Programming Techniques for Scientific Simulations I

Authors: T. Lausberg, R. Worreby, revised by R. Barton

Contact: toml@ethz.ch, rworreby@ethz.ch, rbarton@ethz.ch

In relation to lecture by: Dr. Roger Käppeli

For corrections, questions, remarks please contact the current maintainer: rbarton@ethz.ch

---

Note: Offline cppreference is searchable via "Index" or "std Symbol Index" from homepage, use Crl+F

Fully revised chapters: 1-3, 9, 16-22

Recent Changes (see git also):

- Reorganisation: Added 5. C++ Features & Keywords, moved chapters around
- 6 Functors, Operators, small changes
- 14 Inheritance corrections, some inaccuracies previously

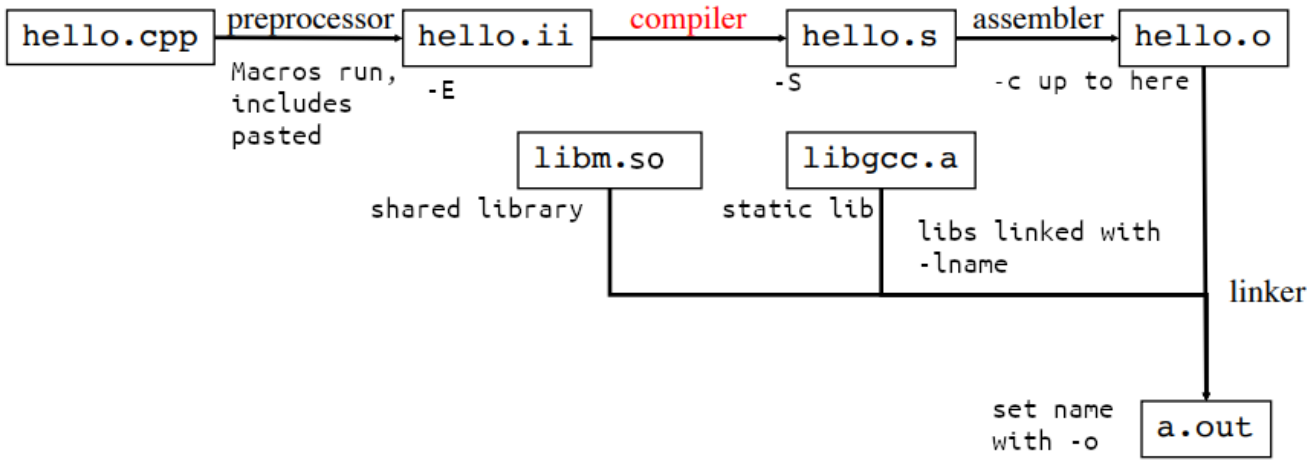
Table of Contents

1. C++ compiling

1.1 Compile Process

Programs can be split into several files/libraries, compiled separately and finally linked together.

- `.ii` pre-processed files, with macros run ie `#` lines
- `.s` assembly files (already specific to a CPU/OS)
- `.o` are compiled files (object code)
- `.a`, `.so` are compiled static, shared libraries respectively
- `.out` or `.exe` is the final program to be run. `a.out` is the default name (machine code)



1. **Preprocessor**, handles all lines starting with a `#` (Macros), outputs c++ text file
2. **Compiler + Assembler**, convert c++ code to assembly code
3. **Linker**, combines compiled `.o` files to a runnable file

1.2 Compile via CLI

This process independent of compiler (c++, g++, clang) and options/flags can be added *mostly* in any order. You have to be careful with the order when *linking* multiple files

```
c++ [options] file1 file2...
c++ main.cpp timer.cpp -std=c++17 -o main.exe
```

Compile Options

- **`-o`** Set name of output (usually last flag)
- **`-Wall -Wextra -Wpedantic`** Turn on more warnings (`-Weverything` with clang)
- **`-std=c++11`** Set C++ version
- **`-Dname=value`** Define macro name (case-sensitive), `=value` optional
- **`-lname`** Link the library named `libname.a`. *If libx.a calls a function in liby.a, you need to link in the right order:* `-lx -ly`

  `-Lpath` Set where to look for libraries. Use just `-L.` to look in current dir

`-Ipath` *(as in -i)* Add directory to look for headers, often used for adding library headers

- `-c` Compile files to `.o`, don't link

  `-E` Preprocess only

  `-S` Preprocess and compile to assembly files `.s`
- `-g` Generate additional debugging information
- `--save-temps` Saves intermediate compile files: .ii .s .o
- `--help` To see more options

Optimisation Options

- `-DNDEBUG` turn off debug code, such as asserts
- `-O3` Set optimisation level 3 (possible 0-5, fast)
  - `-O` flags are shortcuts to turn on many different compiler optimisation techniques. Levels 4+ increase the risk of wrong code being generated.
  - `-Os` optimises a program for size instead of speed
  - `-Og:` enables optimisations that don't affect debugging *(gcc >= 4.8)*
  - `-O0` *(default level)* switch off optimisations
- `-march=native` Minimum processor architecture to be run on, native is the architecture of the computer running the compiler on. Can also put other arch instead of `native`.
- `-fopt-info` gives you information on which optimisations have been done *(gnu compilers)*
- `-funroll-loops` Will optimise short loops by writing out the iterations individually

Libraries

Static/shared libraries are essentially pre-compiled collections of `.cpp` files with header files (`.a` or `.so` + headers).
We `include` the libraries' header files, so we know how to interface with the library when compiling our own `.o` files, i.e. knowing which functions exists, their input/outputs.
During the linking stage we combine the pre-compiled library and our `.o` files, where the linker checks that the functions actually exists

Create **static** library:

1. `c++ -c square.c` To compile files to .o
2. `ar rcs libsquare.a square1.o square2.o` To create the library from compiled .o's. include 'lib' as prefix to easily include later.
   - `r` means replace existing, `c` don't warn if lib doesn't exist, `s` create lib index

Create **shared** library: (a library that exists only once in memory, can be used by several programs at once)

1. `c++ -fPIC -c square.c` To compile files to .o, `-fPIC` for position independent code
2. `c++ -shared -fPIC -o libsquare.so square1.o square2.o` To create library from compiled .o's.

Header Only Libraries

Libraries that are *not pre-compiled and all code is in the headers*, which means more compile time optimisations can be made (but is slower to compile)
Necessary for templated libraries, templates are always in the header as we need to create a compiled version per type that is used. E.g. Eigen

Example

square.hpp

```
double square(double x);
```

square.cpp

```
#include "square.hpp"
double square(double x) {
   return x*x;
}
```

main.cpp

```
#include <iostream>
#include "square.hpp"   //cpp not included

int main() {
     std::cout << square(5.);
}
```

Compiling:

1. Compile the file `square.cpp`, with the `-c` option (no linking, creates `square.o`)
   `$ c++ -c square.cpp`
2. Compile the file `main.cpp`, link `square.o` and output `square.exe`
   `$ c++ main.cpp square.o -o square.exe`
3. *Alternatively*, compile `main.cpp` with no linking as in step 1. Then perform just the linking
   `$ c++ main.o square.o –o square.exe`

Using Library:

1. `$ c++ -c square.cpp`
2. Package square.o into a static library
   `$ ar rcs libsquare.a square.o`
3. Compile `main.cpp` and link the lib, `-L.` to look for lib in current folder (signified by the `.`)
   `$ c++ main.cpp -L. -lsquare` or directly
   `$ c++ main.cpp libsquare.a`

After a while this gets tedious $\implies$ use a build system, e.g. make, cmake...

1.3 Pre-processor Macros

All lines starting with a `#` are handled by the pre-processor (1st compilation step). These manipulate the `cpp` text file, removing/adding/replacing text, before compilation.

Remember compiler flags: `-DNAME=VALUE` to define a variable with an *optional* value and `-E` to output only pre-processed `.ii` file.

Common Macros:

- **`#include <filename>`** pastes the file into that line, brackets affect where to look for the file
  - `<>` looks in list of directories known to the compiler, where libs are installed, commonly `/usr/include`
  - `“”` looks in the current directory first then where `<>` looks, *can specify a relative path e.g.* `#include “lib/simpson.h”`
- **`#define NAME VALUE`** defines variable in code, value is *optional* (default is `1`)
  - Important: then all occurrences of `NAME` below will be replaced by `VALUE`
  - Using the `-D` compiler flag is equivalent to writing `#define` on the first line
  - `#define NAME(param1, param2) function` advanced defines exist with parameters (see example)
- **`#undef NAME`** undefine/delete var
- **`#ifdef NAME`** Checking if variables are defined (exist) is a good way of doing boolean logic, as the user only has to define the variables they are interested in when compiling.
- **`#if CONDITION`** if statement, works with normal logical operators `&&`, `||`, `!` etc
  - can also use `defined(NAME)` in the condition, to check if `NAME` is defined
- **`#endif`** close an if-statement, like the `}` in c++
- `#else`, `#elif` (else-if)
- `#error message` throw error, printed to console (`#warning` also exists)

Use Cases:

- Including files
- Conditional compilation, e.g. platform specific
- Control from terminal, how to compile
- Error checking, e.g. on invalid defined variables
- Text manipulation, replacement *(not recommended)*

Tips:

- **Don’t use macros unless you have to**, using this can lead to:
  - Unexpected side effects
  - Problems with debugger and other tools
  - No type safety
- Use *all caps names* to prevent unwanted name conflicts with c++ code, causing the code to be replaced

Macros are very important in C but have far fewer uses in C++. Almost every macro demonstrates a flaw in the programming language or programmer.
Because they rearrange the program text before the compiler properly sees it, macros are also a major problem for many programming support tools.

So when you use macros, you should expect inferior service from tools such as debuggers, cross- reference tools, and profilers.
If you must use macros, read the reference manual and try not to be too clever.

Include Guards

`#include` will paste the source file in place. But if we include the same file in *multiple* places we are dulicating the code and so *redeclaring* the same classes/functions. This gives linker errors (when we combine the compiled `.o` files)

So we use macros, to ensure the code is only 'included' once for the whole project. Essentially this makes the file appear empty when pasting with the `#include`. The macros below should surround *all* the code in the file.

```
#ifndef GRANDFATHER_H //convention: name the symbol as the name of the file
#define GRANDFATHER_H //define the symbol so afterwards we do not enter here
struct foo{
   int member;
};
#endif /* GRANDFATHER_H */
```

Other `#include` in the file can be placed anywhere as they are also guarded. Its best to have them below the `#define …` as the pre-processor only needs to execute that once.

Examples

Conditional Compilation

```
#define SYMBOL    //or compile with: "c++ -DSYMBOL main.cpp"

#ifdef SYMBOL
something
#else
alternative
#endif

//converted to
something
```

Replacement

```
#define A 12
std::cout << A;
//converted to
std::cout << 12;

#define SUM(A,B) A+B
std::cout << SUM(3,4);
//converted to
std::cout << 3+4;

#define std    //example of accidental name conflict, avoid this
```

```
#define std 1  //equivalent
std::cout << "hello";
//converted to
1::cout << "hello";  //syntax error
```

Undefine Macros

```
#define XXX “Hello”
std::cout << XXX;
#undef XXX
std::cout << “XXX”;

//converted to
std::cout << “Hello”;
std::cout << “XXX”;
```

If-Statement

```
#if !defined (__GNUC__)
   #error This program requires the GNU compilers
#elif __GNUC__>=3
   std::cout << “gcc version 3 or higher”;
#endif
```

2. Make

This is used to compile a *project* with several files into an executable file, hence a *build automation tool*. It handles dependencies and detects changes to files, to only recompile necessary files.

- We write a `Makefile` *(no file extension)* to define how to compile our project in that directory.
- We create a **rule** for each **target** (output/compiled file), which defines how to create/update that file
- We can add **dependencies** to a target, this will ensure that each of the dependencies is present and updated (if there is a rule for it) before we run the commands for the **target**

Use in CLI:

- `make` usually used to build the whole project. Will build the first target in the `Makefile`
- `make mytarget` run the rule for `mytarget`
- `make -f file` use a specific makefile
- `make -n` 'dry run' just prints commands to be executed, ***useful to debug***, see if variables are correct etc
- `make -C path` run a makefile in the path directory. Used later to run a makefile from within a makefile

2.1 Makefiles

```
# This is the 'rule' for the output file 'target'
target: dependencie/s
[TAB]command/s
```

- `target` the name of the rule and its output file
- `dependencies` list all files that are *required* to exist/be up-to-date to create the file `rule`
  - Be sure to include the header as a dependency, so that it checks there for changes as well
- `command` write normal terminal commands, usually to compile a `cpp` or create a `library`
  - Important: needs to have a tab before, *not 3 spaces*
- `.PHONY: target` is used to say that a rule does not correspond to a file, e.g. `all` or `clean`
  - The target `all` is often added as the first rule which defines what is to be compiled. As it is the first rule, we are setting it as the default target.
- `export` to expose variables to the environment. Allows other makefiles called to see the same variables *(see longer example)*

Short Example:

```
main.exe: main.cpp simpson.hpp simpson.o
   c++ -o main.exe main.cpp simpson.hpp simpson.o

simpson.o: simpson.cpp simpson.hpp
   c++ -c simpson.cpp
```

Instead of issuing commands manually:

```
$ c++ -c simpson.cpp
$ c++ -o main.exe main.cpp simpson.hpp simpson.o
```

2.2 Variables

- `my_var = value` to set variable
- `${my_var}` or `$(my_var)` to get value

It is best to **create an additional config.mk file** to store the *local* values of the variables (don't ad to git). Then include the file with `-include`. By adding the `-` in front of `include` we say that this is *optional*.

```
-include config.mk
```

Special Variables:

- `$@` Name of the *target*, for the current rule
- `$<` Name of the *first* dependency (like arrow the leftmost dependency)
- `$^` List of *all* dependencies (with spaces)

Predefined Variables:

- `CXX` Command for compiling C++ *(default `g++`)*
  `CXXFLAGS` flags/options for the compiler
- `LDFLAGS` flags/options for compilers when linking
  `LDLIBS` Library flags or names for compilers when linking
- `MAKE` The make command we used to run the file. Use this when running another makefile from within make, as the options are passed as well. Avoid using just the `make` command.
- `RM` Command to remove a file *(default `rm -f`)*

2.3 Examples

config.mk, stores variables

```
CXX = g++
CXXFLAGS = -std=c++11
CXXFLAGS += -Wall -Wextra -Wpedantic
CXXFLAGS += -O3 -march=native
LDFLAGS = -Lintegrator
LDLIBS = -lintegrate
```

integrator/Makefile, makefile to create the library

```
# Create a library from the object code
libintegrate.a: simpson.o
   ar rvs $@ $^
#  ar rvs libintegrate.a simpson.o     (evaluated code)

# Object code depending on header and implementation.
simpson.o: simpson.cpp simpson.hpp
   $(CXX) $(CXXFLAGS) -c -o $@ $<
#  g++ std=c++11 <more flags> -c -o simpson.o simpson.cpp      (evaluated code)

.PHONY: clean
clean:
   rm -f *.o *.a
```

Makefile

```
-include config.mk

# Export variables into the environment. So that other makefiles see the same variables
export

# The first target is called 'all' by convention and used to set what we want to run
.PHONY: all
all: main.exe

# Compile our code and generate an executable binary together with the library
main.exe: main.cpp integrator/libintegrate.a
   $(CXX) $(CXXFLAGS) -o $@ $< $(LDFLAGS) $(LDLIBS)
#  evaluates to
#  c++ -std=c++11 <more flags> -o main.exe main.cpp -Lintegrator -lintegrate
#  test yourself with: make -n

# Exported variables will also be visible in a sub-make
integrator/libintegrate.a:
   $(MAKE) -C integrator      # Call the makefile in the integrator folder

.PHONY: clean
clean:
   rm -f *.o *.a main.exe # could also use $(RM) instead of 'rm -f'
   $(MAKE) -C integrator clean   # call the clean of the other makefile
```

For projects with **multiple directories**, it is common to create a *Makefile per directory*, then call them with `$(MAKE) -C path` as in the `libintegrate.a` or `clean` above

3. CMake

CMake is a *cross-platform* ***build system generator***. It is used to define ***how to compile*** projects. This is so that we can simply specify which executables/libraries we want to create with which files.

Commonly it creates a `Makefile` which we can then run to compile our project. However, it can be used for many compilers/platforms/build systems (not just make).

3.1 General

We create a `CMakeLists.txt` file where we define how to build our project. Where we write all CMake code

Running CMake

1. **`cmake <path>`** create a Makefile using the `CMakeLists.txt` from the `<path>`
    - *Reminder: Current directory is* `.` *and one directory up is* `..`
    - The Makefile is created in the current directory by default
    - It is recommended to create a separate `build` directory so that our compiled files do not fill up our source directory
2. **`make`** compile the project with make
    - `make target` compile a specific target *(target names defined in CMakeLists)*
3. `make install` install files, usually for libraries *(defined in CMakeLists what to install)*
4. `make clean` delete all compiled files

Short Example:

```
Starting in source directory, where CMakeLists is located
$ mkdir build
$ cd build
$ cmake ..
$ make
or compile a specific target
$ make square.exe
```

Basic CMake Functions

Short list of basic functions that can be added to the CMakeLists.txt:

- **`cmake_minimum_required()`** sets the minimum version of CMake for this project, prevents errors from backward compatibility issues
    - You should use the current version of CMake on your system `cmake --version`
- **`project()`** sets the project name, useful for debugging
- **`add_executable(<Target Name> <Source Files>)`** compile an executable
    - `Target Name` name of the executable, use this later when adding libraries etc
    - `Source files` list of all files/dependencies, separated with spaces
    - Source files do not have to be in the current folder, must write with path
- **`add_library(<Name> <TYPE> <Source Files>)`** create a library, similar to `add_executable`

    - *Note: The filename has the `lib` prefix (i.e. `lib<Name>`)*
    - `Type` is either `STATIC` or `SHARED`
- **`target_link_libraries(<target> <libraries>)`** Makes the library accessible to the target, adds it in the linking stage of compilation, (`-l` flag)
    - Use the name of the library without the `lib` prefix
- **`add_subdirectory(<path>)`** run the CMakeLists.txt in a subdirectory, e.g. for your own library
    - We can then use the targets/libraries created in the other CMakeList

Compiling:

- **`add_definitions (-D<NAME>)`** sets compiler definitions for use with macros
- **`include_directories(<path>)`** Tells the pre-processor where to look for `#include <>` files
    - Use this when header files, e.g. for a library, are not in the default install folders or the project.
    - Or to avoid writing path, `#include "QR"` instead of `#include "Eigen/QR"` *(not recommended)*
- `target_include_directories(<Target> PRIVATE <path>)` Newer version of `include_directories` specific for a target and has an access level
- `find_package(<NAME>)` searches for an installed library to be linked (see eigen example)
- `add_compile_options(<flags>)` way of setting compile flags manually
    - e.g. `add_compile_options(-Weverything)`

Variables:

- **`set(variable value)`** to set the value of a variable, can also have multiple values to create a list
    - **`$(variable)`** to use the variable, as usual
- `set(CMAKE_BUILD_TYPE Release)` sets release (vs development) to optimise further
- `set(CMAKE_CXX_STANDARD 11)` use C++11
- `file(GLOB <variable> <expression>)` finds all files that meet the expression and saves it to the var
    - e.g. `file(GLOB varName “integrator/*.cpp”)` *(not recommended, should be explicit)*
    - Then we can just `add_executable(main ${varName})`

Installing Files:

Note: Will only install with `make install`
The idea is to copy the output files to a standard location so they can be easily accessed from anywhere
e.g. so that we can use the `#include <myLibrary>` without having `myLibrary` in our project
or run the program from any directory in the terminal

- **`install(TARGETS <targets> DESTINATION <path>)`** installs the `target` outputs (ie `.exe` or `.a`) by ***copying*** them to the `path`
    - Can specify for only parts to be installed by adding `LIBRARY`, `RUNTIME` etc. before `DESTINATION`
    - `FILES` instead of `TARGETS` just copies the files, *used for headers*
    - `path` will be a relative path, prefix with `~`(home) or `/`(root) to use the absolute paths

Testing:

See Exercise 6 Testing

We create another executable, with a `main()`, for the test. When we throw an unhandled exception or an assert is false, the test fails

- **`enable_testing()`** need to set this otherwise tests will not be compiled
- **`add_test(NAME test_name COMMAND test_target_name)`** to add a target as a test, need to `add_executable(test_target_name test.cpp)` to compile the test target
    - This just classifies an executable as a test, so that we can easily call the test and also disable the test from being compiled

      ```
      add_executable(genome_test genome_test.cpp)  # create the executable test
      target_link_libraries(genome_test penna) # add necessary libraries
      add_test(NAME Genome COMMAND genome_test) # add the test
      ```

- **`make all test`** to execute all tests

Minimal Example

```
cmake_minimum_required(VERSION 2.8) # Require certain version
project(ProjectName) # Name your project

# Compile the square program from the two source files
add_executable(square.exe main.cpp square.cpp)
```

It is easy to forget to write the `VERSION` in the first line

Adding Warning Flags (Common)

```
if(CMAKE_CXX_COMPILER_ID MATCHES "(C|c?)lang") #need to distinguish between compilers
  add_compile_options(-Weverything)
else()
  add_compile_options(-Wall -Wextra -Wpedantic)
endif()
```

3.2 Examples

Libraries with Subdirectories

Generally, we have a *separate directory* per library. There are two options for subdirectories:

1. Use one CMakeLists and add path prefixes, e.g. `timer/timer.h`
2. Create a separate CMakeLists for each library/subdirectory and ensure it is called with `add_subdirectory(path)`
    - Normally used for a library or other distinct module of the program

Subdirectories Example (Ex03 Simpson)

We have a subdirectory `integrator` with files: CMakeLists.txt, simpson.cpp, simpson.h

Option 1:

```
cmake_minimum_required(VERSION 3.1)
project(simpson_cmake)

add_library(simpson_lib STATIC integrator/simpson.cpp) # Create the static library
# ensure header files can be included for the simpson.cpp
target_include_directories(simpson_lib PUBLIC integrator)

add_executable(main main.cpp)
target_link_libraries(main simpson_lib) # Link the library to the executable
```

Option 2:

```
cmake_minimum_required(VERSION 3.1)
project(simpson_cmake)

# this tells cmake to look for a CMakeLists.txt file in the folder integrator
# (which is where the add_library command is defined)
add_subdirectory(integrator)

# all that remains is to define our executable and the library to link against
add_executable(main main.cpp)
target_link_libraries(main simpson_lib)#Link the library created in the other CMakeList
```

CMakeLists.txt in `integrate/` subdirectory

```
add_library(simpson_lib STATIC simpson.cpp) # Create the static library
target_include_directories(simpson_lib PUBLIC ./)#Include the headers in the currentdir
```

Eigen Library

```
cmake_minimum_required(VERSION 3.10)
project(Exercises)

add_executable(main main.cpp)

# Add Eigen libary, use #include <Eigen/Dense> afterwards
find_package (Eigen3 3.3 REQUIRED NO_MODULE)
target_link_libraries (main Eigen3::Eigen)
```

Note: this requires that Eigen is installed, i.e. exists in `/usr/include/eigen3`
`sudo apt install libeigen3-dev`

Installing Example (Ex06 Benchmark)

Reminder: installing just means copy the files to a specific folder

```
# Benchmark/CMakeLists.txt
cmake_minimum_required(VERSION 3.1)
project(benchmark)
```

```
set(CMAKE_CXX_STANDARD 11)

# If the library is installed in a standard location, we would use find_library
# and not specify this manually
# We choose to do it manually to keep it simpler
if(NOT LIBRARY_INSTALL_PATH)  # CMake was not invoked with -DLIBRARY_INSTALL_PATH.
    set(LIBRARY_INSTALL_PATH ${CMAKE_SOURCE_DIR}/../install)  # hardcode path
endif()

link_directories(${LIBRARY_INSTALL_PATH}/lib) # *Look* for the libraries in that path
include_directories(${LIBRARY_INSTALL_PATH}/include) # Include the library headers

add_executable(benchmark main.cpp)
target_link_libraries(benchmark random timer) # Link the libraries to the executable

install(TARGETS benchmark DESTINATION bin) # Copy the output .exe to the bin/ folder
```

`Random/CMakeLists.txt`, *(identical with* `Timer/CMakeLists`*)*

```
cmake_minimum_required(VERSION 3.1)
project(random)
set(CMAKE_CXX_STANDARD 11)

add_library(random STATIC random.cpp) # Create the library

install(TARGETS random DESTINATION lib) #Copy the library output to the lib folder
install(FILES random.hpp DESTINATION include) #Copy the header to the include folder
```

Note: In this example we need to run cmake and compile the libraries separately, as the benchmark CMake assumes the libraries exist in the `lib` *folder*

4. Classes

Definition of a class (Data declarators)

Classes are collections of “members” representing one entity

Members can be:

- functions
- data
- types

These members can be split into

- `public` accessible interface to the outside.
  Should not be modified later! Only representation-independent interface, accessible to all.
- `private` hidden representation of the concept. Can be changed without breaking any program using the class. Representation-dependent functions and data members.
- `friend` declarators allow related classes access to representation.

Objects of this type can be modified only through these member functions $\Longrightarrow$ localization of access, easier debugging.
The default in a class is always private.
In a struct the default is public.

How to design classes

What are the logical entities (nouns)?
$\Longrightarrow$ classes
What are the internal state variables ?
$\Longrightarrow$ private data members
How will it be created/initialized and destroyed?
$\Longrightarrow$ constructor and destructor
What are its properties (adjectives)?
$\Longrightarrow$ public constant member functions u how can it be manipulated (verbs)?
$\Longrightarrow$ public operators and member functions

Example - Traffic Light

- Property: The state of the traffic light (green, orange or red)
- Operation: Set the state
- Construction: Create a light in a default state (e.g. red) or in a given state
- Destruction: Nothing special needs to be done
- Internal representation: Store the state in a variable
  - Alternative: connect via a network to a real traffic light

```
class Trafficlight {
public:
```

```
    enum light { green, orange, red };
    Trafficlight(light l = red) : state_(l) {} //default constructor
    Trafficlight(const Trafficlight& rhs) : state_(rhs.state_) {} //copy constructor
    Trafficlight& operator=(const Trafficlight& rhs) {
        if (this != &rhs) {
            state_ = rhs.get_state();
        }
        return *this;
    }
    ~Trafficlight(){}
    void print_state() const;

    light get_state() const { return state_; };
    void set_state(light l) { state_ = l; };

private:
    light state_;
};

int main() {
    Trafficlight a(Trafficlight::green); //green
    Trafficlight b(a); //green

    Trafficlight* c_ptr = new Trafficlight(Trafficlight::orange); //orange
    Trafficlight d = *c_ptr; //orange

    Trafficlight e; //red
    Trafficlight& f = a; //green

    Trafficlight::light l; //create variable of type light
    l = Trafficlight::light::red; //change color to red

    l = e.get_state(); //assign state of a Trafficlight to light

    delete c_ptr;
}
```

Example - Point

- Internal state: x- and y- coordinates
- Construction: default: (0,0), from x- and y- values, same as another point
- Properties: distance to another point u, x- and y- coordinates, polar coordinates
- Operations: Inverting a point, assignment

```
class Point {
private:
   double x, y;
public:
   Point(); //default constructor, (0,0)
   Point(double, double); //constructor from two numbers
   Point(const Point&); //copy constructor
   ~Point();   // destructor
```

```
    double dist(const Point& ) const;
    double x() const; double y() const;
    double abs() const;
    double angle() const;
    void invert();
    Point& operator=(const Point&);
};
```

Remember the **rule of three**: If there is any one of the three following, all three have to be defined:

- Copy Constructor (is automatically generated (a.k.a. synthesized) as memberwise copy, unless otherwise specified)
- Destructor (normally empty and automatically generated)
- Copy Assignment

Nontrivial destructor only if resources (memory) allocated by the object. This usually also requires nontrivial copy constructor and assignment operator. (example: array class)

Class terminology

Declaration

`class Point;`

Definition

```
class Point {
   private:
      double x_,y_;
   public:
      Point();
      void Invert();
      ...
};
```

Note: here we have defined the `Point` class but only declared its functions
Declaration generally ends in a `;` whereas definition is the `{}` part

Implementation

```
double Point::abs() const {
     return std::sqrt(x_*x_+y_*y_);
}
```

Constructor (ctor)

```
//preferred method with constructor initializer list, calls the ctors of the members
Point::Point(double x, double y) : x_(x), y_(y) {
   ...
}

Point::Point() = default; //use compiler generated ctor
```

5. C++ Features & Keywords

5.1 Features

Namespaces

These are used to organise classes and functions into 'groups' to prevent name conflicts (classes with the same name). With a conflict it is often uncertain which is used.

```
#include <cmath>
namespace MyNamespace {
    void sin(float);
}

MyNamespace::sin(1); //call like this
std::sin(1);
sin(1); //uncertain
```

Can be nested with scope resolution operator `::` e.g. `namespace A::B::C {}`

Using Namespace:
`using namespace std;`
Is used to prevent writing the namespace each time, but increases risk of name conflicts.
Important! Do **not** use this in header files, as it carries over to where it is included in.

We can also just do this for a single function

- `using std::cout;` now we can write `cout`

Ternary Operator

Shorthand if-else statement, meant to be used as part of an expression

```
//expression ? case-when-true : case-when-false
std::log(a >= 0 ? a : -a); //make value absolute
```

Range-based for loops

Executes a for loop over a range. Works by using the `begin()` and `end()` of a container. Similar to python for loops

```
std::vector<double> v(4);
for (double x : v) // for every double x in the container v
   std::cout << x; //0000
for (int x : {1, 2, 5})
   std::cout << x; //125
for (double& x : v)
   x = 5;
```

Note: if you add/remove elements the *iterators can be invalidated* so the loop can create errors. Use index based loops when modifying the vector. This happens when the vector has to be resized and copied, so the original pointers are invalid. This is not true for all containers (e.g. lists work fine), see cppref Iterator Invalidation or "container" in index

Lambda Expressions

Form: `[value](int x) -> bool {return x > value; };`
`[…]` is called caption
`(…)` are the parameters
`-> bool` is the return type
`{…}` is the instruction to be done

Capture list:

- `[x]` x passed by value (copied)
- `[&x]` x passed by reference
- `[&x, y]` Access to reference of x and copied value of y
- `[&]` Default reference access to all objects in the scope of where lambda is *declared*
- `[=]` Default value access to all objects in the scope of the lambda

5.2 Keywords

Keyword Auto

Type of a variable is implied by the right hand side. Reference and constness works in the same way

```
std::vector<double> v(5);
auto i = v[3]; //double
const auto& r = v[0]; //const double&
```

Just need to be careful with templated expressions from optimisation, as we will have an expression type

Typedef / Using

From C++11 `using` replaces the old `typedef`

```
using element_t = double;
```

Generally **don't avoid** using typedefs. Use them to allow easy modification of your code later on (you only need to change the typedef line and there is no need for refactoring).
Note: the `_t` is used to denote types, like `uint64_t`.

```
class Animal {
   public:
      typedef unsigned short age_t;
       age_t age() const;
   private:
      age_t age_;
};
```

Constants, Mutable & Volatile

`const` **Variable**, Can only be set once with the constructor

Must use *initialiser list* when setting constant members of a class
It calls the constructors of the members (hence you should always use the initialiser list to set variables)

```
struct MyClass {
   const int a;
   MyClass(int value) : a(value) { //<- initialiser list
      a = value; //won't work
   }
};
```

`const` **Member Functions**, prevents modifying of the object inside the function, ie `this` is constant
Only constant functions can be called for constant instances of the class

Note: Calling non-const functions from const functions in the same class is not allowed

```
double Point::abs() const;
double Point::abs() const {...}
```

```
//only const member functions can be called for const objects
const Point P(5,6);
P.invert(); //error: calling non-const function on const object
```

`constexpr` indicates it is possible to evaluate at *compile time*. Used for both variables and functions Different to `const`, which just indicates the value is not changed
Note: `constexpr` is written at the front of the declaration. They should be combined for member functions

```
constexpr void MyFunction();
constexpr void MyMemberFct() const;
```

`mutable` - Allows modification of members even in const objects or inside const functions, *"overrides const status"*

```
struct MyClass {
   mutable int value;
   void func() const {
      value++; //this is allowed
   }
};

int main(){
   const MyClass a;
   a.value++; //allowed
}
```

`volatile` - Tells compiler value may change between every access (read or write) or cause side-effects
Prevents some optimisations

Static Variables

Variables persist through the whole program, are not deleted at the end of the scope
Can be used to prevent a variable from being reinitialised, e.g. in a loop or function

Note: The variable will still only be accessible inside its scope, as usual

```
foo() {
   static int count = 0; //only executed the first time
   count++; //executed every time, value persists from last call
}
std::cout << count; //error: count only accessible in its scope

for(int x = 0; x < 10; x *= 2) {
   static int number_of_times = 0;
   number_of_times++;
}
```

Static Inside Classes

Variable/Function exists once for all instances, does not require an instance of the class to exist
Accessed through class `MyClass::myVar` or directly if inside the class

```
class Genome  {
public:
   static const unsigned short gene_number;
   Genome clone() const; //non-static
   static void set_mutation_rate (unsigned int value){
      mutation_rate = value;
   }

private:
   unsigned long gene_;
   static unsigned int mutation_rate;
};

int main(){
   Genome::gene_number = 128;
   Genome::set_mutation_rate(3);
}
```

Friends

Allowing classes to have access to each others internal representation (private parts).

```
class Vector;

class Point {
   //...
private:
   double x,y,z;
   friend class Vector;
```

```
};

class Vector {
   //...
private:
   double x,y,z;
   friend class Point;
};
```

This also works for functions

```
//written in another class (not Point)
friend Point Point::invert(...); //allow the invert function to access privates
friend int func(...);
friend Point::Point(); //make a ctor a friend
```

*this

`this` is a pointer to the instance with which a member function was called
It is available in all non-static member functions

```
double Point::x() const {
     return this->x; //or (*this).x or simply x
}
```

```
//array copy assignment
const Array& Array::operator=(const Array& arr) {
   //copy the array
   return *this; //return the object with which the function was called
}

int main(){
   Array a;
   a = ...; //now in operator= *this is a
   a = b = c = d; //possible as we always return the instance so it can be chained
   a = (b = (c = d)) //where (c = d) return c
}
```

=default & =delete

These are used to *use/prevent compiler generated* functions, often used for ctors/dtors, operators

```
class Point{
   Point() = default; //shows generate ctor is used
   ~Point() = delete; //we must implement our own dtor
   Point operator=(Point&&) = default; //use generate move assignment
};
```

Misc

- `decltype(variable)` returns the type of the variable or expression, useful with templating

```
std::complex<double> cd;
decltype(cd) other; //other has the same type as cd
```

- `explicit` function: Prevent implicit conversions on the input, types must match the parameters exactly
- Const Pointers:
  - `T* const ptr;` $\Longrightarrow$ pointer itself can't change value of address it stores (similar to reference)
  - `const T* ptr;` $\Longrightarrow$ pointer can move but the integer value must stay the same
  - Note left of the `*` is the type we are pointing to
- Cast-conversion: allows compiler to convert the type of a variable if it knows how to
  - static casting: explicit casting, always use static casting in exam!
  - `static_cast<to_type>(from)`
- Encapsulation: abstraction of use from the implementation, getters/setters

6. Functions & Operators

Inlining

Copies the code of the function and pastes it where it is called, function does not exist after compilation.
Can improve performance if function is called often.
Avoid excessive inlining as it leads to large executable file. Try to avoid with larger functions.

Note that the keyword `inline` is *only a suggestion* to the compiler. For higher compiler optimisation levels, some functions will be inlined even without the `inline` keyword.

When to use:

- Small functions, saves having to rewrite each time, often used for getters/setters
- Large functions, if they are only called once or so *(can be used for better organisation)*

```
class complex {
   private:
      double re_, im_;
   public:
       inline double real() const;
       inline double imag() const;
};

double complex::real() const {
   return re_;
}
double complex::imag() const {
   return im_;
}
```

Functors & Passing Functions

*See **Exercise 8**, Simspson Benchmark, esp. simpson.hpp in benchmark*

Functions can be passed by:

1. **Functor**/Function Object
    - Class with overloaded `operator()`
    - Pass instance of function
2. **Function Pointer**
    - Pass a function directly as a parameter
    - Create function pointer: `return_type (*myfunc) (param1, param2)`
    - For *member* functions use `std::mem_fn`, needs to be wrapped

    ```
    auto greet = std::mem_fn(&Foo::display_greeting);
    greet(f);
    ```

    - Or `std::function<return_type (param1, param2)>` (advanced function pointer)

- Use `auto`

These can be made generic with *templates* or *abstract base class functors*

Operators

*See **Week 4 Slide 31** for all operators and member vs non-member operators*

*See **Week 0506a Slide 51** for rule of 3/5*

```
class Inverse {
   double operator() (double d) { //overload call operator
      return 1.0 / d;
   }
};

Inverse V;
std::cout << V(5.0);
```

Point Example

```
struct Point {
   double x, y;
   const Point& operator+=(const Point& rhs) {
      x += rhs.x;
      y += rhs.y;
      return *this;
   }
   Point operator+(const Point& other) {
      Point tmp{x + other.x, y + other.y};
   }
};

//stream operators usually non-member functions
std::ostream& operator<<(std::ostream& out, const Point& p) {
   out << "(" << p.x << ", " << p.y << ")";
   return out;
}

//we can now use print:
int main(){
   Point p1, p2;
   std::cout << "The point is " << p1 + p2 << std::endl;
}
```

Which Operator to overload

```
A a; ++a; //requires either
const A& A::operator++(); //operator as member
const A& operator++(A&); //or as a non-member

A a; a++; //requires
A A::operator++(int);
//The additional int argument is just to distinguish the pre- and postincrement

A b = a; A b(a); //both use the copy constructor
A::A(const A&);
```

Conversion Operators

Type conversion of A → B as in:

```
A a; B b=B(a);
//can be implemented in two ways:

B::B(const A&);  //constructor
A::operator B(); //conversion operator
```

Keep in mind the implicit type conversion order:
i) `bool, char < int < unsigned int < float < double`
ii) `short < int < long < long long`

7. Traits

Also see http://blog.aaronballman.com/2011/11/a-simple-introduction-to-type-traits/

Recall the template for the min function

```
template <typename T>
T const& min(T const& x, T const& y) {
   return x < y ? x : y;
}
```

We now want to allow things like `min(1.0,2)` and `min(1,2.0)`

Now, instead of doing it manually with 3 different typenames and having to call the function with the return type `min<int>(1.0,2)` or `min<double>(1,2.0)`, we use traits.

```
template <class T, class U>
typename min_type<T,U>::type& min(T const& x, U const& y)
```

The keyword typename is needed here so that C++ knows the member is a type and not a variable or function. This is required to parse the program code correctly – it would not be able to check the syntax otherwise.

How to derive min_type:

```
//empty template type to trigger error messages if used
template <typename T, typename U>
struct min_type {};

//partially specialized valid templates:
template <class T>
struct min_type<T,T> {typedef T type; };

//fully specialized valid templates:
template <>
struct min_type<double,float> {typedef double type; };

template <>
struct min_type<float,double> {typedef double type; };

template <>
struct min_type<float,int> {typedef float type; };

template <>
struct min_type<int,float> {typedef float type; };

//add more specialized templates here
```

```
template <typename T>
struct average_type {
```

```
   typedef typename
   helper1<T,std::numeric_limits<T>::is_specialized>::type type;
};

//the first helper:
template<typename T, bool F>
struct helper1 {typedef T type;
};

//the first helper if numeric limits is specialized for T (a partial specialization)
template<class T>
struct helper1<T,true> {
   typedef typename
   helper2<T,std::numeric_limits<T>::is_integer>::type type;
};

//the second helper
template<class T, bool F>
struct helper2 {
   typedef T type;
};

//the second helper if T is an integer type (a partial specialization)
template<class T>
struct helper2<T,true> {
   typedef double type;
};
```

C++ Concepts

*A concept is a **set of requirements** on types:*

- The operations the types must provide
- Their semantics (i.e. the meaning of these operations)
- Their time/space complexity

A type that satisfies the requirements is said to *model* the concept. A concept can extend the requirements of another concept, which is called *refinement*. The standard defines few fundamental concepts, e.g.

- CopyConstructible
- Assignable
- EqualityComparable
- Destructible

8. Exceptions

Overview:

- Use *asserts* when *testing* your code, check errors coming from your *own code* or for *high performance*
- Use *exceptions* when writing code that is used by others, when the error *can be resolved/handled*, you want to give more data on the error

Exceptions enable good error handling, especially with use of libraries or functions. There are some more primitive ways of handling errors than exceptions:

- Global error variables/Flags (*not recommended*)
- Functions returning error codes or objects

Global variables: *Typical for older C-code*,
Set a global variable when an error occurs, then check at other places if the error has occurred.
This typically fills the code with a lot of seemingly random checks. Cannot call in parallel.

Functions returning error code: *Typical for big APIs (OS level or http)*,
Functions exit when an error occurs and return a 'status' variable. Then check the status code with if-statements at the place of calling, instead of *try-catch*.
Problems:

- Makes it harder to return other values or give detail on the error.
- Need to handle deleting of memory manually

Asserts *terminate the program if a condition is not met*,
These should only be used for internal testing, when something is violated that means the program cannot operate. However, these are good when testing *your own* code. Asserts can be turned off for performance, use the `-DNDEBUG` compile flag. E.g. checking invariants in your algorithm, necessary pre-conditions for internally used functions.

Exceptions: *Change the control flow*,
Exceptions ensure the function is left safely (deletes allocated memory) with an error message and allows *handling of the error*. It will keep exiting the functions up the stack until it is caught (works on multiple levels of function calls). Exceptions can hence be used to print a stack-trace, to see where the error occurred and what led to it.

How to use:

- `try` will attempt to execute the enclosed code, when an exception is thrown the scope is exited safely and can be caught. It will not execute the code in the `try` scope below where the error occurred.
- `catch(Type e)` written after a `try` if we know how to resolve a particular type of error. Write one per *type* of error
- `catch(...)` Catch all exceptions, should be the last `catch`. We do not have a parameter as we don't know the type of what we are catching. It is usually a sign of a fatal error.
- `throw <object>` to signal an error an leave the function. We can *re-throw* the error in a catch. Can throw any type, but `std::exception` is recommended.
- Use `e.what()` to get the message from an error (named `e`)

*When **not** to use:*

- Do not use exceptions to change the *control flow* of your program.
- Avoid exceptions in *destructors*

Example

```
void my_function(){
   if(n<=0)
      throw "n too small"; //throwing a char array
   if(index >= size)
      throw std::range_error("description");
}

try{
   my_function(); //only put code in here which is likely to throw an exception
}
catch(std::logic_error& e){  //Handle a logic_error
   std::cout << e.what();
   throw e; //re-throw error like this
}
catch(...){ //catch all other error types
   std::cerr << "A fatal error occurred.\n";
}
```

Standard exceptions are in `<stdexcept>`, derived from `std::exception`.
(`<exception>` is part of `<stdexcept>` and only contains the base class `std::exception`)

Logic errors (base class `std::logic_error`) error due to incorrect semantics. Violation in pre-condition, invariants

- `domain_error`
- `invalid_argument`
- `length_error`
- `out_of_range`

Runtime errors (base class `std::runtime_error`) error beyond what the function was intended to handle

- `range_error` an invalid value occurred as part of a calculation
- `overflow_error` a value got too large
- `underflow_error` a value got too small

9. Timer

The standard library contains:

- Header `<ctime>`: date & time
- Header `<chrono>` (since C++11): precision time measurements

Benchmarking example:

```
//requires #include <chrono>
std::chrono::time_point< std::chrono::high_resolution_clock > t_start, t_end;
t_start = std::chrono::high_resolution_clock::now();

//put code to benchmark here

t_end = std::chrono::high_resolution_clock::now();

std::cout << "It took: " << static_cast<std::chrono::duration<double> >(t_end -
t_start).count()
        << " seconds." << std::endl;
```

10. Random Number Engines

Random numbers

Real random numbers: are hard to obtain. It can be done from physical phenomena

Pseudorandom numbers:

- Get random numbers from an algorithm
- Totally deterministic and therefore not random at all, useful for debugging
- But maybe good enough for your application
- Never trust (just one) (pseudo) random number generator

Utilities for dealing with random numbers in standard library since C++11: Header `<random>`

Useful and good generators:

```
#include <random>
//Mersenne-twisters
std::mt19937 rng1;
std::mt19937_64 rng2;

//lagged Fibonacci generators
std::ranlux24_base rng3;
std::ranlux48_base rng4;

//linear congruential generators
std::minstd_rand0 rng5;
std::minstd_rand rng6;
```

```
//set the seed of a RNG
#include <random>

//default random engine
std::default_random_engine e;
e.seed(42); // set seed
```

Distributions

Uniform distributions:

- Integer: `std::uniform_int_distribution<int> dist1(a,b)`
- Floating point: `std::uniform_real_distribution<double> dist2(a,b)`

Exponential distribution:

- `std::exponential_distribution<double> dist3(lambda);`

Normal distribution:

- `std::normal_distribution<double> dist4(mu, sigma);`

11. Data Structures in C++

Arrays

Are consecutive range in memory. Fast arbitrary element access. Profits from cache effects. Constant time insertion and removal at the end. Searching in sorted array is O(ln N). Insertion and removal at arbitrary position in O(N).

- **C array**
- **vector**
- **valarray**
- **deque:**
  The deque is more complicated to implement, but yields constant time insertion and removal at the beginning.

Linked lists

- **list:**
  A linked list is a collection of objects linked by pointers in sequence. Constant time insertion and removal anywhere (just reconnect the pointers). Does not profit from cache effects. Access to an arbitrary element is O(N), searching is also O(N).
  Functions:
  - `splice` joins lists without copying, moves elements from one to end of the other.
  - `sort` optimized sort, just relinks the list without copying elements
  - `merge` preserves order when "splicing" sorted lists
  - `remove(T x)`
  - `remove_if(criterion)` criterion is a function object or function, returning a bool and taking a const T& as argument.
    Example:
    - `bool is_negative(const T& x) { return x<0; }`
    - `list.remove_if(is_negative);`

Trees

- **map**
- **set:**
  Unordered container, entrys are unique.
- **multimap:**
  Can contain more than one entry (e.g. phone number) per key.
- **multiset:**
  Unordered container, multiple entries possible

Queues and Stacks

- **queue:**
  The queue works like a Mensa, FIFO (first in first out). In constant time you can push an element to the end of the queue, access the first and last element and remove the first element.
  Functions:
  - `void push(const T& x)` //inserts at end
  - `void pop()` //removes front
  - `T& front()`, `T& back()`, `const T& front()`, `const T& back()`
- **priority_queue:**
  The priority_queue is like a Mensa, but professors are allowed to go to the head of the queue (not passing other professors!). The element with the highest priority is the first one to get out. For elements with equal priority, the first one in the queue is the first one out. Prioritizing is done with < operator.
- **stack:**
  The stack works like a pile of books, LIFO (last in first out). In constant time you can push an element to the top, access the top-most element and remove the top-most element.
  Functions:
  - `void push(const T& x)` //insert at top
  - `void pop()` //removes top
  - `T& top()`
  - `const T& top() const`

Generic traversal of containers

We want to traverse a vector and a list in the same way:

```
for (auto p = a.begin(); p != a.end(); ++p)
   cout << *p;

//Specific to Array<T> (replace T if necessary)
for (typename Array<T>::iterator p = a.begin(); p != a.end(); ++p)
   cout << *p;
```

Note: We can replace `auto` with the containers `iterator`, but this makes it less generic

Array implementation:

```
template<class T>
class Array {
   public:
      typedef T* iterator;
      typedef unsigned size_type;
      Array();
      Array(size_type);

      iterator begin(){
         return p_;
      }
      iterator end() {
         return p_+sz_;
      }
```

```
   private:
      T* p_;
      size_type sz_;
};
```

Linked list implementation:

```
template <class T>
struct node_iterator {
   Node<T>* p;
   node_iterator(Node<T>* q) : p(q) {}
   node_iterator<T>& operator++() {
      p=p->next;
      return *this;
   }
   T* operator ->() {
      return &(p->value);
   }
   T& operator*() {
      return p->value;
   }
   bool operator!=(const node_iterator<T>& x) {
      return p!=x.p;
   }
   // more operators missing ...
};

template<class T>
class list {
   private:
      Node<T>* first;
   public:
      typedef node_iterator<T> iterator;
      iterator begin() {
         return iterator(first);
      }
       iterator end() {
         return iterator(0);
      }
};
```

12. Algorithms Overview

Mostly found in `<algorithm>` header, some also from `cmath`, `numeric`
See ***'Algorithms Library'*** in cppreference!

Example - linspaced vector (1-10), square each element, shuffle vector, copy range, print each element

```
#include <algorithm> //for_each, transform
#include <numeric>   //iota

int main() {
   std::vector<int> v(10);
   std::iota(v.begin(), v.end(), 1); //start 1, spacing 1 to get (1, 2, 3...10)

   auto squarelam = [](int x){ return x*x; }; //square the element
   std::transform(v.begin(), v.end(), v.begin(), *squarelam); //*apply* for each
   //v = 1 4 9 16 25 36 49 64 81 100

   std::random_shuffle(v.begin(), v.end()); //randomise v

   std::vector<int> v1(5);
   std::copy_n(v.begin() + 2, 5, v1.begin()); //copy elements [2, 2+5) to v1

   auto printlam  = [](int x){ std::cout << x << " "; };
   std::for_each(v1.begin(), v1.end(), *printlam); //call print for each element
   //output: 9 16 25 36 49

   std::for_each(animals.begin(), animals.end(), std::mem_fn(&Animal::grow));
   //use mem_fn if the function is a member
   //required as else we do not know what grow is, could be a var
}
```

Non-modifying:

- **for_each(begin, end, function)**
- find, find_if, find_first_of
- adjacent_find
- count, count_if
- mismatch
- equal
- search
- find_end
- search_n

Modifying:

- **transform(begin, end, output begin, function)**
- copy, copy_backward
- swap, iter_swap, swap_ranges
- replace, replace_if, replace_copy, replace_copy_if
- fill, fill_n

- generate, generate_n
- remove, remove_if, remove_copy, remove_copy_if
- unique, unique_copy
- reverse, reverse_copy
- rotate, rotate_copy
- **random_shuffle**

Sorted sequences:

- sort, stable_sort
- partial_sort, partial_sort_copy
- nth_element
- lower_bound, upper_bound
- equal_range
- binary_search
- merge, inplace_merge
- partition, stable_partition

Permutations:

- next_permutation
- prev_permutation

Set Algorithms:

- includes
- set_union
- set_intersection
- set_difference
- set_symmetric_difference

Minimum and Maximum:

- min
- max
- min_element
- max_element
- lexicographical_compare

13. Templates

Function Overloading

We can declare functions with the same name if they have a *different number/types of parameters*.
The compiler will choose the matching version of the function

```
int    min(int a, int b)       { return a<b ? a : b; }
double min(double a, double b) { return a<b ? a : b; }

min(1,3);      //calls min(int, int)
min(1.0,3.0);  //calls min(double, double)
```

Problems arise with implicit conversions

```
min(1, 3.14); // Problem! which type?
min(1.0, 3.14); // OK
min(1, int(3.14)); // OK but does not make sense
// could also define new function "double min(int,double);"
```

During compilation, this works as they don't really have the same name. The function argument types are appended to the function name, e.g. our integer min is `_Z3minii` (`i` for integer)

Note: For generic functions use templates.
This is more used when there are different number of arguments, but it may be best to use default argument values, i.e. `int min(int a, int b = 0);`

Generic Algorithms

We can implement the example above generically by using a template:

```
template <typename T>
T const& min (T x, T y)
{
  return (x < y ? x : y);
}
```

What happens if we want to use a mixed call in the example above `min(1,3.141)`?
→ Now we need to specify the first argument since it cannot be deduced

```
min<double>(1,3.141);
min<int>(3,4);
```

Advantages of using templates are, that we get functions that:

- work for many types T
- are as generic and abstract as the formal definition
- are one-to-one translations of the abstract algorithm

Templated Classes

```
template <typename T>
class A {
   public:
      typedef T T2; //alternate name for T
       A(); //constructor
       //...
      T func1(T x) {return x; } //definition
      T func2(T); //declaration only
   private:
      //...
};

//define a member outside of the class body:
template <typename T> T A::func2(T x) {
   //...
}
```

```
template <typename T>
class sarray {
   public:
      typedef std::size_t sz_t;  //size type
      typedef T elem_t;          //element type
         // ... as before
   private:
      sz_t size_;    //size of array
      elem_t* elem_; //pointer to array
};
```

`std::size_t` is simply an unsigned integer type that can store the maximum size of a theoretically possible object of any type.

If we want to create a fully compatible array class, we use typedef to create our generic types:

```
template <typename T>
class sarray {
   public:
      typedef std::size_t size_type; //size type
      typedef T value_type; //value / element type
      typedef T& reference; //reference type
      typedef T const& const_reference; // const reference type
         // ...
   private:
      size_type size_;   //size of array
      value_type* elem_; //pointer to array
};
```

Now we need to add the keyword `typename` to our operator overloads, or the compiler won't know that the member is a type and not a variable or function.

```
// subscript operator
template <typename T>
typename sarray<T>::reference sarray<T>::operator[](size_type index) {
   assert(0 <= index && index < size());
   return elem_[index];
}

// const subscript operator
template <typename T>
typename sarray<T>::const_reference sarray<T>::operator[](size_type index) const {
   assert(0 <= index && index < size());
   return elem_[index];
}
```

Template Specialization

Take as an example an array of bools. Internally it is stored as one byte for every entry. We want to specialize this such that every bool only takes up one bit of storage.

```
template <class T>
class sarray {
   //generic implementation
   //...
};

template <>
class sarray<bool> {
   //optimized version for bool
   ...
};
```

14. Inheritance

See Exercise 9 Inheritance

- Store shared behaviour/variables in a base class
  - Can then create derived classes to extend the base class for a more specific case
- Form of Polymorphism, can look at instances in a generic sense, but treat them differently
  - We can have a vector of the base class, but actually have a vector of varying types

Properties:

- `virtual` allows the function to be *overriden* (modified) by derived classes
  - For derived instances the derived function will be called
- `override` not required but enables compiler checks to check for a matching virtual function
  - Only a matching declaration is needed to override a virtual function
- Call base class function from the derived class equivalent with the scope modifier `BaseClass::func()`
- The function remains virtual for a class deriving from the derived class, sticky
- Can inherit from any class and have multiple base classes

```
struct Animal{
   double age;
   virtual void Eat();
};

struct Fish : public Animal {
   void Swim();
   void Eat() override; //modifies the base function
};

struct Bird : public Animal {
   void Fly();
};
```

Here `Fish`, `Bird` are *derived* classes and `Animal` is the *base* class.

What is not inherited:

- Anything Private
- Con/Destructors, Assignment operators
- Friends

Access Specifiers

We can set access levels of variables and functions. These define who can read/write/call the variables/functions
Important: Derived classes will still inherit *all* the functionality and data

- **public** anyone can access (read/write/call)
- **protected** only accessible to derived classes

- **private** only the class itself has access
    - Instances of the derived class will have these variables/functions, but they are not accessible directly or via member functions of the derived class. They can only be seen by the member functions of the base class.

We can set also an access specifier on the inheritance itself. This sets the *max* access level of the inherited variables/functions

The default access specifier for the base class is the same as the default for `class` (i.e. `private`) or `struct` (i.e. `public`). Usually we want to inherit publicly, as we want to have the same access levels as the base class

Example:

```
class A
{
public:
    int x;
protected:
    int y;
private:
    int z;
};

class B : public A
{
    // x is public
    // y is protected
    // z is not accessible from B
};

class C : protected A
{
    // x is protected
    // y is protected
    // z is not accessible from C
};

class D : private A    // 'private' is default for classes
{
    // x is private
    // y is private
    // z is not accessible from D
};
```

Note: Classes B, C, D all contain the variables x, y, z. It is just question of access, can we modify/read

Example from https://stackoverflow.com/a/1372858/9295437

Abstract Base Classes (ABC)

Prevent instances of the base class itself by forcing overriding of functions for derived classes
Using `= 0` for a function forces the derived class to override the function. Makes the class abstract

Important: Must pass ABC by reference/pointer, as pass by value creates a copy which is not allowed

```
#include <iostream>

struct MyBase{
   virtual void print() const{}
   virtual void run() =0; //must be overriden, defined in derived class
};

struct Derived: public MyBase {
   void print() const override{
      //extra functionality
      MyBase::print(); //call base function of run, works as print is not abstract
   }
   void run() override {}
};

struct OtherDerived: public MyBase{
   //print() does not have to be overriden
   void run() override{}
};

void Func(MyBase& b){ //must not pass by value, else compiler error
   b.print(); //will call the derived version
}

int main(){
   MyBase b; //Compiler error: class is abstract
   Derived a;
   a.run(); //calls Derived::run
}
```

15. Programming Styles

Run-time vs Compile-time Polymorphism

Run-time (Dynamic)	Compile-time (Static)
Uses **virtual functions** (Inheritance)	Uses **templates**
Decision at run-time	Decision at compile-time
Works for objects derived from the common base	Works for objects with the right members
One function created for the base, class $\Longrightarrow$ **saves space**	A new function is created for each class used $\Longrightarrow$ **takes more space**
Virtual function call **needs lookup in type table** $\Longrightarrow$ **slower**	No virtual function call, **can be inlined** $\Longrightarrow$ **faster**
Extension possible using only definition of base class	Extension needs definitions and implementations of all functions
Most useful for **application frameworks**, user interfaces, "big" functions	Useful for **small, low level constructs**, small fast functions and generic algorithms

Comparison on programming Styles

The following chapter will show the different advantages and disadvantages of programming styles by means of the implementation of a stack

Procedural programming

Procedural programming is one of the more simple programming paradigms. It is used in many different programming languages.
See simpson integration in Ex 1.6 for more details

```
void push(double*& s, double v){
    *(s++) = v;
}
double pop(double *&s) {
    return *--s;
}
int main() {
    double stack[1000];
    double* p = stack;
    push(p,10.);
    std::cout << pop(p) << "\n";
    std::cout << pop(p) << "\n";
    // error of popping below
    // beginning goes undetected!
}
```

Modular programming

The modular implementation of a stack allows transparent change in underlying data structure without breaking the user's program.
See simpson integration in Ex 2.2 for more details

```
namespace Stack {
   struct stack {
       double* s;
       double* p;
       int n; };

   void init(stack& s, int l) {
       s.s=new double[l];
       s.p=s.s;
       s.n=l; }

   void destroy(stack& s) {
       delete[] s.s; }

   void push(stack& s, double v) {
       if (s.p==s.s+s.n-1) throw
           std::runtime_error("overflow");
       *s.p++=v; }

   double pop(stack& s) {
       if (s.p==s.s) throw std::runtime_error("underflow");
       return *--s.p; }
}

int main() {
Stack::stack s;
Stack::init(s,100); // must be called*
Stack::push(s,10.);
Stack::pop(s);
Stack::pop(s);      // throws error
Stack::destroy(s);  // must be called
}
```

Object orientated programming

By implementing the stack as a class, one is able to encapsulate the data and make use of the automatic initialisation and cleanup of the class.
See simpson integration in Ex 4.2/5.3 for more details

```
namespace Stack {
class stack {
    double* s;
    double* p;
    int n;
public:
```

```
    stack(int=1000); // like init
    ~stack(); // like destroy
    void push(double);
    double pop();
};

int main() {
    Stack::stack s(100);    // initialization done automatically
    s.push(10.);
    std::cout << s.pop();   // destruction done automatically
}
```

Generic programming

By templating the class and implementing it generically you can ensure that the stack works for any data type. It also creates an efficient datatype because of the fact that the instantiation happens at compile time.
See simpson integration in Ex 8.1 for more details

```
namespace Stack {
template <class T>
class stack {
    T* s;
    T* p;
    int n;
public:
    stack(int=1000); // like init
    ~stack(); // like destroy
    void push(T);
    T pop();
};

int main() {
    Stack::stack<double> s(100); // works for any type!
    s.push(1.3);
    std::cout << s.pop();
}
```

16. Hardware

We need to understand *hardware optimisations and limitations* so we know how to optimise in C++

16.1 CPUs

Basic components of the CPU:

- **Memory controller**: Manages loading from and storing to memory
- **Registers**: Fastest memory storage inside CPU, very small, can store integers, floating point numbers, specified constants
- **Arithmetic and logic units** (ALU):
    - Perform arithmetic operations and comparisons
    - Operates on registers (fast)
    - On some CPUs can directly operate on memory (slow)
- **Fetch and decode unit**: Fetches instructions from memory, interprets the numerical value, decides what to do and sends them to the ALU or memory controller

Instruction Sets

Every processor has a fixed instruction set determined by the hardware, essentially a list of all fundamental operations it can do. Every program uses only these operations, which can be seen in the assembly code.

There are two alternatives, either to have a large instruction set (CISC) with lots of possible operations. Or create everything from a small set of instructions (RISC). Both instruction sets can perform any operation/program and can be considered equivalent, depends on usage case.

CISC: Complex Instruction Set Computer

Instruction set that implements many high level instructions (e.g sin, cos). They usually have high clock cycles but take many cycles to complete instructions

RISC: Reduced Instruction Set Computer

Use low level instructions to compute everything.
They execute instructions very quickly and can be pipelined well, but need a large amount of assembly instructions to create programs $\implies$ More memory usage

Von Neumann Bottleneck

A computers 'speed' is always limited by the slowest component, called bottleneck. The CPU cannot perform operations if it does not have the data for it. Usually this bottleneck is loading data from memory or the hard-drive into the CPU, called the *von Neumann* bottleneck

This is important as it is often what we need to optimise the most.

Developments over time
Processor speeds have increased significantly. Memory improvements have mostly been in density – the ability to store more data in less space – rather than transfer rates.
As speeds have increased, the processor has spent an increasing amount of time idle, waiting for data to be

fetched from memory. No matter how fast a given processor can work, in effect it is limited to the rate of transfer allowed by the bottleneck, the whole system needs to be fast.

16.2 CPU Optimisations

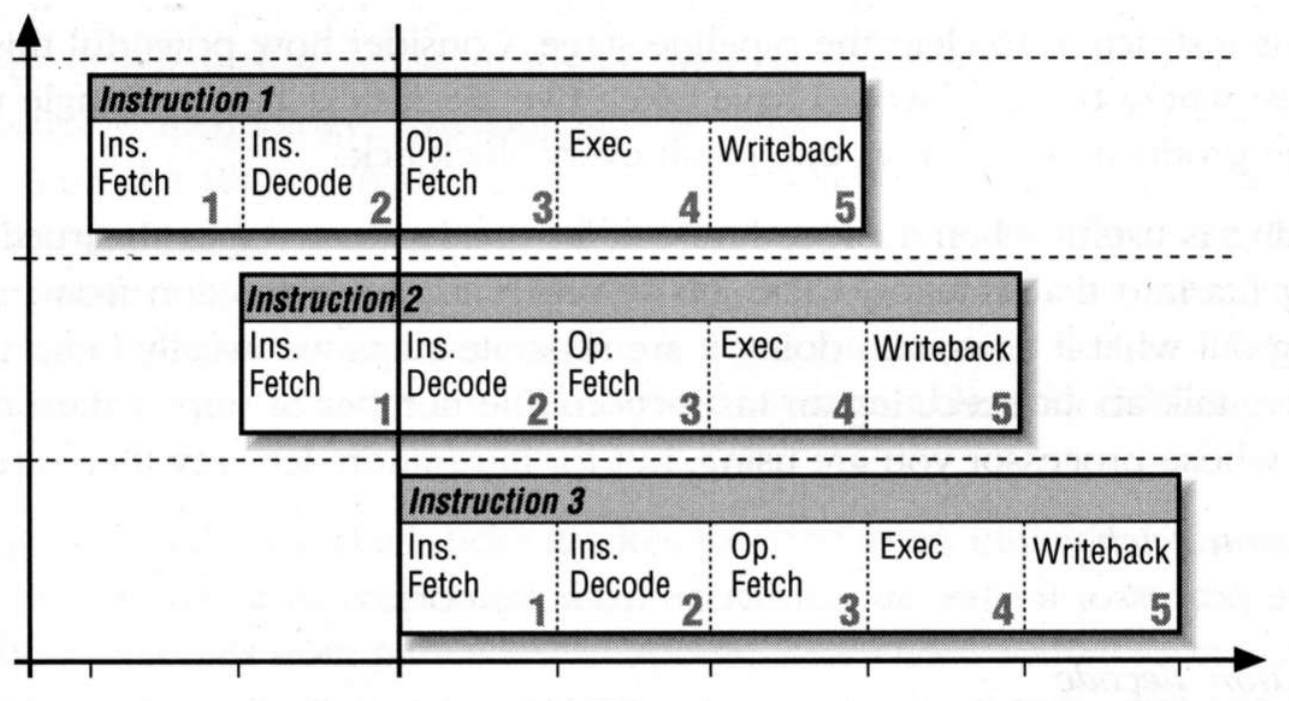
Specifically Single Core

Beware that CPUs will further optimise the binary machine code to execute it as fast as possible, these are some of the tricks it does or can do.
They are based on: utilising all resources fully (pipelining), predicting (branch prediction) and Instruction Level Parallelism *ILP* (Pipelining/Vectorisation)

These optimisations are related to processing, but there are also memory optimisations with similar concepts, e.g.
Predicting - temporal locality (keep newest in cache) and spatial locality/pipelining (read cache line)
Parallelisation - using a bus, read multiple data at once

Pipelining

A form of ILP
Pipelining is the process of increasing performance by running consecutive *independent* instructions before the previous ones are finished in the same processor core (i.e. without threading). CPUs do this with their circuits, they send input signals into the circuit before the previous signal has finished. Only works if we structure our code correctly.



Loop unrolling is a way of making the operations independent, allows for pipelining to happen. But branch prediction is also used for this...

Branch Prediction

Pipelining is not possible if there are branches in the program (if-statements), since we need to know the output before starting the next operation.
CPUs circumvent this problem by **predicting/guessing** the result of the branches and simply starting the execution of the most likely branch (e.g. for loop, continuing the loop).

If predicting is correct, then the pipeline continues as normal but has saved time. Else, if the prediction is wrong, the pipeline aborts and starts again at the right branch.
Sometimes even both branches are computed and the wrong one is discarded.

Vectorisation

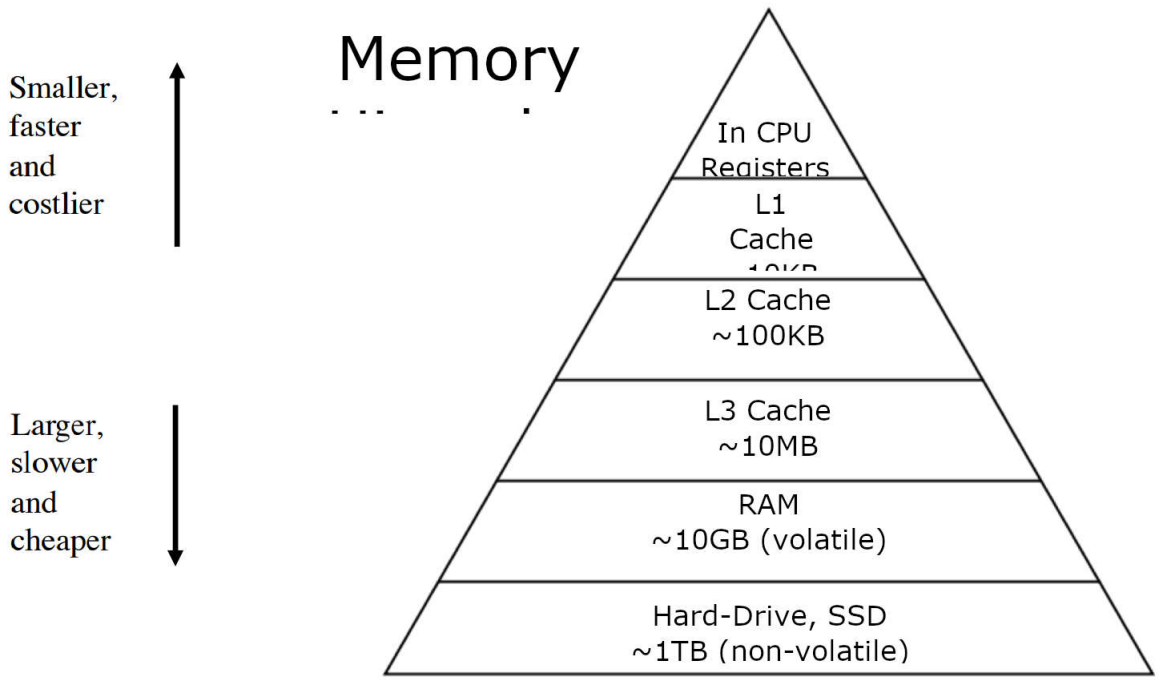
Various Names/Versions: MMX, SSE, SIMD(Single Instruction Multiple Data), AVX
Also a form of Instruction Level Parallelism (ILP)
Vector operations perform the same operation on multiple values at the time. Typical operations are elementwise addition/multiplication on vectors and matrices. The compiler will attempt to vectorise (`-ftree-vectorize` flag), but it is best to do it manually

GPUs

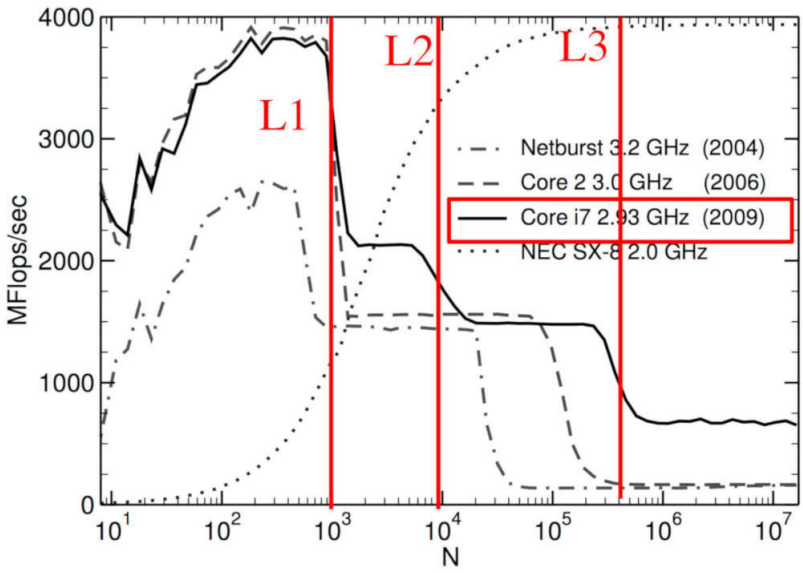
GPUs are specifically designed for performing vectorised (SIMD) instructions.
They have many cores (100-3000) but the cores are less complex, and can only do basic operations. Less for logic, more for lots of data. They are more difficult to program due to parallelisation, need to use other languages e.g CUDA, OpenCL, CG. Sharing memory between CPU and GPU is also a problem.

16.3 Memory

As faster memory is expensive, we have a hierarchy of different sized memory to prevent the CPU waiting for data. At the top is the storage actually used by the processor (registers). The less memory a program needs, the faster it runs as we can fit most of the program into a higher cache (not necessarily faster, but reduces the memory bottleneck)



We can see how there is a slowdown when each cache level is full:



Random Access Memory (RAM)

Note: We often ambiguously use 'memory' to refer to RAM

There are many different types of varying size, price and speed.

- **SRAM:** static RAM, very fast, expensive, data stored on flip-flops, data stays as long as there is power
- **DRAM:** dynamic RAM, slower, much cheaper than SRAM, data stored on tiny capacitors, capacitors need to constantly be recharged
- **SDRAM:** *synchronous* dynamic RAM, synchronised to cache cycle, faster than DRAM
- **DDR RAM:** double data rate RAM, can send data twice per clock cycle
- **Interleaved Memory Systems:** uses multiple memory banks
  - Can read from all simultaneously $\implies$ same latency, more band width

Caches

As the bottleneck is usually reading data from the hard-drive/memory, we cache data read in several levels (L1-L3). The CPU will check if the data it needs exists in the memory hierarchy from the top downwards. When checking in a level, if the data exists it is called a *cache hit* (else *cache miss an*d the search continues, slower)

16.4 Memory Optimisations

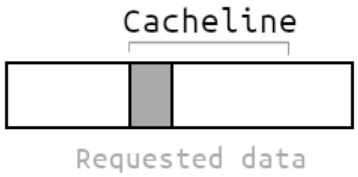
Two *Assumptions*:

1. **Temporal Locality** - Data tends to be reused, data currently being used will be used in future.
   - So we keep data in the cache until it is full
   - We discard oldest data first
2. **Spatial Locality** - Data needed next is near the data currently being used
   - i.e. data use is chronologically ordered, e.g. arrays
   - Read data surrounding data as well $\implies$ cache line

Note: These optimisations occur between/on levels of the cache. Test values with `$ getconf -a`

Cache Line

When reading data into a higher level in the memory hierarchy, there is always an initial delay. So it is best to read more data at once and request less often. Due to **spatial locality** we read the data requested plus data around it in a block. This block has a fixed size called **cache line size**. Each level has its own cache line size but they are usually equal.



When the CPU reads data from memory, a **full cache line is loaded** into the L3 cache and further up the memory hierarchy. This means that the data itself and more data sequentially after it is in the cache. So if we need the surrounding data next, we saved the initial delay as it is already in cache.

So in C++ we should organise the data/usage so it is reads sequentially, so the data read in the cache line is always used. E.g. matrix row vs column major needs to match in matrix storage and reading sequence.

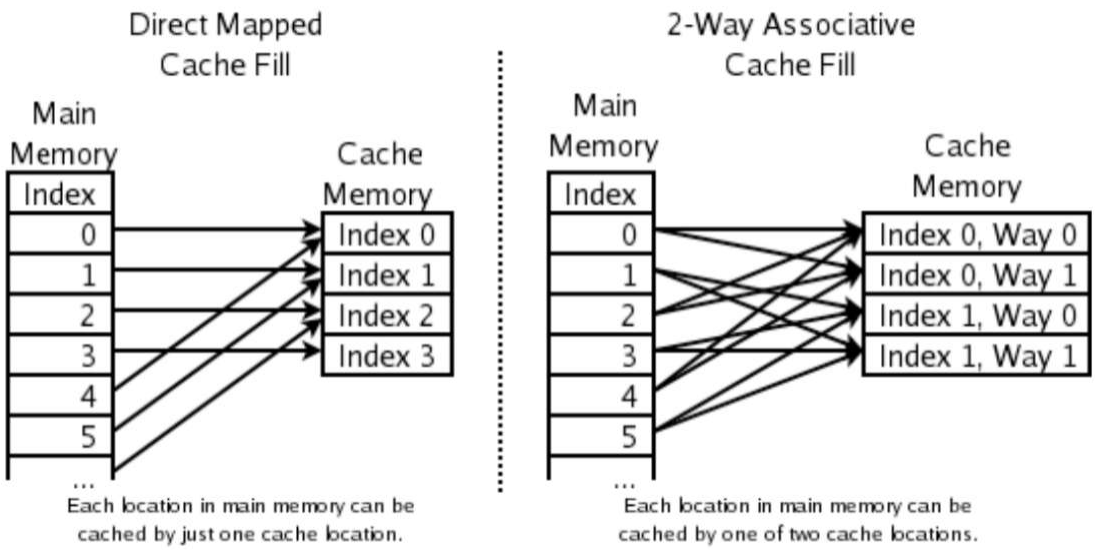
Note! 64/32-bit is not related to the cache line size, it is about the number of bits used for addresses in the memory.

Cache Associativity

Cache associativity is how data is *mapped* from a larger to a smaller sized storage, similar to hashing. Direct mapping is the cheapest as it requires least complex calculations (and so circuits).

- **Direct mapped** - Each memory location can only be stored at **one** cache location.
  - Address in cache can be calculated with modulo, $index_{\text{cache}} = index_{\text{memory}} \% size_{\text{cache}}$
- **n-way associative** - Each memory location can be stored at **n** cache locations
  - E.g. does a similar modulo, then a linear search in the *next n elements* for the oldest/empty location
- **Fully associative** - Each memory location can be stored at **any** cache location
  - Performs a linear search through the whole cache

Due to temporal locality, if an index is full, we discard the oldest.



Note: It is often easier to look at the n-way associativity as a grid, with y-axis being the 'Index', x-axis the 'Way'. The number of ways is the **n**.
An address is assigned to an index and does search through all the ways in that index.

Cache associativity is useful as it allows us to rearrange the data from the cacheline and **prevent cache thrashing**, which is when we constantly replace data we are using. Means we have to re-read each time, defeating the concept of caching.
Cache thrashing is the worst when the stride size equals the cache line size. Larger strides may improve performance, due to the modulo. With higher associativity we have this less.

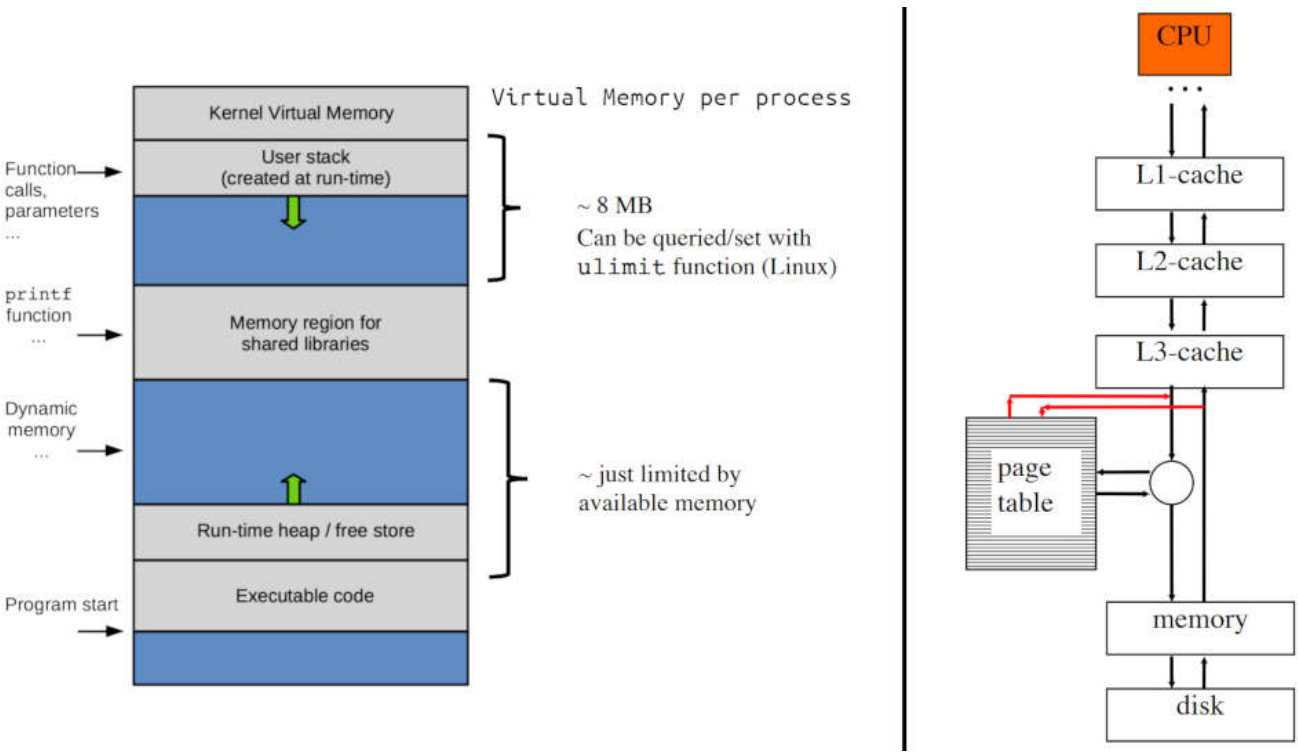
E.g. in image above, what happens if we compare every 4th element and have a cache line size of 4. We will always fully replace the previous data the cache and have to re-read each time. With 2-way we will always have the last 2 in cache, for full-associativity we will have the last 4.

Virtual Memory

The abstraction of mapping logical memory onto physical memory addresses. Handled by hardware+OS

1. Processes run independently, for each process it appears it is running on its own
    - Prevents memory conflicts/overlaps between processes
2. We can extend RAM when full with the slower hard-drive
    - Prevents crashing when memory is full
    - The mapping from virtual to physical memory addresses is stored in **page tables** in the RAM
    - To get data we need to first read address from page table $\implies$ slow
        - So we cache the page table, called Translation Lookaside Buffer (TLB)

Note: Does not apply to caches, only to RAM. When the caches are full we replace old data



Worst Case Caching

Always try to reuse data as much as possible. Worst case when fetching data:

- Request an address
- Cache miss in L1-3
- Lookup physical address
    - Cache miss in TLB (translation lookaside buffer)
    - Request page table entry
    - Load page table from memory (slow)
- Page fault in page table
    - Store a page to disk (extremely slow)
    - Create and initialise a new page (very slow)
    - Load page from disk (extremely slow)
- Load value from memory (slow)

17. Optimisation

> Do easy and significant optimisations first

Optimal approach to optimisation:

1. Use **compiler flags** *(see section 1 C++ compiling)*
2. Use **profiling** to determine slow parts of program
3. Find best data structure & algorithm and a **library** which implements it
4. Look into more advanced optimisations
    1. Inlining
    2. Read data in a sequential order, to use data in cache line effectively
    3. Unroll loops
    4. Vectorisation/SIMD (single instruction multiple data e.g. MMX, AVX)
    5. Lazy evaluation and Expression Templates, compile time calculations
    6. Use processor specific intrinsics

17.1 Profiling Runtime

1. Time program in CLI with `time`
2. More detailed profile with `gprof`, to see calls and runtime of individual functions
3. Time **inside C++** with `chrono` (Manual Instrumentation)
4. Use a more advanced profiler, e.g. `pref` or vtune, or also profile memory usage etc

A. `time` is a simple shell command which returns the runtime of a program. It is the simplest way to time your program.
You can add the option `-p` in order to get the time in seconds

```
$ time program.exe
real    0m0.872s            # effective total time
user    0m0.813s            # cputime of the program
sys     0m0.016s            # cputime of other software whilst the program is running.
```

B. `gprof` can profile C++ more exactly, timing individual functions

1. Compile the program with `-pg`
2. Run it, this creates the `gmon.out` file
3. View results with `gprof`

```
$ g++ myProgram.cpp -pg
$ ./a.out
$ gprof ./a.out gmon.out      # optionally add '| less' to view with pager
```

1. *A function will not appear in the output if it is never called or has been inlined with compiler optimisations*
2. *There may be some inaccuracies if running times are short, e.g. %s over 100, this is due to approximations*

C. Manual instrumentation the `<chrono>` header enables "manual instrumentation" or timing in a program. Run the program/function multiple times and average to reduce noise
See Timer chapter

```
std::chrono::high_resolution_clock::now();  // get the current time, do before/after
```

17.2 General Optimisations

Datastructures and Algorithms

Use (correct) **STL containers** wherever possible

- Arrays/Vectors: fast random access
- Lists: fast insertion, slow traversal
- Trees: middle ground, fastest if both features are needed

Use (correct) library for an algorithm. Many available at Netlib.org
The main advantages of professional libraries:

- bug free, thoroughly tested
- optimised
- well documented (at least better than your code)
- support most architectures

Loop unrolling

```
//Dot product
double s = x[0] * y[0] + x[1] * y[1] + x[2] * y[2];

//instead of
for (int i=0; i<3; ++i)
    s += x[i] * y[i];
```

Loop unrolling creates faster code for two main reasons:

- No control statements/branches in the loop
- Easier to pipeline

Partial loop unrolling is also possible:

```
for (int i=0; i<N; i+=4) {
   a[i]   = b[i]   * c[i];
   a[i+1] = b[i+1] * c[i+1];
   a[i+2] = b[i+2] * c[i+2];
   a[i+3] = b[i+3] * c[i+3];
}
```

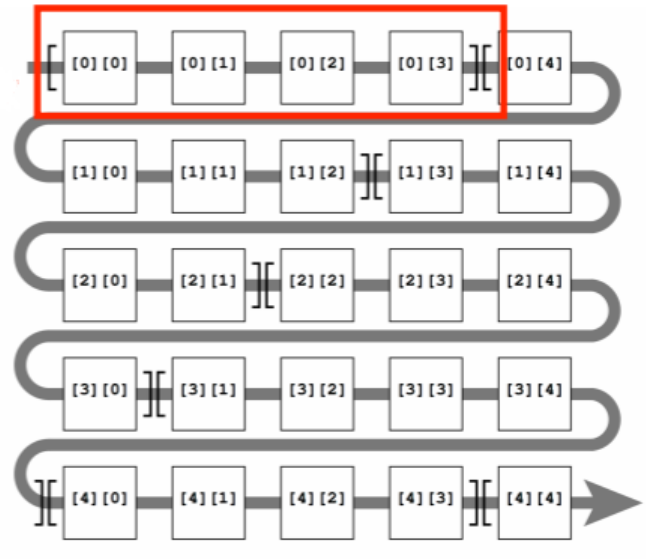
Compiler Optimisations

See Week 11 Slides 19-33
Note: Functions may be inlined before the optimisations are done, e.g. getters

See 1.2 for compiler flags

Storage order and optimising cache

See Cache Chapter 17.
Multi dimensional arrays are generally stored linearly in memory. C/C++ uses **row-major** order (depicted), Fortran, Matlab use column-major. Red box represents one cache line



For large matrices you can use **block multiplication** to insure that the calculations can remain in-cache

17.3 Template Meta Programming

Using templates to execute (parts of) a program or optimise it ***at compile time***
Examples: calculating constants, loop unrolling, expression templates

Also see [Wikipedia]

I. Calculating Constants - Factorial

We can store the value in several ways: enums, static consts or with functions
Result will be fully calculated at *compile time* in all cases (some require more optimisation)

A. Result stored in the `enum` (or `static const` in comments)

```
template<int i>
struct Factor{
   enum { Result = i * Factor<i-1>::Result };
   //static const int Result = N * Factorial<N-1>::Result;
   //static constexpr int Result = N * Factorial<N-1>::Result;
};

template <>
struct Factor<1>{
   enum { Result = 1 };
   //static const int Result = 1;
   //static constexpr int Result = 1;
};
```

```
int main(int argc, char const *argv[]) {
   std::cout << Factor<5>::Result; //retrieve value like this
}
```

B. Using `static functions` in a struct, template number `N` stored by struct

```
template<int N>
struct Factorial{
    static int f(){
        return N * Factorial<N-1>::f();
    }
};

template<>
struct Factorial<0> { //specialised for N=0, use as a base case for the recursion
    static int f() {
        return 1;
    }
};

int main() {
   std::cout << Factorial<5>::f(); //can call directly as it's static
}
```

C. Using normal functions

```
template<int N>
int f(){
   return N * f<N -1>();
}

template<>
int f<0>() {
   return 1;
}

int main() {
    std::cout << f<5>();
}
```

Note: It can be easy to confuse normal template parameters with meta-template parameters (N here)
E.g. when making factorial work for `T` *not just* `int`

II. Loop Unrolling

Use meta-programming to unroll loops, optimising for short loops (removed loop overhead)

See Week 11a Slides 9-15

- Use `meta_dot` to recursively expand the cwise operation
  - Hence we need a base case $\implies$ template specialisation for `I = 0`
- Uses a static function similar to C. above

- Generic for any size(`I`) *and type(`T`)*

III. Expression Templates

Store an expression in an object and only evaluate at end, often used in Linear Algebra
Removes writing temp results, define how to best evaluate an expression at *compile time* $\implies$ faster
Essentially *Templated Lazy Evaluation*

Sidenote: Do not use the `auto` keyword with expression templates, as it will take the type of the expression object

General Idea:

1. Overload operators to return an expression object (store what needs to be evaluated)
2. Say how to evaluate expression object best, either
    1. `operator=` in the vector class
    2. `operator[]` in the expression class, to evaluate just a single element

*See non-templated **Lazy Evaluation**: Week 11a Slides 20-21*

- `vectorsum` is the expression object

*See proper **Expression Template**: Week 11 Demos etvector.hpp*

- `X` is the expression object, is generic for any LHR, RHS and operator
- Have a class per operator, stores how to apply the operation
- `operator+` at end sets up expression

18. BLAS & LAPACK

Fortran libraries are very fast and have been optimised over a long time. We can use them similar to a normal C++ library. BLAS is for basic linear algebra, LAPACK extends BLAS

Naming Convention:

Function names follow `PTTXXX`

- `P` - Precision of the data, `D`=double, `S`=Single, `C`=Complex, `Z`=Double Complex
- `TT` - Matrix Type, `GE`=General, `SY`=Symmetric, `HE`=Hermitian
- `XXX` - Operation Name, e.g. `EV`=Eigen Values

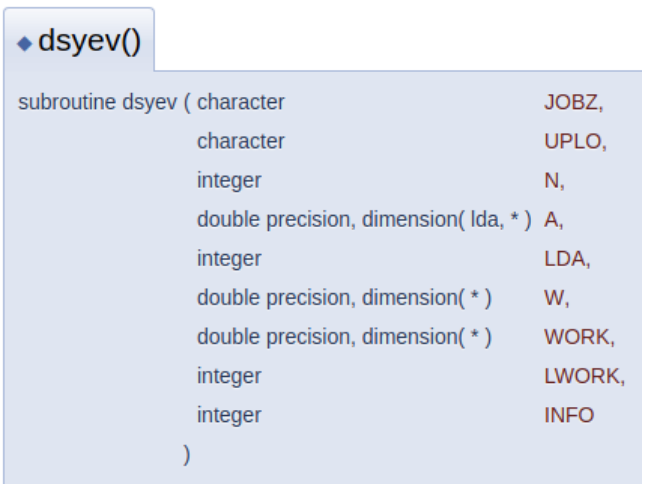
Note: Fortran is case insensitive

How to use:

1. **Declare function** in C++, with correct naming and parameters, see docs(!)
    1. We only pass references or pointers of built-in data types, *references by default*
    2. Need to surround declaration with `extern "C"{...}` to prevent name mangling when compiling
    3. Function names *end with an underscore* (except with certain compilers)
    4. Functions have a `void` return type, but return data through the referenced parameters, so we can return *multiple* variables, e.g. LU decomposition returns both matrices
       Worth looking at what the inputs and outputs are in the docs
2. Call function in C++ normally
3. **Link libraries** when compiling
    1. Link library itself, `-llapack` or `-lblas`
    2. Link the Fortran compiler called gfortran with `-lgfortran` *(not always required)*

Example

1. a) Find docs



◆ dsyev()

```
subroutine dsyev ( character                          JOBZ,
                   character                          UPLO,
                   integer                            N,
                   double precision, dimension( lda, * ) A,
                   integer                            LDA,
                   double precision, dimension( * )   W,
                   double precision, dimension( * )   WORK,
                   integer                            LWORK,
                   integer                            INFO
                 )
```

1. b) Declare the function in C++ by copying the docs

```
extern "C" {   //Function ends in underscore
    void dsyev_( const char& JOBZ   //Reference by default
               , const char& UPLO
               , const int& N
               , double* A          //Pointer as signified by the (*) in the docs
               , const int& LDA
               , double* W
               , double* WORK
               , const int& LWORK
               , int& INFO );
}
```

2. Call the function

```
std::vector<double> M(N * N, 0); // pre-allocate the memory for the output in C++!
std::vector<double> omega(N);
std::vector<double> exact_omega(N);
double D_LWORK;

dsyev_('V', 'L', N, M.data(), N, omega.data(), &D_LWORK, -1, info);
      //use vector.data() to get a pointer to the first element, as with an array
      //as we need to pass by pointer
```

4. Link library in compilation, link `gfortran` last

```
$ g++ main.cpp -llapack -lblas -lgfortran
```

OR with CMake

```
target_link_libraries(mytarget lapack blas gfortran) #link blas in the same way
```

OR link a pre-compiled library, e.g. *on Mac we can use accelerate*

```
find_library(ACCELERATE_LIB Accelerate )
mark_as_advanced(ACCELERATE_LIB)
target_link_libraries(mytarget ${ACCELERATE_LIB})
```

19. Input / Output

19.1 Formatting Streams

Streams will first convert the value to a string then print/store it, we can control this conversion with `<iomanip>` or `<iostream>`

Use these like: `cout << std::setw(5)` similar to `std::endl`

- **`setw(5)`** sets width/characters to print for the next value, *non-sticky*
- **`setfill('0')`** set fill character, so output matches defined width
- **`left`**, **`right`**, **`internal`** sets the text alignment *by setting position of fill characters to the opposite*
  - `internal` is works for specific types, see example
- **`scientific`** prints floats in scientific notation, i.e. with exponent
- **`setprecision(6)`** changes the float precision, significant digits

Note: Functions are sticky unless specified, meaning they affect all following output, not just the next value

Examples

```
std::cout << std::setw(4) << std::setfill('0') << std::right << 5;
//output: 0005

std::cout << std::setw(4) << std::setfill('#') << std::left << 5;
//5###

//adding in std::
cout << setw(8) << setfill('@') << internal << setprecision(2) << -1.234;
//-@@@@1.2
```

```
for (int i = beg; i < end; ++i) { //Common in loop to print a table
   cout << std::scientific;
   cout << std::setw( 5) << i << " "
        << std::setw(15) << std::pow(M_PI,  i) << " "
        << std::setw(15) << std::pow(M_PI,2*i) << std::endl;
}
```

Sidenote: Streams should be passed by a non-const reference, cannot be copied

19.2 File Streams

File IO in CLI

Programs can be executed in sequence in the terminal by redirecting output, called **pipelining**

- `$ a.exe | b.exe | c.exe` - `b` has the input from `a` etc...

We can **redirect** the output to a *file*, same for input

- `$ ./simulation.exe > data.txt` output to file

- `$ ./a.out < input` input from file

You can redirect only one stream:

- `1>` stdout
- `2>` stderr
- `&>` error & output

File IO in C++

`ofstream` behaves similar to `std::cout`, same for `ifstream` ('of' for 'output file' stream)

```
#include <iostream>
#include <fstream>
int main () {
   std::ofstream outFile("output.txt");       //Open file with filestream
   outFile << "Hello";
   outFile.close();                           //Close the file once finished(!)

   int a;
   std::ifstream inFile;                      //Same for ifsteams
   inFile.open("input.txt");
   inFile >> a;
   inFile.close(); //Close also called in dtor at end of scope, best to close yourself
}
```

Can set different *modes* for an `fstream` in the ctor/open:

- `fstream::in` input mode or `out`, already set by default for `ifstream` or `ofstream`
- `fstream::binary` binary mode
- `fstream::ate` output position starts at end
- `fstream::app` append to end of file
- `fstream::trunc` truncate, overwrite existing file *(default mode)*

```
std::ifstream in;
in.open ("input.txt", std::fstream::binary);

std::fstream out; //note only an fstream not ofstream, so need to set out mode
out.open ("output.txt", std::fstream::out | std::fstream::ate);

//input and output also possible
std::fstream file("out", std::ios::in | std::ios::out | std::ios::ate);
```

Note: The bitwise-or `|` is used to set multiple options as a bitmask

HDF5

Common format for storing data for input/output, is quite complicated at first

- Use this for larger datasets
- Enables parallel I/O, compression, multiple files in Groups (file system)

20. Python

- **Interpreted** - when run, code is 'compiled' line by line directly from source code (`.py`)
  - Can run in a shell, e.g. *ipython*
- **Untyped, Dynamic variables** - type is not declared *explicitly* and the variables can change type
  - Classes are not required to have all their variables defined initially
- Variables are references by default, almost all things in Python are **objects**, including functions
  - Has a **Garbage Collector**, do not need to manage memory ourselves
- There is **no privacy**, just follows convention of underscores
  - `var` public, `_var` private, `__var` strictly private
- Naming conventions follow *PEP8*, generally everything is lowercase, underscore separated except classes being CamelCase
- See the Python cheatsheet

20.1 Python Modules

Modules are roughly equivalent to libraries in C++. Any `.py` file is a module, do not need to do anything special to use it as a module. We can `#include` them by using `import`

```
import file        # Imports whole file, similar to #include, then use file.function1()
import path.file   # Can use a relative path, use . instead of /
import numpy as np # Can rename imported file/function/class
from file import function1, function2 # import only specific functions/classes
     # Can call function1 directly like this -> may get name conflicts
```

Note: a package is a collection of modules, whereas a module is usually a single `.py`

Instead of a `main()` in C++ we can have several 'entry points' in the project. The module which is called directly has its name set the main

```
if __name__ == "__main__":          # equivalent to main() in C++
   # This .py was executed directly, not imported
```

We can write the code without this, but then the code is executed when we import the `.py` as a module

Common Packages:

- **NumPy** - Fast package for multi-dimensional arrays with linear algebra functions
  - See the NumPy Cheat Sheet, **SciPy** extends NumPy
- **Matplotlib** - 2D plotting library
- **H5py** - Python interface for HDF5 file format
- **Mpi4py** - Message Passing Interface. Allows parallel programming with Python
- **SymPy** - Computer Algebra System, enables symbolic mathematics

20.2 Classes

The constructor is a magic method named `__init__`

- As we **cannot overload functions** in python, we create a **single ctor** with all possible parameters, then do a case distinction. *Remember: arguments have default values and we can set them explicitly*

```
class Genome(object):
   def __init__(self, genome=None):
      if genome is None:
         self.genes = np.zeros(self.__class__.gene_size, dtype=bool)
      else:
         self.genes = copy.deepcopy(genome.genes)

# Can construct with either
a = Genome()
b = Genome(a)
c = Genome(genome=a) # setting args explicitly is more useful when there are more
```

Static variables

- Create by defining inside class, not in a function
- *Inside class*, access with `self.__class__.static_var`
  - Do not use `static_var` directly, it sees it as a local variable (so may get conflicts)
- Only use `MyClass.static_var` *outside the class*, as it can be overriden by an instance specific variable within the class

```
class Genome(object):
   gene_size = 64 # static class variable, shared by all instances
   def __init__(self):
      self.genes = np.zeros(self.__class__.gene_size, dtype=bool)

# Access from outside class
Genome.gene_size = 128
```

Also see link

Copying Instances

- As everything is a reference, if we assign a variable to another we are creating an equality
- `copy.copy(instance)`, does a **shallow copy**
  - First layer of variables is cloned, but these may be references to more objects
- `copy.deepcopy(instance)`, does a **deep copy**, usually what is wanted
  - This 'clones' everything recursively

```
import copy # required module for copying
x = [0,1,2]
y = x       # makes x and y equivalent, y is a reference to x essentially
y[2] = 666
print(x) # [0, 1, 666]     x is also changed
print(y) # [0, 1, 666]
```

```
x = [0,1,2]
y = copy.copy(x)
y[2] = 666
print(x) # [0, 1, 2]       x unaffected
print(y) # [0, 1, 666]

x = [0,1,[2,3]] # Note 3rd elem of array is a *reference* now to another array
y = copy.copy(x)
y[2][0] = 666
y[0] = 333
print(x) # [0, 1, [666, 3]]      x[0] unaffected but x[2] has changed
print(y) # [333, 1, [666, 3]]

x = [0,1,[2,3]]
y = copy.deepcopy(x)
y[2][0] = 666
print(x) # [0, 1, [2 , 3]]
print(y) # [0, 1, [666, 3]]
```

Inheritance

- Classes always inherit everything, as there is no privacy, everything is virtual
- We can call base methods with the `super()`
  - Only derived function is called by default, including in `__init__`
- Functions are overriden if the names match

```
class A:
   def __init__(self):
      self.a = 1
   def print_A(self):
      print(self.a)

class B(A): # B inherits from A
   def __init__(self):
      super().__init__() # self.a == 1
      self.b = 2
   def print_B(self):
      print(self.b)

b = B()
b.print_A() # 1, so print_A is inherited and base class constructor is called to set a
b.print_B() # 2
```

20.3 Decorators

Decorators are functions that modify the functionality of other functions. We apply these with the `@decorator` in front of a function definition

Built-in Decorators

Static Functions

- `@staticmethod` - makes a function static, does not receive the `self`
- `@classmethod` - *like static method* but also has the `cls` argument
  - `cls` is the class type we call the function with
  - Useful with inheritance, as we can get the derived classes

Property
This is makes the function behave like a getter, but it allows us to use the function as if it were a variable, so don't need to write the parentheses `()`

```
class Point:
   ...
   @property   # Like a getter, note: @x.getter does not exist
   def x(self):
      """The point's x coordinate"""
      return self._x

   @x.setter  # Setter for the property x, unrelated to _x
   def x(self, val):
      if val < 0: # e.g. Validate the new value in setter
         val = 0
      self._x = val

P1 = Point(1.,3.)
print("P1.x =", P1.x) # dont need to write brackets, i.e. P1.x(), use like a variable
P1.x = 2    # Set like a variable
```

- Define a function `myproperty` that returns a value, like a getter, then decorate with `@property`
  - The corresponding setter is then `@myproperty.setter`
  - `@myproperty.getter` does **not** exist, it is fulfilled by the `@property`

Custom Decorators

Here we define ourselves how a decorator modifies a function. The key is `f = mydecorator(f)`

```
def mydecorator(func):
    def inner():
         print('Hello', end=' ')
         func()
    return inner # when called we return the *function* inner as the modified function

@mydecorator # f = mydecorator(f)
def f():
    print('World')

f() # Hello World
```