).
4. **Google style**: see [here](#).

Docstrings: the NumPy style

```
def function_name(param1, param2, param3):
    """First line is a short description of the function.

    A paragraph describing in a bit more detail what the
    function does and what algorithms it uses and common
    use cases.

    Parameters
    -----
    param1 : datatype
        A description of param1.
    param2 : datatype
        A description of param2.
    param3 : datatype
        A longer description because maybe this requires
        more explanation and we can use several lines.

    Returns
    -----
    datatype
        A description of the output, datatypes and behaviours.
        Describe special cases and anything the user needs to
        know to use the function.

    Examples
    -----
    >>> function_name(3, 8, -5)
    2.0
    """
```

Docstrings: the NumPy style (example)

```
def make_palindrome(string):
    """Turns the string into a palindrome by concatenating
    itself with a reversed version of itself.

    Parameters
    -----
    string : str
        The string to turn into a palindrome.

    Returns
    -----
    str
        string concatenated with a reversed version of string

    Examples
    -----
    >>> make_palindrome('tom')
    'tommot'
    """
    return string + string[::-1]
```

Docstrings with optional arguments

```
def repeat_string(s, n = 2):
    """
        Repeat the string s, n times.

    Parameters
    -----
    s : str
        the string
    n : int, optional
        the number of times, by default = 2

    Returns
    -----
    str
        the repeated string

    Examples
    -----
    >>> repeat_string("Blah", 3)
    "BlahBlahBlah"
    """
    return s * n
```



Modules and packages.
Object-oriented programming.
Classes, inheritance and polymorphism.

Lecture 11

Object-oriented programming. Classes, inheritance and polymorphism. Modules and packages.

Advanced Programming - SISSA, UniTS, 2023-2024

Pasquale Claudio Africa

05 Dec 2023

Outline

1. OOP in Python
2. Decorators
3. Inheritance and polymorphism
4. Modules and packages

Object-oriented programming in Python

Object-oriented programming in Python

We've encountered built-in data types like `dict` and `list`. However, Python allows us to define our own data types using classes. A class serves as a blueprint for creating objects, following the principles of **object-oriented programming**.

```
d = dict()
```

In this example, `d` is an object, while `dict` is a type.

```
type(d)
```

```
dict
```

```
type(dict)
```

```
type
```

We refer to `d` as an **instance** of the **type** `dict`.

The need for custom classes

Custom classes become invaluable when we need to organize and manage complex data structures efficiently. Let's illustrate this with an example involving the Advanced Programming course (AdvProg) members. Initially, we store information in a dictionary:

```
advprog_1 = {'first': Pasquale, 'last': 'Africa', 'email': 'pasquale.africa@sissa.it'}
```

To extract a member's full name, we define a function:

```
def full_name(first, last):
    return f"{first} {last}"
```

This approach requires repetitive code for each member:

```
advprog_2 = {'first': 'Marco', 'last': 'Feder', 'email': marco.feder@sissa.it'}
full_name(advprog_2['first'], advprog_2['last'])
```

Creating a class for efficiency

To address the inefficiency, we can create a class as a blueprint for AdvProg members:

```
class AdvProgMember:  
    pass
```

We enhance this blueprint by adding an `__init__` method to initialize instances with specific data:

```
class AdvProgMember:  
    def __init__(self, first, last):  
        self.first = first  
        self.last = last  
        self.email = first.lower() + "." + last.lower() + "@sissa.it"
```

The `self`

Class methods have **only one** specific difference from ordinary functions: they must have an extra first name that has to be added to the beginning of the parameter list, but you **do not** give a value for this parameter when you call the method, Python will provide it. This particular variable refers to the object *itself*, and by convention, it is given the name `self`.

You must be wondering how Python gives the value for `self` and why you don't need to give a value for it. An example will make this clear. Say you have a class called `MyClass` and an instance of this class called `myobject`. When you call a method of this object as `myobject.method(arg1, arg2)`, this is automatically converted by Python into `MyClass.method(myobject, arg1, arg2)` - this is all the special `self` is about.

This also means that if you have a method which takes no arguments, then you still have to have one argument, i.e. the `self`.

The `__init__` method

There are many method names which have special significance in Python classes. We will see the significance of the `__init__` method now.

The `__init__` method is run as soon as an object of a class is instantiated (i.e. created). The method is useful to do any *initialization* (i.e. passing initial values to your object) you want to do with your object. Notice the double underscores both at the beginning and at the end of the name.

We do not explicitly call the `__init__` method, but it is automatically invoked when creating an instance of a class:

```
advprog_1 = AdvProgMember('Pasquale', 'Africa')
print(advprog_1.first)
print(advprog_1.last)
print(advprog_1.email)
```

Methods

To simplify accessing a member's full name, we integrate it as a class method:

```
class AdvProgMember:  
    def __init__(self, first, last):  
        self.first = first  
        self.last = last  
        self.email = first.lower() + "." + last.lower() + "@sissa.it"  
  
    def full_name(self): # Notice 'self' as an input argument.  
        return f"{self.first} {self.last}"  
  
advprog_1 = AdvProgMember('Pasquale', 'Africa')  
print(advprog_1.full_name())
```

Class vs. object (or instance) attributes

Attributes can be instance-specific (`advprog_1.first`) or shared among all instances (`AdvProgMember.campus`). Class attributes are defined outside the `__init__` method.

```
class AdvProgMember:  
    role = "Advanced Programming member"  
    campus = "SISSA"  
  
    def __init__(self, first, last, email):  
        self.first = first  
        self.last = last  
        self.email = first.lower() + "." + last.lower() + "@sissa.it"  
  
advprog_1 = AdvProgMember('Pasquale', 'Africa')  
print(f"{advprog_1.first} is at campus {advprog_1.campus}.")  
print(f"{advprog_1.first} is at campus {AdvProgMember.campus}.")
```

⚠️ Changing class attributes affects all instances!

Class methods

Besides regular methods, classes offer class methods and static methods. Class methods, identified by `@classmethod`, act on the class itself, often serving as alternative constructors.

```
class AdvProgMember:  
    @classmethod  
    def from_csv(cls, csv_name):  
        first, last = csv_name.split(',')  
        return cls(first, last)  
  
advprog_1 = AdvProgMember.from_csv('Pasquale,Africa')  
advprog_1.full_name()
```

Static methods

Static methods, marked with `@staticmethod`, operate independently of instances and classes but are relevant to the class.

```
class AdvProgMember:  
    @staticmethod  
    def is_exam_date(date):  
        return True if date in ["Jan 17th", "Feb 13th"] else False  
  
print(f"Is Dec 5th an exam date? {AdvProgMember.is_exam_date('Dec 5th')}")  
print(f"Is Feb 13th an exam date? {AdvProgMember.is_exam_date('Feb 13th')}")
```

Magic methods (1/3)

In Python, magic methods, also known as dunder (double underscore) methods, are special methods that start and end with double underscores. Magic methods are automatically invoked by the Python interpreter in response to certain events or operations.

Example: __add__

```
class Point:  
    def __init__(self, x, y):  
        self.x = x  
        self.y = y  
  
    def __add__(self, other_point):  
        return Point(self.x + other_point.x, self.y + other_point.y)  
  
point1 = Point(1, 2)  
point2 = Point(3, 4)  
result = point1 + point2
```

Magic methods (2/3)

Example: `__eq__` and `__call__`

```
class CustomObject:  
    def __init__(self, value):  
        self.value = value  
  
    def __eq__(self, other):  
        return self.value == other.value  
  
    def __call__(self, *args, **kwargs):  
        return f"Called with args: {args}, kwargs: {kwargs}"  
  
obj1 = CustomObject(42)  
obj2 = CustomObject(42)  
print(obj1 == obj2) # Output: True  
  
result = obj1(1, 2, key="value")  
print(result) # Output: Called with args=(1, 2), kwargs={'key': 'value'}
```

Magic methods (3/3)

Example: `__getitem__` and `__setitem__`

```
class MyContainer:  
    def __init__(self, data):  
        self.data = data  
  
    def __getitem__(self, index):  
        return self.data[index]  
  
    def __setitem__(self, index, value):  
        self.data[index] = value  
  
container = MyContainer([1, 2, 3, 4, 5])  
print(container[2])      # Output: 3  
container[2] = 10  
print(container[2])      # Output: 10
```

Summary of common magic methods (1/5)

Object initialization and cleanup

- `__init__(self[, ...])` : Constructor method, initializes a new instance.
- `__del__(self)` : Destructor method, called when the object is about to be destroyed.

Object representation

- `__str__(self)` : Used by `str()` and `print()` to get a human-readable string representation.
- `__repr__(self)` : Used by `repr()` and the interactive interpreter for a developer-friendly representation.
- `__format__(self, format_spec)` : Customizes the formatting when using the `format()` function.

Summary of common magic methods (2/5)

Attribute access

- `__getattr__(self, name)` : Called when an attribute lookup fails.
- `__setattr__(self, name, value)` : Called when an attribute is set.
- `__delattr__(self, name)` : Called when an attribute is deleted.

Container and iteration

- `__len__(self)` : Returns the length of the object; used by `len()`.
- `__getitem__(self, key)` : Enables indexing and slicing; used by `obj[key]`.
- `__setitem__(self, key, value)` : Enables index assignment `obj[key] = value`.
- `__delitem__(self, key)` : Enables deletion of an index; used by `del obj[key]`.
- `__iter__(self)` : Returns an iterator object; used by `iter()`.
- `__next__(self)` : Retrieves the next item from the iterator; used by `next()`.

Summary of common magic methods (3/5)

Comparison

- `__eq__(self, other)` : Defines equality; used by `==`.
- `__ne__(self, other)` : Defines non-equality; used by `!=`.
- `__lt__(self, other)` : Defines less than; used by `<`.
- `__le__(self, other)` : Defines less than or equal to; used by `<=`.
- `__gt__(self, other)` : Defines greater than; used by `>`.
- `__ge__(self, other)` : Defines greater than or equal to; used by `>=`.
- `__bool__(self)` : Defines truthiness; used by `bool()`.

Summary of common magic methods (4/5)

Mathematical operations

- `__add__(self, other)` : Defines addition; used by `+`.
- `__sub__(self, other)` : Defines subtraction; used by `-`.
- `__mul__(self, other)` : Defines multiplication; used by `*`.
- `__truediv__(self, other)` : Defines true division; used by `/`.
- `__floordiv__(self, other)` : Defines floor division; used by `//`.
- `__mod__(self, other)` : Defines modulo; used by `%`.
- `__pow__(self, other[, modulo])` : Defines exponentiation; used by `**`.

Summary of common magic methods (5/5)

Callable objects

- `__call__(self[, args[, kwargs]])` : Allows an instance to be called as a function.

Context management

- `__enter__(self)`
- `__exit__(self, exc_type, exc_value, traceback)`

Used for resource acquisition and release in a `with` statement:

```
with open("example.txt", "r") as file:  
    content = file.read()  
    print(content)  
# File is automatically closed outside the 'with' block.
```

Notes for C++ programmers

- In Python, even integers are treated as objects (of the `int` class). This is unlike C++ where integers are primitive native type.
- The `self` in Python is equivalent to the `this` pointer in C++.
- All class members (including the data members) are *public* and all the methods are *virtual* in Python.
- If you use data members with names using the *double underscore prefix* such as `__myvar`, Python uses name-mangling to effectively make it (almost) a private variable. Any identifier of the form `__myvar` (at least two leading underscores or at most one trailing underscore) is replaced with `_MyClass__myvar`, where `MyClass` is the current class name with a leading underscore(s) stripped. After all, private members can still be accessed... You can check using the built-in `dir()` function.

Decorators

Decorators

Decorators in Python offer a powerful way to enhance the functionality of functions or methods. They act as wrappers, allowing you to extend or modify the behavior of the original function. Let's delve deeper into decorators with examples and explore their practical applications.

Decorators can be imagined to be a shortcut to calling a wrapper function (i.e. a function that "wraps" around another function so that it can do something before or after the inner function), so applying the `@classmethod` decorator is the same as calling:

```
from_csv = classmethod(from_csv)
```

Defining decorators

Decorators are essentially functions that take another function as input, enhance its capabilities, and return a modified version of the original function.

```
def my_decorator(original_func):
    def wrapper():
        print(f"A decoration before {original_func.__name__}.")
        result = original_func()
        print(f"A decoration after {original_func.__name__}.")
        return result
    return wrapper

my_decorator(original_func)()

# Or, re-assigning the original symbol:
original_func = my_decorator(original_func)
original_func()
```

NB: `__name__` is a special attribute that returns the name of a function, class or module as a string.

Improved syntax using @

While the previous example works, Python provides a more readable syntax using the @ symbol.

The equivalent of the previous example using this syntax is:

This decorator, when applied to a function, surrounds the function call with additional actions. For instance:

```
@my_decorator  
def original_func():  
    print("I'm the original function!")
```

```
original_func()
```

A decoration before original_func.

I'm the original function!

A decoration after original_func.

Practical example: timer decorator (1/2)

Now, let's create a more practical decorator that measures the execution time of a function. This example utilizes the `time` module:

```
import time

def timer(my_function):
    def wrapper():
        t1 = time.time()
        result = my_function()
        t2 = time.time()
        print(f"{my_function.__name__} ran in {t2 - t1:.3f} sec")
        return result
    return wrapper
```

(More details about `import` will follow).

Practical example: timer decorator (2/2)

Applying this decorator to a function allows us to measure its execution time:

```
@timer
def silly_function():
    for i in range(1e7):
        if (i % 1e6) == 0:
            print(i)

silly_function()
```

```
0
1000000
2000000
...
9000000
silly_function ran in 0.601 sec
```

Decorators and classes (1/2)

A decorator can be applied to classes as well:

```
def add_method(cls):
    def new_method(self):
        return f"Hello from the new method of {cls.__name__}!"
    cls.new_method = new_method
    return cls

@add_method
class MyClass:
    def existing_method(self):
        return "Hello from the existing method!"

obj = MyClass()
result_new = obj.new_method()
```

Decorators and classes (2/2)

... or be a class itself:

```
class CustomDecorator:  
    def __init__(self, func):  
        self.func = func  
  
    def __call__(self, *args, **kwargs):  
        print(f"Decorating function {self.func.__name__}")  
        result = self.func(*args, **kwargs)  
        print(f"Function {self.func.__name__} finished execution")  
        return result  
  
@CustomDecorator  
def my_function():  
    print("Executing my_function")  
  
my_function()
```

Built-in decorators

Python comes with built-in decorators like `classmethod` and `staticmethod`, which are implemented in C for efficiency. Although we won't dive into their implementation, they are widely used in practice.

Other than logging and timing/profiling, decorators can be used to **add validation checks for input parameters** and **output cleanup** to functions or methods, or to implement **caching** mechanisms, where the result of a function is stored for a specific set of inputs, and subsequent calls with the same inputs can return the cached result.

In conclusion, decorators provide a flexible and elegant way to enhance the behavior of functions in Python. While creating custom decorators may not be a daily necessity, understanding them is crucial for leveraging Python's full potential.

Inheritance and polymorphism

Inheritance and subclasses

Inheritance in Python enables classes to inherit methods and attributes from other classes. Previously, we worked with the `AdvProgMember` class, but now let's delve into creating more specialized classes like `AdvProgStudent` and `AdvProgInstructor`.

```
class AdvProgMember:  
    # ...
```

Now, to create an `AdvProgStudent` class inheriting from `AdvProgMember`:

```
class AdvProgStudent(AdvProgMember):  
    pass
```

Inheritance and subclasses

Creating instances of `AdvProgStudent` and accessing inherited methods:

```
student_1 = AdvProgStudent('Craig', 'Smith')
student_2 = AdvProgStudent('Megan', 'Scott')
print(student_1.full_name())
print(student_2.full_name())
```

Here, `AdvProgStudent` inherits methods like `full_name()` from `AdvProgMember`.

To fine-tune the `AdvProgStudent` class, we adjust attributes:

```
class AdvProgStudent(AdvProgMember):
    role = "AdvProg student"
```

Inheritance and subclasses

Now, creating a student instance reflects the updated role:

```
student_1 = AdvProgStudent('John', 'Smith')
print(student_1.role)
```

Adding an *instance attribute* like `grade` using `super()`, or the base class name:

```
class AdvProgStudent(AdvProgMember):
    role = "AdvProg student"

    def __init__(self, first, last, grade):
        # super().__init__(first, last) # Or the following:
        AdvProgMember.__init__(first, last)
        self.grade = grade

student_1 = AdvProgStudent('John', 'Smith', 28)
```

Inheritance and subclasses

Creating another subclass, `AdvProgInstructor`, with additional methods:

```
class AdvProgInstructor(AdvProgMember):
    role = "AdvProg instructor"

    def __init__(self, first, last, students=None):
        super().__init__(first, last)
        self.students = ([] if students is None else students)

    def add_student(self, student):
        self.students.append(student)

    def remove_course(self, student):
        self.students.remove(student)

instructor_1 = AdvProgInstructor('Pasquale', 'Africa')
instructor_1.add_student(student1)
instructor_1.add_student(student2)
instructor_1.remove_student(student1)
```

How inheritance works

To use inheritance, we specify the base class names in a tuple following the class name in the class definition (for example, `class Teacher(SchoolMember)`).

Next, we observe that the `__init__` method of the base class is explicitly called using the `self` variable so that we can initialize the base class part of an instance in the subclass. **This is very important to remember.**

Since we are defining a `__init__` method in `AdvProgStudent` and `AdvProgInstructor` subclasses, Python does not automatically call the constructor of the base class `AdvProgMember` , you have to explicitly call it yourself.

In contrast, if we have not defined an `__init__` method in a subclass, Python will call the constructor of the base class automatically.

Getters, setters, deleters

For effective class management, Python provides getters, setters, and deleters. Consider the former `AdvProgInstructor` class:

```
class AdvProgInstructor(AdvProgMember):
    role = "AdvProg instructor"

    # ...
```

Instances of `AdvProgMember` can be created and accessed:

```
advprog_1 = AdvProgMember('Pasquale', 'Africa') # Typo!
                                                ^^^^^^
print(advprog_1.first)
print(advprog_1.last)
print(advprog_1.email)
print(advprog_1.full_name())
```

The `@property` decorator (1/2)

Imagine that I mis-spelled the name of the first name and wanted to correct it. Watch what happens.

```
advprog_1.first = 'Pasquale'  
print(advprog_1.first)  
print(advprog_1.last)  
print(advprog_1.email) # Still prints pasqlae.africa@sissa.it!  
print(advprog_1.full_name())
```

Utilizing a `@property` decorator defines `email` like a method, but keeps it as an **attribute**:

```
class AdvProgMember:  
    # ...  
  
    @property  
    def email(self):  
        return self.first.lower() + "." + self.last.lower() + "@sissa.it"
```

The `@property` decorator (2/2)

Now, changes to the `first` name reflect in the `email`:

```
advprog_1 = AdvProgMember('Pasquale', 'Africa')
advprog_1.first = 'Pasquale'
print(advprog_1.first)
print(advprog_1.last)
print(advprog_1.email) # Now the correct value is printed.
print(advprog_1.full_name())
```

We could do the same with the `full_name()` method:

```
class AdvProgMember:
    # ...

    @property
    def full_name(self):
        return f'{self.first} {self.last}'
```

Setter methods (1/2)

Introducing a `full_name` setter to update `first` and `last`:

But what happens if we instead want to make a change to the full name now?

```
advprog_1.full_name = 'Pasquale Africa'
```

| AttributeError: can't set attribute

We get an error: class instance doesn't know what to do with the value it was passed. Ideally, we'd like our class instance to use this full name information to update `self.first` and `self.last`.

Setter methods (2/2)

To handle this action, we need a `setter`, defined using the decorator `@<attribute>.setter`:

```
class AdvProgMember:  
    # ...  
  
    @full_name.setter  
    def full_name(self, name):  
        first, last = name.split(' ')  
        self.first = first  
        self.last = last
```

Setting the `full_name` now updates the attributes:

```
advprog_1 = AdvProgMember('X', 'Y')  
advprog_1.full_name = 'Pasquale Africa'
```

Deleters

We've talked about getting information and setting information, but what about deleting information? This is typically used to do some clean up and is defined with the `@<attribute>.deleter` decorator.

```
class AdvProgMember:  
    # ...  
  
    @full_name.deleter  
    def full_name(self):  
        print('Name deleted!')  
        self.first = None  
        self.last = None
```

Deleting the `full_name` attribute results in a cleanup:

```
advprog_1 = AdvProgMember('Pasquale', 'Africa')  
delattr(advprog_1, "full_name")
```

Modules and packages

Modules

Modules: reusable code in Python

In Python, the ability to reuse code is facilitated by modules. A module is a file with a `.py` extension that contains functions and variables. There are various methods to write modules, including using languages like C to create compiled modules.

When importing a module, to enhance import performance, Python creates byte-compiled files (`__pycache__/filename.pyc`). These files, platform-independent and located in the same directory as the corresponding `.py` files, speed up subsequent imports by storing preprocessed code.

Using Standard Library Modules

You can import modules in your program to leverage their functionality. For instance, consider the `sys` module in the Python standard library. Below is an example:

```
# Example: module_using_sys.py
import sys

print("Command line arguments:", sys.argv)
```

When executed, this program prints the command line arguments provided to it. The `sys.argv` variable holds these arguments as a list. For instance, running `python module_using_sys.py we are arguments` results in `sys.argv[0]` being `'module_using_sys.py'`, `sys.argv[1]` being `'we'`, `sys.argv[2]` being `'are'`, and `sys.argv[3]` being `'arguments'`.

The `from... import...` Statement

You can selectively import variables from a module using the `from... import...` statement. However, it's generally advised to use the `import` statement to avoid potential name clashes and enhance readability.

```
from math import sqrt  
print("Square root of 16 is", sqrt(16))
```

A special case is `from math import *`, where all symbols exported by the `math` module are imported.

A module's `__name__`

Every module has a `__name__` attribute that indicates whether the module is being run standalone or imported. If `__name__` is '`__main__`', the module is being run independently.

```
# Example: module_using_name.py
if __name__ == '__main__':
    print("This module is being run independently.")
```

Creating your own modules

Creating modules is straightforward: every Python program is a module!

Save it with a `.py` extension. For example:

```
# Example: mymodule.py
def say_hi():
    print("Hello, this is mymodule speaking.")

__version__ = '1.0'
```

Now, you can use this module in another program:

```
# Example: mymodule_demo.py
import mymodule

mymodule.say_hi()
print("Version:", mymodule.__version__)
```

The `dir` function

The built-in `dir()` function lists all symbols defined in an object. For a module, it includes functions, classes, and variables. It can also be used without arguments to list names in the current module.

```
# Example: Using the dir function.  
import sys  
  
# Names in sys module.  
print("Attributes in sys module:", dir(sys))  
  
# Names in the current module.  
print("Attributes in current module:", dir())
```

Packages

Packages: organizing modules hierarchically

Packages are folders of modules with a special `__init__.py` file, indicating that the folder contains Python modules. They provide a hierarchical organization for modules.

```
<some folder in sys.path>/  
└── datascience/  
    ├── __init__.py  
    ├── preprocessing/  
    │   ├── __init__.py  
    │   ├── cleaning.py  
    │   └── scaling.py  
    └── analysis/  
        ├── __init__.py  
        ├── statistics.py  
        └── visualization.py
```

The `__init__.py` files (1/5)

The `__init__.py` file in a Python package serves multiple purposes. It's executed when the package or module is imported, and it can contain initialization code, set package-level variables, or define what should be accessible when the package is imported using `from package import *`.

Here are some common examples of using `__init__.py` files.

The `__init__.py` files (2/5)

1. Initialization code

```
# __init__.py in a package.

# Initialization code to be executed when the package is imported.
print("Initializing my_package...")

# Define package-level variables.
package_variable = 42

# Import specific modules when the package is imported.
from . import module1
from . import module2
```

In this example, the `__init__.py` file initializes the package, sets a package-level variable (`package_variable`), and imports specific modules from the package.

The `__init__.py` files (3/5)

2. Controlling `from package import *`

```
# __init__.py in a package.

# Define what should be accessible when a user writes 'from package import *'.
__all__ = ['module1', 'module2']

# Import modules within the package.
from . import module1
from . import module2
```

By specifying `__all__`, you explicitly control what is imported when using `from package import *`. It's considered good practice to avoid using `*` imports, but if you need to, this can help manage what gets imported.

The `.` symbol means that `module1.py` and `module2.py` are to be located in the same folder as the `__init__.py` file.

The `__init__.py` files (4/5)

3. Lazy loading

```
# __init__.py in a package.

# Initialization code.
print("Initializing my_lazy_package...")

# Import modules only when they are explicitly used.
def lazy_function():
    from . import lazy_module
    lazy_module.do_something()
```

In this example, the module is initialized only when the `lazy_function` is called. This can be useful for performance optimization, especially if some modules are rarely used.

The `__init__.py` files (5/5)

4. Setting package-level configuration

```
# __init__.py in a package.

# Configuration settings for the package.
config_setting1 = 'value1'
config_setting2 = 'value2'
```

You can use the `__init__.py` file to set package-level configuration settings that can be accessed by modules within the package.

Python modules as 1. scripts vs. 2. pre-compiled libraries

In Python, modules and packages can be implemented either as Python scripts or as pre-compiled dynamic libraries. Let's explore both concepts:

1. Python modules as scripts:

- **Extension:** Modules implemented as scripts usually have a `.py` extension.
- **Interpretation:** The Python interpreter reads and executes the script line by line.
- **Readability:** Scripts are human-readable and editable using a text editor.
- **Flexibility:** This is the most common form of Python modules. You can write and modify the code easily.
- **Portability:** Python scripts can be easily shared and run on any system with a compatible Python interpreter.

2. Python modules as dynamic libraries

- **Compilation:** Modules can be pre-compiled into shared libraries for performance optimization.
- **Execution:** The compiled code is loaded into memory and executed by Python.
- **Protection of intellectual property:** Pre-compiled modules can be used to distribute proprietary code without exposing the source.
- **Performance:** Pre-compiled modules may offer better performance as they are already in machine code.

It's essential to note that Python itself is an interpreted language, and even when using pre-compiled modules, the Python interpreter is still involved in executing the code. The use of pre-compiled modules is more about optimizing performance and protecting source code than altering the fundamental nature of Python as an interpreted language.

You can use tools like Cython or PyInstaller to generate pre-compiled modules or standalone executables, respectively, depending on your specific use case and requirements.

The Python Standard Library

Python's standard library is very extensive, offering a wide range of facilities as indicated by the long table of contents listed on the website. The library contains built-in modules (written in C) that provide access to system functionality such as file I/O that would otherwise be inaccessible to Python programmers, as well as modules written in Python that provide standardized solutions for many problems that occur in everyday programming. Some of these modules are explicitly designed to encourage and enhance the portability of Python programs by abstracting away platform-specifics into platform-neutral APIs.

In addition to the standard library, there is an active collection of hundreds of thousands of components (from individual programs and modules to packages and entire application development frameworks), available from the [Python Package Index](#).

Summary

Packages are a convenient way to organize modules hierarchically, often seen in the Python standard library.

In summary, modules and packages enhance code reusability in Python. The standard library showcases the power of these concepts, and creating your own modules and packages can significantly improve code organization and maintainability.

Next, we will delve into common Python packages for scientific computing, namely NumPy, SciPy, matplotlib/seaborn and pandas.

Zen of Python:
"Explicit is better than implicit."
Run `import this` in Python to learn more.

 **Introduction to NumPy and SciPy for scientific computing. Plotting.**
Introduction to pandas for data analysis.

Exercise session 10

Object-oriented programming. Classes, inheritance and polymorphism. Modules and packages.

Advanced Programming - SISSA, UniTS, 2023-2024

Pasquale Claudio Africa

07 Dec 2023

Exercise 1: generators for the solution of ODEs

Solving the differential equation $u' = -\sin(u)$ by applying the explicit Euler method results in the recursion:

$$u_{n+1} = u_n - h \sin(u_n).$$

1. Write a generator that computes the solution values u_n for a given initial value $u_0 = 1$ and a given value of the time step $h = 0.1$.
2. Implement a generator decorator `step_counter` that counts how many time steps have been performed.

Exercise 2: Polynomial class (1/3)

You are tasked with implementing a Python class called `Polynomial` that represents polynomials. The class should have the following features:

- 1. Constructor:** The class should have a custom constructor that takes variable coefficients as arguments. The coefficients should be provided in increasing order of degree ($a_0 + a_1x + \dots + a_nx^n$).
- 2. String representation:** Implement the `__repr__` method to provide a string representation of the polynomial. The string should display the polynomial in a human-readable form. For example, for the polynomial with coefficients `[1, 2, 3]`, the string representation should be
`"1 + 2x + 3x^2"`.
- 3. Addition and multiplication:** Implement the `__add__` and `__mul__` methods to allow addition and multiplication of polynomials. The methods should return a new polynomial.

Exercise 2: Polynomial class (2/3)

4. **Class method to create from string:** Implement a `@classmethod` called `from_string` that creates a `Polynomial` object from a string representation. Assume that the input string will be a polynomial in the form of `"a + bx + ... + cx^(n-1) + dx^n"`.

5. The base class `Polynomial` should be extended by two subclasses:

- `StandardPolynomialEvaluator` : Implements the standard polynomial evaluation method:

$$P(x) = a_0 + a_1 \cdot x + a_2 \cdot x^2 + \dots + a_n \cdot x^n$$

- `HornerPolynomialEvaluator` : Implements Horner's rule for polynomial evaluation:

$$P(x) = a_0 + x \cdot (a_1 + x \cdot (a_2 + \dots + x \cdot (a_{n-1} + x \cdot a_n) \dots))$$

Exercise 2: Polynomial class (3/3)

6. Implement a `measure_time` decorator, which measures the time taken by a function to execute.
7. Instantiate objects of both `StandardPolynomialEvaluator` and `HornerPolynomialEvaluator` with the same set of coefficients.
8. Apply the `measure_time` decorator to a function that takes a `PolynomialEvaluator` object and evaluates it at a given list of points.
9. Evaluate the polynomial at the same 1000 points using both methods and compare the results. Raise an assertion error if the results do not match.
10. Use the decorated function to evaluate the polynomial using both the standard method and Horner's rule, and observe the logged results and execution times.

Exercise 3: modular data processing package

Refactor the existing data processing code provided in `hints/ex3.py` into a modular package with multiple modules, functions, classes.

1. Refactoring: Refactor the code into a modular package `dataprocessor` with the following modules:

- `__init__.py` : Entry point for the package, import necessary functions, classes, and data. Implement `__all__`.
- `operations.py` : Contains functions for data processing and analysis.
- `data_analysis.py` : Introduce a class `DataAnalyzer` that encapsulates data processing and analysis functionalities.

2. Documentation: Provide docstrings for functions and classes. Explain the purpose and usage of each function and configuration option.

3. Test cases: Create a test `main.py` script to demonstrate the usage of the package.

Lecture 12

**Introduction to NumPy and SciPy for scientific computing.
Data visualization. Introduction to pandas for data analysis.**

Advanced Programming - SISSA, UniTS, 2023-2024

Pasquale Claudio Africa

12 Dec 2023

Outline

1. The role of Python in modern scientific computing

2. NumPy

- Creating and manipulating arrays
- Linear and matrix algebra
- Data processing and beyond

3. SciPy

- Relationship between NumPy and SciPy
- Core modules in SciPy

4. Data visualization

- Overview of Matplotlib for plotting
- Introduction to seaborn

5. pandas

- Dataframes
- Operations on dataframes

The role of Python in modern scientific computing

The role of Python in modern scientific computing

Python has emerged as a pivotal language in scientific computing, distinguished by:

- Intuitive and readable syntax, making coding accessible to scientists from various fields.
- A vast array of libraries and tools tailored for scientific applications.

Python's versatility extends across numerous scientific domains:

- In physics, it's used for simulations and theoretical calculations.
- In biology and chemistry, Python aids in molecular modeling and genomic data analysis.
- Its application in astronomy includes data processing from telescopes and space missions.
- In environmental science, it's pivotal in climate modeling and biodiversity studies.

Python's library ecosystem for scientific computing

The power of Python in scientific computing is amplified by its extensive library ecosystem:

- NumPy and SciPy are fundamental for numerical computations.
- pandas enhances data manipulation and analysis capabilities.
- Matplotlib and Seaborn excel in creating scientific visualizations.
- TensorFlow and PyTorch are at the forefront of machine learning research and applications.

Python's role in democratizing scientific research is underscored by its open-source nature, fostering collaboration and innovation.

Real-world applications of Python in scientific research

Python's impact in scientific research is evident through numerous real-world applications:

- In physics, it has been used to analyze data from the Large Hadron Collider.
- In biology, Python is integral in genome sequencing projects like the Human Genome Project.
- Environmental scientists utilize Python in modeling the effects of climate change on different ecosystems.
- In astronomy, it played a key role in processing the first image of a black hole.

These applications underscore Python's versatility and effectiveness in advancing scientific knowledge.

How to get your system ready

Most Python libraries can be installed with `pip`, with `Conda`, with a package manager on Linux and macOS, or from source.

- Using `pip`:

```
pip install numpy scipy matplotlib seaborn pandas
```

- Using `Conda`:

```
conda create -n sci-env  
conda activate sci-env  
conda install numpy scipy matplotlib seaborn pandas
```

Best practices in setting up a scientific computing environment include creating isolated environments and maintaining updated library versions.

NumPy

Introduction

NumPy is a core library for numerical computing in Python, offering an efficient interface for working with arrays and matrices. Known for its high performance, it forms the basis of many other scientific computing tools.

To get started with NumPy:

```
import numpy as np
```

NumPy arrays provide an efficient way to store and manipulate numerical data, offering advantages over traditional Python lists, particularly in terms of performance and functionality.

Creating NumPy arrays

There are several ways to create arrays in NumPy:

- Converting from Python lists or tuples.
- Using array-generating functions like `np.arange` , `np.linspace` , etc.
- Reading data from files.

From lists

Creating arrays from Python lists:

```
# Creating a vector from a list
vector = np.array([1, 2, 3, 4])

# Creating a matrix from a nested list
matrix = np.array([[1, 2], [3, 4]])
```

These arrays are instances of NumPy's `ndarray` type.

The shape of an array can be accessed using the `shape` attribute:

```
print(vector.shape) # Output: (4, )
print(matrix.shape) # Output: (2, 2)
```

Lists vs. NumPy arrays

NumPy arrays offer several advantages over Python lists, such as:

- Faster access in reading and writing items.
- More convenient and efficient for mathematical operations.
- Occupying less memory.

Unlike Python lists, NumPy arrays are statically typed, homogeneous, memory-efficient, and support efficient mathematical operations implemented in compiled languages like C and Fortran.

dtype (1/2)

The `dtype` (data type) property reveals the type of an array's data:

```
M.dtype
```

```
| dtype('int64')
```

Attempting to assign an incompatible type raises an error:

```
M[0, 0] = "hello"
```

```
| ValueError: invalid literal for long() with base 10: 'hello'
```

dtype (2/2)

Explicitly defining the array data type during creation is possible using the `dtype` keyword argument:

```
M = np.array([[1, 2], [3, 4]], dtype=complex)
```

Common data types for `dtype` include `int`, `float`, `complex`, `bool`, and others. Additionally, bit sizes like `int64`, `int16`, `float128`, and `complex128` can be specified.

Using array-generating functions (1/2)

NumPy provides various functions to generate arrays:

```
x = np.arange(0, 10, 1) # Arguments: start, stop, step.  
  
# Using linspace, where both end points are included.  
x = np.linspace(0, 10, 25)  
  
# Create an array with logarithmically spaced elements.  
x = np.logspace(0, 10, 10, base=np.e)  
  
x, y = np.mgrid[0:5, 0:5] # Similar to 'meshgrid' in MATLAB.
```

Using array-generating functions (2/2)

Additional array-generating functions include random number generation, diagonal matrices, zeros, and ones:

```
np.random.rand(5, 5) # Uniform random numbers in [0, 1].  
np.random.randn(5, 5) # Standard normal distributed random numbers.  
np.diag([1, 2, 3]) # Diagonal matrix.  
np.diag([1, 2, 3], k=1) # Diagonal with an offset from the main diagonal.  
np.zeros((3, 3))  
np.ones((3, 3))
```

File I/O (1/2)

Comma-separated values (CSV)

Reading data from comma-separated values (CSV) files into NumPy arrays is accomplished using

`np.genfromtxt` :

```
data = np.genfromtxt('filename.csv')
data.shape
```

Storing a NumPy array to a CSV file can be done with `np.savetxt` :

```
M = random.rand(3, 3)
np.savetxt("random-matrix.csv", M)
np.savetxt("random-matrix.csv", M, fmt='%.5f') # fmt specifies the format.
```

File I/O (2/2)

NumPy's native file format

NumPy provides its own file format for storing and reading array data using `np.save` and `np.load`:

```
np.save("random-matrix.npy", M)  
np.load("random-matrix.npy")
```

Manipulating arrays

Indexing

Array elements are accessed using square brackets and indices for reading and writing:

```
# v is a vector, taking one index.  
v[0]  
  
# M is a matrix, taking two indices.  
M[1, 1]  
  
# Omitting an index returns the whole row or a N-1 dimensional array.  
M[1]  
  
M[1, :] # Row 1.  
M[:, 1] # Column 1.
```

Index slicing

Index slicing (`M[lower:upper:step]`) extracts portions of an array:

```
A = np.array([1, 2, 3, 4, 5])
A[1:3]
A[0:2] = [-2, -3]

A[::-1] # Lower, upper, and step default to the beginning, end, and 1.
A[::-2] # Step is 2, lower and upper default to the beginning and end.
A[:3] # First three elements.
A[3:] # Elements from index 3.

# Negative indices count from the end of the array.
A[-1] # The last element in the array.
A[-3:] # The last three elements.
```

Fancy indexing

Fancy indexing involves using an array or list in place of an index:

```
row_indices = [1, 2, 3]
A[row_indices]
```

```
col_indices = [1, 2, -1]
A[row_indices, col_indices]
```

Index masks, arrays of type `bool`, can also be used to select elements:

```
mask = np.array([True, False, True])
A[mask]
```

```
x = np.arange(0, 10, 0.5)
mask = (5 < x) * (x < 7.5)
x[mask]
```

Linear algebra (1/2)

NumPy is well-suited for linear algebra operations:

- Scalar-array and element-wise array-array operations.
- Matrix multiplication using the `dot` function or `@` operator.
- Computing inverses, determinants, and solving linear equations.

Scalar-array operations

Arithmetic operators are employed for scalar-array operations:

```
v1 = np.arange(0, 5)
v1 * 2
v1 + 2
A * 2
A +2
```

Linear algebra (2/2)

Element-wise array-array operations

When we add, subtract, multiply and divide arrays with each other, the default behavior is **element-wise** operations:

```
A * A  
v1 * v1
```

If we multiply arrays with compatible shapes, we get an element-wise multiplication of each row:

```
A * v1
```

Matrix algebra (1/2)

What about matrix multiplication? There are two ways. We can either use the `@` or `dot` function, which applies a matrix-matrix, matrix-vector, or inner vector multiplication to its arguments:

```
A @ A  
np.dot(A, A)  
np.dot(A, v1)  
np.dot(v1, v1)
```

Alternatively, we can cast the array objects to the type `matrix`. This changes the behavior of the standard arithmetic operators `+`, `-`, `*` to use matrix algebra.

```
M = np.matrix(A)  
v = np.matrix(v1).T # Make it a column vector.  
M * M  
v.T * v, v + M * v
```

Matrix algebra (2/2)

Inverse and determinant

```
np.linalg.inv(M) # Same as M.I.  
M.I * M
```

```
np.linalg.det(M)  
np.linalg.det(M.I)
```

Data processing

NumPy provides various functions to calculate statistics of datasets in arrays. Here are some examples:

```
np.mean(data[:, 3])
np.std(data[:, 3]), np.var(data[:, 3])
np.min(data[:, 3])
np.max(data[:, 3])

np.sum(d) # Sum up all elements.
np.prod(d + 1) # Product of all elements.
np.cumsum(d) # Cumulative sum.
np.cumprod(d + 1) # Cumulative product.

np.trace(A) # Same as: np.diag(A).sum()
```

Calculations with higher-dimensional data

When functions such as `min`, `max`, etc. are applied to multidimensional arrays, it is sometimes useful to apply the calculation to the entire array or on a row or column basis. Using the `axis` argument, we can specify how these functions should behave:

```
M = random.rand(3, 3)

m.max()      # Global max.
m.max(axis=0) # Max in each column.
m.max(axis=1) # Max in each row.
```

Many other functions and methods in the `array` and `matrix` classes accept the same (optional) `axis` keyword argument.

Reshaping, resizing, and stacking arrays

The shape of a NumPy array can be modified without copying the underlying data, making it a fast operation even for large arrays:

```
n, m = A.shape
B = A.reshape((1, n * m))

B[0, 0:5] = 5 # Modify the array.
# The original variable is also changed. B is only a different view of the same data.
```

We can also use the function `flatten` to make a higher-dimensional array into a vector. But this function creates a copy of the data:

```
B = A.flatten()
B[0:5] = 10
```

Adding a new dimension: `newaxis`

With `newaxis`, we can insert new dimensions in an array, for example converting a vector to a column or row matrix:

```
v = np.array([1, 2, 3])
np.shape(v)
```

```
(3,)
```

```
# Make a column matrix of the vector v.
v[:, np.newaxis]
```

```
# Make a row matrix of the vector v.
v[np.newaxis, :]
```

Stacking and repeating arrays (1/2)

Using functions `repeat`, `tile`, `vstack`, `hstack`, and `concatenate`, we can create larger vectors and matrices from smaller ones:

```
a = np.array([[1, 2], [3, 4]])
```

```
# Repeat each element 3 times.  
np.repeat(a, 3)
```

```
array([1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4])
```

```
# Tile the matrix 3 times.  
np.tile(a, 3)
```

```
array([[1, 2, 1, 2, 1, 2], [3, 4, 3, 4, 3, 4]])
```

Stacking and repeating arrays (2/2)

```
b = np.array([[5, 6]])  
  
# Add a new row.  
np.concatenate((a, b), axis=0)  
# Same as:  
np.vstack((a, b))  
  
# Add a new column.  
np.concatenate((a, b.T), axis=1)  
# Same as:  
np.hstack((a, b.T))
```

Reference vs. deep copy

To achieve high performance, assignments in Python usually do not copy the underlying objects. This is important, for example, when objects are passed between functions to avoid an excessive amount of memory copying when it is not necessary:

```
A = np.array([[1, 2], [3, 4]])  
  
B = A # B is referring to the same array data as A.  
B[0, 0] = 10 # Changing B affects A.  
  
C = np.copy(A) # Deep copy.  
C[0, 0] = -5 # If we modify C, A is not affected.
```

Iterating over array elements (1/2)

Generally, we want to avoid iterating over the elements of arrays whenever possible. The Python `for` loop is the most convenient way to iterate over an array when necessary:

```
v = np.array([1, 2, 3, 4])
for element in v:
    print(element)

M = np.array([[1, 2], [3, 4]])
for row in M:
    print("row", row)
    for element in row:
        print(element)
```

Iterating over array elements (2/2)

When we need to iterate over each element of an array and modify its elements, it is convenient to use the `enumerate` function to obtain both the element and its index in the `for` loop:

```
for row_idx, row in enumerate(M):
    print("row_idx", row_idx, "row", row)

    for col_idx, element in enumerate(row):
        print("col_idx", col_idx, "element", element)

    # Update the matrix M: square each element.
    M[row_idx, col_idx] = element ** 2
```

Vectorizing functions (1/2)

As mentioned several times, to achieve good performance, we should avoid looping over elements in our vectors and matrices and instead use vectorized algorithms. The first step is to make sure that functions work with vector inputs:

```
def Heaviside(x):
    """
    Scalar implementation of the Heaviside step function.
    """
    if x >= 0:
        return 1
    else:
        return 0

Heaviside(np.array([-3, -2, -1, 0, 1, 2, 3]))
```

ValueError: The truth value of an array with more than one element is ambiguous. Use a.any() or a.all()

Vectorizing functions (2/2)

Implement the function to accept a vector input from the beginning:

```
def Heaviside(x):
    """
    Vector-aware implementation of the Heaviside step function.
    """
    return 1 * (x >= 0)

Heaviside(np.array([-3, -2, -1, 0, 1, 2, 3]))
```

See also the NumPy function `vectorize`.

Using arrays in conditions

When using arrays in conditions, for example `if` statements and other boolean expressions, use `any` or `all`, requiring that any or all elements in the array evaluate to `True`:

```
if (M > 5).any():
    print("At least one element in M is larger than 5.")
```

```
if (M > 5).all():
    print("All elements in M are larger than 5.")
```

Type casting

Since NumPy arrays are *statically typed*, the type of an array does not change once created. But we can explicitly cast an array of some type to another using the `astype` functions (see also the similar `asarray` function). This always creates a new array of new type:

```
M.dtype
```

```
| dtype('int64')
```

```
M2 = M.astype(float)
M3 = M.astype(bool)
```

Further reading

- NumPy documentation
- NumPy tutorials

SciPy

SciPy overview

SciPy, built on NumPy's foundation, offers higher-level scientific algorithms and modules for specialized tasks like optimization, integration, signal processing, and linear algebra. It provides seamless interoperability with NumPy for efficient data representation and manipulation.

Relationship between NumPy and SciPy:

- NumPy serves as the foundation for numerical operations in SciPy.
- SciPy utilizes NumPy arrays for efficient data representation.
- Seamless interoperability between NumPy and SciPy.

Best Practice: *NumPy for basic operations, SciPy for specialized tasks.*

SciPy modules

- `scipy.special` for special functions like Bessel or gamma functions.
- `scipy.integrate` for numerical integration and solving differential equations.
- `scipy.optimize` for optimization algorithms.
- `scipy.interpolate` for interpolating functions and data.
- `scipy.fft` for Fourier transforms.
- `scipy.signal` for signal processing tools.
- `scipy.sparse` for sparse matrices and related algorithms.
- `scipy.linalg` for advanced linear algebra operations.
- `scipy.stats` for statistical distributions and functions.
- `scipy.ndimage` for multi-dimensional image processing.
- `scipy.io` for file I/O operations.

Importing SciPy

In this lecture, we will explore how to use some of these subpackages.

To access the SciPy package in a Python program, we start by importing everything from the `scipy` module:

```
from scipy import *
```

If we only need to use part of the SciPy framework, we can selectively include only those modules we are interested in. For example, to include the linear algebra package under the name `la`, we can do:

```
import scipy.linalg as la
```

Numerical integration (1/3)

For the numerical evaluation of a definite integral of the type $\int_a^b f(x), dx$, SciPy provides a series of functions for different kinds of quadrature, such as `quad`, `dblquad`, and `tplquad` for single, double, and triple integrals, respectively.

```
from scipy.integrate import quad, dblquad, tplquad
```

The `quad` function takes a large number of optional arguments, which can be used to fine-tune the behavior of the function (try `help(quad)` for details).

Numerical integration (2/3)

The basic usage is as follows:

```
def f(x):
    return x

x_lower = 0
x_upper = 1

val, abserr = quad(f, x_lower, x_upper)
```

For simple functions, we can use a lambda function:

```
val, abserr = quad(lambda x: exp(-x ** 2), -Inf, Inf)
```

Numerical integration (3/3)

Higher-dimensional integration works in the same way:

```
def integrand(x, y):
    return exp(-x**2 - y**2)

x_lower = 0
x_upper = 10
y_lower = lambda x: x
y_upper = lambda x: x + 1

val, abserr = dblquad(integrand, x_lower, x_upper, y_lower, y_upper)
```

Ordinary Differential Equations (ODEs) (1/2)

A system of ODEs is usually formulated in standard form before it is attacked numerically. The standard form is

$$y' = f(y, t),$$

where $y = [y_1(t), y_2(t), \dots, y_n(t)]$, and f is some function that gives the derivatives of the function $y_i(t)$.

To solve an ODE, we need to know the function f and an initial condition, $y(0)$.

Note that higher-order ODEs can always be written in this form by introducing new variables for the intermediate derivatives.

Ordinary Differential Equations (ODEs) (2/2)

SciPy provides two different ways to solve ODEs: an API based on the function `odeint` and an object-oriented API based on the class `ode`. Usually, `odeint` is easier to get started with, but the `ode` class offers some finer level of control. Here we will use the `odeint` functions. For more information about the class `ode`, try `help(ode)`.

To use `odeint`, first import it from the `scipy.integrate` module

```
from scipy.integrate import odeint, ode
```

Once we have defined the Python function `f` and array `y_0`, we can use `odeint` as:

```
y_t = odeint(f, y_0, t)
```

where `t` is an array with time-coordinates for which to solve the ODE problem. `y_t` is an array with one row for each point in time in `t`, where each column corresponds to a solution $y_i(t)$ at that point in time.

Fourier transform

SciPy's `fft` module enables easy computation of Fourier transforms, a critical tool in computational physics, signal processing and data analysis.

```
from scipy.fft import fft, ifft, fftfreq
import numpy as np

N = 600 # Number of sample points.

T = 1.0 / 800.0 # Sample spacing.

x = np.linspace(0.0, N*T, N, endpoint=False)
y = np.sin(50.0 * 2.0*np.pi*x) + 0.5*np.sin(80.0 * 2.0*np.pi*x)

yf = fft(y)
xf = fftfreq(N, T)[:N//2]
```

Linear algebra

The linear algebra module contains various matrix-related functions, including linear equation solving, eigenvalue solvers, matrix functions (e.g., matrix exponentiation), decompositions (SVD, LU, Cholesky), etc.

Linear systems

```
from scipy.linalg import *
A = array([[1,2,3], [4,5,6], [7,8,9]])
b = array([1,2,3])
x = solve(A, b)
```

Linear algebra

Eigenvalues and eigenvectors

The eigenvalue problem for a matrix A reads $Av_n = \lambda_n v_n$, where v_n is the n th eigenvector and λ_n is the n th eigenvalue.

To calculate eigenvalues of a matrix, use the `eigvals`, and for calculating both eigenvalues and eigenvectors, use the function `eig`:

```
lambda = eigvals(A)
lambda, v = eig(A)
```

The eigenvectors corresponding to the n th eigenvalue (stored in `lambda[n]`) is the n th *column* in `v`, i.e., `v[:, n]`.

There are also more specialized eigensolvers, like the `eigh` for Hermitian matrices.

Linear algebra

Matrix operations

```
inv(A) # Matrix inverse.  
det(A) # Matrix determinant.  
  
# Matrix norms of various orders.  
norm(A, ord=1)  
norm(A, ord=2)  
norm(A, ord=Inf)
```

Sparse matrices (1/3)

Sparse matrices are often useful in numerical simulations dealing with large systems, where the problem can be described in matrix form, and matrices or vectors mostly contain zeros. SciPy has good support for sparse matrices, with basic linear algebra operations (e.g., equation solving, eigenvalue calculations, etc.).

There are many possible strategies for storing sparse matrices efficiently, such as coordinate form (COO), list of lists (LIL) form, and compressed-sparse column CSC (and row, CSR). Each format has some advantages and disadvantages. Most computational algorithms (equation solving, matrix-matrix multiplication, etc.) can be efficiently implemented using CSR or CSC formats, but they are not so intuitive and not so easy to initialize. So often, a sparse matrix is initially created in COO or LIL format (where we can efficiently add elements to the sparse matrix data), and then converted to CSC or CSR before used in real calculations.

Sparse matrices (2/3)

When we create a sparse matrix, we have to choose which format it should be stored in. For example,

```
from scipy.sparse import *

# Dense matrix.
M = array([[1,0,0,0], [0,3,0,0], [0,1,1,0], [1,0,0,1]])

# Convert from dense to sparse.
A = csr_matrix(M)

# Convert from sparse to dense.
A.todense()
```

Sparse matrices (3/3)

More efficient way to create sparse matrices: create an empty matrix and populate it using matrix indexing (avoids creating a potentially large dense matrix).

```
A = lil_matrix((4,4)) # Empty 4x4 sparse matrix.  
A[0,0] = 1  
A[1,1] = 3  
A[2,2] = A[2,1] = 1  
A[3,3] = A[3,0] = 1  
  
# Sparse matrix - dense array multiplication.  
A * v
```

LIL stands for *L*sts of *L*ists.

Optimization

Optimization (finding minima or maxima of a function) is a large field in mathematics, and optimization of complicated functions or in many variables can be rather involved.

Finding minima

We can use several algorithms to find the minima of a function:

```
from scipy import optimize

def f(x):
    return 4*x***3 + (x-2)**2 + x***4

x_min1 = optimize.fmin_bfgs(f, -2)
x_min2 = optimize.brent(f)
x_min3 = optimize.fminbound(f, -4, 2)
```

Optimization

Finding function roots

To find the root for a function of the form $f(x) = 0$, we can use the `fsolve` function. It requires an initial guess:

```
from scipy import optimize

def f(omega):
    return tan(2 * np.pi * omega) - 3.0 / omega

optimize.fsolve(f, 0.1)
```

Interpolation

The `interp1d` function, when given arrays describing X and Y data, returns an object that behaves like a function that can be called for an arbitrary value of x (in the range covered by X), and it returns the corresponding interpolated y value:

```
from scipy.interpolate import *

def f(x):
    return sin(x)

n = arange(0, 10)
x_meas = linspace(0, 9, 100)
y_meas = f(n) + 0.1 * randn(len(n))

linear_interpolation = interp1d(x_meas, y_meas)
y_interp1 = linear_interpolation(x)

cubic_interpolation = interp1d(x_meas, y_meas, kind='cubic')
y_interp2 = cubic_interpolation(x)
```

Statistics

The `scipy.stats` module contains a large number of statistical distributions, statistical functions, and tests.

```
from scipy import stats  
  
X = stats.poisson(3.5)  
Y = stats.norm()  
  
X.mean(), X.std(), X.var()  
Y.mean(), Y.std(), Y.var()
```

Statistical tests

SciPy includes functions for conducting statistical tests, like t-tests and ANOVA, aiding in hypothesis testing and data analysis.

Calculate the T-test for the means of two independent samples:

```
t_statistic, p_value = stats.ttest_ind(X.rvs(size=1000), X.rvs(size=1000))
```

Test if the mean of a single sample of data is 0.1:

```
stats.ttest_1samp(Y.rvs(size=1000), 0.1)
```

A low p-value means that we can reject the hypothesis.

Further reading

- SciPy documentation

Data visualization

Introduction to Matplotlib and Seaborn

Data visualization plays a vital role in data analysis, enabling the effective communication of complex information. Matplotlib and Seaborn are two widely-used Python libraries for data visualization. Matplotlib offers a wide range of plotting tools, while Seaborn provides a high-level interface for drawing attractive statistical graphics.

Matplotlib (1/2)

Matplotlib is a widely-used 2D plotting library in Python. It provides a high-level interface for drawing attractive and informative statistical graphics. Let's start with a simple example to create a basic line plot:

```
import matplotlib.pyplot as plt
import numpy as np

# Generate data.
x = np.linspace(0, 10, 100)
y = np.sin(x)

# Create a simple line plot.
plt.plot(x, y, label='sin(x)')
plt.title('Simple line plot')
plt.xlabel('x-axis')
plt.ylabel('y-axis')
plt.legend()
plt.show()
```

Matplotlib (2/2)

In this example, we use NumPy to generate data points for the x-axis and calculate corresponding y-values. The `plot` function is then used to create a line plot. Finally, `title`, `xlabel`, `ylabel`, and `legend` functions are used to add a title, axis labels, and a legend to the plot.

Matplotlib supports various types of plots, including scatter plots, bar plots, histograms, and more. Explore the documentation for more plot types and customization options.

2D plots with Matplotlib

Let's create a 2D contour plot using Matplotlib.

```
import matplotlib.pyplot as plt
import numpy as np

# Generate data.
x = np.linspace(-5, 5, 100)
y = np.linspace(-5, 5, 100)
X, Y = np.meshgrid(x, y)
Z = np.sin(np.sqrt(X**2 + Y**2))

# Create a 2D contour plot.
plt.contourf(X, Y, Z, cmap='viridis')
plt.colorbar(label='sin(sqrt(x^2 + y^2))')
plt.title('Contour plot')
plt.xlabel('x-axis')
plt.ylabel('y-axis')
plt.show()
```

Seaborn

Seaborn, built on Matplotlib, provides a more intuitive interface for creating statistical plots. It integrates well with pandas data structures and offers built-in themes for enhanced visual appeal.

```
import seaborn as sns
import numpy as np

# Generate data.
x = np.random.randn(100)
y = 2 * x + np.random.randn(100)

# Create a scatter plot using Seaborn.
sns.scatterplot(x=x, y=y, color='blue')
plt.title('Scatter Plot with Seaborn')
plt.xlabel('x-axis')
plt.ylabel('y-axis')
plt.show()
```

Customizing histograms with Seaborn

Let's create a customized histogram with Seaborn, including specific bin edges, colors, and additional statistical annotations.

```
import seaborn as sns
import matplotlib.pyplot as plt

# Load a dataset.
tips = sns.load_dataset('tips')

# Create a histogram with customizations.
sns.histplot(tips['total_bill'], bins=[10, 20, 30, 40, 50], color='salmon')
plt.title('Customized histogram')
plt.xlabel('Total bill ($)')
plt.ylabel('Frequency')

# Annotate with mean and median.
plt.axvline(tips['total_bill'].mean(), color='blue', linestyle='dashed', linewidth=2, label='Mean')
plt.axvline(tips['total_bill'].median(), color='green', linestyle='dashed', linewidth=2, label='Median')

plt.legend()
plt.show()
```

Scatter plots with Seaborn

Create a scatter plot with Seaborn that includes a regression line, different colors based on a categorical variable, and markers with varied sizes.

```
import seaborn as sns
import matplotlib.pyplot as plt

# Load a dataset.
tips = sns.load_dataset('tips')

# Create a scatter plot.
sns.scatterplot(x='total_bill', y='tip', hue='day', size='size', sizes=(20, 200),
                 data=tips, palette='Set2', alpha=0.8)
plt.title('Advanced Scatter Plot')
plt.xlabel('Total bill ($)')
plt.ylabel('Tip ($)')
plt.legend(title='Day')
plt.show()
```

Combining Matplotlib and Seaborn

One of the strengths of Seaborn is its ability to work seamlessly with Matplotlib. You can use Matplotlib functions alongside Seaborn to customize your plots further. Here's an example combining Matplotlib and Seaborn to create a histogram with a kernel density estimate:

```
import seaborn as sns
import matplotlib.pyplot as plt

# Load a dataset.
tips = sns.load_dataset('tips')

# Create a histogram with a Kernel Density Estimate using Seaborn.
sns.histplot(tips['total_bill'], kde=True, color='skyblue')

# Customize Matplotlib features.
plt.title('Histogram with KDE')
plt.xlabel('Total bill ($)')
plt.ylabel('Frequency')
plt.show()
```

Advanced plotting

Matplotlib supports advanced plots like contour plots, 3D plots, and subplots:

- Contour plots for visualizing three-dimensional data.
- Subplots for displaying multiple plots in a single figure.

Seaborn excels in creating complex statistical plots:

- Heatmaps for representing matrix data.
- Pair plots for exploring relationships in a dataset.
- Facet grids for plotting subsets of data on multiple axes.

Matplotlib and Seaborn can directly plot from pandas DataFrame, simplifying the workflow in data analysis tasks.

Further reading

- Matplotlib documentation
- Seaborn documentation

pandas

pandas overview

pandas is a powerful Python library for data manipulation and analysis. It provides fast, flexible data structures like Series and DataFrame, designed to work with structured data intuitively and efficiently.

In the pandas library, the standard import convention involves using the aliases `np` for NumPy and `pd` for pandas:

```
import numpy as np  
import pandas as pd
```

Fundamental data structures in pandas

- **Series**: A one-dimensional labeled array that can hold various data types.
- **DataFrame**: A two-dimensional, size-mutable, and potentially heterogeneous tabular data structure with labeled axes (rows and columns).

Creating objects (1/2)

Series creation

You can create a Series by providing a list of values. pandas will generate a default RangeIndex:

```
s = pd.Series([1, 3, 5, np.nan, 6, 8])
```

DataFrame creation

Creating a DataFrame involves passing a NumPy array with a datetime index and labeled columns:

```
dates = pd.date_range("20130101", periods=6)
df = pd.DataFrame(np.random.randn(6, 4), index=dates, columns=list("ABCD"))
```

Creating objects (2/2)

DataFrame creation

Alternatively, a DataFrame can be formed from a dictionary of objects:

```
df2 = pd.DataFrame(  
{  
    "A": 1.0,  
    "B": pd.Timestamp("20130102"),  
    "C": pd.Series(1, index=list(range(4)), dtype="float32"),  
    "D": np.array([3] * 4, dtype="int32"),  
    "E": pd.Categorical(["test", "train", "test", "train"]),  
    "F": "foo",  
})
```

The resulting DataFrame has diverse data types.

Viewing data

To examine the top and bottom rows of a DataFrame, use `head()` and `tail()`:

```
df.head()  
df.tail(3)
```

Retrieve the DataFrame's index or column labels:

```
df.index  
df.columns  
  
df.to_numpy() # Convert the DataFrame to a NumPy array.
```

Note: NumPy arrays have one dtype for the entire array while pandas DataFrames have one dtype per column. When you call `DataFrame.to_numpy()`, pandas will find the NumPy dtype that can hold *all* of the dtypes in the DataFrame. If the common data type is `object`, `DataFrame.to_numpy` will require copying data.

Data selection (1/3)

pandas offers various methods for data selection. We'll explore both label-based and position-based approaches.

Getitem ([])

For a **DataFrame**, passing a single label selects a column and yields a **Series** equivalent to

```
df.A :
```

```
df["A"]
```

For a **DataFrame**, passing a slice `:` selects matching rows:

```
df[0:3]  
df["20130102":"20130104"]
```

Data selection (2/3)

Label-based selection

Use `loc` and `at` for label-based indexing:

```
df.loc[dates[0]] # Selecting a row by label.  
df.loc[:, ["A", "B"]] # Selecting all rows for specific columns.  
df.loc["20130102":"20130104", ["A", "B"]] # Both endpoints are included.  
df.at[dates[0], "A"] # Fast scalar access.
```

Position-based selection

For position-based indexing, employ `iloc` and `iat`:

```
df.iloc[3] # Selecting via position.  
df.iloc[3:5, 0:2] # Slicing rows and columns.  
df.iat[1, 1] # Fast scalar access.
```

Data selection (3/3)

Selecting values from a **DataFrame** where a boolean condition is met:

```
df[df > 0]
```

Select rows based on a condition:

```
df[df["A"] > 0]
```

Use `Series.isin` method for filtering:

```
df2 = df.copy()
df2["E"] = ["one", "one", "two", "three", "four", "three"]
df2[df2["E"].isin(["two", "four"])]
```

Viewing and sorting data

Generate quick statistics using `describe()`:

```
df.describe()
```

Transpose the DataFrame with `.T`:

```
df.T
```

Sort the DataFrame by index or values:

```
df.sort_index(axis=1, ascending=False) # Sort by index.  
df.sort_values(by="B") # Sort by values.
```

Operations (1/3)

Statistics

Compute the mean for each column or row:

```
df.mean()  
df.mean(axis=1)
```

Operations with Series or DataFrame

Perform operations with another Series or DataFrame:

```
s = pd.Series([1, 3, 5, np.nan, 6, 8], index=dates).shift(2)  
df.sub(s, axis="index")
```

Operations (2/3)

User-defined functions

Apply user-defined functions using `agg` and `transform`:

```
df.agg(lambda x: np.mean(x) * 5.6)
df.transform(lambda x: x * 101.2)
```

Value counts

Compute value counts for a Series:

```
s = pd.Series(np.random.randint(0, 7, size=10))
s.value_counts()
```

Operations (3/3)

pandas simplifies handling missing data using methods like `dropna()`, `fillna()`, and `interpolate()`.

Transform data using operations like pivoting, melting, and applying custom functions.

pandas excels in time series data analysis, offering functionalities for resampling, shifting, and window operations.

pandas supports categorical data types, which can be more efficient and expressive for certain types of data.

Plotting

pandas integrates with Matplotlib for easy data visualization. Here's a basic example:

```
import matplotlib.pyplot as plt

ts = pd.Series(np.random.randn(1000), index=pd.date_range("1/1/2000", periods=1000))
ts = ts.cumsum()

ts.plot()
plt.show()
```

Importing and exporting data

pandas can read and write to various file formats, including CSV, Excel, SQL, and more.

CSV

```
df = pd.DataFrame(np.random.randint(0, 5, (10, 5)))
df.to_csv("foo.csv")
pd.read_csv("foo.csv")
```

Excel

```
df.to_excel("foo.xlsx", sheet_name="Sheet1")
pd.read_excel("foo.xlsx", "Sheet1", index_col=None, na_values=["NA"])
```

Further reading

- pandas documentation
- 10 minutes to pandas



Integrating C++ and Python codes.

Exercise session 11

**Introduction to NumPy and SciPy for scientific computing.
Data visualization. Introduction to pandas for data analysis.**

Advanced Programming - SISSA, UniTS, 2023-2024

Pasquale Claudio Africa

14 Dec 2023

Exercise 1: NumPy

1. Array creation and manipulation

- i. Create a 2D NumPy array of shape 5×5 filled with random integers between 1 and 10.
- ii. Extract the second row, third column element, and the diagonal elements.
- iii. Reshape it into a 1D array of shape 1×25 .

2. Linear algebra operations

- i. Generate two 3×3 matrices with random integers from 1 to 10 and perform element-wise and matrix-matrix multiplication.
- ii. Create a 3×3 matrix with random values, compute its inverse and determinant.

3. Statistical analysis

- i. Generate a 1D NumPy array with 20 random integers between 1 and 100.
- ii. Calculate the mean, median, standard deviation, and variance.

Exercise 2: SciPy (1/2)

1. Solving a linear system of equations

- i. Define a 100×100 sparse tridiagonal matrix A , with 2 over the main diagonal, and -1 over the first lower and upper diagonals.
- ii. Let $b = Ax_{\text{ex}}$ where $x_{\text{ex}} = [1, 1, \dots, 1]^T \in \mathbb{R}^{100}$
- iii. Solve the linear system $Ax = b$ and compute the residual $\|b - Ax\|$ and the error $\|x - x_{\text{ex}}\|$ in norm 1, 2 and infinity.

2. Function optimization

- i. Consider the function $f(x) = \sin(\pi x) \exp(-x/10)$ over the interval $[-2, 4]$.
- ii. Plot the function using Matplotlib to visually identify potential minima.
- iii. Use `scipy.optimize.minimize` with different initial guesses to find these minima.

Exercise 2: SciPy (2/2)

3. Data interpolation and integration

- i. An electric vehicle charging station erogates the following series of energy measurements over time:

```
time = np.arange(0, 46, 3) # Hours.  
energy = np.array([27.29, 23.20, 24.93, 28.72, 27.60, 19.06, 24.85, 21.54, 21.69, 23.23, 22.43, 26.36, 24.28, 22.36, 23.33, 23.00]) # kW.
```

- ii. Use SciPy to build a cubic interpolator of these data points.
- iii. Evaluate the interpolator over 1000 equispaced nodes between 0 and 45 and plot the values obtained.
- iv. Integrate the interpolant over $(0, 45)$.

Exercise 3: pandas (1/2)

1. DataFrame operations and visualization

- i. Import the `sales_data.txt` dataset as a pandas DataFrame.
- ii. Extract data from the 'South' region, sort them by descending 'Quantity' and add a new column 'Total revenue' = 'Quantity' × 'Price'.
- iii. Visualize trends of 'Total revenue' by 'Date' (line plot) and by 'Product' (bar plot).

2. Exploratory data analysis with the *iris* dataset

- i. Load the `iris` dataset from seaborn.
- ii. Group the data by 'species' and compute summary statistics for `sepal_length` and `sepal_width`.
- iii. Use seaborn to plot the histogram of the sepal length distribution for each species.
- iv. Use seaborn to generate a scatter plot of sepal width vs. sepal length.

Exercise 3: pandas (2/2)

3. Time series analysis with real data

- i. Import the `weather_data.txt` dataset.
- ii. Resample the dataset to compute monthly averages.
- iii. Computing a 7-day rolling mean.
- iv. Visualize the original data and the rolling mean using line plots.

Lecture 13

Integrating C++ and Python codes.

Advanced Programming - SISSA, UniTS, 2023-2024

Pasquale Claudio Africa

19 Dec 2023

Outline

1. Integrating C++ and Python

2. pybind11

- Installation
- Basics
- Binding object-oriented code

3. How to build and import pybind11 modules

4. Examples

Part of these lecture notes and examples is re-adapted from [the official pybind11 documentation](#) ([license](#)) and from [this repository](#) ([license](#)).

Integrating C++ and Python

Why integrating C++ and Python code

C++ and Python are powerful in their own right, but they excel in different areas. C++ is renowned for its performance and control over system resources, making it ideal for CPU-intensive tasks and systems programming. Python, on the other hand, is celebrated for its simplicity, readability, and vast ecosystem of libraries, especially in data science, machine learning, and web development.

- In research areas like machine learning, scientific computing, and data analysis, the need for processing speed and efficient resource management is critical.
- The industry often requires solutions that are both efficient and rapidly developed.

By integrating C++ with Python, you can create applications that harness the raw power of C++ and the versatility and ease-of-use of Python. Python, despite its popularity in these fields, often falls short in terms of performance. Knowledge of how to integrate C++ and Python equips with a highly valuable skill set.

Common libraries for C++ and Python integration (1/5)

Integrating C++ and Python code is a common need in software development, especially when you want to combine the high performance of C++ with the ease of use of Python. Several libraries are available for this purpose, each with its own set of advantages and drawbacks. Here are some of the most commonly used libraries:

1. Boost.Python

- Pros:
 - Well-documented, widely used.
 - Seamless interoperability between C++ and Python.
 - Exposes C++ classes to Python and vice versa.
- Cons:
 - Complex setup for beginners.
 - Larger binary size.
 - Slower compile times.

Common libraries for C++ and Python integration (2/5)

2. **SWIG** (Simplified Wrapper and Interface Generator)

- Pros:
 - Generates bindings for multiple languages.
 - Relatively easy for simple tasks.
 - Useful for multi-language projects.
- Cons:
 - Less efficient, less "pythonic" interface code.
 - Difficult to debug.
 - Complex for advanced use cases.

Common libraries for C++ and Python integration (3/5)

3. **pybind11**

- Pros:
 - Modern, lightweight, easy to use.
 - Header-only library.
 - More pythonic bindings.
- Cons:
 - Less advanced features than Boost.Python.
 - Less documentation, community support.
 - More manual work for complex bindings.

Common libraries for C++ and Python integration (4/5)

4. Cython

- Pros:
 - C extensions in Python-like syntax.
 - Significant performance improvements.
 - Good integration with Python ecosystem.
- Cons:
 - Requires learning new syntax.
 - Not for exposing existing C++ codebases.
 - Variable performance gains.

Common libraries for C++ and Python integration (5/5)

5. `ctypes`

- Pros:
 - Part of Python standard library.
 - Simple for basic tasks.
 - Good for calling C functions from Python.
- Cons:
 - Limited to C functions, not C++.
 - Manual type conversions.
 - Complex error handling.

Why pybind11?

- Modern, relevant, and practical for industry demands.
- Header-only library, which simplifies the build process.
- Lightweight, and easy to use.
- Balances ease of use with powerful features.
- Generates more pythonic bindings compared to SWIG.
- Suitable for a range of projects, enhancing problem-solving skills.

 **Note:** pybind11 may require more manual work for complex bindings.

PyPy and Numba overview

- **PyPy**
 - Alternative Python implementation focusing on speed.
 - JIT Compiler for runtime compilation.
 - Less memory usage, compatible with CPython.
 - Faster for long-running processes.
 - **Limitations:** Library support, JIT warm-up time.
- **Numba**
 - JIT compiler for Python and NumPy code.
 - Easy to use, significant performance improvements.
 - Integrates with Python scientific stack.
 - Supports CUDA GPU programming.
 - **Limitations:** Focused on numerical computing, learning curve for parallel programming, debugging challenges.

pybind11: installation

pybind11 overview

pybind11 is a lightweight, header-only library that connects C++ types with Python. This tool is crucial for creating Python bindings of existing C++ code. Its design and functionality are similar to the Boost.Python library but with a focus on simplicity and minimalism. pybind11 stands out for its ability to avoid the complexities associated with Boost by leveraging C++11 features.

[Documentation link](#)

How to install (1/3)

There are multiple methods to acquire the pybind11 source, available on [GitHub](#). The recommended approaches include via PyPI, through Conda, building from source, or importing it as a Git submodule.

Include with PyPI

pybind11 can be installed as a Python package from PyPI using pip:

```
pip install pybind11
```

Include with conda-forge

Conda users can obtain pybind11 via conda-forge:

```
conda install -c conda-forge pybind11
```

How to install (2/3)

Global install with brew

For macOS and Linux users, the brew package manager offers pybind11. Install it using:

```
brew install pybind11
```

Build from source

If you prefer to build from source, use the following commands:

```
wget https://github.com/pybind/pybind11/archive/refs/tags/v2.11.1.tar.gz  
tar xzvf v2.11.1.tar.gz
```

```
cd pybind11-2.11.1/  
mkdir build && cd build
```

```
cmake .. -DCMAKE_INSTALL_PREFIX=/opt/pybind11  
[sudo] make -j<N> install
```

How to install (4/4)

Include as a submodule

For Git-based projects, pybind11 can be incorporated as a submodule. In your Git repository, execute the following commands:

```
git submodule add -b stable https://github.com/pybind/pybind11 extern/pybind11  
git submodule update --init
```

This method assumes dependency placement in `extern/`. Remember, some servers might require the `.git` extension.

After setup, include `extern/pybind11/include` in your project, or employ pybind11's integration tools.

pybind11: basics

Header and namespace conventions

For brevity, all code examples assume that the following two lines are present:

```
#include <pybind11/pybind11.h>
namespace py = pybind11;
```

Some features may require additional headers, but those will be specified as needed.

Creating bindings for a simple function

Let's start by creating Python bindings for an extremely simple function, which adds two numbers and returns their result. For simplicity, we'll put both this function and the binding code into a file named `example.cpp` with the following contents:

```
int add(int i, int j) {
    return i + j;
}

PYBIND11_MODULE(example, m) {
    m.doc() = "pybind11 example plugin"; // Optional module docstring.

    m.def("add", &add, "A function that adds two numbers");
}
```

In practice, implementation and binding code will generally be located in separate files.

The `PYBIND11_MODULE` macro

The `PYBIND11_MODULE` macro creates a function that will be called when an `import` statement is issued from within Python. The module name (`example`) is given as the first macro argument (it should not be in quotes). The second argument (`m`) defines a variable of type `py::module_<module>` which is the main interface for creating bindings. The method `module_::def` generates binding code that exposes the `add()` function to Python.

Note (1/2)

Notice how little code was needed to expose our function to Python: all details regarding the function's parameters and return value were automatically inferred using template metaprogramming. This overall approach and the used syntax are borrowed from Boost.Python, though the underlying implementation is very different.

pybind11 is a header-only library, hence it is not necessary to link against any special libraries and there are no intermediate (magic) translation steps. On Linux, the above example can be compiled using the following command:

```
g++ -O3 -Wall -shared -std=c++11 -fPIC \  
 $(python3 -m pybind11 --includes) \  
 example.cpp -o example$(python3-config --extension-suffix)
```

If you included pybind11 as a Git submodule, then use `$(python3-config --includes) -Iextern/pybind11/include` instead of `$(python3 -m pybind11 --includes)` in the above compilation.

Note (2/2)

The `python3 -m pybind11 --includes` command fetches the include paths for both `pybind11` and Python headers. This assumes that `pybind11` has been installed using `pip` or `conda`. If it hasn't, you can also manually specify `-I <path-to-pybind11>/include` together with the Python includes path `python3-config --includes`.

On macOS: the build command is almost the same but it also requires passing the `-undefined dynamic_lookup` flag so as to ignore missing symbols when building the module.

How to import a C++ bound module

Building the above C++ code will produce a binary module file that can be imported to Python. Assuming that the compiled module is located in the current directory, the following interactive Python session shows how to load and execute the example:

```
import example  
example.add(1, 2) # Output: 3
```

Keyword arguments

With a simple code modification, it is possible to inform Python about the names of the arguments ("i" and "j" in this case).

```
m.def("add", &add, "A function which adds two numbers",
      py::arg("i"), py::arg("j"));
```

`arg` is one of several special tag classes which can be used to pass metadata into `module_::def`. With this modified binding code, we can now call the function using keyword arguments, which is a more readable alternative particularly for functions taking many parameters:

```
import example
example.add(i=1, j=2) # Output: 3L
```

Documentation

The keyword names also appear in the function signatures within the documentation.

```
help(example)
```

```
...
FUNCTIONS
add( . . . )
  Signature : (i: int, j: int) -> int
  A function which adds two numbers
```

Shorthand for keyword arguments

A shorter notation for named arguments is also available:

```
// Regular notation.  
m.def("add1", &add, py::arg("i"), py::arg("j"));  
  
// Shorthand.  
using namespace py::literals;  
m.def("add2", &add, "i"_a, "j"_a);
```

The `_a` suffix forms a C++11 literal which is equivalent to `arg`. Note that the literal operator must first be made visible with the directive `using namespace py::literals`. This does not bring in anything else from the `pybind11` namespace except for literals.

Default arguments

Suppose now that the function to be bound has default arguments, e.g.:

```
int add(int i = 1, int j = 2) {
    return i + j;
}
```

Unfortunately, pybind11 cannot automatically extract these parameters, since they are not part of the function's type information. However, they are simple to specify using an extension of `arg`:

```
// Regular notation.
m.def("add", &add, "A function which adds two numbers", py::arg("i") = 1, py::arg("j") = 2);

// Shorthand.
m.def("add2", &add, "i"_a=1, "j"_a=2);
```

The default values also appear within the documentation.

Exporting variables

To expose a value from C++, use the `attr` function to register it in a module as shown below. Built-in types and general objects (more on that later) are automatically converted when assigned as attributes, and can be explicitly converted using the function `py::cast`.

```
PYBIND11_MODULE(example, m) {
    m.attr("the_answer") = 42;
    py::object world = py::cast("World");
    m.attr("what") = world;
}
```

These are then accessible from Python:

```
import example
example.the_answer # Output: 42
example.what # Output: 'World'
```

Supported data types

A large number of data types are supported out of the box and can be used seamlessly as functions arguments, return values or with `py::cast` in general.

For a full overview, see the [official documentation](#).

pybind11: binding object-oriented code

Creating bindings for a custom type

Let's now look at a more complex example where we'll create bindings for a custom C++ data structure named `Pet`.

```
struct Pet {
    Pet(const std::string &name) : name(name) { }
    void set_name(const std::string &name_) { name = name_; }
    const std::string &get_name() const { return name; }

    std::string name;
};

PYBIND11_MODULE(example, m) {
    py::class_<Pet>(m, "Pet")
        .def(py::init<const std::string &>())
        .def("set_name", &Pet::set_name)
        .def("get_name", &Pet::get_name);
}
```

class_

`class_` creates bindings for a C++ *class* or *struct*-style data structure. `init` is a convenience function that takes the types of a **constructor**'s parameters as template arguments and wraps the corresponding constructor. An interactive Python session demonstrating this example is shown below:

```
import example
p = example.Pet("Molly")
print(p) # Output: <example.Pet object at 0x10cd98060>
p.get_name() # Output: 'Molly'
p.set_name("Charly")
p.get_name() # Output: 'Charly'
```

Note: Static member functions can be bound in the same way using `class_::def_static`.

Note: It is possible to specify keyword and default arguments using the syntax discussed in the previous section.

Binding lambda functions (1/2)

Note how `print(p)` produced a rather useless summary of our data structure in the example above:

```
print(p) # Output: <example.Pet object at 0x10cd98060>
```

To address this, we could bind a utility function that returns a human-readable summary to the special method slot named `__repr__`. Unfortunately, there is no suitable functionality in the `Pet` data structure, and it would be nice if we did not have to change it.

Binding lambda functions (2/2)

This can easily be accomplished by binding a lambda function instead:

```
py::class_<Pet>(m, "Pet")
    .def(py::init<const std::string &>())
    .def("set_name", &Pet::set_name)
    .def("get_name", &Pet::get_name)
    .def("__repr__",
        [](const Pet &a) {
            return "<example.Pet named '" + a.name + "'>";
        }
    );

```

With the above change, the same Python code now produces the following output:

```
print(p) # Output: <example.Pet named 'Molly'>
```

Instance and static fields

We can also directly expose the `name` field using the `class_::def_readwrite` method. A similar `class_::def_READONLY` method also exists for `const` fields.

```
py::class_<Pet>(m, "Pet")
    .def(py::init<const std::string &>())
    .def_readwrite("name", &Pet::name)
    // ...
```

This makes it possible to write

```
p = example.Pet("Molly")
p.name # Output: 'Molly'
p.name = "Charly"
p.name # Output: 'Charly'
```

Private fields (1/2)

Now suppose that `Pet::name` was a private internal variable that can only be accessed via setters and getters.

```
class Pet {  
public:  
    Pet(const std::string &name) : name(name) { }  
    void set_name(const std::string &name_) { name = name_; }  
    const std::string &get_name() const { return name; }  
  
private:  
    std::string name;  
};
```

Private fields (2/2)

In this case, the method `class_::def_property` (`class_::def_property_READONLY` for read-only data) can be used to provide a field-like interface within Python that will transparently call the setter and getter functions:

```
py::class_<Pet>(m, "Pet")
    .def(py::init<const std::string &>())
    .def_property("name", &Pet::get_name, &Pet::set_name)
    // ...
```

Write only properties can be defined by passing `nullptr` as the input for the read function.

Note: Similar functions `class_::def_readwrite_static`, `class_::def_READONLY_static`, `class_::def_PROPERTY_STATIC`, and `class_::def_PROPERTY_READONLY_STATIC` are provided for binding static variables and properties.

Dynamic attributes (1/3)

Native Python classes can pick up new attributes dynamically:

```
class Pet:  
    name = "Molly"  
  
p = Pet()  
p.name = "Charly" # Overwrite existing.  
p.age = 2 # Dynamically add a new attribute.
```

Dynamic attributes (2/3)

By default, classes exported from C++ do not support this and the only writable attributes are the ones explicitly defined using `class_::def_readwrite` or `class_::def_property`.

```
py::class_<Pet>(m, "Pet")
    .def(py::init<>())
    .def_readwrite("name", &Pet::name);
```

Trying to set any other attribute results in an error:

```
p = example.Pet()
p.name = "Charly" # Ok: attribute defined in C++.
p.age = 2
```

AttributeError: 'Pet' object has no attribute 'age'

Dynamic attributes (3/3)

The `py::dynamic_attr` tag enables dynamic attributes for C++ classes:

```
py::class_<Pet>(m, "Pet", py::dynamic_attr())
    .def(py::init<>())
    .def_readwrite("name", &Pet::name);
```

Now everything works as expected:

```
p = example.Pet()
p.name = "Charly" # Ok: overwrite value in C++.
p.age = 2 # Ok: dynamically add a new attribute.
p.__dict__ # Output: {'age': 2}
```

Note that there is a small runtime cost for a class with dynamic attributes. Not only because of the addition of a `__dict__`, but also because of more expensive garbage collection tracking which must be activated to resolve possible circular references. Native Python classes incur this same cost by default.

Inheritance, downcasting and polymorphism (1/6)

Suppose now that the example consists of two data structures with an inheritance relationship:

```
struct Pet {  
    Pet(const std::string &name) : name(name) {}  
    std::string name;  
};  
  
struct Dog : Pet {  
    Dog(const std::string &name) : Pet(name) {}  
    std::string bark() const { return "woof!"; }  
};
```

Inheritance, downcasting and polymorphism (2/6)

There are two different ways of indicating a hierarchical relationship to pybind11: the first specifies the C++ base class as an extra template parameter of the `class_`:

```
py::class_<Pet>(m, "Pet")
    .def(py::init<const std::string &>())
    .def_readwrite("name", &Pet::name);

// Method 1: template parameter:
py::class_<Dog, Pet /* <- Specify C++ parent type- */>(m, "Dog")
    .def(py::init<const std::string &>())
    .def("bark", &Dog::bark);
```

Inheritance, downcasting and polymorphism (3/6)

Alternatively, we can also assign a name to the previously bound `Pet` `class_` object and reference it when binding the `Dog` class:

```
py::class_<Pet> pet(m, "Pet");
pet.def(py::init<const std::string &>())
    .def_readwrite("name", &Pet::name);

// Method 2: pass parent class_ object.
py::class_<Dog>(m, "Dog", pet /* <- Specify Python parent instance. */)
    .def(py::init<const std::string &>())
    .def("bark", &Dog::bark);
```

Inheritance, downcasting and polymorphism (4/6)

Functionality-wise, both approaches are equivalent.

```
p = example.Dog("Molly")
p.name # Output: 'Molly'
p.bark() # Output: 'woof!'
```

The C++ classes defined above are regular non-polymorphic types with an inheritance relationship. This is reflected in Python:

```
// Return a base pointer to a derived instance.
m.def("pet_store", [](){ return std::unique_ptr<Pet>(new Dog("Molly")); });
```

```
p = example.pet_store()
type(p) # Output: Pet
# No pointer downcasting for regular non-polymorphic types.
p.bark() # Output: AttributeError: 'Pet' object has no attribute 'bark'
```

Inheritance, downcasting and polymorphism (5/6)

The function returned a `Dog` instance, but because it's a non-polymorphic type behind a base pointer, Python only sees a `Pet`. In C++, a type is only considered polymorphic if it has at least one virtual function and pybind11 will automatically recognize this:

```
struct PolymorphicPet {
    virtual ~PolymorphicPet() = default;
};

struct PolymorphicDog : PolymorphicPet {
    std::string bark() const { return "woof!"; }
};

py::class_<PolymorphicPet>(m, "PolymorphicPet");
py::class_<PolymorphicDog, PolymorphicPet>(m, "PolymorphicDog")
    .def(py::init<>())
    .def("bark", &PolymorphicDog::bark);

// Again, return a base pointer to a derived instance.
m.def("pet_store2", []() { return std::unique_ptr<PolymorphicPet>(new PolymorphicDog); });
```

Inheritance, downcasting and polymorphism (6/6)

```
p = example.pet_store2()  
type(p) # Output: PolymorphicDog  
p.bark() # Output: 'woof!'
```

Given a pointer to a polymorphic base, pybind11 performs automatic downcasting to the actual derived type. Note that this goes beyond the usual situation in C++: we don't just get access to the virtual functions of the base, we get the concrete derived type including functions and attributes that the base type may not even be aware of.

Overloaded methods (1/2)

Sometimes there are several overloaded C++ methods with the same name taking different kinds of input arguments:

```
struct Pet {  
    Pet(const std::string &name, int age) : name(name), age(age) {}  
  
    void set(int age_) { age = age_; }  
    void set(const std::string &name_) { name = name_; }  
  
    std::string name;  
    int age;  
};
```

Attempting to bind `Pet::set` will cause an error since the compiler does not know which method the user intended to select.

Overloaded methods (2/2)

We can disambiguate by casting them to function pointers. Binding multiple functions to the same Python name automatically creates a chain of function overloads that will be tried in sequence.

```
py::class_<Pet>(m, "Pet")
    .def(py::init<const std::string &, int>())
    .def("set", static_cast<void (Pet::*)(int)>(&Pet::set), "Set the pet's age")
    .def("set", static_cast<void (Pet::*)(const std::string &)>(&Pet::set), "Set the pet's name");
```

The overload signatures are also visible in the method's docstring. If you have a C++14 compatible compiler, you can use an alternative syntax to cast the overloaded function:

```
py::class_<Pet>(m, "Pet")
    .def("set", py::overload_cast<int>(&Pet::set), "Set the pet's age")
    .def("set", py::overload_cast<const std::string &>(&Pet::set), "Set the pet's name");
```

Here, `py::overload_cast` only requires the parameter types to be specified. The return type and class are deduced.

Overloaded `const` methods

If a function is overloaded based on constness, the `py::const_` tag should be used:

```
struct Widget {
    int foo(int x, float y);
    int foo(int x, float y) const;
};

py::class_<Widget>(m, "Widget")
    .def("foo Mutable", py::overload_cast<int, float>(&Widget::foo))
    .def("foo const", py::overload_cast<int, float>(&Widget::foo, py::const_));
```

Note: this approach also works for multiple overloaded constructors.

Enumerations and internal types (1/2)

Let's now suppose that the example class contains internal types like enumerations, e.g.:

```
struct Pet {
    enum Kind {
        Dog = 0,
        Cat
    };

    struct Attributes {
        float age = 0;
    };

    Pet(const std::string &name, Kind type) : name(name), type(type) { }

    std::string name;
    Kind type;
    Attributes attr;
};
```

Enumerations and internal types (2/2)

The binding code for this example looks as follows:

```
py::class_<Pet> pet(m, "Pet");

pet.def(py::init<const std::string &, Pet::Kind>())
    .def_readwrite("name", &Pet::name)
    .def_readwrite("type", &Pet::type)
    .def_readwrite("attr", &Pet::attr);

py::enum_<Pet::Kind>(pet, "Kind")
    .value("Dog", Pet::Kind::Dog)
    .value("Cat", Pet::Kind::Cat)
    .export_values();

py::class_<Pet::Attributes>(pet, "Attributes")
    .def(py::init<>())
    .def_readwrite("age", &Pet::Attributes::age);
```

Scoping

To ensure that the nested types `Kind` and `Attributes` are created within the scope of `Pet`, the `pet` `class_` instance must be supplied to the `enum_` and `class_` constructor. The `enum_::export_values` function exports the enum entries into the parent scope, which should be skipped for newer C++11-style enum classes.

```
p = Pet("Lucy", Pet.Cat)
p.type # Output: Kind.Cat
int(p.type) # Output: 1L
```

The entries defined by the enumeration type are exposed in the `__members__` property:

```
Pet.Kind.__members__ # Output: {'Dog': Kind.Dog, 'Cat': Kind.Cat}
```

The `name` property returns the name of the enum value as a unicode string. Contrary to Python customs, enum values from the wrappers should not be compared using `is`, but with `=`.

How to build and import pybind11 modules

How to build and import pybind11 modules

The [Python example](#) and [CMake example](#) repositories good places to start to understand how to build and import pybind11 modules. There are three main ways to do it:

1. Manual compilation.
2. Compilation using `CMake`.
3. Compilation using `setuptools`.

In order to be able to import the compiled module in Python, add the folder containing your dynamic library to the environment variable `PYTHONPATH` accordingly.

Manual compilation

To manually compile C++ code with pybind11, use one of the following commands:

```
# pybind11 installed via pip or conda.  
g++ -O3 -Wall -shared -std=c++11 -fPIC \  
  $(python3 -m pybind11 --includes) \  
  example.cpp -o example$(python3-config --extension-suffix)  
  
# pybind11 built from source.  
g++ -std=c++11 -O3 -shared -fPIC \  
  -I/path/to/pybind11/include $(python3-config --cflags --ldflags --libs) \  
  example.cpp -o example$(python3-config --extension-suffix)  
  
# pybind11 included as a Git submodule.  
g++ -std=c++11 -O3 -shared -fPIC \  
  -Iextern/pybind11/include $(python3-config --cflags --ldflags --libs) \  
  example.cpp -o example$(python3-config --extension-suffix)
```

On macOS: add the `-undefined dynamic_lookup` flag so as to ignore missing symbols when building the module.

How to compile using CMake (1/2)

To compile and run your pybind11 code with CMake, create a `CMakeLists.txt` script as follows:

```
cmake_minimum_required(VERSION 3.5)
project(example)

find_package(pybind11 REQUIRED)
include_directories(SYSTEM ${pybind11_INCLUDE_DIRS})
pybind11_add_module(example example.cpp)
```

or, if pybind11 is included as a subfolder:

```
cmake_minimum_required(VERSION 3.5)
project(example)

add_subdirectory(pybind11)
pybind11_add_module(example example.cpp)
```

Please refer to the [CMake example](#) repository for further details.

How to compile using CMake (2/2)

Then:

```
cd /path/to/example/  
mkdir build && cd build  
cmake -Dpybind11_DIR=/path/to/pybind ..  
make -j<N>
```

How to compile using `setuptools` (1/3)

For projects on PyPI, building with setuptools is the way to go. Sylvain Corlay has kindly provided an example project which shows how to set up everything, including automatic generation of documentation using Sphinx. Please refer to the [Python example](#) repository for further details.

`setup.py` is a Python script commonly used in Python projects for tasks like building, distributing, and installing module packages. In pybind11, it compiles and links C++ code into Python modules.

How to compile using `setuptools` (2/3)

Example of a `setup.py` file:

```
from setuptools import setup, Extension
from setuptools.command.build_ext import build_ext
import sys
import setuptools

class get_pybind_include(object):
    # Helper class to determine the pybind11 include path.
    def __str__(self):
        import pybind11
        return pybind11.get_include()

ext_modules = [
    Extension(
        'your_module_name', # Name of the module.
        ['your_module.cpp'], # C++ source files.
        include_dirs=[
            get_pybind_include(), # Path to pybind11 includes.
            '/path/to/other/includes', # Additional include paths.
        ],
        language='c++'
    ),
]
```

```
setup(
    name='your_module_name',
    version='0.1',
    author='Your Name',
    author_email='your.email@example.com',
    description='A Python module using pybind11',
    long_description='',
    ext_modules=ext_modules,
    install_requires=['pybind11>=2.5.0'], # pybind11 requirement.
    cmdclass={'build_ext': build_ext},
    zip_safe=False,
)
```

How to compile using `setuptools` (3/3)

1. **Create the `setup.py` file:** Place it in your project's root directory.
2. **Modify the file:** Change `your_module_name` and `your_module.cpp` to your module's name and C++ source file name. Adjust paths as needed.
3. **Build the module:**

```
python setup.py build_ext --inplace
```

This compiles the C++ code into a shared object file (.so).

4. **Install the module:**

```
python setup.py install # Or: pip install .
```

The build step of `setuptools` can be run through CMake itself. The [CMake example](#) repository provides a prototype `setup.py` to adjust to your needs

Note: A similar approach using a `pyproject.toml` file is also possible.

Examples

C++ vs. native Python benchmark (1/2)

The example provided in the `c++_vs_py` folder showcases a performance comparison test between bound C++ code and native Python code on a simple linear algebra operation, such as a matrix-matrix product.

The code can be compiled with:

```
g++ -O3 -Wall -shared -std=c++11 -fPIC \
$(python3 -m pybind11 --includes) \
matrix_multiplication.cpp -o matrix_ops$(python3-config --extension-suffix)
```

C++ vs. native Python benchmark (2/2)

Expected results

Typically, the C++ implementation should be significantly faster than the pure Python implementation for several reasons:

- **Execution speed:** C++ is a compiled language and is generally faster than Python, an interpreted language, especially for computation-intensive tasks.
- **Optimization:** Compilers for C++ can optimize the code for performance, whereas Python's flexibility and dynamic typing can introduce overhead.
- **Handling of loops:** C++ is more efficient in handling loops and arithmetic operations compared to Python.

Notes

- The actual performance gain can vary depending on the system, the size of the matrices, and the compiler optimizations.
- For matrix operations, libraries like NumPy in Python are highly optimized and can offer performance close to C++, but in this comparison, we are using a pure Python implementation to illustrate the difference more clearly.
- Remember that developing and maintaining C++ code requires more effort compared to Python, so the decision to use C++ should consider both performance benefits and development costs.

Other examples

The examples provided in the `examples` folder are adapted and extended versions from [this GitHub repository](#). They demonstrate the use of pybind11 in various scenarios.

Further readings

- [pybind11 documentation](#)
- [pybind11 testsuite](#)
- [Using pybind11](#)



That's all Folks!

Exercise session 12

Integrating C++ and Python codes.

Advanced Programming - SISSA, UniTS, 2023-2024

Pasquale Claudio Africa

21 Dec 2023

Exercise 1: binding classes and magic methods

Provide Python bindings using pybind11 for the code provided as the solution to exercise 1 from session 03.

1. Bind the `DataProcessor` class and its member functions. Using a lambda function, expose a constructor taking a Python list as an input, to be converted to a `std::vector` and invoking the actual constructor.
2. Provide Python bindings for the addition (`__add__`), the read (`__getitem__`) and write (`__setitem__`) access, and the output stream (`__str__`) operators.
3. Package the Python module with the compiled C++ library using `setuptools`.
4. Write a Python script to replicate the functionalities implemented in the `main.cpp` file.

Exercise 2: binding class templates

Provide Python bindings using pybind11 for the code provided as the solution to exercise 2 from session 05.

1. Modify the `NewtonSolver::solve()` method in order to throw a `std::runtime_error` exception instead of returning `Nan` when failed to converge to a root.
2. Bind the `NewtonClass` class and its member functions, providing explicit instantiations for `double` and `std::complex<double>` numbers. The Python interface should provide consistent default arguments. Python bindings should be implemented in a separate `newton_py.cpp` file. Translate the `std::runtime_error` C++ exception to a `RuntimeError` Python exception.
3. Use CMake to setup the build process.
4. Write a Python script to replicate the functionalities implemented in the `main.cpp` file.
5. Verify that exception handling works properly.

Exercise 3: binding with external libraries

1. Implement C++ functions using the Eigen library to perform matrix-matrix multiplication and matrix inversion.
2. Provide Python bindings using pybind11 for the code implemented.
3. Use CMake and `setuptools` to setup the build process.
4. Write a Python script to test the performance of the Eigen-based operations. Implement a `log_execution_time` decorator to print the execution time of a function.
5. Compare the execution time of these operations to equivalent operations in NumPy (e.g., `numpy.matmul` for multiplication and `numpy.linalg.inv` for inversion). Use a large matrix (e.g., 1000×1000) of random integers between 0 and 1000 for the test.

Exercise 4: code obfuscation

What's the output resulting from the execution of the code contained in `wish.cpp` ?