

The Basics of C++ Programming

Francesco Giacomini

Carlo Battilana

Simone Rossi Tisbeni

INFN and University of Bologna

Ebernburg Castle, 29 Jan - 2 Feb 2024

<https://github.com/giacomini/ebernburg2024>

<https://l.infn.it/12x>



Introduction

Course outline

- Elements of computer architecture and operating systems
- Objects, types, variables
- Expressions
- Statements and structured programming
- Functions
- User-defined types and classes
- Generic programming and templates
- The Standard Library, containers, algorithms
- Compilation model
- Elements of software engineering and supporting tools
- Dynamic memory allocation
- Dynamic polymorphism (aka object-oriented programming)

Platforms and tools

- The reference operating system is Linux (Ubuntu 22.04), but any platform with a recent compiler is fine
- There are installation and configuration guides for:
 - [Linux](#)
 - [Windows](#)
 - [macOS](#)
- Unless you already have a preferred text editor/IDE, we recommend [Visual Studio Code](#), with at least the [C/C++ Extension Pack](#)

Organization

- Monday 14–17:30
- Tuesday to Thursday 9–17:30
- Friday 9–15
- Coffee breaks: 10:30–11 and 15:30–16
- Lunch: 12:30–14
- During the lectures we'll do some short hands-on exercises
- During the last hour of the day we'll work on a week-long project

Useful online resources to keep at hand

- C++ language and library reference
 - <https://cppreference.com/>
 - Somewhat formal, but precise, with examples
- Compiler Explorer
 - <https://godbolt.org/>
 - Compile and run in the browser
- C++ Core Guidelines
 - <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines>
 - How to use C++ correctly

Elements of computer architecture

The language of a computer

- A *computer* is a device that executes programs
- A *program* is a precise sequence of instructions enabling a computer to perform a specific task
- For a computer to understand instructions, these need to be expressed in a language that the computer can understand

Many types of computers

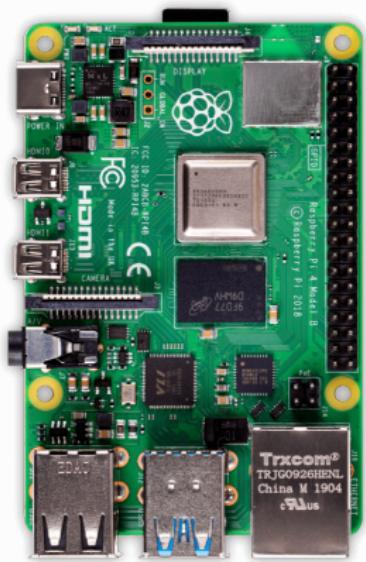


(from <https://www.samsung.com/>)



(from <https://www.ifixit.com/>)

Many types of computers (cont.)



(from <https://www.raspberrypi.org/>)



(from <https://www.fairphone.com/>)

Many types of computers (cont.)



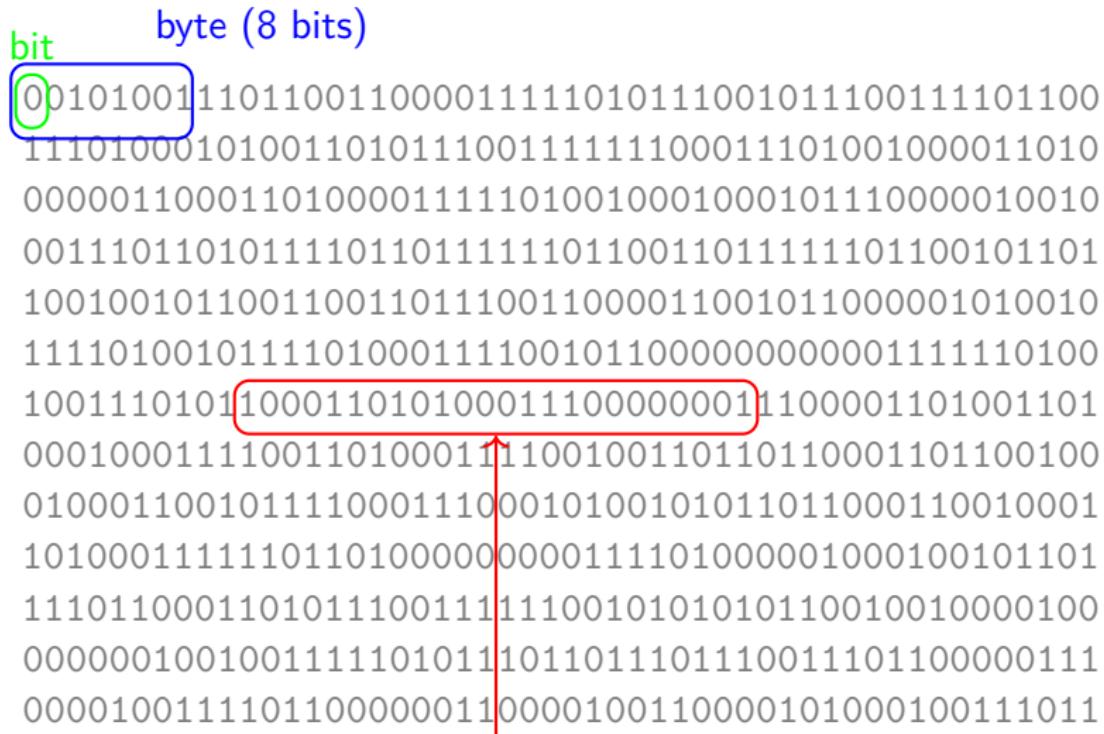
Many types of computers (cont.)



(from <https://wiki.u-gov.it/confluence/display/SCAIUS>)

Computers understand a binary language

bit byte (8 bits)



The image shows a long string of binary digits (0s and 1s) representing a sequence of bytes. A green box highlights the first byte (the first 8 bits), which is 00101001. A red box highlights the 11th byte (the 8th byte from the start of the sequence). A red arrow points from the text 'increment a number by 1' at the bottom to the 11th byte.

00101001110110011000011111010111001011100111101100
1110100010100110101110011111100011101001000011010
00000110001101000011111010010001000101110000010010
001110110101111011011111011001101111101100101101
10010010110011001101110011000011001011000001010010
111101001011110100011110010110000000000111110100
100111010110001101000111000000011100001101001101
00010001111001101000111100100110110110001101100100
01000110010111100011100010100101011011000110010001
1010001111101101000000000111101000001000100101101
1110110001101011100111110010101010110010010000100
0000001001001111101011101101110011101100000111
00001001111011000001110000100110000101000100111011

increment a number by 1 (for my laptop)

Programming language

- A (high-level) programming language is an artificial language to write programs that is closer to humans

```
int increment(int n)
{
    return n + 1;
}
```

- More or less equivalent to the mathematical function

$$\text{increment}(n) = n + 1 \quad \forall n \in \mathbb{Z}$$

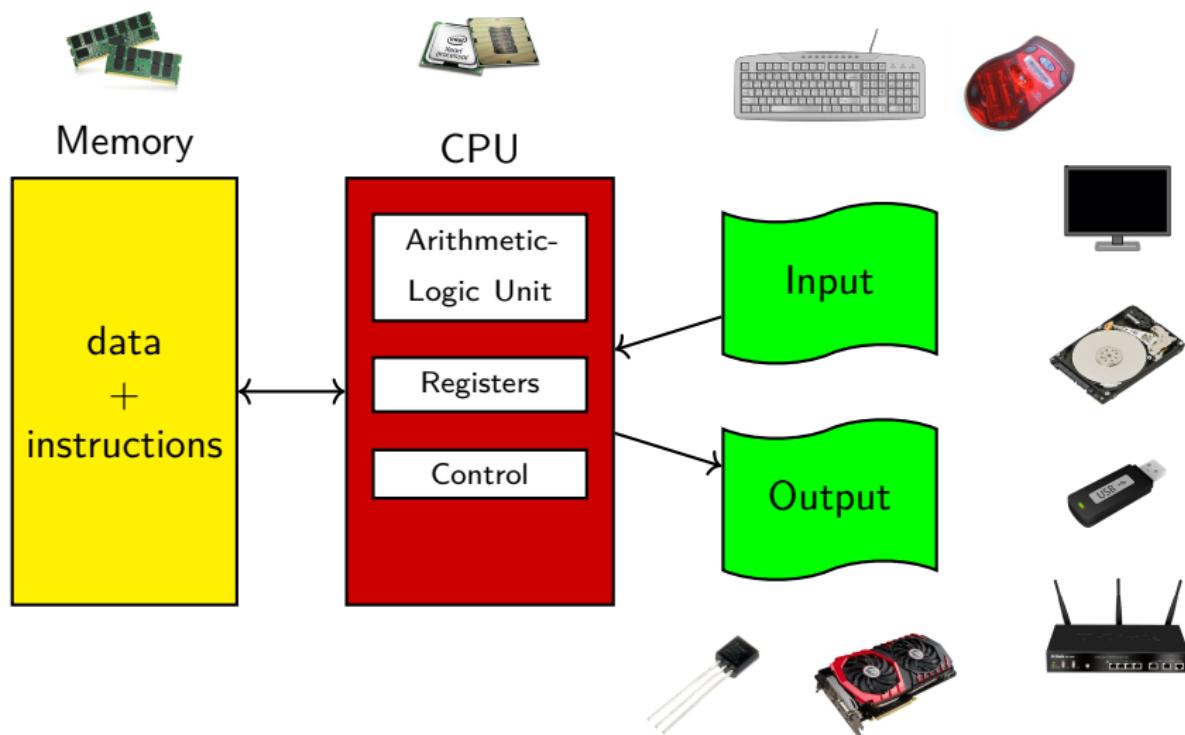
- Some form of translation needs to be applied to the program written in a high-level language to transform it into a program expressed in a binary language
 - A program expressed in a binary language is usually called *executable*

Programming language (cont.)

- To complicate things, the binary language is computer-specific (the correct term is *architecture*-specific)
 - An Instruction Set Architecture (ISA) defines, among other things, the binary instructions understood by a computer implementing that architecture
 - Many ISAs have been defined over the years, many still in use
 - i386, **x86_64**, SPARC, MIPS, **ARM**, VAX, Alpha, RiscV, PowerPC, ...
- The translation from high-level language to binary language is done by other programs
 - **compilers** and interpreters

The Von Neumann architecture

Despite their differences, all architectures are inspired at the architecture first described by John Von Neumann in 1945.



The Von Neumann architecture (cont.)

- Imagine memory as a looooong tape divided into locations whose content you can read and write
 - Each location has the same size (let's assume one byte, i.e. 8 bits)
 - A piece of data can occupy multiple locations
- Each memory location is identified by an index
 - e.g. location at position 8'363'944
- During execution, executables (i.e. binary programs) and the data they manage stay both in memory, typically in different regions
- The CPU fetches binary instructions from memory into its registers and executes them
- Typical instructions:
 - read data (e.g. a number) from a memory location into a register
 - write data (e.g. a number) from a register to a memory location
 - manipulate data (e.g. increment a number or add two numbers) in registers

Also data are binary

```
...01001110110011000011111010111001011100111101100  
1110100010100110101110011111100011101001000011010  
00000110001101000011111010010001000101110000010010  
001110110101111011011111011001101111101100101101  
10010010110011001101110011000011001011000001010010  
1111010010111101000111100101100000000000111110100  
10011101100011011010010110000101101111000000001101  
00010001111001101000111100100110110110001101100100  
01000110010111100011100010100101011011000110010001  
101000111110110100000000111101000001000100101101  
00001001111011000000110000100110000101000100111...
```

- read as **integer number**, the value is 1667850607
- read as **floating-point number**, the value is 4.30511×10^{21}
- read as **character string**, the value is “ciao”

Basics

C++

- There are many programming languages, with very different characteristics
- Why C++?
 - general purpose
 - support for multiple styles of programming (*paradigms*)
 - much used in scientific fields, but also in games, finance, telecommunications, embedded, ...
 - available on most architectures and operating systems
 - efficient
 - ISO standard

A minimal C++ program

- This program does nothing, successfully (details will be explained)

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

- Let's write these 13 characters (including spaces) in a **text document**, which we call `minimal.cpp`, using Visual Studio Code

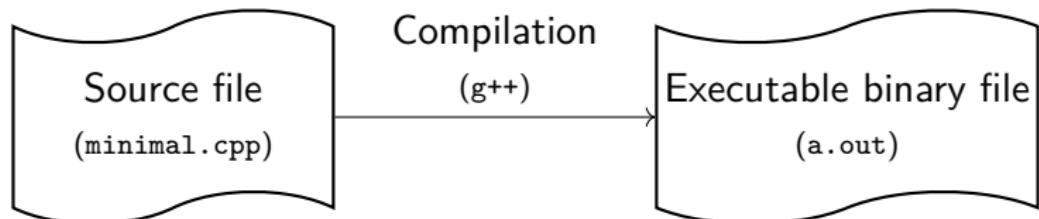
- In C++ terms, this text document is called **source file**
 - Also, let's start using the *shell*, writing commands in a *Terminal* instead of clicking icons with the mouse

```
$ ← this is the shell prompt
$ code minimal.cpp ↵ ← write this command (hit "Enter" at the end)
$ cat minimal.cpp ↵ ← the cat command shows the contents of a text file
int main() {}
$
```

- Before **executing** the program we need to translate it into the language of the computer

A minimal C++ program (cont.)

- In C++ the translation into a binary format is the job of the **compiler**, which produces an **executable** file



```
$ g++ minimal.cpp
$ ls ← the ls command lists the contents of a directory
a.out minimal.cpp
$ ./a.out ← this command executes the program
$ g++ minimal.cpp -o minimal
$ ls
a.out minimal minimal.cpp
$ ./minimal
$
```

- The `-o` option passed to the `g++` command allows to give another name to the final executable

Spaces

- Spaces are (almost) irrelevant

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

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

space needed here

```
intmain(){ }
```

- Tools exist to consistently format source code to improve readability
 - They are customizable
 - You are **required** to use them

```
$ clang-format minimal.cpp  
int main()  
{  
}  
$
```

```
# .clang-format file  
...  
AllowShortFunctionsOnASingleLine: false  
BraceWrapping:  
    AfterFunction:    true  
...
```

- NL.4

Syntax check

- The compiler can translate a program into a binary executable only if the code is syntactically correct

```
intmain() {}
```

```
$ g++ minimal.cpp
minimal.cpp:1:9: error: ISO C++ forbids declaration of 'intmain' with ...
  1 | intmain() {}
      |         ^
$
```

- Error messages are usually precise about the cause of the error, but not always
- Learn to interpret error messages
- Use multiple compilers

Comments

- Code can contain comments

```
int main() // comment until the end of the line
{
}
```

```
int main()
{
    /* possibly multi-line comment
       that goes until the final marker
       and cannot nest */
}
```

- Comments are ignored by the compiler and are equivalent to spaces
- Comments are for humans
 - Prefer expressive code
 - NL.1, NL.2, NL.3

Basic notions of Input and Output

- The input and output system allows a program to interact with the external world
- One mechanism offered by C++ to do input and output is the *I/O streams* interface, which allows the creation and the manipulation of *stream* entities
 - Values are extracted from an input stream
 - Values are inserted into an output stream
- Input and output streams connected to the terminal are automatically available to a program
 - `std::cin`, `std::cout` (and `std::cerr` for errors)
- To extract and insert values, use the stream operators `>>` and `<<`, respectively

Hello

A less minimal C++ program (details will be explained)

```
#include <iostream> // import I/O utilities
#include <string> // import string utilities

int main() // start the program from here
{
    std::cout << "What's your name? "; // print a message on the terminal
    std::string name; // some space is needed in memory for a string
    std::cin >> name; // read a string from the terminal into that space
    std::cout << "Hello, " << name << '\n'; // print a multi-part message
}
```

```
$ g++ hello.cpp -o hello ↵
$ ./hello ↵
What's your name? Francesco ↵
Hello, Francesco
$
```

Objects

- The constructs in a C++ program create, destroy, refer to, access, and manipulate **objects**
- An object is a region of storage (i.e. memory)
 - it has a **type**
 - it has a **lifetime**
 - it can have a **name**
- Objects can contain other objects, called *subobjects*

Types

- A type gives meaning to a piece of storage
 - What's the meaning of a piece of storage that contains the sequence of bits 01100011011010010110000101101111?
 - Read as a sequence of alphabetic characters, it's the letters *c, i, a, o*
 - Read as an integer number, it's 1667850607
 - Read as a floating-point number, it's about 4.30511×10^{21}
- A type identifies a set of values and the operations that can be applied to those values
 - C++ is a **strongly typed** language (mostly)
- A type is also associated with a machine representation for the values belonging to the type

Types (cont.)

- The compiler checks that program instructions are compatible with the type system
 - C++ is a **statically typed** language (mostly)
- C++ defines a few fundamental types and provides mechanisms to build compound types on top of them

Fundamental types

- arithmetic types
 - integral types
 - signed integer types: `short int`, `int`, `long int`, `long long int`
 - unsigned integer types: `unsigned short int`, `unsigned int`,
`unsigned long int`, `unsigned long long int`
 - character types: `char`, `signed char`, `unsigned char`, ...
 - boolean types: `bool`
 - floating-point types: `float`, `double`, `long double`
- `std::nullptr_t`
 - type of the null pointer `nullptr`
- `void`
 - denotes absence of type information

Size and representation may depend on the actual computer architecture where the program runs

int

Type representing a signed integer number

- Set of values: subset of \mathbb{Z}
- Operations: addition, subtraction, multiplication, division, remainder, comparisons, ...
- Representation: 2's complement
- With N bits, values are in the range $[-2^{N-1}, 2^{N-1} - 1]$
- Intended to have the natural size suggested by the architecture. Typical size is 32 bits (4 bytes)
 - $[-2\ 147\ 483\ 648, +2\ 147\ 483\ 647]$
- Default type you should use to count and do integer arithmetic
 - ES.102

7	0111
6	0110
5	0101
4	0100
3	0011
2	0010
1	0001
0	0000
-1	1111
-2	1110
-3	1101
-4	1100
-5	1011
-6	1010
-7	1001
-8	1000

Identifiers

- An *identifier* is a sequence of letters (including _) and digits, starting with a letter
 - Avoid _ at the beginning
- Identifiers are used to name entities in a program
 - Choose meaningful names
 - Get used to write in English
- NL.8

Keywords

The following identifiers are reserved

alignas	consteval	final	or_eq	throw
alignof	constexpr	float	override	true
and	constinit	for	private	try
and_eq	const_cast	friend	protected	typedef
asm	continue	goto	public	typeid
auto	co_await	if	register	typename
bitand	co_return	import	reinterpret_cast	union
bitor	co_yield	inline	requires	unsigned
bool	decltype	int	return	using
break	default	long	short	virtual
case	delete	module	signed	void
catch	do	mutable	sizeof	volatile
char	double	namespace	static	wchar_t
char8_t	dynamic_cast	new	static_assert	while
char16_t	else	noexcept	static_cast	xor
char32_t	enum	not	struct	xor_eq
class	explicit	not_eq	switch	
compl	export	nullptr	template	
concept	extern	operator	this	
const	false	or	thread_local	

Variables

- A variable is an identifier that gives a **name** to an *object*

- ① int i; // declaration; the value is undefined
- ② i = 4321; // assignment of a constant
- ③ int j{1234}; // declaration and initialization in one step
- ④ i = j; // assignment of j's value to i

	i	Memory	j
①		?	
②		4321	
③		4321	1234
④		1234	1234

NB At the end i and j have the same value but remain **distinct** objects

Literals

A literal is a constant **value** of a certain **type** included in the source code

- integer
- floating point
- character
- string
- boolean
- null pointer
- user-defined

Integer literals

decimal non-0 decimal digit followed by zero or more digits

- 1 -98 123456789 -1'234'567'890

binary 0b or 0B followed by binary digits

- 0b1101111010101101
0B111'0101'1011'1100'1101'0001'0101

exadecimal 0x or 0X followed by hexadecimal digits

- -0xdead 0xDEad123f 0XdeAD'123F

octal 0 followed by octal digits

- 01 -077 07'654'321
- N.B. A 0 in front of a number is meaningful!

Integer literals are of type int

std::string

- A *compound (user-defined)* type to represent a string of characters
- Provided by the C++ Standard Library
- Many operations available
- A std::string can be initialized with a string literal, a sequence of escaped or non-escaped characters between double quotes
 - "hello" "hello'\n'world" "hello \"world\\""
 - \n means “newline”
- The type of a string literal is **not** std::string

```
std::string display{"C++ rocks!"};  
display = display + "\n@Ebernburg";
```

concatenate

std::vector<T>

- A compound type *template* to represent a uniform collection of objects of type T, where T can be (almost) any other type
- Provided by the C++ Standard Library
- Many operations available
- A std::vector can be initialized with a sequence of values of a certain type

```
std::vector<int> primes{2, 3, 5, 7, 11};
```

- The [] operator gives access to the i^{th} element, **counting from 0**

```
std::cout << "The 2nd prime number is " << primes[1] << '\n';
```

Expressions

- An expression is a sequence of operators and their operands that specifies a computation
- Literals and variables are typical operands, but there are others

```
1 + 2
i = 1 + 2      // assignment
i == j         // equality comparison
std::sqrt(x) > 1.42
std::cout << "hello, " << name << '\n'
primes[i]
a > 0 ? a - 1 : a + 1
```

- The evaluation of an expression typically produces a result
- Some expressions have *side-effects*
 - They modify the state of the program, i.e. the state of memory, or the external world
- Avoid complicated expressions (**ES.40**)

Operators

Arithmetic	Logical	Comparison	Increment	Assignments	Access	Other
+a	!a	a == b	++a	a = b	a[b]	a(...)
-a	a && b	a != b	--a	a += b	*a	a, b
a + b	a b	a < b	a++	a -= b	&a	? :
a - b		a > b	a--	a *= b	a->b	
a * b		a <= b		a /= b	a.b	
a / b		a >= b		a %= b		
a % b				a &= b		
~a				a = b		
a & b				a ^= b		
a b				a <= b		
a ^ b				a <<= b		
a << b				a >>= b		
a >> b						

Plus: casts, allocation and deallocation, static introspection, ...

- Rules exist for associativity, commutativity and precedence
 - When in doubt use parentheses ([ES.41](#))
- Many operators can be **overloaded** for user-defined types
 - e.g. + to concatenate strings

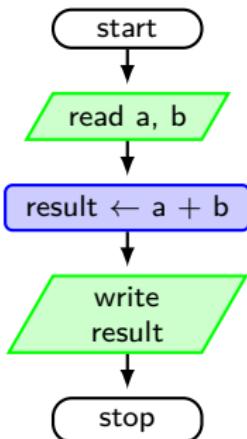
Flow control

Algorithm

A finite sequence of precisely defined steps to solve a problem

Sum of two numbers

A program that reads two numbers from input and writes their sum to output



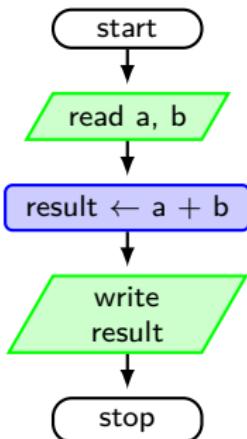
Statement

Statements are units of code that are executed in sequence

- expression statement
- compound statement or block
- declaration statement
- selection statement
- iteration statement
- jump statement
- ...

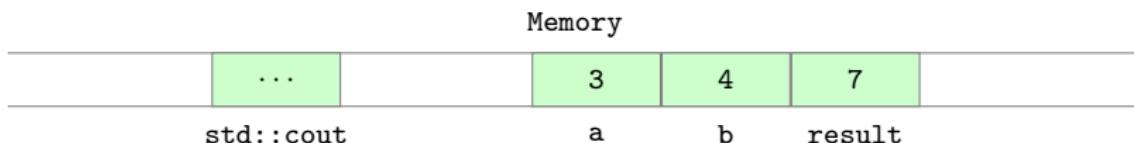
Sum of two numbers

A program that reads two numbers from input and writes their sum to output



```
#include <iostream>

int main()
{
    int a;
    int b;
    std::cin >> a >> b;
    int result{a + b};
    std::cout << result << '\n';
}
```



Expression statement

- An expression followed by a semicolon (;)
- The value of the expression (if any) is discarded
- The expression can have side effects
 - Indeed, we often write statements with the purpose to change data in memory or to do I/O (input/output)

```
b + 2;  
a = b + 2;  
std::cin >> a >> b;  
; // empty statement
```

Compound statement (or block)

A sequence of zero or more statements enclosed between braces ({})

```
{  
    found = true;  
    ++i;  
    int n{3};  
    std::cout << n;  
}
```

```
{ found = true; ++i; ... } // all on one line
```

Declaration statement

- A declaration statement introduces one or more new identifiers into a C++ program, possibly initializing them
 - typically variables, but not only
- A declaration of a **variable** in a **block** makes the variable of **automatic storage duration**, unless otherwise specified
 - the corresponding object is automatically created each time the declaration is executed
 - the corresponding object is automatically destroyed each time the execution reaches the end of the block
- A declaration should introduce only one identifier (**ES.10**)
- A variable should be declared only in the moment it's actually needed (**ES.21**)
- A variable should be initialized at the point of declaration (**ES.20**)
 - There are very few exceptions, if any, to this recommendation

Scope

The scope of a name appearing in a program is the, possibly discontiguous, portion of source code where that name is valid

- block scope
- function scope
- class scope
- namespace scope (including the global one)
- ...

Block scope

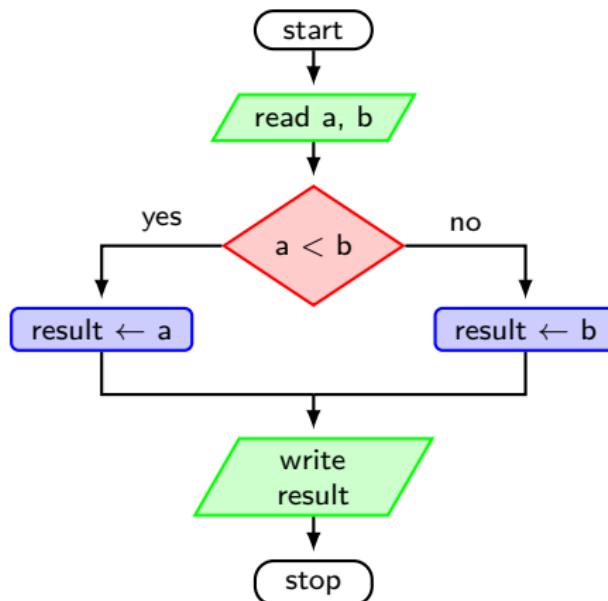
- The scope of a name declared in a block starts at the point of declaration and ends at the }

```
{  
    num + 1;          // error  
    int num;          // scope of num begins only here  
    std::cin >> num; // ok  
}                  // scope of num ends here  
std::cout << num; // error
```

- The scope of a variable should be as small as possible ([ES.5](#))
 - Declare a variable only when it's needed
 - If possible/meaningful, initialize it

The smallest of two numbers

A program that reads two numbers from input and writes the smallest to output



```
#include <iostream>

int main()
{
    int a;
    int b;
    std::cin >> a >> b;
    int result;
    if (a < b) {
        result = a; // (1)
    } else {
        result = b; // (2)
    }
    std::cout << result << '\n'; // (3)
}
```

result cannot be defined in (1) and/or (2), otherwise it would not be visible in (3)

if then else

- Selection statement to choose one of two flows of instructions depending on a boolean condition
- Two (basic) forms:
 - `if (condition-expr) statement else statement`
 - `if (condition-expr) statement`

```
if (a < b) {  
    result = a;    // "true" branch  
} else {  
    result = b;    // "false" branch  
}
```

```
int i{...};  
if (i < 0) {  
    i = -i;    // "true" branch  
}
```

- *condition-expr* can be any expression whose result is of type (convertible to) `bool`, i.e. either `true` or `false`
- *statement* can be a block, possibly including multiple statements
 - In fact, the statement should always be a block

bool

Type representing a boolean value

- Set of values: *false, true*
- Operations: logical conjunction (*and*), disjunction (*or*) and negation (*not*), assignment, comparison
- Size: typically 1 byte
- Representation: like the ints 0 and 1
- Literals: *false* and *true*

```
bool b{true};  
b = i == j;  
bool b2{i != 1234;}  
bool b3{b2 && i < j};    // and  
bool b4{i < j || i < k}; // or  
bool b5{!(i == j)};     // not
```

- Note the use of parenthesis in $!(i == j)$ to have the right precedence of application of operators

assignment

equality

inequality

Logical operations

and True if both operands are true

op1	op2	op1 && op2
false	false	false
false	true	false
true	false	false
true	true	true

op2 is evaluated only if op1 is true

or True if at least one operand is true

op1	op2	op1 op2
false	false	false
false	true	true
true	false	true
true	true	true

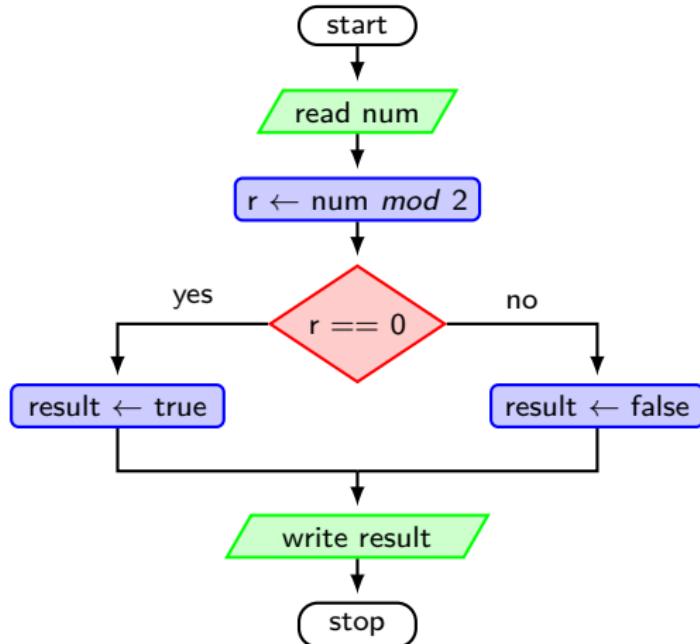
op2 is evaluated only if op1 is false

not Operand's negation (or logical complement)

op	!op
false	true
true	false

Example: is a number even?

A program that reads a number from input and tells if it's even



```
#include <iostream>

int main()
{
    int num;
    std::cin >> num;
    int r{num % 2};
    bool result;
    if (r == 0) {
        result = true;
    } else {
        result = false;
    }
    // print 0/1
    std::cout << result << '\n';
    // print false/true
    std::cout << std::boolalpha
        << result << '\n';
}
```

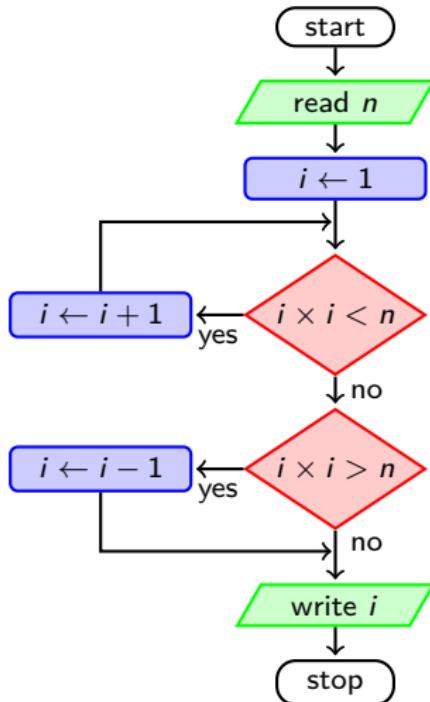
The whole if can be removed defining `bool result{r == 0};`

Exercise: the smallest of three numbers

Write a program that reads three numbers from standard input and writes the smallest one to standard output

Integer square root

Write a program that computes the integer square root of a non-negative integer number, i.e. the largest integer number whose square is not greater than the given number



```
#include <iostream>

int main()
{
    int n;
    std::cin >> n;
    int i{1};
    while (i * i < n) {
        ++i; // equivalent to i = i + 1
    }
    if (i * i > n) {
        --i; // equivalent to i = i - 1
    }
    std::cout << i << '\n';
}
```

while loop

`while (condition-expr) statement`

- Execute repeatedly *statement* until *condition-expr* becomes false
- *condition-expr* is evaluated at the beginning of each iteration
 - If *condition-expr* is already false at the beginning, *statement* is never executed
- *statement* can be any statement
 - It's better if the statement is always a block
- *statement* should modify something so that the evaluation of *condition-expr* may change
 - Otherwise the loop may never terminate

Sum of the first N numbers

Write a program that reads a non-negative integer number N from standard input, computes the sum of the first N numbers and writes the result to standard output

```
#include <iostream>

int main()
{
    int n;
    std::cin >> n;
    int sum{0}; // or just sum{}
    int i{1};
    while (i <= n) {
        sum += i;    // sum = sum + i
        ++i;
    }
    std::cout << sum << '\n';
}
```

```
#include <iostream>

int main()
{
    int n;
    std::cin >> n;
    int sum{0};

    for (int i{1}; i <= n; ++i) {
        sum += i;
    }
    std::cout << sum << '\n';
}
```

for loop

`for (init-statement condition-expropt ; expressionopt) statement`

- Execute *init-statement*, which may be a single ;
 - If *init-statement* contains declarations (e.g. the i variable in the previous example), the scope of the declared names is the loop
- Execute repeatedly *statement* until *condition-expr* becomes false
- *condition-expr* is evaluated at the beginning of each iteration
 - If *condition-expr* is already false at the beginning, *statement* is never executed
 - But *init-statement* is always executed

for loop (cont.)

`for (init-statement condition-expropt ; expressionopt) statement`

- *expression* is evaluated at the end of each iteration
- *statement* and/or *expression* should modify something so that the evaluation of *condition-expr* may change
 - Otherwise the loop may never terminate
- A for loop can always be transformed into a while loop and viceversa
 - Prefer a for loop when there is an obvious variable that controls the loop
 - ES.72, ES.73

Integer square root with a for loop

Write a program that computes the integer square root of a non-negative integer number, i.e. the largest integer number whose square is not greater than the given number

```
#include <iostream>

int main()
{
    int n;
    std::cin >> n;
    int i{1};
    while (i * i < n) {
        ++i;
    }
    if (i * i > n) {
        --i;
    }
    std::cout << i << '\n';
}
```

```
#include <iostream>

int main()
{
    int n;
    std::cin >> n;
    int i{1}; // NB outside the loop
    for ( ; i * i < n; ++i) ; // or {}
    if (i * i > n) {
        --i;
    }
    std::cout << i << '\n';
}
```

Exercise: the largest/smallest of N numbers

Write a program that reads an arbitrary sequence of numbers from standard input and writes the largest (or smallest) one to standard output

Hint 1: press Ctrl-D in the terminal to tell the program there is nothing more to read from standard input

Hint 2: the expression `std::cin.good()` tells if it's still possible to read something from `std::cin`

double

Type representing a floating-point number

- Set of values: subset of \mathbb{R}
- Operations: addition, subtraction, multiplication, division, comparisons, . . .
- Representation: **IEEE 754**
- 64 bits, smallest values $\approx \pm 10^{-308}$, largest values $\approx \pm 10^{308}$
- Precision is about 16 decimal digits
- Literals in the form $sign_{opt} \ significand \ exponent_{opt}$
 - 42.0 1. -1.5 12.34e3 -.34E-3 -1234e-2
 - $\cdots \cdot en / \cdots En$ means $\times 10^n$

float

Type representing a floating-point number

- Set of values: subset of \mathbb{R}
- Operations: addition, subtraction, multiplication, division, comparisons, . . .
- Representation: IEEE 754
- 32 bits, smallest values $\approx \pm 10^{-38}$, largest values $\approx \pm 10^{38}$
- Precision is about 7 decimal digits
- Literals in the form $sign_{opt} \ significand \ exponent_{opt} \ F$
 - 42.0f 1.F -1.5f 12.34e3F -.34E-3f -1234e-2F
 - Same as double but with an f or F suffix

Handle floating-point numbers with care

- Floating-point numbers are **not** real numbers
 - Finite representation \implies there is a previous and a next
- When managing floating-point numbers one should take into account rounding errors and inexact representations

```
float x{1'000'000'000.f};  
std::cout << std::setprecision(10)  
<< x - 32 << ' ' << x << ' ' << x + 32 // 1000000000 1000000000 1000000000  
<< x + 32 - x << ' ' << x - x + 32; // 0 32
```

- Floating-point math in general is not associative
- When comparing floating-point numbers avoid the use of equality
- See [What Every Computer Scientist Should Know About Floating Point Arithmetic](#)

Exercise: accumulate 0.001

Write a program that accumulates the float 0.001 (think a time step of one millisecond) 1'000'000 times and prints the accumulated value. Discuss the result.

Standard mathematical functions

The `cmath` header includes many ready-to-use mathematical functions

- Exponential
- Power
- Trigonometric
- Interpolation
- Hyperbolic
- Floating-point manipulation, classification and comparison
- ...

```
#include <cmath>
...
double x{...};
std::sqrt(x);
std::pow(x, .5);
std::sin(x);
std::log(x);
std::abs(x);
```

Type conversions

- A value of type T1 may be converted **implicitly** to a value of type T2 in order to match the expected type in a certain situation

```
1 + 2.3
```

- between numbers and bool
- between signed and unsigned numbers
- between numbers of different size
- between integral and floating-point numbers
- ...

- Conversions sometimes are surprising
- Conversions can be explicit using `static_cast`

```
1 + static_cast<int>(2.3)
```

- Mechanisms exist to define implicit and explicit conversions involving user-defined types

const-safety

- Data qualified as `const` is logically immutable
- Data that is meant to be immutable should be `const` (**Con.4**)

```
int const x{1'000'000'000}; // or const int
std::cout << x + 32;      // ok, read-only
x += 32;                  // error, trying to modify
int const y;              // error, not initialized and not modifiable later
```

```
std::string const message{"Hello"};
std::cout << message + " Francesco";        // ok, read-only
message += " Francesco";                    // error, trying to modify
std::string const empty_message;            // ok! empty string
```

- Primitive types (e.g. `int`) and user-defined types such as `std::string` behave differently with respect to *default initialization*, i.e. without an explicit initial value
 - User-defined types can define what default initialization means
 - For `std::string` it means “empty string”
 - More later

Functions

Functions

- A function abstracts a piece of code that performs a well-defined task behind a well-defined interface
- A function associates a block of statements with
 - a name
 - a list of zero or more parameters
- A function may return a result
- Let's consider the code that computes the integer square root
 - Let's give it a name → `isqrt`
 - We pass `isqrt` a number → the list of parameters has only one item of type `int`
 - `isqrt` computes a value that we want back → the function returns a value of type `int`
- F.1, F.10

The isqrt function

```
#include <iostream>

int main()
{
    int n;
    std::cin >> n;
    int i{1};
    while (i * i < n) {
        ++i;
    }
    if (i * i > n) {
        --i;
    }
    std::cout << i << '\n';
}
```

Names on the two sides of
the call are independent

```
#include <iostream>

int isqrt(int n) // function definition
{
    int i{1};
    while (i * i < n) {
        ++i;
    }
    if (i * i > n) {
        --i;
    }
    return i; // return statement
}

int main()
{
    int n;
    std::cin >> n;
    int i{isqrt(n)}; // function call
    std::cout << i << '\n';
}
```

Function declaration

- A function **declaration** contains the essential information needed to invoke the function
return-type function-name (parameter-list);
- If the declaration is followed by the actual block of statements (the *implementation* of the function), it is also a **definition**
return-type function-name (parameter-list) { ... }
 - Note the block scope
- Each parameter in the parameter list is of the form
type name_{opt}
 - *type* is mandatory
 - *name* is optional
 - in the declaration, but useful for documentation purposes
 - in the definition if it's not used (F.9)

Function declaration (cont.)

- Parameters are separated by commas
- The parameter list can be empty, i.e. ()
- If the function returns nothing, the return type is void

```
int isqrt(int);           // declaration
int count_words(std::string s) { ... } // definition
double pow(double base, double exp); // declaration
void print(std::string);           // declaration
int generate_random_number() { ... } // definition
```

Returning from a function

- Within the function block, the `return` statement returns the result (and the control) to the calling function
- For a function returning a non-void type
`return expression ;`
The result of *expression* must be convertible to the return type
- For a function returning void
`return;`
At the end of the function, `return;` is optional

Returning from a function (cont.)

- A function has only one entry point, but it may have multiple exit points, i.e. there can be multiple `return` statements

```
bool is_prime(int n) {
    // simple cases
    if (n == 2) { return true; }
    if (n == 1 || n % 2 == 0) { return false; }

    // complex cases
    ...

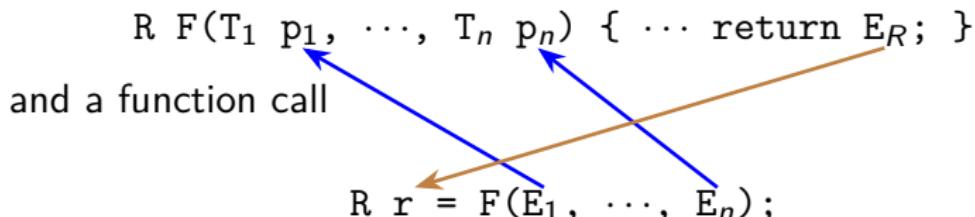
    return result;
}
```

Invoking a function

- Invoking/calling a function is a type of expression of the form $F(E_1, E_2, \dots, E_N)$
 - F is an expression that identifies a function, typically its name
 - The E_i 's are expressions, whose values are “passed” to the function

```
int s = isqrt(24);
std::cout << count_words("Hello, " + name);
print(std::to_string(pow(isqrt(24), 2)));
```

- Given a function



- Each p_i is initialized with the value of expression E_i
- r is initialized with the value of expression E_R

Invoking a function (cont.)

- A function needs to be declared/defined before it's used

```
int main()
{
    ...
    isqrt(num); // error
    ...
}

int isqrt(int n)
{
    ...
}
```

```
int isqrt(int n)
{
    ...
}

int main()
{
    ...
    isqrt(num); // ok
    ...
}
```

```
int isqrt(int);
int main()
{
    ...
    isqrt(num); // ok
    ...
}

int isqrt(int n)
{
    ...
}
```

Function definition

- The lifetime of parameters ends at the end of the block

```
bool is_prime(int n)
{
    ... n ...
} // the lifetime of n ends here
```

- A function can call other functions

```
bool is_prime(int n)
{
    ...
    int div{2};
    int const s{isqrt(n)};
    while (div <= s) {
        ...
    }
}
```

- A function should not be too long (F.3)
- A function should perform a single logical operation (F.2)
- Split long/complex functions into multiple parts, each implemented as a function

Recursive functions

- A function can call itself, directly or indirectly
 - This is called *recursion*
 - Often an elegant alternative to a loop
 - Not easy to master, don't abuse

```
int sum_n(int n)
{
    // assume n >= 0
    if (n == 0) { // base case
        return 0;
    } else {      // recursive case
        return n + sum_n(n - 1);
    }
}
```

Function overloading

- Multiple functions can have the same name
 - But different lists of parameters (number and/or types)
- The compiler chooses the function that best matches the arguments in the call
 - It usually does what is expected, possibly applying appropriate implicit conversions, but not always
 - Compilation error if there is no match or no unique best match
- The return type doesn't matter

```
void foo(int);
int  foo(int, char);
bool foo(double);
int  foo(std::string s);

foo();           // call foo(int)
foo(0, '0');    // call foo(int, char);
foo(0.);        // call foo(double)
foo(std::string{}); // call foo(std::string)
foo(0L);        // long int, ambiguous, error
foo('a');       // call foo(int)
foo("a");       // call foo(std::string)
```

The main function

- The `main` (special) function is the entry point of a program
- It can have two forms
 - `int main() {...}`
 - ... (see later)
- If there is no `return` statement at the end, an implicit `return 0;` is assumed
 - 0 means success, different from 0 means failure
 - Or use `EXIT_SUCCESS` and `EXIT_FAILURE` from `<cstdlib>`
 - The returned value is available to the shell via the `$?` variable

```
#include <cstdlib>

int main() {
    int n;
    std::cin >> n;
    if (std::cin.fail() || n < 0) {
        std::cerr << "Invalid number\n";
        return EXIT_FAILURE;
    }
    ...
}
```

Exercises

- Write a function `pow` that takes two `ints` and computes and returns the value of the first (the base) raised to the power of the second (the exponent)
- Write a function `gcd` that takes two `ints` and computes the Greatest Common Denominator using the [Euclid's algorithm](#)
- Write a function `lcm` that takes two `ints` and computes the Least Common Multiple
- Write a function `is_prime` that takes an `int` and tells if it's a prime number
- More on [edabit](#), [leetcode](#), ...

How do we know the code we write is correct?

- Correctness is the absence of defects (also known as bugs) in a software
- Correctness is the result of the application of multiple good practices to the software development process
- One of the most effective techniques is **testing**
 - Execute the code with reasonable and unreasonable input and see if it behaves according to the expectations
 - The purpose of testing is to (try to) **break** the code

**Testing can reveal the presence of bugs,
not prove their absence!**

- Here we focus on a form of testing called *unit testing*, where the units (of code) under test are, for example, functions
- There are many tools/frameworks to do unit testing (Google Test, Catch, Doctest, Boost.Test, ...)
- Let's use Doctest (<https://github.com/doctest/doctest>)

How to use Doctest

- On **Compiler Explorer** it's available selecting it under “Libraries”
- Otherwise download locally a single *header file* into the directory containing your code, e.g. using wget or curl
- At the top of your C++ file add the lines

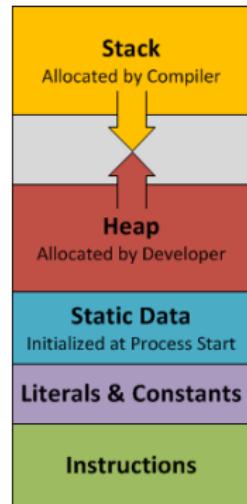
```
#define DOCTEST_CONFIG_IMPLEMENT_WITH_MAIN  
#include "doctest.h"
```

- Don't add main into your file
- After the declarations of the function(s) you want to test, add lines like the following:

```
TEST_CASE("Testing isqrt") {  
    CHECK(isqrt(0) == 0);  
    CHECK(isqrt(9) == 3);  
    CHECK(isqrt(10) == 3);  
    CHECK(isqrt(-1) == 0);  
    ...  
}
```

Memory layout of a process

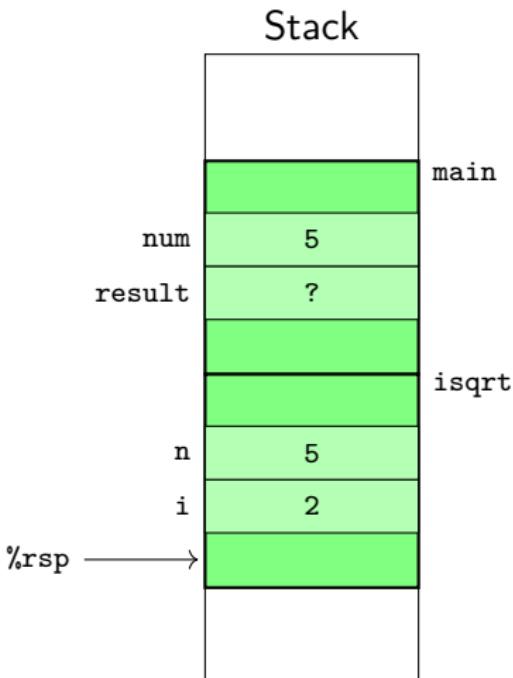
- A process is a running program
- When a program is started the operating system brings the contents of the corresponding file into memory according to well-defined conventions
 - Stack
 - function local variables
 - function call bookkeeping
 - Heap
 - dynamic allocation
 - Global data
 - literals and variables
 - initialized and uninitialized (set to 0)
 - Program instructions



Functions and the stack

```
int isqrt(int n)
{
    int i{1};
    while (i * i < n) {
        ++i;
    }
    if (i * i > n) {
        --i;
    }
    return i;
}

int main()
{
    int num;
    std::cin >> num;
    int result{isqrt(num)};
    std::cout << result << '\n';
}
```



The state of the stack just
before returning from `isqrt`

Pass by-value, return by-value

Given a function

$$R \ F(T_1 \ p_1, \dots, \ T_n \ p_n) \{ \dots \text{return } E_R; \ }$$

and a function call

$$R \ r = F(E_1, \dots, \ E_n);$$

- Each p_i is initialized with the value of expression E_i
 - Every time the function is called a new p_i is created, which gets destroyed at the end of the function
 - NB if E_i is just a variable, p_i is another object (a copy) and changing it inside the function doesn't change the original object corresponding to the variable
- r is initialized with the value of expression E_R

Stack frame

- A piece of memory allocated and dedicated to the execution of a function
- It contains local variables (including function parameters), return address, saved registers, ...
- Managed in a Last-In, First-Out (LIFO) way
- The size of the stack frame is computed by the compiler
- There is a special register (the stack pointer register, `%rsp`) that indicates the frame of the currently running function
- At runtime the allocation/deallocation of a frame consists simply in subtracting/adding that frame size to the stack pointer register

Conditional/ternary operator expression

$expression_{condition} ? expression_{true} : expression_{false}$

```
int gcd(int a, int b)
{
    return (b == 0) ? a : gcd(b, a % b);
}
```

- Evaluate $expression_{condition}$, whose value is of type (convertible to) bool
- If true, $expression_{true}$ is evaluated and the resulting value is the value of the whole expression
- If false, $expression_{false}$ is evaluated and the resulting value is the value of the whole expression
- Similar to an if statement, but usable (and useful) where only an expression is allowed
- The types of $expression_{true}$ and $expression_{false}$ are subject to some constraints, but let's assume that they have to be the same

break and continue for loops

- Within a loop block, the `break` statement allows to terminate the loop
- Within a loop block, the `continue` statement allows to jump to the end of the current iteration of the loop
- The same effect can be obtained with appropriate use of conditionals, but the resulting code may be more complicated
(ES.77)

Object initialization with braces

- An object can be initialized specifying its value between {}

```
int i{123};  
float f{123.F};  
std::string s{"hello"};
```

- Introduced as a *universal* form of initialization that could replace all others, but there are situations where it's not usable
- Protects against *narrowing*, i.e. loss of information caused by implicit conversions

```
double d{1.};           // ok, no conversion  
float f1{1.};          // ok, no information loss  
float f2{d};            // error  
float f3{9'999'999};   // ok, no information loss  
float f3{99'999'999}; // error  
int i{1.};              // error  
int g{d};               // error
```

char

Type representing a character

- Set of values: letters in the alphabet (lower- and upper-case), digits, punctuation marks, some special characters, ...
- Size: typically 1 byte, but not necessarily
- Representation: let's assume the **ASCII** encoding
- Literals: characters between single quotes
 - 'a' 'B' '7' ' ', ' ?' '#' '/'
- Some character literals need to be expressed as \-escaped sequences
 - '\\' '\\\\' '\\n' '\\t' '\\0'
- A **char** is an integral type so it supports integral operations
 - `std::cout << '9' - '0'; // 9`
 - `c < 'z';`

Exercises

- Write a function that takes a `char` and returns the corresponding lowercase character if it is a letter; the same `char` otherwise
 - e.g. `'A'` → `'a'`, `'a'` → `'a'`, `';'` → `';'`
- Write a function that takes two numeric operands of type `double` and one operator of type `char` and returns the result of applying that operator to the two operands. For example if the two operands have values 2.0 and 3.0 respectively and the operator has value `'+'`, then the function returns a result with value 5.0. If the operator is invalid, the function returns 0.

Pointers and references

Passing by value may be inconvenient

- Consider a function that increments an int object

```
void increment(??? n) {  
    // ++n  
}  
  
int number{42};  
increment(number);  
// number == 43
```

- NB The function does not **return** the new value; it modifies the passed object **in place**
- We cannot write `void increment(int n)`, because the function would modify a **copy** of the original object

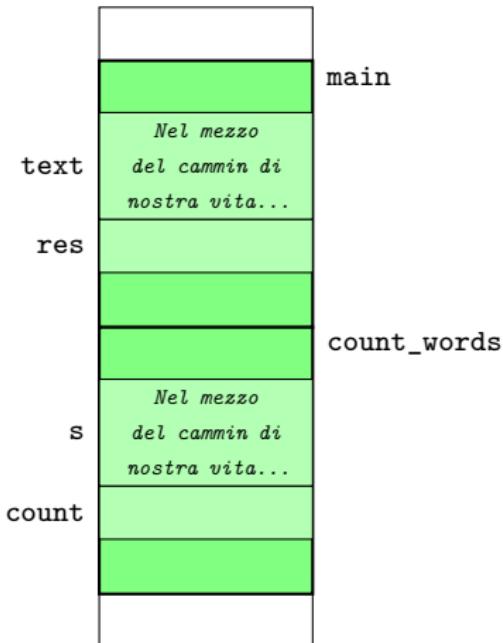
Passing by value may be expensive

Let's consider a function that counts the number of words in a string

```
int count_words(std::string s)
{
    int count{0};
    ... s ...
    return count;
}

int main()
{
    std::string text{...};
    int const res{count_words(text)};
    std::cout << res << '\n';
}
```

- `s` is a **copy** of `text`
- I'm slightly cheating: not all string memory goes on the stack



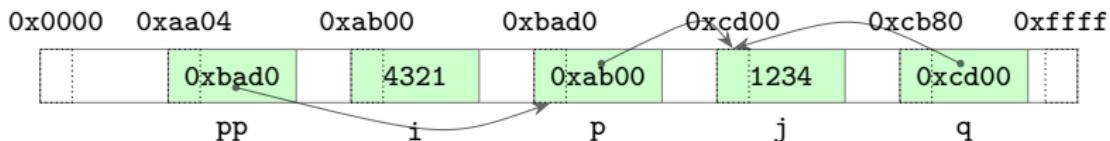
Hexadecimal notation

- Numbers in hexadecimal notation are represented in base sixteen
- Each digit has a value between zero and fifteen (included)
- Sixteen symbols are needed for the digits
 - 0 … 9, A, B, C, D, E, F (also lowercase)
- In C++ numeric literals (i.e. constants) can be expressed in hexadecimal notation, prepending 0x (or 0X) to the hexadecimal representation
 - 0x9 == 9, 0xA == 10, 0x10 == 16
- Easy conversion between hex and binary representation: each hex digit corresponds to four bits
- Hex literals are typically used to represent raw contents of memory or memory addresses

Hex	Dec	Bin
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Pointers

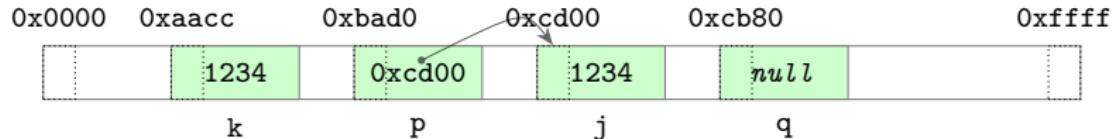
Where in memory does a given object reside?



```
int i{4321};  
int j{1234};  
std::cout << &i; // 0xab00, address-of operator  
std::cout << &j; // 0xcd00  
int* p{&i}; // pointer declarator  
std::cout << &p; // 0xbada  
int** pp{&p}; // &p is of type int**  
p = &j;  
int* q{p}; // p and q point to the same object
```

Note that `int*` is a new type and so is `int**`

Pointers (cont.)



```
int j{1234};  
int* p{&j};  
std::cout << *p; // 1234, dereference operator  
int k{*p}; // k is a copy of j  
*p = 5678;  
++*p; // or ++(*p) for more clarity  
int* q{p};  
q = nullptr;  
*q; // undefined behavior
```

Dereferencing a null pointer is a logical error (i.e. a bug)

Pointers (cont.)

- *address-of* operator: &
 - Given an object, it returns its address in memory
- *dereference* operator: *
 - Given a pointer to an object, it gives access to that very object

Passing a pointer

```
void increment(int* n) {  
    ++(*n);  
}  
  
int number{42};  
increment(&number);  
// number == 43
```

```
int count_words(std::string* s)  
{  
    int count{0};  
    ... *s ...  
    return count;  
}  
  
std::string text{...};  
count_words(&text);
```

- The caller takes the address of the object and passes it to the function
- The function dereferences the pointer to get access to the object
- Be careful not to dereference a null pointer

References

- A variable declared as a **reference** of a type T is another name (an *alias*) for an existing object of type T



```
int i{12};  
int j{56};  
int& ri{i};           // reference declarator  
ri == 12;             // true  
ri = 34;  
ri == 34 && i == 34; // true  
ri = j;  
ri == 56 && i == 56; // true  
int& r;              // error
```

References (cont.)

- A reference must be initialized to refer to a valid object
 - A reference cannot be *null*
- A reference cannot *rebind* (be re-associated) to another object
- For a given type T, T& is a *compound* type, distinct from T
- Dereferencing a pointer gives back a reference to that object

```
int i{42};  
int* p{&i};  
int& r{*p}; // i and r are names for the same object
```

Passing by reference

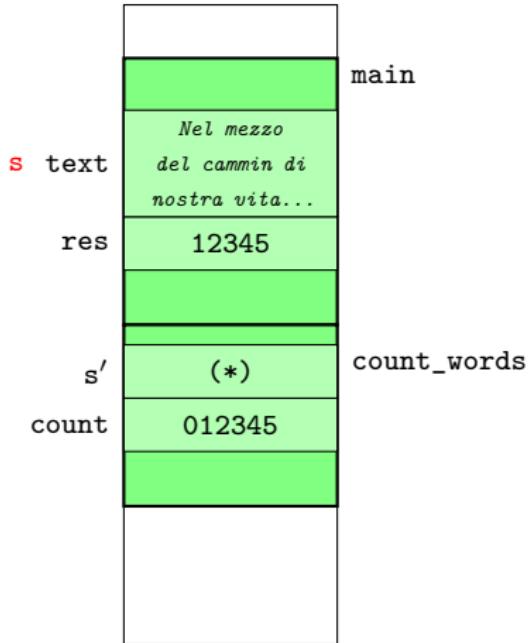
```
void increment(int& n) {  
    ++n;  
}  
  
int number{42};  
increment(number);  
// number == 43
```

- There is no difference in the caller compared to pass-by-value
- There is no difference in the body of the function compared to pass-by-value
- The only visual clue is in the parameter declaration

Passing by reference (cont.)

```
int count_words(std::string& s)
{
    int count{0};
    ... s ... // just use s
    return count;
}

int main()
{
    std::string text{...};
    int const res{count_words(text)};
    std::cout << res << '\n';
}
```



(*) In general it's unspecified if a reference occupies storage. Here probably it does and a pointer (called *s'*) is passed behind the scenes.

const and references

```
std::string text{...};  
std::string& rtext{text};           // ok, can read/modify text via rtext  
std::string const& crtext{text}; // ok, crtext is a read-only view of text
```

```
std::string const text{...};  
std::string& rtext{text};           // error, else could modify text via rtext  
std::string const& crtext{text}; // ok, can only read text via crtext
```

```
int count_words(std::string const& s)  
{  
    // this function can only read from s  
    ...  
}
```

const and pointers

```
std::string text{...};  
std::string* ptext{&text};           // ok, can read/modify name via *ptext  
std::string const* cptext{&text}; // ok, can only read text via *cptext
```

```
std::string const text{...};  
std::string* ptext{&text};           // error, else could modify text via *ptext  
std::string const* cptext{&text}; // ok, can only read text via *cptext
```

How to pass arguments to functions

- For input parameters

- If the type is primitive, pass by value

```
int isqrt(int n);           // good
int isqrt(int const& n); // bad! pessimization
```

- Otherwise pass by const reference

```
int count_words(std::string const&);
```

- For output parameters, prefer to return a value or pass by non-const reference

```
int read_from_cin();        // may be more difficult to overload for other types
void read_from_cin(int& n);
```

- For input-output parameters, pass by non-const reference

```
void to_lowercase(std::string& s);
```

- Parameter passing

Returning a reference

- A function can return a reference only if the referenced object survives the end of the function
 - Otherwise, in the caller, the reference would refer to an object that doesn't exist anymore
- In particular do not return a reference to a function local variable
 - F.43

```
// bad
int& add(int a, int b) {
    int result{a + b};
    return result;
}
```

```
// acceptable
int& increment(int& a) {
    ++a;
    return a;
}
```

- Useful to chain multiple function calls on the same object

```
std::string& to_lower(std::string& s) { ... ; return s; }
std::string& trim_right(std::string& s) { ... ; return s; }
std::string& trim_left(std::string& s) { ... ; return s; }

std::string s{...};
trim_left(trim_right(tolower(s)));
```

range-for loop

for (range-declaration : range-expression) statement

- Simplified form of a *for* loop, to iterate on all the elements of a *range* (sequence), such as a string of characters ([ES.71](#))
- Execute repeatedly *statement* for all the elements of the range
- *range-declaration* declares a variable of the same type of an element of the range
 - Can (and should) be a (const) reference
- *range-expression* represents the range to iterate over
- More on ranges later

```
std::string s{"Hello!"};
for (char& c : s) {
    c = std::toupper(c);
}

for (int i : {1, 2, 3, 4, 5}) {
    std::cout << i << ' ';
}
```

auto

Let the compiler deduce the type of a variable from the initializer,
i.e. from the expression used to initialize the object

```
auto z;                      // error, no initializer
auto i = 0;                   // int
auto const f = 0.F;           // float const
auto r = i + f;               // float
auto s = std::string{"hello"}; // std::string
auto& rs = s;                // std::string&
auto const& cri = i;          // int const&
auto j = rs;                  // std::string
auto& g = f;                 // float const&
auto p = &i;                  // int*
auto q = p;                  // int*
auto* t = p;                 // int*
```

- auto never deduces a reference
- auto preserves constness of references
- Personal preference to use = instead of {} when initializing an auto variable
 - Both auto i{0} and auto i = 0; are fine

auto/trailing return type

- When the return type of a function is auto the compiler will deduce it from the return statement(s)
 - Only usable in a function definition

```
auto isqrt(int n)    // auto deduced as int
{
    int result;
    ...
    return result;
}
```

- If there are multiple return statements their expressions must have the same type
- Trailing return type

```
int foo(...);
auto foo(...) -> int; // equivalent
```

Enumerations

- An *enumeration* is a distinct type with named constants, called *enumerators* ([Enum.2](#))

```
enum class Operator { Plus, Minus, Multiplies, Divides };
```

- When used, an enumerator is specified with the name of the enumeration followed by the scope-resolution operator (::)

```
auto op = Operator::Plus; // op is of type Operator
```

- By default, an enumerator has the value of the previous enumerator incremented by one and the first enumerator has value 0
- An enumerator can be given a value explicitly ([Enum.8](#))

```
enum class Operator { Plus = -2, Minus, Multiplies = 42, Divides };
```

The values are respectively -2 , -1 , 42 and 43 .

Enumerations (cont.)

- An enumeration has an integral *underlying type*, by default `int`
- A different underlying type can be selected ([Enum.7](#))

```
enum class byte : unsigned char { };
```

Note that in this case there are no enumerators → a way to define a new integral type different from the underlying type

- All values of the underlying type are valid values for the enumeration object

```
Operator op{55}; // ok
```

- Conversions to the underlying type need to be explicit

```
int i{Operator::Plus}; // error  
int i{static_cast<int>(Operator::Plus)}; // ok
```

Enumerations (cont.)

- There is also a less strict version of an enumeration: the *unscoped* enumeration ([Enum.3](#))

```
enum Operator { Plus, Minus, Multiplies, Divides }; // NB no class
```

- The symbols of the enumerators are visible in the enclosing scope
 - i.e. the enumerators are not specified with the name of the enumeration followed by ::

```
Operator op{Plus}; // ok, no need to use Operator:::
```

- The conversion to the underlying type is implicit

```
int i{Plus}; // ok
```

- Prefer the scoped enumeration

The switch statement

The switch statement transfers control to one of multiple statements, depending on the value of a condition (ES.70)

```
double compute(char op, double left, double right)
{
    double result;

    switch (op) {
        case '+':
            result = left + right;
            break;
        ...
        case '/':
            result = (right != 0.) ? left / right : 0.;
            break;
        default:
            result = 0.;
    }

    return result;
}
```

The switch statement (cont.)

- The condition is an expression whose evaluation gives an integral or enumeration value
 - Cannot switch on strings, for example
- An enumeration plays well with switch statements

```
double compute(Operator op, double left, double right)
{
    switch (op) {
        case Operator::Plus:      return left + right;
        case Operator::Minus:     return left - right;
        case Operator::Multiplies: return left * right;
        case Operator::Divides:   return right != 0. ? left / right : 0.;
        default: return 0.;
    }
}
```

The switch statement (cont.)

- Each statement, possibly compound, is introduced either by a case label or by a default label
- Each case label specifies a unique integral constant
- Typically each statement is followed by a break statement, to jump after the switch, otherwise control *falls through* the next instruction, even if this is part of a statement introduced by another label
 - The compiler typically warns about falling through, but sometimes it's ok and the warning can be silenced with the `[[fallthrough]]` ([ES.78](#)) attribute
- There can be at most one default label, not necessarily at the end ([ES.79](#))
- The same statement can be introduced by multiple case labels

Data abstraction

Data abstraction

- The C++ language has a strong focus on building lightweight data abstractions
 - The source code can use terminology and notation close to the problem domain, making it more expressive
 - There is little (if any) overhead in terms of space or time during execution
- `class` and `struct` are the primary mechanism to define new compound types on top of fundamental types C.1

Data structure

- Let's introduce a type to represent complex numbers
- We can pass/return Complex objects to/from functions, take pointers and references, ...

```
struct Complex {  
    double r; // data member (field)  
    double i;  
};
```

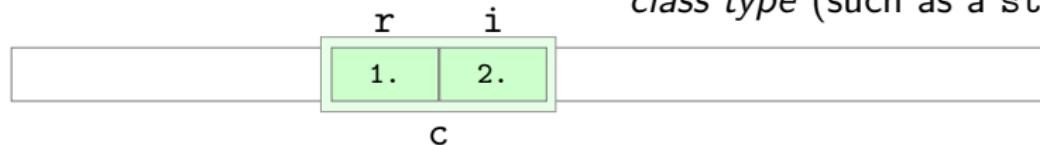
```
double norm2(Complex c);  
Complex sqrt(Complex c);
```

```
Complex c{...};  
double n{ norm2(c) };  
Complex c2{ sqrt(c) };
```

```
Complex& cr{c}; // reference  
Complex* cp{&c}; // pointer
```

```
double norm2(Complex c) {  
    return c.r * c.r + c.i * c.i;  
}  
  
double norm2(Complex const& c) {  
    return c.r * c.r + c.i * c.i;  
}
```

- A Complex is composed of two doubles
- The . (dot) operator allows to access a member of an object of *class type* (such as a struct)



Operations

It's possible to define operations on user-defined types

```
struct Complex {  
    double r;  
    double i;  
};  
  
bool operator==(Complex const& a, Complex const& b) {  
    return a.r == b.r && a.i == b.i;  
}  
  
Complex operator+(Complex const& a, Complex const& b) {  
    return Complex{a.r + b.r, a.i + b.i};  
}  
  
c2 = c1      // generated by the compiler, if used  
c1 == c2  
c1 + c2  
z = z * z + c  
2. * c1  
...
```

Operator overloading

- C++ offers the possibility to define the meaning of most of the operators available for fundamental types when applied to user-defined types
- Syntactically this is done via the definition of appropriate functions
- Given an operator @ , the function name is called `operator@`
- Although there are some constraints on the number and types of the parameters and on the type of the return value, the overloaded operators are just functions
- The behaviour of the overloaded operator should reproduce the behaviour of the original one
- Some behaviours cannot be changed, e.g. associativity

Exercises

- Define some other operators and functions for Complex
- Write a function to compute the solutions of an equation of the form $ax^2 + bx + c = 0$, given the three coefficients (Hint: F.21)

More data abstraction

Imagine to change the Complex type to use the polar form

```
struct Complex {  
    double rho;  
    double theta;  
};
```

- As a consequence, all client code has to change

```
double norm2(Complex const& c) { return c.rho * c.rho; }
```

- Not all combinations of (ρ, θ) are valid
 - $\rho \geq 0, \theta \in [0, 2\pi)$

```
Complex c{-1, -1}; // valid?  
c.rho = -1;         // valid?
```

- We wish we could
 - have more isolation between client code and implementation
 - enforce an internal relation (*class invariant*) between data members

Private representation, public interface

- The internal representation of a data structure should be considered an implementation detail
- The manipulation of objects should happen through a well-defined function-based interface
- Known as **encapsulation** C.3 C.9
- Data member variables (also known as fields) are often named in a special way
 - _ suffix or m_ prefix

```
class Complex {  
private: // cartesian form  
    double r_;  
    double i_;  
public:  
    // associated functions (member functions, also known as methods)  
};  
  
Complex c{1., 2.}; // error  
auto norm2(Complex const& c) {  
    return c.r_ * c.r_ + c.i_ * c.i_; // error  
}
```

Construction

- A class can have a special function, called **constructor**, which is called to initialize the storage of an object when it is created
 - It should initialize all data members, in order to establish the *class invariant* C.40 C.41
- The constructor's name is the same as the class name
- Data members are initialized in the order of declaration and should preferably be initialized in the *member initialization list* C.47 C.49

```
class Complex {  
private:  
    double r_;  
    double i_;  
public:  
    Complex(double x, double y) // no return type  
        : r_{x}, i_{y} // member initialization list  
    { /* nothing else to do */ }  
    ...  
};  
  
Complex c{1., 2.}; // or (1., 2.)
```

Private representation, public interface (cont.)

- The internal representation of a class should be considered an implementation detail
- The manipulation of objects should happen through a well-defined function-based interface

```
class Complex {  
    private:  
        double r_;  
        double i_;  
    public:  
        Complex(double x, double y) : r_{x}, i_{y} {}  
        double real() { return r_; } // member function (method)  
        double imag() { return i_; }  
};  
  
double norm2(Complex c) {  
    return c.real() * c.real() + c.imag() * c.imag();  
}
```

- For a class, private is the default and can be omitted

Private representation, public interface (cont.)

- Member functions (methods) can of course also mutate an object

```
class Complex {  
public:  
    void add(Complex const& other) {  
        r_ += other.r_; // no need to use other.real() (*)  
        i_ += other.i_;  
    }  
    ...  
};  
  
Complex c{1., 2.};  
Complex d{3., 4.};  
c.add(d);  
c.real(); // 4.  
  
c.add(Complex{3., 4.}); // valid  
c.add({3., 4.}); // also valid
```

- (*) Control of access to the private part is per class, not per object

Private representation, public interface (cont.)

- Member functions that don't modify the object should be declared **const**

```
class Complex {  
    double r_;  
    double i_;  
public:  
    Complex(double x, double y) : r_{x}, i_{y} {}  
    auto real() const           // cannot modify this object  
    { return r_; }  
    auto add(Complex const& other) // can modify this object  
    { r_ += ...; }  
};  
  
Complex c{1.,2.};  
c.real();           // ok  
c.add({3.,4.});   // ok  
Complex const cc{1.,2.};  
cc.real();         // ok  
cc.add({3.,4.});  // error
```

Private representation, public interface (cont.)

- It's not rare to have functions with the same name, but in two different forms: one that reads and one that modifies the internal state
 - Example of function overloading

```
class Complex {  
public:  
    double real() const { return r_; }  
    void   real(double d) { r_ = d; }  
    ...  
};
```

```
Complex c{1.,2.};  
c.real(0.);  
c.real(); // return 0.
```

```
Complex const c{1.,2.};  
c.real(0.); // error  
c.real(); // ok
```

- Alternatively, methods can be prefixed with `get_`/`set_`
 - e.g. `double get_real();`, `void set_real(double);`

Member vs free function

```
class Complex {  
public:  
    double norm2() const { // member function  
        return r_ * r_ + i_ * i_;  
    }  
    ...  
};  
  
double norm2(Complex const& c) { // free function  
    return c.real() * c.real() + c.imag() * c.imag();  
}  
  
Complex c{...};  
c.norm2(); // call the member function  
norm2(c); // call the free function
```

- The public part of a class should ideally provide a safe, efficient and complete interface, yet minimal
- Prefer a free function, if possible C.4
 - Extend the functionality of a class without modifying existing code

std::string (cont.)

- A user-defined type to represent a sequence of characters
- Provided by the C++ Standard Library
- Provides many member functions
 - Construction
 - Capacity (e.g. `size()`, `empty()`)
 - Assignment (e.g. `=`, `assign()`)
 - Comparison (`==`, `!=`, `<`, `>`, `<=`, `>=`)
 - Access to a character (e.g. `[]`, `back()`, `front()`)
 - Insertion/removal (e.g. `insert()`, `append()`, `erase()`)
 - Search (e.g. `find()`)
 - ...

Construction (cont.)

- A class can have multiple constructors

```
class Complex {  
    double r_;  
    double i_;  
public:  
    Complex(double x, double y) : r_{x}, i_{y} {}  
    Complex(double x) : r_{x}, i_{0.} {}  
    Complex() : r_{0.}, i_{0.} {} // default constructor  
    ...  
};  
  
Complex c1{1., 2.};  
Complex c2{1.};      // meaning {1., 0.}  
Complex c3;          // or {}, meaning {0., 0.}; () cannot be used here  
Complex cf();        // this is a function declaration!
```

- The default constructor (i.e. the constructor without parameters) can be automatically generated by the compiler, but only if there are no other constructors

Construction (cont.)

- A constructor can **delegate** the construction to another, usually more generic, constructor C.51

```
class Complex {  
    double r_;  
    double i_;  
public:  
    Complex(double x, double y) : r_{x}, i_{y} {}  
    Complex(double x) : Complex{x, 0.} {}          // delegating constructor  
    Complex() : Complex{0., 0.} {}                  // delegating constructor  
    ...  
};  
  
Complex c1{1., 2.};  
Complex c2{1.};        // meaning {1., 0.}  
Complex c3;           // meaning {0., 0.}
```

Construction (cont.)

- A data member can be given a **default initializer**, which is used if the member is not explicitly initialized in the called constructor

```
class Complex {  
    double r_{0.};  
    double i_{0.};  
public:  
    Complex(double x, double y) : r_{x}, i_{y} {}  
    Complex(double x) : r_{x} {} // i_ initialized with 0  
    Complex() = default; // r_ and i_ initialized with 0  
    ...  
};  
  
Complex c1{1., 2.};  
Complex c2{1.};      // meaning {1., 0.}  
Complex c3;         // meaning {0., 0.}
```

- `= default` tells the compiler to generate the default implementation for a *special member function* (such as the default constructor) **C.45 C.80**
 - Do not use `{}`

Construction (cont.)

- For any function, a parameter can be given a default value, in case the corresponding argument is omitted
- Arguments can be omitted starting from the right
- **Default function arguments** can be used also for a constructor

```
class Complex {  
    double r_;  
    double i_;  
public:  
    Complex(double x = 0., double y = 0.) : r_{x}, i_{y} {}  
    ...  
};  
  
Complex c1{1., 2.};  
Complex c2{1.};           // meaning {1., 0.}  
Complex c3;              // meaning {0., 0.}
```

explicit constructor

```
class Complex {  
    ...  
    Complex(double x = 0., double y = 0.) : r_{x}, i_{y} {}  
};  
  
double norm2(Complex const& c) { ... }  
  
norm2(1.); // callable with a double (-> Complex)  
norm2(1); // callable with an int (-> double -> Complex)  
c + 3; // call operator+ with two Complex  
3 + c; // call operator+ with two Complex, only if free function
```

- The (one-parameter) constructor is used for the conversion
- An explicit constructor prevents the implicit conversion

```
class Complex {  
    ...  
    explicit Complex(double x = 0., double y = 0.) ...  
};  
  
norm2(1.); // error  
norm2(Complex{1.}); // ok
```

explicit constructor (cont.)

- An explicit constructor prevents also the implicit construction of an object from a list of arguments between braces
 - e.g. in a function call or in a return statement

```
class Complex {  
    ...  
    Complex(double x, double y) ...  
};  
  
Complex operator+(...)  
{  
    ...  
    return {r, i}; // ok  
}
```

```
class Complex {  
    ...  
    explicit Complex(double x, double y) ...  
};  
  
Complex operator+(...)  
{  
    ...  
    // return {r, i}; // error  
    return Complex{r, i}; // ok  
}
```

- Better start with explicit constructors, especially for the constructor callable with one argument, and relax the constraint later C.46

Pointers and data structures

- *address-of operator:* &
 - Given an object it returns its address in memory
- *dereference operator:* *
 - Given a pointer to an object it returns a reference to that object
- *structure dereference operator:* ->
 - Given a pointer to an object of class/struct type, it returns a reference to a member of that object

```
struct S {  
    int n;  
    void f();  
};  
  
S q{...};  
S* p = &q;  
p->n;        // equivalent to (*p).n  
p->f();      // equivalent to (*p).f()
```

The `this` pointer

Within the body of a member function of a class T, the keyword `this`

- is a pointer of type `T*` (or `T const*` if the method is `const`)
- points to the object for which the method was called

```
struct Foo {  
    void bar() {  
        ... this ...;  
    }  
};  
  
Foo alfa;  
Foo beta;  
alfa.bar(); // inside bar, this is &alfa  
beta.bar(); // inside bar, this is &beta
```

operator@ in terms of operator@=

- Typically, a symmetric operator@ (e.g. operator+) is implemented in terms of a member operator@= (e.g. operator+=)

```
Complex operator+(Complex const& lhs, Complex const& rhs)
{
    Complex result{lhs};    // create the result as a copy of lhs
    return result += rhs;  // add rhs to the result and return it
}
```

- Typically, operator@= returns a reference to the object being operated on

```
class Complex {
    ...
    Complex& operator+=(Complex const& rhs) {
        r_ += rhs.r_;
        i_ += rhs.i_;
        return *this; // *this means self
    }
};
```

Example: rational numbers

- Let's implement a class to represent rational numbers
- The representation is with two integers, such that
 - The fraction is irreducible, i.e. their GCD (Greatest Common Denominator) is 1
 - If the number is negative the sign is kept in the numerator
 - The denominator is different from 0

Example: rational numbers (cont.)

```
class Rational
{
    int n_;
    int d_;
public:
    Rational(int num = 0, int den = 1) : n_{num}, d_{den}
    {
        if (d_ == 0) {
            // construction must fail
        }
        // reduce
        auto const g{ std::gcd(n_, d_) }; // in <numeric>
        n_ /= g;
        d_ /= g;
        // fix sign
        if (d_ < 0) {
            n_ = -n_;
            d_ = -d_;
        }
    }
    ...
};
```

Class invariant

- A **class invariant** is a relation between the data members of a class that constrains the values that such members can assume
 - It defines what is a valid state for an object of that class
 - It must always hold for an object of that class
- The invariant is established by the constructor
- The invariant is preserved by **public** methods
 - The invariant holds when the function is entered \implies the implementation can make assumptions
 - The invariant may be violated during the execution of the function
 - The invariant is re-established before exiting the function
- C.2

assert

Check that a certain boolean expression is satisfied at run time

- for example, a class invariant at the end of a constructor or a condition at the beginning of a function (a *pre-condition*)

```
#include <cassert>

class Rational {
    ...
    Rational(int num = 0, int den = 1) : n{num}, d{den} {
        ...
        assert(std::gcd(n, d) == 1 && d > 0);
    }
    Rational& operator/=(int n) {
        assert(n != 0);
        ...
    }
};

bool operator==(Rational const& l, Rational const& r)
{
    assert(std::gcd(l.num(), l.den()) == 1 && std::gcd(r.num(), r.den()) == 1);
    return l.num() == r.num() && l.den() == r.den();
}
```

assert (cont.)

- If the asserted condition is not satisfied, it means that the state of the program does not conform to the expectations of the programmer, i.e. to the design
- The state may even be corrupted → it's probably wiser to terminate the program as soon as possible to avoid causing damage
- Very useful to identify logical errors (i.e. bugs)
 - Be generous with the number of asserts in your code
- Can be disabled for performance reasons (`g++ -DNDEBUG ...`)
 - Avoid side effects in asserts (e.g. assignments or calls to non-const methods), because they would disappear from the executable

assert (cont.)

- assert (but also Doctest's CHECK) is a *preprocessor macro*
- Macros obey syntactic rules that are different from those of C++ proper

```
assert(Rational{1,2} == Rational{2,4});    // error (*)
assert(Rational(1,2) == Rational(2,4));    // ok
assert((Rational{1,2} == Rational{2,4})); // ok
```

- * macro "assert" passed 3 arguments, but takes just 1
- Macros *expand* to arbitrary text, which is then passed to the real C++ compiler
- A bit more about the preprocessor later in the course

Exceptions

- Exceptions provide a general mechanism to:
 - notify the occurrence of an error in the program execution, typically when a function is not able to accomplish its task, i.e. to satisfy its *post-condition*
 - using a `throw` expression
 - transfer control to a handler defined in a previous function in the call chain, where the exception can be managed
 - using a `try/catch` statement
- Exceptions help separate application logic from error management
- A typical use is in constructors and operators, that have a fixed function signature
- Exceptions cannot be ignored
 - If a handler is not found the program is terminated

Exceptions (cont.)

```
struct E {};  
  
auto function3() {  
    ... // this part is executed  
    throw E{};  
    ... // this part is not executed  
}  
  
auto function2() {  
    ... // this part is executed  
    function3();  
    ... // this part is not executed  
}  
  
auto function1() {  
    try {  
        ... // this part is executed  
        function2();  
        ... // this part is not executed  
    } catch (E const& e) {  
        ... // use e  
    }  
}
```

- An exception is an object
- After being raised (`thrown`), an exception is propagated up the stack of function calls until a suitable catch clause (`handler`) is found
- If no suitable handler (i.e. one compatible with the type of the exception) is found, the program is terminated
- Exceptions should be caught by (`const`) reference
- During *stack unwinding* the stack frames are properly cleaned up

Exceptions in constructors

- An exception is typically raised by a constructor to inform that it is not able to properly initialize the object C.42
 - i.e. it's not able to establish the class invariant
- Let's apply this to Rational, using the standard-provided exception `std::runtime_error`
 - it can be constructed with a string or a string literal
 - it provides a `what()` method to retrieve that string, e.g. in the handler

Exceptions in constructors (cont.)

```
class Rational {  
    ...  
    Rational(int num = 0, int den = 1) : n{num}, d{den} {  
        if (d == 0) {  
            throw std::runtime_error{"denominator is zero"}; // in <stdexcept>  
        }  
        ...  
    }  
};  
  
auto do_computation() {  
    ...  
    Rational{n,m} // m here happens to be 0  
    ...  
}  
  
try {  
    do_computation();  
    ...  
} catch (std::runtime_error const& e) {  
    // manage the error, e.g. log a message on standard error  
    std::cerr << e.what() << '\n';  
}
```

Nested class

- A class can be defined within the definition of another class
 - in either the public or private section

```
class Regression {  
    ...  
    class Result { ... };  
    Result fit() const { ... }  
};  
  
Regression reg;  
...  
Regression::Result result{ reg.fit() }; // or auto result{...}
```

- The nested class can access the private members of the outer class

Type alias

- A type alias is another name for an existing type

```
using Length = double;  
typedef double Length; // equivalent, old alternative  
  
Length len{1.}; // len is of type double
```

- A type alias does **not** introduce a new type

```
void walk(double d) { ... }  
void walk(Length l) { ... } // error, redefinition of walk
```

Type alias (cont.)

- Type aliases are often used to declare types within a class

```
class FitResult { ... };

class Regression {
    ...
public:
    using Result = FitResult;
    Result fit() const { ... }
};

Regression::Result result{ reg.fit() }; // result is of type FitResult
```

Structured binding

- A *structured binding* declaration declares multiple variables and initializes them from values of struct members (and other entities, following a certain *protocol*)

```
struct Point {  
    double x;  
    double y;  
};  
  
Point p{1.,2.};  
auto [a, b] = p;  
std::cout << a << ' ' << b; // print 1 2
```

- The variables can be declared as (const) references to the struct members

```
Point p{1.,2.};  
auto& [a, b] = p; // a is a ref to p.x, b is a ref to p.y  
a = 3.;  
b = 4.;  
std::cout << p.x << ' ' << p.y; // print 3 4
```

Templates

Class template

- Let's consider again the Complex class
- What if we want float members?

```
class Complex {  
    double r;  
    double i;  
public:  
    Complex(  
        double x = double{}  
        , double y = double{})  
    ) : r{x}, i{y} {}  
    double real() const { return r; }  
    double imag() const { return i; }  
};
```

```
class Complex {  
    float r;  
    float i;  
public:  
    Complex(  
        float x = float{}  
        , float y = float{})  
    ) : r{x}, i{y} {}  
    float real() const { return r; }  
    float imag() const { return i; }  
};
```

Class template (cont.)

We can transform `Complex` into a *template*, with the floating-point type a parameter of that template

```
template<typename FP> // or, template<class FP>
class Complex {
    static_assert(std::is_floating_point_v<FP>); // (*)
    FP r;
    FP i;
public:
    Complex(FP x = FP{}, FP y = FP{}) : r{x}, i{y} {}
    FP real() const { return r; }
    FP imag() const { return i; }
};

Complex c;                                // error
Complex<double> d;                      // instantiation of a Complex<double> type
Complex<float> f;                        // different type than d
Complex e{1.,1.};                         // Complex<double> deduced
d + e;                                    // ok
d + f;                                    // possibly error
Complex<int> i;                          // acceptable?
```

* compile-time check + type introspection

Class template (cont.)

- A class template can be seen as a type *generator*
- A class template is parameterized by one or more template parameters (types, but not only)
- A template *specialization* identifies a template name and a set of template arguments corresponding to the template parameters
 - Template arguments can be explicitly specified (e.g. following the template name between angular brackets <>) or deduced based on constructor arguments
- When a class template specialization is used in a context where a type is needed, the compiler *instantiates* a new type
 - Identical instantiations are merged together
- Only member functions that are actually used are instantiated
- There are ways to constrain the arguments used to instantiate a template
 - Notably, C++20 has introduced *concepts*

Function template

```
double norm2(Complex const& c) { ... } // not valid any more  
norm2(f); // error; f is of type Complex<float>
```

```
auto norm2(Complex<float> const& c) {  
    return c.real() * c.real() + c.imag() * c.imag();  
}  
norm2(f); // ok  
norm2(d); // error; d is of type Complex<double>
```

```
auto norm2(Complex<double> const& c) {  
    return c.real() * c.real() + c.imag() * c.imag();  
}  
norm2(d); // ok
```

```
template<typename FP> // or template<class FP>  
auto norm2(Complex<FP> const& c) {  
    return c.real() * c.real() + c.imag() * c.imag();  
}  
auto nf = norm2(f); // nf is of type float  
auto nd = norm2(d); // nd is of type double
```

Function template (cont.)

```
template<typename FP>
auto norm2(Complex<FP> const& c)
{ return c.real() * c.real() + c.imag() * c.imag(); }

template<typename C>
auto norm2(C const& c)
{ return c.real() * c.real() + c.imag() * c.imag(); }

norm2(f); // ok
norm2(d); // ok
std::complex<float> g;
norm2(g); // ok!

namespace std {
    template<class T>
    class complex {
        ...
        public:
            T real() const;
            T imag() const;
    };
}
```

This shows the basic idea behind
generic programming

Function template (cont.)

- A function template can be seen as a function *generator*
- A function template is parameterized by one or more template parameters (types, but not only)
- A template *specialization* identifies a template name and a set of template arguments corresponding to the template parameters
 - Template arguments can be explicitly specified (e.g. following the template name between angular brackets <>) or deduced based on function arguments
- When a function template specialization is used in a context where a function is needed, the compiler *instantiates* a new function
 - Identical instantiations are merged together
- Only functions that are actually used are instantiated
- There are ways to constrain the arguments used to instantiate a template
- Usually there is no separate definition of a function template

Template argument deduction

- In order to instantiate a template every template argument has to be known
- Template arguments can be deduced from function call arguments

```
template<class F> F norm2(Complex<F> const& c) {···}

Complex<float> f;
norm2(f);           // no need to specify norm2<float>(f)
norm2<float>(f);   // ok, be explicit
norm2<double>(f); // instantiate and call norm2<double>, probably failing
```

- Sometimes it's not possible to deduce all template arguments from function parameters

```
template<class Number>
Number lexical_cast(std::string const& s) // convert a string to a number
{···}

auto d = lexical_cast<double>("3.14"); // the destination type is needed
```

Template argument deduction (cont.)

- Constructor calls can be used to deduce *class* template arguments
 - Constructor Template Argument Deduction (CTAD)

```
template<class FP> class Complex { ... };

Complex d;           // error, cannot deduce FP
Complex<double> e; // ok
Complex f{1.};      // ok, Complex<double>
Complex<double> g{1.}; // ok
Complex<float> h{1.}; // ok, 1. -> 1.F
```

- If CTAD is not available (e.g. in older C++ versions), leverage function template argument deduction

```
template<class FP>
auto make_complex(FP r, FP i) {
    return Complex<FP>(r, i);
}

auto c = make_complex(1.,2.); // instead of Complex c{1.,2.}
```

Exercises

- Write an `operator+` that adds two complex numbers of different types
- Transform the `Rational` class into a class template, where the parameter is the type used to represent numerator and denominator. Adapt the operators.

Write a program that instantiates the `Rational` class with integral types such as `int`, `long int`, `long long int`, `short int`. Combine objects of those types using the operators.

Non-type template parameters

- A class/function template parameter need not be a type
- A template parameter can be a value, which must be known at compile time

```
template<int I> struct C { ... };
C<2> c2;
C<3> c3;           // different type
int n; std::cin >> n;
C<n> c4;          // error, n is not a compile-time constant
```

- The type of the value needs to satisfy certain constraints
- Integral types (e.g. int, char, bool) and enumeration types are allowed, other *structural* types (including floating-point types) since C++20

```
template<double D> struct F { ... }; // ok since C++20
```

- Type and non-type template parameters can be mixed

```
template<class T, int N> struct array { ... };
template<class T, T v> struct integral_constant { ... };
```

The C++ Standard Library

Namespaces

- How can we guarantee that in a program composed of thousands of files, written by thousands of people, using third-party libraries, there are no conflicts between identifiers?
- Namespaces are a mechanism to partition the space of names in a program to prevent such conflicts

```
// in <vector>
namespace std {
    template<class T> vector { ... };
}

// in <algorithm>
namespace std {
    template<class It, class T> find(It first, It last, T const& v) { ... }
}
```

- Namespaces can be re-opened, even in other files
 - But namespace std cannot be re-opened by a user, unless explicitly allowed by the specification
- Namespaces can be nested

Namespaces (cont.)

- A **namespace alias** gives another name to an existing namespace

```
namespace ch = std::chrono;  
  
auto t0 = ch::system_clock::now(); // std::chrono::system_clock::now()
```

- A **using declaration** makes a namespace symbol visible in the current scope

```
using std::string;  
  
string s;
```

- A **using directive** makes **all** the namespace symbols visible in the current scope

```
using namespace std;  
  
string s;
```

- Avoid “using directives”. They are especially bad in the global scope and in header files

Namespaces (cont.)

- It's good practice to put all entities of a software component into a namespace

```
namespace stats {  
  
    struct Result { ... };  
    class Regression { ... };  
  
    Result fit(Regression const&);  
  
}
```

- Within a namespace, when using a name declared in that namespace, there is no need to namespace-qualify it (i.e. prepending *namespace::*)

The C++ standard library

- The standard library contains components of general use
 - containers (data structures)
 - algorithms
 - strings
 - input/output
 - mathematical functions
 - random numbers
 - regular expressions
 - concurrency and parallelism
 - filesystem
 - ...
- The subset containing containers and algorithms is known as STL (Standard **Template** Library)
 - But templates are everywhere

Containers of objects

- A program often needs to manage collections of objects
 - e.g. a string of characters, a dictionary of words, a list of particles, a matrix, ...
- A *container* is an object that contains other objects
- The C++ Standard Library provides a few container classes
 - implemented as class templates
 - different characteristics and operations, some common traits

std::vector<T>

Dynamic container of elements of type T

- its size can vary at runtime
- layout is contiguous in memory
- container you should use by default
- be careful with initialization: {} vs ()
 - check if there is a constructor that takes an `std::initializer_list`

```
#include <vector>

std::vector<int> a;          // empty vector of ints
std::vector<int> b{2};        // one element, initialized to 2
std::vector<int> c(2);        // two elements (!), value-initialized (0 for int)
std::vector<int> d{2,1};       // two elements, initialized to 2 and 1
std::vector<int> e(2,1);       // two elements, both initialized to 1

auto f = b; // make a copy, f and b are two distinct objects
f == b;     // true
```

std::vector<T> (cont.)

- The `size` method gives the number of elements in the vector
- The `empty` method tells if the vector is empty
- `operator[]` gives access to the i^{th} element

```
std::vector<int> vec{4,2,7};

assert(!vec.empty());
std::cout << vec.size(); // print 3
vec[1] = 5;             // vec is now {4,5,7}
std::cout << vec[1];    // print 5
```

Counting starts from 0!

```
vec[0];           // first element
vec[1];           // second element
vec[vec.size()-1]; // vec[2], third and last element
vec[vec.size()];  // vec[3], element doesn't exist, undefined behaviour!
```

std::vector<T> (cont.)

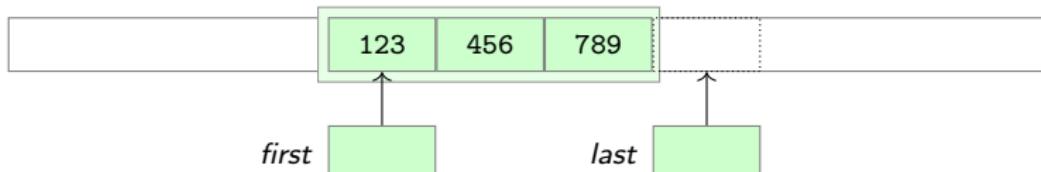
- The `push_back` method adds an element at the end of the vector

```
vec.push_back(-2);           // vec is now {4,5,7,-2}
vec.push_back(0);           // vec is now {4,5,7,-2,0}
std::cout << vec.size(); // print 5
```

```
// fill a vector with numbers read from standard input
std::vector<double> v;
for (double d; std::cin >> d; ) {
    v.push_back(d);
}
```

Iterators and ranges

- An **iterator** is an object that indicates a position within a **range**
 - A container, such as a vector, is a range
- In fact, a pair of iterators $[first, last)$ represents a range
 - the range is *half-open*
 - *first* points to the first element of the range
 - *last* points to **one past** the last element of the range
 - *first == last* means the range is empty



- Ranges are typically obtained from containers calling methods **begin** and **end**

```
std::vector<int> v {...};  
auto first = v.begin(); // std::vector<int>::iterator  
auto last = v.end(); // std::vector<int>::iterator
```

Operations on iterators

- Syntactically, operations on iterators are inspired by pointers
- There is a minimal set of operations supported by an iterator `it`
- `*it` gives access to the element pointed to by `it`

```
std::vector<int> v {1,2,3};  
auto it = v.begin();  
std::cout << *it; // print 1  
*it = 4;           // v is now {4,2,3}
```

```
auto it = v.end();  
  
*it; // undefined behaviour, it doesn't point inside the range
```

Operations on iterators (cont.)

- `it->member` gives access to a member (data or function) of the element pointed to by `it`
 - equivalent to `(*it).member`, note the parenthesis

```
std::vector<Point> v {Point{1,2}, Point{3,4}};  
auto it = v.begin();  
std::cout << (*it).x; // print 1  
std::cout << it->x; // equivalent
```

```
struct Point {  
    double x;  
    double y;  
};
```

```
std::vector<std::string> v {"hello", "world"};  
auto itv = v.begin(); // itv points to the first string in the vector  
auto its = itv->begin(); // its points to the first character  
                        //   of the first string ('h');  
                        //   a string is a container of characters
```

Operations on iterators (cont.)

- `++it` advances it so that it points to the next element in the range

```
std::vector<int> v {123, 456, 789};  
auto first = v.begin();  
auto last = v.end();  
std::cout << *first; // print 123
```



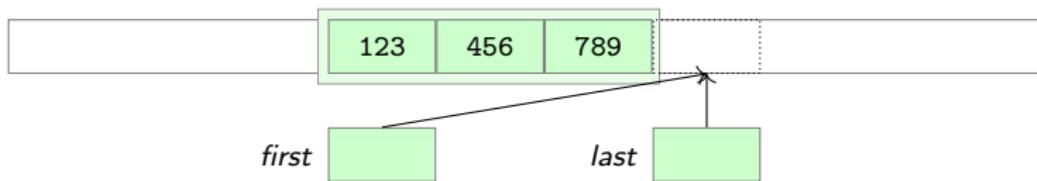
```
++first;  
std::cout << *first; // print 456
```



Operations on iterators (cont.)

- `it1 == it2` (`it1 != it2`) tells if `it1` and `it2` point to the same element (different elements) of the range

```
++first; ++first;  
first == last; // true
```



- Other operations (`--it`, `it + n`, `it += n`, `it < it2`, ...) may be supported, depending on the underlying range
 - vector iterators support them all

Example: add all elements of a vector (Iteration on a container)

```
std::vector<int> c { ... };
```

```
auto sum = 0;
for (int i {0}, n = c.size(); i != n; ++i) {
    auto const& v = c[i];
    sum += v;
}
```

```
auto sum = 0;
for (auto it = c.begin(), last = c.end(); it != last; ++it) {
    auto const& v = *it;
    sum += v;
}
```

```
auto sum = 0;
for (auto const& v : c) { // translates to the loop with iterators
    sum += v;
}
```

```
auto sum = std::accumulate(c.begin(), c.end(), 0); // algorithm
```

std::vector<T> (cont.)

- The `erase` method removes the element pointed to by the iterator passed as argument (must not be `end()`)

```
// remove the central element
auto it = v.begin() + v.size() / 2; // iterator to the middle element
v.erase(it);                      // size decreased by 1
```

- The `erase` method can remove a range, passing two iterators

```
// erase the 2nd half of the vector
auto it = v.begin() + v.size() / 2; // iterator to the middle element
v.erase(it, v.end());            // size ~halved here
```

- In general, iterators pointing to erased elements are not valid anymore
 - For a vector, iterators pointing to elements following the erased one are also invalidated, including the `end` iterator

```
v.erase(it);
*it;      // undefined behaviour (UB)
++it;    // UB, be careful in loops
it = ...; // ok
```

std::vector<T> (cont.)

- The `insert` method inserts an element before the position indicated by an iterator

```
// insert the value 42 in the middle of the vector
auto it = v.begin() + v.size() / 2;
v.insert(it, 42);                      // size increased by 1
```

- Other overloads of `insert` are available, see reference
- Existing iterators pointing to elements after the insertion position are invalidated
 - If a memory reallocation occurs (see later), all existing iterators pointing into the vector are invalidated

std::array<T, N>

Container of N elements of type T

- N known at compile time
- layout is contiguous in memory

```
#include <array>

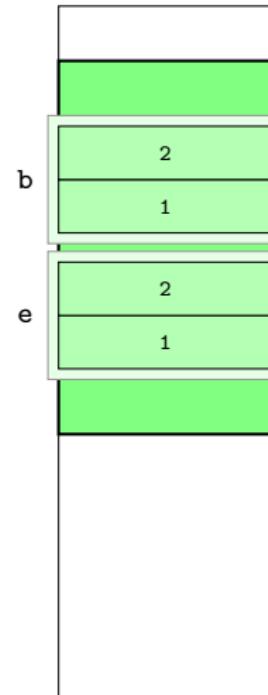
// 2 ints, uninitialized
std::array<int,2> a;

// 2 ints, initialized to 1 and 2
std::array<int,2> b{1,2};

// 2 ints, value-initialized (0 for int)
std::array<int,2> c{};

// 2 ints, initialized to 1 and 0
std::array<int,2> d{1};

// make a copy
auto e = b; // std::array<int,2>
assert(e == b);
```



std::array<T, N> (cont.)

- The `size` method gives the number of elements in the array (which corresponds to `N`)
- `operator[]` gives access to the i^{th} element

```
int const M {4};  
std::array<int, M> arr{1,2}; // {1,2,0,0}  
  
std::cout << arr.size(); // print 4  
arr[1] = 3; // arr is now {1,3,0,0}  
std::cout << arr[1]; // print 3
```

- `begin`, `end`, `empty`, `front`, `back` methods
- No `push_back` or `insert`, the size is fixed.

Type aliases (cont.)

- Type aliases can be templates

```
// array of 3 T's
template<class T> using Array3 = std::array<T, 3>;
Array3<double> a; // std::array<double, 3>
```

```
// array of N bytes
template<int N> using ArrayOfBytes = std::array<std::byte, N>;
ArrayOfBytes<16> b; // std::array<std::byte, 16>
```

Algorithms

- Generic functions that operate on **ranges** of objects
- Implemented as function templates
- Sum all the elements of a container `cont` of `ints`

```
int sum {0};  
for (auto it = cont.begin(), last = cont.end(); it != last; ++it) {  
    sum += *it;  
}
```

```
auto sum = std::accumulate(cont.begin(), cont.end(), 0); // better
```

- Find an element equal to `val` in a container `cont`

```
auto it = cont.begin();  
auto const last = cont.end();  
for (; it != last; ++it) {  
    if (*it == val) {  
        break;  
    }  
}
```

```
auto it = std::find(cont.begin(), cont.end(), val); // better
```

Generic programming

- A style of programming in which **algorithms** are written in terms of **concepts**

```
template<class InputIterator, class Tp>
Tp accumulate(InputIterator first, InputIterator last, Tp init)
{
    for (; first != last; ++first)
        init = init + *first;
    return init;
}
```

```
template <class Iterator, class T>
Iterator find(Iterator first, Iterator last, const T& value)
{
    for (; first != last; ++first)
        if (*first == value) break;
    return first;
}
```

Concepts

- A *concept* is a set of requirements that a type needs to satisfy at compile time
 - e.g. supported expressions, nested typedefs, memory layout, ...
- Concepts constrain the types that can be used as template arguments
 - Implicit until C++20, based on how those types are used in a class/function template definition
 - C++20 introduces a specific syntax and a set of generally useful concepts (not further discussed)

```
template<class T>
concept Incrementable = requires(T t) { ++t; };

template<Incrementable T>
auto advance(T& t) { ++t; }

int i {42};
advance(i); // ok, int is a model of Incrementable

struct S {};
S s;
advance(s); // error, S is not a model of Incrementable
```

Hierarchy of iterators

```
// InputIterator (find)  
while (first != last && !(first == value)) ++first;
```

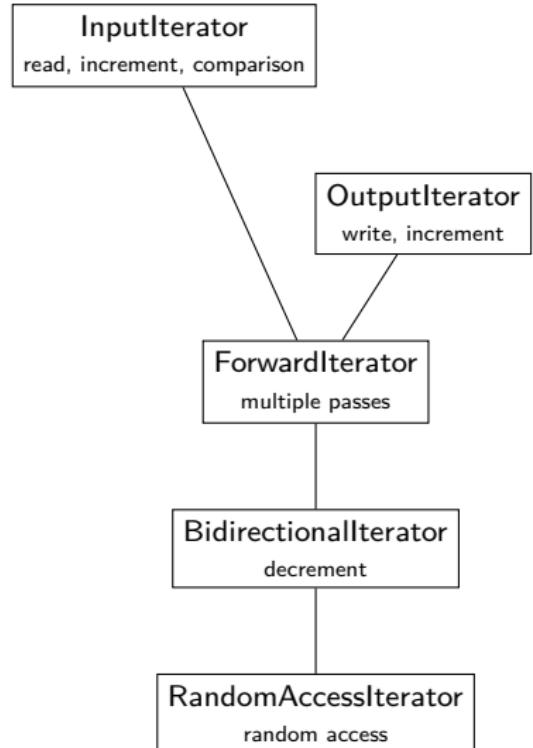
```
// OutputIterator (generate_n)  
for (; n > 0; ++first, --n) *first = gen();
```

```
// ForwardIterator (generate)  
for (; first != last; ++first) *first = gen();
```

```
// ForwardIterator (adjacent_find)  
auto i = first;  
while (++i != last) ...
```

```
// BidirectionalIterator (reverse)  
if (first == --last) break;
```

```
// RandomAccessIterator (reverse)  
for (; first < --last; ++first) ...
```



Examples of algorithms

Non-modifying `all_of` `any_of` `for_each` `count` `count_if`
`mismatch` `equal` `find` `find_if` `adjacent_find`
`search` ...

Modifying `copy` `copy_if` `fill` `generate` `transform`
`remove` `replace` `swap` `reverse` `rotate` `shuffle`
`sample` `unique` ...

Partitioning `partition` `stable_partition` ...

Sorting `sort` `partial_sort` `nth_element` ...

Set `includes` `set_union` `set_intersection` ...

Min/Max `min` `max` `minmax` `clamp` ...

Comparison `equal` `lexicographical_compare` ...

Numeric `iota` `accumulate` `inner_product` `partial_sum`
`adjacent_difference` `reduce` ...

...

Algorithms in action

```
std::array a {23, 54, 41, 0, 18};

// sort the array in ascending order
std::sort(std::begin(a), std::end(a));

// sum up the array elements, initializing the sum to 0
auto s = std::accumulate(std::begin(a), std::end(a), 0);

// append the partial sums of the array elements into a vector
std::vector<int> v;
std::partial_sum(std::begin(a), std::end(a), std::back_inserter(v));

auto p = std::inner_product(std::begin(a), std::end(a), std::begin(v), 0);

// find the first element with value 42, if existing
auto it = std::find(std::begin(v), std::end(v), 42);
```

Why using standard algorithms

- They are correct
- They express intent more clearly than a raw `for/while` loop
- They are efficient
 - They give computational complexity guarantees
 - How fast do they run? how much additional memory do they need?
- They enable *parallelism for the masses*

```
#include <execution>

std::vector<int> v { ... };

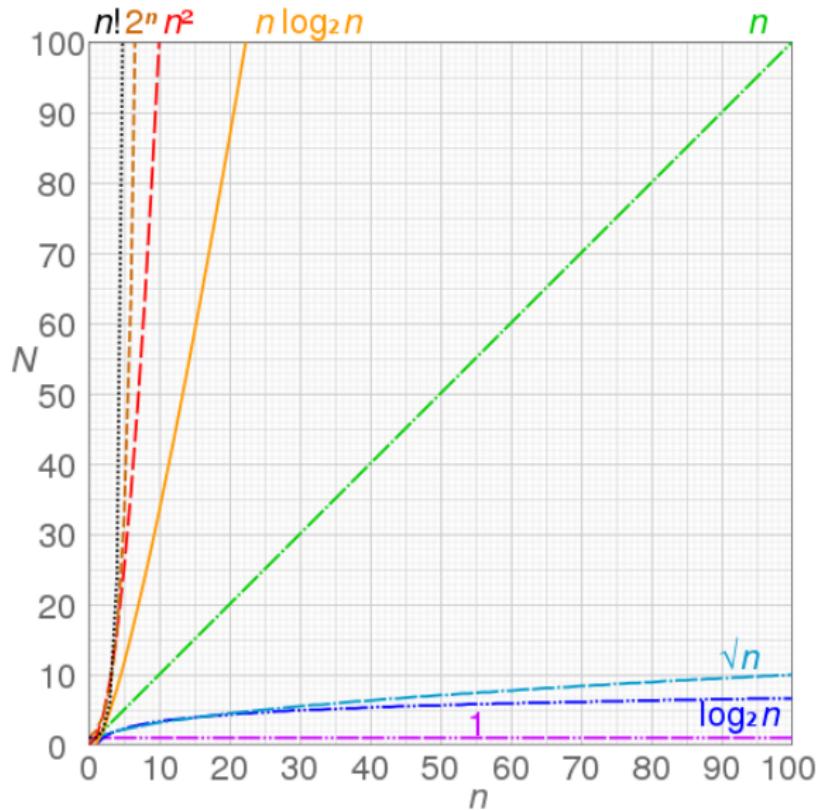
std::sort(std::execution::par, v.begin(), v.end());

auto it = std::find(std::execution::par, v.begin(), v.end(), 123);
```

Computational complexity

- A measure of how many resources a computation will need for a given input size
 - Typically the resource is time but can be space (memory)
 - For example: how many comparisons does the sort algorithm do for a range of one million elements?
- Of typical interest are the average case and the worst case
- The complexity is a function f of the input size n , but usually only the asymptotic behaviour is given
 - Big-O notation
 - $\mathcal{O}(g(n))$ means that, for a large n , $f(n) \leq cg(n)$, for some constant c
 - Note how constant factors don't matter in big-O notation
- For example
 - `std::vector<T>::push_back` is (amortized) $\mathcal{O}(1)$
 - `std::binary_search` is $\mathcal{O}(\log n)$
 - `std::find` is $\mathcal{O}(n)$
 - `std::sort` is $\mathcal{O}(n \log n)$

Computational complexity (cont.)



Functions

- A function associates a sequence of statements (the function *body*) with a name and a list of zero or more parameters
- A function may return a value
- Multiple functions can have the same name → *overloading*
 - different parameter lists
- A function returning a bool is called a *predicate*

```
bool less(int n, int m) { return n < m; }
```

Algorithms and functions

```
template <class Iterator, class T>
Iterator find(Iterator first, Iterator last, const T& value)
{
    for (; first != last; ++first)
        if (*first == value)
            break;
    return first;
}

auto it = find(v.begin(), v.end(), 42);
```

```
template <class Iterator, class Predicate>
Iterator find_if(Iterator first, Iterator last, Predicate pred)
{
    for (; first != last; ++first)
        if (pred(*first))           // unary predicate
            break;
    return first;
}

bool lt42(int n) { return n < 42; }

auto it = find_if(v.begin(), v.end(), lt42);
auto it = find_if(v.begin(), v.end(), [](int n) { return n < 42; } );
```

Some algorithms are customizable passing a function

Function objects

A mechanism to define *something-callable-like-a-function*

- A class with an operator()

```
auto lt42(int n)
{
    return n < 42;
}

auto b = lt42(32); // true

std::vector v {61,32,51};
auto it = std::find_if(
    v.begin(), v.end(),
    lt42
); // *it == 32
```

```
struct LessThan42 {
    auto operator()(int n) const
    {
        return n < 42;
    }
};

LessThan42 lt42{};
// or: auto lt42 = LessThan42{};
auto b = lt42(32); // true

std::vector v {61,32,51};
auto it = std::find_if(
    v.begin(), v.end(),
    lt42 // or directly: LessThan42{}
); // *it == 32
```

Function objects (cont.)

A function object, being the instance of a class, can have state

```
class LessThan {
    int m_;
public:
    explicit LessThan(int m) : m_{m} {}
    auto operator()(int n) const {
        return n < m_;
    }
};

LessThan lt42 {42};
auto b1 = lt42(32); // true
// or: auto b1 = LessThan{42}(32);
LessThan lt24 {24};
auto b2 = lt24(32); // false
// or: auto b2 = LessThan{24}(32);

std::vector v {61,32,51};
auto i1 = std::find_if(..., lt42); // *i1 == 32
// or: auto i1 = std::find_if(..., LessThan{42});
auto i2 = std::find_if(..., lt24); // i2 == v.end(), i.e. not found
// or: auto i2 = std::find_if(..., LessThan{24});
```

Function objects (cont.)

An example from the standard library

```
#include <random>

// random bit generator
std::default_random_engine eng;

// generate N 32-bit unsigned integer numbers
for (int n = 0; n != N; ++n) {
    std::cout << eng() << '\n';
}

// generate N floats distributed normally (mean: 0., stddev: 1.)
std::normal_distribution<float> dist;
for (int n = 0; n != N; ++n) {
    std::cout << dist(eng) << '\n';
}

// generate N ints distributed uniformly between 1 and 6 included
std::uniform_int_distribution<> roll_dice(1, 6);
for (int n = 0; n != N; ++n) {
    std::cout << roll_dice(eng) << '\n';
}
```

Lambda expression

- A concise way to create an unnamed function object
- Useful to pass actions/callbacks to algorithms, threads, frameworks, ...

```
struct LessThan42 {  
    auto operator()(int n)  
    {  
        return n < 42;  
    }  
};  
  
class LessThan {  
    int m_;  
public:  
    explicit LessThan(int m)  
        : m_{m} {}  
    auto operator()(int n) const  
    {  
        return n < m_;  
    }  
};
```

```
std::find_if(..., LessThan42{});  
  
std::find_if(..., [](int n) {  
    return n < 42;  
});  
  
std::find_if(..., LessThan{m});  
  
auto m = ...;  
std::find_if(..., [=](int n) {  
    return n < m;  
});  
std::find_if(..., [m = ...](int n) {  
    return n < m;  
});
```

Lambda closure

The evaluation of a lambda expression produces an unnamed function object (a *closure*)

- The operator() corresponds to the code of the body of the lambda expression
- The data members are the captured local variables

```
auto v = 42;  
  
auto lt = [v](int n)  
          { return n < v; }  
  
auto r = lt(5); // true
```

```
class SomeUniqueName {  
    int v;  
public:  
    explicit SomeUniqueName(int v)  
        : v{v} {}  
    auto operator()(int n) const  
    { return n < v; }  
};  
  
auto v = 42;  
auto lt = SomeUniqueName{v};  
auto r = lt(5); // true
```

- Two lambda expressions produce objects of different types, even if they are identical

Lambda capture

- Automatic variables used in the body of the lambda need to be captured
 - [] capture nothing
 - [=] capture all (what is needed) by value
 - [k] capture k by value
 - [&] capture all (what is needed) by reference
 - [&k] capture k by reference
 - [=, &k] capture all by value but k by reference
 - [&, k] capture all by reference but k by value

```
auto v = 3;  
auto l = [&v] {};
```

```
class SomeUniqueName {  
    int& v;  
public:  
    explicit SomeUniqueName(int& v)  
        : v{v} {}  
    ...  
};  
  
auto l = SomeUniqueName{v};
```

- Global variables are available without being captured

Lambda explicit return type

- The return type of the call operator can be explicitly specified

```
[=](int n) -> bool { return n < v; }
```

becomes

```
class SomeUniqueName {
    ...
    bool operator()(int n) const
    { return n < v; }
};
```

Generic lambda

- If a parameter of the lambda expression is `auto`, the lambda expression is *generic*
- The call operator is a template

```
[](auto n) { ... }
```

becomes

```
class SomeUniqueName {
...
template<typename T>
auto operator()(T n) const { ... }
};
```

Lambda: const and mutable

- By default the call to a lambda is `const`
 - Variables captured by value are not modifiable
- A lambda can be declared `mutable`
 - The parameter list is mandatory
- If present, the explicit return type goes after `mutable`

```
[i] () mutable -> bool {... ++i ...}
```

```
class SomeUniqueName {  
    int i;  
    ...  
    bool operator()() {... ++i ...}  
};
```

- Variables captured by reference can be modified
 - There is no way to capture by `const&`

```
int v = 3;  
[&v] { ++v; } (); // NB the lambda is immediately invoked  
assert(v == 4);
```

Lambda: dangling reference

- Be careful not to have dangling references in a closure
- It's similar to a function returning a reference to a local variable

```
auto make_lambda() // auto here is unavoidable
{
    int v = 3;
    return [&] { return v; }; // return a closure
}

auto l = make_lambda();
auto d = l(); // the captured variable is dangling here
```

- Capture by reference only if the lambda closure doesn't survive the current scope

Exercises

- Given a vector of ints, compute the product of the numbers
(Hint: use `std::accumulate`)
- Generate N numbers, using one of the **available random number distributions**, and insert them in a `std::vector` (Hint: use `std::generate_n`)
- Compute the mean and the standard deviation of the generated numbers (Hint: use `std::accumulate`)

Don't forget the tests!

std::function

- Type-erased wrapper that can store and invoke any callable entity with a certain signature
 - function, function object, lambda, member function

```
#include <functional>

using Function = std::function<int(int,int)>; // signature

Function f1 { std::plus<int>{} };
Function f2 { [](int a, int b) { return a * b; } };
Function f3 { [ ](auto a, auto b) { return std::gcd(a,b); } };
```

- Some space and time overhead, so use only if a template parameter is not satisfactory

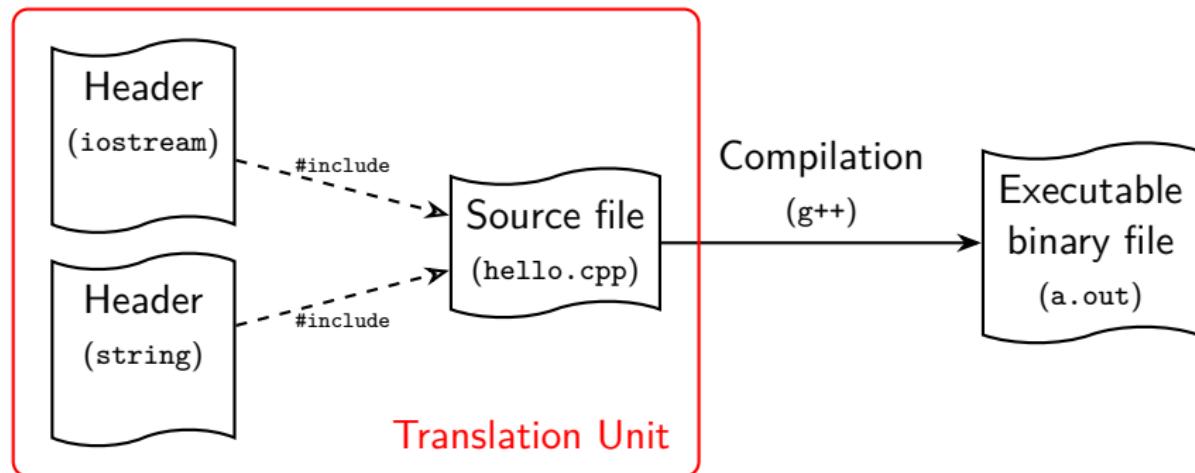
```
std::vector<Function> functions { f1, f2, f3 };

for (auto& f : functions) {
    std::cout << f(121, 42) << '\n'; // 163 5082 1
}
```

Compilation model

The C++ compilation model

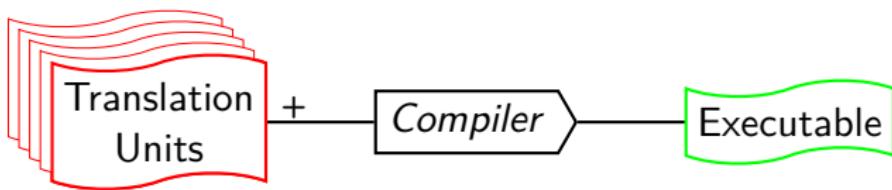
This is the **current** model; C++20 has introduced *modules* (not further discussed)



- The production of the *translation unit* is the first step of the overall *compilation* process
- It is accomplished by running the *preprocessor* on a source file
- To stop the compilation process after the preprocessing:
 - `g++ -E hello.cpp`

The C++ compilation model (cont.)

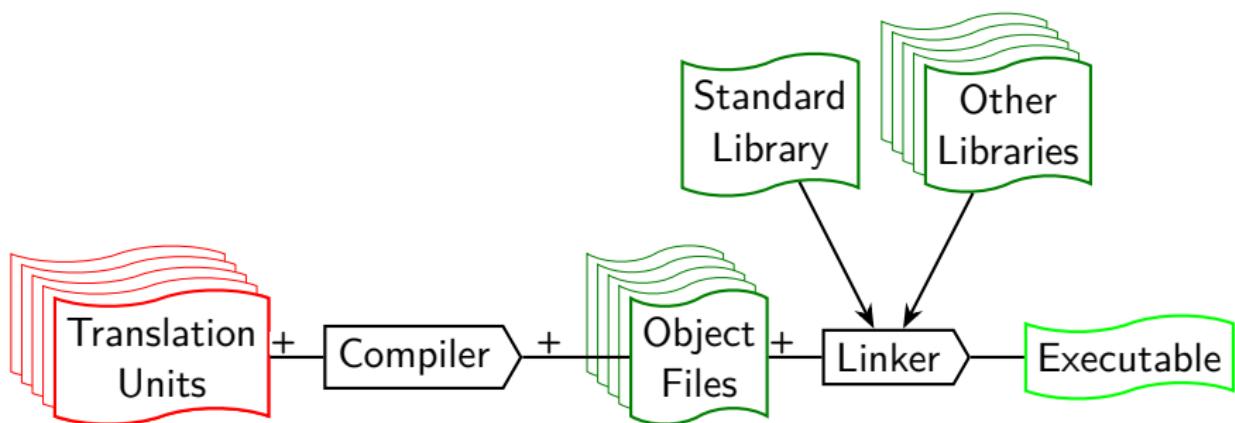
- A typical C++ program is composed of many files, i.e. there are many translation units



- How to distribute source code over multiple files?
- How to expose functionality from one translation unit so that it can be used in another?
- **Physical** design

The C++ compilation model (cont.)

- The overall process to translate C++ source code to the final binary executable consists of multiple stages



- To produce object files without linking:
 - `g++ -c hello.cpp`
- To keep all the intermediate files generated during the compilation process:
 - `g++ -f-savetemps hello.cpp`

Definition vs Declaration

- A **definition** is a declaration that fully defines an entity
- Examples:

```
bool less_than(int a, int b) { // function definition
    return a < b;
}

class Sample { // class definition
public:
    void add(double x) // member function definition
    {
        ++N_;
        sum_x_ += x;
        ...
    }
    ...
};
```

Definition vs Declaration (cont.)

- A **declaration** that is **not** a definition just introduces the name (and, when applicable, the type) of the entity
- Examples:

```
bool less_than(int a, int b); // function declaration (prototype)

class Sample; // class (forward) declaration

class Sample { // class definition
public:
    void add(double x); // member function declaration
    ...
};
```

One-Definition Rule (ODR)

- An entity can be defined only once in each translation unit
- More generally, an entity can be defined only once in the whole program, with exceptions
- Some entities can be defined in multiple translation units, provided the definitions are identical (*token-by-token* in the source code)
 - Class definitions
 - *inline* functions and variables
 - Class and function templates
 - ...
- To guarantee that definitions are identical in all translation units they typically appear in **header files**, which are then #included where needed
- The compiler may not be able to diagnose violations of the ODR
 - The generated executable may behave incorrectly

Definition vs Declaration during compilation and linking

- During the compilation step:
 - Calling a function requires the previous declaration of that function
 - The declaration of a function requires the previous declaration of the types involved
 - Creating and manipulating an object require the definition of the corresponding type (the type has to be *complete*)
- During the linking step:
 - Everything has to be properly defined

Header and source files

- Header files are the primary mechanism to guarantee that declarations and definitions are identical in all translation units
 - Because they are `#included` in other header and source files where those declarations and definitions are needed
- For a given software component, a typical situation foresees
 - One **header** file containing free function declarations, class definitions with member function declarations, template definitions
 - One **source** file containing definitions of free and member functions and of any other entity needed for the implementation
 - **Suggestion** the source file `#includes` the header file as the first `#include`
 - One file with the unit **tests**
- A function defined in a header file needs to be **inline**
 - NB the **inline** keyword is not (any more) an optimization hint to the compiler
- Member functions (methods) defined inside the class and function templates are implicitly **inline**

Member functions defined outside the class

- A method must be declared inside the class, but can be **just** declared inside the class, i.e. not also defined

```
class Sample {  
    ...  
    void add(double);  
};
```

- When defining a method outside the class, the method name must be prefixed with the class name followed by the *scope* operator `::`, i.e. *class-name::method-name*

```
void Sample::add(double x) {  
    ++N_;  
    sum_x_ += x;  
    ...  
}
```

Member functions defined outside the class (cont.)

- The method can be defined in the same file (normally a header file) or even in another file (normally the corresponding source file)
- If the definition is in a header file, the method should be declared **inline**

```
inline void Sample::add(double d) {  
    ++N_;  
    sum_x_ += x;  
    ...  
}
```

This informs the compiler/linker that all the definitions of the function, even if they are in multiple translation units, are equal

Include guards

- A header (file) may be included, directly or indirectly, multiple times in the **same** translation unit

```
#include "result.hpp"
#include "sample.hpp" // sample.hpp already includes result.hpp
```

- Multiple definitions of the same entity would be available in the **same** translation unit, which is illegal
- Placing an *include guard* at the beginning of each header file prevents multiple inclusions in the **same** translation unit

```
// file result.hpp
#ifndef RESULT_HPP
#define RESULT_HPP
...
// classes, functions, templates, ...
...
#endif
```

To inline or not to inline?

- Pros:
 - A compiler can optimize more because it sees more code → faster execution
 - Definitions of inline functions contained in included header files become part of the translation unit
 - Header-only libraries (i.e. everything is in one or more header files) are easier to distribute (e.g. doctest)
- Cons:
 - More physical coupling between software components
 - Visibility of implementation details
 - Every time a header file changes all the including files need to be recompiled → longer compilation times
 - The code of the inlined functions may be included many times in the final binary, making it larger and causing inefficient use of the memory system → slower execution

Build systems

- When a project includes more than a few files, keeping track of compilation dependencies between them becomes difficult
- A project may foresee multiple *artifacts*: executables, libraries (i.e. collection of object files), tests, ...
- A project may depend on other projects, e.g. other libraries
- Usually multiple variants of a project are needed: one for debug, one for testing with code coverage, one for release, one for measuring performance (*profiling*), ...
 - Each variant foresees different compilation flags
- ...
- In practice, all the above cannot be done manually and an automatic system is required
- Such a system is usually called **build system**
- Many build systems exist. We'll use **CMake**

CMake

- CMake is an open-source, cross-platform family of tools designed to build, test and package software
- Most used tool to build C++ projects
- Build targets, dependencies, options, ... are expressed declaratively in a file called CMakeLists.txt

```
cmake_minimum_required(VERSION 3.16)
project(mandelbrot VERSION 0.1.0)

find_package(SFML 2.5 COMPONENTS graphics REQUIRED)

string(APPEND CMAKE_CXX_FLAGS " -Wall -Wextra")

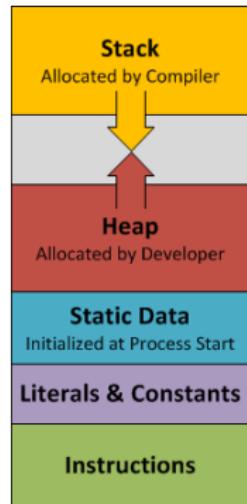
add_executable(mandelbrot main.cpp)
target_link_libraries(mandelbrot PRIVATE sfml-graphics)
```

- See **mandelbrot** for a basic example

Explicit Memory Management (Resource Management)

Memory layout of a process

- A process is a running program
- When a program is started the operating system brings the contents of the corresponding file into memory according to well-defined conventions
 - Stack
 - function local variables
 - function call bookkeeping
 - Heap
 - dynamic allocation
 - Global data
 - literals and variables
 - initialized and uninitialized (set to 0)
 - Program instructions



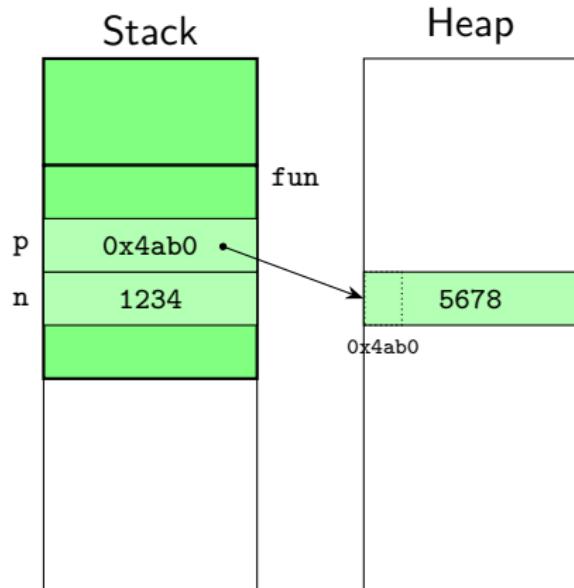
Dynamic memory allocation

- It's not always possible or convenient to construct objects on the stack, where they would be destroyed at the end of the function that created them
- An object (array of objects) can be constructed on the *free store (heap)*
 - operator `new` (for an array: `new []`)
 - allocate memory for the object(s)
 - run the object(s) constructor
- The lifetime of an object (array of objects) on the heap is **explicitly** managed by the developer
 - explicit destruction of the object(s) when not needed any more
 - operator `delete` (for an array: `delete []`)
 - run the object(s) *destructor* (see later)
 - deallocate the memory previously allocated for the object(s)
- Contrast this to the **automatic** destruction of an object at the end of the scope when the object is created on the stack

Dynamic allocation of an object

```
auto fun()
{
    int n {1234};
    int* p = new int{5678};
    ...
    delete p;
}
```

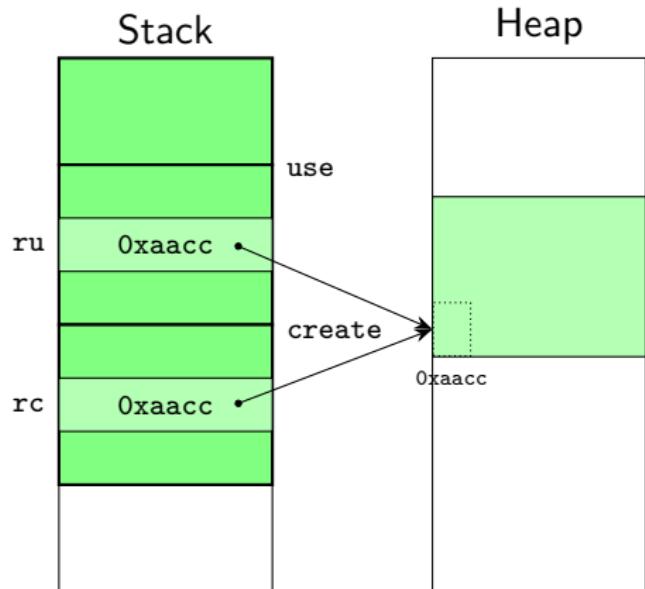
- `delete p` gives the area on the heap back to the system
- `delete p` does **not** modify `p`
- After `delete p`, the only safe operation on `p` is an assignment
 - `p = ...;`
- if `p` is `nullptr`, `delete p` is well defined and does nothing



Returning a dynamically-allocated object from a function

```
Sample* create()
{
    auto rc = new Sample{};
    rc->add(...);
    ...
    return rc;
}

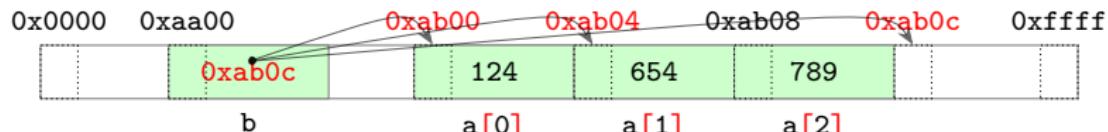
auto use()
{
    auto ru = create();
    ru->stats();
    ...
    delete ru;
}
```



Note how the lifetime of a dynamically-allocated object is explicitly managed by the developer

(native) Array of objects

Contiguous sequence of homogeneous objects in memory

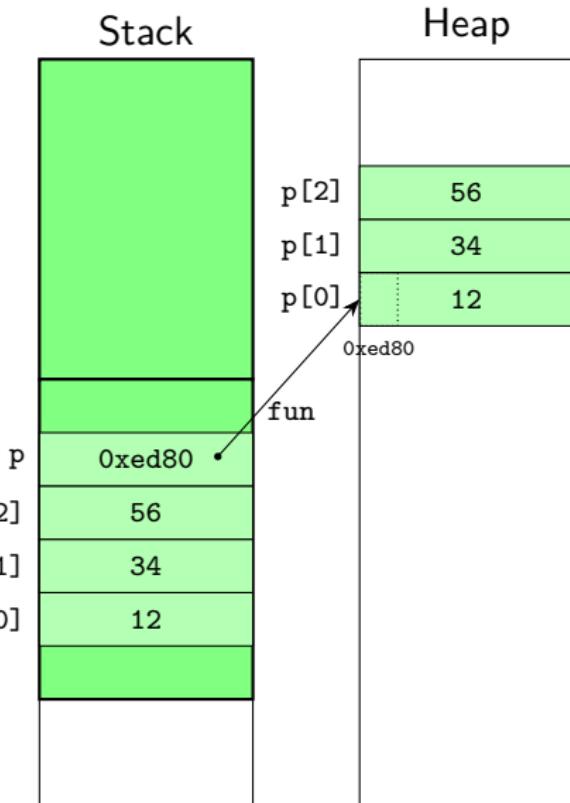


```
int a[3] = {123, 456, 789}; // int[3], the size must be a constant
                             // and can be deduced from the initializer
++a[0];
a[3];                      // undefined behavior
// "arrays decay to pointers at the slightest provocation"
auto b = a;                 // int*, size information lost
assert(b == &a[0]);
++b;                        // increase by sizeof(int)
assert(b == &a[1]);
*b = 654;
b += 2;                     // increase by 2 * sizeof(int)
*b;                          // undefined behavior
if (b == a + 3) { ... }    // ok, but not more than this
```

Dynamic allocation of an array of objects

```
auto fun()
{
    int a[3] {12, 34, 56};
    int* p = new int[3] {12, 34, 56};
    ...
    delete [] p;
}
```

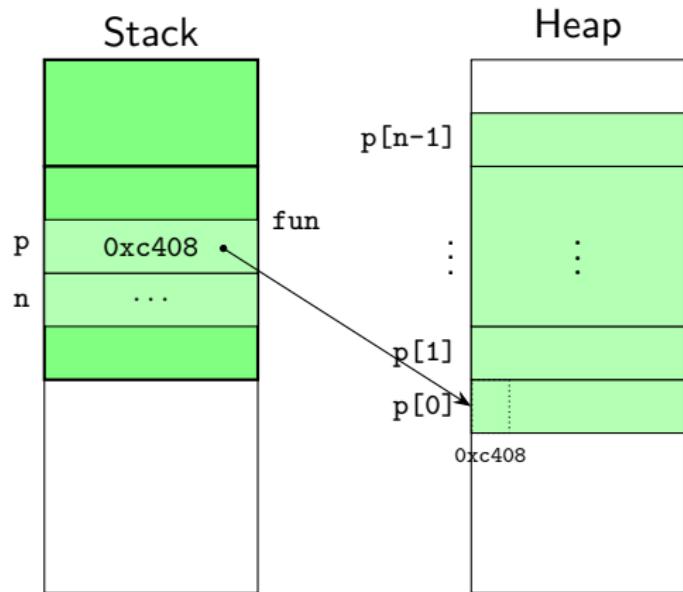
- p points to the first element of the array
- p does **not** carry “array” information



Dynamic allocation of an array of objects of dynamic size

Arrays allocated on the heap are most useful when their size is available only at runtime

```
auto fun()
{
    int n;
    std::cin >> n;
    auto p = new int[n];
    ...
    delete [] p;
}
```



Null-terminated byte strings

- A null-terminated byte string (NTBS) is an array of non-null characters followed by the null character (`char{0}` or `'\0'`)
 - `char[]` (`char*`) if characters are modifiable
 - `char const[]` (`char const*`) if characters are not modifiable
 - Also known as “C-strings”
- The `std::strlen` function returns the length of an NTBS
 - Linear complexity, scan the array until `'\0'` is found
- A string literal has type `char const[N]`, with `N` constant
 - For example, the type of `"ciao"` is `char const[5]`
- An `std::string` can be initialized with an NTBS
- To get the NTBS representation of an `std::string` use the `c_str` method
 - It returns a `char const*` to the internal array

The main function

- The `main` (special) function is the entry point of a program
- It can have two forms
 - `int main() {...}`
 - `int main(int argc, char* argv[]) {...}`
- If there is no `return` statement, an implicit `return 0;` is assumed
 - 0 means success, different from 0 means failure
- `argc` is the number of arguments on the command line, `argv` is an array of C-strings representing the arguments
 - `argv[0]` is (usually) the name of the program
 - `argv[argc]` is `nullptr`
- Many libraries exist to parse the command line, e.g. [Lyra](#), [cxxopts](#), [Boost.Program_Options](#)

Managing raw arrays

- Managing raw arrays **correctly** is difficult
- Passing an array around makes it decay to a pointer to its first element, loosing important type information
- How do you pass an array to a function?
 - Need to pass pointer and size
 - C++20 introduces `std::span`
- How do you return an array from a function?
 - Left as an exercise
- Better use higher-level tools, such as `std::string`, `std::array` and `std::vector`

Weaknesses of a T*

- Critical information is not encoded in the type
 - Am I the owner of the pointee? Should I delete it?

```
Shape* s = create_shape();                      // (probably) delete

std::string str{"hello"};
char const* cs = str.c_str();                    // don't delete
char      * cd = str.data();                     // don't delete, const is irrelevant
```

- Is the pointee an object or an array of objects? of what size?

```
Shape* s = create_shapes();                     // (probably) delete []
```

- Was it allocated with new, malloc or even something else?
 - malloc is a C function that allocates raw memory, which is then de-allocated passing the pointer to the function free
 - e.g. the fopen function returns a FILE*, which will be released passing it to the fclose function

Weaknesses of a T* (cont.)

- Owning pointers are prone to leaks

```
{  
    auto p = new Circle{...};  
    ...  
    // ops, forgot to delete p  
}
```

- Owning pointers are prone to double deletes

```
{  
    auto p = new Circle{...};  
    ...  
    delete p;  
    ...  
    delete p; // ops, delete again  
}
```

Weaknesses of a T* (cont.)

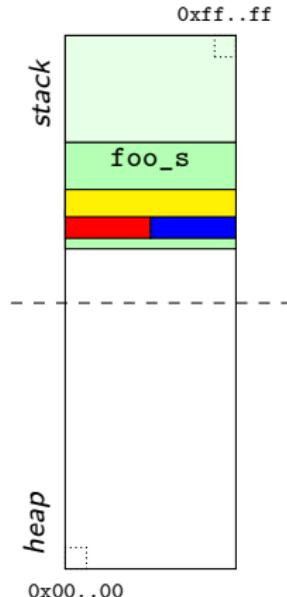
- Owning pointers are unsafe in presence of exceptions

```
{  
    auto p = new Circle{...};  
    ...                                // potentially throwing code  
    delete p;                          // leak if exception thrown  
}
```

- Runtime overhead, in space and time
 - dynamic allocation/deallocation
 - indirection

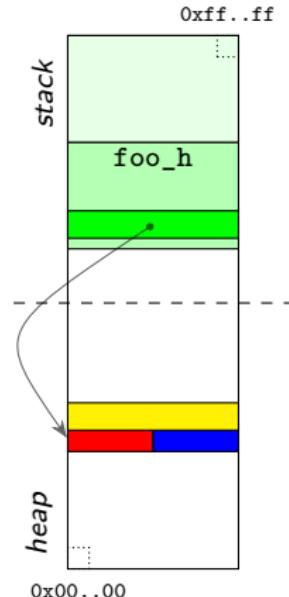
Stack vs Heap: space

```
struct S {  
    int     n;  
    float   f;  
    double  d;  
};  
  
auto foo_s() {  
    S s;  
    ...  
}  
  
auto foo_h() {  
    S* s = new S;  
    ...  
}
```



Occupancy:

- `sizeof(S)`



Occupancy:

- `sizeof(S) + sizeof(S*)`
- plus new internal space overhead

Stack vs Heap: time

Stack

```
void stack()
{
    int m{123};
    ...
}
```

Heap

```
void heap()
{
    int* m = new int{123};
    ...
    delete m;
}
```

```
stack():
    subq %4, %rsp
    movl $123, (%rsp)
    ...
    addq $4, %rsp
    ret
```

```
heap():
    subq $8, %rsp
    movl $4, %edi
    call operator new(unsigned long)
    movl $123, (%rax)
    movq %rax, (%rsp)
    ...
    movl $4, %esi
    movq %rax, %rdi
    call operator delete(void*, unsigned long)
    addq $8, %rsp
    ret
```

```
$ g++ -O3 heap.cpp && ./a.out
1000000 iterations: 0.035745 s
```

i.e. ~35 ns for each new/delete (on my laptop)

Investigating memory problems

- One of the simplest tools to use is the *Address Sanitizer* (ASan)
- The compiler instruments the executable so that at runtime ASan can catch many (but not all!) memory problems
- Some space and time overhead at runtime, but acceptable at least during testing

```
$ g++ -fsanitize=address leak.cpp
$ ./a.out

=====
==18338==ERROR: LeakSanitizer: detected memory leaks
...
...
```

Guidelines on dynamic objects

- Do not unnecessarily allocate on the heap, i.e. use the stack when possible
- If unavoidable, think thoroughly about ownership and use a resource-managing object, e.g.
 - containers and strings
 - *smart pointers* (see later)
- Ideally the residual meaning of a raw pointer (T^*) should be “I’m pointing to **one** element and I’m **not** its owner”

Resource management

- Dynamic memory is just one of the many types of resources manipulated by a program:
 - thread, mutex, socket, file, ...
- C++ offers powerful tools to manage resources
 - "*C++ is my favorite garbage collected language because it generates so little garbage*"

Destruction

- At the end of a block statement (i.e. when the closing brace } is encountered) all objects of automatic storage duration are automatically *destroyed*
- Destroying an object means
 - for a class type, a special member function, called *destructor*, if present, is run
 - all the sub-objects (e.g. data members) are destroyed, recursively
 - the storage is freed
- The order of destruction is the opposite of the order of construction/definition
 - a later-defined object can use a previously-defined one

```
{  
    S s{...};  
    ...  
    T t{s};  
    ...  
} // t is destroyed first, then s
```

The destructor

- The *destructor* is a special class member function that is called when an object of that class goes out of scope and has to be destroyed
- The purpose of the destructor is to leave no garbage behind
- A class can have only one destructor, declared as `classname ()`

```
class DynamicArray {  
    ...  
    int* m_data;  
public:  
    DynamicArray(int n): m_data{new int[n]} {...}  
    ~DynamicArray()  
    {  
        delete [] m_data;  
    }  
    ...  
};
```

RAlI idiom

- Resource Acquisition Is Initialization
 - The constructor accepts/acquires a resource
 - The destructor releases it
- The object is responsible for the correct lifetime management of that resource
- Guaranteed no leak nor double release, even in presence of exceptions

Controlling copying

- The compiler automatically generates the copy operations for a class, if used
 - The copy constructor creates a **new** object as a copy of another object

```
DynamicArray original{...};  
auto copy{original}; // auto copy = original
```

- The copy assignment operator changes the value of an **existing** object as a copy of another object

```
DynamicArray v1{...};  
DynamicArray v2{...};  
v2 = v1;
```

- Ideally, after a copy the two objects should compare equal
- The generated operations may not be correct, especially in presence of managed resources (e.g. dynamically-allocated memory)
- The copy operations can be explicitly provided or suppressed

Controlling copying (cont.)

- The copy constructor typically takes an object of the same class by const reference

```
DynamicArray(DynamicArray const& other) : ... { ... }
```

- The copy assignment operator typically:
 - takes an object of the same class by const reference
 - returns the object itself by reference
 - checks for auto-assignment

```
DynamicArray& operator=(DynamicArray const& other)
{
    if (this != &other) {
        ...
    }
    return *this;
}
```

Controlling copying (cont.)

```
class DynamicArray
{
    int m_size = 0; int* m_data = nullptr;
public:
    DynamicArray(int n, int v = int{}) : m_size{n}, m_data{new int[m_size]}
    { std::fill(m_data, m_data + m_size, v); }
    ~DynamicArray() { delete[] m_data; }
    DynamicArray(DynamicArray const& other)
        : m_size{other.m_size}, m_data{new int[m_size]}
    { std::copy(other.m_data, other.m_data + m_size, m_data); }
    DynamicArray& operator=(DynamicArray const& other)
    {
        if (this != &other) {
            delete[] m_data;
            m_size = other.m_size;
            m_data = new int[m_size];
            std::copy(other.m_data, other.m_data + m_size, m_data);
        }
        return *this;
    }
    ...
};
```

Smart pointers

- A *smart pointer* is an object that behaves like a pointer, but also manages the lifetime of the pointed-to object (i.e. the pointee)
- It leverages the RAII idiom
 - Resource Acquisition Is Initialization
 - Resource (e.g. memory) is acquired in the constructor
 - Resource (e.g. memory) is released in the destructor

Smart pointers (cont.)

```
template<typename Pointee>
class SmartPointer {
    Pointee* m_p;
public:
    explicit SmartPointer(Pointee* p): m_p{p} {}
    ~SmartPointer() { delete m_p; }
    Pointee* operator->() { return m_p; }
    Pointee& operator*() { return *m_p; }
};

class Sample { ... };

{
    SmartPointer<Sample> sp{new Sample{}};
    sp->add(...);
    (*sp).stats();
}
```

At the end of the scope (i.e. at the closing `}`) `sp` is destroyed and its destructor deletes the pointee

std::unique_ptr<T>

Standard smart pointer

- Exclusive ownership
- Minimal overhead, if any
- Non-copyable, movable

```
class Sample { ... };

void take(std::unique_ptr<Sample> q);           // by value

std::unique_ptr<Sample> p{new Sample{} }; // explicit new
auto p = std::make_unique<Sample>();        // better (*)
auto r = p;                                // error, non-copyable
take(p);                                  // error, non-copyable
auto r = std::move(p);                     // ok, movable
take(std::move(r));                        // ok, movable
```

(*) The possible arguments passed to `make_unique` are *forwarded* to the constructor

Disabling copy operations

- If a class cannot support copy semantics, its copy operations should be suppressed
- The cleanest way to do it is to mark the copy operations as `= delete`

```
template<typename Pointee>
class UniquePtr {
    Pointee* m_p;
public:
    explicit UniquePtr(Pointee* p) : m_p{p} {}
    ~UniquePtr() { delete m_p; }
    UniquePtr(UniquePtr const&) = delete;
    UniquePtr& operator=(UniquePtr const&) = delete;
    Pointee* operator->() { return m_p; }
    Pointee& operator*() { return *m_p; }
};
```

- Note that a deleted copy ctor is still a ctor and would disable the generation of the default ctor
- `= delete` is a general mechanism that can be applied to any function

Move semantics

- It allows to properly *move* (i.e. pass) the responsibility of a resource from one managing object to another
- Mainly driven by optimization considerations but also a solution to a proper management of resources
- Introduced in the language by C++11, thanks to a new type of reference, called *rvalue reference*
- An *rvalue reference* is identified by the token `&&`, e.g. `int&&`

Controlling moving

- The compiler automatically generates the move operations for a class, if used
 - The move constructor creates a **new** object, **re-using** the resources owned by another object

```
UniquePtr<int> p{new int{42}};  
auto q{std::move(p)}; // auto q = std::move(p)
```

- The move assignment operator changes the value of an **existing** object, **re-using** the resources owned by another object

```
UniquePtr<int> p{new int{12}};  
UniquePtr<int> q{new int{34}};  
q = std::move(p);
```

- `std::move` enables the use of a move operation instead of the corresponding copy operation
 - In practice it's a `static_cast` of an lvalue reference to an rvalue reference

Controlling moving (cont.)

- If there are no resources involved, a move operation in fact becomes a copy operation
- Move operations typically modify the source object, to steal its internal resources
- After a move the two objects are typically different
- A move should leave the original object in a “*valid but unspecified state*”
 - i.e. **the class invariant is preserved**
 - not always easy, especially for the move constructor
- Move operations should be declared `noexcept`, which means “this function/operation doesn’t fail” (more or less)
- The automatically generated operations may not be correct
- The move operations can be explicitly provided or suppressed

Controlling moving (cont.)

```
template <typename Pointee> class UniquePtr
{
    Pointee* m_p;
public:
    explicit UniquePtr(Pointee* p = nullptr) : m_p{p} {}
    ~UniquePtr() { delete m_p; }
    UniquePtr(UniquePtr const&) = delete;
    UniquePtr& operator=(UniquePtr const&) = delete;
    UniquePtr(UniquePtr&& other) noexcept
        : m_p{std::exchange(other.m_p, nullptr)}
    {
    }
    UniquePtr& operator=(UniquePtr&& other) noexcept
    {
        delete m_p;
        m_p = std::exchange(other.m_p, nullptr);
        return *this;
    }
    ...
};
```

Controlling moving (cont.)

- The move constructor typically takes an object of the same class by (non-const!) rvalue reference

```
UniquePtr(UniquePtrs&& other) : ... {...}
```

- The move assignment operator typically:
 - takes an object of the same class by (non-const) rvalue reference
 - returns the object itself by reference
 - does **not** check for auto-assignment

```
UniquePtr& operator=(UniquePtr&& other)
{
    ...
    return *this;
}
```

Special member functions

- A class has five special member functions
 - Plus the default constructor

```
class MyClass {  
    MyClass(MyClass const&);           // copy constructor  
    MyClass& operator=(MyClass const&); // copy assignment  
    MyClass(MyClass&&);              // move constructor  
    MyClass& operator=(MyClass&&);   // move assignment  
    ~MyClass();                      // destructor  
};
```

- The compiler can generate them automatically according to some convoluted rules
 - The behavior depends on the behavior of data members
- General guidelines:

Rule of zero Don't declare them and rely on the compiler

Rule of five If you need to declare one, declare them all

- Consider using =default and =delete

std::shared_ptr<T>

Standard smart pointer

- Shared ownership (reference counted)
- Some space and time overhead
 - for the management, not for access
- Copyable and movable

```
class Sample { ... };

void take(std::shared_ptr<Sample> q);           // by value

std::shared_ptr<Sample> p{new Sample{} };        // explicit new
auto p = std::make_shared<Sample>();             // better (*)
auto s = p;                                      // ok, copyable
take(p);                                         // ok, copyable
auto s = std::move(p);                          // ok, movable
take(std::move(s));                           // ok, movable
```

(*) The possible arguments passed to `make_shared` are *forwarded* to the constructor

Using smart pointers

- Give an owning raw pointer (e.g. the result of a call to `new`) to a smart pointer as soon as possible
- Prefer `unique_ptr` unless you need `shared_ptr`
 - You can always move a `unique_ptr` into a `shared_ptr`
 - But not viceversa
- Access to the raw pointer is available
 - e.g. to pass to legacy APIs
 - `smart_ptr<T>::get()`
 - returns a **non-owning** `T*`
 - `unique_ptr<T>::release()`
 - returns an **owning** `T*`
 - must be explicitly managed
- Arrays are supported

```
std::unique_ptr<int[]> p{new int[n]}; // destructor calls 'delete []'
```

Smart pointers and functions

Pass a smart pointer to a function only if the function needs to rely on the smart pointer itself

- by value of a `unique_ptr`, to transfer ownership

```
void take(std::unique_ptr<Sample> q);
auto p = std::make_unique<Sample>();
take(p);           // error
take(std::move(p)); // ok
```

- by value of a `shared_ptr`, to keep the resource alive

```
auto s = std::make_shared<Sample>();
std::thread t{[=] { do_something_with(s); }};
```

- by reference (possibly `const`), to interact with the smart pointer itself

```
void print_count(std::shared_ptr<Sample> const& s) {
    std::cout << s.use_count() << '\n';
}
auto s = std::make_shared<Sample>();
print_count(s);
```

Smart pointers and functions (cont.)

- Otherwise pass the pointee by (const) reference/pointer

```
void stats(std::shared_ptr<Sample> s) { if (s) s->stats(); }
void stats(Sample* t)                  { if (t) t->stats(); } // better
void stats(Sample& t)                { t.stats(); }          // better

auto s = std::make_shared<Sample>();
stats(s);
stats(s->get());
if (s) stats(*s);
```

- Return a smart pointer from a function if the function has dynamically allocated a resource that is passed to the caller

```
auto create() { return std::make_unique<Sample>(); }

auto u = create();      // std::unique_ptr<Sample>
std::shared_ptr<Sample> s = std::move(u);

std::shared_ptr<Sample> s = create();
```

Smart pointer custom deleter

- A smart pointer is a general-purpose resource handler
- The resource release is not necessarily done with `delete`
- `unique_ptr` and `shared_ptr` support a *custom deleter*

```
FILE* f = std::fopen(...);  
...  
std::fclose(f);
```

Usual problems:

- Who owns the resource?
- Forgetting to release
- Releasing twice
- Early return/throw

```
std::shared_ptr<FILE> file{  
    std::fopen(...), // pointer  
    [](FILE* f) { std::fclose(f); } // deleter  
};
```

Taxonomy of STL Containers

Sequence The client decides where an element gets inserted

- `array`, `deque`, `forward_list`, `list`, `vector`

Associative The container decides where an element gets inserted

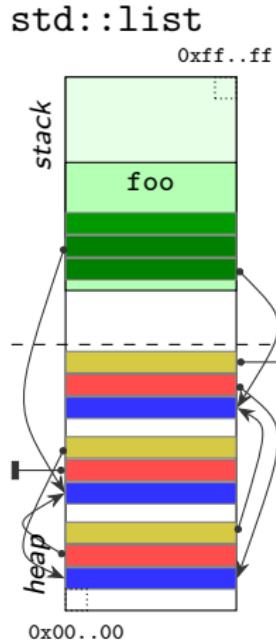
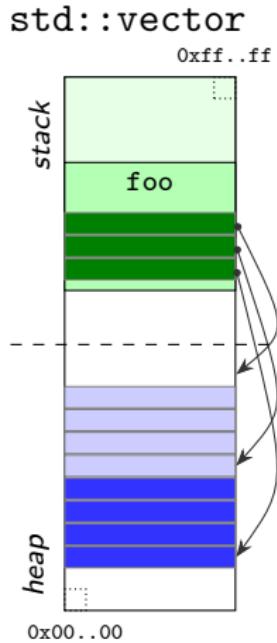
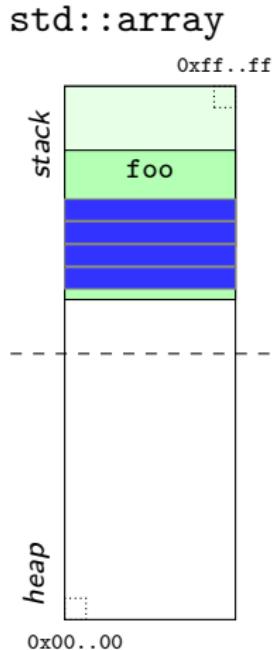
Ordered The elements are sorted based on a key

- `map`, `multimap`, `set`, `multiset`

Unordered The position of an element depends on the *hash* of its key

- `unordered_map`,
`unordered_multimap`,
`unordered_set`,
`unordered_multiset`

Sequence containers

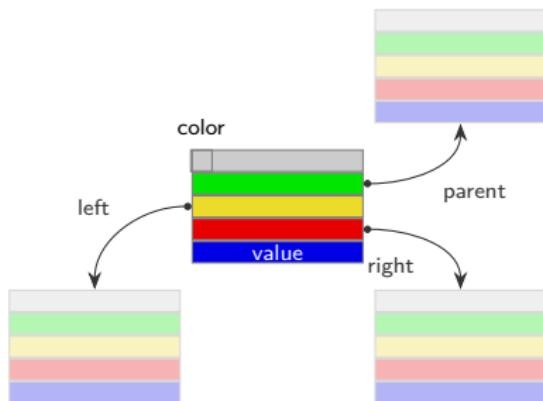
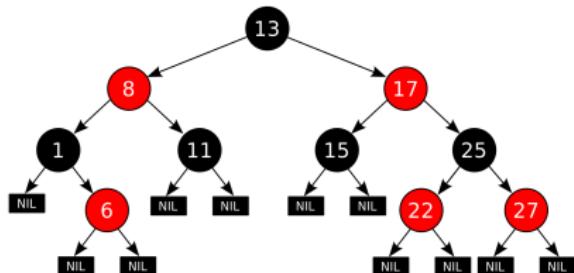


Sequence containers (cont.)

- `std::array`
 - fixed size, contiguous in memory
 - typical operations
 - `operator[]`
 - iteration with `begin/end`
- `std::vector`
 - dynamic size, contiguous in memory
 - typical operations
 - `push_back`
 - `operator[]`
 - iteration with `begin/end`
- `std::list`
 - dynamic size, non-contiguous in memory, iterator stability
 - typical operations
 - `push_back, push_front, insert`
 - iteration with `begin/end`

Associative ordered containers

- They contain ordered values (set and multiset) or key-value pairs (map and multimap)
- Search, removal and insertion have logarithmic complexity
- Typically implemented as balanced (red-black) trees



By Cburnett – Own work, CC BY-SA 3.0
<https://commons.wikimedia.org/w/index.php?curid=1508398>

Static data and functions

Non-local objects

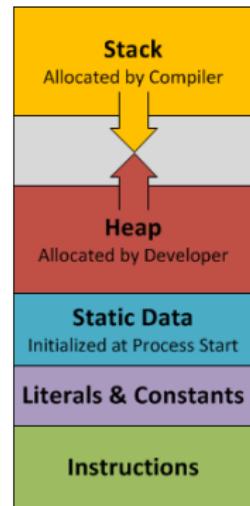
- Objects can be created outside a function block

```
#include <random>

std::default_random_engine eng;

auto seed_rand(int s) { eng.seed(s); }
auto get_rand()          { return eng(); }
auto print_rand()        { std::cout << get_rand(); }
```

- `eng` above is a *global* variable
- They have *static* storage duration, i.e. they live for the whole program duration
- They live in the **Static Data** memory segment
- They are initialized before `main` is called and destroyed after `main` has finished



Non-local objects (cont.)

- Initialization
 - Memory for non-local variables is guaranteed to be at least initialized to all zeroes
 - They may be initialized to a constant value known at compile time
 - They may be initialized dynamically at runtime, e.g. with the result of a function call
- Warning
 - Non-local variables make reasoning about the program more difficult because they encourage to work by side effects
 - The order of initialization and destruction is deterministic only in the same translation unit
 - **Better avoid them**, especially if mutable (i.e. non constant)

Constants

- One useful application of non-local variables is to define constants, possibly inside a namespace

```
namespace std::numbers {  
    inline constexpr double e = ... ;  
    inline constexpr double pi = ... ;  
    ...  
}
```

- `inline` means that there is only one object in the whole program (like for function definitions)
 - to be used if the definitions are in a header file
- `constexpr` guarantees that the initialization can be done at compile time
 - possibly through the execution of a *constexpr* function (not further discussed)
 - it implies `const`
- Of course constants can be defined locally as well

Class static data members

- A class data member declared **static** is not part of any object of that class
- A static data member
 - exists even if there are no objects of that class
 - has static storage duration
 - is defined out-of-class ...
 - ...unless it's declared **inline** (C++17) or **constexpr** (which implies **inline**) or is a **const integral type**

```
struct X {  
    static int n;  
};  
// probably in a .cpp file  
int X::n = 42;
```

```
struct X {  
    inline static int n = 42;  
};
```

Class static data members (cont.)

- A static data member is usually accessed using the scope operator ::

```
class std::chrono::system_clock {  
public:  
    static constexpr bool is_steady = ...;  
    ...  
};  
  
std::cout << std::chrono::system_clock::is_steady;
```

- But if there is an object of that type, one can use the . operator on that object

```
std::chrono::system_clock clock;  
std::cout << clock.is_steady;
```

Class static member functions

- A class member function declared **static** is not associated with any object of that class
- They are similar to normal functions, but defined in the scope of a class, so that they can access the other members
- A static member function can access static data members, but cannot access non-static data members
 - It cannot be declared as **const**

```
class std::chrono::system_clock {  
public:  
    static time_point now() { ... }  
    ...  
};  
// system_clock::now() could also be defined out-of-class
```

```
using namespace std::chrono_literals;  
std::this_thread::sleep_until(std::chrono::system_clock::now() + 15ms);
```

Dynamic polymorphism

Polymorphism

The provision of a single interface to entities of different types

static Based on concepts and templates

```
template<class Iterator, class T>
Iterator std::find(Iterator f, Iterator l, T const& v);

std::vector<int> v{...};
auto it = std::find(v.begin(), v.end(), 12);

std::list<Command> l{...};
auto it = std::find(l.begin(), l.end(), Command{"send"});
```

dynamic Based on inheritance and virtual functions

Inheritance

```
struct Base {  
    int a;  
    Base(int a) : a{a} {}  
    void f();  
    int operator()() const;  
};  
  
struct Derived : Base {  
    double d;  
    Derived(int i, double d)  
        : Base{i+1}, d{d} {}  
    int h() const;  
};  
  
Derived de{42, 3.14};  
de.d;  
de.h();  
de.a; // Base::a  
de.f(); // Base::f  
de(); // Base::operator()  
Base* b1 = &de;  
Base& b2 = de;
```

- A class may be declared as *derived* from one or more *base* classes
 - A hierarchy can be formed
- Members of the base class are also members of the derived class
 - Let's ignore access control (private, public) for the moment
- Constructing a Derived object constructs also the corresponding Base *sub-object*, either explicitly (like in this case) or implicitly
- Derived* can be implicitly converted to Base*
- A Derived object can bind to Base&

Dynamic polymorphism

Typical use case: a graphics system of shapes

```
struct Circle { // is a Shape
    Point c;
    double r;
    void move(Point p);
    Point where() const;
};

struct Rectangle { // is a Shape
    Point ul;
    Point lr;
    void move(Point p);
    Point where() const;
};

std::vector<Shape> shapes;           // I wish I could do this
for (auto const& s : shapes) { s.where(); } // wish
for (auto& s : shapes) { s.move(Point{1,2}); } // wish
```

Let's ignore access control (private, public) for a moment

Dynamic polymorphism (cont.)

```
struct Shape {                      // abstract base class
    virtual Point where() const = 0; // pure virtual function
    virtual ~Shape();              // virtual dtor; no '=' 0' here
};

struct Circle : Shape { // derived class
    Point c;
    int r;
    Point where() const override;
};

struct Rectangle : Shape { // derived class
    Point ul;
    Point lr;
    Point where() const override;
};

Shape* create_shape();   // return new Circle/Rectangle
auto s = create_shape();
s->where();            // better use a smart pointer
```

Dynamic polymorphism (cont.)

- A non-static member function (i.e. a method) is a **virtual** function if it is first declared with the keyword **virtual** or if it **overrides** a virtual member function declared in a base class
- A class with a virtual member function is called a **polymorphic** class.
- Virtual functions support **dynamic binding** and **object-oriented programming**
 - A call to a virtual function made through a pointer/reference to a base class is dispatched to the corresponding function of the actual derived class object behind that pointer/reference
- Overridden functions must have the same signature
 - Beware of member function hiding

Dynamic polymorphism (cont.)

- A virtual function is **pure** if its declaration ends with = 0
 - A pure virtual function cannot be defined (i.e. have an implementation) – with one exception (see later)
- A class is **abstract** if it has at least one pure virtual function
 - No objects of an abstract class can be created
 - An abstract class is typically used to represent an *interface*

Dynamic polymorphism (cont.)

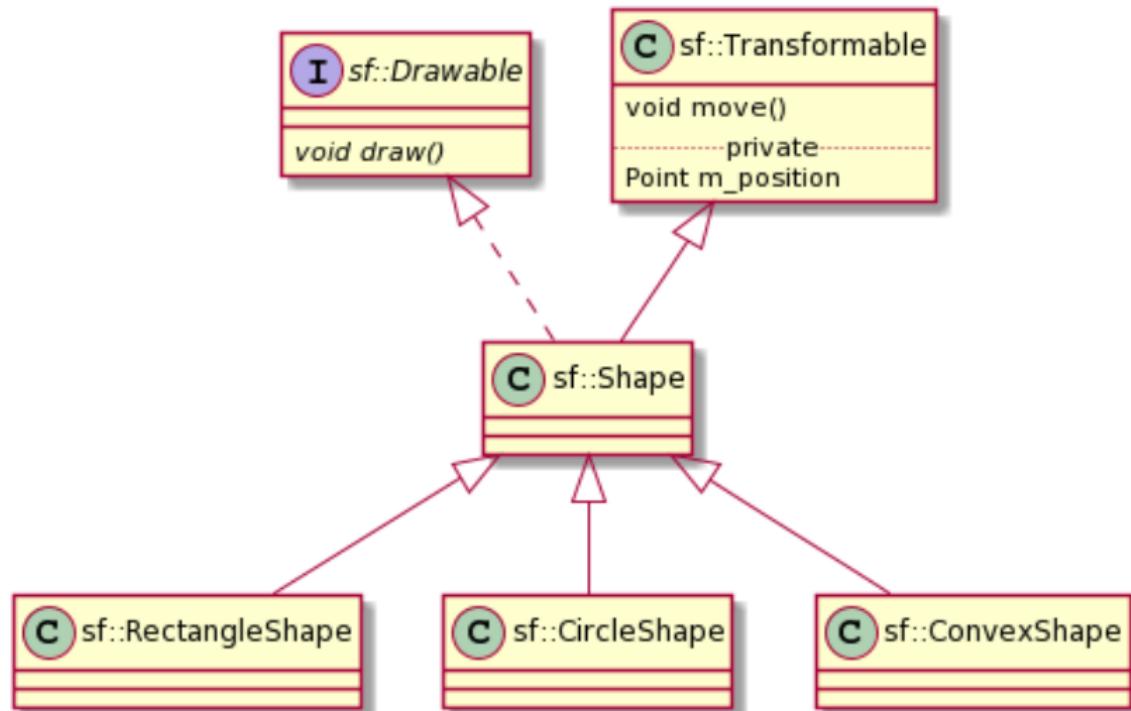
```
struct Base {  
    virtual void f(int) = 0;  
};  
  
struct Derived : Base {  
    void f(int) override {...}  
};  
  
Base b;                  // error  
Base* b1 = new Derived; // ok, owning pointer, remember to delete  
b1->f(0);              // calls Derived::f  
Derived d;              // ok  
Base* b2 = &d;          // ok, non-owning pointer, don't delete  
b2->f(0);              // calls Derived::f  
Base& b3 = d;          // ok  
b3.f(0);               // calls Derived::f
```

Mixing interface and implementation

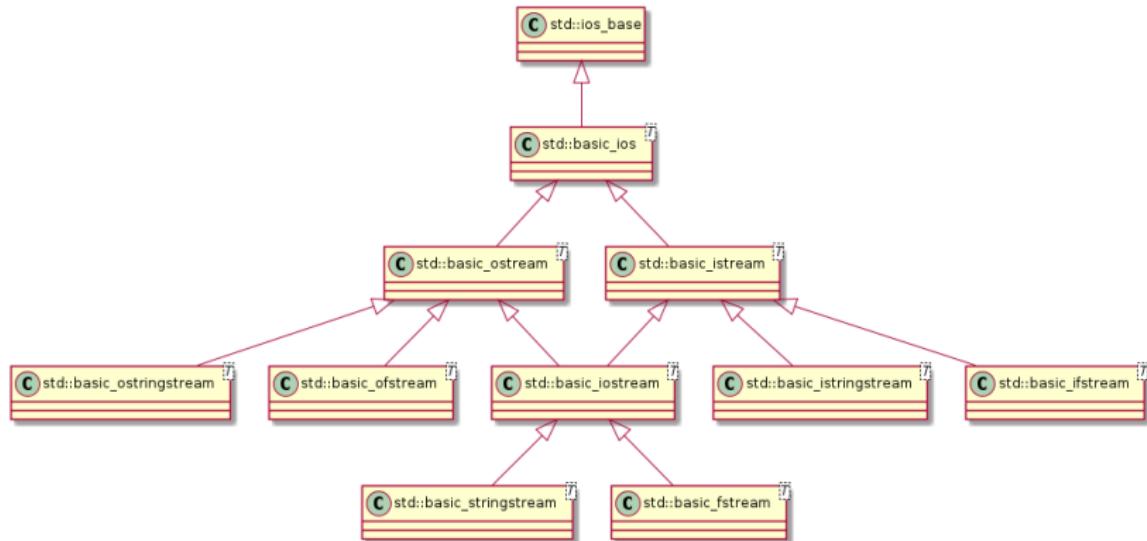
```
struct Shape {  
    Point p;  
    Shape(Point p) : p{p} {}  
    virtual ~Shape();  
    virtual Point where() const { return p; } // not pure; default implementation  
};  
  
struct Circle : Shape {  
    int r;  
    Circle(Point p, double d) : Shape{p}, r{d} {}  
    // where() implementation is inherited from Shape  
};  
  
struct Rectangle : Shape {  
    Point lr;  
    Rectangle(Point p1, Point p2) : Shape{p1}, lr{p2} {}  
    Point where() const override { return (p + lr) / 2; } // p is inherited  
};
```

Not recommended, especially for data members

Example: SFML



An example from the Standard Library: I/O streams



```
namespace std {
    using istream = basic_istream<char>;
    using ostream = basic_ostream<char>;
    using istringstream = basic_istringstream<char>;
    using ostringstream = basic_ostringstream<char>;
    using ifstream = basic_ifstream<char>;
    using ofstream = basic_ofstream<char>;
    ...
    istream cin;
    ostream cout;
}
```

I/O Streams

- C++ provides an Input/Output library based on the concept of *streams*, which abstract input/output devices
- `std::cin` and `std::cout` are streams attached to the standard input and output of the program (typically attached to the terminal)
- Other stream classes allow I/O from/to files and strings
- Since they are part of a polymorphic hierarchy these objects can be passed to functions implemented in terms of the base class
 - For example the *streaming* operators `<<` and `>>`

File I/O

```
std::ifstream is{/tmp/in}; // open /tmp/in for reading
std::ofstream os{/tmp/out}; // open /tmp/out for writing
if (!is || !os) {
    // manage error
}
double d;
while (is >> d) {
    os << std::setw(12) << d * 2 << '\n';
}
```

- `fstream` constructor opens the file, destructor closes it
- There are explicit `open` and `close` methods
- Many other operations/options (mostly inherited)

String I/O

```
std::istringstream is{"12. 14. -42."};  
std::ostringstream os;  
  
double d;  
while (is >> d) {  
    os << std::setw(12) << d * 2 << '\n';  
}  
  
std::cout << os.str();  
// " 24\n 28\n -84\n"
```

- `stringstream` constructor possibly allocates memory, destructor deallocates
- The `str` method can be used to get/set the contents of the `stringstream`
- Many other operations/options (mostly inherited)

operator<<

- operator<< is usually implemented as a free function
 - Note that the object we want to stream is the second parameter

```
class Complex {  
    double r;  
    double i;  
public:  
    double real() const;  
    double imag() const;  
    ...  
};  
  
inline std::ostream& operator<<(std::ostream& os, Complex const& c)  
{  
    os << '(' << c.real() << ',' << c.imag() << ')';  
    return os;  
}
```

- To be more general, it should actually be a function template and accept/return `std::basic_ostream<...>`

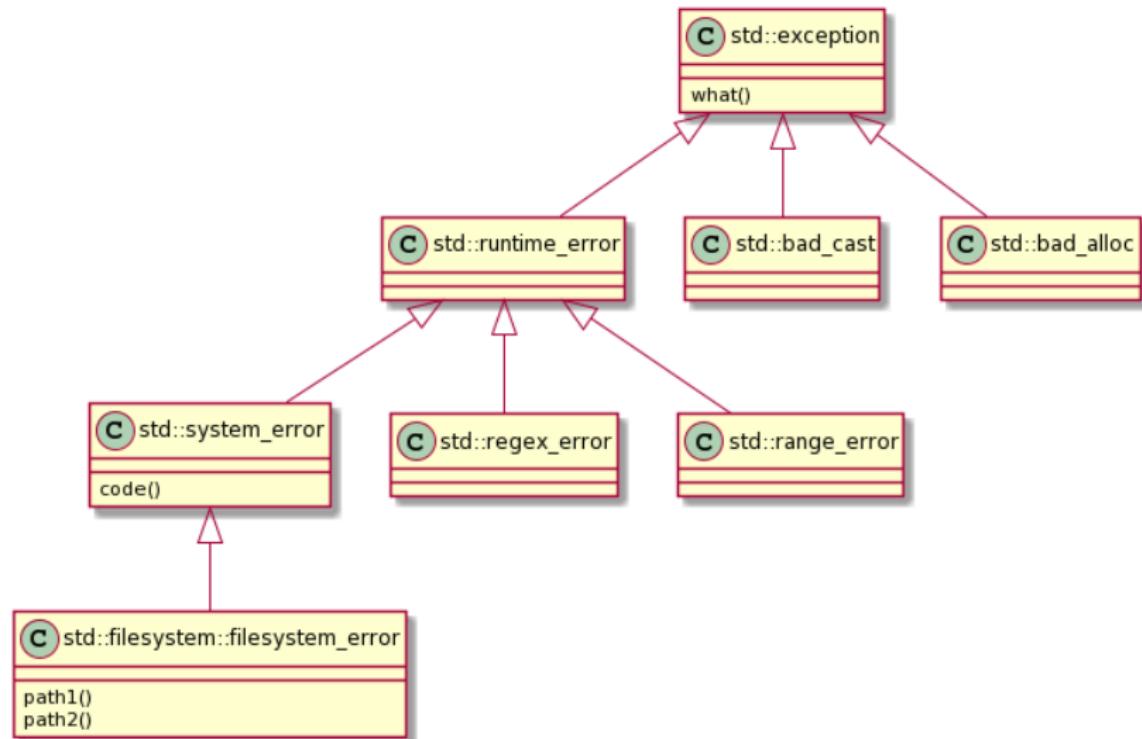
friend functions

- Sometimes a free function (e.g. `operator<<` needs private data that are not exposed (or not exposed efficiently) via the public interface
- In that case it can be implemented as a `friend` free function

```
class Complex {  
    double r; double i;  
public:  
    friend std::ostream& operator<<(std::ostream& os, Complex const& c) {  
        os << '(' << c.r << ',' << c.i << ')';  
        return os;  
    }  
    ...  
};
```

- Note how the function, being defined inside the class definition, is automatically `inline`
- It can be just declared inside the class definition and be defined outside, even in another file
- `friendship` has a much broader usage than shown here
 - E.g. a whole class can be `friend` of another

Another example from the Standard Library: Exceptions



Note: the hierarchy is incomplete

Multiple exception-catch clauses

- A try-block can have multiple catch clauses
- The first that matches the type of the exception is chosen
- The order is important: put the more specific ones first

```
auto read_from(std::filesystem::path const& p) {
    std::ifstream is(p);
    if (!is) {
        throw std::filesystem::filesystem_error{
            "read_from", p, std::make_error_code(std::errc::invalid_argument)
        };
    }
    ...
}

auto g() {
    try {
        read_from("/tmp/data");
        ...
    } catch (std::filesystem::filesystem_error const& e) {
        std::cerr << e.what();
    } catch (std::exception const& e) {
        std::cerr << e.what();
    } catch (...) { // it's really three dots!
        std::cerr << "unknown exception";
    }
}
```

Overriding vs hiding

```
struct Base {  
    virtual void f(int);  
};  
  
struct Derived : Base {  
    void f(int); // overriding, virtuality is "inherited"  
};  
  
Derived d;  
Base& b = d;  
b.f(1);           // call Derived::f(int)
```

- A virtual function should specify exactly one of `virtual`, `override`, or `final`
- A `final` virtual function cannot be overridden in a derived class
- A class can be declared `final` and cannot be derived from

```
struct Derived final { ... };
```

Overriding vs hiding (cont.)

```
struct Base
{
    virtual void f(int);
};

struct Derived : Base
{
    virtual void f(unsigned); // this is another function, hiding Base::f
};

Derived d;
Base& b = d;
b.f(1);           // call Base::f(int)
b.f(1U);          // call Base::f(int)
d.f(1);           // call Derived::f(unsigned)
d.f(1U);          // call Derived::f(unsigned)
```

- Virtual functions should specify exactly one of `virtual`, `override`, or `final`
- Specifying `override` (or `final`) would have caused a compilation error

Overriding vs hiding (cont.)

```
struct Base
{
    virtual void f(int) const;
};

struct Derived : Base
{
    virtual void f(int); // this is another function, hiding Base::f
};

Derived d;
Base& b = d;
b.f(1);           // call Base::f(int) const
```

- Virtual functions should specify exactly one of `virtual`, `override`, or `final`
- Specifying `override` (or `final`) would have caused a compilation error

Slicing

A base class without pure virtual functions is not abstract any more

- It can be instantiated
- It can be copied
 - unless precautions are taken, e.g. copy/move operations are deleted

```
struct Shape
{
    Point p;
    Shape(Point p): p{p} {}
    virtual ~Shape() = default;
    virtual Point where() const { return p; }
};

struct Rectangle : Shape { ... };

Shape s; // desirable?
Shape s2 = s1; // desirable?
Rectangle rect{...};
Shape s = rect; // desirable??
```

Slicing (cont.)

```
void process1(Shape& shape) // by reference
{
    ... shape.where() ... // Point{2., 4.}
}

void process2(Shape shape) // by value
{
    ... shape.where() ... // Point{1., 7.}
}

auto rect = Rectangle{Point{1., 7.}, Point{3., 1.}};
process1(rect);
process2(rect);
```

- When an object obj of a derived class is passed to a function that takes a parameter of a base class **by value**, only the base class sub-object of obj is passed to the function

Keeping the base class abstract

- It's good practice to keep base classes abstract
- A base class without pure virtual functions is not abstract any more
- To prevent this, declare the destructor as pure virtual
- Yet the destructor needs to be defined
 - So that derived classes are properly destroyed

```
struct Shape {  
    Point ul;  
    Shape(Point p): p{p} {}  
    virtual ~Shape() = 0;  
    virtual Point where() const; // non-pure virtual function  
};  
inline Shape::~Shape() = default; // or any other implementation  
  
Shape s; // not really desirable
```

Access control

A member of a class can be

public Its name can be used **anywhere** without access restriction

private Its name can be used **only by members** and friends of the class in which it is declared

protected Its name can be used only **by members** and friends of the class in which it is declared, **by classes derived** from that class, and by their friends

Accessibility through derivation

```
class Base {  
    private:  
    ...  
    protected:  
    ...  
    public:  
    ...  
};  
  
class Derived : public|private|protected Base {};
```

Derivation itself can be

public public in B → public in D

protected in B → protected in D

- Sub-typing (*is-a* relationship)
- This is the one you should use

private public or protected in B → private in D

- Implementation inheritance

protected public or protected in B → protected in D

- Rarely, if ever, used

protected access

```
class Shape {
protected:
    Point p;
public:
    Shape(Point p) : p{p} {}
    virtual ~Shape() = default;
    virtual Point where() const { return p; }
    virtual double area() const = 0;
};

class Rectangle : public Shape { // is-a
private:
    Point lr;
public:
    Rectangle(Point p1, Point p2) : Shape{p1}, lr{r} {}
    double area() const override { ... std::abs(p.x - lr.x) ... }
};
```

- protected and public members represent an interface
- Data members are a poor interface, keep them private

Structural inheritance

- Inheritance is applicable also in non-polymorphic situations
- To reuse and possibly extend the implementation and the interface of a class
- A way to create a distinct type with the same implementation and interface of another
- Consider composition, i.e. a member, or private inheritance

```
class Vector : public std::vector<int> {  
    using std::vector<int>::vector; // inherit constructors  
};  
  
void process(std::vector<int>&); // #1  
void process(Vector&); // #2  
void signal(std::vector<int>&); // #3  
  
std::vector<int> v1{...};  
Vector v2{...}; // all std::vector ctors are available  
process(v1); // call #1  
process(v2); // call #2  
signal(v2); // call #3
```

Destruction and inheritance

- In case of polymorphic inheritance, the destructor of a base class should be
 - public and virtual, or
 - protected and non-virtual
- In case of structural inheritance, don't delete through a pointer to the base class

```
class Vector : public std::vector<int> {};  
  
Vector v;                                // ok  
  
Vector* pv = new Vector;  
delete pv;                                // ok  
  
std::vector<int>* ps = new Vector;  
delete ps;                                 // undefined behavior
```

Copying/moving and inheritance

- Dynamic polymorphism and value semantics don't play well together
 - See slicing
- The copy/move operations of a base class shouldn't be publicly accessible, but they need to be accessible to a derived class
 - Declare them `protected`

```
class Base
{
    ...
protected:
    Base(Base const&);
    Base& operator=(Base const&);
    Base(Base&&);
    Base& operator=(Base&&);
public:
    ...
};
```

Copying/moving and inheritance (cont.)

- If you need some form of *deep* copy, consider providing a virtual *clone* operation

```
class Base
{
public:
    virtual Base* clone() const = 0;
};

class Derived: public Base
{
public:
    Derived* clone() const override
    {
        return new Derived{*this}; // call the copy ctor of Derived
    }
};
```

- Note the return type: an overridden function can return a derived class
 - *co-variant* return type

Epilog

What we have learnt in this course

- Elements of computer architecture and operating systems
- Objects, types, variables
- Expressions
- Statements and structured programming
- Functions and function objects
- User-defined types and classes and operator overloading
- Generic programming and templates
- The Standard Library, containers and algorithms
- Compilation model
- Elements of software engineering and supporting tools
- Dynamic memory allocation and resource management
- Dynamic polymorphism (aka object-oriented programming)

Where to go from here

- C++ is a large and complex language and has a large and complex standard library
 - Many more libraries are available from third parties
 - See, for example, `vcpkg` or `conan`
- We have just scratched the surface of what the language and the libraries provide
- There are many high-quality resources to go deeper
 - Learn C++
 - News, Status & Discussion about Standard C++
 - C++ reference
 - C++ Core Guidelines
 - C++ Conference (presentations, videos)
 - C++ Now Conference (presentations, videos)
 - Meeting C++ Conference (presentations, videos)
 - C++ Weekly
 - Existing source code (e.g. boost libraries or SFML)