C++ Basics

Get a feel for the language

- get comfortable with using a C++ standard reference
- think of C++ as a compound of 5 parts
  - everything inherited from C
  - the C++ 98 core language
  - the Standard Template Library (STL)
  - C++ templates
  - the C++ 11 core language
- if you are a beginner: ignore the template part for this lecture
- use a reference: I can recommend this one

Use CMake to organize your projects

- compiler/platform independent build management tool
- when developing a software project there is more to it than just YOUR code
  - you will use libraries (other than the STL)
    in gcc you would have to write:
    `gcc -I <library-header-directory> -L <library-binary-directory> -D <library-flags> …`
    for every(!) unique library path you are using
  - you will compile several configurations (fast for release builds, verbose debug version during development, …)
  - you want to test your software and automate the testing process, especially as your project grows
- CMake will generate the Makefile (under Linux, NMake scripts or MSVC projects under windows, etc.) with all these aspects respected FOR YOU
- please: don’t write your Makefiles by hand ...

Prefer nullptr over NULL

- in C++, NULL is just a Macro replacement for the integer 0
- the intention of NULL is to use it as the “null-pointer” literal
- but it can be used EVERYWHERE where an integer type is expected
- this can lead to undesired behavior in conjunction with overloaded functions
  - think of pointer and integer overloads for a function foo and a call to foo with NULL passed
- in C++98, the recommendation was to just not overload a function this way
  - hard to verify automatically
  - can be necessary sometimes
- in C++11: use nullptr, a literal that ONLY matches against pointer types
- besides the technical aspect, it is also more readable

C++ Random number generation

- in the C days one could be tempted to generate random numbers like this

```
#include <ctime>
#include <cstdlib>
int main() {
  std::srand(std::time(nullptr));
  const int rnd_nr = std::rand();
}
```

- this comes with quite a few drawbacks
  - std::time in most implementations only has a resolution of 1 second →
    running a program twice within the same second will generate the same numbers;
    not unlikely on a highly concurrent server environment
  - std::srand only takes an unsigned it, which is 32-bit on most platforms
    that is too little entropy, and you cannot give it more, even if you wanted to
  - std::rand() produces values from 0 to RAND_MAX, a constant that is 32767 on most platforms, which is a very small range

C++ Random number generation (2)

- std::rand is typically implemented as a Linear Congruential Generator, which are known to have a short period → numbers repeat after small cycles, among many other defects
- many programmers rely on modulo as a means to create random numbers within the range [0, N)

```
const int rnd_nr = std::rand() % N;
```

  - even if std::rand() would create a uniform distribution (which it does not) the distribution resulting from a modulo like above is almost always(!) non-uniform

C++ Random number generation (3)

- in C++11 the STL was augmented with the header <random>
- now we have reliable generators and a vast collection of distributions right within the STL

```
#include <random>
int main() {
  std::mt19937 gen(std::random_device{}());
  std::uniform_real_distribution<> dis(0, 1000);
  // generate random number within [0, 1000)
  const double x = dis(gen);
}
```

Prefer std::cout over std::printf

- std::printf is a relict from the C world, as is hinted by its header <cstdio>
- basic data types are covered in printf, but even then their usage is rather cryptic

  ```
  std::printf("%3.2f\n",d);
  ```
- structs/classes cannot be printed in a straight-forward way
- type-safety: the type printed is known at compile time for std::cout, whereas it is dynamically deduced at runtime for printf via % markers
- redundancy = error-prone: number of %-markers have to match the argument list
- inheritance: once a operator<< is defined for std::ostream, it can be used in any standard output context (file, memory buffer, console, …) inherited from std::ostream

Make things as simple as possible, but not simpler

- include all headers that you use in your file
  - e.g. if you use std::size_t, include <cstddef>, even though you might also use <vector> which includes <cstddef> already → <vector> could be removed and all of a sudden a non-vector related line in the code breaks
- don’t include anything more than necessary (compile time!)
- if you don’t need command line parameter passing, remove arguments from main; if you don’t alter the fault code, remove the return 0 line

```
int main() {}
```

vs.

```
int main(int argc, char** argv) {
  return 0;
}
```

- automize the process using tools like [clang-tidy]()

Code is mostly for humans, not machines

- your code is read many more times than it is written
- optimize for readability!
- use style guides, to create uniformity within your team or community
  - I recommend the Google C++ style guide for this course
  - you can also automate formatting using the clang-format tool
    clang-format -style=Google input-source.cc > output-source.cc
- keep the namespace prefix, especially in header files
  - avoid premature use of *using namespace xxx*; → hard to detect errors when used in header files, but also hinders readability everywhere else →
    regex foo; → is this from boost or does this refer to the C++11 version ???
- read the Google C++ style guide (roughly) at least once to get a feel for what makes good readability

Don’t use Variable-length arrays (VLA)

- was an extension to gcc for a long time and included into C99
- surprisingly, is available by default in g++, too!

  ```
  #include <cstdlib>
  int main(int argc, char** argv) {
    const int N = std::atoi(argv[1]);
    int array1[N];
  }
  ```
- will compile without a warning unless you set the proper flags:

  ```
  -ansi -pedantic -Wall -Werror
  ```
- space for the array is allocated on the Stack, which is rather limited in size → your program will usually crash already on array sizes of two million elements
- just use std::vector for dynamically allocated arrays (on the Heap)

  ```
  #include <vector>
  int main(int argc, char** argv) {
    const int N = std::atoi(argv[1]);
    std::vector<int> array1;
  }
  ```

Measuring time

- you will need to time your code very often in this course
- ancient online tutorials still mostly use C-style timing code

```
#include <ctime>
int main() {
  clock_t tic = clock();
  foo();
  clock_t toc = clock();
  std::printf("Elapsed: %f seconds\n", (double)(toc - tic) / CLOCKS_PER_SEC);
}
```

- or even worse

```
#include <sys/time.h>
int main() {
struct timeval  tv1, tv2;
gettimeofday(&tv1, NULL);
gettimeofday(&tv2, NULL);
std::print::("Total time = %f seconds\n",
             (double) (tv2.tv_usec - tv1.tv_usec) / 1000000 +
             (double) (tv2.tv_sec - tv1.tv_sec));
}
```

Measuring time (2) - Use <chrono>

```
#include <chrono>
void foo() {}
using std::chrono::system_clock;
using std::chrono::duration;
int main() {
  auto start = system_clock::now();
  foo();
  auto end = system_clock::now();
  const double elapsed_seconds = duration<double>(end - start).count();
}
```

- using chrono you get the highest time resolution your hardware provides
- easier to read and comprehend without crazy time arithmetic
- type-and thread-safe, in contrast to <ctime>

Reading arguments from the command line

- don’t make your program interactive, unless it REALLY needs to be (rarely the case), i.e. bad example

```
#include <iostream>
int main() {
  int x;
  std::cout << "Please type your favorite number: ";
  std::cin >> x;
}
```

- it is harder to use scripts/pipes around your program, among other things
- prefer using argc/argv (or even better: a library like gflags)

```
#include <cstdlib>
int main(int argc, char** argv) {
  if (2 != argc) {
    std::cout << "usage: " << argv[0] << " <integer>" << std::endl;
    return -1;
  }
  int x = std::atoi(argv[1]);
}
```

Avoid using unsigned

- students in the past often used the unsigned integer type
- often confused with a semantic that it doesn't provide
  - non-negativity is NOT enforced by the compiler

    ```
    void f(unsigned) {}
    int main() {
     f(-1);
    }
    ```

    will compile and run with its wrap-around value
  - underflow errors are hard to catch

    ```
    #include <vector>
    int main() {
     std::vector<int> vec;
     for (unsigned i = vec.size() - 1; i >= 0; --i) ;
    }
    ```

    will never terminate
  - it does not give you that much more range: only 1 bit!
- use assertions if you NEED non-negativity

Avoid manually managing memory

- to allocate dynamic memory in C++ one can call new

  ```
  int * x = new int;
  double * y = new double[10];
  ```

- in order to return the memory to the system, delete needs also to be called

  ```
  delete x;
  delete[] y;
  ```

- by using new/delete you enter the world/hell of exception-safety
  you need to make sure the matching delete calls are always reached
  → hard to guarantee, especially as a beginner
- simply avoid it by using STL containers, especially vector for memory management
  - by doing so, you can rely on the RAII principle to take care of all the cleanup necessary
- if you REALLY need to manually allocate memory, wrap it around a smart pointer (shared_ptr, unique_ptr)

Misc

- prefer enum classes over enums
  ```
  enum class {BLUE, RED};
  ```
  regular enums stem from C and are implicitly convertible to int
- if you need compile time constants, still don’t use C-style enums, use constexpr: `constexpr double PI = 3;`
- if you pass an argument (to a function, constructor, lambda, …) just for reading: pass it by const-reference, and not copy! See here
- for readability: only one variable declaration per line!
  ```
  int x, y;  // don't do this
  ```
- readability/teamwork: both comments and variable-, function- or class names should be written in English using ASCII characters

Vector vs. List