



PROGRAMMING IN C++

Jülich Supercomputing Centre

May 9, 2022 | Sandipan Mohanty | Forschungszentrum Jülich, Germany

Day 1

ELEGANT AND EFFICIENT ABSTRACTIONS

Software development challenges

- Handle increasingly more complex problems
- Rich set of concepts with which to imagine what can be done
- Collaborative development
- Long term maintainability
- Do all of the above, and yet deliver code that runs as fast as possible

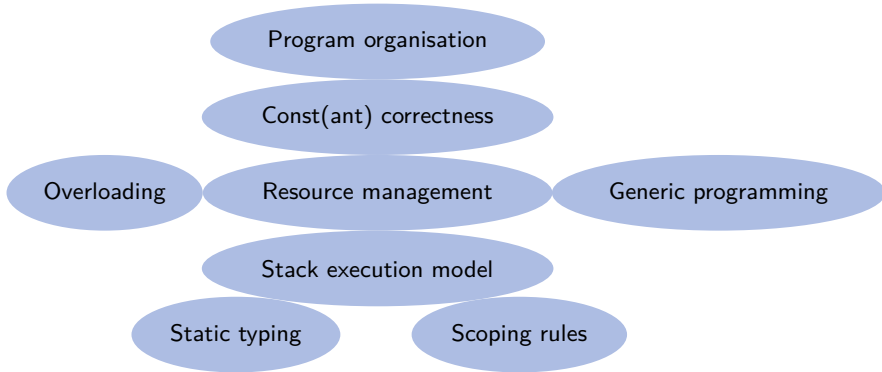
C++ provides ...

- Direct mapping of built in operations and types to hardware
- Powerful and efficient abstraction mechanisms
- Multiparadigm programming: Procedural, object oriented, generic and functional programming

C++

- General purpose: no specialization to specific usage areas
- No over simplification that precludes a direct expert level use of hardware resources
- Leave no room for a lower level language
- What you don't use, you don't pay for
- Express
 - ideas directly in code
 - simple ideas with simple code
 - independent ideas independently in code
 - relationships among ideas directly in code
- Combine ideas expressed in code freely

C++ “GENES”



LEARNING C++

- It takes time.

LEARNING C++

- It takes time.
- Strong foundations in the building blocks of the language

LEARNING C++

- It takes time.
- Strong foundations in the building blocks of the language
- Self-study over a much longer period

LEARNING C++

- It takes time.
- Strong foundations in the building blocks of the language
- Self-study over a much longer period
- Collaborative projects with good senior programmers

LEARNING C++

- It takes time.
- Strong foundations in the building blocks of the language
- Self-study over a much longer period
- Collaborative projects with good senior programmers
- Curiosity about an evolving language

LEARNING C++

- It takes time.
- Strong foundations in the building blocks of the language
- Self-study over a much longer period
- Collaborative projects with good senior programmers
- Curiosity about an evolving language
- Two kinds of challenges: How can I do this ? What can I do with this ?

LEARNING C++

- It takes time.
- Strong foundations in the building blocks of the language
- Self-study over a much longer period
- Collaborative projects with good senior programmers
- Curiosity about an evolving language
- Two kinds of challenges: How can I do this ? What can I do with this ?
- Goals for this course: emphasis on fundamentals, a tour of what exists, methods to facilitate continued learning

C++ IN MAY 2022

- Current standard with stable implementations: C++17.
- Latest standard approved by the ISO C++ committee: C++20
- All language features and almost all library features of C++17 are available in the two major open source compilers: GCC and Clang.
- C++20 features are currently being implemented, but some important features such as `concepts`, `modules` and `ranges` are available for testing
- Microsoft's MSVC compiler is currently the compiler with more implemented C++20 features than any other compiler

Summary of compiler support for different language library features for different C++ standards can be looked up at cpreference.com

```
1 xarray<double> rt
2     = load_csv<double>(fin, '\t');
3
4 rt -= mean(rt, 0);
5
6 xarray<double> cross =
7     linalg::dot(transpose(rt), rt);
8
9 auto [lambda, v] = linalg::eig(cross);
```

- Easier, cleaner and more efficient language
- Elegant syntax, without compromising speed or safety

C++ IN MAY 2022

- Current standard with stable implementations: C++17.
- Latest standard approved by the ISO C++ committee: C++20
- All language features and almost all library features of C++17 are available in the two major open source compilers: GCC and Clang.
- C++20 features are currently being implemented, but some important features such as `concepts`, `modules` and `ranges` are available for testing
- Microsoft's MSVC compiler is currently the compiler with more implemented C++20 features than any other compiler

Summary of compiler support for different language library features for different C++ standards can be looked up at cpreference.com

```
1 using namespace std::chrono;
2 using Date = year_month_day;
3
4 year Y { asked_year.value_or(current_year()) };
5
6 Date s4 { Y / December / Sunday[4] };
7 Date s3 { Y / December / Sunday[3] };
8 Date xmas { Y / December / 25d };
9 Date lastadv { s4 >= xmas ? s3 : s4 };
```

- Easier, cleaner and more efficient language
- Elegant syntax, without compromising speed or safety

COMPILER SUPPORT FOR C++ STANDARDS

- Check the latest status of compiler support for C++11, C++14, C++17, C++20 etc by following this link.
- Open source GCC and Clang compilers have held the edge since 2011 in providing access to the latest language features. Both have C++17 language features, and most of the library features implemented. C++20 support is still patchy, although steadily improving. It's usually better to use as new a version as possible
- GCC 11.x uses C++17 as its default.
- Clang makes the default standard a CMake configuration option, but is very often built with C++98 as the default. In that case, we would need to explicitly specify the standard we want to use with a command line option, such as `-std=c++17` or `-std=c++20`. The version installed for the course defaults to C++17 like GCC.

COURSE CONTENT

- Language fundamentals

which means...

- Basic structure of a program
- Variables
- Mutability controls
- Statements, blocks
- Branches, loops
- Functions and lambda expressions
- Scope
- Error handling

COURSE CONTENT

- Language fundamentals
- Small applications using C++ standard library facilities

which means...

- Strings
- Containers and algorithms
- Input/Output

COURSE CONTENT

- Language fundamentals
- Small applications using C++ standard library facilities
- C++ classes in detail

which means...

- Detailed syntax explanation
- RAII
- Operator overloading
- Invariants
- Inheritance and virtual dispatch
- SOLID principles

COURSE CONTENT

- Language fundamentals
- Small applications using C++ standard library facilities
- C++ classes in detail
- C++ templates

which means...

- Function, class and variable templates
- Constrained templates using
concepts
- Variadic templates

COURSE CONTENT

- Language fundamentals
- Small applications using C++ standard library facilities
- C++ classes in detail
- C++ templates
- Standard template library in detail

which means...

- Iterator based design of containers
- Containers and algorithms
- Ranges
- Date and time
- Random numbers
- Smart pointers
- Text formatting

COURSE CONTENT

- Language fundamentals
- Small applications using C++ standard library facilities
- C++ classes in detail
- C++ templates
- Standard template library in detail
- Some useful open source C++ libraries

which means...

- Open source libraries enabling the use of some C++20 features before they are implemented in compilers
- Better regular expressions

COURSE CONTENT

- Language fundamentals
- Small applications using C++ standard library facilities
- C++ classes in detail
- C++ templates
- Standard template library in detail
- Some useful open source C++ libraries
- Program organisation: expected changes

which means...

- Modules

GETTING STARTED

Set up course room access...

- Recommendation: use two browser windows, one for the BBB lecture room, one for the JupyterLab session.
In case you have direct ssh access to JUSUF, you can use a terminal for most of the tasks.

GETTING STARTED

Set up course room access...

- Recommendation: use two browser windows, one for the BBB lecture room, one for the JupyterLab session. In case you have direct ssh access to JUSUF, you can use a terminal for most of the tasks.
- Follow the instructions in the `everyday.pdf` file to set up your working area (it's just one command to run!)

GETTING STARTED

Set up course room access...

- Recommendation: use two browser windows, one for the BBB lecture room, one for the JupyterLab session. In case you have direct ssh access to JUSUF, you can use a terminal for most of the tasks.
- Follow the instructions in the `everyday.pdf` file to set up your working area (it's just one command to run!)
- Download material from your private working area through JupyterLab. It contains a copy of the slides and all the exercises for the day.

GETTING STARTED

Set up course room access...

- Recommendation: use two browser windows, one for the BBB lecture room, one for the JupyterLab session. In case you have direct ssh access to JUSUF, you can use a terminal for most of the tasks.
- Follow the instructions in the `everyday.pdf` file to set up your working area (it's just one command to run!)
- Download material from your private working area through JupyterLab. It contains a copy of the slides and all the exercises for the day.
- Draw attention if you have difficulties

Fundamentals

A COMPILED LANGUAGE

```
1 // Hello World!
2 #include <iostream>
3 auto main() -> int
4 {
5     std::cout<<"Hello, world!\n";
6 }
```

```
g++ -std=c++20 hello.cc
./a.out
```

- Human readable source code is translated to the machine language by the **compiler**
- Strictly enforces rules of the language
- Rules enable accurate expression of intent
- Compiler performs analysis of syntax tree, optimisation passes, automatic discovery of shortcuts
- Same observable effects as the source code, but not necessarily doing everything exactly as you say.

Compiler Explorer — Mozilla Firefox

GitHub · jupyter-notebook · x86_64 · x86_64 · Project training2113 · Compiler Explorer · +

COMPILER EXPLORER

Add... More

Watch C++ Weekly to learn new C++ features

Sponsors intel PC-lint

Share Other Policies

C++ source #1

```
1 auto sum_upto(unsigned num) -> unsigned
2 {
3     auto ans{0U};
4     for (auto i = 0U; i < num; ++i) {
5         ans += i;
6     }
7     return ans;
8 }
9
```

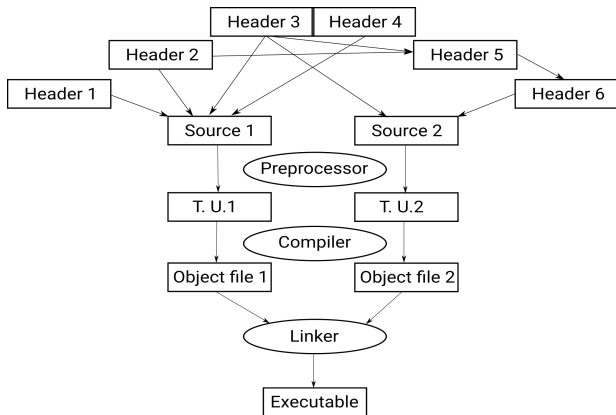
x86-64 clang (trunk) (Editor #1, Compiler #1) C++

x86-64 clang (trunk) -O3

```
1 sum_upto(unsigned int):
2     test    edi, edi
3     je      .LBB0_1
4     lea     eax, [rdi - 1]
5     lea     ecx, [rdi - 2]
6     imul    rcx, rcx
7     shr     rcx
8     lea     eax, [rcx + rdi]
9     add     eax, -1
10    ret
11 .LBB0_1:
12    xor     eax, eax
13    ret
```

Output (0/0) x86-64 clang (trunk) - cached (11714B) ~238 lines filtered

THE COMPILATION PROCESS



COMMAND LINE ARGUMENTS

- In the `argc, argv` form of `main`, the command line is broken into a sequence of character strings and passed as the array `argv`
- The name of the program is the first string in this list, `argv[0]`. Therefore `argc` is never 0.

```
1 // examples/hello_xyz.cc
2 #include <iostream>
3 auto main(int argc, char *argv[]) -> int
4 {
5     std::cout<<"Hello, ";
6     if (argc > 1)
7         std::cout <<argv[1]<< "!\n";
8     else
9         std::cout<<"world!\n";
10 }
11 g++ main.cpp && ./a.out rain clouds
```

Exercise 1.1:

Open <http://coliru.stacked-crooked.com/>, copy and paste the above program and run it with some command line parameters! Alternatively, use the compiler you set up yourself. Alternatively, save it in a text file on your computer. Upload it to our JupyterHub. Use the terminal in your JupyterLab to compile and run.

THE MAIN() FUNCTION

- All C++ programs must contain a unique `main()` function
- All executable code is contained either in `main()` , or in functions invoked directly or indirectly from `main()`
- The return value for `main()` is canonically an integer. A value 0 means successful completion, any other value means errors. UNIX based operating systems make use of this.
- In a C++ `main` function, the `return 0;` at the end of `main()` can be omitted.

FUNCTION CALL TREE

```
auto main() -> int
{
    auto N = 10;
    if (f(N) < g(N)) {
        h1(N);
    } else {
        h2(N);
    }
}

auto f(int i) () -> int
{
    return (i * i) % 12;
}

auto g(int i) () -> int
{
    return i % 12;
}

auto h1(int i) () -> int
{
    return h11(i);
}

auto h2(int i) () -> int
{
    return h21(i);
}

auto h11(int i) () -> int
{
    return i * i;
}

auto h21(int i) () -> int
{
    return -i;
}
```

- Every function contains control flow regulating keywords or expressions.
- Some of the expressions may be function calls which will cause instructions in that other function to be executed
- The execution tree starts at the `main`

CODE LEGIBILITY

- Human brains are not made for searching { and } in dense text

```
1  double foo(double x, int i)
2  {
3  double y=1;
4  if (i>0) {
5  for (int j=0;j<i;++j) {
6  y *= x;
7  }
8  } else if (i<0) {
9  for (int j=0;j>i;--j) {
10 y /= x;
11 }}
12 return y;
13 }
```

STYLE

```
1  double foo(double x, int i)
2  {
3      double y = 1;
4      if (i > 0) {
5          for (int j = 0; j < i; ++j) {
6              y *= x;
7          }
8      } else if (i < 0) {
9          for (int j = 0; j > i; --j) {
10             y /= x;
11         }
12     }
13     return y;
14 }
```

- Indenting code clarifies the logic
- Misplaced brackets, braces etc. are easier to detect
- 4-5 levels of nesting is sometimes unavoidable
- Recommendation: indent with 2-4 spaces and be consistent!

STYLE

```
1 double foo(double x, int i)
2 {
3     double y = 1;
4     if (i > 0) {
5         for (int j = 0; j < i; ++j) {
6             y *= x;
7         }
8     } else if (i < 0) {
9         for (int j = 0; j > i; --j) {
10             y /= x;
11         }
12     }
13     return y;
14 }
```

- Use a consistent convention for braces ({ and }).
- Use a tool like `clang-format` to clean up formatting before committing code to your version control system
- The utility `cf` included with your course material (Usage: `cf sourcefile.cc`) formats code using `clang-format` with the WebKit style.
- Set up your editor to indent automatically! In Qt creator, set up auto indentation with “clang format” by going to Tools → Options → Beautifier.

- These are for the human reader (most often, yourself!). Be nice to yourself, and write code that is easy on the eye!

READ C++

```
1 // examples/hello_qa.cc
2 #include <string>
3 #include <iostream>
4
5 auto main() -> int
6 {
7     std::string name;
8     std::cout << "What's your name ? ";
9     std::cin >> name;
10    std::cout << "Hello, " << name << "\n";
11 }
```

Exercise 1.2:

What does this code do ? What if you answer with a name with multiple parts ? Replace the line where we read in to the variable `name` with `getline(std::cin, name);`, and repeat. If you run the program from your IDE, you may have to adjust your “run” settings (Qt creator: Projects → Build and run → Run : “run in terminal”).

TYPES, VARIABLES AND DECLARATIONS

```
1  auto force(double m1, double m2, double r12)
2      -> double
3  {
4      const auto G{ 6.67408e-11 };
5      return G * m1 * m2 / (r12 * r12);
6  }
```

```
1  // Old style, but still fine
2  unsigned long x = 0;
3  string name{"Maple"};
4  vector<int> v{1, 2, 3, 4, 5};
5  tuple<int, int, string> R{0, 0, "A"};
6  complex<double> z{0.5, 0.6};
```

- A "type" defines the possible values and operations for an object
- An "object" is some memory holding a value of a certain type
- A "value" is bits interpreted according to a certain type
- A "variable" is a named object
- A "declaration" is a statement introducing a name into the program
- Statically typed: types of all created variables are known at compilation time.
A variable can not change its type.

BUILT IN AND USER DEFINED TYPES

Built in types

- Types like `char`, `bool`, `int`, `float`, `double` are known as fundamental types
- Fundamental types are implicitly inter-converted when required
- Arithmetic operations `+`, `-`, `*`, `/`, as well as comparisons `<`, `>`, `<=`, `>=`, `==`, `!=` are defined for the fundamental types, and mapped directly to low level instructions
- Like in many languages, `=` is assignment where as `==` is equality comparison
- Note how variables are "initialized" to sensible values when they are declared

Class types

- Additional types can be introduced to a program using keywords `class`, `struct`, `enum` and `enum class`, and much less commonly `union`
- Behaviour of a user defined type is programmable

INITIALIZATION

- Both `int i = 23` and `int i{ 23 }` are valid initializations
- The newer curly bracket form should be preferred, as it does not allow "narrowing" initialisation:
`int i{ 2.3 }; // Compiler error`
- The curly bracket form can also be used to initialise C++ collections:

```
1  std::list<double> masses{0.511, 938.28, 939.57};
2  std::vector<int> scores{667, 1}; // Vector of two elements, 667 and 1
3  std::vector<int> lows(250, 0); // vector of 250 zeros
```

- In rare cases, initialization requires `()` for disambiguation
- Since C++17, standard container types use a new language feature called "class template argument deduction" (CTAD) to infer the element type from the initialiser expression
- Variables can be declared anywhere in the program. Avoid declaring a variable until you have something meaningful to store in it

INITIALIZATION

- Both `int i = 23` and `int i{ 23 }` are valid initializations
- The newer curly bracket form should be preferred, as it does not allow "narrowing" initialisation:
`int i{ 2.3 }; // Compiler error`
- The curly bracket form can also be used to initialise C++ collections:

```
1  std::list masses{0.511, 938.28, 939.57};  
2  std::vector scores{667,1}; // Vector of two elements, 667 and 1  
3  std::vector lows(250, 0) ; // vector of 250 zeros
```

- In rare cases, initialization requires `()` for disambiguation
- Since C++17, standard container types use a new language feature called "class template argument deduction" (CTAD) to infer the element type from the initialiser expression
- Variables can be declared anywhere in the program. Avoid declaring a variable until you have something meaningful to store in it

THE UNIFORM INITIALISATION SYNTAX

```
1  int I{20};  
2  // define integer I and set it to 20  
3  string nat{"Germany"};  
4  // define and initialise a string  
5  double a[4]{1.,22.1,19.3,14.1};  
6  // arrays have the same syntax  
7  tuple<int,int,double> x{0,0,3.14};  
8  // So do tuples  
9  list<string> L{"abc","def","ghi"};  
10 // and lists, vectors etc.  
11 double m=0.5; // Initialising with '='  
12 // is ok for simple variables, but ...  
13 int k=5.3; // Allowed, although the  
14 // integer k stores 5, and not 5.3  
15 int j{5.3}; // Helpful compiler error.  
16 int i{}; // i=0  
17 vector<int> u{4,0}; // u={4, 0}  
18 vector<int> v(4,0); // v={0, 0, 0, 0}
```

- Variables can be initialised at the point of declaration with a suitable value enclosed in `{ }`
- Historical note: Pre-C++11, only the `=` and `()` notations (also demonstrated in the left panel) were available. Initialising non trivial collections was not allowed.

- **Recommendation:** Use `{ }` initialisation syntax as your default. A few exceptional situations requiring the `()` or `=` syntax can be seen in the left panel.

THE KEYWORDS AUTO AND DECLTYPE

```
1 auto sqr(int x) -> int { return x * x; }
2 auto main() -> int {
3     char oldchoice{'u'}, choice{'y'};
4     size_t i = 20'000'000;
5     //group digits for readability!
6     double electron_mass{ 0.511 };
7     int mes[6]{33, 22, 34, 0, 89, 3};
8     bool flag{ true };
9     decltype(i) j{ 9 };
10    auto positron_mass = electron_mass;
11    auto f = sqr; // Without "auto", f can
12    // be declared like this:
13    //int (*f)(int) = &sqr;
14    std::cout << f(j) << '\n';
15    auto muon_mass{ 105.6583745 };
16    // If somefunc() returns
17    // tuple<string, int, double>
18    auto [name, nspinstates, lifetime]
19        = somefunc(serno);
20 }
```

- If a variable is initialised when it is declared, in such a way that its type is unambiguous, the keyword `auto` can be used to declare its type
- The keyword `decltype` can be used to say "same type as that one"
- Since C++17, new names can be bound to components of a tuple, as shown

USING LITERALS WITH PRECISE TYPES

```
1 auto age = 7;  
2 auto pi = 3.141592653589793;  
3 auto energy = 0;  
4 auto city = "Barcelona";
```

- What are the types of the variables declared here ?

USING LITERALS WITH PRECISE TYPES

```
1 auto age = 7;  
2 auto pi = 3.141592653589793;  
3 auto energy = 0;  
4 auto city = "Barcelona";
```

- What are the types of the variables declared here ?
- How can we make sure that `age` is unsigned, `pi` and `energy` are double precision, and `city` is a `string` ?

USING LITERALS WITH PRECISE TYPES

```
1 auto age = 7u;
2 auto pi = 3.141592653589793;
3 auto energy = 0.;
4 using namespace std::string_literals;
5 auto city = "Barcelona"s;
6 auto bigpositive = 0UL;
7 auto fort_real = 0.0F;
8 // With proper user defined functions
9 auto T1 = 300_Kelvin;
10 auto T2 = 100_Celcius;
11 auto dist = 4.5_KM + 6.3_Miles;
```

- What are the types of the variables declared here ?
- How can we make sure that `age` is unsigned, `pi` and `energy` are double precision, and `city` is a `string` ?
- Use the proper literal types.

USING LITERALS WITH PRECISE TYPES

```
1 auto age = 7u;
2 auto pi = 3.141592653589793;
3 auto energy = 0.;
4 using namespace std::string_literals;
5 auto city = "Barcelona"s;
6 auto bigpositive = 0UL;
7 auto fort_real = 0.0F;
8 // With proper user defined functions
9 auto T1 = 300_Kelvin;
10 auto T2 = 100_Celcius;
11 auto dist = 4.5_KM + 6.3_Miles;
```

- What are the types of the variables declared here ?
- How can we make sure that `age` is unsigned, `pi` and `energy` are double precision, and `city` is a `string` ?
- Use the proper literal types.
- C++ allows you to make literals for user defined types

C++ STANDARD LIBRARY STRINGS

```
1  #include <string>
2  std::string fullname;
3  std::string name{"Albert"};
4  using namespace std::string_literals;
5  auto surname{"Einstein"s};
6  //Concatenation and assignment
7  fullname = name + " " + surname;
8
9  //Comparison
10 if (name == "Godzilla") run();
11
12 std::cout << fullname << '\n';
13
14 for (size_t i = 0; i < fullname.size(); ++i) {
15     if (fullname[i] > 'j') blah += fullname[i];
16 }
17 std::cout << "Substring after last z is "
18           << name.substr(
19               name.find_last_of('z'));
```

- String of characters
- Knows its size (see example)
- Allocates and frees memory as needed
- Simple syntax for assignment (=), concatenation(+), comparison (<, ==, >)
- The namespace std::string_literals defines the necessary functions to write literals which are interpreted as std::string instead of raw character arrays

CONVERTING TO AND FROM STRINGS

```
1  std::cout << "integer : " << std::to_string(i) << '\n';  
2  tot += std::stod(line); // String-to-double
```

- The standard library `string` class provides functions to inter-convert with variables of type `int`, `double`

Exercise 1.3:

Test example usage of string ↔ number conversions in `examples/to_string.cc` and `examples/stoX.cc`

STD::STRING_VIEW

```
1 std::string_view viewse{"Norrsken"};
2 using namespace std::string_view_literals;
3 auto viewen{"Northern lights"sv};
4
5 auto proc(std::string_view inp) -> bool
6 {
7     if (inp.ends_with("et")) {
8         if (inp.substr(0UL, 3UL) == blah)
9             // ...
10     }
11 }
```

- "View" over an existing array of characters, either in a string or in a character literal or a plain character array
- Does not own any data, does not try to do any memory management
- Provides an interface similar to `std::string`
- Can be compared like (and with) `std::string` objects
- Can not grow (no memory management!), but can shrink
- Cheap to pass to functions by value
- Has its own literal definitions in the namespace `std::string_view_literals`

RAW STRING LITERALS

```
// Instead of ...  
string message{"The tag \"\\maketitle\" is unexpected here."};  
// You can write ...  
string message{R"(The tag \"maketitle\" is unexpected here.)"};
```

- Can contain line breaks, `'\'` characters without escaping them, like the tripple quote strings in Python
- Very useful with regular expressions
- Starts with `R" (` and ends with `) "`
- More general form `R"delim(text)delim"`

Exercise 1.4:

The file `examples/rawstring.cc` illustrates raw strings in use. The file `examples/raw1.cc` has a small program printing a message about using the continuation character `'\'` at the end of the line to continue the input. Modify using raw string literals.

BLOCKS

- A C++ statement is a step in the recipe of the program
- either declaring a new symbol for later use, expressing a computation or some other action on pre-declared symbols
- Blocks are groups of statements enclosed by a pair of braces.

```
1  { // begin : block 0
2      auto i = 0;
3      while (i >= 0) { // begin : block 1
4          // calc with i
5          { // begin : block 2
6              auto x = cos(i * pi/180);
7              auto y = sin(i * pi/180);
8              // more
9          } // end : block 2
10     } // end : block 1
11 } // end : block 0
```

SCOPE OF VARIABLE NAMES

```
1 auto find_root() -> double
2 {
3     for (int i = 0; i < N; ++i) {
4         //counter i defined only in this "for" loop.
5     }
6     double newval = 0; // This is ok.
7     for (int i = 0; i < N; ++i) {
8         // The counter i here is a different entity
9         if (newval < 5) {
10             string fl{"small.dat"};
11             // do something
12         }
13         newval=...;
14         cout << fl << '\n'; // Error!
15     }
16     int fl = 42; // ok, but shadowed below
17     if (auto fl = filename; val < 5) { // C++17
18         // fl is available here
19     } else {
20         // fl is also available here
21     }
22 }
```

- Variable declarations are allowed throughout the code
- The scope of a variable is the lines of code where a variable can be accessed
- A scope is :
 - For variables declared in a block, bounded by { and }, the lines from the point of declaration till the }
 - A loop or a function body
 - Both if and else parts of an if statement
- Variables defined in a block exist from the point of declaration till the end of the scope. After that, the name may be reused.
- Type attached to a name at any point in a C++ program can always be determined by the examining scopes and declarations, without considering the path taken at runtime to reach that point

SCOPE OF VARIABLE NAMES

```
1 // Somewhere in a function ...
2 auto imp = imp_calc();
3 while (some_condition_holds) {
4     // Calculations
5     // more calc
6     // more calc
7     if (imp > 0) {
8
9     } else {
10
11    }
12    // hundred more lines till the end
13    // of the while loop body
```

- To deduce the type of entity the symbol `imp` represents in line 7, you have to look upwards from that point to the nearest declaration for that name.
- Nothing that happens in the loop can change this deduction
- Nature and properties of symbols in C++ can always be deduced by a purely spatial analysis in the space of source lines.
- Static typing and C++ scoping rules ensure that we don't have to perform a space-time analysis

CONSTANTS

```
1  auto G = 6.674e-11 ;
2  auto pi = 3.141592653589793 ;
3  auto m1 = 1.0e10, m2 = 1.0e4;
4  auto r = 10;
5  std::cout << "Force = "
6      << -G * m1 * m2 / (r * r)
7      << "\n"; // great!
8  G = G + 1;
9  std::cout << "Force = "
10     << -G * m1 * m2 / (r * r)
11     << "\n"; // wrong!
12
13 for (auto i = 0; i < 360; ++pi) {
14     std::cout << sin(i * pi / 180);
15 }
```

- Some entities we need in computations should not be able to change

CONSTANTS

```
1  auto G = 6.674e-11 ;
2  auto pi = 3.141592653589793 ;
3  auto m1 = 1.0e10, m2 = 1.0e4;
4  auto r = 10;
5  std::cout << "Force = "
6      << -G * m1 * m2 / (r * r)
7      << "\n"; // great!
8  G = G + 1;
9  std::cout << "Force = "
10     << -G * m1 * m2 / (r * r)
11     << "\n"; // wrong!
12
13  for (auto i = 0; i < 360; ++pi) {
14      std::cout << sin(i * pi / 180);
15  }
```

- Some entities we need in computations should not be able to change
- Simple typos might lead to horribly incorrect (if we are lucky) or subtly incorrect results which can go unnoticed for a long time

CONSTANTS

```
1  auto const G = 6.674e-11 ;
2  auto const pi = 3.141592653589793 ;
3  auto m1 = 1.0e10, m2 = 1.0e4;
4  auto r = 10;
5  std::cout << "Force = "
6      << -G * m1 * m2 / (r * r)
7      << "\n"; // great!
8  G = G + 1;
9  std::cout << "Force = "
10     << -G * m1 * m2 / (r * r)
11     << "\n"; // wrong!
12
13 for (auto i = 0; i < 360; ++pi) {
14     std::cout << sin(i * pi / 180);
15 }
```

- Some entities we need in computations should not be able to change
- Simple typos might lead to horribly incorrect (if we are lucky) or subtly incorrect results which can go unnoticed for a long time
- The `const` qualifier in C++ is used to mark variables as constants

CONSTANTS

```
1  auto const G = 6.674e-11 ;
2  auto const pi = 3.141592653589793 ;
3  auto m1 = 1.0e10, m2 = 1.0e4;
4  auto r = 10;
5  std::cout << "Force = "
6      << -G * m1 * m2 / (r * r)
7      << "\n"; // great!
8  G = G + 1; // compiler error!
9  std::cout << "Force = "
10     << -G * m1 * m2 / (r * r)
11     << "\n"; // wrong!
12
13  for (auto i = 0; i < 360; ++pi) {
14      // compiler error!
15      std::cout << sin(i * pi / 180);
16  }
```

- Some entities we need in computations should not be able to change
- Simple typos might lead to horribly incorrect (if we are lucky) or subtly incorrect results which can go unnoticed for a long time
- The `const` qualifier in C++ is used to mark variables as constants
- Attempting to modify a `const` qualified variable is a compiler error, so that we immediately notice such errors

CONSTANTS

```
1  auto const G = 6.674e-11 ;
2  auto const pi = 3.141592653589793 ;
3  auto m1 = 1.0e10, m2 = 1.0e4;
4  auto r = 10;
5  std::cout << "Force = "
6      << -G * m1 * m2 / (r * r)
7      << "\n"; // great!
8  G = G + 1; // compiler error!
9  std::cout << "Force = "
10     << -G * m1 * m2 / (r * r)
11     << "\n"; // wrong!
12
13  for (auto i = 0; i < 360; ++pi) {
14      // compiler error!
15      std::cout << sin(i * pi / 180);
16  }
```

- Some entities we need in computations should not be able to change
- Simple typos might lead to horribly incorrect (if we are lucky) or subtly incorrect results which can go unnoticed for a long time
- The `const` qualifier in C++ is used to mark variables as constants
- Attempting to modify a `const` qualified variable is a compiler error, so that we immediately notice such errors
- In general fewer mutable variables makes code easier to debug, so that making a habit of first making all new variables `const` and then consciously relaxing the qualifier for some is now considered good practice.

CONSTANTS

```
1  auto ask_user() -> double
2  {
3      double tmp{};
4      std::cout << "Enter R0: ";
5      std::cin >> tmp;
6      return tmp;
7  }
8  void elsewhere()
9  {
10     const auto r = ask_user(); // OK
11     r = r * r; // Not OK
12 }
```

- `const` does not mean compile time constant. Just that the variable will not be changed post initialisation.

CONSTANTS

```
1  constexpr auto G = 6.674e-11 ;
2  constexpr auto pi = 3.141592653589793 ;
3  auto m1 = 1.0e10, m2 = 1.0e4;
4  auto r = 10;
5  std::cout << "Force = "
6      << -G * m1 * m2 / (r * r)
7      << "\n"; // great!
8  G = G + 1; Compiler error
9  std::cout << "Force = "
10     << -G * m1 * m2 / (r * r)
11     << "\n"; // wrong!
12
13 for (auto i = 0; i < 360; ++pi) {
14     // Compiler error!
15     std::cout << sin(i * pi / 180);
16 }
```

- `const` does not mean compile time constant. Just that the variable will not be changed post initialisation.
- For variables known to be compile time constants, one could use `constexpr`

CONSTANTS

```
1  constexpr auto G = 6.674e-11 ;
2  constexpr auto pi = 3.141592653589793 ;
3  auto m1 = 1.0e10, m2 = 1.0e4;
4  auto r = 10;
5  std::cout << "Force = "
6      << -G * m1 * m2 / (r * r)
7      << "\n"; // great!
8  G = G + 1; Compiler error
9  std::cout << "Force = "
10     << -G * m1 * m2 / (r * r)
11     << "\n"; // wrong!
12
13 for (auto i = 0; i < 360; ++pi) {
14     // Compiler error!
15     std::cout << sin(i * pi / 180);
16 }
```

- `const` does not mean compile time constant. Just that the variable will not be changed post initialisation.
- For variables known to be compile time constants, one could use `constexpr`
- The compiler may use the value of such variables to produce better code

REFERENCES

```
1  const auto x{5.0};  
2  const double y{6.0};  
3  
4  // different entities with same initial values  
5  auto x2{ x }; // Obs: x2 is not const!  
6  double y2{ y };
```

- Variable declaration: create object with initial value, and attach a name tag (reference) to it

REFERENCES

```
1  const auto x{5.0};  
2  const double y{6.0};  
3  
4  // different entities with same initial values  
5  auto x2{ x }; // Obs: x2 is not const!  
6  double y2{ y };
```

- Variable declaration: create object with initial value, and attach a name tag (reference) to it
- If a variable name is used to initialise a new one, `auto x2{x}`, the new variable will have the same value, but will be a different entity

REFERENCES

```
1  const auto x{5.0};
2  const double y{6.0};
3
4  // different entities with same initial values
5  auto x2{ x }; // Obs: x2 is not const!
6  double y2{ y };
7
8  // additional references for the same object
9  const auto& xr{ x };
10 const double& yr{ y };
```

- Variable declaration: create object with initial value, and attach a name tag (reference) to it
- If a variable name is used to initialise a new one, `auto x2{ x }`, the new variable will have the same value, but will be a different entity
- It is possible to “attach another name tag” to an existing variable.

REFERENCES

```
1  const auto x{5.0};
2  const double y{6.0};
3
4  // different entities with same initial values
5  auto x2{ x }; // Obs: x2 is not const!
6  double y2{ y };
7
8  // additional references for the same object
9  const auto& xr{ x };
10 const double& yr{ y };
```

- Variable declaration: create object with initial value, and attach a name tag (reference) to it
- If a variable name is used to initialise a new one, `auto x2{ x }`, the new variable will have the same value, but will be a different entity
- It is possible to “attach another name tag” to an existing variable.
- Since the new names are not independent objects, they can't have greater modification privileges compared to the original variable name

REFERENCES

```
1  const auto x{5.0};
2  const double y{6.0};
3
4  // different entities with same initial values
5  auto x2{ x }; // Obs: x2 is not const!
6  double y2{ y };
7
8  // additional references for the same object
9  const auto& xr{ x };
10 const double& yr{ y };
```

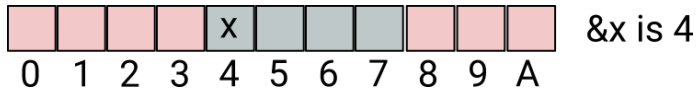
- Variable declaration: create object with initial value, and attach a name tag (reference) to it
- If a variable name is used to initialise a new one, `auto x2{x}`, the new variable will have the same value, but will be a different entity
- It is possible to “attach another name tag” to an existing variable.
- Since the new names are not independent objects, they can't have greater modification privileges compared to the original variable name
- `xr` and `yr` here are constant L-value references of type `double`

REFERENCES

```
1  const auto x{5.0};
2  const double y{6.0};
3
4  // different entities with same initial values
5  auto x2{ x }; // Obs: x2 is not const!
6  double y2{ y };
7
8  // additional references for the same object
9  const auto& xr{ x };
10 const double& yr{ y };
```

- Variable declaration: create object with initial value, and attach a name tag (reference) to it
- If a variable name is used to initialise a new one, `auto x2{x}`, the new variable will have the same value, but will be a different entity
- It is possible to “attach another name tag” to an existing variable.
- Since the new names are not independent objects, they can't have greater modification privileges compared to the original variable name
- `xr` and `yr` here are constant L-value references of type `double`
- References are important for information exchange with functions

POINTERS



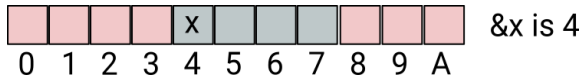
```
1  int i{5};
2  int* iptr{&i}; // iptr points at i
3  i += 1;
4  std::cout << *iptr ; // 6
5  (*iptr) = 0;
6  std::cout << i ; // 0
7  int& iref{i}; // iref "refers" to i
8  iref = 4;
9  std::cout << i ; // 4
```

- A pointer is a built in type to store the memory address of objects, with its own different arithmetic rules
- For a variable `X`, its memory address is `&X`

- If `iptr` is a pointer, `*iptr` is the object it is pointing at
- Adding 1 to the pointer `iptr` shifts it by `sizeof(typeof i)` bytes in memory
- A reference is effectively another name for the same object
- When in use, a reference appears as if it were a regular variable

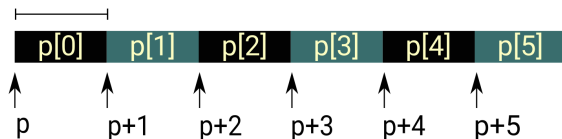
POINTERS

- Imagine computer memory as a long sequence of bytes where information is stored
- Imagine all the bytes being numbered like houses in a very long street
- An `int` object in a program would be stored somewhere, and occupy 4 bytes, the address of its first byte is called the address of the integer. If the integer object has a name `x`, its address can be found as `&x`
- If multiple `int` objects are stored next to each other, with no gaps, address of the integer coming after `x` is `sizeof(x)` bytes after `&x`
- The address of an object of any type `T`, can be stored in variables of type `T*`, pointers to `T`.



- `int*` is different from `double*`, `char*` and even `unsigned int*` or `const int*`
- For any given type `T`, if `sizeof(T) == n`, pointers of that type jump `n` bytes when we add 1 to them

n bytes



POINTERS

- If `p` is a pointer to an `T`, `*p` is a reference to `T`. This process of getting a reference out of a pointer is called “dereferencing”.
- If `T` is a class type, and `p` is a pointer to `T`, members for the current object `p` is pointing to can be accessed as `p->member` or `(*p).member`
- If `x` is of type `T`, `&x` is of type `T*`. This implies that the pointer for a `const` object is also `const` qualified
- In some ways references behave like fixed, automatically dereferenced pointers. But pointers are themselves object types. They themselves have addresses and sizes. They can be stored in arrays. References can not be.
- If `p` is a pointer holding the address of an element of an array of type `T`, `p+1`, `p+2` ... are the subsequent elements.
- `*(p+2)` is synonymous with `p[2]`, `*(p+1)` with `p[1]` and, `*p` with `p[0]`.
- `p` is the same location as `&p[0]`

BRANCHES/SELECTIONS

```
1  if (condition) {  
2    // code  
3  } else if (another condition) {  
4    // code  
5  } else {  
6    //code  
7  }  
8  switch (enumerable) {  
9  case 1:  
10    // code  
11    break;  
12  case 2:  
13    // code  
14    break;  
15  default:  
16    // code  
17  };  
18  x = N > 10 ? 1.0 : 0.0;
```

- The `if` and `switch` constructs can be used to select between different alternatives at execution time.
- Conditional assignments are frequently written with the `ternary operator` as shown

LOOPS

```
1  for (initialisation; condition; increment) {  
2      // Loop body  
3  }  
4  for (int i = 0; i < N; ++i) s += a[i];  
5  while (condition) {}  
6  while (T > t0) {}  
7  do {} while (condition);  
8  do {  
9  } while (ch == 'y');  
10 for (variable : collection) {}  
11 for (int i : {1,2,3}) f(i);  
12 for (int i = 0; i < N; ++i) {  
13     if (a[i] < cutoff) s+=a[i];  
14     else break;  
15 }  
16 for (std::string s : names) {  
17     if (s.size() > 10) {  
18         longnames.push_back(s);  
19         continue;  
20     }  
21     // process other names  
22 }
```

- Execute a block of code repeatedly
- Loop counter for the `for` loop can and should usually be declared in the loop head
- The `break` keyword in a loop immediately stops the loop and jumps to the code following it
- The `continue` keyword skips all remaining statements in the current iteration, and continues in the loop

POINTERS

```
int A[10]{0, 2, 1, 0, 3, 1, 1, 0, 0, 1};  
int w{};  
for (int i = 0; i < 10; ++i) w += A[i];
```

What does this code do ?

POINTERS

```
int A[10]{0, 2, 1, 0, 3, 1, 1, 0, 0, 1};  
int w{};  
for (int i = 0; i < 10; ++i) w += *(A+i);
```

What does this code do ?

POINTERS

```
int A[10]{0, 2, 1, 0, 3, 1, 1, 0, 0, 1};  
int w{};  
for (int* p{A}; p != A + 10; ++p) w += *p;
```

What does this code do ?

POINTERS

```
int A[10]{0, 2, 1, 0, 3, 1, 1, 0, 0, 1};  
int w{};  
int* start{A};  
int* stop{A + 10};  
for (int* p{start}; p != stop; ++p) {  
    w += *p;  
}
```

What does this code do ?

POINTERS

```
int A[10]{0, 2, 1, 0, 3, 1, 1, 0, 0, 1};  
int w{};  
int* start{A};  
int* stop{A + 10};  
for (; start != stop; ++start) w += *start;
```

What does this code do ?

POINTERS

```
auto whatisit(int* start, int* stop) -> int
{
    int w{};
    for (; start != stop; ++start) w += *start;
    return w;
}
```

What does this code do ?

POINTERS

```
void whatisit(int* start, int* stop, int* start2)
{
    for (; start != stop; ++start, ++start2) *start2 = *start;
}
```

What does this code do ?

Exercise 1.5:

The basic concepts of the language are explained using a series of Jupyter notebooks in the folder `notebooks` in the course materials. Depending on your previous knowledge, you may need to focus on different topics. The notebooks are full of explanatory text. Work through the notebooks `Fundamentals_1.ipynb`, and `Fundamentals_2.ipynb`, before we continue. Ask any topic that you find unclear and needs an explanation.

FUNCTIONS

```
1 auto function_name(parameters) -> return_type
2 {
3     // function body
4 }
5 auto sin(double x) -> double
6 {
7     // Somehow calculate sin of x
8     return answer;
9 }
10 auto main() -> int
11 {
12     constexpr double pi{3.141592653589793};
13     for (int i = 0; i < 100; ++i) {
14         std::cout << i * pi / 100
15             << sin(i * pi / 100) << "\n";
16     }
17     std::cout << sin("pi") << "\n"; //Error!
18 }
```

- To the first approximation, all executable code is in functions
- In order to execute the code in a function, we “call” the function
- `main` is a special function. When you run a program, the OS, the debugger or IDE, calls `main`. The code in `main` may call other functions, which call even more functions and so on, till all work in `main` is done
- A function can receive some data as input and manipulate the information provided in its input, and “return” some information as its output
- The input to a function comes through its arguments, and the output is called its return value.

FUNCTIONS: SYNTAX

```
1 // Old syntax
2 bool pythag(int i, int j, int k); // prototype
3 int hola(int i, int j) // definition
4 {
5     int ans{0};
6     if (pythag(i,j,23)) {
7         // A prototype or definition must be
8         // visible in the translation unit
9         // at the point of usage
10        ans=42;
11    }
12    return ans;
13 }
14 // Definition of pythag. Not that old syntax
15 auto pythag(int i, int j, int k) -> bool
16 {
17     // code
18 }
```

- A function prototype introduces a **name** as a function, its **return type** as well as its **parameters**
- The type of the arguments must match or be implicitly convertible to the corresponding type in the function parameter list

```
1 auto max(double x, double y, double z)
2     -> double
3 {
4     if (y > x) x = y;
5     if (z > x) x = z;
6     return x;
7 }
8 auto main(int argc, char * argv[]) -> int
9 {
10     std::cout << max(1., 2., 3.) << '\n';
11 }
```

Exercise 1.6:

Write a function to tell if a quadratic equation of the form $ax^2 + bx + c = 0$ has real number roots. The function should take 3 arguments of type `double`, and return either true or false.

Exercise 1.7:

Finish the program `examples/gcd.cc` so that it computes and prints the greatest common divisor of two integers. The following algorithm (attributed to Euclid!) achieves it :

- 1 Input numbers : smaller , larger
- 2 remainder = larger mod smaller
- 3 larger = smaller
- 4 smaller = remainder
- 5 if smaller is not 0, go back to 2.
- 6 larger is the answer you are looking for

Note: There is a function `std::gcd(n1, n2)` since C++17, but we are not using it for this exercise.

FUNCTIONS AT RUN TIME

Sin(double x)
x:0.125663..

RP:<in main()>

main() x:3.14159265...
i:4
RP:OS

```
1 auto sin(double x) -> int {  
2     // Somehow calculate sin of x  
3     return answer;  
4 }  
5 auto main() -> int {  
6     double x{3.141592653589793};  
7     for (int i = 0; i < 100; ++i) {  
8         std::cout << i * x / 100  
9         << sin(i * x / 100) << "\n";  
10    }  
11 }
```

When a function is called, e.g., when we write

`f(value1,value2,value3)` for a function `f` declared as

`ret_type f(type1 x, type2 y, type3 z) :`

- A "workbook" in memory called a stack frame is created for the call
- The local variables `x`, `y`, `z` are created, as if using instructions `type1 x{value1}`, `type2 y{value2}`, `type3 z{value3}`.
- A return address is stored.
- The actual body of the function is executed
- When the function concludes, execution continues at the stored return address, and the stack frame is destroyed

RECURSION

- A function calling itself
- Each level of "recursion" has its own stack frame

■ SP=<in someother()> RP=<...>

```
1  auto factorial(unsigned int n) -> unsigned int
2  {
3      int u = n; // u: Unnecessary
4      if (n > 1) return n * factorial(n - 1);
5      else return 1;
6  }
7  auto someother() -> int
8  {
9      factorial(4);
10 }
```

RECURSION

- $SP = \langle \text{in factorial()} \rangle$ $n=4$ $u=4$ $RP = \langle 9 \rangle$
- $SP = \langle \text{in someother()} \rangle$ $RP = \langle \dots \rangle$

```
1  auto factorial(unsigned int n) -> unsigned int
2  {
3      int u = n; // u: Unnecessary
4      if (n > 1) return n * factorial(n - 1);
5      else return 1;
6  }
7  auto someother() -> int
8  {
9      factorial(4);
10 }
```

- A function calling itself
- Each level of "recursion" has its own stack frame
- Function parameters are copied to the stack frame

RECURSION

- $SP = \langle \text{in factorial()} \rangle$ $n=3$ $u=3$ $RP = \langle 4 \rangle$
- $SP = \langle \text{in factorial()} \rangle$ $n=4$ $u=4$ $RP = \langle 9 \rangle$
- $SP = \langle \text{in someother()} \rangle$ $RP = \langle \dots \rangle$

```
1  auto factorial(unsigned int n) -> unsigned int
2  {
3      int u = n; // u: Unnecessary
4      if (n > 1) return n * factorial(n - 1);
5      else return 1;
6  }
7  auto someother() -> int
8  {
9      factorial(4);
10 }
```

- A function calling itself
- Each level of "recursion" has its own stack frame
- Function parameters are copied to the stack frame
- Local variables at different levels of recursion live in their own stack frames, and do not interfere

RECURSION

- $SP = \langle \text{in factorial()} \rangle$ $n=2$ $u=2$ $RP = \langle 4 \rangle$
- $SP = \langle \text{in factorial()} \rangle$ $n=3$ $u=3$ $RP = \langle 4 \rangle$
- $SP = \langle \text{in factorial()} \rangle$ $n=4$ $u=4$ $RP = \langle 9 \rangle$
- $SP = \langle \text{in someother()} \rangle$ $RP = \langle \dots \rangle$

```
1  auto factorial(unsigned int n) -> unsigned int
2  {
3      int u = n; // u: Unnecessary
4      if (n > 1) return n * factorial(n - 1);
5      else return 1;
6  }
7  auto someother() -> int
8  {
9      factorial(4);
10 }
```

- A function calling itself
- Each level of "recursion" has its own stack frame
- Function parameters are copied to the stack frame
- Local variables at different levels of recursion live in their own stack frames, and do not interfere

RECURSION

- $SP = \langle \text{in factorial()} \rangle$ $n=1$ $u=1$ $RP = \langle 4 \rangle$
- $SP = \langle \text{in factorial()} \rangle$ $n=2$ $u=2$ $RP = \langle 4 \rangle$
- $SP = \langle \text{in factorial()} \rangle$ $n=3$ $u=3$ $RP = \langle 4 \rangle$
- $SP = \langle \text{in factorial()} \rangle$ $n=4$ $u=4$ $RP = \langle 9 \rangle$
- $SP = \langle \text{in someother()} \rangle$ $RP = \langle \dots \rangle$

```
1  auto factorial(unsigned int n) -> unsigned int
2  {
3      int u = n; // u: Unnecessary
4      if (n > 1) return n * factorial(n - 1);
5      else return 1;
6  }
7  auto someother() -> int
8  {
9      factorial(4);
10 }
```

- A function calling itself
- Each level of "recursion" has its own stack frame
- Function parameters are copied to the stack frame
- Local variables at different levels of recursion live in their own stack frames, and do not interfere

RECURSION

- $SP = \langle \text{in factorial()} \rangle$ $n=2$ $u=2$ $RP = \langle 4 \rangle$
- $SP = \langle \text{in factorial()} \rangle$ $n=3$ $u=3$ $RP = \langle 4 \rangle$
- $SP = \langle \text{in factorial()} \rangle$ $n=4$ $u=4$ $RP = \langle 9 \rangle$
- $SP = \langle \text{in someother()} \rangle$ $RP = \langle \dots \rangle$

```
1  auto factorial(unsigned int n) -> unsigned int
2  {
3      int u = n; // u: Unnecessary
4      if (n > 1) return n * factorial(n - 1);
5      else return 1;
6  }
7  auto someother() -> int
8  {
9      factorial(4);
10 }
```

- A function calling itself
- Each level of "recursion" has its own stack frame
- Function parameters are copied to the stack frame
- Local variables at different levels of recursion live in their own stack frames, and do not interfere

RECURSION

- $SP = \langle \text{in factorial()} \rangle$ $n=3$ $u=3$ $RP = \langle 4 \rangle$
- $SP = \langle \text{in factorial()} \rangle$ $n=4$ $u=4$ $RP = \langle 9 \rangle$
- $SP = \langle \text{in someother()} \rangle$ $RP = \langle \dots \rangle$

```
1  auto factorial(unsigned int n) -> unsigned int
2  {
3      int u = n; // u: Unnecessary
4      if (n > 1) return n * factorial(n - 1);
5      else return 1;
6  }
7  auto someother() -> int
8  {
9      factorial(4);
10 }
```

- A function calling itself
- Each level of "recursion" has its own stack frame
- Function parameters are copied to the stack frame
- Local variables at different levels of recursion live in their own stack frames, and do not interfere

RECURSION

- $SP = \langle \text{in factorial()} \rangle$ $n=4$ $u=4$ $RP = \langle 9 \rangle$
- $SP = \langle \text{in someother()} \rangle$ $RP = \langle \dots \rangle$

```
1  auto factorial(unsigned int n) -> unsigned int
2  {
3      int u = n; // u: Unnecessary
4      if (n > 1) return n * factorial(n - 1);
5      else return 1;
6  }
7  auto someother() -> int
8  {
9      factorial(4);
10 }
```

- A function calling itself
- Each level of "recursion" has its own stack frame
- Function parameters are copied to the stack frame
- Local variables at different levels of recursion live in their own stack frames, and do not interfere

Exercise 1.8:

The tower of Hanoi is a mathematical puzzle with three towers and a set of disks of increasing sizes. In the beginning, all the disks are at one tower. In each step, a disk can be moved from one tower to another, with the rule that a larger disk must never be placed over a smaller one. The example `examples/hanoi.cc` solves the puzzle for a given input number of disks, using a recursive algorithm. Test the code and verify the solution.



STATIC VARIABLES IN FUNCTIONS

```
1 void somefunc()
2 {
3     static int ncalls=0;
4     ++ncalls;
5     // code --> something unexpected
6     std::cerr << "Encountered unexpected"
7         << "situation in the " << ncalls
8         << "th call to " << __func__ << "\n";
9 }
```

- Private to the function, but survive from call to call.
- Initialisation only done on first call.
- **Aside:** The built in macro `__func__` always stores the name of the function

FUNCTION OVERLOADING

```
1  auto power(int x, unsigned n) -> unsigned
2  {
3      ans = 1;
4      for (; n > 0; --n) ans *= x;
5      return ans;
6  }
7  auto power(double x, double y) -> double
8  {
9      return exp(y * log(x));
10 }
```

```
1  auto someother(double mu, double alpha,
2                  int rank) -> double
3  {
4      double st=power(mu,alpha)*exp(-mu);
5
6      if (n_on_bits(power(rank,5))<8)
7          st=0;
8
9      return st;
10 }
```

- The same function name can be used for different functions if the parameter list is different
- Function name and the types of its parameters are combined to create an "internal" name for a function. That name must be unique
- It is not allowed for two functions to have the same name and parameters and differ only in the return value
- Make as many functions as you need with the same name, if the number or types of the input parameters are different. Just make sure the names tell you syntactically what they do, without having to look at the implementation. E.g., good names: `max`, `min`, `power`, bad names: `do_stuff`, unnecessary names `power_d_d`, `power_i_u`

FUNCTION OVERLOADING

```
1  auto power(int x, unsigned n) -> unsigned
2  {
3      ans = 1;
4      for (; n > 0; --n) ans *= x;
5      return ans;
6  }
7  auto power(double x, double y) -> double
8  {
9      return exp(y * log(x));
10 }
```

```
1  auto someother(double mu, double alpha,
2                  int rank) -> double
3  {
4      double st=power(mu,alpha)*exp(-mu);
5
6      if (n_on_bits(power(rank,5))<8)
7          st=0;
8
9      return st;
10 }
```

- The group of functions with the same name, differing in their input parameter list, is called an “overload set”
- It is useful to assign meaning to these overload sets, and think in terms of them. The individual functions inside an overload set are details depending on things like whether an input is an integer or a double.
- The compiler to find the correct match from the overload set. This kind of *polymorphic* behaviour costs nothing at run time.

USER DEFINED TYPES AND OPERATOR OVERLOADING

```
1  struct AtomId { int val = 0; };
2  struct MolId { int val = 0; };
3
4  void display_info(AtomId i)
5  {
6      // show atom related info
7  }
8  void display_info(MolId i)
9  {
10     // display completely different
11     // stuff about molecule
12 }
13 void elsewhere()
14 {
15     MolId j = select_a_molecule();
16     for (AtomId i; i.val < ; ++i.val) {
17         if (i == j) { // Compiler error!
18             //
19         }
20     }
21 }
```

- `struct` or `class` introduce new types to a program. We leave details for later, but for now, just observe how we bring a new category of variables like `int` or `double` in to existence
- We can create variables of the new type, pass them to functions as arguments ...
- Functions can be overloaded with user defined types

USER DEFINED TYPES AND OPERATOR OVERLOADING

```
1 struct AtomId { int val = 0; };
2 struct MolId { int val = 0; };
3
4 void display_info(AtomId i)
5 {
6     // show atom related info
7 }
8 void display_info(MolId i)
9 {
10    // display completely different
11    // stuff about molecule
12 }
13 void elsewhere()
14 {
15     MolId j = select_a_molecule();
16     for (AtomId i; i.val < ; ++i.val) {
17         if (i == j) { // Compiler error!
18             //
19         }
20     }
21 }
```

- **struct** or **class** introduce new types to a program. We leave details for later, but for now, just observe how we bring a new category of variables like **int** or **double** in to existence
- We can create variables of the new type, pass them to functions as arguments ...
- Functions can be overloaded with user defined types
- Operators can be overloaded with user defined types

```
1 struct minutes { int i = 0; };
2 auto operator+(minutes m1, minutes m2) -> minutes
3 {
4     return { (m1.i + m2.i) % 60 };
5 }
6 // elsewhere with i and j of type minutes
7 auto k = i + j;
```

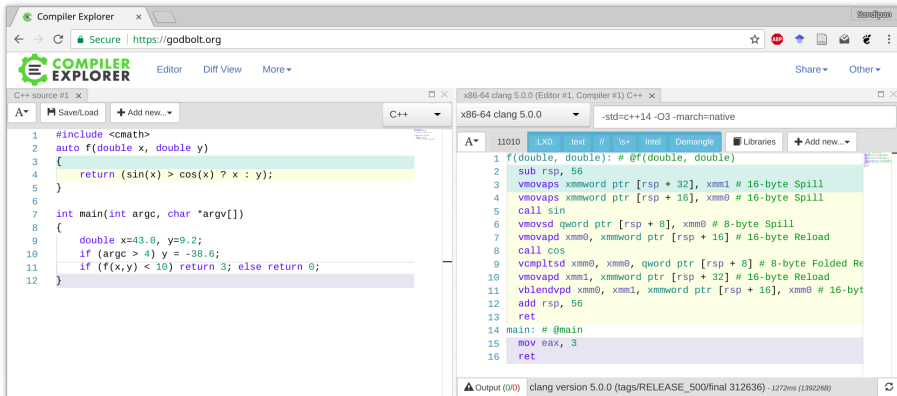
INLINE FUNCTIONS

```
1 auto sqr(double x) -> double
2 {
3     return x * x;
4 }
```

```
1 inline auto sqr(double x) -> double
2 {
3     return x * x;
4 }
```

- To eliminate overhead when a function is called, request the compiler to insert the entire function body where it is called, preserving the function call semantics
- Once a function is inlined, the calling function can be further optimised as if it was one function
- Small frequently called functions are usual candidates
- Compiler may or may not actually insert code inline, but any function marked inline is exempt from the “one definition rule”
- Different popular use: define the entire function (even if it is large) in the header file, as identical inline objects in multiple translation units are allowed. (E.g. header only libraries)

INLINE FUNCTIONS



The screenshot shows the Compiler Explorer interface. The left pane displays the C++ source code for a program that includes `<cmath>`, defines an inline function `f` that returns `sin(x) > cos(x) ? x : y`, and has a `main` function that calls `f`. The right pane shows the assembly output for the `f` function, which is inlined into the `main` function. The assembly code for `f` includes stack frame setup, calls to `sin` and `cos`, and a conditional move instruction. The `main` function assembly shows the call to `f` and the final return statement.

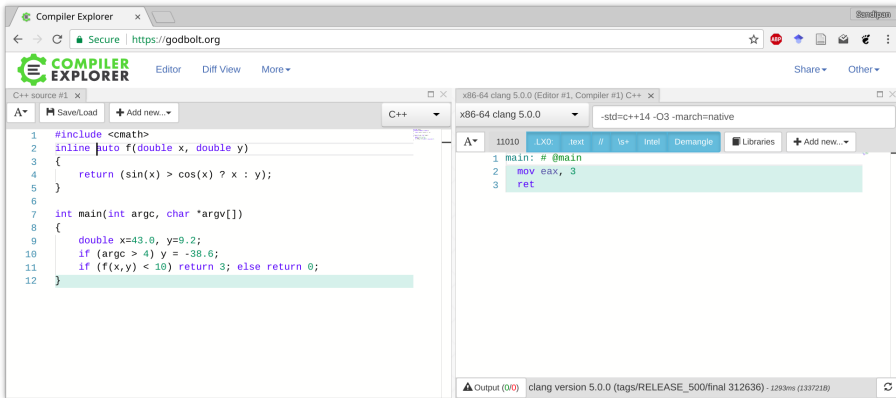
```
#include <cmath>
auto f(double x, double y)
{
    return (sin(x) > cos(x) ? x : y);
}

int main(int argc, char *argv[])
{
    double x=43.0, y=9.2;
    if (argc > 4) y = -38.6;
    if (f(x,y) < 10) return 3; else return 0;
}
```

```
f(double, double): # @f(double, double)
2   sub rsp, 56
3   vmovaps xmmword ptr [rsp + 32], xmm1 # 16-byte Spill
4   vmovaps xmmword ptr [rsp + 16], xmm0 # 16-byte Spill
5   call sin
6   vmovsd qword ptr [rsp + 8], xmm0 # 8-byte Spill
7   vmovapd xmm0, xmmword ptr [rsp + 16] # 16-byte Reload
8   call cos
9   vcmpltsd xmm0, xmm0, qword ptr [rsp + 8] # 8-byte Folded Re
10  vmovapd xmm1, xmmword ptr [rsp + 32] # 16-byte Reload
11  vblendvpd xmm0, xmm1, xmmword ptr [rsp + 16], xmm0 # 16-byt
12  add rsp, 56
13  ret
main: # @main
15  mov eax, 3
16  ret
```

- No assembly is generated unless the function is used
- Large files with lots of inline functions may slow down compilation, but the compiled machine code is not necessarily larger

INLINE FUNCTIONS



- No assembly is generated unless the function is used
- Large files with lots of inline functions may slow down compilation, but the compiled machine code is not necessarily larger

ANOTHER USE OF INLINE

- At each point in code, when we refer to the name of a variable, function, class, template, concept etc., it must be unambiguous
- One definition rule (ODR): Only one definition of any such entity is allowed in any translation unit
- Only one definition of an entity is allowed to appear in the entire program including the sources and any linked libraries
- Variables and functions declared as `inline` can appear in multiple translation units. These multiple incarnations are regarded as the same entity by the linker.
- Functions and variables (in global scope) defined in headers can be labeled as `inline` so that multiple instances in different translation units do not conflict
- General function templates are automatically `inline`

AUTO RETURN TYPE FOR FUNCTIONS

- Since C++14, automatic type deduction can be used for function return values
- Return type ambiguity will be a compiler error in such situations
- `decltype(auto)` can also be used like `auto` for the return type, along with the different type deduction rules which apply for `decltype(auto)` (Later!)

```
1 auto greet(std::string nm)
2 {
3     for (auto& c: nm) c = std::toupper(c);
4     std::cout << nm << std::endl;
5     return nm.size() > 10;
6 }
```

LAMBDA FUNCTIONS

```
1 void onefunc(double inp) -> double
2 {
3     const auto x{ inp };
4
5     auto anotherfunc(double in) -> double
6     {
7         return in * in;
8     }
9
10    x = inp * anotherfunc(x);
11    return x;
12 }
```

- In C++, ordinary functions can not be defined locally in block scope

LAMBDA FUNCTIONS

```
1 void onefunc(double inp) -> double
2 {
3     const auto x{ inp };
4
5     auto anotherfunc = [](double in) -> double
6     {
7         return in * in;
8     };
9
10    x = inp * anotherfunc(x);
11    return x;
12 }
```

- In C++, ordinary functions can not be defined locally in block scope
- That is the role of lambda functions

LAMBDA FUNCTIONS

```
1 void onefunc(double inp) -> double
2 {
3     const auto x{ inp };
4
5     auto anotherfunc = [] (double in) -> double
6     {
7         return in * in;
8     };
9
10    x = inp * anotherfunc(x);
11    return x;
12 }
```

- In C++, ordinary functions can not be defined locally in block scope
- That is the role of lambda functions
- Introduced using lambda expressions

LAMBDA FUNCTIONS

```
1 void onefunc(double inp) -> double
2 {
3     const auto x{ inp };
4
5     auto anotherfunc = [x](double in) -> double
6     {
7         return in * in * sin(x);
8     };
9
10    x = inp * anotherfunc(x);
11    return x;
12 }
```

- In C++, ordinary functions can not be defined locally in block scope
- That is the role of lambda functions
- Introduced using lambda expressions
- The starting square brackets are called “capture brackets”, and they can make in-scope variables visible inside the lambda. We can choose how much of its environment is visible inside the lambda

CONSTEXPR AND CONSTEVAL FUNCTIONS

```
1  constexpr auto cube(unsigned u)
2  {
3      return u * u * u;
4  }
5  consteval auto cube2(unsigned u)
6  {
7      return u * u * u;
8  }
9  void elsewhere(unsigned inp)
10 {
11     std::array<int, cube(10)> A;
12     constexpr auto myvar = cube(99U);
13     auto myvar2 = cube(inp);
14
15     std::array<int, cube2(10)> B;
16     constexpr auto myvar = cube2(99U);
17     auto myvar2 = cube2(inp);
18 }
```

- A function can be declared `constexpr` or `consteval`. Both versions make them available for use at compilation time, to initialise `constexpr` variables or in contexts where only compile time constants are allowed
- `constexpr` functions can be called with values not known at compilation time, in which case they behave as ordinary functions
- It is a compiler error to call a `consteval` function with arguments with values not known at compilation time. `consteval` functions are called “immediate functions”

C++ NAMESPACE S

```
1 // Somewhere in the header iostream
2 namespace std {
3     ostream cout;
4 }
5 // In your program ...
6 #include <iostream>
7 auto main() -> int
8 {
9     {
10         using namespace std;
11         cout << __func__ << "\n";
12     }
13     int cout = 0;
14     for (cout=0; cout<5; ++cout)
15         std::cout << "Counter = " << cout << '\n';
16     // Above, plain cout is an integer,
17     // but std::cout is an output stream
18     // The syntax to refer to a name
19     // defined inside a namespace is:
20     // namespace_name::identifier_name
21 }
```

- A **namespace** is a named context in which variables, functions etc. are defined.
- The symbol **::** is called the **scope resolution operator**.
- **using namespace** blah imports all names declared inside the **namespace** blah to the current scope.

NAMESPACES

```
1 // examples/namespaces.cc
2 #include <iostream>
3 using namespace std;
4 namespace UnitedKingdom {
5     string London{"Big city"};
6     void load_slang() {...}
7 }
8 namespace UnitedStates {
9     string London{"Small town in Kentucky"};
10    void load_slang() {...}
11 }
12 auto main() -> int
13 {
14     using namespace UnitedKingdom;
15     cout << London << '\n';
16     cout << UnitedStates::London << '\n';
17 }
```

- Same name in different namespaces do not result in a name clash
- Functions defined inside namespaces need to be accessed using the same scope rules as variables

C++ NAMESPACES: FINAL COMMENTS

```
1 //examples/namespaces2.cc
2 #include <iostream>
3 namespace UnitedKingdom {
4     std::string London{"Big city"};
5 }
6 namespace UnitedStates {
7     namespace KY {
8         std::string London{" in Kentucky"};
9     }
10    namespace OH {
11        std::string London{" in Ohio"};
12    }
13 }
14 // With C++17 ...
15 namespace mylibrary::onefeature {
16     auto solve(int i) -> double;
17 }
18 auto main() -> int
19 {
20     namespace USOH=UnitedStates::OH;
21     std::cout << "London is "
22               << USOH::London << '\n';
23 }
```

- `namespace` s can be nested. Since C++17, direct nested declarations are allowed.
- Long `namespace` names can be given aliases
- Tip1: Don't indiscriminately put `using namespace ...` tags, especially in headers. Use them in tight scopes instead. Alternatively, define short aliases to long namespace names wherever you need to repeat them
- Tip2: The purpose of `namespace` s is to avoid name clashes. Not taxonomy!

ENUMERATIONS

```
1 enum color { red, green, blue };
2 // ...
3 color c{green};
4 // ...
5 switch (c) {
6     case red : do_stuff1(); break;
7     case green : do_stuff2(); break;
8     case blue:
9     default: do_stuff3();
10 };
```

- Internally represented as (and convertible to) an integer
- All type information is lost upon conversion into an integer

- A type whose instances can take a few different values (e.g., directions on the screen, colours, supported output modes ...)
- Less error prone than using integers with ad hoc rules like, "1 means red, 2 means green ..."

SCOPED ENUMERATIONS

- Defined with `enum class`
- Must always be fully qualified when used: `traffic_light::red` etc.
- In C++20, we can enable one specific `enum class` in a scope by using the `using enum XYZ;` declaration.
- No automatic conversion to `int`.
- Possible to use the same name, e.g., `green`, in two different scoped enums.

```
1  enum class color { red, green, blue };
2  enum class traffic_light {
3      red, yellow, green
4  };
5  bool should_brake(traffic_light c);
6
7  if (should_brake(blue)) apply_brakes();
8  //Syntax error!
9  if (state == traffic_light::yellow) ...;
10
11 auto respond(traffic_light L) {
12     using enum traffic_light;
13     switch (L) {
14         case red: {
15             //...
16         }
17     }
```

INPUT AND OUTPUT WITH IOSTREAM

- To read user input into variable `x`, simply write `std::cin >> x;`
- To read into variables `x`, `y`, `z`, `name` and `count`
`std::cin >> x >> y >> z >> name >> count;`
- `std::cin` will infer the type of input from the type of variable being read.
- For printing things on screen the direction for the arrows is towards `std::cout`:

```
std::cout << x << y << z << name << count << '\n';
```

READING AND WRITING FILES

- Declare your own source/sink objects, which will have properties like `std::cout` or `std::cin`

```
1  #include <fstream>
2  std::ifstream fin{"inputfile"};
3  // Or, std::ifstream fin; and later, fin.open("inputfile");
4  std::ofstream fout{"outputfile"};
```

- Use them like `std::cout` or `std::cin`

```
1  double x,y,z;
2  int i;
3  std::string s;
4  fin >> x >> y >> z >> i >> s;
5  fout << x << y << z << i << s << '\n';
```

STRING STREAMS

```
1  auto report(float x) -> std::string
2  {
3      auto a = f(x);
4      auto b = g(x);
5      // We need the output to be
6      // a string, perhaps to be
7      // processed further elsewhere.
8      std::ostringstream ost;
9      ost << "f(x) returned " << a << "\n";
10     ost << "g(x) returned " << b << "\n";
11     return ost.str();
12 }
```

- `ostringstream` is an output stream for output into a string.
- `istringstream` is an input stream to read values from a string.
- Same usage syntax as `cout` and `cin`

STREAM INPUT IN A LOOP

```
1  std::ifstream fin{"somefile.dat"};
2  double x;
3  while (fin >> x) {
4      // while it is possible to read a new
5      // value for x, do something.
6  }
7  std::string line;
8  while (getline(fin, line)) {
9      // while it is possible to read a
10     // line of input, do something
11 }
12 ifstream fin{ argv[1] };
13 for (auto it = istream_iterator<int>(fin);
14      it != istream_iterator<int>{};
15      ++it) {
16     std::cout << "Token : " << *it << "\n";
17 }
```

- Each of the 3 input stream types introduced here works as a boolean in conditionals or loop conditions.
- Loop ends when there is no more valid input
- We can even pretend they are sequences with "iterators" to their start and end

Exercise 1.9: Strings and I/O

Write a program to find the largest word in a plain text document.

EXAMPLE PROGRAMS USING FILE IO

```
1 // examples/onespace.cc
2 #include <iostream>
3 auto main(int argc, char* argv[]) -> int
4 {
5     std::string line;
6     while (getline(std::cin, line)) {
7         if (line.empty()) continue;
8         bool sp{true};
9         for (auto c : line) {
10             if (isspace(c)) {
11                 if (not sp) std::cout << '\t';
12                 sp = true;
13             } else {
14                 sp = false;
15                 std::cout << c;
16             }
17         }
18         std::cout << "\n";
19     }
20 }
```

Replace instances of multiple consecutive white space characters with a single TAB character

- Often needed to clean up data files formatted to look good to human eyes for processing with tools which rely on consistent spacing.
- The program here uses the standard input and output, but can be used to process actual data files like this:

```
cat datafile | onespace.ex > datafile.cln
```

- Observe how we process the file by lines
- The `continue` instruction means "skip the rest of the body of this loop and proceed directly to the evaluation of loop continuation".

EXAMPLE PROGRAMS USING FILE IO

```
1 // examples/numsort.cc
2 #include <iostream>
3 #include <string>
4 #include <fstream>
5 #include <filesystem>
6 #include <vector>
7 #include <sstream>
8
9 namespace fs = std::filesystem;
10 auto as_lines(fs::path file) ->
11     std::vector<std::string>
12 {
13     std::ifstream fin{ file };
14     std::string line;
15     std::vector<std::string> lines;
16     while (getline(fin, line))
17         lines.push_back(line);
18     return lines;
19 }
20 auto main(int argc, char* argv[]) -> int
21 {
```

```
22     if (argc != 2) {
23         std::cerr << "Usage:\n"
24             << argv[0] << " filename\n";
25         return 1;
26     }
27     auto content = as_lines(argv[1]);
28     std::sort(content.begin(), content.end(),
29         [](auto l1, auto l2) {
30             std::istringstream istr1{ l1 };
31             std::istringstream istr2{ l2 };
32             auto x1{0.}, x2{0.};
33             istr1 >> x1;
34             istr2 >> x2;
35             return x1 < x2;
36         }
37     );
38     for (std::string_view line : content) {
39         std::cout << line << "\n";
40     }
41 }
```

Numerically sort an input file.

```
1  #include <what is necessary>
2  auto main() -> int
3  {
4      const std::vector v{1, 2, 3, 4, 5};
5      const auto tot{0};
6      for (const auto el : v) tot += el;
7      std::cout << tot << "\n";
8  }
```

Which of the following is true ?

- A. `v` can not be a `const` as we are looping through its contents
- B. `tot` can not be a `const` as we are adding to it in the loop
- C. `el` can not be a `const` as it is obviously meant to change through the sequence
- D. All of the above

Exercise 1.10:

What is the largest number in the Fibonacci sequence which can be represented as a 64 bit integer? How many numbers of the sequence can be represented in 64 bits or less? Write a C++ program to find out. Start from `examples/fibonacci.cc`, and insert your code where indicated.

Exercise 1.11:

Work through the notebooks `Functions.ipynb` and `BlocksScopesNamespaces.ipynb` and ask any topic that you find unclear and needs an explanation.