PROGRAMMING IN C++

Jülich Supercomputing Centre

8 – 12 May 2023 | Sandipan Mohanty | Forschungszentrum Jülich, Germany

Chapter 1

Introduction

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

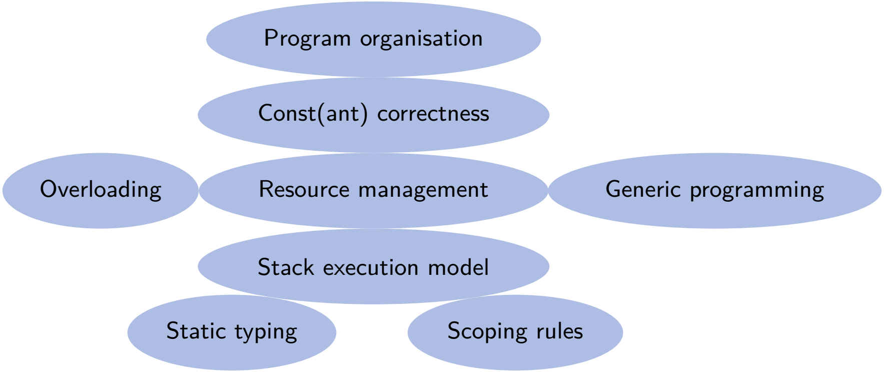
C++ provides ...

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

C++

- General purpose: no specialisation to specific usage areas
- No mandatory simplification that precludes a direct expert level use of hardware resources
- Leave no room for a lower level language
- You don't pay for features you don't use
- 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”


Program organisation
Const(ant) correctness
Overloading
Resource management
Generic programming
Stack execution model
Static typing
Scoping rules

LEARNING C++ IN 2023

- "Why, when there are easier, safer languages around?"

LEARNING C++ IN 2023

- “Why, when there are easier, safer languages around?”
- Isn't AI going to write all our programs from now on?

File Edit View History Bookmarks Tools Help

Destructor not call ×

https://chat.openai.com/ 160% Search

Arch Linux Forums · Most Visited · Default Crap · Personal · Jülich · Computing · Research · teaching · Manuals · News · tmp · Homepages · Talks · QC · C++ · BBB@FZJ · Jupyter-JSC · Compiler Explorer · Corona-Update: Todes... · Other Bookmarks

+ New chat

Today

Destructor not called o

Upgrade to Plus NEW

sandipan.mohanty@tuta...

SA

Show me an example of a C++ code where the destructor of a block scope automatic variable is not called when the variable goes out of scope.

In C++, the destructor of an automatic variable within a block scope is automatically called when the variable goes out of scope. However, there are certain scenarios where the destructor may not be called. One such scenario is when an exception is thrown before the variable goes out of scope. Here's an example:

cpp Copy code

```cpp
#include <iostream>

class Example {
public:
    Example() {
        std::cout << "Constructor called" << std::endl;
    }

    ~Example() {
        std::cout << "Destructor called" << std::endl;
    }
};

void foo() {
    Example obj;  // Automa
```

Regenerate response

Send a message.

Free Research Preview. ChatGPT may produce inaccurate information about people, places, or facts. ChatGPT Mar 23 Version

File Edit View History Bookmarks Tools Help

Destructor not call ×

https://chat.openai.com/ 160% Search

Arch Linux Forums Most Visited Default Crap Personal Jülich Computing Research teaching Manuals News tmp Homepages Talks QC C++ BBB@FZJ Jupyter-JSC Compiler Explorer Corona-Update: Todes... Other Bookmarks

+ New chat

Today

Destructor not called o

Upgrade to Plus NEW

sandipan.mohanty@tuta...

cpp Copy code

```cpp
#include <iostream>

class Example {
public:
    Example() {
        std::cout << "Constructor called" << std::endl;
    }

    ~Example() {
        std::cout << "Destructor called" << std::endl;
    }
};

void foo() {
    Example obj;  // Automatic variable

    throw std::runtime_error("Exception occurred");  // Exception thrown bef
}

int main() {
    try {
        foo();
    } catch (const std::runtime_error& e) {
        std::cout << e.what() << std::endl;
    }
```

Regenerate response

Send a message.

Free Research Preview. ChatGPT may produce inaccurate information about people, places, or facts. ChatGPT Mar 23 Version

+ New chat

Today

Destructor not called o

```
void foo() {
    Example obj;  // Automatic variable

    throw std::runtime_error("Exception occurred");  // Exception thrown bef
}

int main() {
    try {
        foo();
    } catch (const std::runtime_error& e) {
        std::cout << e.what() << std::endl;
    }

    return 0;
}
```

In the above code, the `Example` class has a constructor and a destructor. The `foo()` function throws a `std::runtime_error` exception before the `obj` variable goes out of scope. As a result, the destructor of `obj` is not called, and you won't see the "Destructor called" message in the output. Instead, the exception is caught in the `main()` function.

Please note that this is just one example where the destructor of an automatic variable may not be called. In general, it is best to rely on destructors to clean up resources and ensure proper cleanup even in the presence of exceptions.

Regenerate response

Send a message.

Upgrade to Plus NEW

sandipan.mohanty@tuta... ···

File Edit View History Bookmarks Tools Help

Destructor not call ×

https://chat.openai.com/ 160% Search

Arch Linux Forums · Most Visited · Default Crap · Personal · Jülich · Computing · Research · teaching · Manuals · News · tmp · Homepages · Talks · QC · C++ · BBB@FZJ · Jupyter-JSC · Compiler Explorer · Corona-Update: Todes... · Other Bookmarks

+ New chat

Today

Destructor not called o

Upgrade to Plus NEW

sandipan.mohanty@tuta...

SA

What compiler did you use to compile and run this code?

As an AI language model, I don't have direct access to a specific compiler to compile and run code. However, the code provided is standard C++ code and should work with any modern C++ compiler.

You can use popular C++ compilers such as:

- GCC (GNU Compiler Collection): https://gcc.gnu.org/
- Clang: https://clang.llvm.org/
- Microsoft Visual C++: https://visualstudio.microsoft.com/vs/features/cplusplus/

These compilers are widely used and support the C++ language standard.

To compile and run the code, you can save it into a file with a `.cpp` extension (e.g., `example.cpp`), and then use the appropriate compiler command to compile it. For example, with GCC, you can use the following command in the terminal:

Copy code

```
g++ example.cpp -o example
```

Regenerate response

Send a message.

Free Research Preview. ChatGPT may produce inaccurate information about people, places, or facts. ChatGPT Mar 23 Version

File Edit View History Bookmarks Tools Help

Destructor not call ×

https://chat.openai.com/ 160% Search

Arch Linux Forums Most Visited Default Crap Personal Jülich Computing Research teaching Manuals News tmp Homepages Talks QC C++ BBB@FZJ Jupyter-JSC Compiler Explorer Corona-Update: Todes... Other Bookmarks

+ New chat

Today

Destructor not called o

SA

None of those compilers produce the effect you claimed. The "Destructor called" message is printed before the "Exception occurred" message, in every case.

You are correct, and I apologize for the mistake in my previous response. The destructor of an object with automatic storage duration is guaranteed to be called when it goes out of scope, regardless of whether an exception is thrown or not.

Here's the corrected code:

cpp Copy code

```cpp
#include <iostream>

class Example {
public:
    Example() {
        std::cout << "Constructor called" << std::endl;
    }

    ~Example() {
        std::cout << "Destructor called" << std::endl;
    }
};

void foo() {
```

Regenerate response

Send a message.

Upgrade to Plus NEW

SA sandipan.mohanty@tuta...

Free Research Preview. ChatGPT may produce inaccurate information about people, places, or facts. ChatGPT Mar 23 Version

+ New chat
Today
Destructor not called o

```
void foo() {
    Example obj;  // Automatic variable

    throw std::runtime_error("Exception occurred");  // Exception thrown bef
}

int main() {
    try {
        foo();
    } catch (const std::runtime_error& e) {
        std::cout << e.what() << std::endl;
    }

    return 0;
}
```

When you compile and run this code, you will see that the "Destructor called" message is indeed printed before the "Exception occurred" message. The output will be:

sql Copy code

```sql
Constructor called
Destructor called
Exception occurred
```

Regenerate response

Send a message.

Upgrade to Plus NEW
sandipan.mohanty@tuta...

File Edit View History Bookmarks Tools Help

Destructor not call ×

https://chat.openai.com/ 160% Search

Arch Linux Forums Most Visited Default Crap Personal Jülich Computing Research teaching Manuals News tmp Homepages Talks QC C++ BBB@FZJ Jupyter-JSC Compiler Explorer Corona-Update: Todes... Other Bookmarks

+ New chat

Today

Destructor not called o

```
}

int main() {
    try {
        foo();
    } catch (const std::runtime_error& e) {
        std::cout << e.what() << std::endl;
    }

    return 0;
}
```

When you compile and run this code, you will see that the "Destructor called" message is indeed printed before the "Exception occurred" message. The output will be:

sql Copy code

```sql
Constructor called
Destructor called
Exception occurred
```

I apologize for the confusion caused by my previous response. Thank you for bringing it to my attention, and I appreciate your understanding.

Regenerate response

Upgrade to Plus NEW

sandipan.mohanty@tuta... ···

Send a message.

Free Research Preview. ChatGPT may produce inaccurate information about people, places, or facts. ChatGPT Mar 23 Version

USING AI LANGUAGE MODELS AS A PART OF LEARNING

- Many answers are good, which is very impressive! And it is getting better.

USING AI LANGUAGE MODELS AS A PART OF LEARNING

- Many answers are good, which is very impressive! And it is getting better.
- Many are also plain wrong. Not surprising, considering that the internet is full of bad (code) examples and demonstrably false but religiously held convictions (sometimes about C++!). And the AI trains on that. Unfortunately, even nonsensical answers are delivered with good grammar and authoritative language.

USING AI LANGUAGE MODELS AS A PART OF LEARNING

- Many answers are good, which is very impressive! And it is getting better.
- Many are also plain wrong. Not surprising, considering that the internet is full of bad (code) examples and demonstrably false but religiously held convictions (sometimes about C++!). And the AI trains on that. Unfortunately, even nonsensical answers are delivered with good grammar and authoritative language.
- An experienced developer can, perhaps, use something like `chatgpt` as a glorified auto-complete plugin. When the AI goes astray, the developer can ignore the suggestion from the language engine and fix the code.

USING AI LANGUAGE MODELS AS A PART OF LEARNING

- Many answers are good, which is very impressive! And it is getting better.
- Many are also plain wrong. Not surprising, considering that the internet is full of bad (code) examples and demonstrably false but religiously held convictions (sometimes about C++!). And the AI trains on that. Unfortunately, even nonsensical answers are delivered with good grammar and authoritative language.
- An experienced developer can, perhaps, use something like `chatgpt` as a glorified auto-complete plugin. When the AI goes astray, the developer can ignore the suggestion from the language engine and fix the code.
- When learning, you need to steer clear, and develop your foundations yourself.

LEARNING C++

- It takes time.
- It takes caution!
- There is an astonishing amount of misinformation about the language on the internet, some outdated, some were never true...
- Strong foundations in the building blocks of the language, will help you filter out incorrect information.

LEARNING C++

- It takes time.
- It takes caution!
- There is an astonishing amount of misinformation about the language on the internet, some outdated, some were never true...
- Strong foundations in the building blocks of the language, will help you filter out incorrect information.
- Scientific method: conduct your own "experiments" to test ideas

LEARNING C++

- It takes time.
- It takes caution!
- There is an astonishing amount of misinformation about the language on the internet, some outdated, some were never true...
- Strong foundations in the building blocks of the language, will help you filter out incorrect information.
- Scientific method: conduct your own "experiments" to test ideas
- Self-study over a much longer period
- Collaborative projects with good senior programmers
- Curiosity about an evolving language

LEARNING C++

- It takes time.
- It takes caution!
- There is an astonishing amount of misinformation about the language on the internet, some outdated, some were never true...
- Strong foundations in the building blocks of the language, will help you filter out incorrect information.
- Scientific method: conduct your own "experiments" to test ideas
- 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.
- It takes caution!
- There is an astonishing amount of misinformation about the language on the internet, some outdated, some were never true...
- Strong foundations in the building blocks of the language, will help you filter out incorrect information.
- Scientific method: conduct your own "experiments" to test ideas
- 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 2023

- 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.
- Some C++20 features are still not satisfactorily implemented, but available implementations are adequate for learning and testing
- Microsoft's MSVC compiler is currently the compiler with more implemented C++20 features than any other compiler

```
xarray<double> rt
    = load_csv<double>(fin, '\t');

rt -= mean(rt, 0.);

xarray<double> cross =
    linalg::dot(transpose(rt), rt);

auto [lambda, v] = linalg::eig(cross);
```

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

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

C++ IN MAY 2023

- 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.
- Some C++20 features are still not satisfactorily implemented, but available implementations are adequate for learning and testing
- Microsoft's MSVC compiler is currently the compiler with more implemented C++20 features than any other compiler

```cpp
using namespace std::chrono;
using Date = year_month_day;

year Y { asked_year.value_or(current_year())};

Date s4 { Y / December / Sunday[4] };
Date s3 { Y / December / Sunday[3] };
Date xmas { Y / December / 25d };
Date lastadv { s4 >= xmas ? s3 : s4 };
```

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

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

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 held the edge in providing access to the latest language features. For C++20, their support in open source compilers is still patchy, although steadily improving. It's usually better to use as new a version as possible
- Since version 11.x, GCC 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 any case, there is usually an option to explicitly specify the standard we want to use with a command line option, such as `-std=c++17` or `-std=c++20`.

COURSE CONTENT

- Language fundamentals

which means...

- Basic structure of a program
- Types, values and variables
- Mutability controls
- Statements, blocks
- Branches, loops
- Exceptions and C++ control flow
- Functions and lambda expressions
- Scope

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

Fundamentals

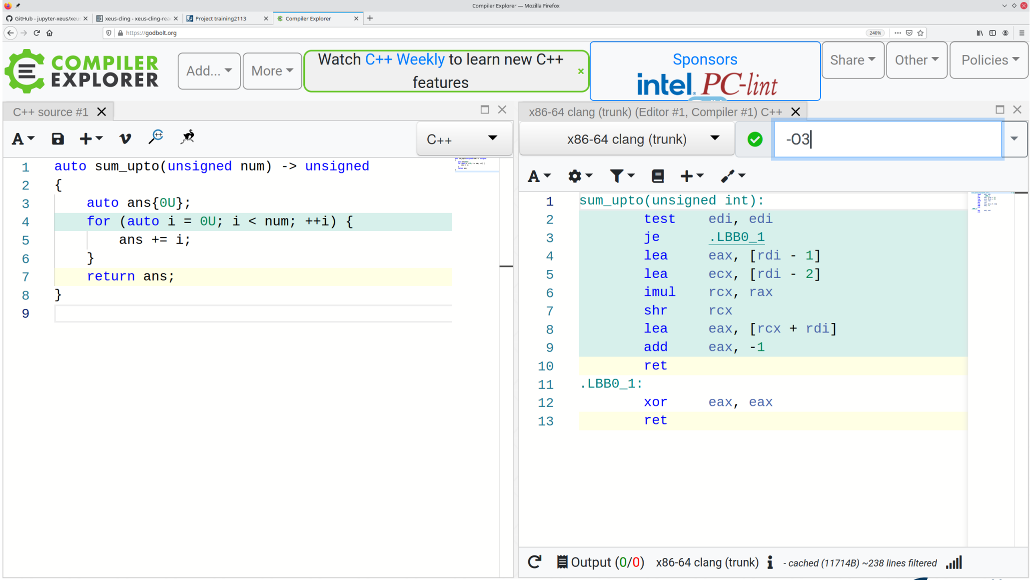
A COMPILED LANGUAGE

```cpp
// Hello World!
#include <iostream>
auto main() -> int
{
  std::cout<<"Hello, world!\n";
}
```

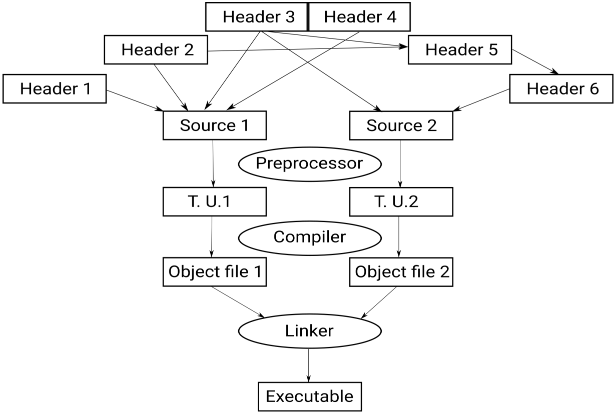
```
g++ -std=c++20 hello.cc
./a.out

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

```
cppint
>>> std::cout << "Hello, world!\n";
>>> quit
```

- Program: Step by step recipe for performing a computational task
- Expressed using precise deterministic rules in human readable programming languages
- Source code is translated to the machine language by the **compiler**
- The compiler enforces rules of the language
- Rules enable accurate expression of intent
- The 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.
- There are some “interpreters” to try out small bits of code, and one such interpreter (your shortcut: `cppint`) is installed in the class room computers.


Compiler Explorer — Mozilla Firefox
https://godbolt.org
COMPILER EXPLORER
Add...
More
Watch C++ Weekly to learn new C++ features
Sponsors
intel. PC-lint
Share
Other
Policies
C++ source #1
C++
auto sum_upto(unsigned num) -> unsigned
{
    auto ans{0U};
    for (auto i = 0U; i < num; ++i) {
        ans += i;
    }
    return ans;
}
x86-64 clang (trunk) (Editor #1, Compiler #1) C++
x86-64 clang (trunk)
-O3
sum_upto(unsigned int):
        test   edi, edi
        je   .LBB0_1
        lea   eax, [rdi - 1]
        lea   ecx, [rdi - 2]
        imul   rcx, rax
        shr   rcx
        lea   eax, [rcx + rdi]
        add   eax, -1
        ret
.LBB0_1:
        xor   eax, eax
        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.

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

Exercise 1.1:

Open the example `examples/hello_xyz.cc` in a text editor or IDE. Familiarise yourself with the process of compiling and running simple programs. Run this program with different command line options. Alternatively, open `http://coliru.stacked-crooked.com/`, copy and paste the above program and run it with some command line options!

THE MAIN() FUNCTION

- All C++ programs must contain a unique `main()` function
- All executed code, that is not related to the initialisation of a global variable, 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. This omission is only allowed for `main()`. Any other function promising to return something must have an appropriate `return` statement.

FUNCTION CALL TREE

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

```cpp
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);
}
```

```cpp
auto h11(int i)() -> int
{
    return i * i;
}
auto h21(int i)() -> int
{
    return i + h211(i);
}
```

```cpp
auto h211(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` function

CODE LEGIBILITY

```cpp
auto foo(double x, int i) -> double
{
double y=1;
if (i>0) {
for (int j=0;j<i;++j) {
y *= x;
}
} else if (i<0) {
for (int j=0;j>i;--j) {
y /= x;
}}
return y;
}
```

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

STYLE

```
auto foo(double x, int i) -> double
{
  double y = 1;
  if (i > 0) {
    for (int j = 0; j < i; ++j) {
      y *= x;
    }
  } else if (i < 0) {
    for (int j = 0; j > i; --j) {
      y /= x;
    }
  }
  return y;
}
```

- 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

```
auto foo(double x, int i) -> double
{
  double y = 1;
  if (i > 0) {
    for (int j = 0; j < i; ++j) {
      y *= x;
    }
  } else if (i < 0) {
    for (int j = 0; j > i; --j) {
      y /= x;
    }
  }
  return y;
}
```

- 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++

```cpp
// examples/hello_qa.cc
#include <string>
#include <iostream>

auto main() -> int
{
    std::string name;
    std::cout << "What's your name ? ";
    std::cin >> name;
    std::cout << "Hello, " << name << "\n";
}
```

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").

WRITE A VERY SIMPLE FUNCTION...

```
// examples/min_of_three.cc
#include <iostream>
auto min_of_three(int a, int b, int c) -> int
{
    // recipe needed!
    return a;
}
auto main() -> int
{
    int i = 0, j = 0, k = 0;
    std::cout << "Enter i, j and k: ";
    std::cin >> i >> j >> k;
    std::cout << "The smallest of the three is " << min_of_three(i, j, k) << "\n";
}
```

Exercise 1.3:

Fill in the code in `examples/min_of_three.cc` so that the function returns the smallest of the 3 input values.

WRITE A VERY SIMPLE FUNCTION...

```cpp
// examples/midpt.cc
#include <iostream>
auto mid(int a, int b) -> int
{
    // recipe needed!
    return a;
}
auto main() -> int
{
    int i = 0, j = 0;
    std::cout << "Enter i, j: ";
    std::cin >> i >> j;
    std::cout << "A number half way between " << i << " and " << j
              << " is " << mid(i, j) << "\n";
}
```

Exercise 1.4:

Fill in the code necessary in `examples/midpt.cc` so that the function returns a value half way between the two inputs, for small integers.

DATA TYPES

New Tab | Split View | Copy | Paste | Find | Edit Current Profile

build : bash | modules : bash | fig : bash | examples : binforms.g | Book : bash | () courseroom

() courseroom

```
cxx@zam2706: cppint
>>> std::cout << 13 / 2 << "\n";
6
>>> std::cout << 13.0 / 2.0 << "\n";
6.5
>>> std::cout << 13.0 / 2 << "\n";
6.5
>>> std::cout << 13 / 2. << "\n";
6.5
>>>
```

() courseroom

```
cxx@zam2706: python3
Python 3.6.15 (default, Sep 23 2021, 15:41:43) [GCC] on linux
Type "help", "copyright", "credits" or "license" for more informa
tion.
>>> 13 / 2
6.5
>>> 13 // 2
6
>>>
```

- Since 13 and 2 are integers, `13 / 2` means *integer division* in C++. `13 % 2` would return the remainder of the integer division
- Arithmetic operations between values of identical built in types produce the same type of output
- The meaning of operations on values depends on the *type* of the values

DATA TYPES

- A digital computer stores and processes information in binary bits
- Bit representation of even the simplest entities like integers or floating point numbers is a matter of convention; compare

```
int   i = 1; // 00000000000000000000000000000001
float i = 1; // 00111111100000000000000000000000
```

- Semantic meanings associated with a collection of bits is not inherent to the bits, but is imparted by the type associated with those bits
- Small differences in the text representation of numbers like `1` or `1.` might translate to much bigger differences for the processor

TYPES, VARIABLES AND DECLARATIONS

```cpp
auto force(double m1, double m2, double r12)
    -> double
{
  const auto G{ 6.67408e-11 };
  return G * m1 * m2 / (r12 * r12);
}
```

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

New Tab Split View Copy Paste Find Edit Current Profile

build : bash | modules : bash | fig : bash | examples : binforms.g | Book : bash | () courseroom

() courseroom

```
cxx@zam2706: cppint
>>> int k = 1;
>>> std::cout << typeid(k).name() << "\n";
i
>>> double x = 4.3;
>>> std::cout << typeid(x).name() << "\n";
d
>>> k = x;
>>> std::cout << k << "\n";
4
>>> std::cout << typeid(k).name() << "\n";
i
>>>
```

() courseroom

```
cxx@zam2706: python3
Python 3.6.15 (default, Sep 23 2021, 15:41:43) [GCC] on linux
Type "help", "copyright", "credits" or "license" for more informa
tion.
>>> k = 1
>>> type(k)
<class 'int'>
>>> x = 4.3
>>> type(x)
<class 'float'>
>>> k = x
>>> print(k)
4.3
>>> type(k)
<class 'float'>
>>>
```

New Tab | Split View | Copy | Paste | Find | Edit Current Profile

build : bash | modules : bash | fig : bash | examples : binforms.g | Book : bash | () courseroom

() courseroom

```
cxx@zam2706: cppint
>>> $#include <valarray>$
>>> std::valarray A{1, 2, 3, 4, 5};
>>> for (auto x: A) std::cout << x << "\n";
1
2
3
4
5
>>> A = 2;
>>> for (auto x: A) std::cout << x << "\n";
2
2
2
2
2
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>> 
```

() courseroom

```
cxx@zam2706: python3
Python 3.6.15 (default, Sep 23 2021, 15:41:43) [GCC] on linux
Type "help", "copyright", "credits" or "license" for more informa
tion.
>>> import numpy as np
>>> A = np.array([1, 2, 3, 4, 5])
>>> for x in A:
...   print(x)
...
1
2
3
4
5
>>> A = 2
>>> for x in A:
...   print(x)
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not iterable
>>> 
```

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 "initialised" 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

INITIALISATION

- Both `int i = 23` and `int i{ 23 }` are valid initialisation
- 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:

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

- In rare cases, initialisation 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

INITIALISATION

- Both `int i = 23` and `int i{ 23 }` are valid initialisation
- 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:

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

- In rare cases, initialisation 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

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

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

- The keyword `auto` can be used to declare a variable as `auto x{initialiser}` or `auto x = initialiser`. The variable is then created with the type and value of the `initialiser`.
- 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

```cpp
auto age = 7;
auto pi = 3.141592653589793;
auto energy = 0;
auto city = "Barcelona";
```

- What are the types of the variables declared here?

USING LITERALS WITH PRECISE TYPES

```cpp
auto age = 7;
auto pi = 3.141592653589793;
auto energy = 0;
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

```cpp
auto age = 7U;
auto pi = 3.141592653589793;
auto energy = 0.;
using namespace std::string_literals;
auto city = "Barcelona"s;
auto bigpositive = 0UL;
auto fort_real = 0.0F;
// With proper user defined functions
auto T1 = 300_Kelvin;
auto T2 = 100_Celcius;
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`?
- Writing literals with precise types is a good habit, i.e., `0.` rather than `0` if you mean a floating point value, `0U` rather than `0` if you mean an unsigned value...

USING LITERALS WITH PRECISE TYPES

```cpp
auto age = 7U;
auto pi = 3.141592653589793;
auto energy = 0.;
using namespace std::string_literals;
auto city = "Barcelona"s;
auto bigpositive = 0UL;
auto fort_real = 0.0F;
// With proper user defined functions
auto T1 = 300_Kelvin;
auto T2 = 100_Celcius;
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`?
- Writing literals with precise types is a good habit, i.e., `0.` rather than `0` if you mean a floating point value, `0U` rather than `0` if you mean an unsigned value...
- C++ allows you to make new literals for user defined types

C++ STANDARD LIBRARY STRINGS

```
#include <string>
std::string fullname;
std::string name{"Albert"};
using namespace std::string_literals;
auto surname{"Einstein"s};
//Concatenation and assignment
fullname = name + " " + surname;

//Comparison
if (name == "Godzilla") run();

std::cout << fullname << '\n';

for (size_t i = 0; i < fullname.size(); ++i) {
  if (fullname[i] > 'j') blah += fullname[i];
}
std::cout << "Substring after last z is "
          << name.substr(
               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

JÜLICH Forschungszentrum

CONVERTING TO AND FROM STRINGS

```
int i{10}, j{20};
std::string s{ std::to_string(i + j) }; // s: "30"
std::string t{ std::to_string(i) + std::to_string(j) }; // t: "1020"
tot += std::stod(line); // String-to-double
```

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

Exercise 1.5:

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

STD::STRING_VIEW

```cpp
std::string_view viewse{"Norrsken"};
using namespace std::string_view_literals;
auto viewen{"Northern lights"sv};

auto proc(std::string_view inp) -> bool
{
    if (inp.ends_with("et")) {
        if (inp.substr(0UL, 3UL) == blah)
        // ...
    }
}
```

- Lightweight entity similar to `std::string`. Does not own its content.
- "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
- 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

```cpp
// 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 triple quote strings in Python
- Very useful with regular expressions
- Starts with `R"(` and ends with `)"`
- More general form `R"delim(` text `)delim"`

Exercise 1.6:

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.

```cpp
{ // begin : block 0
    auto i = 0;
    while (i >= 0) { // begin : block 1
        // calc with i
        { // begin : block 2
            auto x = cos(i * pi/180);
            auto y = sin(i * pi/180);
            // more
        } // end : block 2

    } // end : block 1
} // end : block 0
```

SCOPE OF VARIABLE NAMES

```
auto find_root() -> double
{
  for (int i = 0; i < N; ++i) {
  //counter i defined only in this "for" loop.
  }
  double newval = 0; // This is ok.
  for (int i = 0; i < N; ++i) {
  // The counter i here is a different entity
    if (newval < 5) {
        string fl{"small.dat"};
        // do something
    }
    newval=...;
    cout << fl << '\n'; // Error!
  }
  int fl = 42; // ok, but shadowed below
  if (auto fl = filename; val < 5) { // C++17
    // fl is available here
  } else {
    // fl is also available here
  }
}
```

- A variable declaration creates a variable
- The scope of a variable is the lines of code where a variable can be accessed (unless shadowed)
- 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

SCOPE OF VARIABLE NAMES

```cpp
void example()
{
    std::string moon{"Titan"};
    std::string name = moon;
    std::cout << name;
    {
        std::cout << name;
        int name{10};
        name = name - 3;
        std::cout << name;
    }
    std::cout << name ;
}
```

- A variable declaration creates a variable
- The scope of a variable is the lines of code where a variable can be accessed (unless shadowed)
- 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.
- A nested child block may define a new variable with a name already in use. The new variable is then said to "shadow" the existing one. The visibility of the outer variable can then be discontinuous.

SCOPE OF VARIABLE NAMES

```cpp
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
```

- 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
- 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 below that line can change this deduction, unlike the meaning of the symbol `fact` in line 16 of the `python` example here.

SCOPE OF VARIABLE NAMES

```python
1   # Python code (pyscope2.py). This is purposely
2   # badly written code to illustrate possible
3   # dangers of dynamic variable scope.
4   import sys
5   if __name__ == "__main__":
6       if len(sys.argv) > 1:
7           N = int(sys.argv[1])
8       else:
9           N = 5
10
11  def fact(n):
12      if n > 1:
13          return n * fact(n-1)
14      return 1
15  while N > 0:
16      print(fact(N))
17      if N % 4 == 0:
18          fact = N * (N - 1) / 2
19      N = N -1
```

- 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
- 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 below that line can change this deduction, unlike the meaning of the symbol `fact` in line 16 of the `python` example here.

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
```

- 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
- 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 below that line can change this deduction, unlike the meaning of the symbol `fact` in line 16 of the `python` example here.

- 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

```
auto G = 6.674e-11 ;
auto pi = 3.141592653589793 ;
auto m1 = 1.0e10, m2 = 1.0e4;
auto r = 10;
std::cout << "Force = "
    << -G * m1 * m2 / (r * r)
    << "\n"; // great!
G = G + 1;
std::cout << "Force = "
    << -G * m1 * m2 / (r * r)
    << "\n"; // wrong!

for (auto i = 0; i < 360; ++pi) {
    std::cout << sin(i * pi / 180);
}
```

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

CONSTANTS

```cpp
auto G = 6.674e-11;
auto pi = 3.141592653589793;
auto m1 = 1.0e10, m2 = 1.0e4;
auto r = 10;
std::cout << "Force = "
    << -G * m1 * m2 / (r * r)
    << "\n"; // great!
G = G + 1;
std::cout << "Force = "
    << -G * m1 * m2 / (r * r)
    << "\n"; // wrong!

for (auto i = 0; i < 360; ++pi) {
    std::cout << sin(i * pi / 180);
}
```

- 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

```cpp
auto const G = 6.674e-11 ;
auto const pi = 3.141592653589793 ;
auto m1 = 1.0e10, m2 = 1.0e4;
auto r = 10;
std::cout << "Force = "
    << -G * m1 * m2 / (r * r)
    << "\n"; // great!
G = G + 1;
std::cout << "Force = "
    << -G * m1 * m2 / (r * r)
    << "\n"; // wrong!

for (auto i = 0; i < 360; ++pi) {
    std::cout << sin(i * pi / 180);
}
```

- 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

```
auto const G = 6.674e-11 ;
auto const pi = 3.141592653589793 ;
auto m1 = 1.0e10, m2 = 1.0e4;
auto r = 10;
std::cout << "Force = "
    << -G * m1 * m2 / (r * r)
    << "\n"; // great!
G = G + 1; // compiler error!
std::cout << "Force = "
    << -G * m1 * m2 / (r * r)
    << "\n"; // wrong!

for (auto i = 0; i < 360; ++pi ) {
    // compiler error!
    std::cout << sin(i * pi / 180);
}
```

- 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 can not proceed without fixing these errors

CONSTANTS

```cpp
auto const G = 6.674e-11 ;
auto const pi = 3.141592653589793 ;
auto m1 = 1.0e10, m2 = 1.0e4;
auto r = 10;
std::cout << "Force = "
    << -G * m1 * m2 / (r * r)
    << "\n"; // great!
G = G + 1; // compiler error!
std::cout << "Force = "
    << -G * m1 * m2 / (r * r)
    << "\n"; // wrong!

for (auto i = 0; i < 360; ++pi ) {
    // compiler error!
    std::cout << sin(i * pi / 180);
}
```

- 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 can not proceed without fixing these 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

```cpp
auto ask_user() -> double
{
    double tmp{};
    std::cout << "Enter R0: ";
    std::cin >> tmp;
    return tmp;
}
void elsewhere()
{
    const auto r = ask_user(); // OK
    r = r * r; // Not OK
}
```

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

CONSTANTS

```cpp
auto ask_user() -> double
{
    double tmp{};
    std::cout << "Enter R0: ";
    std::cin >> tmp;
    return tmp;
}
void elsewhere()
{
    const auto r = ask_user(); // OK
    r = r * r; // Not OK
}
```

- `const` does not mean compile time constant. Just that the variable will not be changed post initialisation.
- When you declare a variable as `const`, you are making a promise to not change it after initialisation. The compiler holds you to that promise.

CONSTANTS

```cpp
constexpr auto G = 6.674e-11 ;
constexpr auto pi = 3.141592653589793 ;
auto m1 = 1.0e10, m2 = 1.0e4;
auto r = 10;
std::cout << "Force = "
    << -G * m1 * m2 / (r * r)
    << "\n"; // great!
G = G + 1; Compiler error
std::cout << "Force = "
    << -G * m1 * m2 / (r * r)
    << "\n"; // wrong!

for (auto i = 0; i < 360; ++pi) {
    // Compiler error!
    std::cout << sin(i * pi / 180);
}
```

- `const` does not mean compile time constant. Just that the variable will not be changed post initialisation.
- When you declare a variable as `const`, you are making a promise to not change it after initialisation. The compiler holds you to that promise.
- For variables known to be compile time constants, one could use `constexpr`

CONSTANTS

```
constexpr auto G = 6.674e-11 ;
constexpr auto pi = 3.141592653589793 ;
auto m1 = 1.0e10, m2 = 1.0e4;
auto r = 10;
std::cout << "Force = "
    << -G * m1 * m2 / (r * r)
    << "\n"; // great!
G = G + 1; Compiler error
std::cout << "Force = "
    << -G * m1 * m2 / (r * r)
    << "\n"; // wrong!

for (auto i = 0; i < 360; ++pi) {
    // Compiler error!
    std::cout << sin(i * pi / 180);
}
```

- `const` does not mean compile time constant. Just that the variable will not be changed post initialisation.
- When you declare a variable as `const`, you are making a promise to not change it after initialisation. The compiler holds you to that promise.
- 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

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, `condition ? value1 : value2`, as shown on line 18. The expression with the ternary operator has a value `value1` if the `condition` is true. Otherwise it has a value `value2` The two options `value1` and `value2` must have the same type.

LOOPS

```
for (initialisation; condition; increment) {
    // Loop body
}
for (int i = 0; i < N; ++i) s += a[i];
while (condition) {}
while (T > t0) {}
do {} while (condition);
do {
} while (ch == 'y');
for (variable : collection) {}
for (int i : {1,2,3}) f(i);
for (int i = 0; i < N; ++i) {
  if (a[i] < cutoff) s+=a[i];
  else break;
}
for (std::string s : names) {
  if (s.size() > 10) {
    longnames.push_back(s);
    continue;
  }
  // process other names
}
```

- 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

Exercise 1.7:

Write a program to print the command line arguments in the reverse order.

Exercise 1.8:

Write a function to check if a given number is a prime number. Fill in the relevant lines in `examples/check_prime.cc`.

Exercise 1.9:

Let x is a positive real number, and r its square root, i.e., $x = r^2$. For any number y between 1 and x, $z = \frac{x}{y}$ is another such number. z and y are on opposite sides of r. In fact, iterating $r_{i+1} = \frac{1}{2}(r_i + \frac{x}{r_i})$, for any starting r_0 between 1 and x, creates a series gradually approaching r. Use this to write your own function to calculate the square root of a real number! Verify the answer by using C++ standard library square root function, `std::sqrt`.

REFERENCES

```cpp
const auto x{5.0};
const double y{6.0};

// different entities with same initial values
auto x2{ x }; // Obs: x2 is not const!
double y2{ y };
```

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

REFERENCES

```cpp
const auto x{5.0};
const double y{6.0};

// different entities with same initial values
auto x2{ x }; // Obs: x2 is not const!
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

```cpp
const auto x{5.0};
const double y{6.0};

// different entities with same initial values
auto x2{ x }; // Obs: x2 is not const!
double y2{ y };

// additional references for the same object
const auto& xr{ x };
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

```cpp
const auto x{5.0};
const double y{6.0};

// different entities with same initial values
auto x2{ x }; // Obs: x2 is not const!
double y2{ y };

// additional references for the same object
const auto& xr{ x };
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

```cpp
const auto x{5.0};
const double y{6.0};

// different entities with same initial values
auto x2{ x }; // Obs: x2 is not const!
double y2{ y };

// additional references for the same object
const auto& xr{ x };
const double& yr{ y };
```

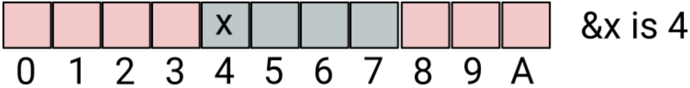
- Variable declaration: create object with initial value, and attach a name tag (reference) to it
- If a variable name is used to initialise a new one, `auto x2{x}`, the new variable will have the same value, but will be a different entity
- It is possible to "attach another name tag" to an existing 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" (entities allowed on the left side of an `=` sign) of type `double`

REFERENCES

```cpp
const auto x{5.0};
const double y{6.0};

// different entities with same initial values
auto x2{ x }; // Obs: x2 is not const!
double y2{ y };

// additional references for the same object
const auto& xr{ x };
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" (entities allowed on the left side of an `=` sign) of type `double`
- References are important for information exchange with functions

POINTERS



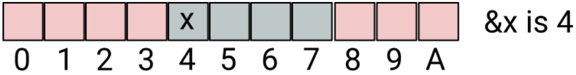

```cpp
int i{5};
int* iptr{&i}; // iptr points at i
i += 1;
std::cout << *iptr ; // 6
(*iptr) = 0;
std::cout << i ; // 0
int& iref{i}; // iref "refers" to i
iref = 4;
std::cout << i ; // 4
```

- 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`

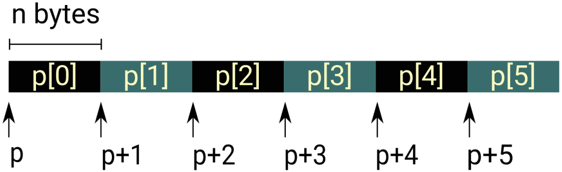
- If `iptr` is a pointer, `*iptr` is the object it is pointing at
- Adding 1 to the pointer `iptr` shifts it by `sizeof(typeofi)` bytes in memory
- A reference is effectively another name for the same object
- When in use, a reference appears as if it were a regular variable

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`, it's 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



JÜLICH Forschungszentrum

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]`

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

```cpp
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

```cpp
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

```cpp
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 1.10:

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 note books `Fundamentals_1.ipynb`, and `Fundamentals_2.ipynb`, before we continue. Ask any topic that you find unclear and needs an explanation.

FUNCTIONS

```cpp
auto function_name(parameters) -> return_type
{
    // function body
}
auto sin(double x) -> double
{
    // Somehow calculate sin of x
    return answer;
}
auto main() -> int
{
    constexpr double pi{3.141592653589793};
    for (int i = 0; i < 100; ++i) {
        std::cout << i * pi / 100
            << sin(i * pi / 100) << "\n";
    }
    std::cout << sin("pi") << "\n"; //Error!
}
```

- 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

```cpp
// Old syntax
bool pythag(int i, int j, int k); // prototype
int hola(int i, int j) // definition
{
    int ans{0};
    if (pythag(i,j,23)) {
        // A prototype or definition must be
        // visible in the translation unit
        // at the point of usage
        ans=42;
    }
    return ans;
}
// Definition of pythag. Not that old syntax
auto pythag(int i, int j, int k) -> bool
{
    // code
}
```

- 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

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

Exercise 1.11:

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.

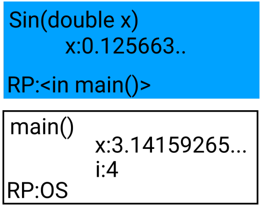
Exercise 1.12:

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




```cpp
auto sin(double x) -> int {
  // Somehow calculate sin of x
  return answer;
}
auto main() -> int {
    double x{3.141592653589793};
    for (int i = 0; i < 100; ++i) {
        std::cout << i * x / 100
            << sin(i * x / 100) <<"\n";
    }
}
```

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=<...>

```cpp
auto factorial(unsigned int n) -> unsigned int
{
  int u = n; // u: Unnecessary
  if (n > 1) return n * factorial(n - 1);
  else return 1;
}
auto someother() -> int
{
  factorial(4);
}
```

RECURSION

- SP=<in factorial()> n=4 u=4 RP=<9>
- SP=<in someother()> RP=<...>

```cpp
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=<in factorial()> n=3 u=3 RP=<4>
- SP=<in factorial()> n=4 u=4 RP=<9>
- 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 }
```

- 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=<in factorial()> n=2 u=2 RP=<4>
- SP=<in factorial()> n=3 u=3 RP=<4>
- SP=<in factorial()> n=4 u=4 RP=<9>
- SP=<in someother()> RP=<...>

```cpp
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=<in factorial()> n=1 u=1 RP=<4>
- SP=<in factorial()> n=2 u=2 RP=<4>
- SP=<in factorial()> n=3 u=3 RP=<4>
- SP=<in factorial()> n=4 u=4 RP=<9>
- 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 }
```

- 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=<in factorial()> n=2 u=2 RP=<4>
- SP=<in factorial()> n=3 u=3 RP=<4>
- SP=<in factorial()> n=4 u=4 RP=<9>
- SP=<in someother()> RP=<...>

```
auto factorial(unsigned int n) -> unsigned int
{
  int u = n; // u: Unnecessary
  if (n > 1) return n * factorial(n - 1);
  else return 1;
}
auto someother() -> int
{
  factorial(4);
}
```

- 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=<in factorial()> n=3 u=3 RP=<4>
- SP=<in factorial()> n=4 u=4 RP=<9>
- 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 }
```

- 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

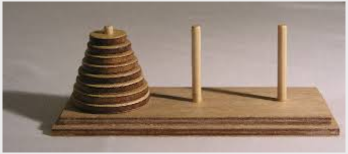
- SP=<in factorial()> n=4 u=4 RP=<9>
- SP=<in someother()> RP=<...>

```
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 }
```

- 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 1.13:

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

```
void somefunc()
{
  static int ncalls=0;
  ++ncalls;
  // code --> something unexpected
  std::cerr << "Encountered unexpected"
    << "situation in the " << ncalls
    << "th call to " << __func__ << "\n";
}
```

- 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

```cpp
auto power(int x, unsigned n) -> unsigned
{
    ans = 1;
    for (; n > 0; --n) ans *= x;
    return ans;
}
auto power(double x, double y) -> double
{
    return exp(y * log(x));
}
```

```cpp
auto someother(double mu, double alpha,
                    int rank) -> double
{
    double st=power(mu,alpha)*exp(-mu);

    if (n_on_bits(power(rank,5))<8)
        st=0;

    return st;
}
```

- 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 semantically 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

```
auto power(int x, unsigned n) -> unsigned
{
    ans = 1;
    for (; n > 0; --n) ans *= x;
    return ans;
}
auto power(double x, double y) -> double
{
    return exp(y * log(x));
}
```

```
auto someother(double mu, double alpha,
               int rank) -> double
{
    double st=power(mu,alpha)*exp(-mu);

    if (n_on_bits(power(rank,5))<8)
        st=0;

    return st;
}
```

- 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

```cpp
struct AtomId { int val = 0; };
struct MolId { int val = 0; };

void display_info(AtomId i)
{
    // show atom related info
}
void display_info(MolId i)
{
    // display completely different
    // stuff about molecule
}
void elsewhere()
{
    MolId j = select_a_molecule();
    for (AtomId i; i.val < ; ++i.val) {
        if (i == j) { // Compiler error!
            //
        }
    }
}
```

- `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

```cpp
struct AtomId { int val = 0; };
struct MolId { int val = 0; };

void display_info(AtomId i)
{
    // show atom related info
}
void display_info(MolId i)
{
    // display completely different
    // stuff about molecule
}
void elsewhere()
{
    MolId j = select_a_molecule();
    for (AtomId i; i.val < ; ++i.val) {
        if (i == j) { // Compiler error!
            //
        }
    }
}
```

- `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

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

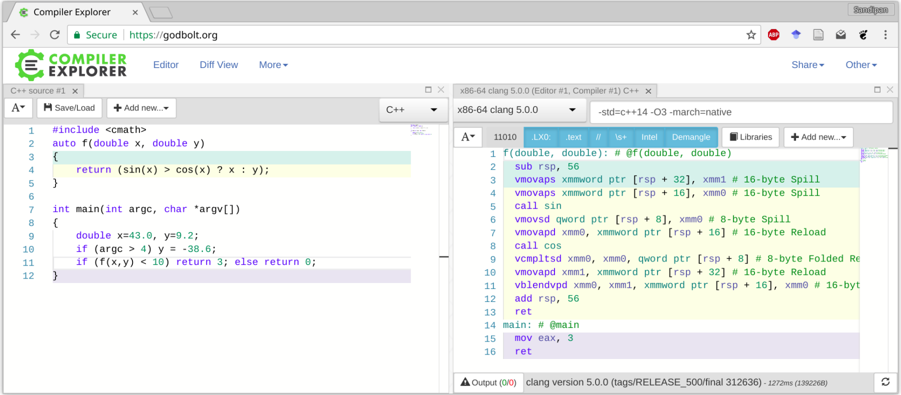
INLINE FUNCTIONS

```cpp
auto sqr(double x) -> double
{
  return x * x;
}
```

```cpp
inline auto sqr(double x) -> double
{
  return x * x;
}
```

- 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)

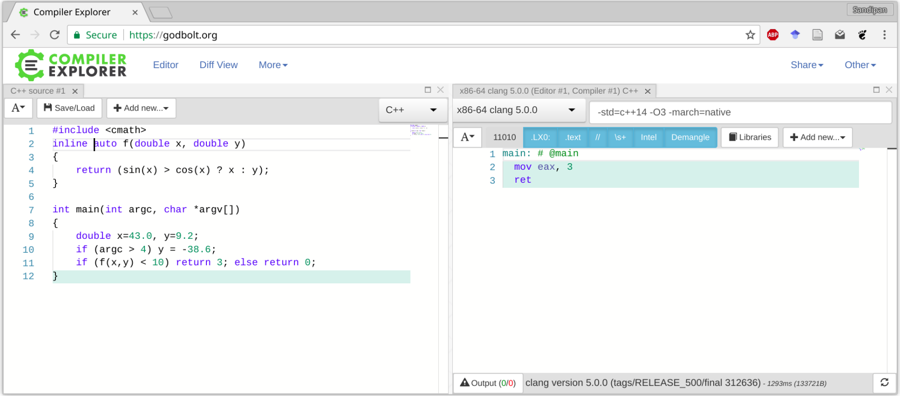
INLINE FUNCTIONS



- 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

JÜLICH Forschungszentrum

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 labelled 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. Here, instead of explicitly indicating the return type with, e.g., the `-> bool` notation, we let the compiler deduce the return type from the `return` statement(s) in the function.
- In case of multiple `return` statements, inconsistent return types will lead to a compiler error
- `decltype(auto)` can also be used in place of `auto` for this purpose, but that involves different type deduction rules. `decltype(auto)` infers a reference type when possible, whereas a simple `auto` infers a value type.

```cpp
auto greet(std::string nm)
{
    for (auto& c: nm) c = std::toupper(c);
    std::cout << nm << std::endl;
    return nm.size() > 10;
}
```

LAMBDA FUNCTIONS

```
auto onefunc(double inp) -> double
{
    auto x{ inp };
    // The following is not allowed.
    auto anotherfunc(double in) -> double
    {
        return in * in;
    }

    x = inp * anotherfunc(x);
    return x;
}
```

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

LAMBDA FUNCTIONS

```
auto onefunc(double inp) -> double
{
    auto x{ inp };

    auto anotherfunc = [](double in) -> double
    {
        return in * in;
    };

    x = inp * anotherfunc(x);
    return x;
}
```

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

LAMBDA FUNCTIONS

```
auto onefunc(double inp) -> double
{
    auto x{ inp };

    auto anotherfunc = [](double in) -> double
    {
        return in * in;
    };

    x = inp * anotherfunc(x);
    return x;
}
```

- 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

```
auto onefunc(double inp) -> double
{
    auto x{ inp };

    auto anotherfunc = [ x ](double in) -> double
    {
        return in * in * sin( x );
    };

    x = inp * anotherfunc(x);
    return x;
}
```

- 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

```cpp
constexpr auto cube(unsigned u)
{
    return u * u * u;
}
consteval auto cube2(unsigned u)
{
    return u * u * u;
}
void elsewhere(unsigned inp)
{
    std::array<int, cube(10) > A;
    constexpr auto myvar = cube(99U) ;
    auto myvar2 = cube(inp) ;

    std::array<int, cube2(10) > B;
    constexpr auto myvar = cube2(99U) ;
    auto myvar2 = cube2(inp) ;
}
```

- 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

```cpp
// Somewhere in the header iostream
namespace std {
    ostream cout;
}
// In your program ...
#include <iostream>
auto main() -> int
{
    {
        using namespace std;
        cout << __func__ << "\n";
    }
    int cout = 0;
    for (cout=0; cout<5; ++cout)
        std::cout << "Counter = " << cout << '\n';
        // Above, plain cout is an integer,
        // but std::cout is an output stream
        // The syntax to refer to a name
        // defined inside a namespace is:
        // namespace_name::identifier_name
}
```

- 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

```
// examples/namespaces.cc
#include <iostream>
using namespace std;
namespace UnitedKingdom {
  string London{"Big city"};
  void load_slang() {...}
}
namespace UnitedStates {
  string London{"Small town in Kentucky"};
  void load_slang() {...}
}
auto main() -> int
{
  using namespace UnitedKingdom;
  cout << London << '\n';
  cout << UnitedStates::London << '\n';
}
```

- 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

```cpp
//examples/namespaces2.cc
#include <iostream>
namespace UnitedKingdom {
  std::string London{"Big city"};
}
namespace UnitedStates {
  namespace KY {
    std::string London{" in Kentucky"};
  }
  namespace OH {
    std::string London{" in Ohio"};
  }
}
// With C++17 ...
namespace mylibrary::onefeature {
    auto solve(int i) -> double;
}
auto main() -> int
{
  namespace USOH=UnitedStates::OH;
  std::cout << "London is "
            << USOH::London <<'\n';
}
```

- `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

```
enum colour { red, green, blue };
// ...
colour c{green};
// ...
switch (c) {
    case red : do_stuff1(); break;
    case green : do_stuff2(); break;
    case blue:
    default: do_stuff3();
};
```

- 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 ..."

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

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.

```
enum class colour { red, green, blue };
enum class traffic_light {
    red, yellow, green
};
bool should_brake(traffic_light c);

if (should_brake(blue)) apply_brakes();
//Syntax error!
if (state == traffic_light::yellow) ...;

auto respond(traffic_light L) {
    using enum traffic_light;
    switch (L) {
        case red: {
        //...
        }

```


JÜLICH Forschungszentrum

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`

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

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

```cpp
  double x,y,z;
  int i;
  std::string s;
  fin >> x >> y >> z >> i >> s;
  fout << x << y << z << i << s << '\n';
```


JÜLICH Forschungszentrum

STRING STREAMS

```cpp
auto report(float x) -> std::string
{
  auto a = f(x);
  auto b = g(x);
  // We need the output to be
  // a string, perhaps to be
  // processed further elsewhere.
  std::ostringstream ost;
  ost << "f(x) returned " << a <<"\n";
  ost << "g(x) returned " << b <<"\n";
  return ost.str();
}
```

- `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

```
std::ifstream fin{"somefile.dat"};
double x;
while (fin >> x) {
  // while it is possible to read a new
  // value for x, do something.
}
std::string line;
while (getline(fin, line)) {
  // while it is possible to read a
  // line of input, do something
}
ifstream fin{ argv[1] };
for (auto it = istream_iterator<int>(fin);
          it != istream_iterator<int>{};
          ++it) {
  std::cout << "Token : " << *it << "\n";
}
```

- 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 1.14: Strings and I/O

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

EXAMPLE PROGRAMS USING FILE IO

```cpp
// examples/onespace.cc
#include <iostream>
auto main(int argc, char* argv[]) -> int
{
    std::string line;
    while (getline(std::cin, line)) {
        if (line.empty()) continue;
        bool sp{true};
        for (auto c : line) {
            if (isspace(c)) {
                if (not sp) std::cout << '\t';
                sp = true;
            } else {
                sp = false;
                std::cout << c;
            }
        }
        std::cout << "\n";
    }
}
```

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

```cpp
// examples/numsort.cc
#include <iostream>
#include <string>
#include <fstream>
#include <filesystem>
#include <vector>
#include <sstream>

namespace fs = std::filesystem;
auto as_lines(fs::path file) ->
    std::vector<std::string>
{
    std::ifstream fin{ file };
    std::string line;
    std::vector<std::string> lines;
    while (getline(fin, line))
        lines.push_back(line);
    return lines;
}
auto main(int argc, char* argv[]) -> int
{
    if (argc != 2) {
        std::cerr << "Usage:\n"
                  << argv[0] << " filename\n";
        return 1;
    }
    auto content = as_lines(argv[1]);
    std::sort(content.begin(), content.end(),
        [](auto l1, auto l2) {
            std::istringstream istr1{ l1 };
            std::istringstream istr2{ l2 };
            auto x1{0.}, x2{0.};
            istr1 >> x1;
            istr2 >> x2;
            return x1 < x2;
        }
    );
    for (std::string_view line : content) {
        std::cout << line << "\n";
    }
}
```

Numerically sort an input file.

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

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 1.15:

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 1.16:

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

Function call stack

```
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 + h211(i);
}
```

```
auto h211(int i)
    -> int
{
    return -i;
}
```

```
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 + h211(i);
}
```

```
auto h211(int i)
    -> int
{
    return -i;
}
```

main()

```
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 + h211(i);
}
```

```
auto h211(int i)
    -> int
{
    return -i;
}
```

main()

f() int i=10

```
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 + h211(i);
}
```

```
auto h211(int i)
    -> int
{
    return -i;
}
```

main()

```
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 + h211(i);
}
```

```
auto h211(int i)
    -> int
{
    return -i;
}
```

main()

g() int i = 10

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

```cpp
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);
}
```

```cpp
auto h11(int i)() -> int
{
    return i * i;
}
auto h21(int i)() -> int
{
    return i + h211(i);
}
```

```cpp
auto h211(int i)
    -> int
{
    return -i;
}
```

main()

```
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 + h211(i);
}
```

```
auto h211(int i)
    -> int
{
    return -i;
}
```

main()

h1() int i = 10

```
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 + h211(i);
}
```

```
auto h211(int i)
    -> int
{
    return -i;
}
```

main()

h1() int i = 10

h11() int i = 10

```
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 + h211(i);
}
```

```
auto h211(int i)
    -> int
{
    return -i;
}
```

main()

h1() int i = 10

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

```cpp
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);
}
```

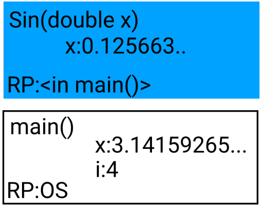
```cpp
auto h11(int i)() -> int
{
    return i * i;
}
auto h21(int i)() -> int
{
    return i + h211(i);
}
```

```cpp
auto h211(int i)
    -> int
{
    return -i;
}
```

main()

JÜLICH Forschungszentrum

FUNCTIONS AT RUN TIME




```cpp
auto sin(double x) -> int {
  // Somehow calculate sin of x
  return answer;
}
auto main() -> int {
    double x{3.141592653589793};
    for (int i = 0; i < 100; ++i) {
        std::cout << i * x / 100
            << sin(i * x / 100) <<"\n";
    }
}
```

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
- The memory used for the stack frames is usually cached and can be accessed quickly

FUNCTION ARGUMENTS

```
int x{ 1 };
int y{ x };

y = y + 1;
// What is x now?
```

- Recall the difference between creating a new variable and creating a reference to an existing object

FUNCTION ARGUMENTS

```
int x{ 1 };
int& y{ x };

y = y + 1;
// What is x now?
```

- Recall the difference between creating a new variable and creating a reference to an existing object

FUNCTION ARGUMENTS

```
auto f(int x) -> int
{
    x = x + 1;
    return x;
}
void elsewhere()
{
    int z{ 0 };
    f(z);
    // what is z now?
}
```

- Recall the difference between creating a new variable and creating a reference to an existing object
- For a function `f` declared as `ret_type f(type1 x, type2 y, type3 z)` when we call it using an expression like `f(value1, value2, value3)`, we perform the following initialisations on the stack frame for `x`, `y`, `z`: `type1 x{value1}`, `type2 y{value2}`, `type3 z{value3}`.

FUNCTION ARGUMENTS

```cpp
auto f(int& x) -> int
{
    x = x + 1;
    return x;
}
void elsewhere()
{
    int z{ 0 };
    f(z);
    // what is z now?
}
```

- Recall the difference between creating a new variable and creating a reference to an existing object
- For a function `f` declared as `ret_type f(type1 x, type2 y, type3 z)` when we call it using an expression like `f(value1, value2, value3)`, we perform the following initialisations on the stack frame for `x`, `y`, `z`: `type1 x{value1}`, `type2 y{value2}`, `type3 z{value3}`.
- Think about what information we are transmitting to the function, and how it might affect the behaviour of outside variables used when calling the function, based on whether `type1`, `type2` etc. are value or reference types.

```cpp
void get_lims(int i, int j)
{
    i = 10;
    j = 20;
}
auto main() -> int
{
    auto i = 2, j = 3;
    get_lims(i,j);
    std::cout << i << ", " << j << "\n";
}
```

What does the `std::cout` line print ?

A. `2, 3`

B. `10, 20`

C. `0, 0`

D. `3, 2`

```cpp
void get_lims(int& i, int& j)
{
    i = 10;
    j = 20;
}
auto main() -> int
{
    auto i = 2, j = 3;
    get_lims(i,j);
    std::cout << i << ", " << j << "\n";
}
```

What does the `std::cout` line print ?

A. `2, 3`

B. `10, 20`

C. `0, 0`

D. `3, 2`

THE REFERENCE TYPE IN FUNCTION PARAMETERS

```cpp
// Argument passed by value
auto find_arsenic_tolerance(Rat R)
    -> double
{
  double qnty = 0, dqnty = 1.0e-5;
  while (not R.dead()) {
      R.inject(dqnty);
      qnty += dqnty;
  }
  return qnty;
}
...
auto lab() -> int
{
  Rat r;
  double t = find_arsenic_tolerance(r);
  // r is still alive! But we know
  // how much arsenic it can take.
}
```

Pass a normal type by value

- The function `find_arsenic_tolerance` needs, as the argument, an object of type `Rat`.
- So you send a copy or clone of `r`
- The clone gets injections and is eventually destroyed.

THE REFERENCE TYPE IN FUNCTION PARAMETERS

```
// Argument passing by reference
auto find_arsenic_tolerance(Rat& R)
    -> double
{
  double qnty = 0, dqnty = 1.0e-5;
  while (not R.dead()) {
      R.inject(dqnty);
      qnty += dqnty;
  }
  return qnty;
}
...
auto lab() -> int
{
  Rat r;
  double t = find_arsenic_tolerance(r);
  // r is no more!
}
```

Pass a reference argument

- The function `find_arsenic_tolerance` needs, as the argument, an object of type `Rat &`, i.e., a reference to which Rat.
- So you send a copy of the Id tag on `r` to the function.
- The function acts on the `Rat` object which was referenced.

THE REFERENCE TYPE IN FUNCTION PARAMETERS

```cpp
// Argument passing by reference
auto find_arsenic_tolerance(Rat& R)
    -> double
{
  double qnty = 0, dqnty = 1.0e-5;
  while (not R.dead()) {
      R.inject(dqnty);
      qnty += dqnty;
  }
  return qnty;
}
...
auto lab() -> int
{
  Rat r;
  double t = find_arsenic_tolerance(r);
  // r is no more!
}
```

Pass a reference argument

- The function `find_arsenic_tolerance` needs, as the argument, an object of type `Rat &`, i.e., a reference to which Rat.
- So you send a copy of the Id tag on `r` to the function.
- The function acts on the `Rat` object which was referenced.

Information about the original rat, but the rat was modified.

THE REFERENCE TYPE IN FUNCTION PARAMETERS

We want to change an object

- When we want our object to be modified in some way by a function, it is no good to pass only a copy.
- In this example, a clone of the wounded leg will be bandaged

```
void bandage_leg(Leg l)
{
  //Select right bandage
  //Wrap bandage around l
}
...
auto main() -> int
{
  Human h;
  ...
  // h got a wounded left leg
  bandage_leg(h.left_leg());
  //No benefits to h.
}
```

THE REFERENCE TYPE IN FUNCTION PARAMETERS

We want to change an object

- Modifying a copy of our object is useless
- But a copy of a reference is good enough.
- In this example, the function works on the leg that was referred to.

```
void bandage_leg(Leg & l)
{
  //Select right bandage
  //Wrap bandage around l
}
...
auto main() -> int
{
  Human h;
  ...
  // h got a wounded left leg
  bandage_leg(h.left_leg());
  //Intended benefits to h
}
```

THE REFERENCE TYPE IN FUNCTION PARAMETERS

We want to change an object

- Modifying a copy of our object is useless
- But a copy of a reference is good enough.
- In this example, the function works on the leg that was referred to.

```
void bandage_leg(Leg & l)
{
  //Select right bandage
  //Wrap bandage around l
}
...
auto main() -> int
{
  Human h;
  ...
  // h got a wounded left leg
  bandage_leg(h.left_leg());
  //Intended benefits to h
}
```

We can use a function working with a reference when we want it to change our original object.

THE REFERENCE TYPE IN FUNCTION PARAMETERS

Cloning is expensive

- Sometimes, the data structures are very large, and copying them is expensive
- Functions taking that kind of classes will implicitly perform big cloning operations, slowing the program down.

```
auto count_bad_tires(Truck t) -> int
{
  int n = 0;
  for (int i = 0; i < t.n_wheels(); ++i) {
      if (not t.wheel(i).good()) ++n;
  }
  return n;
}
...
auto main() -> int
{
  Truck mytruck;
  ...
  nbad = count_bad_tires(mytruck);
  // Unnecessary cloning of mytruck
}
```

THE REFERENCE TYPE IN FUNCTION PARAMETERS

Cloning is expensive

- If the function signature asks for a reference, we only create a reference to the truck when invoking the function
- The same effect can be achieved by a pointer, but the syntax with references is cleaner

```cpp
auto count_bad_tires(Truck& t) -> int
{
  int n = 0;
  for (int i = 0; i < t.n_wheels(); ++i) {
      if (not t.wheel(i).good()) ++n;
  }
  return n;
}
...
auto main() -> int
{
  Truck mytruck;
  ...
  nbad = count_bad_tires(mytruck);
  // another reference to truck, not
  // clone of truck
}
```

THE CONSTANT REFERENCE TYPE

Cloning is expensive

- We want to use a reference as the argument because it is efficient
- How do we ensure that the original object would not be allowed to change ?

```
auto count_bad_tires(Truck& t) -> int
{
  int n = 0;
  for (int i = 0; i < t.n_wheels(); ++i) {
      check_pressure(t.wheel(i));
      if (not t.wheel(i).good()) ++n;
  }
  return n;
}
...
auto main() -> int
{
  Truck mytruck;
  ...
  nbad = count_bad_tires(mytruck);
  // Was there any change to mytruck ?
}
```

THE CONSTANT REFERENCE TYPE

Cloning is expensive

- We want to use a reference as the argument only because it is efficient
- How do we ensure that the original object would not be allowed to change ?
- Using a `const reference`

```cpp
auto count_bad_tires(const Truck& t) -> int
{
  int n = 0;
  for (int i = 0; i < t.n_wheels(); ++i) {
      check_pressure(t.wheel(i));
      if (not t.wheel(i).good()) ++n;
  }
  return n;
}
...
int main()
{
  Truck mytruck;
  ...
  nbad = count_bad_tires(mytruck);
  // Was there any change to mytruck ?
  // Not if this compiled!
}
```

Runtime error handling

RUN-TIME ERROR HANDLING

Exceptions: When there is nothing reasonable to return

```cpp
auto mysqrt(double x) -> double
{
    const auto eps2 = 1.0e-24;
    auto r0 = 0.5 * (1. + x);
    auto r1 = x / r0;
    while ((r0 - r1) * (r0 - r1) > eps2) {
        r0 = 0.5 * (r0 + r1);
        r1 = x / r0;
    }
    return r1;
}
```

Exceptions

- A function may be called with arguments which don't make sense
- An illegal mathematical operation
- Unexpected values, e.g., an arbitrary string when expecting a number
- Too much memory might have been requested

THROWING AND CATCHING EXCEPTIONS

```cpp
using error_code = int;
auto mysqrt(double x) -> double
{
    const auto eps = 1.0e-12;
    const auto eps2 = eps * eps;
    if (x < 0) throw error_code{-1};
    auto r0 = 0.5 * (1. + x);
    auto r1 = x / r0;
    while ((r0 - r1) * (r0 - r1) > eps2) {
        r0 = 0.5 * (r0 + r1);
        r1 = x / r0;
    }
    return r1;
}
```

```cpp
auto appl(double x, double y) -> double
{
    try {
        if (x < y) std::swap(x, y);
        return mysqrt(x + y) + mysqrt(x - y);
    } catch (error_code& error) {
        std::cout << "Caught error_code: "
                  << error << "\n";
        // somehow fix the situation and
        // return something sensible. If that
        // doesn't work...
        throw;
    }
}
```

THROWING AND CATCHING EXCEPTIONS

```cpp
using error_code = std::string;
auto mysqrt(double x) -> double
{
    using std::format;
    const auto eps = 1.0e-12;
    const auto eps2 = eps * eps;
    if (x < 0) throw
        format("Bad input {} for mysqrt", x);

    auto r0 = 0.5 * (1. + x);
    auto r1 = x / r0;
    while ((r0 - r1) * (r0 - r1) > eps2) {
        r0 = 0.5 * (r0 + r1);
        r1 = x / r0;
    }
    return r1;
}
```

```cpp
auto appl(double x, double y) -> double
{
    try {
        if (x < y) std::swap(x, y);
        return mysqrt(x + y) + mysqrt(x - y);
    } catch (error_code& error) {
        std::cout << "Caught error_code: "
                  << error << "\n";
        // somehow fix the situation and
        // return something sensible. If that
        // doesn't work...
        throw;
    }
}
```

THROWING AND CATCHING EXCEPTIONS

```cpp
auto mysqrt(double x) -> double
{
    using std::format;
    const auto eps = 1.0e-12;
    const auto eps2 = eps * eps;
    if (x < 0) throw
      std::runtime_error{
        format("Bad input {} for mysqrt", x)
      };

    auto r0 = 0.5 * (1. + x);
    auto r1 = x / r0;
    while ((r0 - r1) * (r0 - r1) > eps2) {
        r0 = 0.5 * (r0 + r1);
        r1 = x / r0;
    }
    return r1;
}
```

```cpp
auto appl(double x, double y) -> double
{
    try {
        if (x < y) std::swap(x, y);
        return mysqrt(x + y) + mysqrt(x - y);
    } catch (std::runtime_error& error) {
        std::cout << "Caught runtime error: "
                  << error.what() << "\n";
        // somehow fix the situation and
        // return something sensible. If that
        // doesn't work...
        throw;
    }
}
```

TRY AND CATCH BLOCKS

```
void f() {
  try {
  // lines
    try {
      // a line throwing an exception
    } catch (exception_type_0& err) {
      // handle errors of type 0
    }
  // more lines
  } catch (exception_type_1& err) {
  // handle errors of type 1
  } catch (exception_type_2& err) {
  // ...
  }
}
void g(int i) { if (i > -3) f(); }
auto main(int argc, char* argv[]) -> int {
    try {
      g(argc)
    } catch (exception_type_3& err) {
    // handle error type 3
    }
}
```

- Exceptions are monitored and handled in `try..catch` blocks
- When an exception is thrown in the `try` part of a `try..catch` block, the attached `catch` blocks are checked for a handler matching the **type** of the thrown exception.
- If no matching handler is found, we look for the next bigger `try..catch` block surrounding the previous one
- If an exception is thrown in an area inside a function, not inside a `try` section, the enclosing `try` section is searched based on the *call site* for the call
- This search can unwind till it reaches `main()` . If still no matching handler is found, the program exits with error.

TRY AND CATCH BLOCKS

```
void f() {
  try {
  // lines
    try {
      // a line throwing an exception
    } catch (exception_type_0& err) {
      // handle errors of type 0
    }
  // more lines
  } catch (exception_type_1& err) {
  // handle errors of type 1
  } catch (exception_type_2& err) {
  // ...
  }
}
void g(int i) { if (i > -3) f(); }
auto main(int argc, char* argv[]) -> int {
    try {
      g(argc)
    } catch (exception_type_3& err) {
    // handle error type 3
    }
}
```

- Once an exception is thrown, the program control flow enters a special mode
- Imagine all other lines, except `try..catch` blocks and the `throw` expression being "greyed out"
- In this view, the code looks like a smallish tree of `try..catch` blocks. Find the the smallest enclosing `try` block with an attached `catch` block of the matching type! Execution jumps to that `catch` block.
- The type matching and jump destinations can all be determined by the compiler
- This jump in program control still follows all the rules regarding variable scopes: when we leave a block of code by flying away on the back of an exception, **it still counts as leaving the block**. Automatic variables declared in that scope are therefore destroyed.

IS IT NEEDLESSLY EXPENSIVE TO USE EXCEPTIONS?

```cpp
auto f(double x, bool& succeeded) -> double
{
    const auto eps = 1.0e-12;
    const auto eps2 = eps * eps;
    if (x < 0) {
        succeeded = false;
    } else {
        auto r0 = 0.5 * (1. + x);
        auto r1 = x / r0;
        while ((r0 - r1) * (r0 - r1) > eps2) {
            r0 = 0.5 * (r0 + r1);
            r1 = x / r0;
        }
        succeeded = true;
    }
    return r1;
}
```

Contrast: how about we use additional function arguments to indicate success or failure?

```cpp
auto appl(double x, double y) -> double
{
    if (x < y) std::swap(x, y);
    bool ep{false}, em{false};
    auto rp = f(x + y, ep);
    auto rm = f(x - y, em);
    if (ep and em) {
        return rp + rm; // normal case
    } else {
        // handle errors
    }
}
```

- Cumbersome because of extra flag variables
- A value is returned even in the case of failure. A programmer can accidentally or out of carelessness, ignore the success/error flags. The subsequent calculations will be incorrect.

JÜLICH Forschungszentrum

IS IT NEEDLESSLY EXPENSIVE TO USE EXCEPTIONS?

```cpp
auto f(double x) -> double
{
    const auto eps = 1.0e-12;
    const auto eps2 = eps * eps;
    if (x < 0)
        throw std::runtime_error{
          format("Bad input {} for square root!", x)
        };
    auto r0 = 0.5 * (1. + x);
    auto r1 = x / r0;
    while ((r0 - r1) * (r0 - r1) > eps2) {
        r0 = 0.5 * (r0 + r1);
        r1 = x / r0;
    }
    return r1;
}
```

An error handling method with functionality comparable to exceptions will have a similar cost!

```cpp
auto appl(double x, double y) -> double
{
    if (x < y) std::swap(x, y);
    try {
        return f(x + y) + f(x - y);
    } catch (std::runtime_error& err) {
      // handle errors
    }
}
```

- Normal, successful flow is separated from error handling code
- In case there is an error, it is impossible to ignore! The function does not return with a value. The only choices are to handle the error or to terminate the program.

NOEXCEPT

```
auto sum(unsigned i, unsigned j)
    -> unsigned {
    return i + j;
}
void contained(int i) {
    try {
    // some code
    } catch (ET_1& err) {
    } catch (ET_2& err) {
    } catch (...) {
    // handle every exception
    }
}
```

- Sometimes, we know that an exception will never escape certain functions

NOEXCEPT

```cpp
auto sum(unsigned i, unsigned j) noexcept
    -> unsigned {
    return i + j;
}
void contained(int i) noexcept {
    try {
    // some code
    } catch (ET_1& err) {
    } catch (ET_2& err) {
    } catch (...) {
    // handle every exception
    }
}
```

- Sometimes, we know that an exception will never escape certain functions
- Such functions can be decorated with the `noexcept` specifier to tell the compiler that it does not need to make arrangements about propagating exceptions out of those functions

NOEXCEPT

```cpp
auto sum(unsigned i, unsigned j) noexcept
    -> unsigned {
    return i + j;
}
void contained(int i) noexcept {
    try {
    // some code
    } catch (ET_1& err) {
    } catch (ET_2& err) {
    } catch (...) {
    // handle every exception
    }
}
```

- Sometimes, we know that an exception will never escape certain functions
- Such functions can be decorated with the `noexcept` specifier to tell the compiler that it does not need to make arrangements about propagating exceptions out of those functions
- By discarding some exception handling code, the compiler may in some cases generate better optimised code

NOEXCEPT

```cpp
auto sum(unsigned i, unsigned j) noexcept
    -> unsigned {
    return i + j;
}
void contained(int i) noexcept {
    try {
    // some code
    } catch (ET_1& err) {
    } catch (ET_2& err) {
    } catch (...) {
    // handle every exception
    }
}
```

- Sometimes, we know that an exception will never escape certain functions
- Such functions can be decorated with the `noexcept` specifier to tell the compiler that it does not need to make arrangements about propagating exceptions out of those functions
- By discarding some exception handling code, the compiler may in some cases generate better optimised code
- If you lie, and decorate a function with a `noexcept` badge, but an exception reaches the outer most block of the function, the program is `std::terminate`d.

OPTIONAL VALUES

```cpp
#include <optional>
auto f(double x) -> std::optional<double> {
    std::optional<double> ans;
    const auto eps2 = 1.0e-24;
    if (x >= 0) {
        auto r0 = 0.5 * (1. + x);
        auto r1 = x / r0;
        while ((r0 - r1) * (r0 - r1) > eps2) {
            r0 = 0.5 * (r0 + r1);
            r1 = x / r0;
        }
        ans = r1;
    }
    return ans;
}
// Elsewhere...
std::cout << "Enter number : ";
std::cin >> x;
if (auto r = f(x); r.has_value()) {
    std::cout << "The result is "
              << r.value() << '\n';
}
```

- `std::optional<T>` is analogous to a box containing exactly one object of type `T` or nothing at all
- If created without any initialisers, the box is empty
- You store something in the box by assigning to the `optional`
- Evaluating the optional as a boolean gives a `true` outcome if there is an object inside, irrespective of the value of that object
- Empty box evaluates to `false`

OPTIONAL VALUES

```cpp
#include <optional>
auto f(double x) -> std::optional<double> {
    std::optional<double> ans;
    const auto eps2 = 1.0e-24;
    if (x >= 0) {
        auto r0 = 0.5 * (1. + x);
        auto r1 = x / r0;
        while ((r0 - r1) * (r0 - r1) > eps2) {
            r0 = 0.5 * (r0 + r1);
            r1 = x / r0;
        }
        ans = r1;
    }
    return ans;
}
// Elsewhere...
std::cout << "Enter number : ";
std::cin >> x;
if (auto r = f(x); r) {
    std::cout << "The result is "
              << *r << '\n';
}
```

- `std::optional<T>` is analogous to a box containing exactly one object of type `T` or nothing at all
- If created without any initialisers, the box is empty
- You store something in the box by assigning to the `optional`
- Evaluating the optional as a boolean gives a `true` outcome if there is an object inside, irrespective of the value of that object
- Empty box evaluates to `false`

C++23 STD::EXPECTED

```cpp
#include <expected>
auto mysqrt(double x) -> std::expected<double, std::string> {
    const auto eps = 1.0e-12;
    const auto eps2 = eps * eps;
    if (x >= 0.) {
        auto r0 = 0.5 * (1. + x);
        auto r1 = x / r0;
        while ((r0 - r1) * (r0 - r1) > eps2) {
            r0 = 0.5 * (r0 + r1);
            r1 = x / r0;
        }
        return { r1 };
    } else {
        return std::unexpected { "Unexpected input!" };
    }
}
// Elsewhere...
if (auto rm = mysqrt(x); rm) std::cout << "Square root = " << rm.value() << "\n";
else std::cout << "Error: " << rm.error() << "\n";
```

- Similar to `std::optional`, but has more capacity to describe the error
- The *unexpected* value can be of a type of our choosing, making it very flexible

ASSERTIONS

```cpp
#include <cassert>
bool check_things()
{
    // false if something is wrong
    // true otherwise
}
double somewhere()
{
    // if I did everything right,
    // val should be non-negative
    assert(val >= 0);
    assert(check_things());
}
```

- `assert(condition)` aborts if `condition` is false
- Used for non-trivial checks in code during development. The errors we are trying to catch are logic errors in implementation.
- If the macro `NDEBUG` is defined before including `<cassert>` `assert(condition)` reduces to nothing

ASSERTIONS

```cpp
#include <cassert>
bool check_things()
{
    // false if something is wrong
    // true otherwise
}
double somewhere()
{
    // if I did everything right,
    // val should be non-negative
    assert(val >= 0);
    assert(check_things());
}
```

- After we are satisfied that the program is correctly implemented, we can pass `-DNDEBUG` to the compiler, and skip all assertions.

- `assert(condition)` aborts if `condition` is false
- Used for non-trivial checks in code during development. The errors we are trying to catch are logic errors in implementation.
- If the macro `NDEBUG` is defined before including `<cassert>` `assert(condition)` reduces to nothing

Exercise 1.17:

The program `examples/exception.cc` demonstrates the use of exceptions. Rewrite the loop so that the user is asked for a new value until a reasonable value for the function input parameter is given.

Exercise 1.18:

Handle invalid inputs in your `gcd.cc` program so that if we call it as `gcd apple orange` it quits with an understandable error message. Valid inputs should produce the result as before.

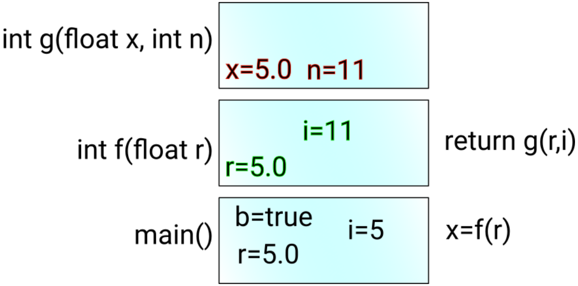
Exercise 1.19:

In the folder `examples/sqrt_error_handling`, you will find the solution to the square root exercise from the first day, with different error handling methods discussed here: exceptions, `std::optional` and `std::expected`. Study the code, experiment, ask for clarifications!

Dynamic memory management

HEAP VS STACK

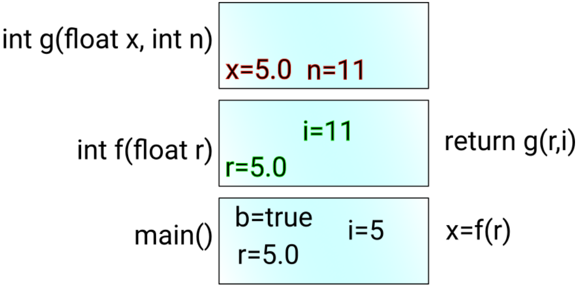
```cpp
auto f(double x) -> double
{
  int i = static_cast<int>( x );
  double M[1000][1000][1000]; // Oops!
  M[123][344][24] = x;
  return x - M[i][555][1];
}
auto main() -> int
{
  std::cout << f(5) << "\n";
  // Immediate SEGFAULT
}
```



- Variables in a function are allocated on the stack, but sometimes we need more space than what the stack permits

HEAP VS STACK

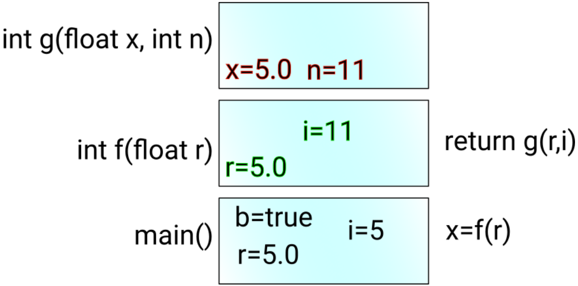
```cpp
auto f(double x) -> double
{
  int i = static_cast<int>( x );
  double M[1000][1000][1000]; // Oops!
  M[123][344][24] = x;
  return x - M[i][555][1];
}
auto main() -> int
{
  std::cout << f(5) << "\n";
  // Immediate SEGFAULT
}
```



- Variables in a function are allocated on the stack, but sometimes we need more space than what the stack permits
- We do not know how much space we should reserve for a variable (e.g. a `string`)

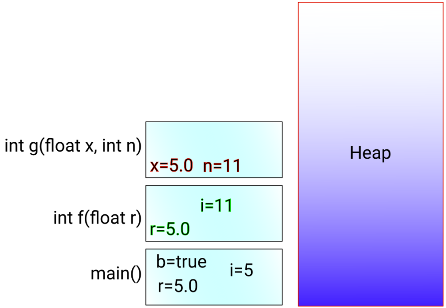
HEAP VS STACK

```cpp
auto f(double x) -> double
{
  int i = static_cast<int>( x );
  double M[1000][1000][1000]; // Oops!
  M[123][344][24] = x;
  return x - M[i][555][1];
}
auto main() -> int
{
  std::cout << f(5) << "\n";
  // Immediate SEGFAULT
}
```



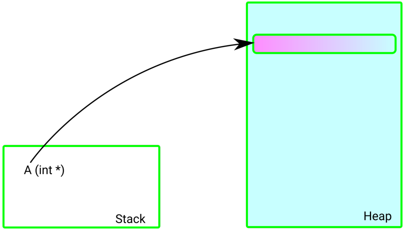
- Variables in a function are allocated on the stack, but sometimes we need more space than what the stack permits
- We do not know how much space we should reserve for a variable (e.g. a `string`)
- $\rightarrow$ We need a way to allocate from the "free store"

HEAP MEMORY



int g(float x, int n)
x=5.0 n=11
int f(float r)
i=11
r=5.0
main()
b=true
i=5
r=5.0
Heap

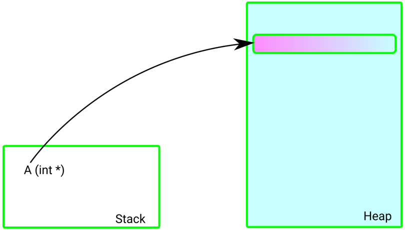
HEAP VS STACK




```
void f()
{
  int* A = new int[1000000];
  // use A
  delete [] A;
}
```

- `operator new`: Request that a specific amount of memory be reserved for you on the free store. The return value of the `new` operation is an address, which you store in a pointer (`A` here).

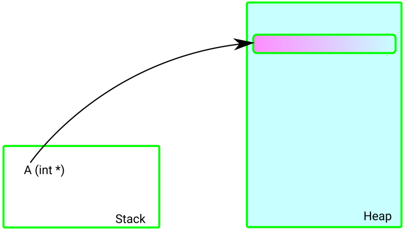
HEAP VS STACK




```cpp
void f()
{
  int* A = new int[1000000];
  // use A
  delete [] A;
}
```

- `operator new`: Request that a specific amount of memory be reserved for you on the free store. The return value of the `new` operation is an address, which you store in a pointer (`A` here).
- The pointer `A` is a normal variable on the stack. But it's value is the address of the allocated space

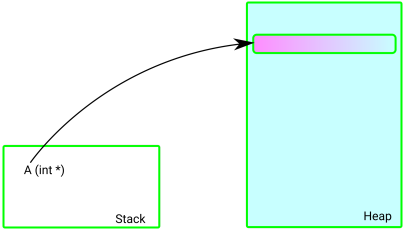
HEAP VS STACK




```
void f()
{
  int* A = new int[1000000];
  // use A
  delete [] A;
}
```

- `operator new`: Request that a specific amount of memory be reserved for you on the free store. The return value of the `new` operation is an address, which you store in a pointer (`A` here).
- The pointer `A` is a normal variable on the stack. But it's value is the address of the allocated space
- Memory allocated from the heap stays with your program until you free it, using `delete`

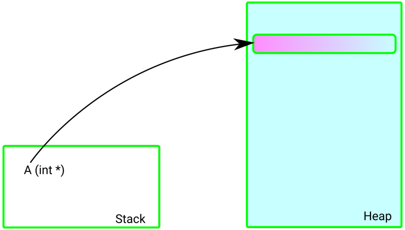
HEAP VS STACK




```
void f()
{
  int* A = new int[1000000];
  // use A
  delete [] A;
}
```

- `operator new`: Request that a specific amount of memory be reserved for you on the free store. The return value of the `new` operation is an address, which you store in a pointer (`A` here).
- The pointer `A` is a normal variable on the stack. But it's value is the address of the allocated space
- Memory allocated from the heap stays with your program until you free it, using `delete`
- The pointer we used to store its address is subject to scoping rules, and might expire at a certain `}`

HEAP VS STACK

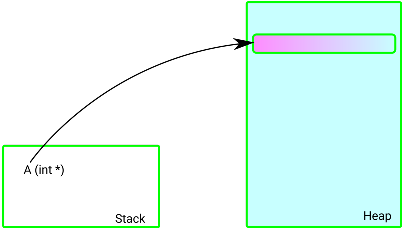



```
void f()
{
  int* A = new int[1000000];
  // use A
  delete [] A;
}
```

Note: Heap allocation and deallocation are slower than accessing stack memory.

- `operator new`: Request that a specific amount of memory be reserved for you on the free store. The return value of the `new` operation is an address, which you store in a pointer (`A` here).
- The pointer `A` is a normal variable on the stack. But it's value is the address of the allocated space
- Memory allocated from the heap stays with your program until you free it, using `delete`
- The pointer we used to store its address is subject to scoping rules, and might expire at a certain `}`
- Unless you ensure that `delete` is called before the pointer expires or that the address is stored elsewhere before that happens, you have a memory leak

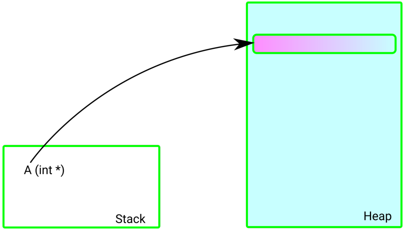
HEAP VS STACK




```
void f()
{
  int* A = new int[1000000];
  // use A
  g(A);
  //
  delete [] A;
}
```

- Allocations with `new` should be matched by corresponding `delete` operations

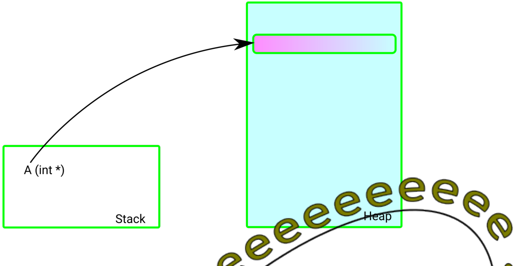
HEAP VS STACK




```
void f()
{
  int* A = new int[1000000];
  // use A
  g(A);
  //
  delete [] A;
}
```

- Allocations with `new` should be matched by corresponding `delete` operations
- But, what if we throw an exception before we reach `delete`?

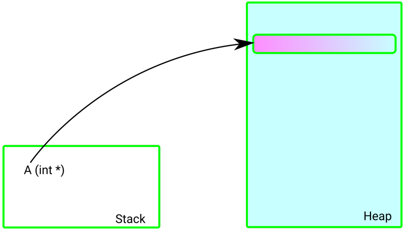
HEAP VS STACK




```
void f()
{
  int* A = new int[1000000];
  // use A
  g(A);
  //
  delete [] A;
}
```

- Allocations with `new` should be matched by corresponding `delete` operations
- But, what if we throw an exception before we reach `delete`?
- We may never reach the meticulously placed `delete` statement!

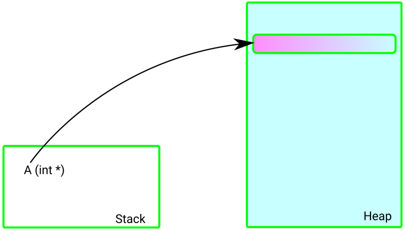
HEAP VS STACK




```
void f()
{
  int* A = new int[1000000];
  // use A
  g(A);
  //
  delete [] A;
}
```

- Allocations with `new` should be matched by corresponding `delete` operations
- But, what if we throw an exception before we reach `delete`?
- We may never reach the meticulously placed `delete` statement!
- Generally, because of exceptions, in C++ programs, placing clean-up operations near "exit sites" is not a sound strategy

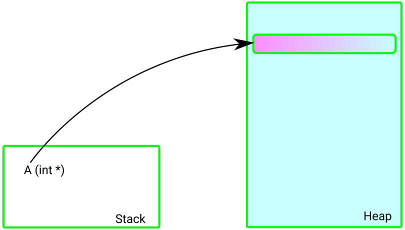
HEAP VS STACK




```
void f()
{
  int* A = new int[1000000];
  // use A
  g(A);
  //
  delete [] A;
}
```

- Allocations with `new` should be matched by corresponding `delete` operations
- But, what if we throw an exception before we reach `delete`?
- We may never reach the meticulously placed `delete` statement!
- Generally, because of exceptions, in C++ programs, placing clean-up operations near “exit sites” is not a sound strategy
- This also applies to other “resources” which must be acquired for use, e.g., threads, mutexes...

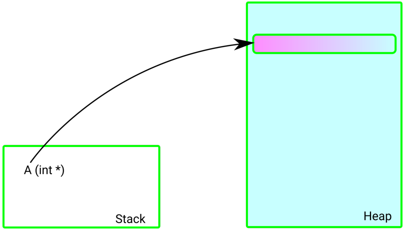
HEAP VS STACK




```
void f()
{
  int* A = new int[1000000];
  // use A
  g(A);
  //
  delete [] A;
}
```

- Allocations with `new` should be matched by corresponding `delete` operations
- But, what if we throw an exception before we reach `delete`?
- We may never reach the meticulously placed `delete` statement!
- Generally, because of exceptions, in C++ programs, placing clean-up operations near "exit sites" is not a sound strategy
- This also applies to other "resources" which must be acquired for use, e.g., threads, mutexes...
- The right way to handle that is by using RAII, as will be explained in our discussion of C++ classes.

HEAP VS STACK




```cpp
void f()
{
  int* A = new int[1000000];
  // use A
  g(A);
  //
  delete [] A;
}
```

- Allocations with `new` should be matched by corresponding `delete` operations
- But, what if we throw an exception before we reach `delete`?
- We may never reach the meticulously placed `delete` statement!
- Generally, because of exceptions, in C++ programs, placing clean-up operations near "exit sites" is not a sound strategy
- This also applies to other "resources" which must be acquired for use, e.g., threads, mutexes...
- The right way to handle that is by using RAII, as will be explained in our discussion of C++ classes.
- Standard library facilities use the same technique

OBJECT LIFETIME MANAGEMENT WITH SMART POINTERS

- 3 kinds of smart pointers were introduced in C++11: `unique_ptr`, `shared_ptr`, and `weak_ptr`

OBJECT LIFETIME MANAGEMENT WITH SMART POINTERS

- 3 kinds of smart pointers were introduced in C++11: `unique_ptr`, `shared_ptr`, and `weak_ptr`
- `unique_ptr` claims exclusive ownership of the allocated array. When it runs out of its scope, it calls delete on the allocated resource. It is impossible to "forget" to delete the memory owned by `unique_ptr`

OBJECT LIFETIME MANAGEMENT WITH SMART POINTERS

- 3 kinds of smart pointers were introduced in C++11: `unique_ptr`, `shared_ptr`, and `weak_ptr`
- `unique_ptr` claims exclusive ownership of the allocated array. When it runs out of its scope, it calls delete on the allocated resource. It is impossible to "forget" to delete the memory owned by `unique_ptr`
- Several instances of `shared_ptr` may refer to the same block of memory. When the last of them expires, it cleans up.

OBJECT LIFETIME MANAGEMENT WITH SMART POINTERS

- 3 kinds of smart pointers were introduced in C++11: `unique_ptr`, `shared_ptr`, and `weak_ptr`
- `unique_ptr` claims exclusive ownership of the allocated array. When it runs out of its scope, it calls delete on the allocated resource. It is impossible to "forget" to delete the memory owned by `unique_ptr`
- Several instances of `shared_ptr` may refer to the same block of memory. When the last of them expires, it cleans up.
- Helper functions `make_unique` and `make_shared` can be used to allocate on heap and retrieve a smart pointer to the allocated memory

OBJECT LIFETIME MANAGEMENT WITH SMART POINTERS

- 3 kinds of smart pointers were introduced in C++11: `unique_ptr`, `shared_ptr`, and `weak_ptr`
- `unique_ptr` claims exclusive ownership of the allocated array. When it runs out of its scope, it calls delete on the allocated resource. It is impossible to "forget" to delete the memory owned by `unique_ptr`
- Several instances of `shared_ptr` may refer to the same block of memory. When the last of them expires, it cleans up.
- Helper functions `make_unique` and `make_shared` can be used to allocate on heap and retrieve a smart pointer to the allocated memory
- Smart pointers are not special language provided entities with magical resource management abilities. They use RAII, and a user defined data structure can do the same kind of resource management by following the same programming techniques.

DYNAMIC MEMORY WITH SMART POINTERS

```
using big = std::array<int, 1000000>;
int f()
{
  auto u1 = std::make_unique<big>();
  // use u1
} // u1 expires, and frees the allocated memory
```

- Current recommendation: avoid free `new` / `delete` calls in normal user code
- Use them to implement memory management components
- Use `unique_ptr` and `shared_ptr` to manage resources
- If you do the above, you can then assume that an ordinary pointer in your code is a ”non-owning“ pointer, and let it expire without leaking memory

MEMORY ALLOCATION/DEALLOCATION

- You don't need it often:
  - `std::string` takes care of itself
  - Using standard library containers like `vector`, `list`, `map`, `deque` even rather complicated structures can be created without explicit memory allocation and de-allocation.
- When you nevertheless must (first choice):

```cpp
auto c = make_unique<complex_number>(1.2,4.2); // on the heap
int asize = 100; // on the stack
auto darray = make_unique<double[]>(asize);
// The stack frame contains the unique_ptr variables c and darray.
// The memory locations they point to on the other hand, are not
// on the stack, but on the heap. But, you don't need to worry about
// releasing that memory explicitly. If you don't have any way of
// accessing the resource (the pointers expire), the memory will be
// freed for you.
//
```

MEMORY ALLOCATION/DEALLOCATION

- You don't need it often:
  - `std::string` takes care of itself
  - Using standard library containers like `vector`, `list`, `map`, `deque` even rather complicated structures can be created without explicit memory allocation and de-allocation.
- When you nevertheless must (second choice):
  - Wrap the managed resource in a class
  - Allocate in constructors (or dedicated member functions invoked from constructors), where you handle all exceptions. If an exception is thrown and the initialisation of the newly allocated objects can not be completed fully, handle errors by restoring all variables to their original states. This kind of a function should either succeed in making the intended change, or not make any changes at all.
  - Use your allocating function in (other) constructors and member functions as needed
  - Clean up in the destructor, using `delete`, if you used `new`
  - Object lifetime rules will ensure the clean up, even in case of exceptions.

UNIQUE POINTER

```cpp
// examples/uniqueptr.cc
auto main() -> int
{
    auto u1 = std::make_unique<MyStruct>(1);
    //auto u2 = u1; //won't compile
    auto u3 = std::move(u1);
    std::cout << "Data value for u3 is u3->vl = " << u3->vl <<'\n';
    auto u4 = std::make_unique<MyStruct[]>(4);
}
```

- Smart pointer: The data pointed to is freed when the pointer expires
- Exclusive access to resource
- Can not be copied (deleted copy constructor and assignment operator)
- Data ownership can be transferred with `std::move`
- Can create single instances as well as arrays through `make_unique`

SHARED POINTER

```cpp
// examples/sharedptr.cc
auto main() -> int
{
    auto u1 = std::make_shared<MyStruct>(1);
    std::shared_ptr<MyStruct> u2 = u1; // Copy is ok
    std::shared_ptr<MyStruct> u3 = std::move(u1);
    std::cout << "Reference count of u3 is "
              << u3.use_count() << '\n';
}
```

- Smart pointer: The data pointed to is freed when the pointer expires
- Can share resource with other shared/weak pointers
- Can be copy assigned/constructed
- Maintains a reference count `ptr.use_count()`

WEAK POINTER

```cpp
// examples/weakptr.cc
auto main() -> int
{
    auto s1 = std::make_shared<MyStruct>(1);
    std::weak_ptr<MyStruct> w1(s1);
    std::cout << "Ref count of s1 = " << s1.use_count() << '\n';
    std::shared_ptr<MyStruct> s3(s1);
    std::cout << "Ref count of s1 = " << s1.use_count() << '\n';
}
```

- Does not own resource
- Can "kind of" share data with shared pointers, but does not change reference count

Exercise 1.20: uniqueptr.cc, sharedptr.cc

Read the 3 smart pointer example files, and try to understand the output. Observe when the constructors and destructors for the data objects are being called.

MEMORY MANAGEMENT ERRORS

```
auto somefunc(inputpars) -> outputtype
{
    auto* heapblock = new double[1024];

    // calculations
    // calculations
    // calculations

    return res;
    // Oops! Forgot to delete heapblock!
}
```

- Explicit handling of heap allocation/deallocation is error prone. Danger: memory leak.

MEMORY MANAGEMENT ERRORS

```cpp
auto somefunc(inputpars) -> outputtype
{
    auto* heapblock = new double[1024];

    // calculations
    // calculations
    // calculations

    delete [] heapblock;
    return res;
}
```

- Explicit handling of heap allocation/deallocation is error prone. Danger: memory leak.
- Must match `new` with `delete` in code

MEMORY MANAGEMENT ERRORS

```
auto somefunc(inputpars) -> outputtype
{
    auto* heapblock = new double[1024];

    // calculations
    // throw an exception!
    // calculations

    delete [] heapblock;
    return res;
}
```

- Explicit handling of heap allocation/deallocation is error prone. Danger: memory leak.
- Must match `new` with `delete` in code
- Even then, leak can happen: e.g., when the code never reaches the `delete`

MEMORY MANAGEMENT ERRORS

```cpp
auto somefunc(inputpars) -> outputtype
{
    auto heapblock =
        std::make_unique<double[]>(1024);

    // calculations
    // throw an exception!
    // => unique_ptr cleans up

    return res;
    // unique_ptr cleans up
}
```

- Explicit handling of heap allocation/deallocation is error prone. Danger: memory leak.
- Must match `new` with `delete` in code
- Even then, leak can happen: e.g., when the code never reaches the `delete`
- Use RAII for resource management instead.

MEMORY MANAGEMENT ERRORS

```
auto somefunc(inputpars) -> outputtype
{
    auto heapblock =
        std::make_unique<double[]>(1024);

    // calculations
    // throw an exception!
    // => unique_ptr cleans up

    return res;
    // unique_ptr cleans up
}
```

- Explicit handling of heap allocation/deallocation is error prone. Danger: memory leak.
- Must match `new` with `delete` in code
- Even then, leak can happen: e.g., when the code never reaches the `delete`
- Use RAII for resource management instead.
- Delegate explicit life time management of heap resources to smart pointers, e.g., `std::unique_ptr`

JÜLICH Forschungszentrum

DANGERS OF DANGLING POINTERS AND REFERENCES

```
{
    int* ptr = nullptr;
    if (something) {
        auto i = std::stoi(argv[1]);
        ptr = &i;
        std::cout << "ptr is pointing at "
            << *ptr << "\n";
    }
    // ptr still in scope, but i isn't!
    std::cout << *ptr << "\n";
    // dangling --> dereference -->
    // undefined behaviour!
}
```

- While consistent and correct use of RAII will avoid heap memory leak errors, other forms of memory errors exist, and are harder to eliminate

DANGERS OF DANGLING POINTERS AND REFERENCES

```cpp
{
    int* ptr = nullptr;
    if (something) {
        auto i = std::stoi(argv[1]);
        ptr = &i;
        std::cout << "ptr is pointing at "
            << *ptr << "\n";
    }
    // ptr still in scope, but i isn't!
    std::cout << *ptr << "\n";
    // dangling --> dereference -->
    // undefined behaviour!
}
```

- While consistent and correct use of RAII will avoid heap memory leak errors, other forms of memory errors exist, and are harder to eliminate
- When storing addresses in pointers, we have to ensure that the pointer is not used beyond the scope of the object it points at.

DANGERS OF DANGLING POINTERS AND REFERENCES

```cpp
auto calc(double inp) -> double&
{
    auto loc = inp * inp;
    // Returning ref to local:
    return loc; // Bad idea!
}
void elsewhere()
{
    auto&& res = calc(4);
    std::cout << res << "\n";
}
```

- While consistent and correct use of RAII will avoid heap memory leak errors, other forms of memory errors exist, and are harder to eliminate
- When storing addresses in pointers, we have to ensure that the pointer is not used beyond the scope of the object it points at.
- If we return a reference from a function, we must make sure, it is not a reference to a temporary object.

DANGERS OF DANGLING POINTERS AND REFERENCES

```cpp
{
    std::vector v{1, 2, 3};
    auto& vstart = v.front();
    v.push_back(4); // may invalidate vstart
    v.push_back(5);
    v.push_back(6);
    v.push_back(7);
    std::cout << vstart << "\n";
}
```

- While consistent and correct use of RAII will avoid heap memory leak errors, other forms of memory errors exist, and are harder to eliminate
- When storing addresses in pointers, we have to ensure that the pointer is not used beyond the scope of the object it points at.
- If we return a reference from a function, we must make sure, it is not a reference to a temporary object.
- If we store references to heap object, there is always the danger that operations on the owning entity will invalidate the reference

DANGLING -> DEREFERENCE -> UNDEFINED BEHAVIOUR

Exercise 1.21:

The folder `examples/dangling_pr` contains examples of the 3 kinds of memory bugs mentioned in this section. Study them, and check what, if any, errors or warnings the compiler generates for them. Try compiling with `-Wall Wextra`. Run them and examine the results. Try compiling with `-fsanitize=address`.

AVOID DANGLING POINTERS AND REFERENCES

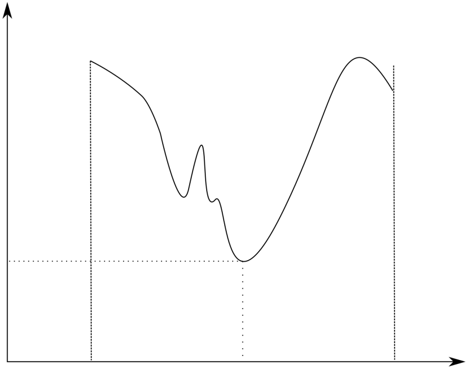
- Ensure that pointers and references do not outlive the referenced objects
- Prefer short lived non-owning pointers
- Do not return references to temporary objects
- Avoid storing references to objects on the heap

Chapter 2

C++ classes and class hierarchies

C++ classes

AD HOC STRUCTS



- Some times calculations involve bundles of entities which belong together, e.g., the location of a minimum of a function and the corresponding minimum value

AD HOC STRUCTS

```cpp
struct minim_return_type {
    double min_loc, min_val;
};
auto minimize(double r1, double r2,
              FunctionType f)
{
    minim_return_type m;
    // Find minimum somehow
    m.min_loc = the_location;
    m.min_val = the_value;
    return m;
}
void elsewhere()
{
    auto sol = minimize(0., 10., cost_func);
    cout << "Minimum found at " << sol.min_loc
         << "with a value " << sol.min_val
         << "\n";
}
```

- `struct` : Staple together objects of arbitrary types
- Can be done in global as well as block scope

AD HOC STRUCTS

```
struct minim_ret_type {
    double min_loc, min_val;
};
auto minimize(double r1, double r2,
              FunctionType f)
{
    minim_ret_type m;
    // Find minimum somehow
    m.min_loc = the_location;
    m.min_val = the_value;
    return m;
}
void elsewhere()
{
    auto [loc, val] = minimize(0., 10.,
                               cost_func);
    cout << "Minimum found at " << loc
         << "with a value " << val << "\n";
}
```

- `struct` : Staple together objects of arbitrary types
- Can be done in global as well as block scope
- We can now use the name of the `struct` to create variables , such that each of them has a `min_loc` *member* and a `min_val` *member*
- Can be function argument (and hence can participate in overload resolution), or return value (and hence gives us a way to return multiple values)
- Names of the components can be chosen to reflect any meanings associated with the content, making it a good practical way of returning multiple objects from a function

AD HOC STRUCTS

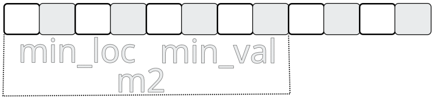
```cpp
struct minim_ret_type {
    double min_loc, min_val;
};
auto minimize(double r1, double r2,
              FunctionType f)
{
    minim_ret_type m;
    // Find minimum somehow
    m.min_loc = the_location;
    m.min_val = the_value;
    return m;
}
void elsewhere()
{
    auto [loc, val] = minimize(0., 10.,
                               cost_func);
    cout << "Minimum found at " << loc
         << "with a value " << val << "\n";
}
```

- `struct` : Staple together objects of arbitrary types
- Can be done in global as well as block scope
- We can now use the name of the `struct` to create variables , such that each of them has a `min_loc` *member* and a `min_val` *member*
- Can be function argument (and hence can participate in overload resolution), or return value (and hence gives us a way to return multiple values)
- Names of the components can be chosen to reflect any meanings associated with the content, making it a good practical way of returning multiple objects from a function
- Structured bindings can be used to create aliases for the components. The binding names are independent of the names in the `struct`

AD HOC STRUCTS

```cpp
struct minim_ret_type {
    double min_loc, min_val;
};
auto minimize(double r1, double r2,
              FunctionType f)
{
    minim_ret_type m;
    // Find minimum somehow
    m.min_loc = the_location;
    m.min_val = the_value;
    return m;
}
void elsewhere()
{
    auto m1 = minimize(-10., 0., constfunc1);
    minim_ret_type m2 = minimize(-10., 0.,
                                 constfunc1);
    auto * mptr = &m2;
    if ( m1.min_val > mptr->min_val )
        haha();
}
```

- A `struct` is a user defined data type
- Each *instance* has a bundle, with a `min_loc` and `min_val` member
- Members are accessed from the object using the `.` notation, and from a pointer to an object using the `->` notation. `(*mptr).min_val` is the same as `mptr->min_val`



DESIGNATED INITIALISERS

```
// examples/desig2.cc
struct v3 { double x, y, z; };
struct pars { int offset; v3 velocity; };
auto operator<<(std::ostream& os, const v3& v) -> std::ostream&
{
    return os << v.x << ", " << v.y << ", " << v.z << " ";
}
auto example_func(pars p)
{
    std::cout << p.offset << " with velocity " << p.velocity << "\n";
}
auto main() -> int
{
    example_func( {.offset = 5, .velocity = {.x=1., .y = 2., .z=3.}} );
}
```

- Simple struct type objects can be initialised by designated initialisers for each field.
- Can be used to implement a kind of "keyword arguments" for functions. But remember, at least as of C++20, the field order can not be shuffled.

C++ CLASSES

```
1  // examples/trivialclassoverload.cc
2  class A {};
3  class B {};
4  void func(int i, A a)
5  {
6      cout << "Called f input types (int, A)\n";
7  }
8  void func(int i, B b)
9  {
10     cout << "Called f input types (int, B)\n";
11 }
12 auto main() -> int
13 {
14     A xa;
15     B xb;
16     func(0, xa);
17     func(0, xb);
18 }
```

- User defined data types. Independently created classes are different, even if they have the same content.
- Function overloading: The two versions of the function `func` shown here are different entities from the compiler's viewpoint. No ambiguity about which function is called in lines 16 and 17 in `main()`.

C++ CLASSES

Overloading operators

```cpp
// examples/op_overload.cc
class A {};
class B {};
auto operator+(A x, A y) -> A
{
    std::cout << "operator+(A, A)\n";
    return x;
}
auto operator+(B x, B y) -> B
{
    std::cout << "operator+(B, B)\n";
    return x;
}
auto operator+(A x, B y) -> A {...} // similar
auto main() -> int {
    A a1, a2;
    B b1, b2;
    a1 + a2;
    a1 + b1;
    b1 + b2; // b1 + a2; doesn't work. Think why!
}
```

- For C++ class types, operators like `+`, `-`, `*`, `/`, `||`, `&&` ...are functions
- As long as at least one of the arguments to an operator is of a class type (not a built-in type like `int`, `double` ...), it is possible to provide a recipe to interpret expressions like `a1 + a2`
- `a1 + a2` is interpreted as a function call `operator+(a1, a2)`
- Using suitably chosen operators to overload, we can make expressions involving objects of a class type more intuitive

OVERLOADING OPERATORS

+	−	*	/	%	&	^	|	
+=	−=	*=	/=	%=	&=	^=	|=	=
++	−−	&&	||	!	!=	==		
<	>	!=	==	<=	>=	<=>	=	=
()	[]	,	->	->*	<<	<<=	>>=	>>
new	delete	new[]	delete[]					

Table: List of operators you can overload. (But remember, *can* and *should* are not the same thing!)

- Think carefully about the impact an overloaded operator will have on the readability of your code. Whether or not it the impact is beneficial depends on the use case
- Many important commonly used C++ features depend on suitably overloaded operators. E.g., `std::cout << "Hello\n";`

C++ CLASSES

```
struct Vector3 {
  double x, y, z;
};
```

- Usually, encapsulates some data to represent an idea

C++ CLASSES

```
struct Vector3 {
    double x, y, z;
    auto mag2() -> double
    {
        return x * x + y * y + z * z;
    }
};
```

- Usually, encapsulates some data to represent an idea
- Specifies possible operations on the data

C++ CLASSES

```cpp
struct Vector3 {
    double x, y, z;
    auto mag2() -> double
    {
        return x * x + y * y + z * z;
    }
};

void somefunc()
{
    int a, b, c;
    Vector3 d, e, f;
    // ...
    if (d.mag2() < e.mag2()) doX();
}
```

- Usually, encapsulates some data to represent an idea
- Specifies possible operations on the data
- Once defined, one can create and use variables of the new type

C++ CLASSES

```cpp
struct Vector3 {
    double x, y, z;
    auto mag2() -> double
    {
        return x * x + y * y + z * z;
    }
};

void somefunc()
{
    int a, b, c; // On the stack
    Vector3 d, e, f; // On the stack
    // ...
    if (d.mag2() < e.mag2()) doX();
}
```

- Usually, encapsulates some data to represent an idea
- Specifies possible operations on the data
- Once defined, one can create and use variables of the new type

In C++, objects of user defined types live on the stack by default, unless explicitly created on the heap.

C++ CLASSES

Functions, relevant for the idea, can be declared inside the `struct` :

```
struct complex {
    double real, imaginary;
    auto modulus() -> double
    {
        return sqrt(real * real +
                    imaginary * imaginary);
    }
};
...
complex a{1, 2}, b{3, 4};
complex* cptr = &a;
auto c = a.modulus();// 1 * 1 + 2 * 2
auto d = b.modulus();// 3 * 3 + 4 * 4
auto e = cptr->modulus();// 1 * 1 + 2 * 2
```

- Data and function members

C++ CLASSES

Functions, relevant for the idea, can be declared inside the `struct` :

```
struct complex {
    double real, imaginary;
    auto modulus() -> double
    {
        return sqrt(real * real +
                    imaginary * imaginary);
    }
};
...
complex a{1, 2}, b{3, 4};
complex* cptr = &a;
auto c = a.modulus();// 1 * 1 + 2 * 2
auto d = b.modulus();// 3 * 3 + 4 * 4
auto e = cptr->modulus();// 1 * 1 + 2 * 2
```

- Data and function members
- A (non-static) member function is invoked on an instance of our structure.

C++ CLASSES

Functions, relevant for the idea, can be declared inside the `struct` :

```
struct complex {
    double real, imaginary;
    auto modulus() -> double
    {
        return sqrt(real * real +
                    imaginary * imaginary);
    }
};
...
complex a{1, 2}, b{3, 4};
complex* cptr = &a;
auto c = a. modulus();// 1 * 1 + 2 * 2
auto d = b. modulus();// 3 * 3 + 4 * 4
auto e = cptr-> modulus();// 1 * 1 + 2 * 2
```

- Data and function members
- A (non-static) member function is invoked on an instance of our structure.
- `a.real` is the real part of `a`. `a.modulus()` is the modulus of a.

C++ CLASSES

Functions, relevant for the idea, can be declared inside the `struct` :

```cpp
struct complex {
    double real, imaginary;
    auto modulus() -> double
    {
        return sqrt(real * real +
                    imaginary * imaginary);
    }
};
...
complex a{1, 2}, b{3, 4};
complex* cptr = &a;
auto c = a.modulus();// 1 * 1 + 2 * 2
auto d = b.modulus();// 3 * 3 + 4 * 4
auto e = cptr->modulus();// 1 * 1 + 2 * 2
```

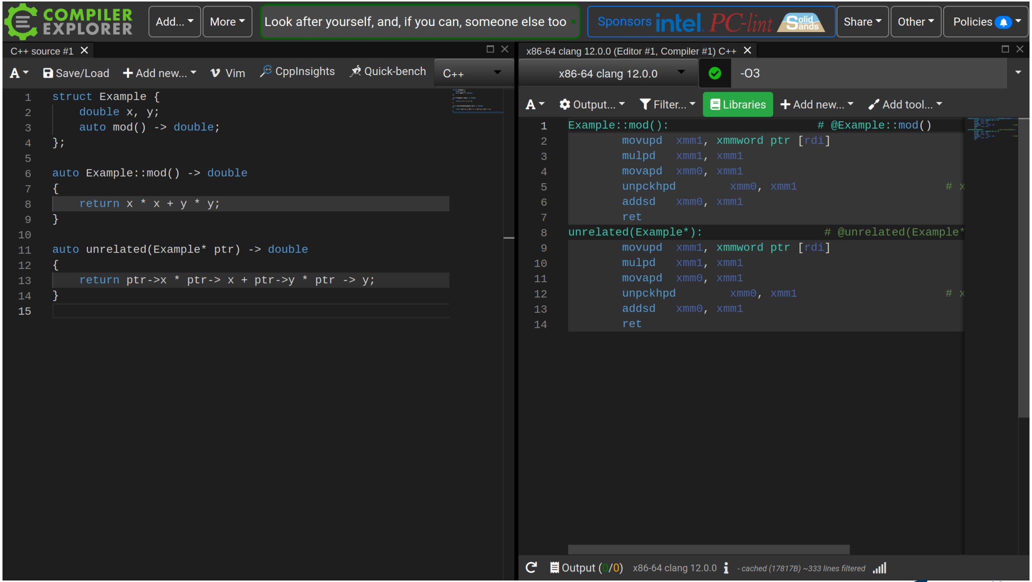
- Data and function members
- A (non-static) member function is invoked on an instance of our structure.
- `a.real` is the real part of `a`. `a.modulus()` is the modulus of a.
- Inside a member function, member variables correspond to the invoking instance.

C++ CLASSES

Functions, relevant for the idea, can be declared inside the `struct` :

```cpp
struct complex {
    double real, imaginary;
    auto modulus() -> double
    {
        return sqrt(real * real +
                    imaginary * imaginary);
    }
};
...
complex a{1, 2}, b{3, 4};
complex* cptr = &a;
auto c = a.modulus();// 1 * 1 + 2 * 2
auto d = b.modulus();// 3 * 3 + 4 * 4
auto e = cptr->modulus();// 1 * 1 + 2 * 2
```

- Data and function members
- A (non-static) member function is invoked on an instance of our structure.
- `a.real` is the real part of `a`. `a.modulus()` is the modulus of a.
- Inside a member function, member variables correspond to the invoking instance.
- Think of a call like `a.modulus()` as `complex::modulus(a)` The address of the object on the left of the "." is the implicit first argument to the member function.


COMPILER EXPLORER
Add...
More
Look after yourself, and, if you can, someone else too
Sponsors
Share
Other
Policies
C++ source #1
Save/Load
Add new...
Vim
CppInsights
Quick-bench
C++
struct Example {
double x, y;
auto mod() -> double;
};
auto Example::mod() -> double
{
return x * x + y * y;
}
auto unrelated(Example* ptr) -> double
{
return ptr->x * ptr-> x + ptr->y * ptr -> y;
}
x86-64 clang 12.0.0 (Editor #1, Compiler #1) C++
x86-64 clang 12.0.0
-O3
Output...
Filter...
Libraries
Add new...
Add tool...
Example::mod(): # @Example::mod()
movupd xmm1, xmmword ptr [rdi]
mulpd xmm1, xmm1
movapd xmm0, xmm1
unpckhpd xmm0, xmm1
addsd xmm0, xmm1
ret
unrelated(Example*): # @unrelated(Example*
movupd xmm1, xmmword ptr [rdi]
mulpd xmm1, xmm1
movapd xmm0, xmm1
unpckhpd xmm0, xmm1
addsd xmm0, xmm1
ret
Output (0/0)
x86-64 clang 12.0.0
- cached (17817B) ~333 lines filtered

OPERATORS AS MEMBER FUNCTIONS

```
struct complex {
    double real, imag;
    auto modulus() -> double
    {
        return sqrt(real * real +
                    imag * imag);
    }
    auto operator+(complex other) -> complex
    {
        return {real + other.real,
            imag + other.imag};
    }
};
```

- Since operators working with class types are normal functions, one can have operators as member functions
- The implicit argument (invoking instance) is on the left hand side for binary operators. That's why the binary operator `+` is defined here as a member function taking only one argument

MEMBER FUNCTIONS AND CONST

```
struct complex {
    double m_real, m_imag;
    auto modulus() -> double;
    auto operator-( const complex&  rhs)
        -> complex;
};
auto operator*( const complex* lhsptr ,
                const complex& rhs )
    -> complex
{
    // ...
}
void somewhere_else()
{
    complex z1, z2;
    auto z3 = z1 - z2;
    // We know z2 didn't change.
    // But did z1 ?
}
```

- Explicit arguments to member functions can be declared `const` similar to arguments for any other function

MEMBER FUNCTIONS AND CONST

```
1  struct complex {
2      double m_real, m_imag;
3      auto modulus() -> double;
4      auto operator-( const complex&  rhs)
5          -> complex;
6  };
7  auto operator*( const complex* lhsptr ,
8                  const complex& rhs )
9      -> complex
10 {
11     // ...
12 }
13 void somewhere_else()
14 {
15     complex z1, z2;
16     auto z3 = z1 - z2;
17     // We know z2 didn't change.
18     // But did z1 ?
19 }
```

- Explicit arguments to member functions can be declared `const` similar to arguments for any other function
- Non-member operator definitions like `operator*` in line 6 have two parameters: the first represents the left side and the second the right side. These are easy to declare `const`.

MEMBER FUNCTIONS AND CONST

```
1  struct complex {
2      double m_real, m_imag;
3      auto modulus() -> double;
4      auto operator-( const complex& rhs)
5          -> complex;
6  };
7  auto operator*( const complex* lhsptr ,
8                  const complex& rhs )
9      -> complex
10 {
11     // ...
12 }
13 void somewhere_else()
14 {
15     complex z1, z2;
16     auto z3 = z1 - z2;
17     // We know z2 didn't change.
18     // But did z1 ?
19 }
```

- But member functions have an implicit argument: the `this` pointer, pointing at the calling instance. In member operators, `this` points at the object on the left side. Only the right side is passed explicitly in a member operator definition (line 4).
- Where do we put a `const` qualifier, if we want to express that the calling instance (LHS argument) must not change?

MEMBER FUNCTIONS AND CONST

```cpp
1  struct complex {
2      double m_real, m_imag;
3      auto modulus() const -> double;
4      auto operator-( const complex& b) const
5          -> complex;
6  };
7  auto operator*( const complex* lhsptr,
8                  const complex& rhs )
9      -> complex
10 {
11     // ...
12 }
13 void somewhere_else()
14 {
15     complex z1, z2;
16     auto z3 = z1 - z2;
17     // We know z2 didn't change.
18     // We know z1 didn't change,
19     // as we called a const member
20 }
```

- But member functions have an implicit argument: the `this` pointer, pointing at the calling instance. In member operators, `this` points at the object on the left side. Only the right side is passed explicitly in a member operator definition (line 4).
- Where do we put a `const` qualifier, if we want to express that the calling instance (LHS argument) must not change?
- Answer: After the closing parentheses of the function signature.

SOME EXAMPLE CLASSES

```cpp
class Angle {
    double rd = 0;
public:
    enum unit {
        radian,
        degree
    };
    Angle operator-(Angle a) const ;
    Angle operator+(Angle a) const ;
    Angle operator-=(Angle a) ;
```

```cpp
class Vector3
{
public:
    enum crdtype {cartesian=0,polar=1};
    inline auto x() const -> double {return dx;}
    inline void x(double gx) {dx=gx;}
    auto dot(const Vector3 &p) const -> double;
    Vector3 cross(const Vector3 &p) const;
```

```cpp
class IsingLattice {
public:
    using update_type =
        std::pair<size_t, size_t>;
    IsingLattice();
    IsingLattice(size_t Nx, double JJ);
    void setLatticeSize(size_t ns);
```

```cpp
class KMer {
public:
    Nucleotide at(size_t i);
    auto operator==(const KMer &) const -> bool;
```

```cpp
class SimulationManager {
public:
    void loadSettings(std::string file);
    auto checkConfig() const -> bool;
    void start();
```

OBJECT INITIALISATION: CONSTRUCTORS

- In C++, initialisation functions for a struct have the same name as the struct. They are called *constructors*.

```
struct complex {
    complex(double re, double im)
    {
        real = re;
        imaginary = im;
    }
};
```

- Alternative syntax to initialise variables in constructors

```
struct complex
{
    complex(double re, double im) : real{re}, imaginary{im} {}
};
```

- A class can have as many constructors as it needs.

CONSTRUCTORS

```
struct complex {
    complex(double re, double im)
    {
      real = re;
      imaginary = im;
    }
    complex()
    {
      real = imaginary = 0;
    }
    double real, imaginary;
};
...
complex a(3.2, 9.3) ; // C++11 and older
complex b{4.3, 1.9} ; // Preferred
complex c{} ;
```

- Constructors may be (and normally are) overloaded.
- When a variable is declared, a constructor whose number and type of arguments matches the initialiser expression is implicitly called
- The default constructor is the one without any arguments. That is the one invoked when no arguments are given while creating the object.

CONSTRUCTORS

```
struct complex
{
    complex(double re, double im)
    {
        real = re;
        imaginary = im;
    }
    complex() {}
    double real{0.};
    double imaginary{0.};
};
...
complex a(4.3, 23.09), b;
```

- Member variables can be initialised to ”default values“ at the point of declaration

CONSTRUCTORS

```
struct complex
{
    complex(double re, double im)
    {
        real = re;
        imaginary = im;
    }
    complex() {}
    double real{0.};
    double imaginary{0.};
};
...
complex a(4.3, 23.09), b;
```

- Member variables can be initialised to ”default values“ at the point of declaration
- Member variables not touched by the constructor stay at their default values

CONSTRUCTORS

```cpp
struct complex
{
    complex(double re, double im)
        : real{re}, imaginary{im}
    {
    }
    complex() {}
    double real{0.};
    double imaginary{0.};
};
...
complex a(4.3, 23.09), b;
```

- Member variables can be initialised to ”default values“ at the point of declaration
- Member variables not touched by the constructor stay at their default values
- Preferred syntax for initialisation of members in a constructor is shown here . This form of initialisation outside the constructor function body is only possible for constructors

FREEING MEMORY FOR USER DEFINED TYPES

```cpp
struct darray
{
    double *data = nullptr;
    size_t sz = 0;
    darray(size_t N) : sz{N} {
        data = new double[sz];
    }
};

auto tempfunc(double phasediff) -> double
{
    // find number of elements
    darray A{large_number};
    // do some great calculations
    return answer;
}
```

What happens to the memory? The struct `darray` has a pointer member, which points to dynamically allocated memory

- When the life of the variable `A` ends, the member variables (e.g. the pointer `data`) go out of scope.
- How does one free the dynamically allocated memory attached to the member `data`?

FREEING MEMORY FOR USER DEFINED TYPES

```
struct darray
{
    double *data{nullptr};
    size_t sz{0};
    darray(size_t N) : sz{N} {
        data = new double[sz];
    }
    ~darray() {
        if (data) delete [] data;
    }
};

auto tempfunc(double phasediff) -> double
{
    // find number of elements
    darray A{large_number};
    // do some great calculations
    return answer;
}
```

For any class which explicitly allocates dynamic memory

- We need a function that cleans up all explicitly allocated memory in use, so that we call it for every object whose lifetime is about to end.
- In C++, such functions are called destructors, and have the name ~ followed by the class name.
- Destructors take no arguments, and there is exactly one for each class
- The destructor is automatically called when a variable expires. You don't call it explicitly. It is **always** called whenever the scope of an object ends! It is impossible to forget.

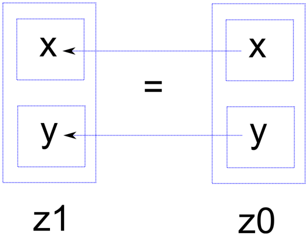
DESTRUCTORS

```
class A {
    A() {}
    ~A() {}
};
auto demo(A)
{
    A v1;
    try {
        A v2;
        // calc

    } // ~A() for v2
    catch {
      // ...
    }
} // ~A() for v1
```

- No matter how you exit a scope, if the scope of a variable ends, its destructor is invoked automatically
- What if we acquire resources in constructors and clean up in the destructor? It would be impossible to forget to free resources when we are done!

COPYING AND ASSIGNMENTS

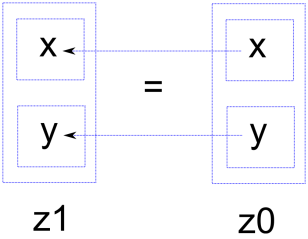
```cpp
struct complex
{
    double x, y;
};
//...
complex z0{2.0, 3.0}, z1;
z1 = z0; // assignment operator
complex z2{z0}; //copy constructor
```



- While copying and assigning, in most cases, we want to assign the data members to the corresponding members

COPYING AND ASSIGNMENTS

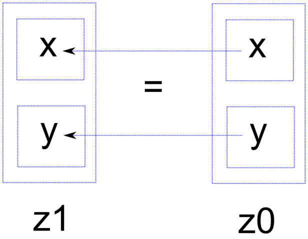
```
struct complex
{
    double x, y;
};
//...
complex z0{2.0, 3.0}, z1;
z1 = z0; // assignment operator
complex z2{z0}; //copy constructor
```



- While copying and assigning, in most cases, we want to assign the data members to the corresponding members
- This happens automatically, but using special functions for these copy operations

COPYING AND ASSIGNMENTS

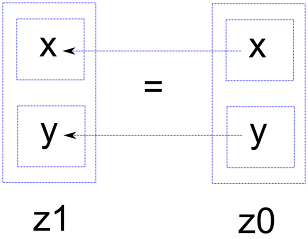
```
struct complex
{
    double x, y;
};
//...
complex z0{2.0, 3.0}, z1;
z1 = z0; // assignment operator
complex z2{z0}; //copy constructor
```



- While copying and assigning, in most cases, we want to assign the data members to the corresponding members
- This happens automatically, but using special functions for these copy operations
- You can redefine them for your class

COPYING AND ASSIGNMENTS

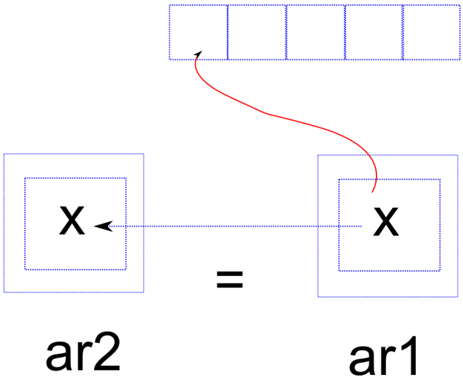
```
struct complex
{
    double x, y;
};
//...
complex z0{2.0, 3.0}, z1;
z1 = z0; // assignment operator
complex z2{z0}; //copy constructor
```



- While copying and assigning, in most cases, we want to assign the data members to the corresponding members
- This happens automatically, but using special functions for these copy operations
- You can redefine them for your class
- Why would you want to ?

COPYING AND ASSIGNMENTS

```
class darray {
    double *x;
};
darray::darray(unsigned n)
{
    x = new double[n];
}
void foo()
{
    darray ar1(5);
    darray ar2{ar1}; //copy constructor
    ar2[3] = 2.1;
    //oops! ar1[3] is also 2.1 now!
} //trouble
```

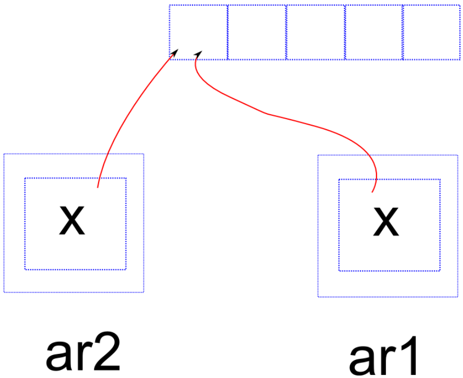


Copying pointers with dynamically allocated memory

- May not be what we want

COPYING AND ASSIGNMENTS

```
class darray {
    double *x;
};
darray::darray(unsigned n)
{
    x = new double[n];
}
void foo()
{
    darray ar1(5);
    darray ar2{ar1}; //copy constructor
    ar2[3] = 2.1;
    //oops! ar1[3] is also 2.1 now!
} //trouble
```



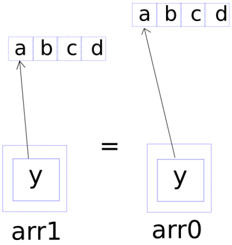
Copying pointers with dynamically allocated memory

- May not be what we want
- Leads to "double free" errors when the objects are destroyed

COPYING AND ASSIGNMENTS

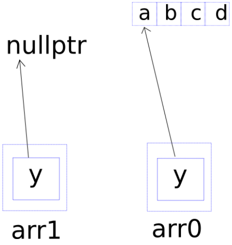
```cpp
class darray {
    double *x{nullptr};
    unsigned int len{};
public:
    // Copy constructor
    darray(const darray &);
    //assignment operator
    auto operator=(const darray&) -> darray&;
};
darray::darray(const darray& other)
{
    if (other.len != 0) {
        len = other.len;
        x = new double[len];
        for (unsigned i = 0; i < len; ++i) {
            x[i] = other.x[i];
        }
    }
}
auto darray::operator=(const darray& other) -> darray&
{
    if (this != &other) {
        if (len != other.len) {
```

```cpp
            len = other.len;
            if (x) delete [] x;
            x = new double[len];
        }
        for (unsigned i = 0; i < len; ++i) {
            x[i] = other.x[i];
        }
    }
    return *this;
}
```



MOVE CONSTRUCTOR/ASSIGNMENT OPERATOR

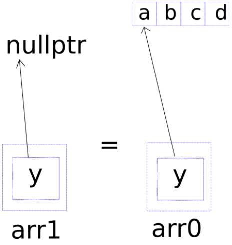
```cpp
class darray {
    darray(darray &&); //Move constructor
    auto operator=(darray &&) -> darray&;
    //Move assignment operator
};
darray::darray(darray&& other)
{
    len = other.len;
    x = other.x;
    other.x = nullptr;
}
auto darray::operator=(darray&& other)
    -> darray& {
    len = other.len;
    x = other.x;
    other.x = nullptr;
    return *this;
}
darray d1(3);
init_array(d1); //d1 = {1.0,2.0,3.0}
darray d2{d1}; //Copy construction
// d1 and d2  are {1.,2.,3.}
darray d3{std::move(d1)}; //Move
// d3 is {1.,2.,3.}, but d1 is empty!
```



- Construct or assign from an R-value reference (`darray &&`)

MOVE CONSTRUCTOR/ASSIGNMENT OPERATOR

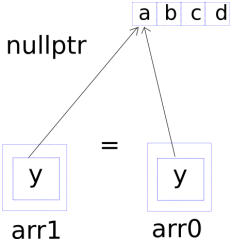
```cpp
class darray {
    darray(darray &&); //Move constructor
    auto operator=(darray &&) -> darray&;
    //Move assignment operator
};
darray::darray(darray&& other)
{
    len = other.len;
    x = other.x;
    other.x = nullptr;
}
auto darray::operator=(darray&& other)
    -> darray& {
    len = other.len;
    x = other.x;
    other.x = nullptr;
    return *this;
}
darray d1(3);
init_array(d1); //d1 = {1.0,2.0,3.0}
darray d2{d1}; //Copy construction
// d1 and d2  are {1.,2.,3.}
darray d3{std::move(d1)}; //Move
// d3 is {1.,2.,3.}, but d1 is empty!
```



- Construct or assign from an R-value reference (`darray &&`)
- Steal resources from RHS

MOVE CONSTRUCTOR/ASSIGNMENT OPERATOR

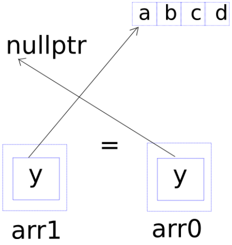
```
class darray {
    darray(darray &&); //Move constructor
    auto operator=(darray &&) -> darray&;
    //Move assignment operator
};
darray::darray(darray&& other)
{
    len = other.len;
    x = other.x;
    other.x = nullptr;
}
auto darray::operator=(darray&& other)
    -> darray& {
    len = other.len;
    x = other.x;
    other.x = nullptr;
    return *this;
}
darray d1(3);
init_array(d1); //d1 = {1.0,2.0,3.0}
darray d2{d1}; //Copy construction
// d1 and d2  are {1.,2.,3.}
darray d3{std::move(d1)}; //Move
// d3 is {1.,2.,3.}, but d1 is empty!
```



- Construct or assign from an R-value reference (`darray &&`)
- Steal resources from RHS

MOVE CONSTRUCTOR/ASSIGNMENT OPERATOR

```cpp
class darray {
    darray(darray &&); //Move constructor
    auto operator=(darray &&) -> darray&;
    //Move assignment operator
};
darray::darray(darray&& other)
{
    len = other.len;
    x = other.x;
    other.x = nullptr;
}
auto darray::operator=(darray&& other)
    -> darray& {
    len = other.len;
    x = other.x;
    other.x = nullptr;
    return *this;
}
darray d1(3);
init_array(d1); //d1 = {1.0,2.0,3.0}
darray d2{d1}; //Copy construction
// d1 and d2  are {1.,2.,3.}
darray d3{std::move(d1)}; //Move
// d3 is {1.,2.,3.}, but d1 is empty!
```



- Construct or assign from an R-value reference (`darray &&`)
- Steal resources from RHS
- Put disposable content in RHS

MOVE CONSTRUCTOR/ASSIGNMENT OPERATOR

- You can enable move semantics for your class by writing a constructor or assignment operator using an R-value reference

MOVE CONSTRUCTOR/ASSIGNMENT OPERATOR

- You can enable move semantics for your class by writing a constructor or assignment operator using an R-value reference
- Usually you will not be using it explicitly. Results of the evaluation of expressions might create a nameless object containing the resultant value (*prvalue*: pure r-value). A function may be returning a named entity which is about to expire (*xvalue*: expiring value) References to such objects are called R-value references. A move constructor or assignment operator is automatically invoked if constructor argument is an R-value reference

MOVE CONSTRUCTOR/ASSIGNMENT OPERATOR

- You can enable move semantics for your class by writing a constructor or assignment operator using an R-value reference
- Usually you will not be using it explicitly. Results of the evaluation of expressions might create a nameless object containing the resultant value (*prvalue*: pure r-value). A function may be returning a named entity which is about to expire (*xvalue*: expiring value) References to such objects are called R-value references. A move constructor or assignment operator is automatically invoked if constructor argument is an R-value reference
- You can invoke the move constructor by casting the function argument to an R-value reference, e.g.
  `darray d3{std::move(d1)}`

BIG FIVE (OR ZERO)

- Default constructor
- Copy constructor
- Move constructor
- Assignment operator
- Move assignment operator

- How many of these do you have to write for each and every class you make ?

BIG FIVE (OR ZERO)

- Default constructor
- Copy constructor
- Move constructor
- Assignment operator
- Move assignment operator

- How many of these do you have to write for each and every class you make ?
- Answer: **None!** If you don't have bare pointers in your class, and don't want anything fancy happening, the compiler will auto-generate reasonable defaults. "Rule of zero"

BIG FIVE

```cpp
class darray {
public:
    darray(double x, double y) : re{x}, im{y} {}
    darray() = default;
    darray(const darray &) = default;
    darray(darray &&) = default;
    auto operator=(const darray&) -> darray& = default;
    auto operator=(darray&&) -> darray& = default;
};
```

- If you have to write any constructor yourself, auto-generation of the default constructor is disabled

BIG FIVE

```cpp
class darray {
public:
    darray(double x, double y) : re{x}, im{y} {}
    darray() = default;
    darray(const darray &) = default;
    darray(darray &&) = default;
    auto operator=(const darray&) -> darray& = default;
    auto operator=(darray&&) -> darray& = default;
};
```

- If you have to write any constructor yourself, auto-generation of the default constructor is disabled
- But you can request default versions of the any of these functions as shown

BIG FIVE

```cpp
class darray {
    darray() = delete;
    darray(const darray &) = delete;
    darray(darray &&) = default;
    auto operator=(const darray &) -> darray& = delete;
    auto operator=(darray &&) -> darray& = default;
};
```

- You can also explicitly request that one or more of these are not auto-generated
- In the example shown here, it will not be possible to copy objects of the class, but they can be moved

COPY AND SWAP

- We want to reuse the code in the copy constructor and destructor to do memory management

```
auto operator=(const darray& oth) -> darray& {
  if (this != &oth) {
    if (arr && sz != oth.sz) {
      sz = oth.sz;
      delete [] arr;
      arr = new T[sz];
    }
    for (size_t i = 0; i < sz; ++i)
      arr[i] = oth.arr[i];
  }
  return *this;
}
auto operator=(darray&& oth) -> darray& {
  swap(oth);
  return *this;
}
```

COPY AND SWAP

- We want to reuse the code in the copy constructor and destructor to do memory management
- Pass argument to the assignment operator by value instead of reference

```
auto operator=(darray d) -> darray& {
  swap(d);
  return *this;
}
// No further move assignment operator!
```

COPY AND SWAP

- We want to reuse the code in the copy constructor and destructor to do memory management
- Pass argument to the assignment operator by value instead of reference
- Use the class member function `swap` to swap the data with the newly created copy

```
auto operator=(darray d) -> darray& {
  swap(d);
  return *this;
}
// No further move assignment operator!
```

- Neat trick that works in most cases
- Reduces the big five to big four

PUBLIC AND PRIVATE MEMBERS

Separating interface and implementation

```cpp
auto foo(complex a, int p, truck c) -> int
{
    complex z1, z2, z3 = a;
...
    z1 = z1.argument() * z2.modulus() * z3.conjugate();
    c.start(z1.imaginary * p);
}
```

Imagine that ...

- We have used our complex number structure in a lot of places

PUBLIC AND PRIVATE MEMBERS

Separating interface and implementation

```
auto foo(complex a, int p, truck c) -> int
{
    complex z1, z2, z3 = a;
...
    z1 = z1.argument() * z2.modulus() * z3.conjugate();
    c.start(z1.imaginary * p);
}
```

Imagine that ...

- We have used our complex number structure in a lot of places
- Then one day, it becomes evident that it is more efficient to define the complex numbers in terms of the modulus and argument, instead of the real and imaginary parts.

PUBLIC AND PRIVATE MEMBERS

Separating interface and implementation

```
auto foo(complex a, int p, truck c) -> int
{
    complex z1, z2, z3 = a;
...
    z1 = z1.argument() * z2.modulus() * z3.conjugate();
    c.start(z1.imaginary * p);
}
```

Imagine that ...

- We have used our complex number structure in a lot of places
- Then one day, it becomes evident that it is more efficient to define the complex numbers in terms of the modulus and argument, instead of the real and imaginary parts.
- We have to change a lot of code.

PUBLIC AND PRIVATE MEMBERS

Separating interface and implementation

```cpp
auto foo(complex a, int p, truck c) -> int
{
    complex z1, z2, z3 = a;
...
    z1 = z1.argument() * z2.modulus() * z3.conjugate();
    c.start(z1.imaginary * p);
}
```

Imagine that ...

- External code calling only member functions to access member data can survive
- Direct use of member variables while using a class is often messy, the implementer of the class then loses the freedom to change internal organisation of the class for efficiency or other reasons

C++ CLASSES

```
class complex
{
public:
    complex(double re, double im)
    : m_real(re), m_imag(im) {}
    complex() = default;
    auto real() const -> double { return m_real; }
    auto imag() const -> double { return m_imag; }
...
private:
    double m_real = 0., m_imag = 0.;
};
```

struct $\Longrightarrow$ members public by default
class $\Longrightarrow$ members private by default

- Members declared under the keyword `private` can not be accessed from outside
- Public members (data or function) can be accessed
- Provide a consistent and useful interface through public functions
- Keep data members hidden
- Make accessor functions `const` when possible

Exercise 2.1:

The program `examples/complex_number_class.cc` contains a version of the complex number class, with all syntax elements we discussed in the class. It is heavily commented with explanations for every subsection. Please read it to revise all the syntax relating to classes. Write a `main` program to use and test the class.

CONSTRUCTOR/DESTRUCTOR CALLS

Exercise 2.2:

The file `examples/verbose_ctordtor.cc` demonstrates the automatic calls to constructors and destructors. The simple class `Vbose` has one `string` member. All its constructors and destructors print messages to the screen when they are called. The `main()` function creates and uses some objects of this class. Follow the messages printed on the screen and link them to the statements in the program. Does it make sense (i) When the copy constructor is called ? (ii) When is the move constructor invoked ? (iii) When the objects are destroyed ?

Suggested reading: http://www.informit.com/articles/printerfriendly/2216986

Exercise 2.3:

The program `examples/onexcept.cc` shows the behaviour of constructor/destructor calls when an exception is called. Observe that exiting a function via an exception is also leaving the scope, and therefore invokes the destructor.

MAKING STD::COUT RECOGNISE CLASS

Teaching cout how to print your type: overload operator `<<`

```
auto operator<<(std::ostream& os, const complex& a) -> std::ostream&
{
    os << a.real();
    if (a.imag() < 0) os << a.imag() << " i ";
    // If imag() is negative, it already has a - sign
    else os << " +" << a.imag() << " i ";
    return os;
}
complex a;
...
std::cout << "The roots are " << a << " and " << a.conjugate() << '\n';
```

AND SIMILARLY FOR STD::CIN

```
auto operator>>(std::istream& is, complex& a) -> std::istream&
{
    double x, y;
    is >> x >> y;
    a.set_real(x);
    a.set_imag(y);
    return is;
}
```

- It is up to you to decide IO operations for your classes
- The stream parameters can not be `const`, because by reading from or writing to the stream, we change its state

PRACTISE: WRITE A DATA ROW CLASS

Exercise 2.4:

You now have all the ingredients to write a data row class. A tabular data file has 5 columns. The first two are integers, the rest are doubles. Let's call the columns id, cat, x, y, and z, respectively. Make sure that there are IO stream overloads for the reading and writing objects of that type. Demonstrate by reading a suitable data file "multicolumn.dat", and storing the rows in a vector of your `DataRow` type. You should then be able to sort the vector according to any of the data columns.

DATATYPES

Type	Bits	Value
Float	0100 0000 0100 1001 0000 1111 1101 1011	3.1415927
Int	0100 0000 0100 1001 0000 1111 1101 1011	1078530011

- Same bits, different rules $\Longrightarrow$ different type

From arbitrary collection of members to a new "data type"

```
class Date {
    int m_day, m_month, m_year;
public:
    static auto today() -> Date;
    auto operator+(int n) const -> Date;
    auto operator-(int n) const -> Date;
    auto operator-(const Date &) const -> int;
};
```

- Make sure every way to create an object results in a valid state
- Provide only those operations on the data which keep the essential properties intact

CLASS INVARIANTS

- A class is supposed to represent an idea: a complex number, a date, a dynamic array.

CLASS INVARIANTS

- A class is supposed to represent an idea: a complex number, a date, a dynamic array.
- It will often contain data members of other types, with assumed constraints on those values:

CLASS INVARIANTS

- A class is supposed to represent an idea: a complex number, a date, a dynamic array.
- It will often contain data members of other types, with assumed constraints on those values:
  - A dynamic array is supposed to have a pointer that is either `nullptr` or a valid block of allocated memory, with the correct size also stored in the structure.

CLASS INVARIANTS

- A class is supposed to represent an idea: a complex number, a date, a dynamic array.
- It will often contain data members of other types, with assumed constraints on those values:
  - A dynamic array is supposed to have a pointer that is either `nullptr` or a valid block of allocated memory, with the correct size also stored in the structure.
  - A Date structure could have 3 integers for day, month and year, but they can not be, for example, 0,-1,1

CLASS INVARIANTS

- A class is supposed to represent an idea: a complex number, a date, a dynamic array.
- It will often contain data members of other types, with assumed constraints on those values:
  - A dynamic array is supposed to have a pointer that is either `nullptr` or a valid block of allocated memory, with the correct size also stored in the structure.
  - A Date structure could have 3 integers for day, month and year, but they can not be, for example, 0,-1,1
- Using `private` data members and well designed `public` interfaces, we can ensure that assumptions behind an idea are always true.

CLASS INVARIANTS

```cpp
class darray {
private:
    double * dataptr = nullptr;
    size_t sz = 0;
public:
// initialize with N elements
    darray(size_t N);
    ~darray();
// resize to N elements
    void resize(size_t N);
// other members who don't change
// dataptr or sz
};
```

- Construct ensuring class Invariants

CLASS INVARIANTS

```
class darray {
private:
    double * dataptr = nullptr;
    size_t sz = 0;
public:
// initialize with N elements
    darray(size_t N);
    ~darray();
// resize to N elements
    void resize(size_t N);
// other members who don't change
// dataptr or sz
};
```

- Construct ensuring class Invariants
- Maintain Invariants in every member

CLASS INVARIANTS

```cpp
class darray {
private:
    double * dataptr = nullptr;
    size_t sz = 0;
public:
// initialize with N elements
    darray(size_t N);
    ~darray();
// resize to N elements
    void resize(size_t N);
// other members who don't change
// dataptr or sz
};
```

- Construct ensuring class Invariants
- Maintain Invariants in every member
- $\rightarrow$ a structure which always has sensible values

STATIC MEMBERS

```
class Triangle {
public:
    static unsigned counter;
    Triangle() : ...
    {
        ++counter;
    }
    ~Triangle() { --counter; }
    static auto instanceCount() -> unsigned
    {
        return counter;
    }
};
... Triangle.cc ...
unsigned Triangle::counter = 0;
```

- Static variables exist only once for all objects of the class.

STATIC MEMBERS

```cpp
class Triangle {
public:
    static unsigned counter;
    Triangle() : ...
    {
        ++counter;
    }
    ~Triangle() { --counter; }
    static auto instanceCount() -> unsigned
    {
        return counter;
    }
};
... Triangle.cc ...
unsigned Triangle::counter = 0;
```

- Static variables exist only once for all objects of the class.
- Can be used to keep track of the number of objects of one type created in the whole application

STATIC MEMBERS

```cpp
class Triangle {
public:
    static unsigned counter;
    Triangle() : ...
    {
        ++counter;
    }
    ~Triangle() { --counter; }
    static auto instanceCount() -> unsigned
    {
        return counter;
    }
};
... Triangle.cc ...
unsigned Triangle::counter = 0;
```

- Static variables exist only once for all objects of the class.
- Can be used to keep track of the number of objects of one type created in the whole application
- Must be initialised in a source file somewhere, or else you get an "unresolved symbol" error

STATIC MEMBERS

```cpp
class Triangle {
public:
    static unsigned counter;
    Triangle() : ...
    {
        ++counter;
    }
    ~Triangle() { --counter; }
    static auto instanceCount() -> unsigned
    {
        return counter;
    }
};
... Triangle.cc ...
unsigned Triangle::counter = 0;
```

- Static member functions do not have an implicit `this` pointer argument. They can be invoked as `ClassName::function()`.

- Static variables exist only once for all objects of the class.
- Can be used to keep track of the number of objects of one type created in the whole application
- Must be initialised in a source file somewhere, or else you get an "unresolved symbol" error

SOME FUN: OVERLOADING THE () OPERATOR

```cpp
class swave
{
private:
  double a = 1.0, omega = 1.0;
public:
  swave() = default;
  swave(double x, double w) :
     a{x}, omega{w} {}
  auto operator()(double t) const -> double
  {
    return a * sin(omega * t);
  }
};
```

```cpp
const double pi = acos(-1);

int N = 100;
swave f{2.0, 0.4};
swave g{2.3, 1.2};

for (int i = 0; i < N; ++i) {
  double ar = 2 * i * pi / N;
  std::cout << i << " " << f(ar)
            << " " << g(ar)
            << '\n';
}
```

Functionals

- Function like objects, i.e., classes which define a `()` operator

SOME FUN: OVERLOADING THE () OPERATOR

```cpp
class swave
{
private:
  double a = 1.0, omega = 1.0;
public:
  swave() = default;
  swave(double x, double w) :
     a{x}, omega{w} {}
  auto operator()(double t) const -> double
  {
    return a * sin(omega * t);
  }
};
```

```cpp
const double pi = acos(-1);

int N = 100;
swave f{2.0, 0.4};
swave g{2.3, 1.2};

for (int i = 0; i < N; ++i) {
  double ar = 2 * i * pi / N;
  std::cout << i << " " << f(ar)
            << " " << g(ar)
            << '\n';
}
```

Functionals

- Function like objects, i.e., classes which define a `()` operator
- If they return a `bool` value, they are called predicates

FUNCTIONALS

Using function like objects

- They are like other variables. But they can be used as if they were functions!
- You can make vectors or lists of functionals, pass them as arguments ...
- Although you can run any recipe you want by overloading an operator, most operators are limited to one or two arguments. `()` can take as many as you need. This also contributes to functionals looking like functions when in use.

WRITE YOUR OWN FUNCTIONAL!

Exercise 2.5:

Write a functional class where the return value of f(x) is given by a user specified piece-wise continuous linear function. You should write a class PieceWise. It should have a function to read a vector of x_i,y_i values from a file. Sort them according to x values. Then implement an `operator()` function, so that when you write

```
PieceWise f;
f.read_file("somefile.dat");
auto y = f(x);
```

you get the correct piece wise linear function evaluated. Use the standard library function `std::lerp` to perform the linear interpolation.

OVERLOADING OTHER OPERATORS FOR EXPRESSIVE CODE

```cpp
// examples/collect.cc
class collect {
    std::vector<int> dat;
public:
    auto operator|(int i) -> collect&
    {
        dat.push_back(i);
        return *this;
    }
    auto operator~() const noexcept -> decltype(dat)
    {
        return dat;
    }
};
auto main() -> int
{
    auto C = collect{};
    C | 1 | 2 | 3 | 4 ;
    for (auto el : (~C)) {
        std::cout << el << "\n";
    }
}
```

- Operator overloading is not limited to arithmetic and shift operators.
- Sometimes, choosing the right operator to overload can increase the expressiveness of the code

```cpp
args | sv::drop(1) | sv::transform(str)
```

USER DEFINED LITERALS

Redefining the "" operator!

- You know how to create objects and set their values
- You even know how to construct with a given initial value

```
auto main() -> int
{
    double N=6.023e23;
    Temperature T;
    T.value(293.0);
    auto U = Temperature{373.0};
    auto T2 = 350_C;
    auto T3 = 900_K;
    complex c = 1+2_i;
    ...
}
```

USER DEFINED LITERALS

Redefining the "" operator!

- You know how to create objects and set their values
- You even know how to construct with a given initial value

```
int main()
{
    double N=6.023e23;
    Temperature T;
    T.value(293.0);
    auto U = Temperature(373.0);
    auto T2 = 350_C;
    auto T3 = 900_K;
    complex c = 1+2_i;
    ...
}
```


JÜLICH Forschungszentrum

USER DEFINED LITERALS

Redefining the "" operator!

- You know how to create objects and set their values
- You even know how to construct with a given initial value
- It's far cooler to initialise with your own literals!
- Redefine how literals are interpreted for your class
- Desirable to enable clean and easily read initialisations

```
int main()
{
    double N=6.023e23;
    Temperature T;
    T.value(293.0);
    auto T2 = 350_C;
    auto T3 = 900_K;
    complex c = 1+2_i;
    ...
}
```


JÜLICH Forschungszentrum

USER DEFINED LITERALS

```cpp
auto operator "" _K(long double d) -> Temperature
{
    return { static_cast<double>(d), Temperature::Unit::K };
}
auto operator "" _C(long double d) -> Temperature
{
    return { static_cast<double>(d), Temperature::Unit::C };
}
```

- Defining your own rules for how literals are interpreted for your class
- Desirable to enable clean and easily read initialisations

USER DEFINED LITERALS

Exercise

- The demo program `examples/literals.cc` shows how this is done using a simple “temperature” class
- Make something similar for a `Distance` class!

```
auto main() -> int
{
    double N = 6.023e23;
    auto T2 = 350_C;
    auto T3 = 900_K;
}
```

Inheritance and class hierarchies

CLASS INHERITANCE

Analogy

- Inherited traits: many properties shared among entities of different related types
- Each branch may add new properties
- Seems like a good fit to different ideas we may want to represent in code

CLASS INHERITANCE

```cpp
struct Point {double X, Y;};
class Triangle {
public:
  // Constructors etc., and then,
  void translate();
  void rotate(double byangle);
  auto area() const -> double;
  auto perimeter() const -> double;
private:
  Point vertex[3];
};
class Quadrilateral {
public:
  void translate();
  void rotate(double byangle);
  auto area() const -> double;
  auto perimeter() const -> double;
  auto is_convex() const -> bool;
private:
  Point vertex[4];
};
```

Geometrical figures

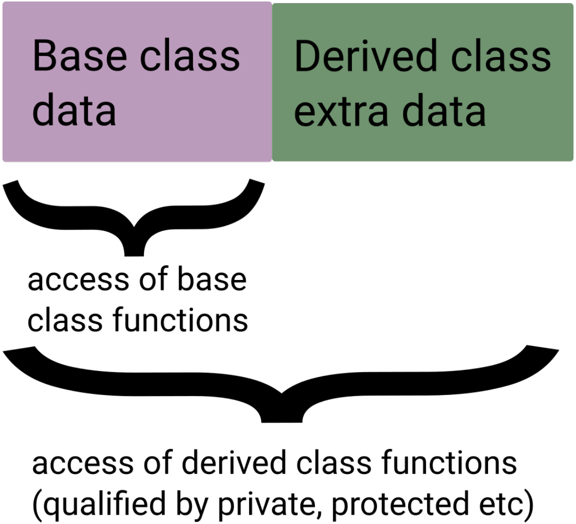
- Many actions (e.g. translate and rotate) will involve identical code
- Properties like `area` and `perimeter` make sense for all, but are better calculated differently for each type
- There may also be new properties (`is_convex`) introduced by a type

INHERITANCE: BASIC SYNTAX

```cpp
class SomeBase {
public:
    double f();
protected:
    int i;
private:
    int j;
};
class Derived : public SomeBase {
    void haha() {
    // can access f() and i
    // can not access j
    }
}; // Derived is also called a "subtype" of SomeBase
void elsewhere()
{
    SomeBase a; // only properties defined in SomeBase
    Derived b;  // b has properties defined in class
                // SomeBase and Derived
    // Can call a.f(),
    // but e.g., a.i = 0; is not allowed
}
```

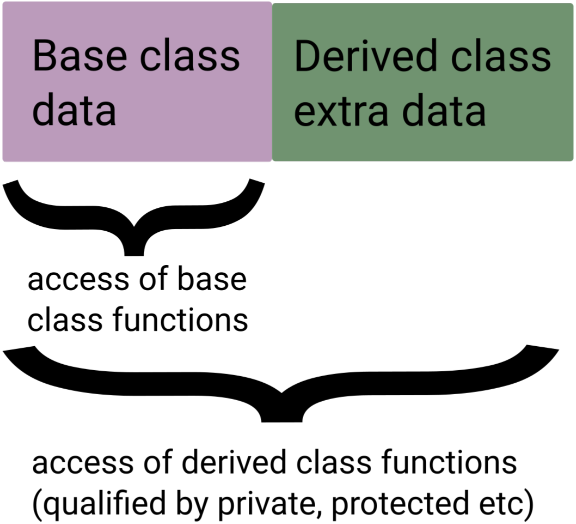
- Class members can be `private`, `protected` or `public`
- `public` members are accessible from everywhere
- `private` members are for internal use in one class
- `protected` members can be seen by derived classes

INHERITANCE



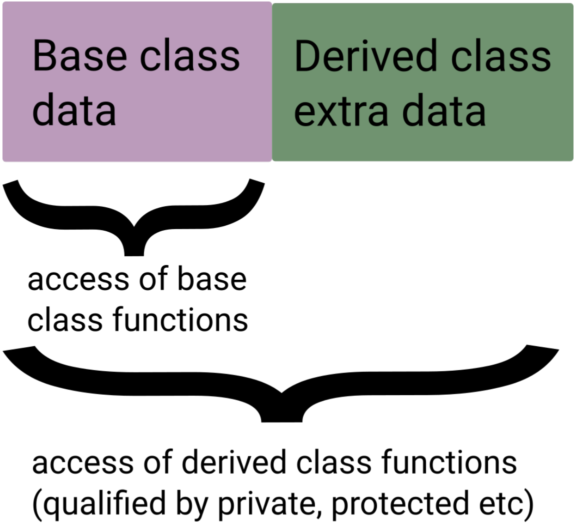
- Inheriting class may add more data, but it retains all the data of the base
- The base class functions, if invoked, will see a base class object
- The derived class object *is a* base class object, but with additional properties

INHERITANCE



- A pointer to a derived class always points to an address which also contains a valid base class object.
- `baseptr=derivedptr` is called "upcasting". Always allowed.
- Implicit downcasting is not allowed. Explicit downcasting is possible with `static_cast` and `dynamic_cast`

INHERITANCE



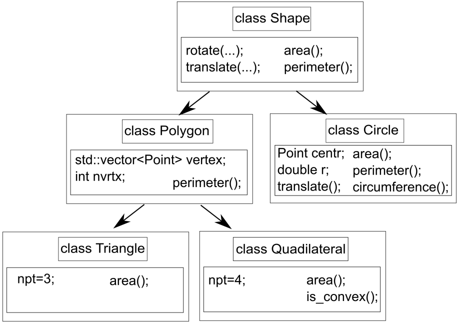

```cpp
class Base {
public:
  void f() { std::cout << "Base::f()\n"; }
protected:
  int i{4};
};
class Derived : public Base {
  int k{0};
public:
  void g() { std::cout << "Derived::g()\n"; }
};
auto main() -> int
{
  Derived b;
  Base *ptr = &b;
  ptr->g(); // Error!
  static_cast<Derived *>(ptr)->g(); //OK
}
```

CLASS INHERITANCE

- We want to write a program to
    - list the area of all the geometric objects
    - select the largest and smallest objects
    - draw

  in our system.
- A loop over a vector of them will be nice. But `vector< ??? >`
- Object oriented languages like C++, Java, Python ... have a concept of "inheritance" for the classes, to describe such conceptual relations between different types.
- 4 ways to solve this problem in C++ will be introduced at various points in this course

INHERITANCE WITH VIRTUAL FUNCTIONS



- Abstract concept class "Shape"
- Inherited classes add/change some properties
- and inherit other properties from "base" class

A triangle *is* a polygon. A polygon *is* a shape. A circle *is* a shape.

CLASS INHERITANCE WITH VIRTUAL FUNCTIONS

```
class Shape {
public:
  virtual ~Shape() = 0;
  virtual void rotate(double) = 0;
  virtual void translate(Point) = 0;
  virtual auto area() const -> double = 0;
  virtual auto perimeter() const -> double = 0;
};
class Circle : public Shape {
public:
  Circle(); // and other constructors
  ~Circle();
  void rotate(double phi) {}
  auto area() const -> double override
  {
    return pi * r * r;
  }
private:
  double r;
};
```

- Circle is a derived class from base class Shape
- A derived class inherits from its base(s), which are indicated in the class declaration.
- Functions marked as `virtual` in the base class *can be re-implemented* in a derived class.

Note: In C++, member functions are not virtual by default.

CLASS INHERITANCE WITH VIRTUAL FUNCTIONS

```
class Shape {
public:
  virtual ~Shape() = 0;
  virtual void rotate(double) = 0;
  virtual void translate(Point) = 0;
  virtual double area() const = 0;
  virtual auto perimeter() const -> double = 0;
};
class Circle : public Shape {
public:
  Circle(); // and other constructors
  ~Circle();
  void rotate(double phi) {}
  auto area() const -> double override
  {
    return pi * r * r;
  }
private:
  double r;
};
Shape a; // Error!
Circle b; // ok.
```

- A derived class inherits all member variables and functions from its base.
- `virtual` re-implemented in a derived class are said to be "overriden", and ought to be marked with `override`
- A class with a pure virtual function (with " = 0" in the declaration) is an abstract class. Objects of that type can not be declared.

CLASS INHERITANCE WITH VIRTUAL FUNCTIONS

```cpp
class Polygon : public Shape {
public:
  auto perimeter() const -> double final
  {
    // return sum over sides
  }
protected:
  vector<Point> vertex;
  int npt;
};
class Triangle : public Polygon {
public:
  Triangle() : npt(3)
  {
    vertex.resize(3); // ok
  }
  auto area() const -> double override
  {
    // return sqrt(s*(s-a)*(s-b)*(s-c))
  }
};
```

Syntax for inheritance

- Triangle implements its own `area()` function, but can not implement a `perimeter()`, as that is declared as `final` in `Polygon`. This is done if the implementation from the base class is good enough for intended inheriting classes.

CLASS INHERITANCE WITH VIRTUAL FUNCTIONS

```cpp
class Polygon : public Shape {
public:
  auto perimeter() const -> double final
  {
    // return sum over sides
  }
protected:
  vector<Point> vertex;
  int npt;
};
class Triangle : public Polygon {
public:
  Triangle() : npt(3)
  {
    vertex.resize(3); // ok
  }
  auto area()  -> double override  // Error!!
  {
    // return sqrt(s*(s-a)*(s-b)*(s-c))
  }
};
```

- The keyword `override` ensures that the compiler checks there is a corresponding base class function to override.
- Virtual functions can be re-implemented without this keyword, but an accidental omission of a `const` or an `&` can lead to really obscure runtime errors.

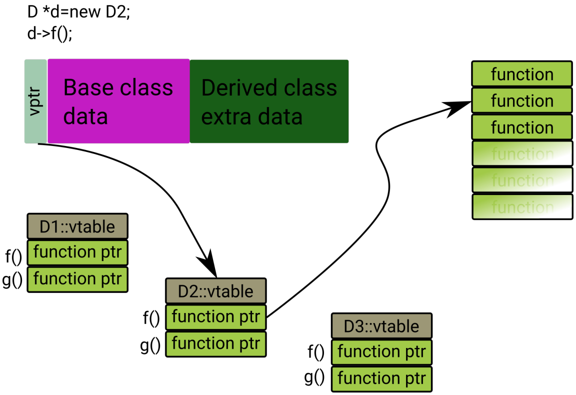
JÜLICH Forschungszentrum

CLASS INHERITANCE WITH VIRTUAL FUNCTIONS

```
auto main() -> int
{
  vector<std::unique_ptr<Shape>> shape;
  shape.push_back(std::make_unique<Circle>(0.5, Point(3,7)));
  shape.push_back(std::make_unique<Triangle>(Point(1,2), Point(3,3), Point(2.5,0)));
...
  for (size_t i = 0;i < shape.size(); ++i) {
    std::cout << shape[i]->area() << '\n';
  }
}
```

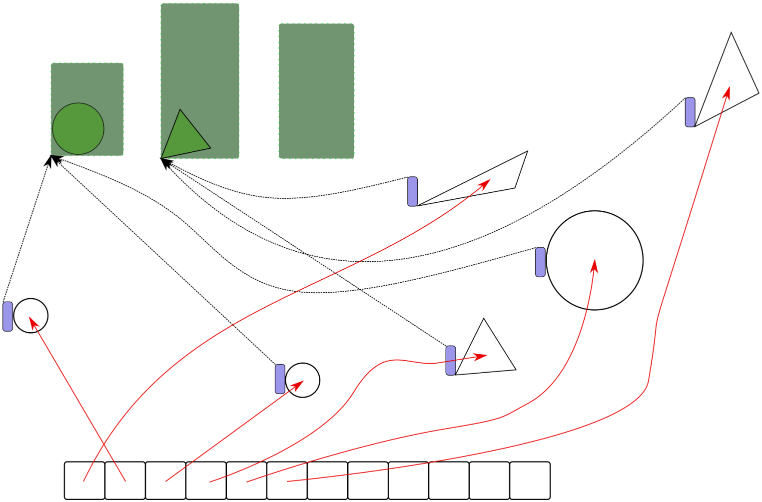
- A pointer to a base class is allowed to point to an object of a derived class
- Here, `shape[0]->area()` will call `Circle::area()`, `shape[1]->area()` will call `Triangle::area()`

CALLING VIRTUAL FUNCTIONS: HOW IT WORKS



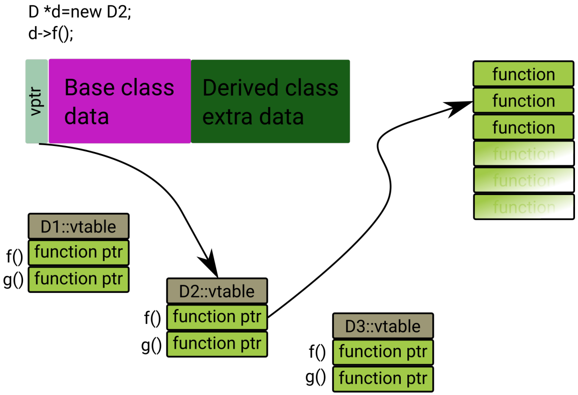
- For classes with virtual functions, the compiler inserts an invisible pointer member to the data and additional book keeping code
- There is a table of virtual functions for each derived class, with entries pointing to function code somewhere
- The `vptr` pointer points to the *vtable* of that particular class

CALLING VIRTUAL FUNCTIONS: HOW IT WORKS



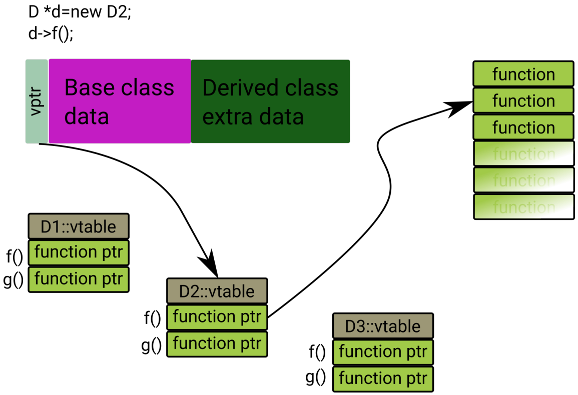
CALLING VIRTUAL FUNCTIONS: HOW IT WORKS

- Virtual function call proceeds by first finding the right *vtable*, then the correct entry for the called function, dereferencing that function pointer and then executing the correct function body
- Don't make everything virtual! The overhead, with modern machines and compilers, is not huge. But abusing this feature will hurt performance



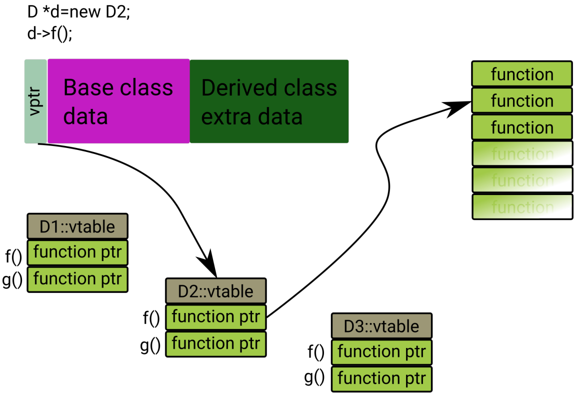
CALLING VIRTUAL FUNCTIONS: HOW IT WORKS

- Virtual function call proceeds by first finding the right *vtable*, then the correct entry for the called function, dereferencing that function pointer and then executing the correct function body
- Don't make everything virtual! The overhead, with modern machines and compilers, is not huge. But abusing this feature will hurt performance



CALLING VIRTUAL FUNCTIONS: HOW IT WORKS

- Virtual function call proceeds by first finding the right *vtable*, then the correct entry for the called function, dereferencing that function pointer and then executing the correct function body
- Don't make everything virtual! The overhead, with modern machines and compilers, is not huge. But abusing this feature will hurt performance



- But if virtual functions offer the cleanest solution with acceptable performance, don't invent weird things to avoid them!

CLASS INHERITANCE

Inherit or include as data member ?

```cpp
class DNA {
...
  std::valarray<char> seq;
};
class Cell : public DNA ???
or
class Cell {
...
  DNA mydna;
};
```

- A derived class extends the concept represented by its base class in some way.
- Although this extension might mean addition of new data members,

$$B = A \oplus newdata$$

does not necessarily mean the class for B should inherit from the class for A

CLASS INHERITANCE

Inherit or include as data member ?

```
class DNA {
...
  std::valarray<char> seq;
};

class Cell : public DNA ???

or

class Cell {
...
  DNA mydna;
};

```

is vs *has*

- A good guide to decide whether to inherit or include is to ask whether the concept B contains an object A, or whether any object of type B is also an object of type A, like a monkey is a mammal, and a triangle is a polygon.
- is $\Longrightarrow$ inherit . has $\Longrightarrow$ include

CLASS INHERITANCE

Inheritance summary

- Base classes to represent common properties of related types : e.g. all proteins are molecules, but all molecules are not proteins. All triangles are polygons, but not all polygons are triangles.
- Less code: often, only one or two properties need to be changed in an inherited class
- Helps create reusable code
- A base class may or may not be constructible (`Polygon` as opposed to `Shape`)

CLASS DECORATIONS

More control over classes

- Possible to initialise data in class declaration
- Initialiser list constructors
- Delegating constructors allowed
- Inheriting constructors possible

```
class A {
  int v[]{1, -1, -1, 1};
public:
  A() = default;
  A(std::initializer_list<int> &);
  A(int i, int j, int k, int l)
  {
    v[0] = i;
    v[1] = j;
    v[2] = k;
    v[3] = l;
  }
  //Delegate work to another constructor
  A(int i, int j) : A(i, j, 0, 0) {}
};
class B : public A {
public:
// Inherit all constructors from A
  using A::A;
  B(string s);
};
```

MORE CONTROL OVER CLASSES

- Explicit `default`, `delete`, `override` and `final`
- "Explicit is better than implicit"
- More control over what the compiler does with the class
- Compiler errors better than hard to trace run-time errors due to implicitly generated functions

```cpp
class A {
  // Automatically generated is ok
  A() = default;
  // Don't want to allow copy
  A(const A &) = delete;
  A & operator=(const A &) = delete;
  // Instead, allow a move constructor
  A(const A &&);
  // Don't try to override this!
  void getDrawPrimitives() final;
  virtual void show(int i);
};
class B : public A
{
  B() = default;
  void show()override; //will be an error!
};

```

Exercise 2.6:

The directory `exercises/geometry` contains a set of files for the classes Point, Shape, Polygon, Circle, Triangle, and Quadrilateral. In addition, there is a `main.cc` and a `CMakeLists.txt`. Observe the use of the keywords like **`default`**, **`override`**, **`final`** etc. Familiarise yourself with

- Implementation of inherited classes
- Compiling multi-file projects
- The use of base class pointer arrays to work with heterogeneous types of objects

```
mkdir build
cd build
CXX=g++ cmake ..
make
```

Chapter 3

Templates

Function and class templates

FUNCTION OVERLOADING

```cpp
auto power(int x, unsigned n) -> unsigned
{
    ans = 1;
    for (; n > 0; --n) ans *= x;
    return ans;
}
auto power(double x, double y) -> double
{
    return exp(y * log(x));
}
```

```cpp
auto someother(double mu, double alpha,
                    int rank) -> double
{
    double st=power(mu,alpha)*exp(-mu);

    if (n_on_bits(power(rank,5))<8)
        st=0;

    return st;
}
```

- When specialised strategies are needed to accomplish the same task for different types

FUNCTION OVERLOADING

```cpp
auto power(int x, unsigned n) -> unsigned
{
    ans = 1;
    for (; n > 0; --n) ans *= x;
    return ans;
}
auto power(double x, double y) -> double
{
    return exp(y * log(x));
}
```

```cpp
auto someother(double mu, double alpha,
               int rank) -> double
{
    double st=power(mu,alpha)*exp(-mu);

    if (n_on_bits(power(rank,5))<8)
        st=0;

    return st;
}
```

- When specialised strategies are needed to accomplish the same task for different types
- Static polymorphism: no virtual dispatch, everything resolved at compilation time

FUNCTION OVERLOADING

```
void copy(int* start, int* end, int* start2)
{
    for (; start != end; ++start, ++start2) {
        *start2 = *start;
    }
}
void copy(string* start, string* end,
                  string* start2)
{
    for (; start != end; ++start, ++start2) {
        *start2 = *start;
    }
}
void copy(double* start, double* end,
                  double* start2)
{
    for (; start != end; ++start, ++start2) {
        *start2 = *start;
    }
}
double a[10], b[10];
copy(a, a + 10, b);
```

- When specialised strategies are needed to accomplish the same task for different types
- Static polymorphism: no virtual dispatch, everything resolved at compilation time
- But sometimes we need the opposite: same operations to be performed on different kinds of input

INTRODUCTION TO C++ TEMPLATES

```
void copy(int* start, int* end, int* start2)
{
    for (; start != end; ++start, ++start2) {
        *start2 = *start;
    }
}
void copy(string* start, string* end,
                 string* start2)
{
    for (; start != end; ++start, ++start2) {
        *start2 = *start;
    }
}
void copy(double* start, double* end,
                 double* start2)
{
    for (; start != end; ++start, ++start2) {
        *start2 = *start;
    }
}
double a[10], b[10];
copy(a, a + 10, b);
```

Same operations on different types

- Exactly the same high level code
- Assigning a string to another may involve very different low level operations compared to assigning an integer to another. But once we have written our string class, we may write the exact same code for the string and integer versions of this kind of operations!
- Couldn't we automate the process of writing the 3 variants shown, by perhaps, using a placeholder type, and generating the right variant wherever required ?

INTRODUCTION TO C++ TEMPLATES

```
Dear compiler, in the following, T is a placeholder!
void copy(T* start, T* end, T* start2)
{
    for (; start != end; ++start, ++start2) {
        *start2 = *start;
    }
}
```

Wouldn't it be nice,

- if we could write the function in terms of some placeholder for the actual type ?

INTRODUCTION TO C++ TEMPLATES

```
Dear compiler, in the following, T is a placeholder!
void copy(T* start, T* end, T* start2)
{
    for (; start != end; ++start, ++start2) {
        *start2 = *start;
    }
}
```

```
double a[10], b[10];
copy<double>(a, a + 10, b);
string names[10], onames[10];
copy<string>(onames, onames + 10, names);
```

Wouldn't it be nice,

- if we could write the function in terms of some placeholder for the actual type ?
- and when we need to use the function, we indicate what to substitute in place of the placeholder ?

INTRODUCTION TO C++ TEMPLATES

```
template <class T>
void copy(T* start, T* end, T* start2)
{
    for (; start != end; ++start, ++start2) {
        *start2 = *start;
    }
}


double a[10], b[10];
copy<double>(a, a + 10, b);
string names[10], onames[10];
copy<string>(onames, onames + 10, names);
```

Wouldn't it be nice,

- if we could write the function in terms of some placeholder for the actual type ?
- and when we need to use the function, we indicate what to substitute in place of the placeholder ?
- For the first point : Sure!

INTRODUCTION TO C++ TEMPLATES

```
template <class T>
void copy(T* start, T* end, T* start2)
{
    for (; start != end; ++start, ++start2) {
        *start2 = *start;
    }
}


double a[10], b[10];
copy(a, a + 10, b);
string names[10], onames[10];
copy(onames, onames + 10, names);
```

Wouldn't it be nice,

- if we could write the function in terms of some placeholder for the actual type ?
- and when we need to use the function, we indicate what to substitute in place of the placeholder ?
- For the first point : Sure!
- For the second point: the compiler already knows those types based on the inputs at the point of usage!

INTRODUCTION TO C++ TEMPLATES

```
template <class T>
void copy(T* start, T* end, T* start2)
{
    for (; start != end; ++start, ++start2) {
        *start2 = *start;
    }
}


double a[10], b[10];
copy(a, a + 10, b);
string names[10], onames[10];
copy(onames, onames + 10, names);
```

Wouldn't it be nice,

- if we could write the function in terms of some placeholder for the actual type ?
- and when we need to use the function, we indicate what to substitute in place of the placeholder ?
- For the first point : Sure!
- For the second point: the compiler already knows those types based on the inputs at the point of usage!
- Test it! `examples/template_intro.cc`

INTRODUCTION TO C++ TEMPLATES

```
template <class T>
void copy(T* start, T* end, T* start2)
{
    for (; start != end; ++start, ++start2) {
        *start2 = *start;
    }
}


double a[10], b[10];
copy(a, a + 10, b);
string names[10], onames[10];
copy(onames, onames + 10, names);
```

Wouldn't it be nice,

- if we could write the function in terms of some placeholder for the actual type ?
- and when we need to use the function, we indicate what to substitute in place of the placeholder ?
- For the first point : Sure!
- For the second point: the compiler already knows those types based on the inputs at the point of usage!
- Test it! `examples/template_intro.cc`

Although we seemingly call a function we only wrote once, with different kinds of inputs, the compiler sees these as calls to two different functions. No runtime decision is needed to find the function to call.

TEMPLATES

Generic code The logic of the copy operation is quite simple. Given a pair of "iterators" (Behaviourally pointer like entities: can be advanced along a sequence, can be dereferenced) `first` and `last` in an input sequence, and a target location `result` in an output sequence, we want to:

- Loop over the input sequence
- For each position of the input iterator, copy the current element to the output iterator position
- Increment the input and output iterators
- Stop if the input iterator has reached `last`

A TEMPLATE FOR A GENERIC COPY OPERATION

```cpp
template <class InputIterator, class OutputIterator>
OutputIterator copy(InputIterator first, InputIterator last, OutputIterator result)
{
  while (first != last) *result++ = *first++;
  return result;
}
```

C++ template notation

- A template with which to generate code!
- If you had iterators to two kinds of sequences, you could substitute them in the above template and have a nice copy function!
- The compiler does the necessary substitution when you try to use the function
- The compiler needs access to the template source code at the point where it is trying to instantiate it!

ORDERED PAIRS

```
struct double_pair
{
    double first, second;
};
...
double_pair coords[100];
...
struct int_pair
{
    int first, second;
};
...
int_pair line_ranges[100];
...
struct int_double_pair
{
    // wait!
    // can I make a template out of it?
};
```

Class templates

- Classes can be templates too

ORDERED PAIRS

```
pair<double,double> coords[100];
pair<int,int> line_ranges[100];
pair<int,double> whatever;
```

`pair<int,double>`, after the template substitutions, becomes

```
struct pair<int, double>
{
    int first;
    double second;
};
```

Class templates

- Classes can be templates too
- Generated when the template is "instantiated"

```
template <class T, class U>
struct pair
{
    T first;
    U second;
};
```

ORDERED PAIRS

```
pair<double,double> coords[100];
pair<int,int> line_ranges[100];
pair<int,double> whatever;
```

`pair<int,double>`, after the template substitutions, becomes

```
struct pair<int, double>
{
    int first;
    double second;
};
```

Class templates

- Classes can be templates too
- Generated when the template is “instantiated”

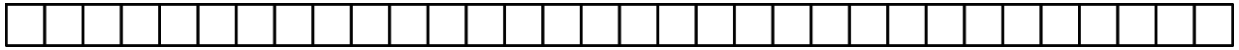
```
template <class T, class U>
struct pair
{
    T first;
    U second;
};
```

- Useful for creating many generic types

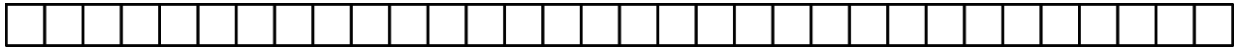
CLASS TEMPLATES YOU HAVE ALREADY SEEN...

- `std::vector<T>`, `std::array<T, N>`, `std::valarray<T>`, `std::map<K, V>`, `std::string` ...

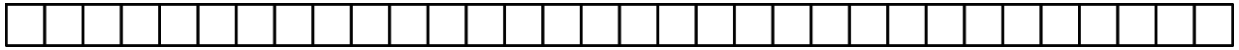
CLASS TEMPLATES YOU HAVE ALREADY SEEN...

- `std::vector<T>`, `std::array<T, N>`, `std::valarray<T>`, `std::map<K, V>`, `std::string` ...
- A vector means ... 

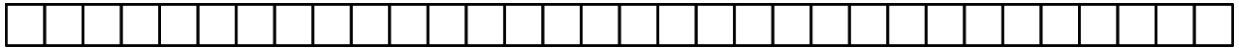
CLASS TEMPLATES YOU HAVE ALREADY SEEN...

- `std::vector<T>`, `std::array<T, N>`, `std::valarray<T>`, `std::map<K, V>`, `std::string` ...
- A vector means ... 
- The code required to write containers of `int`, `double`, `complex_number` or any other class type will only differ by the type of the elements

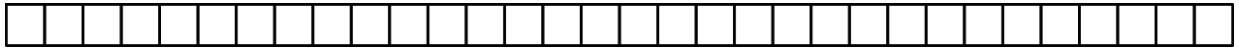
CLASS TEMPLATES YOU HAVE ALREADY SEEN...

- `std::vector<T>`, `std::array<T, N>`, `std::valarray<T>`, `std::map<K, V>`, `std::string` ...
- A vector means ... 
- The code required to write containers of `int`, `double`, `complex_number` or any other class type will only differ by the type of the elements
- The template captures the essential structure, and we don't need to separately develop, debug or test these parameterised types for every possible element type

CLASS TEMPLATES YOU HAVE ALREADY SEEN…

- `std::vector<T>`, `std::array<T, N>`, `std::valarray<T>`, `std::map<K, V>`, `std::string` …
- A vector means … 
- The code required to write containers of `int`, `double`, `complex_number` or any other class type will only differ by the type of the elements
- The template captures the essential structure, and we don't need to separately develop, debug or test these parameterised types for every possible element type
- No inheritance relationship between vectors of different types

CLASS TEMPLATES YOU HAVE ALREADY SEEN...

- `std::vector<T>`, `std::array<T, N>`, `std::valarray<T>`, `std::map<K, V>`, `std::string` ...
- A vector means ... 
- The code required to write containers of `int`, `double`, `complex_number` or any other class type will only differ by the type of the elements
- The template captures the essential structure, and we don't need to separately develop, debug or test these parameterised types for every possible element type
- No inheritance relationship between vectors of different types
- No inheritance relationship required between entities which can be vector elements

VARIABLE TEMPLATES

```cpp
template <class T> constexpr auto algocategory = 0;
template<> constexpr auto algocategory<int> = 1;
template<> constexpr auto algocategory<long> = 1;
template<> constexpr auto algocategory<int*> = 2;
template<> constexpr auto algocategory<long*> = 2;
template <class T>
auto proc(T x)
{
    if constexpr (algocategory<T> == 2) {
        std::cout << "Using method for category 2 \n";
    } else if constexpr (algocategory<T> == 1) {
        std::cout << "Using method for category 1 \n";
    } else {
        std::cout << "Using method for category 0 \n";
    }
}
```

```cpp
auto main() -> int
{
    int v{7};
    proc(1);
    proc(1.);
    proc(1L);
    proc(v);
    proc(&v);
}
```

- Can be a static data member of a class or a global variable parameterised by template parameters
- Can be used along with compile time if statements to decide between different algorithms

ONE CLASS TEMPLATE IN DETAIL

Initialiser list constructors

- The `darray` class we saw earlier in some examples represents a dynamic array, like the `std::vector`. It is a good example to illustrate more about class templates

ONE CLASS TEMPLATE IN DETAIL

Initialiser list constructors

- The `darray` class we saw earlier in some examples represents a dynamic array, like the `std::vector`. It is a good example to illustrate more about class templates
- We want to be able to initialise our `darray<T>` like this:

```
darray<double> D(400, 0.);
darray<string> S{"A", "B", "C"};
darray<int> I{1, 2, 3, 4, 5};
```

ONE CLASS TEMPLATE IN DETAIL

Initialiser list constructors

- The `darray` class we saw earlier in some examples represents a dynamic array, like the `std::vector`. It is a good example to illustrate more about class templates
- We want to be able to initialise our `darray<T>` like this:

```
darray<double> D(400, 0.);
darray<string> S{"A", "B", "C"};
darray<int> I{1, 2, 3, 4, 5};
```

- And then we want to be able to use it as follows...

```
for (auto i = 0UL; i < D.size(); ++i) {
    D[i] = i * i;
    std::cout << D[i] << "\n";
}
```

ONE CLASS TEMPLATE IN DETAIL

Initialiser list constructors

- Making it into a template and writing many of the special functions is easy.

```
template <class T>
class darray {
    std::unique_ptr<T[]> dat;
    size_t sz{};
public:
    darray() = default;
    ~darray() = default;
    darray(const darray& other);
    darray(darray&&) noexcept = default;
    darray& operator=(const darray& other);
    darray& operator=(darray&&) noexcept = default;
};
```

- Using the `unique_ptr` to manage the heap allocation/deallocation means we don't need to do anything special for default constructor, destructor and the move operations. Only copy needs to be carefully implemented!

ONE CLASS TEMPLATE IN DETAIL

Initialiser list constructors

- To initialise our `darray<T>` like this:

```
darray<string> S{"A", "B", "C"};
darray<int> I{1, 2, 3, 4, 5};
```

we need an initialiser_list constructor

```
darray(initializer_list<T> l) {
    arr = std::make_unique<T[]>(l.size());
    for (auto i{0UL}; auto&& el : l) arr[i++] = el;
}
```

A DYNAMIC ARRAY CLASS TEMPLATE

```
template <class T>
class darray {
public:
    auto operator[](size_t i) const
        -> const T& { return arr[i]; }
    auto operator[](size_t i) -> T& { return arr[i]; }
};
```

- Two versions of the `[]` operator for read-only and read/write access

A DYNAMIC ARRAY CLASS TEMPLATE

```
template <class T>
class darray {
public:
    auto operator[](size_t i) const
        -> const T& { return arr[i]; }
    auto operator[](size_t i) -> T& { return arr[i]; }
};
```

- Two versions of the `[]` operator for read-only and read/write access
- Use `const` qualifier in any member function which does not change the object

A DYNAMIC ARRAY CLASS TEMPLATE

```
template <class T>
class darray {
public:
    auto operator[](size_t i) const
        -> const T& { return arr[i]; }
    auto operator[](size_t i) -> T& { return arr[i]; }
};
```

- Two versions of the `[]` operator for read-only and read/write access
- Use `const` qualifier in any member function which does not change the object
- Both versions let us to use a `darray<int>` object, say, `D` with array style indexing, e.g., `D[5UL]`. The second is usable only when `D` is not declared `const`.

JÜLICH Forschungszentrum

TYPE DEDUCTIONS

- Template parameters can be type names or compile time constant values of different types.
- Until C++20, non-type template parameters were limited to integral types. Now, a lot of other types are allowed.

```
template <class T, int N>
struct my_array {
    T data[N];
};
```

TYPE DEDUCTIONS

- Template parameters can be type names or compile time constant values of different types.
- Until C++20, non-type template parameters were limited to integral types. Now, a lot of other types are allowed.
- Can be used to specify compile time constant sizes

```cpp
template <class T, int N>
struct my_array {
    T data[N];
};
```

```cpp
template <class T,
          int nrows, int ncols>
struct my_matrix {
    T data[nrows * ncols];
};
```

TYPE DEDUCTIONS

- Template parameters can be type names or compile time constant values of different types.
- Until C++20, non-type template parameters were limited to integral types. Now, a lot of other types are allowed.
- Can be used to specify compile time constant sizes
- but also give you a peculiar kind of " function " in effect
- Old uses of template integer arithmetic are by now obsolete. `constexpr` functions constitute a vastly superior alternative.
- But, type-deductions remain an important use for template meta-programs

```cpp
template <class T, int N>
struct my_array {
    T data[N];
};
```

```cpp
template <class T,
          int nrows, int ncols>
struct my_matrix {
    T data[nrows * ncols];
};
```

```cpp
template <int i, int j>
struct mult {
    static const int value=i*j;
};
...
my_array< mult<19,21>::value > vals;
```

EVALUATE DEPENDENT TYPES

- Suppose we want to implement a template function

```
template <class T> auto f(T a) -> U;
```

  such that when T is a non-pointer type, U should take the value T. But if T is itself a pointer, U is the type obtained by dereferencing the pointer

EVALUATE DEPENDENT TYPES

- Suppose we want to implement a template function

```
template <class T> auto f(T a) -> U;
```

  such that when T is a non-pointer type, U should take the value T. But if T is itself a pointer, U is the type obtained by dereferencing the pointer
- We could use a template function to "compute" the type U like this:

```
template <class T> struct remove_pointer { using type = T; };
template <class T> struct remove_pointer<T*> { using type = T; };
```

EVALUATE DEPENDENT TYPES

- Suppose we want to implement a template function

```
template <class T> auto f(T a) -> U;
```

  such that when T is a non-pointer type, U should take the value T. But if T is itself a pointer, U is the type obtained by dereferencing the pointer
- We could use a template function to "compute" the type U like this:

```
template <class T> struct remove_pointer { using type = T; };
template <class T> struct remove_pointer<T*> { using type = T; };
```

- We can then declare the function as:

```
template <class InputType>
auto f(InputType a) -> remove_pointer<InputType>::type ;
```

TYPE FUNCTIONS

- Compute properties of types
- Compute dependent types
- Typically used with convenient alias template declarations for the dependent type or the constant value

```
template <class T1, class T2>
std::is_same<T1,T2>::value

template <class T>
std::is_integral<T>::value

template <class T>
std::make_signed<T>::type

template <class T>
std::remove_reference<T>::type

template <class T>
using remove_reference_t =
    typename remove_reference<T>::type;

template <class T>
inline constexpr bool is_integral_v =
    std::is_integral<T>::value;
```

STATIC_ASSERT WITH TYPE TRAITS

```cpp
#include <iostream>
#include <type_traits>
template < class T, class U>
auto some_calc(T x, U y)
{
  static_assert(std::is_convertible_v<T, U>,
  "The type of the argument x must be convertible to type U");
// ...
}
auto main() -> int
{
  some_calc(4.0, "target"); //Compiler error!
  ...
}
```

- `static_assert(condition[, message])` asks the compiler to check if `condition` is valid. If not, `message` is printed as a compiler error.
- Use `static_assert` and `type_traits` in combination with `constexpr`, to test assumptions verifiable by the compiler.

Exercise 3.1: static_assert2.cc

TYPETRAITS

Unary predicates

- `is_integral_v<T>` : `T` is an integer type
- `is_const_v<T>` : has a `const` qualifier
- `is_class_v<T>` : struct or class
- `is_pointer_v<T>` : Pointer type
- `is_abstract_v<T>` : Abstract class with at least one pure virtual function
- `is_copy_constructible_v<T>` : Class allows copy construction
- `is_same_v<T1,T2>` : `T1` and `T2` are the same types
- `is_base_of_v<T,D>` : `T` is base class of `D`
- `is_convertible_v<T,T2>` : `T` is convertible to `T2`

FORWARDING REFERENCES

```cpp
template <class T>
auto wrapperfunc( T&&  t)
{
    other(std::forward<T>(t));
}
auto main() -> int
{
    std::string x{"Solar"};
    std::string y{"System"};
    wrapperfunc(x);
    wrapperfunc(x + " " + y);
}
```

- Function argument written as if it were an R-value reference to a cv-unqualified template parameter
- If `wrapperfunc` is called with a constant L-value, `T` is deduced to be a constant L-value reference, and `other` receives a constant L-value reference
- If `wrapperfunc` is called with an L-value, `T` is deduced to be an L-value reference, and `other` receives an L-value reference
- If the input is an R-value, then `T` is inferred to be a plain type, and `forward` ensures that `other` receives an R-value reference

FORWARDING REFERENCES

```cpp
template <class T>
auto wrapperfunc( T&&  t)
{
    other(std::forward<T>(t));
}
auto main() -> int
{
    std::string x{"Solar"};
    std::string y{"System"};
    wrapperfunc(x);
    wrapperfunc(x + " " + y);
}
```

- Type deduction for variables declared with `auto&&` follows the same rules. The declared variable is a `const` L-value reference, mutable L-value reference or an R-value reference depending on the initialiser.

- Function argument written as if it were an R-value reference to a cv-unqualified template parameter
- If `wrapperfunc` is called with a constant L-value, `T` is deduced to be a constant L-value reference, and `other` receives a constant L-value reference
- If `wrapperfunc` is called with an L-value, `T` is deduced to be an L-value reference, and `other` receives an L-value reference
- If the input is an R-value, then `T` is inferred to be a plain type, and `forward` ensures that `other` receives an R-value reference

NOT A TEXT SUBSTITUTION ENGINE!

Template specialisation

```cpp
template <class T>
class vector {
  // implementation of a general
  // vector for any type T
};
template <>
class vector<bool> {
  // Store the true false values
  // in a compressed way, i.e.,
  // 32 of them in a single int
};
void somewhere_else()
{
  vector<bool> A;
  // Uses the special implementation
}
```

- Templates are defined to work with generic template parameters
- But special values of those parameters, which should be treated differently, can be specified using "template specialisations" as shown
- If a matching specialisation is found, it is preferred over the general template

```cpp
template <class A, class B>
constexpr auto are_same = false;
template <class A>
constexpr auto are_same<A, A> = true;
static_assert(are_same<int, long>); // Fails
using Integer = int;
static_assert(are_same<int, Integer>); // Passes
```

NOT A TEXT SUBSTITUTION ENGINE!

Recursion and integer arithmetic

```
template <unsigned N> constexpr unsigned fact = N * fact<N-1>;
template <> constexpr unsigned fact<0> = 1U;
static_assert(fact<7> == 5040)
```

- Templates support recursive instantiation
- Combined with specialisation to terminate recursion
- Recursion and specialisation can be used to emulate “loop” like calculations via tail-recursion

Exercise 3.2:

The example source file `examples/no_textsub.cc` demonstrates recursion and specialisation in templates, and uses `static_assert` to verify that the arithmetic calculations in this case indeed happen during compilation.

NOT A TEXT SUBSTITUTION ENGINE!

Because: SFINAE

```
template <bool Cond, class T> struct enable_if {};
template <class T> struct enable_if<true, T> { using type = T; }
template <class T>
auto func(T x) -> enable_if<sizeof(T) == 8UL, T>::type {
//impl1
}
template <class T>
auto func(T x) -> enable_if<sizeof(T) != 8UL, T>::type {
//impl2
}
```

- **S**ubstitution **F**ailure **I**s **N**ot **A**n **E**rror
- If substituting a template parameter results in incomplete or invalid function declarations, that overload is ignored.
- The compiler simply tries to find another template with the same name that might match
- If it can't find any, then you have an error
- `func(1)` will resolve to the second version and `func(1.0)` will resolve to the first version during compilation!

NOT A TEXT SUBSTITUTION ENGINE!

Because: concepts

```
template <class T>
auto func(T x) -> T requires (sizeof(T) == 8UL) {
//impl1
}
template <class T>
auto func(T x) -> T requires (sizeof(T) != 8UL) {
//impl2
}
```

- Different implementations can be provided requiring different properties of the input type
- Before C++20, this sort of selection was done using `std::enable_if`. Now, `concepts` provide a far cleaner alternative.
- Again, `func(1)` will resolve to the second version and `func(1.0)` will resolve to the first version during compilation!

Constrained templates

CONSTRAINED TEMPLATES

- We created overloaded functions so that different strategies can be employed for different input types

```
auto power(double x, double y) -> double ;
auto power(double x, int i) -> double ;
```

CONSTRAINED TEMPLATES

- We created overloaded functions so that different strategies can be employed for different input types

```
auto power(double x, double y) -> double ;
auto power(double x, int i) -> double ;
```

- We have function templates, so that the same strategy can be applied to different types, e.g.,

```
template <class T> auto power(double x, T i) -> double ;
```

CONSTRAINED TEMPLATES

- We created overloaded functions so that different strategies can be employed for different input types

```
auto power(double x, double y) -> double ;
auto power(double x, int i) -> double ;
```

- We have function templates, so that the same strategy can be applied to different types, e.g.,

```
template <class T> auto power(double x, T i) -> double ;
```

- Can we combine the two, so that we have two function templates, both looking like the above, but one is automatically selected whenever `T` is an integral type and the other whenever `T` is a floating point type ?

CONSTRAINED TEMPLATES

- We created overloaded functions so that different strategies can be employed for different input types

```
auto power(double x, double y) -> double ;
auto power(double x, int i) -> double ;
```

- We have function templates, so that the same strategy can be applied to different types, e.g.,

```
template <class T> auto power(double x, T i) -> double ;
```

- Can we combine the two, so that we have two function templates, both looking like the above, but one is automatically selected whenever `T` is an integral type and the other whenever `T` is a floating point type ?
- Some way to impose requirements on permissible matches for the template parameters. Something like:

```
template <class T> auto power(double x, T i) -> double requires floating_point<T>;
template <class T> auto power(double x, T i) -> double requires integer<T>;
```

CONSTRAINED TEMPLATES

- We created overloaded functions so that different strategies can be employed for different input types

```
auto power(double x, double y) -> double ;
auto power(double x, int i) -> double ;
```

- We have function templates, so that the same strategy can be applied to different types, e.g.,

```
template <class T> auto power(double x, T i) -> double ;
```

- Can we combine the two, so that we have two function templates, both looking like the above, but one is automatically selected whenever `T` is an integral type and the other whenever `T` is a floating point type ?
- Some way to impose requirements on permissible matches for the template parameters. Something like:

```
template <class T> auto power(double x, T i) -> double requires floating_point<T>;
template <class T> auto power(double x, T i) -> double requires integer<T>;
```

- If we could do that, we can combine the generality of templates with the selectiveness of function overloading

CONSTRAINED TEMPLATES

- We created overloaded functions so that different strategies can be employed for different input types

```
auto power(double x, double y) -> double ;
auto power(double x, int i) -> double ;
```

- We have function templates, so that the same strategy can be applied to different types, e.g.,

```
template <class T> auto power(double x, T i) -> double ;
```

- Can we combine the two, so that we have two function templates, both looking like the above, but one is automatically selected whenever `T` is an integral type and the other whenever `T` is a floating point type ?
- Some way to impose requirements on permissible matches for the template parameters. Something like:

```
template <class T> auto power(double x, T i) -> double requires floating_point<T>;
template <class T> auto power(double x, T i) -> double requires integer<T>;
```

- If we could do that, we can combine the generality of templates with the selectiveness of function overloading
- We can

CONCEPTS

Named requirements on template parameters

- `concept` s are named requirements on template parameters, such as `floating_point`, `contiguous_range`
- If `MyAPI` is a `concept`, and `T` is a type, `MyAPI<T>` evaluates at compile time to either true or false.
- Concepts can be combined using conjunctions (`&&`) and disjunctions (`||`) to make other concepts.
- A `requires` clause introduces a constraint on a template type

A suitably designed set of concepts can greatly improve readability of template code

CREATING CONCEPTS

```
template <template-pars>
concept conceptname = constraint_expr;
```

```
template <class T>
concept Integer = std::is_integral_v<T>;
template <class D, class B>
concept Derived = std::is_base_of<B, D>;

class Counters;
template <class T>
concept Integer_ish = Integer<T> ||
                      Derived<T,Counters>;
```

- Out of a simple `type_traits` style boolean expression
- Combine with logical operators to create more complex requirements
- The `requires` *expression* allows creation of syntactic requirements as shown in the last two examples.

CREATING CONCEPTS

```
template <template-pars>
concept conceptname = constraint_expr;
```

```
template <class T>
concept Integer = std::is_integral_v<T>;
template <class D, class B>
concept Derived = std::is_base_of<B, D>;
```

```
class Counters;
template <class T>
concept Integer_ish = Integer<T> ||
                      Derived<T,Counters>;
```

- Out of a simple `type_traits` style boolean expression
- Combine with logical operators to create more complex requirements
- The `requires` *expression* allows creation of syntactic requirements as shown in the last two examples.

CREATING CONCEPTS

```
template <template-pars>
concept conceptname = constraint_expr;
```

```
template <class T>
concept Integer = std::is_integral_v<T>;
template <class D, class B>
concept Derived = std::is_base_of<B, D>;

class Counters;
template <class T>
concept Integer_ish = Integer<T> ||
                      Derived<T,Counters>;
```

```
template <class T>
concept Addable = requires (T a, T b) {
    { a + b };
};
template <class T>
concept Indexable = requires(T A) {
    { A[0UL] };
};
```

- Out of a simple `type_traits` style boolean expression
- Combine with logical operators to create more complex requirements
- The `requires` *expression* allows creation of syntactic requirements as shown in the last two examples.

JÜLICH Forschungszentrum

CREATING CONCEPTS

```
template <template-pars>
concept conceptname = constraint_expr;
```

```
template <class T>
concept Integer = std::is_integral_v<T>;
template <class D, class B>
concept Derived = std::is_base_of<B, D>;

class Counters;
template <class T>
concept Integer_ish = Integer<T> ||
                      Derived<T,Counters>;
```

```
template <class T>
concept Addable = requires (T a, T b) {
    { a + b };
};
template <class T>
concept Indexable = requires(T A) {
    { A[0UL] };
};
```

- Out of a simple `type_traits` style boolean expression
- Combine with logical operators to create more complex requirements
- The `requires` *expression* allows creation of syntactic requirements as shown in the last two examples.
- The requires expression can contain a parameter list and a brace enclosed sequence of requirements, which can be:
  - type requirements, e.g., typename T::value_type;
  - simple requirements as shown on the left
  - compound requirements with optional return type constraints, e.g.,

    `{ A[0UL] } -> convertible_to<int>;`

USING CONCEPTS

```cpp
template <class T>
requires Integer_ish<T>
auto categ0(T&& i, double x) -> T;

template <class T>
auto categ1(T&& i, double x) -> T
    requires Integer_ish<T>;

template <Integer_ish T>
auto categ2(T&& i, double x) -> T;

void erase(Integer_ish auto&& i)
```

To constrain template parameters, one can

- place a `requires` clause immediately after the template parameter list
- place a `requires` clause after the function parameter parentheses
- Use the `concept` name in place of `class` or `typename` in the template parameter list
- Use `ConceptName auto` in the function parameter list

USING CONCEPTS

```
template <class T>
requires Integer_ish<T>
auto categ0(T&& i, double x) -> T;
template <class T>
auto categ1(T&& i, double x) -> T
    requires Integer_ish<T>;
template <Integer_ish T>
auto categ2(T&& i, double x) -> T;

void erase(Integer_ish auto&& i)
```

To constrain template parameters, one can

- place a `requires` clause immediately after the template parameter list
- place a `requires` clause after the function parameter parentheses
- Use the `concept` name in place of `class` or `typename` in the template parameter list
- Use `ConceptName auto` in the function parameter list

USING CONCEPTS

```
template <class T>
requires Integer_ish<T>
auto categ0(T&& i, double x) -> T;

template <class T>
auto categ1(T&& i, double x) -> T
    requires Integer_ish<T>;

template <Integer_ish T>
auto categ2(T&& i, double x) -> T;

void erase(Integer_ish auto&& i)
```

To constrain template parameters, one can

- place a `requires` clause immediately after the template parameter list
- place a `requires` clause after the function parameter parentheses
- Use the `concept` name in place of `class` or `typename` in the template parameter list
- Use `ConceptName auto` in the function parameter list

USING CONCEPTS

```
template <class T>
requires Integer_ish<T>
auto categ0(T&& i, double x) -> T;

template <class T>
auto categ1(T&& i, double x) -> T
    requires Integer_ish<T>;

template <Integer_ish T>
auto categ2(T&& i, double x) -> T;

void erase(Integer_ish auto&& i)
```

To constrain template parameters, one can

- place a `requires` clause immediately after the template parameter list
- place a `requires` clause after the function parameter parentheses
- Use the `concept` name in place of `class` or `typename` in the template parameter list
- Use `ConceptName auto` in the function parameter list

USING CONCEPTS

```
template <class T>
requires Integer_ish<T>
auto categ0(T&& i, double x) -> T;

template <class T>
auto categ1(T&& i, double x) -> T
    requires Integer_ish<T>;

template <Integer_ish T>
auto categ2(T&& i, double x) -> T;

void erase(Integer_ish auto&& i)
```

To constrain template parameters, one can

- place a `requires` clause immediately after the template parameter list
- place a `requires` clause after the function parameter parentheses
- Use the `concept` name in place of `class` or `typename` in the template parameter list
- Use `ConceptName auto` in the function parameter list

DECLARING FUNCTION INPUT PARAMETERS WITH AUTO

```
template <class T>
auto sqr(const T& x) { return x * x; }
```

- Because of syntax introduced for functions with constrained templates in C++20, we have a new way to write fully unconstrained function templates...

DECLARING FUNCTION INPUT PARAMETERS WITH AUTO

```
auto sqr(const auto& x) { return x * x; }
```

- Because of syntax introduced for functions with constrained templates in C++20, we have a new way to write fully unconstrained function templates...
- Functions with `auto` in their parameter list are implicitly function templates

DECLARING FUNCTION INPUT PARAMETERS WITH AUTO

```cpp

auto sqr(const auto& x) { return x * x; }
```

- Because of syntax introduced for functions with constrained templates in C++20, we have a new way to write fully unconstrained function templates...
- Functions with `auto` in their parameter list are implicitly function templates

Exercise 3.7:

The program `examples/gcd_w_concepts.cc` shows a very small concept definition and its use in a function calculating the greatest common divisor of two integers.

Exercise 3.8:

The series of programs `examples/generic_func1.cc` through `generic_func4.cc` shows some trivial functions implemented with templates with and without constraints. The files contain plenty of comments explaining the rationale and use of concepts.

OVERLOADING BASED ON CONCEPTS

```cpp
template <class N>
concept Number = std::floating_point<N>
                 or std::integral<N>;


void proc(Number auto&& x) {
    std::cout << "Called proc for numbers";
}



auto main() -> int {
    proc(-1);
    proc(88UL);
    proc("0118 999 88199 9119725   3");
    proc(3.141);
    proc("eighty"s);
}
```

- Constraints on template parameters are not just “documentation” or decoration.

OVERLOADING BASED ON CONCEPTS

```
template <class N>
concept Number = std::floating_point<N>
                 or std::integral<N>;
template <class N>
concept NotNumber = not Number<N>;
void proc(Number auto&& x) {
    std::cout << "Called proc for numbers";
}
void proc(NotNumber auto&& x) {
    std::cout << "Called proc for non-numbers";
}
auto main() -> int {
    proc(-1);
    proc(88UL);
    proc("0118 999 88199 9119725   3");
    proc(3.141);
    proc("eighty"s);
}
```

- Constraints on template parameters are not just “documentation” or decoration.
- The compiler can choose between different versions of a function based on concepts

OVERLOADING BASED ON CONCEPTS

```
template <class N>
concept Number = std::floating_point<N>
                 or std::integral<N>;
template <class N>
concept NotNumber = not Number<N>;
void proc(Number auto&& x) {
    std::cout << "Called proc for numbers";
}
void proc(NotNumber auto&& x) {
    std::cout << "Called proc for non-numbers";
}
auto main() -> int {
    proc(-1);
    proc(88UL);
    proc("0118 999 88199 9119725   3");
    proc(3.141);
    proc("eighty"s);
}
```

- Constraints on template parameters are not just "documentation" or decoration.
- The compiler can choose between different versions of a function based on concepts
- The version of a function chosen depends on *properties* of the input types, rather than their identities. "It's not who you are underneath, it's what you (can) do that defines you."

OVERLOADING BASED ON CONCEPTS

```cpp
template <class N>
concept Number = std::floating_point<N>
                 or std::integral<N>;
template <class N>
concept NotNumber = not Number<N>;
void proc(Number auto&& x) {
    std::cout << "Called proc for numbers";
}
void proc(NotNumber auto&& x) {
    std::cout << "Called proc for non-numbers";
}
auto main() -> int {
    proc(-1);
    proc(88UL);
    proc("0118 999 88199 9119725   3");
    proc(3.141);
    proc("eighty"s);
}
```

- Constraints on template parameters are not just "documentation" or decoration.
- The compiler can choose between different versions of a function based on concepts
- The version of a function chosen depends on *properties* of the input types, rather than their identities. "It's not who you are underneath, it's what you (can) do that defines you."
- During overload resolution, in case multiple matches are found, the most constrained overload is chosen.

JÜLICH Forschungszentrum

OVERLOADING BASED ON CONCEPTS

```
template <class N>
concept Number = std::floating_point<N>
                 or std::integral<N>;
template <class N>
concept NotNumber = not Number<N>;
void proc(Number auto&& x) {
    std::cout << "Called proc for numbers";
}
void proc(NotNumber auto&& x) {
    std::cout << "Called proc for non-numbers";
}
auto main() -> int {
    proc(-1);
    proc(88UL);
    proc("0118 999 88199 9119725   3");
    proc(3.141);
    proc("eighty"s);
}
```

- Constraints on template parameters are not just "documentation" or decoration.
- The compiler can choose between different versions of a function based on concepts
- The version of a function chosen depends on *properties* of the input types, rather than their identities. "It's not who you are underneath, it's what you (can) do that defines you."
- During overload resolution, in case multiple matches are found, the most constrained overload is chosen.
- Not based on any inheritance relationships

OVERLOADING BASED ON CONCEPTS

```cpp
template <class N>
concept Number = std::floating_point<N>
                 or std::integral<N>;
template <class N>
concept NotNumber = not Number<N>;
void proc(Number auto&& x) {
    std::cout << "Called proc for numbers";
}
void proc(NotNumber auto&& x) {
    std::cout << "Called proc for non-numbers";
}
auto main() -> int {
    proc(-1);
    proc(88UL);
    proc("0118 999 88199 9119725   3");
    proc(3.141);
    proc("eighty"s);
}
```

- Constraints on template parameters are not just “documentation” or decoration.
- The compiler can choose between different versions of a function based on concepts
- The version of a function chosen depends on *properties* of the input types, rather than their identities. “It’s not who you are underneath, it’s what you (can) do that defines you.”
- During overload resolution, in case multiple matches are found, the most constrained overload is chosen.
- Not based on any inheritance relationships
- Not a “quack like a duck, or bust” approach either.

OVERLOADING BASED ON CONCEPTS

```
template <class N>
concept Number = std::floating_point<N>
                 or std::integral<N>;
template <class N>
concept NotNumber = not Number<N>;
void proc(Number auto&& x) {
    std::cout << "Called proc for numbers";
}
void proc(NotNumber auto&& x) {
    std::cout << "Called proc for non-numbers";
}
auto main() -> int {
    proc(-1);
    proc(88UL);
    proc("0118 999 88199 9119725   3");
    proc(3.141);
    proc("eighty"s);
}
```

- Constraints on template parameters are not just "documentation" or decoration.
- The compiler can choose between different versions of a function based on concepts
- The version of a function chosen depends on *properties* of the input types, rather than their identities. "It's not who you are underneath, it's what you (can) do that defines you."
- During overload resolution, in case multiple matches are found, the most constrained overload is chosen.
- Not based on any inheritance relationships
- Not a "quack like a duck, or bust" approach either.
- Entirely compile time mechanism

Exercise 3.9:

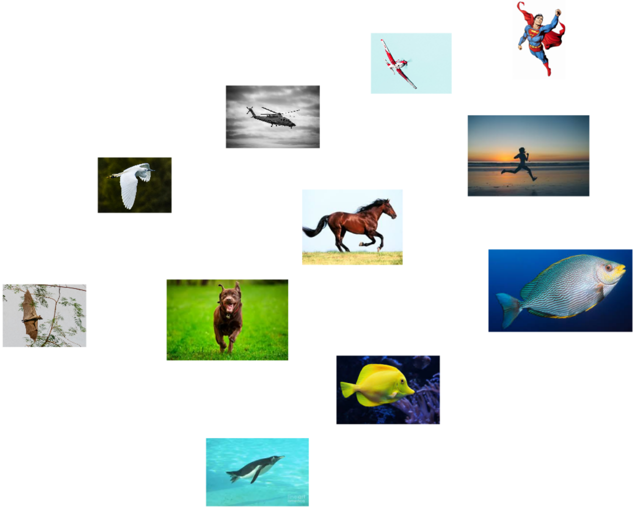
Check how you can use concepts to implement alternative versions of a function based on properties of the input parameters! The program `examples/overload_w_concepts.cc` contains the code just shown. Can you add another overload that is picked if the input type is an array? This means, if `X` is the input parameter, `X[i]` is syntactically valid for unsigned integer `i`. The array version should be picked up if the input is a `vector`, `array`, etc. , but also `string`. How would you prevent the `string` and C-style strings picking the array version?

PREDEFINED USEFUL CONCEPTS

Many concepts useful in building our own concepts are available in the standard library header `<concepts>`.

- `same_as`
- `convertible_to`
- `signed_ingegral`, `unsigned_integral`
- `floating_point`
- `assignable_from`
- `swappable`, `swappable_with`
- `derived_from`
- `move_constructible`, `copy_constructible`
- `invocable`
- `predicate`
- `relation`

CONCEPTS: SUMMARY



f(those who can fly)

f(runners)

f(swimmers)

Variadic templates

VARIADIC TEMPLATES

```
template <class ... Args>
auto countArgs(Args ... args) -> int
{
  return (sizeof ...args);
}
// elsewhere...
std::cout << "Num args = " << countArgs(1, "one", "ein", "uno", 3.232) << '\n';
```

- Another type of abstraction which allows us to code concretely with an arbitrary number of arguments to a function or class template
- Most common example: `std::tuple`. Standard tuple is defined as a class template, but instead of one, two or any finite number of template parameters, it is defined in terms of an arbitrary number of such parameters, so that we can have `std::tuple<int, double>`, `std::tuple<int, int, double, int>`, `std::tuple<std::string, int, double>` all using one class template.
- Recursion, partial specialisation, fold expressions
- **The `...` is actual code! Not blanks for you to fill in!**

JÜLICH Forschungszentrum

PARAMETER PACK

- The ellipsis (`...`) template argument is called a parameter pack [1]
- It represents 0 or more arguments which could be type names, integers or other templates :

```
template <class ... Args> class mytuple;
// The above can be instantiated with :
mytuple<int, int, double, string> t1;
mytuple<int> t2;
mytuple<> t3;
```

- Definition: A template with at least one parameter pack is called a variadic template

[1] https://en.cppreference.com/w/cpp/language/parameter_pack

PARAMETER PACK

```
//examples/variadic_1.cc
template <class ... Types> void f(Types ... args);
template <class Type1, class ... Types> void f(Type1 arg1, Types ... rest) {
  std::cout << typeid(arg1).name() << ``: '' << arg1 << ``\n'';
  f(rest ...);
}
template <> void f() {}
auto main() -> int
{
  int i{3}, j{};
  const char * cst{"abc"};
  std::string cppst{"def"};
  f(i, j, true, k, l, cst, cppst);
}
```

One way to handle variadic argument list:

- Divide argument list into first and rest
- Do something with first and recursively call template with rest
- Specialise for the case with 1 or 0 arguments

PARAMETER PACK EXPANSION

- `pattern ...` is called a parameter pack expansion
- It applies a pattern to a comma separated list of instantiations of the pattern
- If we are in a function :

```
template <class ... Types> void g(Types ... args)
```

- `args...` means the list of arguments used for the function.
- Calling `f(args ...)` in `g` will call `f` with same arguments
- Calling `f(h(args)...)` in `g` will call `f` with an argument list generated by applying function `h` to each argument of `g`
- In `g(true, "abc", 1)`,
  `f(h(args)...)` means `f(h(true), h("abc"), h(1))`

PARAMETER PACK EXPANSION

```cpp
template <class ... Types> void f(Types ... args);
template <class Type1, class ... Types> void f(Type1 arg1, Types ... rest) {
    std::cout <<" The first argument is " << arg1
              <<". Remainder argument list has " << sizeof...(Types) << " elements.\n";
    f(rest ...);
}
template <> void f() {}
template <class ... Types> void g(Types ... args) {
    std::cout << "Inside g: going to call function f with the sizes of "
              << "my arguments\n";
    f(sizeof(args)...);
}
```

- `sizeof...(Types)` retrieves the number of arguments in the parameter pack
- In `g` above, we call `f` with the sizes of each of the parameters passed to `g`
- Similarly, one can generate all addresses as `&args...`, increment all with `++args...` (examples `variadic_2.cc` and `variadic_3.cc`)

JÜLICH Forschungszentrum

PARAMETER PACK EXPANSION: WHERE

```cpp
template <class ... Types> void f( Types& ... args ) {}
template <class ... Types> void h( Types ... args ) {
    f( std::cout << args << ``\t'' ... );
    [=, &args ... ]{ return g( args... ); }();
    int t[sizeof...(args)]={ args ... };
    int s = 0;
    for (auto i : t) s += i;
    std::cout << "\nsum = " << s << "\n";
}
```

- Parameter pack expansion can be done in function parameter list , function argument list , template parameter list or template argument list
- Braced initializer lists
- Base specifiers and member initializer lists in classes
- Lambda captures

Exercise 3.10: Parameter packs

Study the examples `variadic_1.cc`, `variadic_2.cc` and `variadic_3.cc`. See where parameter packs are begin expanded, and make yourself familiar with this syntax.

FOLD EXPRESSIONS IN C++17

- Fold expressions provide a different compact way of expressing operations to be done on variadic argument lists
- Idea: apply a pattern to the argument list

```
#include <iostream>
template <class ... Args>
auto addup(Args ... args)
{
    return (1 + ... + args);
}
auto main() -> int
{
    std::cout << addup(1, 2, 3) << "\n";
    std::cout << addup(1, 2, 3, 4, 5) << "\n";
}
```

- `... op ppack` translates to reduce from the left with operator `op`
- `ppack op ...` means, reduce from the right with `op`
- `init op ... op ppack` reduces from the left, with initial value `init`
- `pack op ... op init` reduces from the right ...

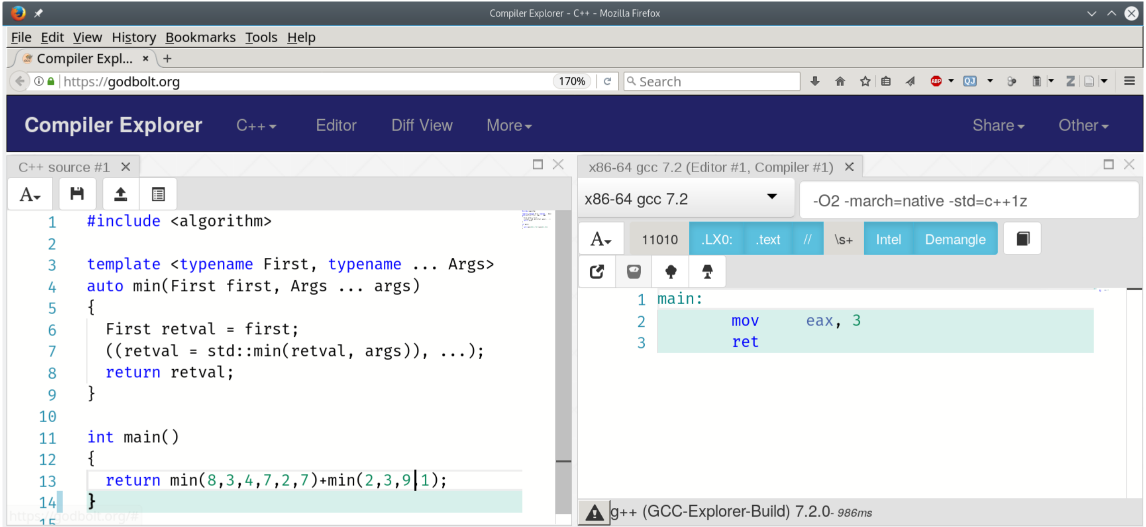
FOLD EXPRESSIONS

```cpp
// examples/foldex_3.cc
#include <algorithm>
template <class First, class ... Args>
auto min(First first, Args ... args)
{
    First retval = first;
    ((retval = std::min(retval, args)), ...);
    return retval;
}

auto main() -> int
{
    return min(8, 3, 4, 7, 2, 7)
        + min(2, 3, 9, 1);
}
```

Application

- Fold expression with the "comma operator"[a]
- Make even the number of arguments abstract

[a] In C++, if a comma separated list of expressions does not constitute an argument list for a function or an initialiser list, it becomes an expression with pairs connected by the "comma" operator. The comma operator simply evaluates both sides, left followed by right, and returns the value of the right hand side. The comma operator has the lowest precedence among operators. https://en.cppreference.com/w/cpp/language/operator_precedence



Note: Variadic templates and fold expressions belong to the compile time abstractions. Using these does not involve additional runtime indirection.

TUPLES

```cpp
#include <tuple>
#include <iostream>
auto main() -> int
{
    std::tuple<int, int, std::string> name_i_j{0, 1, "Uralic"};
    auto t3 = std::make_tuple<int, bool>(2, false);
    auto t4 = std::tuple_cat(name_i_j, t3);
    std::cout << std::get<2>(t4) << '\n';
}
```

- Like `std::pair`, but with arbitrary number of members
- "Structure templates without names"
- Accessor "function templates" `std::get<index>()` with `index` starting at `0`.
- No `operator[]` to access different components, since, for tuple `t`, components `t[0]`, `t[1]` would in general be of different types, which would be unexpected, counter-intuitive behaviour.
- Supports relational operators for lexicographical comparisons
- `tuple_cat(args ...)` concatenates tuples.

TUPLES

```cpp
auto f() -> std::tuple<int, int, string>; // elsewhere
auto main() -> int
{
    int i1;
    std::string name;
    std::tie(i1, std::ignore, name) = f();
}
```

- `tie(args ...)` "extracts a tuple" into pre-existing named variables.
- Some fields may be ignored during extraction using `std::ignore` as shown

PRINTING A TUPLE

```
template <class... Args>
auto operator<<(std::ostream& strm, const std::tuple<Args...>& t) -> std::ostream& {
    using namespace std;
    auto print_one = [&strm](const auto& onearg) -> decltype(strm) {
        using bare_type = remove_cvref_t<decltype(onearg)>;
        if constexpr (is_convertible_v<bare_type, string>)
            strm << quoted(onearg);
        else
            strm << onearg;
        return strm;
    };
    auto print_components = [&](auto&&... args) {
        size_t n {};
        ((print_one(args) << ((++n != sizeof...(args)) ? ", " : "")), ...);
    };
    strm << "[ ";
    apply(print_components, t);
    return strm << " ]";
}
```

- Helper lambda to print one element, quoted when it is a string, plain otherwise
- Second helper lambda, with variadic parameter list to handle component separation
- Fold expression and `std::apply` to print components

Exercise 3.11:

Three ways of printing a tuple is demonstrated in `print_tuple.cc` , `print_tuple_cxx17.cc` and `print_tuple_foldex.cc` .

FUN WITH FOLD EXPRESSIONS

Problem: We have an uncertain number of containers of arbitrary types (some vectors, some linked lists, ...), with arbitrary element types which are known to be `<` comparable (some contain `doubles`, some `std::string`s, some `int` ...), containing an arbitrary number of elements each. We need a tuple consisting of the largest element of each container. Write a function which will create that tuple from our inputs.

FUN WITH FOLD EXPRESSIONS

Problem: We have an uncertain number of containers of arbitrary types (some vectors, some linked lists, ...), with arbitrary element types which are known to be `<` comparable (some contain `doubles`, some `std::string`s, some `int` ...), containing an arbitrary number of elements each. We need a tuple consisting of the largest element of each container. Write a function which will create that tuple from our inputs.

Complete solution:

```
auto max_of_multiple(auto&& ... containers)
{
    return std::make_tuple(std::ranges::max(containers) ...);
}
```

FUN WITH FOLD EXPRESSIONS

Problem: We need a function to replace each element of a vector with the averages of neighbours separated by some shifts. Write a function that takes the vector and the shifts as function arguments, and returns the smoothed vector. It should be possible to use the function for any given number of shifts.

Complete solution:

```cpp
auto conv(const std::vector<double>& inp, auto ... shift)
{
    std::vector<double> out(inp.size(), 0.);
    auto res_exp = std::views::iota(0, static_cast<int>(inp.size()))
        | std::views::transform([inp, shift...](auto index){
            auto S = inp.size();
            return (inp[
            (index + shift) > 0 ? (index + shift) % S : S + (index + shift) % S
                       ] + ...)
                    / (sizeof ... (shift));
        });

    std::ranges::copy(res_exp, out.begin());
    return out;
}
```

Exercise 3.12:

`fold_xpr_demo[2-4].cc` demonstrate the last few applications of variadic templates, fold expressions and the new C++20 syntax for `auto` in function parameters. Build them with the proper include paths for printing tuples and ranges. The necessary headers for this functionality is in the `include` folder for the course.

Chapter 4

SOLID principles

DESIGN GOALS

- Correctness
- Readability
- Extendability
- Speed
- Adaptability

A large scale software project is better off being built out of components which are resilient to unforeseen changes.

DEPENDENCIES

- Impede modifications
- Hamper testing
- Increase rebuild times

- Good design helps us control dependencies.
- Variation points
- Flexible adaptible software

Guideline: Keep dependencies among software components to a minimum.

ENCAPSULATION

- Member functions abstracting properties
- Resilient to internal data reorganisation
- More flexible design

```cpp
class complex_number {
public:
    double real, imag;
    double modulus();
};
```

```cpp
class complex_number {
public:
    auto real() const -> double;
    auto imag() const -> double;
    void real(double x);
    void imag(double x);
    auto modulus() const -> double;
};
```

ENCAPSULATION

- Scott Meyer: degree of encapsulation is gauged by the number of things which break if the internal design changes
- Less member functions : better!
- If a function can be implemented as a non-friend, non-member function, it should be.

```cpp
// Class definition: bare essentials
namespace ns {
class Example {
public:
    auto property1() const -> double;
    auto property2() const -> double;
};
}
```

```cpp
// Use case 1 header
namespace ns {
auto calc(Example & ex) {
  //ex.property1() + ...
  //ex.property2();
  return haha;
}
}
```

THE SOLID PRINCIPLES

- **S**ingle responsibility principle
- **O**pen-closed principle
- **L**iskov's substitution principle
- **I**nterface segregation principle
- **D**ependency inversion principle

SRP: THE SINGLE RESPONSIBILITY PRINCIPLE

Every class should have a single responsibility and that responsibility should be entirely encapsulated by that class.

- However tempting it might seem, avoid adding member functions not related to the core idea of the class
- Related principle: Don't repeat yourself. Avoids opportunity for bugs and reduces maintenance overhead.

```cpp
class Rectangle {
public:
    auto area() const -> double;
    auto width() const -> double;
    auto height() const -> double;
    void width(double x);
    void height(double x);
    void draw() const;
};
```

OCP: THE OPEN CLOSED PRINCIPLE

A software component should be open for extension, but closed for modifications.

- Closed: can be used by other components. Well defined stable interface.
- Open: Available for extension. Add new data fields, new functionality.
- Inheritance (possibly from abstract base classes)

LSP: LISKOV'S SUBSTITUTION PRINCIPLE

"If, for each object o_1 of type S, there is an object o_2 of type T, such that for all programs P defined in terms of T, the behaviour of P is unchanged when o_1 is substituted for o_2, then S is a subtype of T."
– Barbara Liskov

- Subtypes must be able to substitute the base type
- Deriving type fully reflects the behaviour of the base class
- True "is a" relationship
- **Guideline:** Don't inherit and then restrict the derived class so that it loses some behaviour expected from the base class

ISP: THE INTERFACE SEGREGATION PRINCIPLE

Clients should not be forced to depend on methods they do not use.

- See under "encapsulation" above
- Avoid "fat" classes. When one client forces a change, every other client is affected, even if they are not using the same part of the fat class.
- Think how the functionality available through the namespace `std` is segregated.

DIP: THE DEPENDENCY INVERSION PRINCIPLE

1. High-level modules should not depend on low level modules. Both should depend on abstractions.
2. Abstractions should not depend on details. Details should depend on abstractions.

- High level components own the interface they depend on.
- They specify their requirements.
- If low level components implement that interface, they can be used with the high level client interface.
- Cut the dependency chain
- Adaptor layers

SUMMARY: SOLID PRINCIPLES

- Avoiding tight coupling between different components may require extra work at first, but wins out in the life time of a project.
- Assign responsibilities carefully.
- SOLID principles are known to help develop and maintain flexible software.

Chapter 5

Lambda functions

FUNCTION LIKE ENTITIES

- In C++, there are a few different constructs which can be used in a context requiring a "function"
- Functions in all varieties constitute one category (`inline` or not, `constexpr` or not, `virtual` or not ...)
- Classes may overload the function call operator `operator()` to give us another type of callable object
- Lambda functions are similar, language provided entities

```cpp
class Wave {
double A, ome, pha;
public:
auto operator()(double t) -> double
{
    return A * sin(ome * t + pha);
}
};
void elsewhere()
{
    Wave W{1.0, 0.15, 0.9};
    for (auto i = 0; i < 100; ++i) {
        std::cout << i << W(i) << "\n";
    }
}
```

LAMBDA FUNCTIONS

- Locally defined callable entities
- Uses
  - Effective use of STL
  - Initialisation of const
  - Concurrency
  - New loop styles
- Like a function object defined on the spot
- Fine grained control over the visibility of the variables in the surrounding scope

```cpp
sort(begin(v), end(v), [](auto x, auto y) {
    return x > y;
});

const auto inp_file = []{
    string resourcefl;
    cout << "resource file : ";
    cin >> resourcefl;
    return resourcefl;
}();
tbb::parallel_for(0, 1000000, [](int i){
    // process element i
});
```

LAMBDA FUNCTIONS

Function

```
auto sqr(double x) -> double
{
    return x * x;
}
```

Lambda expression

```
auto lsqr = [](double x) -> double
{
    return x * x;
};
```

- Normal C++ functions can not be defined in block scope
- Lambda expressions are expressions, which when evaluated yield callable entities. Like 2^9 is an expression, which when evaluated yields 512.
- Such callable entities can be created in global as well as block scope

LAMBDA FUNCTIONS

Function

```
auto sqr(double x) -> double
{
    return x * x;
}
```

Lambda expression

```
auto lsqr = [](double x) -> double
{
    return x * x;
};
```

- The lambda expression contains information which is used to make the callable entity: such as, expected input, output and the body("recipe").
- Unlike normal functions, which have names, these callable entities themselves are **nameless**, but named variables can be constructed out of them, if desired. Those named variables can then be used like functions.

LAMBDA FUNCTIONS

Function

```
auto sqr(double x) -> double
{
    return x * x;
}
```

Lambda expression

```
auto lsqr = [](double x) -> double
{
    return x * x;
};
```

- The lambda expression contains information which is used to make the callable entity: such as, expected input, output and the body("recipe").
- Unlike normal functions, which have names, these callable entities themselves are **nameless**, but named variables can be constructed out of them, if desired. Those named variables can then be used like functions.

```
std::vector X{0.1, 0.2, 0.3, 0.4};
auto sqsum = 0.;
for (auto i = 0UL; i < X.size(); ++i) {
    sqsum += sqr(X[i]);
}
```

LAMBDA FUNCTIONS

Function

```
auto sqr(double x) -> double
{
    return x * x;
}
```

Lambda expression

```
auto lsqr = [](double x) -> double
{
    return x * x;
};
```

- The lambda expression contains information which is used to make the callable entity: such as, expected input, output and the body(“recipe”).
- Unlike normal functions, which have names, these callable entities themselves are **nameless**, but named variables can be constructed out of them, if desired. Those named variables can then be used like functions.

```
std::vector X{0.1, 0.2, 0.3, 0.4};
auto sqsum = 0.;
for (auto i = 0UL; i < X.size(); ++i) {
    sqsum += lsqr(X[i]);
}
```

LAMBDA FUNCTIONS

```
template <Callable F>
auto aggregate(const std::vector<double>& inp, F f) -> double
{
    auto s{0.};
    for (auto i = 0UL; i < inp.size(); ++i) { s += f(inp[i]); }
    return s;
}
```

- Typical use: arguments to higher order functions. Function parameter that specifies an operation to be performed on a value or (as in this case) a range of values

LAMBDA FUNCTIONS

```
template <Callable F>
auto aggregate(const std::vector<double>& inp, F f) -> double
{
    auto s{0.};
    for (auto i = 0UL; i < X.size(); ++i) { s += f(X[i]); }
    return s;
}
// ...
std::vector X{0.1, 0.2, 0.3, 0.4};
auto sqsum = aggregate(X, sqr);
```

- Typical use: arguments to higher order functions. Function parameter that specifies an operation to be performed on a value or (as in this case) a range of values
- Named callable entities can be used when available.

LAMBDA FUNCTIONS

```cpp
template <Callable F>
auto aggregate(const std::vector<double>& inp, F f) -> double
{
    auto s{0.};
    for (auto i = 0UL; i < X.size(); ++i) { s += f(X[i]); }
    return s;
}
// ...
std::vector X{0.1, 0.2, 0.3, 0.4};
auto sqsum = aggregate(X, lsqr);
```

- Typical use: arguments to higher order functions. Function parameter that specifies an operation to be performed on a value or (as in this case) a range of values
- Named callable entities can be used when available.

LAMBDA FUNCTIONS

```cpp
template <Callable F>
auto aggregate(const std::vector<double>& inp, F f) -> double
{
    auto s{0.};
    for (auto i = 0UL; i < X.size(); ++i) { s += f(X[i]); }
    return s;
}
// ...
std::vector X{0.1, 0.2, 0.3, 0.4};
auto sqsum = aggregate(X, [](double x) -> double { return x * x; });
```

- Typical use: arguments to higher order functions. Function parameter that specifies an operation to be performed on a value or (as in this case) a range of values
- Named callable entities can be used when available.
- Often it is more convenient to pass a lambda expression, and let the higher order function create the callable entity it needs!

LAMBDA FUNCTIONS WITH ALGORITHMS

`std::for_each` is a higher order function, similar to this:

```
template <class InputIterator, class UnaryFunction>
void for_each(InputIterator start, InputIterator end, UnaryFunction f)
{
    for (auto it = start; it != end; ++it) f(*it);
}
```

LAMBDA FUNCTIONS WITH ALGORITHMS

`std::for_each` is a higher order function, similar to this:

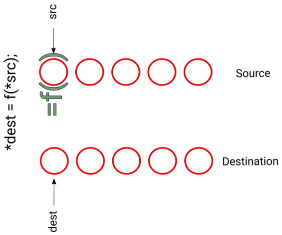
```
template <class InputIterator, class UnaryFunction>
void for_each(InputIterator start, InputIterator end, UnaryFunction f)
{
    for (auto it = start; it != end; ++it) f(*it);
}
```

What do the following lines do ?

```
std::vector X{9, 8, 7, 6, 5, 4, 3, 2, 1, 0};
for_each(X.begin(), X.end(), [](int& elem){ elem = elem * elem; });
for_each(X.begin(), X.end(), [](int& elem){ elem -= 100; });
for_each(X.begin(), X.end(), [](int elem){ std::cout << elem << "\n"; });
```

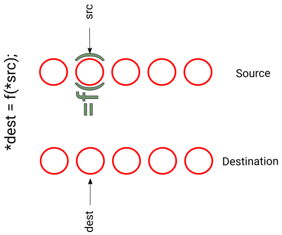
LAMBDA FUNCTIONS WITH ALGORITHMS

`std::transform` is a higher order function, slightly for general than `std::for_each`. It has a few overloads. One of them is similar to this:



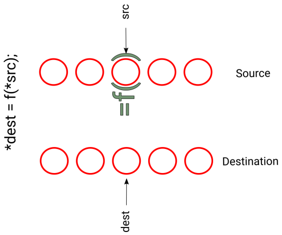
LAMBDA FUNCTIONS WITH ALGORITHMS

`std::transform` is a higher order function, slightly for general than `std::for_each`. It has a few overloads. One of them is similar to this:



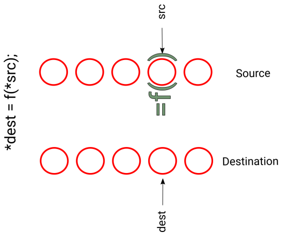
LAMBDA FUNCTIONS WITH ALGORITHMS

`std::transform` is a higher order function, slightly for general than `std::for_each`. It has a few overloads. One of them is similar to this:



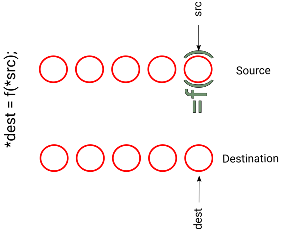
LAMBDA FUNCTIONS WITH ALGORITHMS

`std::transform` is a higher order function, slightly for general than `std::for_each`. It has a few overloads. One of them is similar to this:



LAMBDA FUNCTIONS WITH ALGORITHMS

`std::transform` is a higher order function, slightly for general than `std::for_each`. It has a few overloads. One of them is similar to this:



LAMBDA FUNCTIONS WITH ALGORITHMS

`std::transform` is a higher order function, slightly for general than `std::for_each`. It has a few overloads. One of them is similar to this:

```
template <class InputIt, class OutputIt,
          class UnaryFunction>
void transform(InputIt start, InputIt end,
               OutputIt out,
               UnaryFunction f)
{
    for (; start != end; ++start, ++out)
        *out = f(*start);
}
```

LAMBDA FUNCTIONS WITH ALGORITHMS

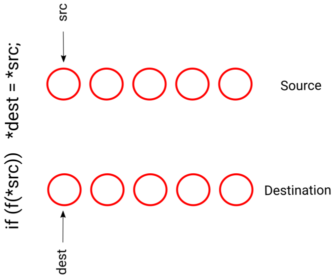
`std::transform` is a higher order function, slightly for general than `std::for_each`. It has a few overloads. One of them is similar to this:

What do the following lines do ?

```
std::vector X{9, 8, 7, 6, 5, 4, 3, 2, 1, 0};
std::vector<int> Y;
transform(X.begin(), X.end(), std::back_inserter(Y),
          [](int elem){ return elem * elem; });
```

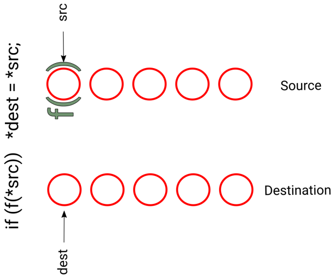
LAMBDA FUNCTIONS WITH ALGORITHMS

`std::copy_if` Conditionally copies elements from a source sequence to a destination sequence:



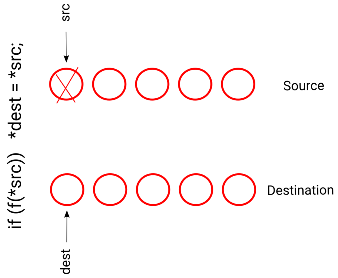
LAMBDA FUNCTIONS WITH ALGORITHMS

`std::copy_if` Conditionally copies elements from a source sequence to a destination sequence:



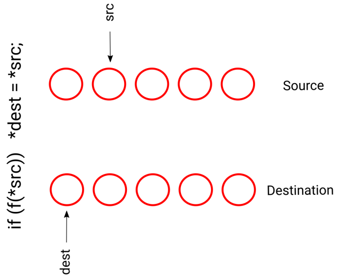
LAMBDA FUNCTIONS WITH ALGORITHMS

`std::copy_if` Conditionally copies elements from a source sequence to a destination sequence:



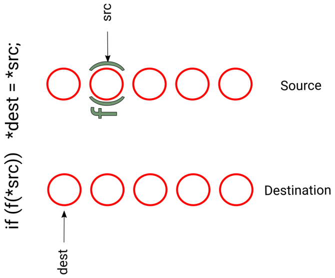
LAMBDA FUNCTIONS WITH ALGORITHMS

`std::copy_if` Conditionally copies elements from a source sequence to a destination sequence:



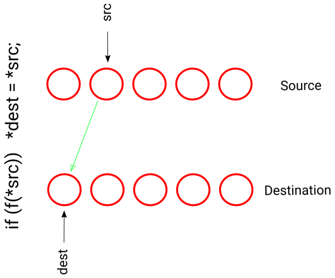
LAMBDA FUNCTIONS WITH ALGORITHMS

`std::copy_if` Conditionally copies elements from a source sequence to a destination sequence:



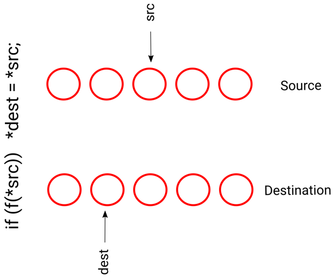
LAMBDA FUNCTIONS WITH ALGORITHMS

`std::copy_if` Conditionally copies elements from a source sequence to a destination sequence:



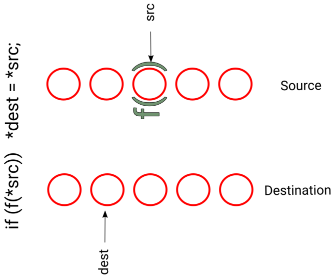
LAMBDA FUNCTIONS WITH ALGORITHMS

`std::copy_if` Conditionally copies elements from a source sequence to a destination sequence:



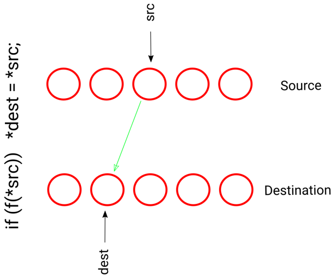
LAMBDA FUNCTIONS WITH ALGORITHMS

`std::copy_if` Conditionally copies elements from a source sequence to a destination sequence:



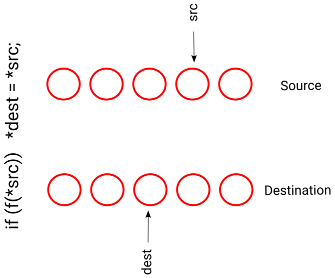
LAMBDA FUNCTIONS WITH ALGORITHMS

`std::copy_if` Conditionally copies elements from a source sequence to a destination sequence:



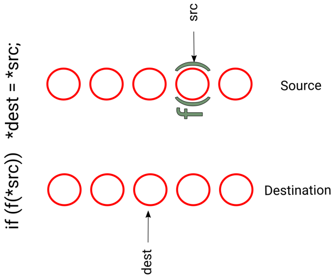
LAMBDA FUNCTIONS WITH ALGORITHMS

`std::copy_if` Conditionally copies elements from a source sequence to a destination sequence:



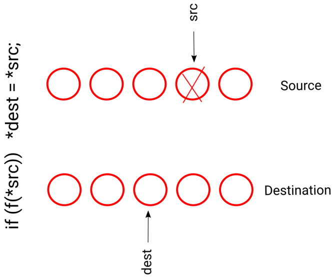
LAMBDA FUNCTIONS WITH ALGORITHMS

`std::copy_if` Conditionally copies elements from a source sequence to a destination sequence:



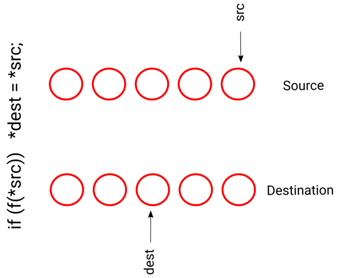
LAMBDA FUNCTIONS WITH ALGORITHMS

`std::copy_if` Conditionally copies elements from a source sequence to a destination sequence:



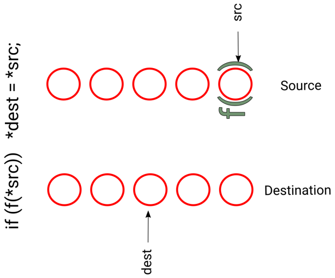
LAMBDA FUNCTIONS WITH ALGORITHMS

`std::copy_if` Conditionally copies elements from a source sequence to a destination sequence:



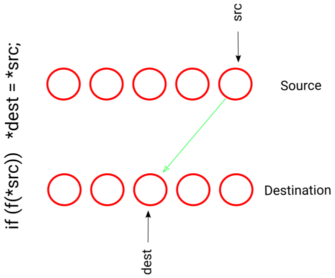
LAMBDA FUNCTIONS WITH ALGORITHMS

`std::copy_if` Conditionally copies elements from a source sequence to a destination sequence:



LAMBDA FUNCTIONS WITH ALGORITHMS

`std::copy_if` Conditionally copies elements from a source sequence to a destination sequence:

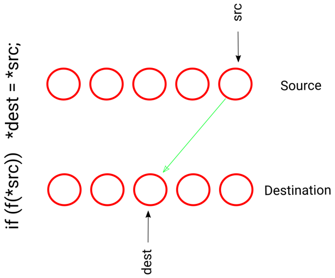


LAMBDA FUNCTIONS WITH ALGORITHMS

`std::copy_if` Conditionally copies elements from a source sequence to a destination sequence:

What do the following lines do ?

```cpp
std::vector X{9, 8, 7, 6, 5, 4, 3, 2, 1, 0};
std::vector<int> Y;
copy_if(X.begin(), X.end(), std::back_inserter(Y),
            [](int elem){ return elem % 3 == 0; });
```



Exercise 5.1:

Use the notebook `lambda_practice_0.ipynb` to quickly practice writing a few small lambdas and using them with a few standard library algorithms.

CAPTURE BRACKETS

- Suppose we want to transfer some elements from one vector to another

```
std::vector<int> v{1, -1, 9, 3, 4, -7, 3, -2}, w;
```

CAPTURE BRACKETS

- Suppose we want to transfer some elements from one vector to another

```
std::vector<int> v{1, -1, 9, 3, 4, -7, 3, -2}, w;
```

- Copy to `w` all positive elements

```
copy_if(v.begin(), v.end(), back_inserter(w), [](int i){ return i>0; });
```

CAPTURE BRACKETS

- Suppose we want to transfer some elements from one vector to another

```
std::vector<int> v{1, -1, 9, 3, 4, -7, 3, -2}, w;
```

- Copy to `w` all positive elements

```
copy_if(v.begin(), v.end(), back_inserter(w), [](int i){ return i>0; });
```

- Copy to `w` all elements larger than a user specified value

CAPTURE BRACKETS

- Suppose we want to transfer some elements from one vector to another

```
std::vector<int> v{1, -1, 9, 3, 4, -7, 3, -2}, w;
```

- Copy to `w` all positive elements

```
copy_if(v.begin(), v.end(), back_inserter(w), [](int i){ return i>0; });
```

- Copy to `w` all elements larger than a user specified value
- This does not work

```
std::cin >> lim;
copy_if(v.begin(), v.end(), back_inserter(w), [](int i){ return i > lim ; });
// Lambda function has its own scope, and lim is not visible
```

CAPTURE BRACKETS

- Suppose we want to transfer some elements from one vector to another

```
std::vector<int> v{1, -1, 9, 3, 4, -7, 3, -2}, w;
```

- Copy to `w` all positive elements

```
copy_if(v.begin(), v.end(), back_inserter(w), [](int i){ return i>0; });
```

- Copy to `w` all elements larger than a user specified value
- This does not work

```
std::cin >> lim;
copy_if(v.begin(), v.end(), back_inserter(w), [](int i){ return i > lim ; });
// Lambda function has its own scope, and lim is not visible
```

- A way to make the lambda selectively aware of chosen variables in its context:

```
std::cin >> lim;
copy_if(v.begin(), v.end(), back_inserter(w),
                                [lim](int i){ return i > lim; });
// Lambda function "captures" lim, and lim is now visible inside the lambda
```

LAMBDA EXPRESSIONS: SYNTAX

```
[capture] <templatepars> (arguments) lambda-specifiers { body }
```

- Variables in the body of a lambda function are either passed as function arguments or "captured", or are global variables
- Function arguments field is optional if empty. e.g. `[&cc]{ return cc++; }`
- The *lambda-specifiers* field can contain a variety of things: Keywords `mutable`, `constexpr` or `consteval`, exception specifiers, attributes, the return type, and any `requires` clauses. All of these are optional.
- The return type is optional if there is one return statement. e.g. `[a,b,c](int i) mutable { return a*i*i + b*i + c; }`
- The optional keyword `mutable` can be used to create lambdas with state
- `auto` can be used to declare the formal input parameters of the lambda (since C++14)
- Template parameters can be optionally provided where shown (since C++20)

JÜLICH Forschungszentrum

EXPLICIT TEMPLATE PARAMETERS FOR LAMBDA FUNCTIONS

```
// examples/saxpy_2.cc
// includes ...
auto main() -> int {
    const std::vector inp1 { 1., 2., 3., 4., 5. };
    const std::vector inp2 { 9., 8., 7., 6., 5. };
    std::vector outp(inp1.size(), 0.);

    auto saxpy = [] <class T, class T_in, class T_out>
                 (T a, const T_in& x, const T_in& y, T_out& z) {
        std::transform(x.begin(), x.end(), y.begin(), z.begin(),
                       [a](T X, T Y){ return a * X + Y; });
    };

    std::ostream_iterator<double> cout { std::cout, "\n" };
    saxpy(10., inp1, inp2, outp);
    copy(outp.begin(), outp.end(), cout);
}
```

For normal function templates, we could easily express relationships among the types of different parameters. With C++20, we can do that for generic lambdas as well

LAMBDA CAPTURE SYNTAX I

```
[capture]<templatepars> (arguments) lambda-specifiers { body }
```

- `[ ](int a, int b) -> bool { return a > b;}` : Capture nothing. Work only with the arguments passed, or global objects.
- `[=](int a) -> bool {return a > somevar;}` : Capture everything needed by value.
- `[&](int a){somevar += a;}` : Capture everything needed by reference.
- `[=,&somevar](int a){ somevar += max(a,othervar); }` : `somevar` by reference, but everything else as value.
- `[a,&b]{ f(a,b); }` : `a` by value, `b` by reference.
- `[a=std::move(b)]{ f(a,b); }` : Init capture. Create a variable `a` with the initializer given in the capture brackets. It is as if there were an implicit `auto` before the `a`.

Exercise 5.2:

The program `lambda_captures.cc` (alternatively, notebook `lambda_practice_1.ipynb`) declares a variable of the `Vbose` type (with all constructors, assignment operators etc. written to print messages), and then defines a lambda function. By changing the capture type, and the changing between using and not using the `Vbose` value inside the lambda function, try to understand, from the output, the circumstances under which the captured variables are copied into the lambda. In the cases where you see a copy, where does the copy take place ? At the point of declaration of the lambda or at the point of use ?

LAMBDA FUNCTIONS: CAPTURES

- Imagine there is a variable `int p=5` defined previously

LAMBDA FUNCTIONS: CAPTURES

- Imagine there is a variable `int p=5` defined previously
- We can “capture” `p` by value and use it inside our lambda

```
auto L = [p](int i){ std::cout << i*3 + p; };
L(3); // result : prints out 14
auto M = [p](int i){ p = i*3; }; // syntax error! p is read-only!
```

LAMBDA FUNCTIONS: CAPTURES

- Imagine there is a variable `int p=5` defined previously
- We can “capture” `p` by value and use it inside our lambda

```
auto L = [p](int i){ std::cout << i*3 + p; };
L(3); // result : prints out 14
auto M = [p](int i){ p = i*3; }; // syntax error! p is read-only!
```

- We can capture `p` by value (make a copy), but use the `mutable` keyword, to let the lambda function change its local copy of `p`

```
auto M = [p](int i) mutable { return p += i*3; };
std::cout << M(1) << " "; std::cout << M(2) <<" "; std::cout << p <<"\n";
// result : prints out "8 14 5"
```

LAMBDA FUNCTIONS: CAPTURES

- Imagine there is a variable `int p=5` defined previously
- We can "capture" `p` by value and use it inside our lambda

```
auto L = [p](int i){ std::cout << i*3 + p; };
L(3); // result : prints out 14
auto M = [p](int i){ p = i*3; }; // syntax error! p is read-only!
```

- We can capture `p` by value (make a copy), but use the `mutable` keyword, to let the lambda function change its local copy of `p`

```
auto M = [p](int i) mutable { return p += i*3; };
std::cout << M(1) << " "; std::cout << M(2) <<" "; std::cout << p <<"\n";
// result : prints out "8 14 5"
```

- We can capture `p` by reference and modify it

```
auto M = [&p](int i){ return p += i*3; };
std::cout << M(1) << " "; std::cout << M(2) << " "; std::cout << p << "\n";
// result : prints out "8 14 14"
```

NO DEFAULT CAPTURE!

`[]`	Capture nothing
`[=]`	Capture used by value (copy)
`[=,&x]`	Capture used by value, except x by reference
`[&]`	Capture used by reference
`[&,x]`	Capture used by reference, except x by value
`[a=init]`	Init capture

- A lambda with empty capture brackets is like a local function, and can be assigned to a regular function pointer. It is not aware of identifiers defined previously in its context
- When you use a (non-global) variable defined outside the lambda in the lambda, you have to capture it

STATEFUL LAMBDAS

- Mutable lambdas have "state", and remember any changes to the values captured by value
- Combined with "init capture", gives us interesting generator functions

```
vector<int> v, w;
generate_n(back_inserter(v), 100, [i=0]() mutable {
    ++i;
    return i*i;
});
// v = [1, 4, 9, 16 ... ]
generate_n(back_inserter(w), 50, [i=0, j=1]() mutable {
    i = std::exchange(j, j+i); // exchange(a,b) sets a to b and returns the old value of a
    return i;
});
// See the videos on Fibonacci sequence on the
// YouTube channel "C++ Weekly" by Jason Turner
// w = [1, 1, 2, 3, 5, 8, 11 ...]
```

Exercise 5.3:

The program `mutable_lambda.cc` shows the use of mutable lambdas for sequence initialisation.

Chapter 6

Standard Template Library

Standard Template Library

STANDARD TEMPLATE LIBRARY

- Utilities
  - `pair`, `tuple`
  - `optional`, `variant`, `any`
  - `bitset`, `bit`, `endian`, `bit_cast`
  - `type_traits`, `concepts`, safe integral comparisons
  - `initializer_list`
  - `system`, `atexit`
  - `bind`, `placeholders`, `apply`, `invoke` ...
- Date and Time
- Random numbers
- Smart pointers
- File system
- Regular expressions
- Containers, `span`
- Algorithms, ranges
- Iterators
- Strings and string view
- Fast character conversions
- Multi-threading, atomic types
- Parallel algorithms
- Text formatting

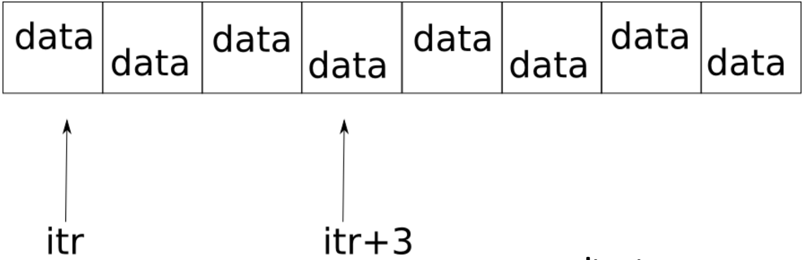
STL CONTAINERS

```cpp
using namespace std;
int sz;
cin >> sz;
// vector<double> B(sz,3.0); // <- C++17 ->
vector B(sz, 3.0); // C++17 ->
vector c{1, 2, 3, 4};
c.push_back(5); // append
list l{1, 2, 3, 4};
l.insert(find(l.begin(),l.end(),2), 14);
// insert in the middle
map<string, int> rank;
rank["Sirius"] = 1;
rank["Canopus"] = 2;
for (auto el : B) cout << el << "\n";
for (auto el : l) cout << el << "\n";
for (auto el : rank)
    cout << el.first <<" -> "
         << el.second << "\n";
```

- Form: `container<datatype>`. Include file `containername`

- Many easy-to-use sequence types available in the STL
  - `vector` : Dynamic array type
  - `list` : Linked list
  - `map` : Sorted associative container
  - `unordered_map` : Hash table
- Not always necessary to explicitly state the element type. If there is an initialiser, element type can be inferred.
- Store a fixed kind of elements, determined at the point of declaration.
- They can grow at run time (except `std::array`)
- Whenever possible, prefer `array` or `vector`

VECTOR: DYNAMIC ARRAY CLASS TEMPLATE



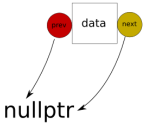
- Element type is a template parameter
- Consecutive elements in memory
- Can be accessed using an "iterator"

Iterator:

- Iterators are classes which pretend to be pointers
- They can be dereferenced with overloaded `*` and `->` operators to retrieve an element
- They can be moved forward or backward using overloaded `++` and `--` operators
- They can be compared for equality or inequality

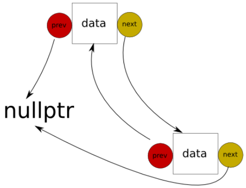
A LINKED LIST

A linked list is a collection of connected nodes. Each node has some data, and one or two pointers to other nodes. They are the "next" and "previous" nodes in the linked list. When "next" or "previous" does not exist, the pointer is set to `nullptr`



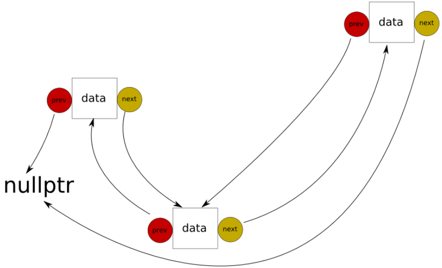
A LINKED LIST

When a new element is added to the end of a list, its "previous" pointer is set to the previous end of chain, and it becomes the target of the "next" pointer of the previous end.



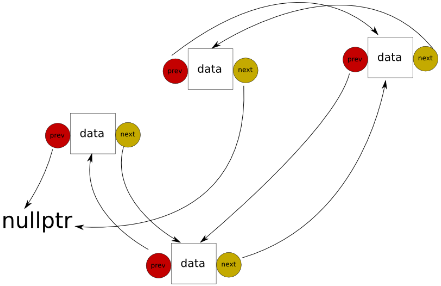
A LINKED LIST

New elements can be added to the front or back of the list with only a few pointers needing rearrangement.



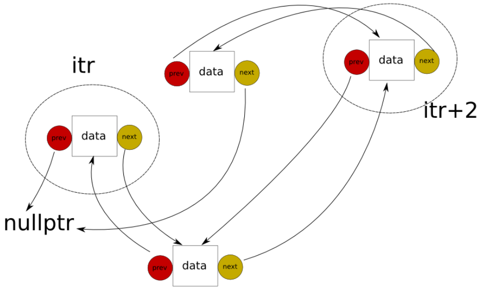
A LINKED LIST

Any element in the list can be reached, if one kept track of the beginning or end of the list, and followed the "next" and "previous" pointers.



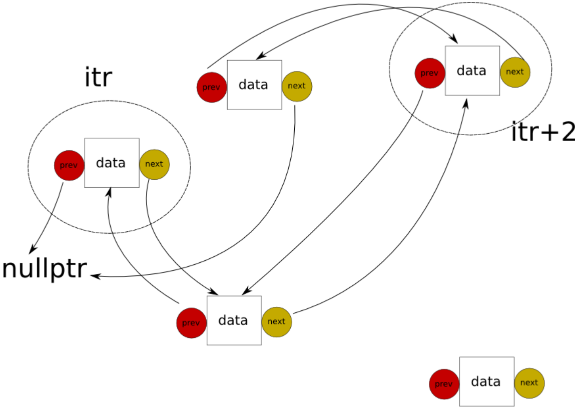
A LINKED LIST

A concept of an "iterator" can be devised, where the `++` and `--` operators move to the next and previous nodes.



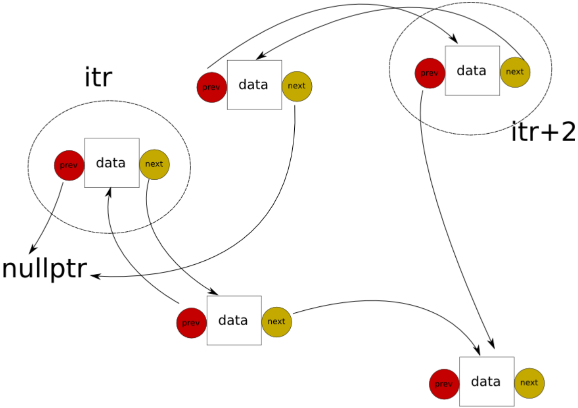
A LINKED LIST

Inserting a new element in the middle of the list does not require moving the existing nodes in memory.



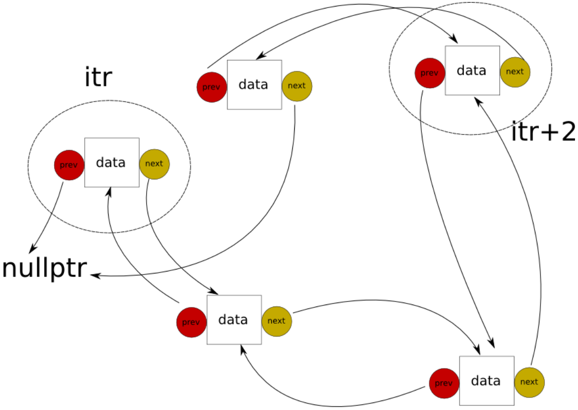
A LINKED LIST

Just rearranging the next and previous pointers of the elements between which the new element must go, is enough. This gives efficient $O(1)$ insertions and deletions.

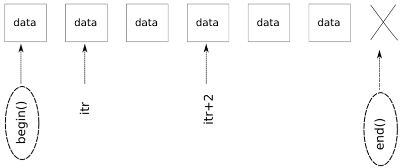


A LINKED LIST

Just rearranging the next and previous pointers of the elements between which the new element must go, is enough. This gives efficient $O(1)$ insertions and deletions.



GENERIC "CONTAINERS"



- Generic data holding constructions
- Can be accessed through a suitably designed "iterator"
- The data type does not affect the design $\Longrightarrow$ template

- Similarity of interface is by design
- With a standard container `c` of type `C`, it's always possible to use `std::begin(c)` to access the start and `std::end(c)` to access the end
- `std::begin()` and `std::end()` return `C::iterator` or `C::const_iterator` depending on whether `c` is const qualified.
- `std::cbegin(c)` and `std::cend(c)` return `C::const_iterator` types irrespective of whether `c` is a const
- Similarly, `std::size(c)` always returns the size of the container, i.e., the number of elements it contains

STL CONTAINERS

- `std::vector<>` : dynamic arrays
- `std::list<>` : linked lists
- `std::queue<>` : queue
- `std::deque<>` : double ended queue
- `std::map<A,B>` : associative container

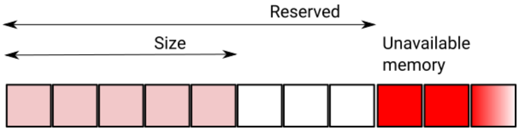
- Structures to organise data
- Include file names correspond to class names
- All of them provide corresponding iterator classes
- If `iter` is an iterator, `*iter` is data.
- All of them provide member functions like `begin()`, `end()`, `size()`, initialiser list constructors, deduction rules for class template argument deduction

```cpp
list L{1, 2, 3, 4, 5}; // std::list<int>, initialised to 1, 2, 3, 4, 5
auto pp = partition(begin(L), end(L), [](auto i){ return i % 3 == 0; });
decltype(L) M;
M.splice(end(M), L, begin(L), pp);
```

USING STD::VECTOR

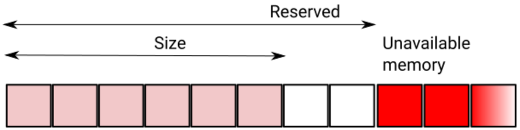
- `vector<int> v(10);` makes a dynamic array of 10 integers, `vector v(10, 0.)` creates a vector of 10 `doubles` initialised to 0, `vector v{1u, 2u, 3u}` creates a vector of `unsigned int` with values 1, 2 and 3.
- Efficient indexing operator `[]`, for unchecked element access
- `v.at(i)` provides range checked access. An exception is thrown if `at(i)` is called with an out-of-range `i`
- `std::vector<std::list<userinfo>> vu(10) ;` array of 10 linked lists.
- Supports `push_back` and insert operations, but sometimes has to relocate the all the elements because of one `push_back` operation (next slide)

STD::VECTOR



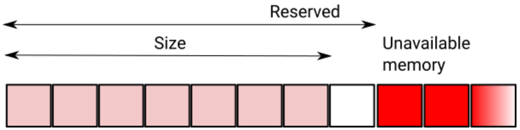
- `std::vector` may reserve a few extra memory blocks to allow a few quick `push_back` operations.
- New items are simply placed in the previously reserved but unused memory and the size member adjusted.

STD::VECTOR



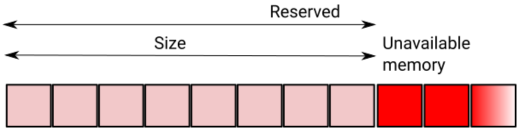
- `std::vector` may reserve a few extra memory blocks to allow a few quick `push_back` operations.
- New items are simply placed in the previously reserved but unused memory and the size member adjusted.

STD::VECTOR



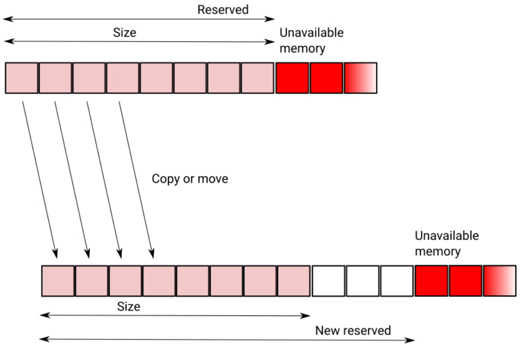
- `std::vector` may reserve a few extra memory blocks to allow a few quick `push_back` operations.
- New items are simply placed in the previously reserved but unused memory and the size member adjusted.

STD::VECTOR



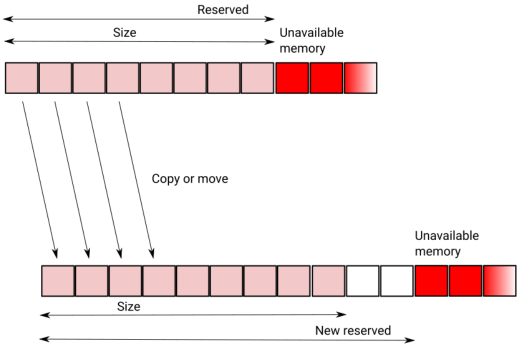
- `std::vector` may reserve a few extra memory blocks to allow a few quick `push_back` operations.
- New items are simply placed in the previously reserved but unused memory and the size member adjusted.

STD::VECTOR



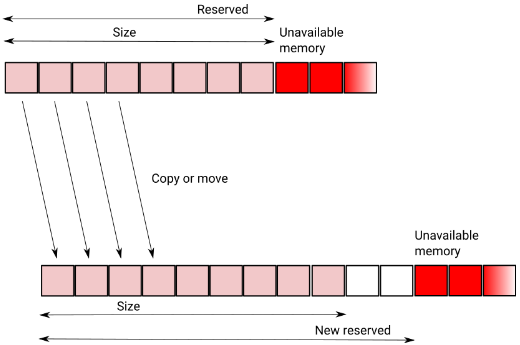
- When this is no longer possible, a new larger memory block is reserved, and all previous content is moved or copied to it.
- A few more quick `push_back` operations are again possible.

STD::VECTOR



- When this is no longer possible, a new larger memory block is reserved, and all previous content is moved or copied to it.
- A few more quick `push_back` operations are again possible.

STD::VECTOR



- When `push_back` is no longer possible, a new larger memory block is reserved, and all previous content is moved or copied to it.
- A few more quick `push_back` operations are again possible.

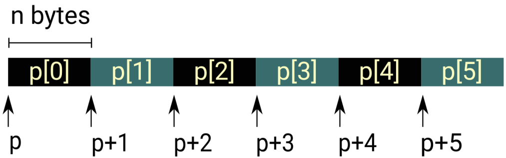
Exercise 6.1:

Construct a list and a vector of 3 elements of the `Vbose` class from your earlier exercise. Add new elements one by one and pause to examine the output. This aspect was also demonstrated in the notebook `CtorDtorDemo.ipynb`.

STD::ARRAY : ARRAYS WITH FIXED COMPILE TIME CONSTANT SIZE

- `std::array<T,N>` is a fixed length array of size N holding elements of type `T`
- It implements functions like `begin()` and `end()` and is therefore usable with STL algorithms like `transform`, `generate` etc.
- The array size is a template parameter, and hence a compile time constant.
- `std::array<std::string,7> week{"Mon","Tue","Wed","Thu","Fri","Sat","Sun"};`

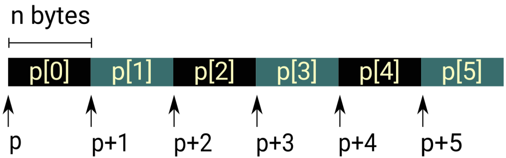
ARRAYS




```cpp
double A[10]; // Built-in or C-style array
int sz;
std::cin >> sz;
int M[sz]; // Not allowed!
#include <array>
...
std::array<double,10> A; // On stack
// Like a built-in array, but obeys
// C++ standard library conventions.
for (size_t i = 0; i < A.size(); ++i) {
    P *= A[i];
}
std::vector<double> B(sz,3.0);
```

- Sequence of N objects stored consecutively in memory, with no gaps
- If `p` is a pointer to the first object of such a sequence, `p+1` , `p+2` etc, will point to the subsequent elements. Elements of the sequence can therefore be accessed as `*(p+0)` , `*(p+1)` , `*(p+2)` ... another notation for that is `p[0]` , `p[1]` ...

ARRAYS




```cpp
double A[10]; // Built-in or C-style array
int sz;
std::cin >> sz;
int M[sz]; // Not allowed!
#include <array>
...
std::array<double,10> A; // On stack
// Like a built-in array, but obeys
// C++ standard library conventions.
for (size_t i = 0; i < A.size(); ++i) {
    P *= A[i];
}
std::vector<double> B(sz,3.0);
```

- Built-in or ”C-style" arrays consist of blocks of memory large enough to hold a fixed number of elements. The array, thought of as a pointer, points to the first element in the sequence. The elements are stored consecutively, but the number of elements is never stored anywhere
- `std::array<type,size>` is a compile-time fixed length array obeying STL conventions. The size is available through a function, although it does not have to be stored with the array data!
- `std::array<type,size>` retains its “personality” (does not decay into a pointer) when used as input to function or when received as the output from a function. This should be your default choice when you need fixed length arrays.

ASSOCIATIVE CONTAINERS: STD::MAP

```
std::map<std::string, int> flsize;
flsize["S.dat"]=123164;
flsize["D.dat"]=423222;
flsize["A.dat"]=1024;
```

- Think of it as a special kind of "vector" where you can have things other than integers as indices.
- Template arguments specify the key and data types
- Could be thought of as a container storing (key,value) pairs : $\{("S.dat", 123164), ("D.dat", 423222), ("A.dat", 1024)\}$
- The less than comparison operation must be defined on the key type
- Implemented as a tree, which keeps its elements sorted

A WORD COUNTER PROGRAM

Exercise 6.2:

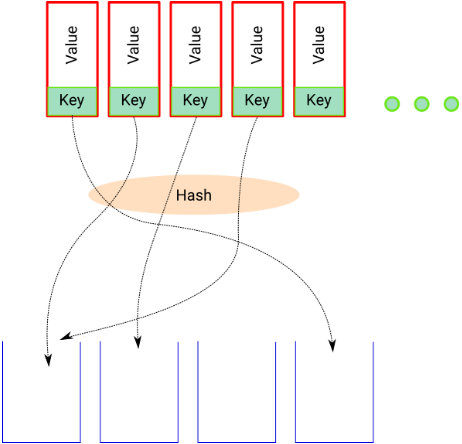
Fake exercise: Write a program that counts all different words in a text file and prints the statistics.

```cpp
#include <iostream>
#include <fstream>
#include <iomanip>
#include <string>
#include <map>
auto main(int argc, char *argv[]) -> int
{
    std::ifstream fin(argv[1]);
    std::map<std::string, unsigned> freq;
    std::string s;
    while (fin >> s) freq[s]++;
    for (auto [key, val] : freq)
        cout << std::setw(12) << key
             << std::setw(4)  << ':'
             << std::setw(12) << val <<"\n";
}
```

A quick histogram!

- `std::map<string, unsigned>` is a container which stores an integer, for each unique `std::string` key.
- The iterator for `std::map` “points to” a `pair<key,value>`

STD::UNORDERED_MAP AND STD::UNORDERED_SET



Unordered map

- Like `std::map<k,v>` and `std::set<v>`, but do not sort the elements
- Internally, these are hash tables, providing faster element access than `std::map` and `std::set`
- Additional template arguments to specify hash functions

VALARRAY

```cpp
#include <valarray>

void varray_ops()
{
    std::valarray V1(0., 1000000UL);
    std::valarray<double> V2;
    v2.resize(1000000UL, 0.);
    auto x = exp(-V1 * V1) * sin(V2);
    if (x.sum() < 100.0) {
    //
    }
}
```

- Another dynamic array type
- Mostly intended for numeric operations
- Expression template based whole array mathematical operations
- Algorithms through `std::begin(v)` etc., instead of own member functions
- Bizarre constructor with different convention compared to any other container in the STL.

STL ALGORITHMS

```
...
std::vector<YourClass> vc(inp.size());
std::copy(inp.begin(), inp.end(), vc.begin());
//Copy contents of list to a vector
auto pos = std::find(vc.begin(), vc.end(), elm);
//Find an element in vc which equals elm
std::sort(vc.begin(),vc.end());
//Sort the vector vc. The operator "<"
//must be defined
...
std::transform(inp.begin(), inp.end(), out.begin(), rotate);
//apply rotate() to each input element,
//and store results in output sequence
```

The similarity of the interface, e.g. `begin()`, `end()` etc., among STL containers allows generic algorithms to be written as template functions, performing common tasks on collections

STL ALGORITHMS

- Typically, the algorithms in the namespace `std` accept one or more ranges as (start, stop) pairs, some other inputs which may include callable objects
- New algorithms were introduced in C++20 in the namespace `std::ranges`, where the input ranges are given as single objects rather than iterator pairs. Think

```
std::ranges::for_each(v, [](auto&& elem){ std::cout << elem << "\n"; })
```

rather than

```
std::for_each(v.begin(), v.end(), [](auto&& elem){ std::cout << elem << "\n"; })
```

Exercise 6.3:

- The standard library provides a large number of template functions to work with containers
- Look them up in `www.cplusplus.com` or `en.cppreference.com`
- Use the suitable STL algorithms to generate successive permutations of the vector

STL ALGORITHMS: SORTING

- `std::sort(iter_1, iter_2)` sorts the elements between iterators `iter_1` and `iter_2`

```cpp
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;
auto main() -> int
{
  vector v{2, -3, 7, 4, -1, 9, 0};
  sort(v.begin(), v.end());
  //Sort using "<" operator
  for (auto el : v) cout << el << "\n";
  sort(v.begin(), v.end(),
          [](int i, int j) {
      return i * i < j * j;
  });
  //Sort using custom comparison
  for (auto el: v) cout << el << "\n";
}
```

STL ALGORITHMS: SORTING

- `std::sort(iter_1, iter_2)` sorts the elements between iterators `iter_1` and `iter_2`
- `std::sort(iter_1, iter_2, lt)` sorts the elements between iterators `iter_1` and `iter_2` using a custom comparison method `lt`, which could be any callable object

```cpp
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;
auto main() -> int
{
  vector v{2, -3, 7, 4, -1, 9, 0};
  sort(v.begin(), v.end());
  //Sort using "<" operator
  for (auto el : v) cout << el << "\n";
  sort(v.begin(), v.end(),
       [](int i, int j) {
    return i * i < j * j;
  });
  //Sort using custom comparison
  for (auto el: v) cout << el << "\n";
}
```

STL ALGORITHMS: SORTING

- `std::sort(iter_1, iter_2)` sorts the elements between iterators `iter_1` and `iter_2`
- `std::sort(iter_1, iter_2, lt)` sorts the elements between iterators `iter_1` and `iter_2` using a custom comparison method `lt`, which could be any callable object
- `std::ranges::sort(range)` and `std::ranges::sort(range, lt)` are corresponding versions using a range as an argument instead of a pair of iterators

```cpp
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;
auto main() -> int
{
  vector v{2, -3, 7, 4, -1, 9, 0};
  sort(v.begin(), v.end());
  //Sort using "<" operator
  for (auto el : v) cout << el << "\n";
  ranges::sort(v, [](int i, int j) {
      return i * i < j * j;
  });
  //Sort using custom comparison
  for (auto el: v) cout << el << "\n";
}
```

STD::TRANSFORM

- `std::transform(begin_1 , end_1, begin_res, unary_function);`
- `std::transform(begin_1 , end_1, begin_2, begin_res, binary_function);`
- Apply callable object to the sequence and write result starting at a given iterator location
- The container holding result must be previously resized so that it has the right number of elements
- The “result” container can be (one of the) input container(s)

```
std::vector v{0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9};
std::list L1(v.size(), 0), L2(v.size(), 0);
std::transform(v.begin(), v.end(), L1.begin(), sin);
std::transform(v.begin(), v.end(), L1.begin(), L2.begin(), std::max);
```

Result: `L1` contains `sin(x)` for each `x` in `v`, and `L2` contains the `greater(x,sin(x))`

STD::RANGES::TRANSFORM

- `std::ranges::transform(range1, begin_res, unary_function);`
- `std::transform(range1, range2, begin_res, binary_function);`
- Apply callable object to the sequence and write result starting at a given iterator location
- The container holding result must be previously resized so that it has the right number of elements
- The “result” container can be (one of the) input container(s)

```
std::vector v{0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9};
std::list L1(v.size(), 0), L2(v.size(), 0);
std::ranges::transform(v, L1.begin(), sin);
std::ranges::transform(v, L1, L2.begin(), std::max);
```

Result: `L1` contains `sin(x)` for each `x` in `v`, and `L2` contains the `greater(x,sin(x))`

ALL_OF, ANY_OF, NONE_OF

```
auto valid(std::string name) -> bool
{
  return all_of(name.begin(),name.end(),
        [](char c) { return (isalpha(c)) || isspace(c); });
}
```

- `std::all_of(begin_ , end_ , condition)` checks if all elements in a given range satisfy `condition`
- `condition` is a callable object
- `std::any_of(begin_ , end_ , condition)` checks if any single element in a given range satisfies `condition`
- `std::none_of(begin_ , end_ , condition)` returns true if not a single element in a given range satisfies `condition`

ALL_OF, ANY_OF, NONE_OF

```cpp
auto valid(std::string name) -> bool
{
  return all_of(name,
        [](char c) { return (isalpha(c)) || isspace(c); });
}
```

- `std::ranges::all_of(range , condition)` checks if all elements in a given range satisfy `condition`
- `condition` is a callable object
- `std::ranges::any_of(range , condition)` checks if any single element in a given range satisfies `condition`
- `std::ranges::none_of(range , condition)` returns true if not a single element in a given range satisfies `condition`

ALGORITHMS

```cpp
vector v{ 1, 2, 3, 4, 5, 6, 7, 8, 9 }, w{ 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 };
vector<int>  x, y, z, m;
if (is_sorted(begin(v), end(v)))
    cout << "The sequence is sorted in the increasing order.\n";
reverse(v.begin(), v.end());
rotate(v.begin(), v.begin() + 3, v.end());
sort(begin(v), end(v));
merge(v.begin(), v.end(), w.begin(), w.end(), back_inserter(m));
set_union(v.begin(), v.end(), w.begin(), w.end(), back_inserter(x));
set_intersection(w.begin(), w.end(), v.begin(), v.end(), back_inserter(y));
set_symmetric_difference(v.begin(), v.end(), w.begin(), w.end(), back_inserter(z));
if (is_permutation(z.begin(), z.end(), v.begin(), v.end())) // do something
```

Exercise 6.4:

A whole lot of operations available for sequence types. The file `seqops.cc` contains the operations shown here. Alternatively, (or, in addition,) use the jupyter notebook `intro_algorithms.ipynb` to examine the effects of the algorithms on sequences. Explore!

ALGORITHMS

- `for_each(start, end, operation)` : As it sounds
- `find(start, end, what)` : returns the location of the looked for value, "end" if not found
- `find_if(start, end, condition)` , find the first element satisfying a condition
- `copy(start1, end1, start2)` : As it sounds
- `copy_if(start1, end1, start2, criterion)` : `criterion` is a unary function taking a value of the type found in the sequence and returning true or false
- `transform(start1, end1, start2, operation)` : applies `operation` on every element in the input sequence and writes the results starting at `start2`

JÜLICH Forschungszentrum

CONSTRAINED ALGORITHMS (RANGES)

- `for_each(range, operation)` : As it sounds
- `find(range, what)` : returns the location of the looked for value, "end" if not found
- `find_if(range, condition)`, find the first element satisfying a condition
- `copy(range1, iterator2)` : As it sounds
- `copy_if(range1, iterator2, criterion)` : `criterion` is a unary function taking a value of the type found in the sequence and returning true or false
- `transform(range1, iterator2, operation)` : applies `operation` on every element in the input sequence and writes the results starting at `iterator2`

ALGORITHMS

```cpp
vector v{ 1, 2, 3, 4, 5, 6, 7, 8, 9 }, w{ 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 };
vector<int>  x, y, z, m;
if (is_sorted(v))
    cout << "The sequence is sorted in the increasing order.\n";
reverse(v);
rotate(v, v.begin() + 3);
sort(v);
merge(v, w, back_inserter(m));
set_union(v, w, back_inserter(x));
set_intersection(w, v, back_inserter(y));
set_symmetric_difference(v, w, back_inserter(z));
if (is_permutation(zv)) // do something
```

Exercise 6.5:

The file `seqops_range.cc` contains the operations shown here. Explore by making modifications. Try GCC 10.0+ compiler for this.

NUMERIC ALGORITHMS

```
#include <numeric>

using std::reduce;
using std::transform_reduce;

auto res = reduce(v.begin(), v.end());
auto res = reduce(v.begin(), v.end(), init);
auto res = reduce(v.begin(), v.end(),
    init, std::plus<double>{});
auto res = transform_reduce(
    u.begin(), u.end(),
    v.begin(), init);
auto res = transform_reduce(
    u.begin(), u.end(),
    v.begin(), init, reduce_op, transf_op);
auto res = transform_reduce(
    std::execution::par,
    u.begin(), u.end(),
    v.begin(), init, reduce_op, transf_op);
```

- Algorithms focused on numeric calculations are in the `numeric` header
- Given `b`, `e` as iterators in a range V, `reduce(b, e)`: $\sum_{i=b}^{e} V_i$
- `transform_reduce(b, e, f)`: $\sum_{i=b}^{e} f(V_i)$
- `adjacent_difference(b, e)`: $\{V_b, (V_{b+1} - V_b), (V_{b+2} - V_{b+1}), \ldots\}$
- Parallel versions also in the library
- To run the numeric operations in parallel, use the parallel execution policy

SPAN

```cpp
using std::span;
using std::transform_reduce;
using std::plus;
using std::multiplies;
auto compute(span<const double> u,
    span<const double> v) -> double
{
    return transform_reduce(
        u.begin(), u.end(),
        v.begin(), 0., plus<double>{},
        multiplies<double>{});
}

void elsewhere(double* x, double* y,
               unsigned N)
{
    return compute(span(x, N), span(y, N));
}
```

- Non-owning view type for a contiguous range
- No memory management
- Numeric operations can often be expressed in terms of existing arrays in memory, irrespective of how they got there and who cleans up after they expire
- `span` is designed to be that input for such functions
- Cheap to copy: essentially a pointer and a size
- STL container like interface

Exercise 6.6:

`examples/spans` is a directory containing the compute function as shown here. Notice how this function is used directly using C++ array types as arguments instead of spans, and indirectly when we only have pointers.

Ranges

RANGES

```cpp
std::vector v{ 1, 2, 3, 4, 5, 6, 7, 8, 9 };
// before std::ranges we did this...
std::reverse(v.begin(), v.end());
std::rotate(v.begin(), v.begin() + 3, v.end());
std::sort(v.begin(), v.end());
```

```cpp
std::vector v{ 1, 2, 3, 4, 5, 6, 7, 8, 9 };
namespace sr = std::ranges;
sr::reverse(v);
sr::rotate(v, v.begin() + 3);
sr::sort(v);
```

- The `<ranges>` header defines a set of algorithms taking "ranges" as inputs instead of pairs of iterators
- A `range` is a **concept** : something with `sr::begin()`, which returns an entity which can be used to iterate over the elements, and `sr::end()` which returns a sentinel which is equality comparable with an iterator, and indicates when the iteration should stop.
- `sr::sized_range` : the range knows its size in constant time
- `input_range`, `output_range` etc. based on the iterator types
- `borrowed_range` : a type such that its iterators can be returned without the danger of dangling.
- `view` is a range with constant time copy/move/assignment

USING RANGES FROM STD OR FROM RANGE-V3

```
// cxx220ranges
#include <version>
#ifdef __cpp_lib_ranges
  #include<ranges>
  namespace sr = std::ranges;
  namespace sv = sr::views;
#elif __has_include (<range/v3/all.hpp>)
  #include<range/v3/all.hpp>
  namespace sr = ranges;
  namespace sv = sr::views;
  #warning Using ranges-v3 3rd party library
#else
#error No suitable header for C++20 ranges was found!
#endif
```

- The C++20 `<ranges>` library is based on the open source `range-v3` library. Parts of the `range-v3` library were adopted for C++20, more might be added in C++23.
- Even if the standard library shipping with some compilers do not have many features of `<ranges>`, one can start using them, with a redirecting header, which makes use of another standard library feature
- Including `<version>` results in the definition of library feature test macros, which can be used to choose between different header files

Our examples are actually written using a redirecting header as shown here. Compilation with GCC uses the compiler's own version. Compilation with Clang uses the `range-v3` version.

FUN WITH RANGES AND VIEWS

```
// examples/ranges0.cc
#include <ranges>
#include <span>
auto sum(std::ranges::input_range auto&& seq) {
    std::iter_value_t<decltype(seq)> ans{};
    for (auto x : seq) ans += x;
    return ans;
}
auto main() -> int
{
    //using various namespaces;
    cout << "vector   : " << sum(vector(  { 9,8,7,2 } )) << "\n";
    cout << "list     : " << sum(list(    { 9,8,7,2 } )) << "\n";
    cout << "valarray : " << sum(valarray({ 9,8,7,2 } )) << "\n";
    cout << "array    : "
        << sum(array<int,4>({ 9,8,7,2 } )) << "\n";
    cout << "array    : "
        << sum(array<string, 4>({ "9"s,"8"s,"7"s,"2"s } )) << "\n";
    int A[]{1,2,3};
    cout << "span(built-in array) : " << sum(span(A)) << "\n";
}
```

FUN WITH RANGES AND VIEWS

- The `ranges` library gives us many useful concepts describing sequences of objects.
- The function template `sum` in `examples/ranges0.cc` accepts any input range, i.e., some entity whose iterators satisfy the requirements of an `input_iterator`.
- Notice how we obtain the value type of the range
- Many STL algorithms have `range` versions in C++20. They are functions like `sum` taking various kinds of ranges as input.
- The range concept is defined in terms of
  - the existence of an iterator type and a sentinel type.
  - the iterator should behave like an iterator, e.g., allow `++it`, `*it` etc.
  - it should be possible to compare the iterators with other iterators or with a sentinel for equality.
  - A `begin()` function returning an iterator and an `end()` function returning a sentinel

JÜLICH Forschungszentrum

FUN WITH RANGES AND VIEWS

```cpp
// examples/iota.cc
#include <ranges>
#include <iostream>
auto main() -> int {
    namespace sv = std::views;
    for (auto i : sv::iota(1UL)) {
        if ((i+1) % 10000UL == 0UL) {
            std::cout << i << ' ';
            if ((i+1) % 100000UL == 0UL)
                std::cout << '\n';
            if (i >= 100000000UL) break;
        }
    }
}
```

- All containers are ranges, but not all ranges are containers
- `std::string_view` is a perfectly fine range. Has iterators with the right properties. Has `begin()` and `end()` functions. It does not own the contents, but "ownership" is not part of the idea of a range.
- We could take this further by creating `views` which need not actually contain any objects of the sequence, but simply fake it when we dereference the iterators.
- Example: the standard view `std::views::iota(integer)` gives us an infinite sequence of integers starting at a given value.

FUN WITH RANGES AND VIEWS

```
// examples/iota.cc
#include <ranges>
#include <iostream>
auto main() -> int {
    namespace sv = std::views;
    for (auto i : sv::iota(1UL)) {
        if ((i+1) % 10000UL == 0UL) {
            std::cout << i << ' ';
            if ((i+1) % 100000UL == 0UL)
                std::cout << '\n';
            if (i >= 100000000UL) break;
        }
    }
}
```

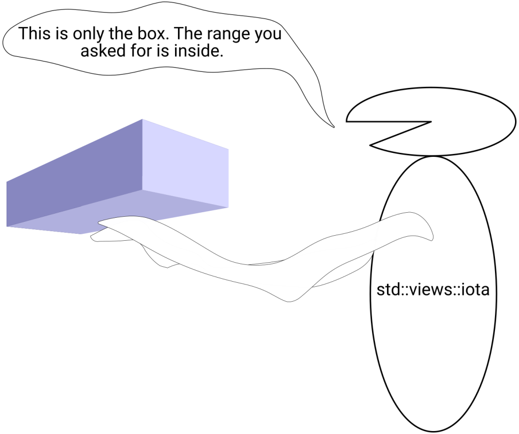
- All containers are ranges, but not all ranges are containers
- `std::string_view` is a perfectly fine range. Has iterators with the right properties. Has `begin()` and `end()` functions. It does not own the contents, but "ownership" is not part of the idea of a range.
- We could take this further by creating `views` which need not actually contain any objects of the sequence, but simply fake it when we dereference the iterators.
- Example: the standard view `std::views::iota(integer)` gives us an infinite sequence of integers starting at a given value.

FUN WITH RANGES AND VIEWS

```cpp
// examples/iota.cc
#include <ranges>
#include <iostream>
auto main() -> int {
    namespace sv = std::views;
    for (auto i : sv::iota(1UL)) {
        if ((i+1) % 10000UL == 0UL) {
            std::cout << i << ' ';
            if ((i+1) % 100000UL == 0UL)
             std::cout << '\n';
            if (i >= 100000000UL) break;
        }
    }
}
```

- All containers are ranges, but not all ranges are containers
- `std::string_view` is a perfectly fine range. Has iterators with the right properties. Has `begin()` and `end()` functions. It does not own the contents, but “ownership” is not part of the idea of a range.
- We could take this further by creating `views` which need not actually contain any objects of the sequence, but simply fake it when we dereference the iterators.
- Example: the standard view `std::views::iota(integer)` gives us an infinite sequence of integers starting at a given value.

FUN WITH RANGES AND VIEWS



- All containers are ranges, but not all ranges are containers
- `std::string_view` is a perfectly fine range. Has iterators with the right properties. Has `begin()` and `end()` functions. It does not own the contents, but "ownership" is not part of the idea of a range.
- We could take this further by creating `views` which need not actually contain any objects of the sequence, but simply fake it when we dereference the iterators.
- Example: the standard view `std::views::iota(integer)` gives us an infinite sequence of integers starting at a given value.

FUN WITH RANGES AND VIEWS

```cpp
#include <ranges>
#include <iostream>
auto main() -> int {
    namespace sv = std::views;
    for (auto i : sv::iota(1UL) ) {
        if ((i+1) % 10000UL == 0UL) {
            std::cout << i << ' ';
            if ((i+1) % 100000UL == 0UL)
                std::cout << '\n';
            if (i >= 100000000UL) break;
        }
    }
}
```

- All containers are ranges, but not all ranges are containers
- `std::string_view` is a perfectly fine range. Has iterators with the right properties. Has `begin()` and `end()` functions. It does not own the contents, but "ownership" is not part of the idea of a range.
- We could take this further by creating `views` which need not actually contain any objects of the sequence, but simply fake it when we dereference the iterators.
- Example: the standard view `std::views::iota(integer)` gives us an infinite sequence of integers starting at a given value.

BORROWED RANGES

```cpp
// examples/dangling0.cc
auto get_vec() {
    std::vector v{ 2, 4, -1, 8, 0, 9 };
    return v;
}
auto main() -> int {
    auto v = get_vec();
    auto iter = std::min_element(v.begin(),
                                 v.end());
    std::cout << "Minimum " << *iter << "\n";
}
```

- The `min_element` function finds the minimum element in a range and returns an iterator

Example from a CPPCon 2020 talk by Tristan Brindle. Link.

BORROWED RANGES

```cpp
// examples/dangling0.cc
auto get_vec() {
    std::vector v{ 2, 4, -1, 8, 0, 9 };
    return v;
}
auto main() -> int {
    auto v = get_vec();
    auto iter = sr::min_element(v);

    std::cout << "Minimum " << *iter << "\n";
}
```

- The `min_element` function finds the minimum element in a range and returns an iterator
- The version from the ranges library takes only a range

Example from a CPPCon 2020 talk by Tristan Brindle. Link.

BORROWED RANGES

```cpp
// examples/dangling0.cc
auto get_vec() {
    std::vector v{ 2, 4, -1, 8, 0, 9 };
    return v;
}
auto main() -> int {

    auto iter = sr::min_element(get_vec());

    std::cout << "Minimum " << *iter << "\n";
}
```

- The `min_element` function finds the minimum element in a range and returns an iterator
- The version from the ranges library takes only a range
- It may be tempting to directly feed the output from a function to the algorithm. But, we would receive an iterator to a container that is already destructed, i.e., a dangling iterator. Dereferencing should therefore lead to a SEGFAULT.

Example from a CPPCon 2020 talk by Tristan Brindle. Link.

BORROWED RANGES

```cpp
// examples/dangling0.cc
auto get_vec() {
    std::vector v{ 2, 4, -1, 8, 0, 9 };
    return v;
}
auto main() -> int {

    auto iter = sr::min_element(get_vec());

    std::cout << "Minimum " << *iter << "\n";
}
```

- The `min_element` function finds the minimum element in a range and returns an iterator
- The version from the ranges library takes only a range
- It may be tempting to directly feed the output from a function to the algorithm. But, we would receive an iterator to a container that is already destructed, i.e., a dangling iterator. Dereferencing should therefore lead to a SEGFAULT.
- In reality, what happens is this!

Example from a CPPCon 2020 talk by Tristan Brindle. Link.

```
error: no match for ‘operator*’ (operand type is 'std::ranges::dangling')
   19 |     std::cout << "Minimum value is " << *iter << "\n";
```


JÜLICH Forschungszentrum

BORROWED RANGES

```cpp
// examples/dangling0.cc
auto get_vec() {
    std::vector v{ 2, 4, -1, 8, 0, 9 };
    return v;
}
auto main() -> int {

    auto iter = sr::min_element(get_vec());

    std::cout << "Minimum " << *iter << "\n";
}
```

- The ranges algorithms are written with overloads such that when you pass an R-value reference of a container as input, the output type is `ranges::dangling`, an empty `struct` with no operations defined.
- `iter` here will be deduced to be of type `ranges::dangling`, and hence `*iter` leads to that insightful error message.

```
error: no match for ‘operator*’ (operand type is 'std::ranges::dangling')
   19 |     std::cout << "Minimum value is " << *iter << "\n";
```

- When the input was an L-value reference, the algorithm returning the iterator returned a valid iterator.
- Therefore: valid use cases work painlessly, and invalid ones result in actionable insights from the compiler!

BORROWED RANGES

```cpp
// examples/dangling1.cc
static std::vector u{2, 3, 4, -1, 9};
static std::vector v{3, 1, 4, 1, 5};
auto get_vec(int c) -> std::span<int> {
    return { (c % 2 == 0) ? u : v };
}
auto main(int argc, char* argv[]) -> int {
    auto iter = sr::min_element(get_vec(argc));
    // iter is valid, even if its parent span
    // has expired.
    std::cout << "Minimum " << *iter << "\n";
}
```

- Sometimes, an iterator can point to a valid element even when the “container” (imposter) has been destructed. `span`, `string_view` etc. do not own the elements in their range.
- No harm in returning real iterators of these objects, even if they are R-values. Even in this case, there is no danger of dangling.
- A `borrowed_range` is a range so that its iterators can be returned from a function without the danger of dangling, i.e.,
  it is an L-value reference or
  has been explicitly certified to be a borrowed range .

BORROWED RANGES

```
// examples/dangling1.cc
static std::vector u{2, 3, 4, -1, 9};
static std::vector v{3, 1, 4, 1, 5};
auto get_vec(int c) -> std::span<int> {
    return { (c % 2 == 0) ? u : v };
}
auto main(int argc, char* argv[]) -> int {
    auto iter = sr::min_element(get_vec(argc));
    // iter is valid, even if its parent span
    // has expired.
    std::cout << "Minimum " << *iter << "\n";
}
```

- Sometimes, an iterator can point to a valid element even when the "container" (imposter) has been destructed. `span`, `string_view` etc. do not own the elements in their range.
- No harm in returning real iterators of these objects, even if they are R-values. Even in this case, there is no danger of dangling.
- A `borrowed_range` is a range so that its iterators can be returned from a function without the danger of dangling, i.e., it is an L-value reference or has been explicitly certified to be a borrowed range .

```
template <class T>
concept borrowed_range = range<T> &&
         ( is_lvalue_reference_v<T> || enable_borrowed_range<remove_cvref_t<T>> )
```

VIEW ADAPTORS

```cpp
namespace sv = std::views;
std::vector v{1,2,3,4,5};
auto v3 = sv::take(v, 3);
// v3 is some sort of object so
// that it represents the first
// 3 elements of v. It does not
// own anything, and has constant
// time copy/move etc. It's a view.

// sv::take() is a view adaptor
```

- A `view` is a range with constant time copy, move etc. Think `string_view`
- A view adaptor is a function object, which takes a "viewable" range as an input and constructs a view out of it. `viewable` is defined as "either a `borrowed_range` or already a view.
- View adaptors in the `<ranges>` library have very interesting properties, and make some new ways of coding possible.

VIEW ADAPTORS

```
Adaptor(Viewable) -> View
Viewable | Adaptor -> View
V | A1 | A2 | A3 ... -> View

Adaptor(Viewable, Args...) ->  View
Adaptor(Args...)(Viewable) ->  View
Viewable | Adaptor(Args...) -> View
```

- A `view` itself is trivially viewable.
- Since a view adaptor produces a view, successive applications of such adaptors makes sense.
- If an adaptor takes only one argument, it can be called using the pipe operator as shown. These adaptors can then be chained to produce more complex adaptors.
- For adaptors taking multiple arguments, there exists an equivalent adaptor taking only the viewable range.

VIEW ADAPTORS

```
Adaptor(Viewable) -> View
Viewable | Adaptor -> View
V | A1 | A2 | A3 ... -> View

Adaptor(Viewable, Args...) ->  View
Adaptor(Args...)(Viewable) ->  View
Viewable | Adaptor(Args...) -> View
```

- A `view` itself is trivially viewable.
- Since a view adaptor produces a view, successive applications of such adaptors makes sense.
- If an adaptor takes only one argument, it can be called using the pipe operator as shown. These adaptors can then be chained to produce more complex adaptors.
- For adaptors taking multiple arguments, there exists an equivalent adaptor taking only the viewable range.

VIEW ADAPTORS

```
Adaptor(Viewable) -> View
Viewable | Adaptor -> View
V | A1 | A2 | A3 ... -> View
```

```
Adaptor(Viewable, Args...) ->  View
Adaptor(Args...)(Viewable) ->  View
Viewable | Adaptor(Args...) -> View
```

- A `view` itself is trivially viewable.
- Since a view adaptor produces a view, successive applications of such adaptors makes sense.
- If an adaptor takes only one argument, it can be called using the pipe operator as shown. These adaptors can then be chained to produce more complex adaptors.
- For adaptors taking multiple arguments, there exists an equivalent adaptor taking only the viewable range.

JÜLICH Forschungszentrum

VIEW ADAPTORS

```
Adaptor(Viewable) -> View
Viewable | Adaptor -> View
V | A1 | A2 | A3 ... -> View
```

```
Adaptor(Viewable, Args...) ->  View
Adaptor(Args...)(Viewable) ->  View
Viewable | Adaptor(Args...) -> View
```

- A `view` itself is trivially viewable.
- Since a view adaptor produces a view, successive applications of such adaptors makes sense.
- If an adaptor takes only one argument, it can be called using the pipe operator as shown. These adaptors can then be chained to produce more complex adaptors.
- For adaptors taking multiple arguments, there exists an equivalent adaptor taking only the viewable range.

VIEW ADAPTORS

```
Adaptor(Viewable) -> View
Viewable | Adaptor -> View
V | A1 | A2 | A3 ... -> View

Adaptor(Viewable, Args...) ->  View
Adaptor(Args...)(Viewable) ->  View
Viewable | Adaptor(Args...) -> View
```

- A `view` itself is trivially viewable.
- Since a view adaptor produces a view, successive applications of such adaptors makes sense.
- If an adaptor takes only one argument, it can be called using the pipe operator as shown. These adaptors can then be chained to produce more complex adaptors.
- For adaptors taking multiple arguments, there exists an equivalent adaptor taking only the viewable range.

So what are we going to do with this ?

VIEW ADAPTORS

Pretend that you want to verify that $sin^2(x) + cos^2(x) = 1$

VIEW ADAPTORS

Pretend that you want to verify that $sin^2(x) + cos^2(x) = 1$

- Start with a range of integers from 0 to $N = 10000$.
- $R_0 = \{0, 1, 2, 3...\}$

VIEW ADAPTORS

Pretend that you want to verify that $sin^2(x) + cos^2(x) = 1$

- Start with a range of integers from 0 to $N = 10000$.
- Map the integer range to real numbers in the range $[0, 2\pi)$

- $R_0 = \{0, 1, 2, 3...\}$
- $R_1 = T_{10} R_0 = T(n \mapsto \frac{n\pi}{N}) R_0$

VIEW ADAPTORS

Pretend that you want to verify that $sin^2(x) + cos^2(x) = 1$

- Start with a range of integers from 0 to $N = 10000$.
- Map the integer range to real numbers in the range $[0, 2\pi)$
- Evaluate $sin^2(x) + cos^2(x) - 1$ over the resulting range

- $R_0 = \{0, 1, 2, 3...\}$
- $R_1 = T_{10} R_0 = T(n \mapsto \frac{n\pi}{N}) R_0$
- $R_2 = T_{21} R_1 = T(x \mapsto (sin^2(x) + cos^2(x) - 1)) R_1$

$$\begin{aligned} R_2 &= T_{21} R_1 = T_{21} T_{10} R_0 \\ &= R_0 | T_{10} | T_{21} \\ &= R_0 | (T_{10} | T_{21}) \end{aligned}$$

VIEW ADAPTORS

Pretend that you want to verify that $sin^2(x) + cos^2(x) = 1$

- Start with a range of integers from 0 to $N = 10000$.
- Map the integer range to real numbers in the range $[0, 2\pi)$
- Evaluate $sin^2(x) + cos^2(x) - 1$ over the resulting range
- If absolute value of any of the values in the result exceeds ϵ, we have found a counter example

- $R_0 = \{0, 1, 2, 3...\}$
- $R_1 = T_{10} R_0 = T(n \mapsto \frac{n\pi}{N}) R_0$
- $R_2 = T_{21} R_1 = T(x \mapsto (sin^2(x) + cos^2(x) - 1)) R_1$

$$\begin{aligned} R_2 &= T_{21} R_1 = T_{21} T_{10} R_0 \\ &= R_0 | T_{10} | T_{21} \\ &= R_0 | (T_{10} | T_{21}) \end{aligned}$$

VIEW ADAPTORS

Pretend that you want to verify that $sin^2(x) + cos^2(x) = 1$

- Start with a range of integers from 0 to $N = 10000$.
- Map the integer range to real numbers in the range $[0, 2\pi)$
- Evaluate $sin^2(x) + cos^2(x) - 1$ over the resulting range
- If absolute value of any of the values in the result exceeds ϵ, we have found a counter example
- Intuitive left-to-right readability

- $R_0 = \{0, 1, 2, 3...\}$
- $R_1 = T_{10}R_0 = T(n \mapsto \frac{n\pi}{N})R_0$
- $R_2 = T_{21}R_1 = T(x \mapsto (sin^2(x) + cos^2(x) - 1))R_1$

$$\begin{aligned} R_2 &= T_{21}R_1 = T_{21}T_{10}R_0 \\ &= R_0|T_{10}|T_{21} \\ &= R_0|(T_{10}|T_{21}) \end{aligned}$$

VIEW ADAPTORS

Pretend that you want to verify that $sin^2(x) + cos^2(x) = 1$

- Start with a range of integers from 0 to $N = 10000$.
- Map the integer range to real numbers in the range $[0, 2\pi)$
- Evaluate $sin^2(x) + cos^2(x) - 1$ over the resulting range
- If absolute value of any of the values in the result exceeds ϵ, we have found a counter example
- Intuitive left-to-right readability

- $R_0 = \{0, 1, 2, 3...\}$
- $R_1 = T_{10} R_0 = T(n \mapsto \frac{n\pi}{N}) R_0$
- $R_2 = T_{21} R_1 = T(x \mapsto (sin^2(x) + cos^2(x) - 1)) R_1$

$$\begin{aligned} R_2 &= T_{21} R_1 = T_{21} T_{10} R_0 \\ &= R_0 | T_{10} | T_{21} \\ &= R_0 | (T_{10} | T_{21}) \end{aligned}$$

```
find . -name "*.cc" | xargs grep "if" | grep -v "constexpr" | less
```

VIEW ADAPTORS

```
find . -name "*.cc" | xargs grep "if" | grep -v "constexpr" | less
```

- Command line of Linux, Mac OS ...

VIEW ADAPTORS

```
find . -name "*.cc" | xargs grep "if" | grep -v "constexpr" | less
```

- Command line of Linux, Mac OS ...
- Small utilities. Each program does one thing, and does it well.

VIEW ADAPTORS

```
find . -name "*.cc" | xargs grep "if" | grep -v "constexpr" | less
```

- Command line of Linux, Mac OS ...
- Small utilities. Each program does one thing, and does it well.
- There is a way to chain them together with the pipe

VIEW ADAPTORS

```
find . -name "*.cc" | xargs grep "if" | grep -v "constexpr" | less
```

- Command line of Linux, Mac OS ...
- Small utilities. Each program does one thing, and does it well.
- There is a way to chain them together with the pipe
- Overall usefulness of the tool set is amplified exponentially!

VIEW ADAPTORS

```
find . -name "*.cc" | xargs grep "if" | grep -v "constexpr" | less
```

- Command line of Linux, Mac OS ...
- Small utilities. Each program does one thing, and does it well.
- There is a way to chain them together with the pipe
- Overall usefulness of the tool set is amplified exponentially!
- What about writing something similar in C++ ?

VIEW ADAPTORS

Pretend that you want to verify that $sin^2(x) + cos^2(x) = 1$

VIEW ADAPTORS

Pretend that you want to verify that $sin^2(x) + cos^2(x) = 1$

- Start with a range of integers from 0 to $N = 10000$. `R0 = iota(0, N)`

VIEW ADAPTORS

Pretend that you want to verify that $sin^2(x) + cos^2(x) = 1$

- Start with a range of integers from 0 to $N = 10000$. `R0 = iota(0, N)`
- Map the integer range to real numbers in the range $[0, 2\pi)$, i.e., perform the transformation $n \mapsto \frac{2\pi n}{N}$ over the range: `R1 = R0 | transform([](int n) -> double { return 2*pi*n/N; })`

VIEW ADAPTORS

Pretend that you want to verify that $sin^2(x) + cos^2(x) = 1$

- Start with a range of integers from 0 to $N = 10000$. `R0 = iota(0, N)`
- Map the integer range to real numbers in the range $[0, 2\pi)$, i.e., perform the transformation $n \mapsto \frac{2\pi n}{N}$ over the range: `R1 = R0 | transform([](int n) -> double { return 2*pi*n/N; })`
- `R2 = R1 | transform([](double x) -> double { return sin(x)*sin(x)+cos(x)*cos(x); } );`

VIEW ADAPTORS

Pretend that you want to verify that $sin^2(x) + cos^2(x) = 1$

- Start with a range of integers from 0 to $N = 10000$. `R0 = iota(0, N)`
- Map the integer range to real numbers in the range $[0, 2\pi)$, i.e., perform the transformation $n \mapsto \frac{2\pi n}{N}$ over the range: `R1 = R0 | transform([](int n) -> double { return 2*pi*n/N; })`
- `R2 = R1 | transform([](double x) -> double { return sin(x)*sin(x)+cos(x)*cos(x); } );`
- If absolute value of any of the values in the result exceeds ϵ, we have found a counter example `if (any_of(R2, [](auto x){return fabs(x) > eps;})) ...`

VIEW ADAPTORS

```cpp
auto main() -> int {
    namespace sr = std::ranges;
    namespace sv = std::views;
    const auto pi = std::acos(-1);
    constexpr auto npoints = 10'000'00UL;
    constexpr auto eps = 100 * std::numeric_limits<double>::epsilon();
    auto to_0_2pi = [=](size_t idx) -> double {
        return std::lerp(0., 2*pi, idx * 1.0 / npoints);
    };
    auto x_to_fx = [ ](double x) -> double {
        return sin(x) * sin(x) + cos(x) * cos(x) - 1.0;
    };
    auto is_bad = [=](double x){ return std::fabs(x) > eps; };

    auto res = sv::iota(0UL, npoints) | sv::transform(to_0_2pi)
                   | sv::transform(x_to_fx);
    if (sr::any_of(res, is_bad) ) {
        std::cerr << "The relation does not hold.\n";
    } else {
        std::cout << "The relation holds for all inputs\n";
    }
}
```

VIEW ADAPTORS

- The job of each small transform in the previous example was small, simple, easily verified for correctness.

VIEW ADAPTORS

- The job of each small transform in the previous example was small, simple, easily verified for correctness.
- The view adaptors allow us to chain them to produce a resulting range

VIEW ADAPTORS

- The job of each small transform in the previous example was small, simple, easily verified for correctness.
- The view adaptors allow us to chain them to produce a resulting range
- Algorithms like `std::range::any_of` work on ranges, so they can work on the views resulting from chained view adaptors.

VIEW ADAPTORS

- The job of each small transform in the previous example was small, simple, easily verified for correctness.
- The view adaptors allow us to chain them to produce a resulting range
- Algorithms like `std::range::any_of` work on ranges, so they can work on the views resulting from chained view adaptors.
- No operation is done on any range when we create the variable `res` above.

VIEW ADAPTORS

- The job of each small transform in the previous example was small, simple, easily verified for correctness.
- The view adaptors allow us to chain them to produce a resulting range
- Algorithms like `std::range::any_of` work on ranges, so they can work on the views resulting from chained view adaptors.
- No operation is done on any range when we create the variable `res` above.
- When we try to access an element of the range in the `any_of` algorithm, one element is taken on the fly out of the starting range, fed through the pipeline and catered to `any_of`

VIEW ADAPTORS

- The job of each small transform in the previous example was small, simple, easily verified for correctness.
- The view adaptors allow us to chain them to produce a resulting range
- Algorithms like `std::range::any_of` work on ranges, so they can work on the views resulting from chained view adaptors.
- No operation is done on any range when we create the variable `res` above.
- When we try to access an element of the range in the `any_of` algorithm, one element is taken on the fly out of the starting range, fed through the pipeline and catered to `any_of`
- `any_of` does not process the range beyond what is necessary to establish its truth value. The remaining elements in the result array are never calculated.

Exercise 6.7:

The code used for the demonstration of view adaptors is `examples/trig_views.cc`. Build this code with GCC and Clang. If the version of your compiler does not have a usable `<ranges>` header, we can use a redirecting header `<cxx20ranges>` examples. When the compiler implements the ranges library, it includes `<ranges>`. Otherwise, it tries to include equivalent headers from the `rangev3` library. It also defines alias namespaces `sr` and `sv` for `std::ranges` and `std::std::views`. To compile, you would need to have the location of this redirecting header in your include path:

```
g++ -std=c++20 -I course_home/local/include trig_views.cc
./a.out

clang++ -std=c++20 -stdlib=libc++ -I course_home/local/include trig_views.cc
./a.out
```

Exercise 6.8:

The trigonometric relation we used is true, so not all possibilities are explored. In `examples/trig_views2.cc` there is another program trying to verify the bogus claim $sin^2(x) < 0.99$. It's mostly true, but sometimes it isn't, so that our `if` and `else` branches both have work to do. The lambdas in this program have been rigged to print messages before returning. Convince yourself of the following:

- The output from the lambdas come out staggered, which means that the program does not process the entire range for the first transform and then again for the second ...
- Processing stops at the first instance where `any_of` gets a `true` answer.

VIEW ADAPTORS

```
// examples/gerund.cc
    using itertype = std::istream_iterator<std::string>;
    std::ifstream fin { argv[1] };
    auto gerund = [](std::string_view w) { return w.ends_with("ing"); };
    auto in = sr::istream_view<std::string>(fin);
    std::cout << (in | sv::filter(gerund)) << "\n";

```

- `sr::istream_view<T>` creates an (input) iterable range from an input stream. Each element of this range is of the type `T`.
- `sv::filter` is a view adaptor, which when applied to a range, produces another containing only the elements satisfying a given condition
- In the above, `std::cout` is shown writing out a range. This works via a separate header file included in `gerund.cc` called `range_output.hh`, which is provided to you with the course material. Ranges in C++20 are not automatically streamable to the standard output.

VIEW ADAPTORS

A program to print the alphabetically first and last word entered on the command line, excluding the program name.

```
// examples/views_and_span.cc
auto main(int argc, char* argv[]) -> int
{
    if (argc < 2) return 1;
    namespace sr = std::ranges;
    namespace sv = std::views;

    std::span args(argv, argc);
    auto str = [](auto cstr) -> std::string_view { return cstr; };
    auto [mn, mx] = sr::minmax(args | sv::drop(1) | sv::transform(str));

    std::cout << "Alphabetically first = " << mn << " last = " << mx << "\n";
}
```

STL utilities

CHRONO: THE TIME LIBRARY

- `namespace std::chrono` defines many time related functions and classes (include file: `chrono`)
- `system_clock` : System clock
- `steady_clock` : Steady monotonic clock
- `high_resolution_clock` : To the precision of your computer's clock
- `steady_clock::now()` : nanoseconds since 1.1.1970
- `duration<double>` : Abstraction for a time duration. Uses `std::ratio<>` internally

Exercise 6.9: chrono_demo.cc

THE TIME LIBRARY

```cpp
// examples/chrono_demo.cc
#include <iostream>
#include <chrono>
#include <vector>
#include <algorithm>
#include <ranges>
bool is_prime(unsigned n);
auto main() -> int
{
    using namespace std::chrono;
    namespace sr = std::ranges;
    namespace sv = std::views;
    std::vector<unsigned> primes;
    auto t = steady_clock::now();
    sr::copy(sv::iota(0UL, 10000UL) | sv::filter(is_prime), back_inserter(primes));
    std::cout << "Primes till 10000 are ... " << '\n';
    for (unsigned i : primes) std::cout << i << '\n';
    auto d = steady_clock::now() - t;
    std::cout<<"Prime search took " << duration<double>(d).count() << " seconds\n";
}
```

CALENDAR AND DATES WITH `STD::CHRONO`

```cpp
auto current_year() -> std::chrono::year
{
    using namespace std::chrono;
    year_month_day date { floor<days>(system_clock::now()) };
    return date.year();
}
auto main(int argc, char* argv[]) -> int
{
    using namespace std::chrono;
    using namespace std::chrono_literals;
    auto Y0 { current_year() };
    auto Y1 = Y0 + years{100};
    if (argc > 1) Y1 = year{std::stoi(argv[1])};
    if (argc > 2) Y0 = year{std::stoi(argv[2])};
    if (Y1 < Y0) std::swap(Y1, Y0);

    for (auto y = Y0; y < Y1; ++y) {
        auto d = y / February / Sunday[5];
        if (d.ok())
            std::cout << static_cast<int>(y) << "\n";
    }
}
```


JÜLICH Forschungszentrum

CALENDAR...

Exercise 6.10:

The programs `examples/feb.cc` and `examples/advent.cc` demonstrate the use of the calendar facilities of the C++ standard library. Familiarise yourself with them.

RANDOM NUMBER GENERATION

- Convenient, flexible, powerful random number library providing high quality (pseudo-)random numbers in standard C++ without any external libraries.
- Include `random`. Namespace `std::random`

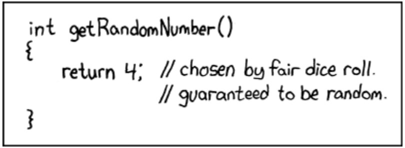


Figure: Source XKCD: `http://xkcd.com`

RANDOM NUMBER GENERATION

- Share a common structure
- Uniform random generator engine with (hopefully) well tested properties
- Distribution generator which adapts its input to a required distribution

$$p(n) = \frac{m^n e^{-m}}{n!}$$

Random distribution

Randomness engine

glue with lambda

```cpp
auto gen = [
    engine = std::mt19937_64{},
    dist=std::poisson_distribution<>{8.5}
           ]() mutable {
    return dist(engine);
};
r = gen();
```

```cpp
std::mt19937_64 engine;
std::poisson_distribution<> dist{8.5};
auto gen = [&dist, &engine] {
    return dist(engine);
};
r = gen();
// if engine or dist are required elsewhere
```

RANDOM NUMBER GENERATORS

```cpp
#include <random>
#include <iostream>
#include <map>
auto main() -> int
{
    auto gen = [ dist=std::poisson_distribution<> {8.5}, engine=std::mt19937_64{} ]
        () mutable { return dist(engine); };
    std::map<int,unsigned> H;
    for (auto i = 0UL; i < 5000000UL; ++i) H[gen()]++;
    for (auto [i, fi] : H) std::cout << i << " " << fi << '\n';
}
```

- `std::mt19937_64` is a 64 bit implementation of Mersenne Twister 19937
- The template `std::poisson_distribution` is a functional implementing the Poission distribution

RANDOM NUMBER GENERATORS

```cpp
std::normal_distribution<> G{3.5, 1.2}; // Gaussian mu = 3.5, sig = 1.2
std::uniform_real_distribution<> U{3.141, 6.282};
std::binomial_distribution<> B{13};
std::discrete_distribution<> dist{0.3, 0.2, 0.2, 0.1, 0.1, 0.1};
// The following is an engine like std::mt19937, but is non-deterministic
std::random_device seed; // int i = seed() will be a random integer
```

- Lots of useful distributions available in the standard
- With one or two lines of code, it is possible to create a high quality generator with good properties and the desired distribution
- `std::random_device` is a non-deterministic random number generator.
  - It is good for setting seeds for the used random number engine
  - It is slower than the pseudo-random number generators

RANDOM NUMBER GENERATOR: EXERCISES

Exercise 6.11:

Make a program to generate normally distributed random numbers with user specified mean and variance, and make a histogram to demonstrate that the correct distribution is produced. Start from `examples/normal_distribution.cc`.

Exercise 6.12:

Make a program to implement a "biased die", i.e., with user specified non-uniform probability for different faces. You will need `std::discrete_distribution<>` Start from `examples/weighted_die.cc`.

EXERCISES

Exercise 6.13:

For a real valued random variable X with normal distribution of a given mean μ and standard deviation σ, calculate the following quantity:

$$K[X] \quad = \quad \frac{\left\langle (X-\mu)^4 \right\rangle}{(\langle (X-\mu)^2 \rangle)^2}$$

Fill in the random number generation parts of the program `examples/K.cc`. Run the program a few times varying the mean and standard deviation. What do you observe about the quantity in the equation above ?

Exercise 6.14: Probabilities with playing cards

The program `examples/cards_problem.cc` demonstrates many topics discussed during this course. It has a `constexpr` function to create a fixed length array to store results, several standard library containers and algorithms as well as the use of the random number machinery for a Monte Carlo simulation. It has extensive comments explaining the use of various features. Read the code and identify the different techniques used, and run it to solve a probability question regarding playing cards.

FORMATTED OUTPUT

```cpp
for (auto i = 0UL; i < 100UL; ++i) {
    std::cout << "i = " << i
        << ", E_1 = " << cos(i * wn)
        << ", E_2 = " << sin(i * wn)
        << "\n";
}
```

```
i = 5, E_1 = 0.55557, E_2 = 0.83147
i = 6, E_1 = 0.382683, E_2 = 0.92388
i = 7, E_1 = 0.19509, E_2 = 0.980785
i = 8, E_1 = 6.12323e-17, E_2 = 1
i = 9, E_1 = -0.19509, E_2 = 0.980785
i = 10, E_1 = -0.382683, E_2 = 0.92388
i = 11, E_1 = -0.55557, E_2 = 0.83147
```

- While convenient and type safe and extensible, the interface of `ostream` objects like `std::cout` isn't by itself conducive to regular well-formatted output

JÜLICH Forschungszentrum

FORMATTED OUTPUT

```cpp
for (auto i = 0UL; i < 100UL; ++i) {
    std::cout << "i = " << i
        << ", E_1 = " << cos(i * wn)
        << ", E_2 = " << sin(i * wn)
        << "\n";
}
```

```
i = 5, E_1 = 0.55557, E_2 = 0.83147
i = 6, E_1 = 0.382683, E_2 = 0.92388
i = 7, E_1 = 0.19509, E_2 = 0.980785
i = 8, E_1 = 6.12323e-17, E_2 = 1
i = 9, E_1 = -0.19509, E_2 = 0.980785
i = 10, E_1 = -0.382683, E_2 = 0.92388
i = 11, E_1 = -0.55557, E_2 = 0.83147
```

- While convenient and type safe and extensible, the interface of `ostream` objects like `std::cout` isn't by itself conducive to regular well-formatted output
- C `printf` often has a simpler path towards visually uniform columnar output, although it is neither type safe nor extensible
- C++ `<iomanip>` header allows formatting with a great deal of control, but has a verbose and inconsistent syntax

FORMATTED OUTPUT

```cpp
for (auto i = 0UL; i < 100UL; ++i) {
    std::cout << fmt::format(
        "i = {:>4d}, E_1 = {:< 12.8f}, "
        "E_2 = {:< 12.8f}\n",
         i, cos(i * wn), sin(i * wn));
}
```

```
i =    5, E_1 =  0.55557023 , E_2 =  0.83146961
i =    6, E_1 =  0.38268343 , E_2 =  0.92387953
i =    7, E_1 =  0.19509032 , E_2 =  0.98078528
i =    8, E_1 =  0.00000000 , E_2 =  1.00000000
i =    9, E_1 = -0.19509032 , E_2 =  0.98078528
i =   10, E_1 = -0.38268343 , E_2 =  0.92387953
i =   11, E_1 = -0.55557023 , E_2 =  0.83146961
```

- While convenient and type safe and extensible, the interface of `ostream` objects like `std::cout` isn't by itself conducive to regular well-formatted output
- C `printf` often has a simpler path towards visually uniform columnar output, although it is neither type safe nor extensible
- C++ `<iomanip>` header allows formatting with a great deal of control, but has a verbose and inconsistent syntax
- C++20 introduced the `<format>` header, which introduces Python like string formatting
- Based on the open source `fmt` library.

FORMATTED OUTPUT

```cpp
for (auto i = 0UL; i < 100UL; ++i) {
    std::cout << fmt::format(
        "i = {:>4d}, E_1 = {:< 12.8f}, "
        "E_2 = {:< 12.8f}\n",
         i, cos(i * wn), sin(i * wn));
}
```

```
i =    5, E_1 =  0.55557023 , E_2 =  0.83146961
i =    6, E_1 =  0.38268343 , E_2 =  0.92387953
i =    7, E_1 =  0.19509032 , E_2 =  0.98078528
i =    8, E_1 =  0.00000000 , E_2 =  1.00000000
i =    9, E_1 = -0.19509032 , E_2 =  0.98078528
i =   10, E_1 = -0.38268343 , E_2 =  0.92387953
i =   11, E_1 = -0.55557023 , E_2 =  0.83146961
```

Perfectly aligned, as all numeric output should be.

- While convenient and type safe and extensible, the interface of `ostream` objects like `std::cout` isn't by itself conducive to regular well-formatted output
- C `printf` often has a simpler path towards visually uniform columnar output, although it is neither type safe nor extensible
- C++ `<iomanip>` header allows formatting with a great deal of control, but has a verbose and inconsistent syntax
- C++20 introduced the `<format>` header, which introduces Python like string formatting
- Based on the open source `fmt` library.
- Elegant. Safe. Fast. Extensible.

FORMATTED OUTPUT

```cpp
// Example of a redirecting header
#include <version>
#ifdef __cpp_lib_format
  #include <format>
  using std::format;
#elif __has_include(<fmt/format.h>)
  #define FMT_HEADER_ONLY
  #include <fmt/core.h>
  using fmt::format;
#else
  #error No suitable header for C++20 format!
#endif
```

- GCC 13 has an implementation. Our redirecting header can help us work with clang as well.
- We can use a redirecting header to use the `fmt` library when the compiler does not have the library feature
- Code simplification and compilation (and runtime) speed $\Longrightarrow$ useful to learn it. Eventually all compilers will have it.

FORMATTED OUTPUT

- `std::format("format string {}, {} etc.", args...)` takes a compile time constant format string and a parameter pack to produce a formatted output string
- `std::vformat` can be used if the format string is not known at compilation time
- If instead of receiving output as a newly created string, output into a container or string is desired, `std::format_to` or `std::format_to_n` are available
- The string contains python style placeholder braces to be filled with formatted values from the argument list
- The braces can optionally contain `id : spec` descriptors. `id` is a 0 based index to choose an argument from `args...` for that slot. `spec` controls how the value is to be written: width, precision, alignment, padding, base of numerals etc. Details of the format specifiers can be found here!

Exercise 6.15:

A simple example demonstrating the text formatting library of C++20 is in `examples/format1.cc`. When this C++20 header is not available in the standard library implementation, we use headers from the `fmt` library giving us approximately the same functionality. Although `fmt` is usually compiled to a static or shared library to link, we define the macro `FMT_HEADER_ONLY` to pretend that we got everything from the standard library.

Exercise 6.16:

The program `examples/word_count.cc` is an improved version of the word counter program from day 4. Here we clear any trailing non-alphabetic characters from strings read as words, e.g., treat "instance," as "instance". We use the ranges algorithms to clean up the string. We then use the formatting library to write the histogram.

std::optional and std::variant

STD::OPTIONAL

- `std::optional<T>` manages an optional value of type `T`, which may or may not be present
- Another way to handle errors during computations to determine a value of some kind
- If the `optional` object has a value, the value resides in the object, i.e., the `optional` type does not do any dynamic memory allocation of its own
- The operators `*` and `->` are given for convenience, so that we can pretend we are dealing with a pointer type when using an `optional`
- If converted to a `bool`, we get **true** if there is a value, **false** otherwise
- Default initialisation as well as initialisation with `nullopt_t` create `optional` objects without value.

```cpp
auto solve_quadratic(double a, double b,
                     double c)
{
    using namespace std;
    optional<pair<double, double>> solution;
    auto D = b * b - 4 * a * c;
    if (D >= 0) {
        auto q = -0.5 * ( b +
                    copysign(sqrt(D), b) );
        solution = make_pair(q / a, c / q );
    }
    return solution;
}
```

Exercise 6.17:

`examples/opt_qsolve.cc` is a small program demonstrating the use of `std::optional`.

STD::VARIANT : A TYPE SAFE UNION

- A `union` is a special kind of class where all the members occupy the same bytes in memory

```
union sameplace { size_t ulong; double real; };
static_assert(sizeof(sameplace) ==
              sizeof(double));
sameplace s;
s.ulong = 0UL;
s.real = 1.0;
cout << s.ulong << "\n";
```

- We can access the elements of a `union` the same way as a `struct` (above).
- Since both members occupy the same bytes, changes to one can affect the other
- If the union contains, e.g., `std::string`, such overriding of bytes would be dangerous.

- `std::variant` is a type safe `union`.
- Unlike the `union`, we don't get to name the different members. The different "members" can be accessed through functions like `std::get<int>(V)`, i.e., we can use the types to select the stored type. We also don't need to say what we are assigning to, since that can be deduced from the type of the object on the right of the `=`
- A `variant` knows what type is currently stored, and calls the destructors etc. when we assign something that would change the stored type

```
variant<double, int, long, string> V;
V = "let's assign a string";
V = 3.141;
// call string destructor and store a double
```

STD::VARIANT : A TYPE SAFE UNION

- A variant type stores one value of any one of a few pre-specified alternatives. To create a `variant` with an integer, a long, a string and a boolean, we would write

```
std::variant<int, long, string, bool> V;
```

- A variant can be assigned a value of any one of its contained types. The variant then remembers the value and the type of the value.

```
V = "0118 999 881 99 9119 725  3"s;
assert(std::holds_alternative<string>(V));
```

- The member function `index()` tells us the zero based index of the currently held type in the list of alternatives for the variant

```
assert(V.index() == 2);
```

- Since the type of the contained object can be changed by an assignment at run time, the variant can not simply have a function `get()` to return the contained value. We have to specify the type of value we want to read as a template argument:

```
cout << get<string>(V);
```

- Unlike the union, we can't store one type and read another

```
V = "0118 999 881 99 9119 725  3"s;
auto num = get<int>(V); //throws exception!
```

- There is also a non throwing version of the accessor:

```
if (auto iptr = get_if<int>(&V); iptr) {
// use iptr as pointer to int value
// Does not get here because get_if<int>
// returns a nullptr in this case.
}
```

STD::VARIANT : A TYPE SAFE UNION

```
using member_t = variant<int, long, string, bool>;
vector<member_t> pop{true, 91, "Monday"s};
for (auto & el : pop) {
    if (auto iptr = get_if<int>(&el)) {
    // *iptr is the int value in the variant el
    } else if (auto lptr = get_if<long>(&el)) {
    // *lptr is the long value in el
    } else if (auto sptr = get_if<string>(&el)) {
    // *sptr is the string value in el
    }
}
```

- Variants can be made to model members of heterogeneous collections, much like pointers to base class in a class hierarchy. The difference is, we can even use built in type like `int`, `double` etc. in a variant based heterogeneous container, because it does not need a class hierarchy!

- Easiest way to model polymorphic behaviour is using a chain of `if ... else if ... else` statements using the `get_if<T>(&v)` function for the different types `T` in the `variant`. `get_if<T>(&v)` returns a valid `T *` if the variant `v` currently holds type `T`. Otherwise it returns `nullptr`.

Exercise 6.18:

The two example programs `examples/variant_0.cc` and `examples/variant_1.cc` demonstrate basic variant usage, such as assignment of values of different types, performing actions based on the content type.

STD::VARIANT : USING STD::VISIT TO SELECT ACTIONS

- Another way to perform different actions based on the currently held type is to use `std::visit`.
- If we have `variant<int, double> V`, `std::visit(F, V)` calls `F(int)` if `V` currently holds an `int` and `F(double)` if `V` currently holds a `double`. `std::visit` unpacks the variant before calling `F` with the stored value. The callable object `F` must have an overload capable of handling the alternatives in the variant
- The overloaded function to be used with `std::visit` can be created in many ways. Three examples in the following boxes:

```cpp
struct my_action {
    auto operator()(int i) { // ... }
    auto operator()(double x) { //  }
};
// ...
std::visit(my_action{}, V);
```

STD::VARIANT : USING STD::VISIT TO SELECT ACTIONS

- Another way to perform different actions based on the currently held type is to use `std::visit`.
- If we have `variant<int, double> V`, `std::visit(F, V)` calls `F(int)` if `V` currently holds an `int` and `F(double)` if `V` currently holds a `double`. `std::visit` unpacks the variant before calling `F` with the stored value. The callable object `F` must have an overload capable of handling the alternatives in the variant
- The overloaded function to be used with `std::visit` can be created in many ways. Three examples in the following boxes:

```cpp
std::visit([](auto upkd) {
    if constexpr (is_same_v<int, decltype(upkd)>) {
        // handle int input
    } else if constexpr (is_same_v<double, decltype(upkd)>) {
        // handle double input
    }
  }, V
);
```

STD::VARIANT : USING STD::VISIT TO SELECT ACTIONS

- Another way to perform different actions based on the currently held type is to use `std::visit`.
- If we have `variant<int, double> V`, `std::visit(F, V)` calls `F(int)` if `V` currently holds an `int` and `F(double)` if `V` currently holds a `double`. `std::visit` unpacks the variant before calling `F` with the stored value. The callable object `F` must have an overload capable of handling the alternatives in the variant
- The overloaded function to be used with `std::visit` can be created in many ways. Three examples in the following boxes:

```
template <class ... Ts> struct stapler : Ts ... { using Ts::operator()... ; };
template <class ... Ts> stapler(Ts ...) -> stapler<Ts...>;
std::visit(stapler{
    [](int i) { /* handle int input */ },
    [](double d) { /* handle double */ }
    }, V
);
```

USING VARIANTS WITH STD::VISITOR

Exercise 6.19:

Example programs `examples/variant_2.cc`, `examples/variant_3.cc` and `examples/variant_4.cc` demonstrate the use of `std::visit` to dispatch different actions depending on the type of the currently held value in a variant. They parallel the approaches in the 3 boxes in the previous slide.

STD::ANY : A TYPESAFE CONTAINER FOR SINGLE VALUES

- A variable of type `std::any` can store 1 value of any type
- Simply by assigning a new value, the contained object is replaced with another of the new type. The variable of type `std::any` is like a box, whose type remains unchanged as the content is swapped. The contained object is indirectly accessed, leading to some overhead.

Exercise 6.20:

`examples/any_demo.cc` demonstrates basic usage of `std::any`.

```cpp
any var = 1;
cout << "Reading int after storing int ... "
     << any_cast<int>(var) << "\n"; // That works
try {
    cout << "Reading float after storing an int ... "
         << any_cast<float>(var) << "\n";
         // This doesn't
} catch (const exception & err) {
    cout << "Float cast after storing int failed. "
         << "Error : " << err.what() << "\n";
}
var = "Europa"s;
map<string, any> config;
config["max_frequency_ghz"] = 3.3;
config["memory_MB"] = 16384;
config["fingerprint_reader"] = true;
```

SEQUENCES OF POLYMORPHIC OBJECTS

(Circle)	(Triangle)	(Circle)	(Circle)

Exercise 6.21:

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.

REGULAR EXPRESSIONS USING C++20

```cpp
constexpr ctll::fixed_string re{ R"xpr(^(https:|http:|www\.)\S*)xpr" };
auto urls_in_input = args | sv::drop(1)
                         | sv::transform([=](auto inp) { return str(inp); })
                         | sv::filter([re](auto inp) { return ctre::search<re>(inp); });
if (auto m = ctre::match<trx>(diststr); m) {
    auto numstr = m.get<1>().to_string();
    // and so on...
}
```

- CTRE: "Compile time regular expressions", header only open source library
- Regular expressions parsed at compile time.
- Smaller binaries than `std::regex`
- Syntax makes excellent use of C++20 features for intuitive handling of regular expressions
- Compile time regex processing is possible, with great performance

REGULAR EXPRESSIONS USING CTRE

Exercise 6.22:

`examples/dist.cc` contains a rudimentary Distance class. Distances can be constructed by giving a value with a unit. Overloaded literal operators allow writing code like `auto d = 14.5_km;` . It is possible to write distances using `std::cout`, or read using `std::cin`. E.g.,

```
$ Enter distance: 13.99_cm
That is 0.1399_m

$ Enter distance: "23   km"
That is 23000_m
```

To read and interpret the input string in the correct units, we make use of regular expressions. Since these can be known at when writing the source code, we use the CTRE library to process our regular expressions. The example demonstrates many different topics explored during the course.

Chapter 7

Type erasure

TYPE ERASURE TECHNIQUE

```cpp
auto f(int i) -> PolyVal;
void elsewhere() {
    std::vector<PolyVal> v;
    v.push_back(1);
    v.push_back(2.0);
    v.push_back("Green"s);

    for (auto&& elem : v) {
        func1(elem);
    }
    PolyVal X = f(i);
}
```

- 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

```cpp
void func1(int x);
void func1(double x);
void func1(std::string x);
auto f(int i) -> PolyVal;
void elsewhere() {
    std::vector<PolyVal> v;
    v.push_back(1);
    v.push_back(2.0);
    v.push_back("Green"s);

    for (auto&& elem : v) {
        func1(elem);
    }
    PolyVal X{3.141};
    // func1(X) should go to func1(double)
    X = PolyVal{"some string"s};
    // func1(X) should now go to func1(string)
    X = f(i);
    // func1(X) should redirect according to what
    // polymorphic value f happens to return
}
```

- 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

```cpp
class PolyVal {
    struct Internal {
        virtual ~Internal() noexcept = default;
        virtual auto clone() const -> std::unique_ptr<Internal> = 0;
        virtual void func1_() const = 0;
    };
    template <class T>
    struct Wrapped : public Internal // continued...

public:
    template <class T>
    PolyVal(const T& var) : ptr{ std::make_unique<Wrapped<T>>(var) } {}
    PolyVal(const PolyVal& other) : ptr { other.ptr->clone() } {}
private:
    std::unique_ptr<Internal> ptr;
};
```

- 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

```
class PolyVal {
    struct Internal {
        virtual ~Internal() noexcept = default;
        virtual auto clone() const -> std::unique_ptr<Internal> = 0;
        virtual void func1_() const = 0;
    };
    template <class T>
    struct Wrapped : public Internal // continued...

public:
    template <class T>
    PolyVal(const T& var) : ptr{ std::make_unique<Wrapped<T>>(var) } {}
    PolyVal(const PolyVal& other) : ptr { other.ptr->clone() } {}
private:
    std::unique_ptr<Internal> ptr;
};
```

- 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

JÜLICH Forschungszentrum

TYPE ERASURE TECHNIQUE

```cpp
class PolyVal {
    template <class T>
    struct Wrapped : public Internal {
        Wrapped(T ex) : obj{ex} {}
        ~Wrapped() noexcept override {}
        auto clone() const -> std::unique_ptr<Internal> override
        {
            return std::make_unique<Wrapped>(obj);
        }
        void func1_() const override { func1(obj); }
        T obj;
    };
};
```

- 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

```
class PolyVal {
    template <class T>
    struct Wrapped : public Internal {
        Wrapped(T ex) : obj{ex} {}
        ~Wrapped() noexcept override {}
        auto clone() const -> std::unique_ptr<Internal> override
        {
            return std::make_unique<Wrapped>(obj);
        }
        void func1_() const override { func1(obj); }
        T obj;
    };
};
```

- 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 ...

```cpp
#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
```

```cpp
#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 `module` mechanism in C++20 offers 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 taken this long to support C++ modules

USING MODULES

```
// examples/hello_m.cc
import <iostream>;

auto main() -> int
{
    std::cout << "Hello, world!\n";
}
```

- 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.
- As of C++20, the standard library is not available as a module. But standard library headers can be imported as “header units”.

```
$ 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)

```cpp
// saxpy.hh
#ifndef SAXPY_HH
#define SAXPY_HH
#include <algorithm>
#include <span>

template <class T>
concept Number = std::floating_point<T>
              or std::integral<T>;
template <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
```

- 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)

```cpp
// usesaxpy.cc
#include <iostream>
#include <array>
#include <vector>
#include <span>
#include "saxpy.hh"

auto main() -> int
{
    using namespace std;
    const array inp1 { 1., 2., 3., 4., 5. };
    const array inp2 { 9., 8., 7., 6., 5. };
    vector outp(inp1.size(), 0.);

    saxpy(10., {inp1}, {inp2}, {outp});
    for (auto x : outp) cout << x << "\n";
    cout << ":::::::::::::::::::::\n";
}
```

- 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)

```cpp
// saxpy.hh -> saxpy.ixx
#ifndef SAXPY_HH
#define SAXPY_HH
#include <algorithm>
#include <span>

template <class T>
concept Number = std::floating_point<T>
              or std::integral<T>;
template <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
```

Make a module interface unit

CREATING A MODULE (EXAMPLE)

```
// saxpy.hh -> saxpy.ixx
#ifndef SAXPY_HH
#define SAXPY_HH
#include <algorithm>
#include <span>

template <class T>
concept Number = std::floating_point<T>
              or std::integral<T>;
template <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
```

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)

```
// saxpy.hh -> saxpy.ixx


#include <algorithm>
#include <span>

template <class T>
concept Number = std::floating_point<T>
              or std::integral<T>;
template <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;
        });
}
```

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)

```cpp
// saxpy.hh -> saxpy.ixx
module;
#include <algorithm>
#include <span>
export module saxpy;

template <class T>
concept Number = std::floating_point<T>
              or std::integral<T>;
template <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;
        });
}
```

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)

```cpp
// saxpy.hh -> saxpy.ixx

export module saxpy;
import <algorithm>;
import <span>;

template <class T>
concept Number = std::floating_point<T>
              or std::integral<T>;
template <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;
        });
}
```

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)

```
// saxpy.hh -> saxpy.ixx

export module saxpy;
import <algorithm>;
import <span>;

template <class T>
concept Number = std::floating_point<T>
              or std::integral<T>;
export template <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;
         });
}
```

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

```
// usesaxpy.cc
#include <iostream>
#include <array>
#include <vector>
#include <span>
#include "saxpy.hh"

auto main() -> int
{
    using namespace std;
    const array inp1 { 1., 2., 3., 4., 5. };
    const array inp2 { 9., 8., 7., 6., 5. };
    vector outp(inp1.size(), 0.);

    saxpy(10., {inp1}, {inp2}, {outp});
    for (auto x : outp) cout << x << "\n";
    cout << ":::::::::::::::::::::\n";
}
```

CREATING A MODULE (EXAMPLE)

```
// usesaxpy.cc
import <iostream>;
import <array>;
import <vector>;
import <span>;
#include "saxpy.hh"

auto main() -> int
{
    using namespace std;
    const array inp1 { 1., 2., 3., 4., 5. };
    const array inp2 { 9., 8., 7., 6., 5. };
    vector outp(inp1.size(), 0.);

    saxpy(10., {inp1}, {inp2}, {outp});
    for (auto x : outp) cout << x << "\n";
    cout << ":::::::::::::::::::::\n";
}
```

Use your module

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

CREATING A MODULE (EXAMPLE)

```cpp
// usesaxpy.cc
import <iostream>;
import <array>;
import <vector>;
import <span>;
import saxpy;

auto main() -> int
{
    using namespace std;
    const array inp1 { 1., 2., 3., 4., 5. };
    const array inp2 { 9., 8., 7., 6., 5. };
    vector outp(inp1.size(), 0.);

    saxpy(10., {inp1}, {inp2}, {outp});
    for (auto x : outp) cout << x << "\n";
    cout << ":::::::::::::::::::::\n";
}
```

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)

```
// usesaxpy.cc
import <iostream>;
import <array>;
import <vector>;
import <span>;
import saxpy;

auto main() -> int
{
    using namespace std;
    const array inp1 { 1., 2., 3., 4., 5. };
    const array inp2 { 9., 8., 7., 6., 5. };
    vector outp(inp1.size(), 0.);

    saxpy(10., {inp1}, {inp2}, {outp});
    for (auto x : outp) cout << x << "\n";
    cout << ":::::::::::::::::::::\n";
}
```

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 2023, 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

Exercise 8.2:

Versions of the `saxpy` program written using header files and then modules can be found in the `examples/saxpy/`. The recipe for building is described in the README.md files in the respective sub-folders. 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. The recipes for building are in the README.md files.

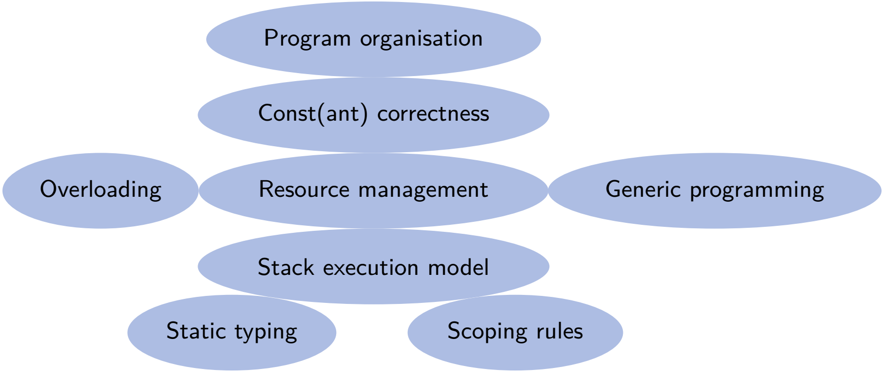
Exercise 8.4:

Each of the above exercise directories contains a `CMakeLists.txt` file. We demonstrate the use of the experimental support for modules in CMake using the combination Clang-16 and Ninja 1.11 to build these simple projects. Test it, and try using it for a different example from the course.

Chapter 9

Closing remarks

C++ "GENES"


Program organisation
Const(ant) correctness
Overloading
Resource management
Generic programming
Stack execution model
Static typing
Scoping rules

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 really 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 really 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

JÜLICH Forschungszentrum

CLOSING REMARKS

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