PROGRAMMING IN C++

Jülich Supercomputing Centre

13 – 17 May 2024 | Sandipan Mohanty | Forschungszentrum Jülich, Germany

Chapter 7

Type erasure

TYPE ERASURE TECHNIQUE

```
1  auto f(int i) -> PolyVal;
2  void elsewhere() {
3      std::vector<PolyVal> v;
4      v.push_back(1);
5      v.push_back(2.0);
6      v.push_back("Green"s);
7
8      for (auto&& elem : v) {
9          func1(elem);
10     }
11     PolyVal X = f(i);
12 }
```

- Polymorphic behaviour attained using a class hierarchy and virtual functions...
  - is extensible by simply inheriting from the `Base` type and overriding the virtual functions
  - But, it has "reference semantics", so that we can not return those polymorphic objects by value from functions
  - Built in types can not be accommodated into the same hierarchy
- `variant` provides a solution to the two problems above, but we need to commit to a fixed number of polymorphic types in the problem, from the outset
- `std::any` is a library provided facility for type erasure

TYPE ERASURE TECHNIQUE

```
1  void func1(int x);
2  void func1(double x);
3  void func1(std::string x);
4  auto f(int i) -> PolyVal;
5  void elsewhere() {
6      std::vector<PolyVal> v;
7      v.push_back(1);
8      v.push_back(2.0);
9      v.push_back("Green"s);
10
11     for (auto&& elem : v) {
12         func1(elem);
13     }
14     PolyVal X{3.141};
15     // func1(X) should go to func1(double)
16     X = PolyVal{"some string"s};
17     // func1(X) should now go to func1(string)
18     X = f(i);
19     // func1(X) should redirect according to what
20     // polymorphic value f happens to return
21 }
```

- We want a type `PolyVal`, so that we can store different types of entities in it
- A uniform container of `PolyVal` should be able to hold values of different types
- When a certain instance is used, it should still be able to behave according to the value it is currently holding.
- We should be able to copy a `PolyVal` object using normal copy construction or copy assignment in such a way that the copy of a `PolyVal` storing a `Triangle` would still behave as a `Triangle`

TYPE ERASURE TECHNIQUE

```
1  class PolyVal {
2      struct Internal {
3          virtual ~Internal() noexcept = default;
4          virtual auto clone() const -> std::unique_ptr<Internal> = 0;
5          virtual void func1_() const = 0;
6      };
7      template <class T>
8      struct Wrapped : public Internal // continued...
9
10 public:
11     template <class T>
12     PolyVal(const T& var) : ptr{ std::make_unique<Wrapped<T>>(var) } {}
13     PolyVal(const PolyVal& other) : ptr { other.ptr->clone() } {}
14 private:
15     std::unique_ptr<Internal> ptr;
16 };
```

- Make a normal class with an internal class with virtual functions defining the desired interface, and another internal wrapper class template deriving from the internal base
- Give the outer class one template constructor (unrestrained here to isolate the TE technique)

TYPE ERASURE TECHNIQUE

```
1  class PolyVal {
2      struct Internal {
3          virtual ~Internal() noexcept = default;
4          virtual auto clone() const -> std::unique_ptr<Internal> = 0;
5          virtual void func1_() const = 0;
6      };
7      template <class T>
8      struct Wrapped : public Internal // continued...
9
10 public:
11     template <class T>
12     PolyVal(const T& var) : ptr{ std::make_unique<Wrapped<T>>(var) } {}
13     PolyVal(const PolyVal& other) : ptr { other.ptr->clone() } {}
14 private:
15     std::unique_ptr<Internal> ptr;
16 };
```

- Let the class contain a smart pointer to this base, but initialise that member using a class template which inherits from the internal base.
- Implement a copy constructor for `PolyVal` by using a virtual `clone()` function for the internal class
- Use the template constructor to create a wrapped object containing a copy of the input parameter

TYPE ERASURE TECHNIQUE

```
1   class PolyVal {
2       template <class T>
3       struct Wrapped : public Internal {
4           Wrapped(T ex) : obj{ex} {}
5           ~Wrapped() noexcept override {}
6           auto clone() const -> std::unique_ptr<Internal> override
7           {
8               return std::make_unique<Wrapped>(obj);
9           }
10          void func1_() const override { func1(obj); }
11          T obj;
12      };
13  };
```

- The internal wrapper should store an object of the template parameter type
- It should provide copy, clone etc.
- It should redirect function calls in our original requirement to free functions

TYPE ERASURE TECHNIQUE

```
1   class PolyVal {
2       template <class T>
3       struct Wrapped : public Internal {
4           Wrapped(T ex) : obj{ex} {}
5           ~Wrapped() noexcept override {}
6           auto clone() const -> std::unique_ptr<Internal> override
7           {
8               return std::make_unique<Wrapped>(obj);
9           }
10          void func1_() const override { func1(obj); }
11          T obj;
12      };
13  };
```

- As long as those free functions exist for a type `F`, it will be possible to create objects of `PolyVal` type from type `F`

Exercise 7.1:

`examples/PolyVal.cc` contains the code corresponding to the slides shown here. Verify that we achieve our purpose of having a copyable object preserving polymorphic behaviour. Add a function `func1()` (processing a new type) into the mix, and extend the existing setup.

Exercise 7.2:

Sequences of objects with polymorphic behaviour is a frequently occurring programming problem. We have seen one example before, with a vector of `unique_ptr<Shape>`, filled with newly created instances of types inherited from `Shape`, such as `Circle`, `Triangle` etc. The problem can be solved in many alternative ways. `examples/polymorphic` contains 4 sub-directories with different approaches to the geometric object example. (i) Inheritance with virtual functions (ii) `std::variant` with visitors (iii) Using `std::any` (iv) Custom type erasure.

Chapter 8

Modules

A PREVIEW OF C++20 MODULES

Traditionally, C++ projects are organised into header and source files. As an example, consider a simple `saxpy` program ...

```
#ifndef SAXPY_HH
#define SAXPY_HH
#include <algorithm>
#include <span>
template <class T> concept Number = std::floating_point<T> or std::integral<T>;
template <class T> requires Number<T>
auto saxpy(T a, std::span<const T> x, std::span<const T> y, std::span<T> z){
    std::transform(x.begin(), x.end(), y.begin(), z.begin(),
        [a](T X, T Y) { return a * X + Y; });
}
#endif
```

```
#include "saxpy.hh"
auto main() -> int {
    //declarations
    saxpy(10., {inp1}, {inp2}, {outp});
}
```

PROBLEMS WITH HEADER FILES

- Headers contain declarations of functions, classes etc., and definitions of inline functions.
- Source files contain implementations of other functions, such as `main`.
- Since function templates and class templates have to be visible to the compiler at the point of instantiation, these have traditionally lived in headers.
- Standard library, TBB, Thrust, Eigen ... a lot of important C++ libraries consist of a lot of template code, and therefore in header files.
- The `#include <abc>` mechanism is essentially a copy-and-paste solution. The preprocessor inserts the entire source of the headers in each source file that includes it, creating large translation units.
- The same template code gets re-parsed over and over for every new translation unit.
- If the headers contain expression templates, CRTP, metaprogramming repeated processing of the templates is a waste of resources.

MODULES

- The C++20 `module`s offer a better organisation
- All code, including template code can now reside in source files
- Module source files will be processed once to produce "precompiled modules", where the essential syntactic information has been parsed and saved.
- These compiled module interface (binary module interface) files are to be treated as caches generated during the compilation process. There are absolutely no guarantees of them remaining compatible between different versions of the same compiler, different machine configurations etc.
- Any source `import`ing the module immediately has access to the precompiled syntax tree in the precompiled module files. This leads to less overall work and faster compilation of individual translation units
- Since a source file may export a module to be imported by another source in the same project, sources must sometimes be compiled in a specific order. Automatically deducing this order is a difficult problem, and is one of the reasons tools like CMake have only recently started supporting C++ modules partially

USING MODULES

```
1  // examples/hello_m.cc
2  import <iostream>;
3
4  auto main() -> int
5  {
6      std::cout << "Hello, world!\n";
7  }
```

- If a module is available, not much special needs to be done to use it. `import` the module instead of `#include`ing a header. Use the exported classes, functions and variables from the module.
- In C++23, the standard library is supposed to be available as a module. But as of May 2024, neither GCC nor Clang supports standard library modules

```
$ clang++ -std=c++20 -stdlib=libc++ -fmodules hello_m.cc
$ ./a.out
$ g++ -std=c++20 -fmodules-ts -xc++-system-header iostream
$ g++ -std=c++20 -fmodules-ts hello_m.cc
$ ./a.out
$
```

- GCC requires that the header units needed are first generated in a separate explicit step.
- If `iostream` had been the name of a module, we would have written `import iostream;` instead of `import <iostream>;`

USING MODULES

Exercise 8.1:

Convert a few of the example programs you have seen during the course to use modules syntax instead. At the moment it means no more than replacing the `#include` lines with the corresponding `import` lines for the standard library headers. The point is to get used to the extra compilation options you need with modules at the moment. Use, for instance, the date time library functions like `feb.cc` and `advent.cc` from the day 4 examples.

CREATING A MODULE (EXAMPLE)

```
1  // saxpy.hh
2  #ifndef SAXPY_HH
3  #define SAXPY_HH
4  #include <algorithm>
5  #include <span>
6
7  template <class T>
8  concept Number = std::floating_point<T>
9                or std::integral<T>;
10 template <Number T>
11 auto saxpy(T a, std::span<const T> x,
12            std::span<const T> y,
13            std::span<T> z)
14 {
15     std::transform(x.begin(), x.end(),
16                    y.begin(), z.begin(),
17         [a](T X, T Y) {
18             return a * X + Y;
19         });
20 }
21 #endif
```

- A header file contains a function template `saxpy`, and a `concept` necessary to define that function
- A source file, `main.cc` which includes the header and uses the function

CREATING A MODULE (EXAMPLE)

```
1  // usesaxpy.cc
2  #include <iostream>
3  #include <array>
4  #include <vector>
5  #include <span>
6  #include "saxpy.hh"
7
8  auto main() -> int
9  {
10     using namespace std;
11     const array inp1 { 1., 2., 3., 4., 5. };
12     const array inp2 { 9., 8., 7., 6., 5. };
13     vector outp(inp1.size(), 0.);
14
15     saxpy(10., {inp1}, {inp2}, {outp});
16     for (auto x : outp) cout << x << "\n";
17     cout << "::::::::::::::::::::::\n";
18 }
```

- A header file contains a function template `saxpy`, and a `concept` necessary to define that function
- A source file, `main.cc` which includes the header and uses the function

CREATING A MODULE (EXAMPLE)

Make a module interface unit

```
1  // saxpy.hh -> saxpy.ixx
2  #ifndef SAXPY_HH
3  #define SAXPY_HH
4  #include <algorithm>
5  #include <span>
6
7  template <class T>
8  concept Number = std::floating_point<T>
9                or std::integral<T>;
10 template <Number T>
11 auto saxpy(T a, std::span<const T> x,
12            std::span<const T> y,
13            std::span<T> z)
14 {
15     std::transform(x.begin(), x.end(),
16                    y.begin(), z.begin(),
17         [a](T X, T Y) {
18             return a * X + Y;
19         });
20 }
21 #endif
```

CREATING A MODULE (EXAMPLE)

```
1  // saxpy.hh -> saxpy.ixx
2  #ifndef SAXPY_HH
3  #define SAXPY_HH
4  #include <algorithm>
5  #include <span>
6
7  template <class T>
8  concept Number = std::floating_point<T>
9                or std::integral<T>;
10 template <Number T>
11 auto saxpy(T a, std::span<const T> x,
12            std::span<const T> y,
13            std::span<T> z)
14 {
15     std::transform(x.begin(), x.end(),
16                    y.begin(), z.begin(),
17         [a](T X, T Y) {
18             return a * X + Y;
19         });
20 }
21 #endif
```

Make a module interface unit

- Include guards are no longer required, since importing a module does not transitively import things used inside the module

CREATING A MODULE (EXAMPLE)

```
1  // saxpy.hh -> saxpy.ixx
2
3
4  #include <algorithm>
5  #include <span>
6
7  template <class T>
8  concept Number = std::floating_point<T>
9                or std::integral<T>;
10 template <Number T>
11 auto saxpy(T a, std::span<const T> x,
12            std::span<const T> y,
13            std::span<T> z)
14 {
15     std::transform(x.begin(), x.end(),
16                    y.begin(), z.begin(),
17         [a](T X, T Y) {
18             return a * X + Y;
19         });
20 }
```

Make a module interface unit

- Include guards are no longer required, since importing a module does not transitively import things used inside the module

CREATING A MODULE (EXAMPLE)

```
1  // saxpy.hh -> saxpy.ixx
2  module;
3  #include <algorithm>
4  #include <span>
5  export module saxpy;
6
7  template <class T>
8  concept Number = std::floating_point<T>
9                or std::integral<T>;
10 template <Number T>
11 auto saxpy(T a, std::span<const T> x,
12            std::span<const T> y,
13            std::span<T> z)
14 {
15     std::transform(x.begin(), x.end(),
16                    y.begin(), z.begin(),
17         [a](T X, T Y) {
18             return a * X + Y;
19         });
20 }
```

Make a module interface unit

- Include guards are no longer required, since importing a module does not transitively import things used inside the module
- A module interface unit is a file which exports a module. The lines between the `module;` and `export module saxpy;` constitute the "global module fragment", where traditional `#include` statements can be used.

CREATING A MODULE (EXAMPLE)

```
1  // saxpy.hh -> saxpy.ixx
2
3  export module saxpy;
4  import <algorithm>;
5  import <span>;
6
7  template <class T>
8  concept Number = std::floating_point<T>
9                or std::integral<T>;
10 template <Number T>
11 auto saxpy(T a, std::span<const T> x,
12            std::span<const T> y,
13            std::span<T> z)
14 {
15     std::transform(x.begin(), x.end(),
16                    y.begin(), z.begin(),
17         [a](T X, T Y) {
18             return a * X + Y;
19         });
20 }
```

Make a module interface unit

- Include guards are no longer required, since importing a module does not transitively import things used inside the module
- A module interface unit is a file which exports a module. The lines between the `module;` and `export module saxpy;` constitute the "global module fragment", where traditional `#include` statements can be used.
- If you can get by with only `import`s, replace `#include` lines with corresponding `import` lines. Omit the `module;` line in this case.

CREATING A MODULE (EXAMPLE)

```
1  // saxpy.hh -> saxpy.ixx
2
3  export module saxpy;
4  import <algorithm>;
5  import <span>;
6
7  template <class T>
8  concept Number = std::floating_point<T>
9                or std::integral<T>;
10 export template <Number T>
11 auto saxpy(T a, std::span<const T> x,
12            std::span<const T> y,
13            std::span<T> z)
14 {
15     std::transform(x.begin(), x.end(),
16                    y.begin(), z.begin(),
17         [a](T X, T Y) {
18             return a * X + Y;
19         });
20 }
```

Make a module interface unit

- Include guards are no longer required, since importing a module does not transitively import things used inside the module
- A module interface unit is a file which exports a module. The lines between the `module;` and `export module saxpy;` constitute the "global module fragment", where traditional `#include` statements can be used.
- If you can get by with only `import`s, replace `#include` lines with corresponding `import` lines. Omit the `module;` line in this case.
- Explicitly export any definitions (classes, functions...) you want for users of the module. Anything not exported by a module is automatically private to the module

CREATING A MODULE (EXAMPLE)

Use your module

```
1   // usesaxpy.cc
2   #include <iostream>
3   #include <array>
4   #include <vector>
5   #include <span>
6   #include "saxpy.hh"
7
8   auto main() -> int
9   {
10      using namespace std;
11      const array inp1 { 1., 2., 3., 4., 5. };
12      const array inp2 { 9., 8., 7., 6., 5. };
13      vector outp(inp1.size(), 0.);
14
15      saxpy(10., {inp1}, {inp2}, {outp});
16      for (auto x : outp) cout << x << "\n";
17      cout << "::::::::::::::::::::::\n";
18  }
```

CREATING A MODULE (EXAMPLE)

```
1  // usesaxpy.cc
2  import <iostream>;
3  import <array>;
4  import <vector>;
5  import <span>;
6  #include "saxpy.hh"
7
8  auto main() -> int
9  {
10     using namespace std;
11     const array inp1 { 1., 2., 3., 4., 5. };
12     const array inp2 { 9., 8., 7., 6., 5. };
13     vector outp(inp1.size(), 0.);
14
15     saxpy(10., {inp1}, {inp2}, {outp});
16     for (auto x : outp) cout << x << "\n";
17     cout << ":::::::::::::::::::::\n";
18 }
```

Use your module

- Replace `#include` lines with corresponding `import` lines. Obs: `import` lines end with a semi-colon!

CREATING A MODULE (EXAMPLE)

```
1  // usesaxpy.cc
2  import <iostream>;
3  import <array>;
4  import <vector>;
5  import <span>;
6  import saxpy;
7
8  auto main() -> int
9  {
10     using namespace std;
11     const array inp1 { 1., 2., 3., 4., 5. };
12     const array inp2 { 9., 8., 7., 6., 5. };
13     vector outp(inp1.size(), 0.);
14
15     saxpy(10., {inp1}, {inp2}, {outp});
16     for (auto x : outp) cout << x << "\n";
17     cout << ":::::::::::::::::::::\n";
18 }
```

Use your module

- Replace `#include` lines with corresponding `import` lines. Obs: `import` lines end with a semi-colon!
- When importing actual modules, rather than header units, use the `module name` without angle brackets or quotes

CREATING A MODULE (EXAMPLE)

```
1  // usesaxpy.cc
2  import <iostream>;
3  import <array>;
4  import <vector>;
5  import <span>;
6  import saxpy;
7
8  auto main() -> int
9  {
10     using namespace std;
11     const array inp1 { 1., 2., 3., 4., 5. };
12     const array inp2 { 9., 8., 7., 6., 5. };
13     vector outp(inp1.size(), 0.);
14
15     saxpy(10., {inp1}, {inp2}, {outp});
16     for (auto x : outp) cout << x << "\n";
17     cout << ":::::::::::::::::::::\n";
18 }
```

Use your module

- Replace `#include` lines with corresponding `import` lines. Obs: `import` lines end with a semi-colon!
- When importing actual modules, rather than header units, use the `module name` without angle brackets or quotes
- Importing `saxpy` here, only grants us access to the explicitly exported function `saxpy`. Not other functions, classes, concepts etc. defined in the module `saxpy`, not any other module imported in the module interface unit.

COMPILATION OF PROJECTS WITH MODULES

- Different compilers require different (sets of) options
- GCC:
  - Auto-detects if a file is a module interface unit (exports a module), and generates the CMI as well as an object file together.
  - No special file extension required for module interface units(Just `.cc`, `.cpp`, ... like regular source files).
  - Requires that standard library header units needed by the project are explicitly generated
  - Does not recognise module interface file extensions used by other compilers (`.ixx`, `.ccm` etc.)
  - Still rather crashy in May 2024, especially if multiple standard library headers are in use.
- Clang:
  - Provides standard library header units.
  - Comparatively stable for module based code.
  - Lots of command line options required
  - Different options to translate module interfaces depending on file extensions!
    - `.ccm` or `.cppm`: `--precompile`
    - `.ixx`: `--precompile -xc++-module`
    - `.cc` or `.cpp`: `-Xclang -emit-module-interface`
  - Separate generation of object file required
  - Module partitions not implemented

C++ MODULES AND CMAKE

```
cmake_minimum_required(VERSION 3.28)

set(CMAKE_CXX_STANDARD 23)
set(CMAKE_CXX_EXTENSIONS OFF)
set(CMAKE_EXPORT_COMPILE_COMMANDS ON)
project(use_saxpy-example)

add_executable(use_saxpy)
target_sources(use_saxpy
    PUBLIC
    usesaxpy.cc
)
target_sources(use_saxpy
  PUBLIC
    FILE_SET saxpy_module
    TYPE CXX_MODULES
    FILES saxpy.ixx
)
```

- `CMake` supports C++ modules since the version 3.28
- The `Ninja` generator is required
- Currently does not support header units, so that we still have to `#include` them
- Massive simplification of the build process!

```
mkdir -p build && cd build
cmake -DCMAKE_GENERATOR=Ninja ..
ninja
```

Exercise 8.2:

Versions of the `saxpy` program written using header files and then modules can be found in the `examples/saxpy/`. Familiarise yourself with the process of building applications with modules. Experiment by writing a new inline function in the module interface file without exporting it. Try to call that function from `main`. Check again after exporting it in the module.

Exercise 8.3:

As a more complicated example, we have in `examples/2_any` the second version of our container with polymorphic geometrical objects. The header and source files for each class `Point`, `Circle` etc have been rewritten for modules. Compare the two versions, build them, run them.

A few useful non-standard libraries

SOME USEFUL NON-STANDARD LIBRARIES

- Command line processing
- `CTRE` Compile time regular expressions
- `xtensor` (the entire `xtensor-stack`) : C++ SIMD wrappers, numpy style multi-dimensional array operations...

LYRA: COMMAND LINE PROCESSING

```cpp
auto cli = lyra::help(showhelp)
    | lyra::opt(N, "N_samples")["-N"]["--number-of-samples"]
        ("The number of samples you want to generate")
    | lyra::opt(mean, "mu")["-m"]["--mean"]("The mean of the distribution")
    | lyra::opt(stdv, "sigma")["-s"]["--standard-deviation"]("Standard deviation");

auto cli_good = cli.parse({ argc, argv });

if (not cli_good) {
    std::cerr << "Error in command line: " << cli_good.errorMessage() << "\n";
    return 1;
}

if (showhelp or argc == 1) {
    std::cout << cli;
    return 0;
}

```

MORE COMMAND LINE PROCESSING: CXXOPTS

```cpp
auto main(int argc, char* argv[]) -> int {
  cxxopts::Options options("myprog", "Toy prog.");
  options.add_options()
    ("d,decrypt", "Decrypt a file",
       cxxopts::value<bool>()
       ->default_value("false"))
    ("filename", "The input file",
       cxxopts::value<std::string>())
    ("o,output", "The resulting output file",
       cxxopts::value<std::string>())
    ("h,help", "Print usage");
  options.parse_positional({"filename"});
  auto args = options.parse(argc, argv);
  if (args.count("help") or
      args.count("filename") == 0) {
    std::print("{}\n", options.help());
    exit(0);
  }
  bool decrpt = args["decrypt"].as<bool>();
  std::string ifile, pass;
  if (args.count("filename"))
    ifile = args["filename"].as<std::string>();
```

- Lightweight option parsing library
- Supports GNU style options syntax
- Long, short arguments, combined short arguments...
- Header only, but with the necessary CMake files

Exercise 8.4:

The file `examples/eratosthenes_sieve.cc` contains a small program to find out all the prime numbers up to a given maximum using the Eratosthene Sieve method. It works, but asks the user for the limit, and whether or not to print the results. Modify the program using a command line parser, so that the limit, and other options can be provided using program options.

CTRE: COMPILE TIME REGULAR EXPRESSIONS

```
1  constexpr ctll::fixed_string re{ R"xpr(^(https:|http:|www\.)\S*)xpr" };
2
3  auto urls_in_input = args | sv::drop(1)
4                            | sv::transform([=](auto inp) { return str(inp); })
5                            | sv::filter([re](auto inp) { return ctre::search<re>(inp); });
```

- Regular expressions parsed at compile time.
- Smaller binaries than `std::regex`

Compiling an up-to-date compiler

BUILDING YOUR OWN GCC AND CLANG

GCC

- Download from gcc.gnu.org
- Unpack and build as shown
- Install where you wish
- Set PATH, `LD_LIBRARY_PATH` and `LD_RUN_PATH` appropriately

```
tar -xvJf gcc-14.1.0.tar.xz
cd gcc-14.1.0
./contribs/download-prerequisites
mkdir build && cd build
../configure --prefix=/home/you/local/gcc/14.1.0 \
   --enable-optimized --disable-multilib \
   --enable-linux-futex \
   --enable-languages=c,c++,fortran,lto

make -j 8
make install
```

BUILDING YOUR OWN GCC AND CLANG

Clang

- Download from https://github.com/llvm/llvm-project/releases/tag/llvmorg-18.1.5
- Download the llvm-project-18.1.5.src.tar.xz file
- Unpack and build as shown
- Install where you wish
- Set PATH, `LD_LIBRARY_PATH` and `LD_RUN_PATH` appropriately

```
tar -xvJf llvm-project-18.1.5.src.tar.xz
cd llvm-project-18.1.5.src
mkdir build && cd build
cmake -DCMAKE_BUILD_TYPE=Release \
    -DCMAKE_INSTALL_PREFIX=\
    /home/you/local/llvm/18.1.5 \
    -DLLVM_ENABLE_PROJECTS=\
    "clang;lld;clang-tools-extra" \
    -DLLVM_ENABLE_RUNTIMES=\
    "libcxx;libcxxabi;libunwind;pstl"\
    -DCLANG_DEFAULT_CXX_STDLIB=libc++ \
    -DLLVM_INSTALL_UTILS=ON \
    -DLLVM_TARGETS_TO_BUILD=host ../llvm

make -j 8
make install
```

Exercise 8.5: Mono-alphabetic substitution cipher

Mono-alphabetic substitution cipher is an ancient encryption method. In this method, a map is constructed of the letters of the alphabet with a scrambled version of the alphabet:

Original	a b c d e f g h i j k l m n o p q r s t u v w x y z
Cipher	d m t w g f x v j z r n e l k u o q b s c p a i h y

For instance, with the above substitution table, the word "ancient" will be enciphered as "dltjgls". For

convenience, the scrambled order can be generated by taking a passphrase, deleting repetitions, and giving the remaining characters of the passphrase the first positions in the substitution table. The characters not used after this process are then given the remaining positions in the substitution table in their own order. For instance, if the passphrase is "apples are tasty", we would start by removing spaces and repetitions to get "aplesrty". Our scrambled alphabet will then be, "aplesrtybcdfghijkmnoquvwxz". (continued on next slide ...)

Exercise 8.5: (continued ...)

Write a program which receives from the command line options specifying

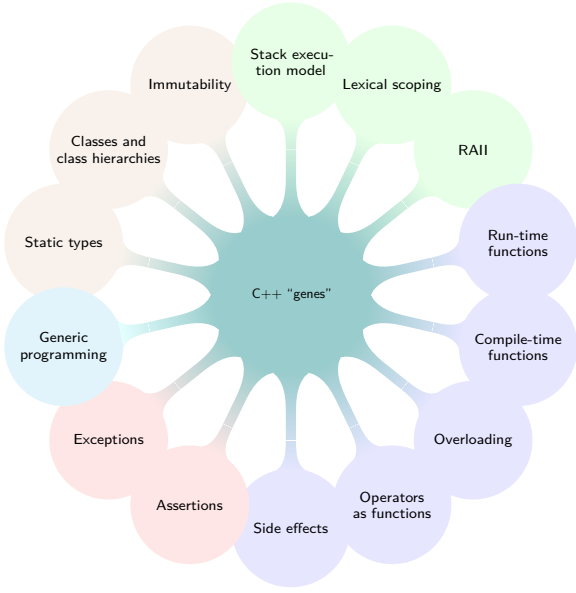
- the name of an input text file
- another filename to save the results
- a flag indicating if we want to enciphers/deciphers the text

It should then proceed to ask the user to enter a passphrase, produces a scrambled alphabet (using both capital and small letters) with the passphrase according to the above scheme, and then apply the substitution table to the input text to produce the output text. Most of the code is in `exercises/subcipher.cc`, with some important parts left for you to fill in.

For this exercise, you can leave punctuation and white space untouched.
Note: This is not a particularly good cipher!

Chapter 9

Closing remarks


C++ “genes”
Stack execu-
tion model
Lexical scoping
RAII
Run-time
functions
Compile-time
functions
Overloading
Operators
as functions
Side effects
Assertions
Exceptions
Generic
programming
Static types
Classes and
class hierarchies
Immutability

CLOSING REMARKS

- Most examples were simply demo code to show you how it works

CLOSING REMARKS

- Most examples were simply demo code to show you how it works
- To internalise the ideas, you have to solve those or similar problems yourself

CLOSING REMARKS

- Most examples were simply demo code to show you how it works
- To internalise the ideas, you have to solve those or similar problems yourself
- Information summarised for you, so that you hear it once, and then hopefully when you need it, you will remember having heard about a feature, and then look it up

CLOSING REMARKS

- Most examples were simply demo code to show you how it works
- To internalise the ideas, you have to solve those or similar problems yourself
- Information summarised for you, so that you hear it once, and then hopefully when you need it, you will remember having heard about a feature, and then look it up
- Rapidly evolving language

CLOSING REMARKS

- Most examples were simply demo code to show you how it works
- To internalise the ideas, you have to solve those or similar problems yourself
- Information summarised for you, so that you hear it once, and then hopefully when you need it, you will remember having heard about a feature, and then look it up
- Rapidly evolving language
- en.cppreference.com

CLOSING REMARKS

- Most examples were simply demo code to show you how it works
- To internalise the ideas, you have to solve those or similar problems yourself
- Information summarised for you, so that you hear it once, and then hopefully when you need it, you will remember having heard about a feature, and then look it up
- Rapidly evolving language
- en.cppreference.com
- isocpp.org

CLOSING REMARKS

- Most examples were simply demo code to show you how it works
- To internalise the ideas, you have to solve those or similar problems yourself
- Information summarised for you, so that you hear it once, and then hopefully when you need it, you will remember having heard about a feature, and then look it up
- Rapidly evolving language
- en.cppreference.com
- isocpp.org
- YouTube channel: Jason Turner's C++ weekly

CLOSING REMARKS

- Most examples were simply demo code to show you how it works
- To internalise the ideas, you have to solve those or similar problems yourself
- Information summarised for you, so that you hear it once, and then hopefully when you need it, you will remember having heard about a feature, and then look it up
- Rapidly evolving language
- en.cppreference.com
- isocpp.org
- YouTube channel: Jason Turner's C++ weekly
- YouTube channel: CppCon conference talks

A GAME TO BREAK THE SUBSTITUTION CIPHER

Exercise 9.1:

The one-to-one substitution we created in an earlier exercise, called the monoalphabetic substition cipher, is easy to break. The key insight is this: if we decide to write E with a different symbol, say, α, it does not alter the fact that it is the most frequently used character in English. Whichever symbol we use for it will end up appearing very frequently in the text. The "cipher" text we created is simply our original text with written with different symbols, which happen to be chosen from the same alphabet. The frequency of characters, consecutive character pairs etc. will be represented approximately by the corresponding substituted letters.

We can make a little game out of breaking this cipher like this: We make a program to display the cipher text in capital letters. Make a little interpreter with a text menu, where it is possible to enter a substition, undo, display character and word frequencies. After each substition, we display the characters already substited in small case. A preliminary working implemention exists in the exercises folder. See how it works. Improve it!