CINECA

Introduction to Scientific and Technical Computing in C++

Leonardo Salicari, Alessandro Romeo, Michael Redenti, Lara Querciagrossa
l.salicari@cineca.it

Maggio 2026

When and Why C++

First release in 1985 in response to the need for a new programming language with the following features[1]:

- Reuse of code
- Fast and portable
- Easy to read and write
- User-defined types with low overhead
- Multi-paradigm
- Interoperability with other languages

"No programming language is perfect. Fortunately, a programming language does not have to be perfect to be a good tool for building great systems. In fact, a general-purpose programming language cannot be perfect for all of the many tasks to which it is put. [...] Thus, C++ was designed to be a good tool for building a wide variety of systems and to allow a wide variety of ideas to be expressed directly."

Bjarne Stroustrup

[1] reference

C++ characteristics

C++ developed as a superset of C, but from C99 it is strictly not anymore

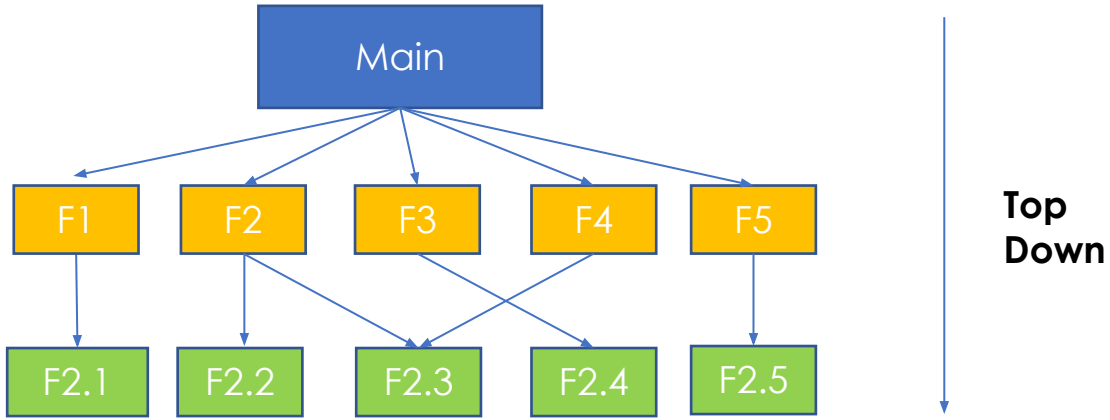
- Strongly and statically typed
- General-purpose
- Multi-paradigms
- From “low” to “high” level
- Standardized (“you code will compile for decades”)

C and C++

Features	C	C++
Keywords	53 as for C23	98 (`class`, `private`, `public`, `protected` etc) as for C++26
Memory Function	`stdlib.h`: `malloc`, `realloc`, `free`	`new`, `delete`
Input/output	`stdio.h`: `scanf`, `printf`, `fgets`	`iostream`: `cin`, `cout`
Namespace	No	Yes
Pointer & Reference	Pointers	**References** and pointers
Functions	No overloading Cannot define functions as struct members No default argument No generic programming C pass arguments by value only	**Overloading** Functions in Struct Default arguments **Templates** C++ pass arguments by value and by reference
Data Types	Built-in (`float`, `int`, `double`, `char`, `long`) + Boolean from C23! Derived (array, pointer) Structured (`enum`, `union`, `struct`)	Built-in (`float`, `int`, `double`, `char`, `long`) + Boolean Derived (array, pointer) Structured (`enum`, `union`, `struct`) **Class** – user-defined
Exception Handling	No built-in	`Try` and `catch` blocks, `throw`

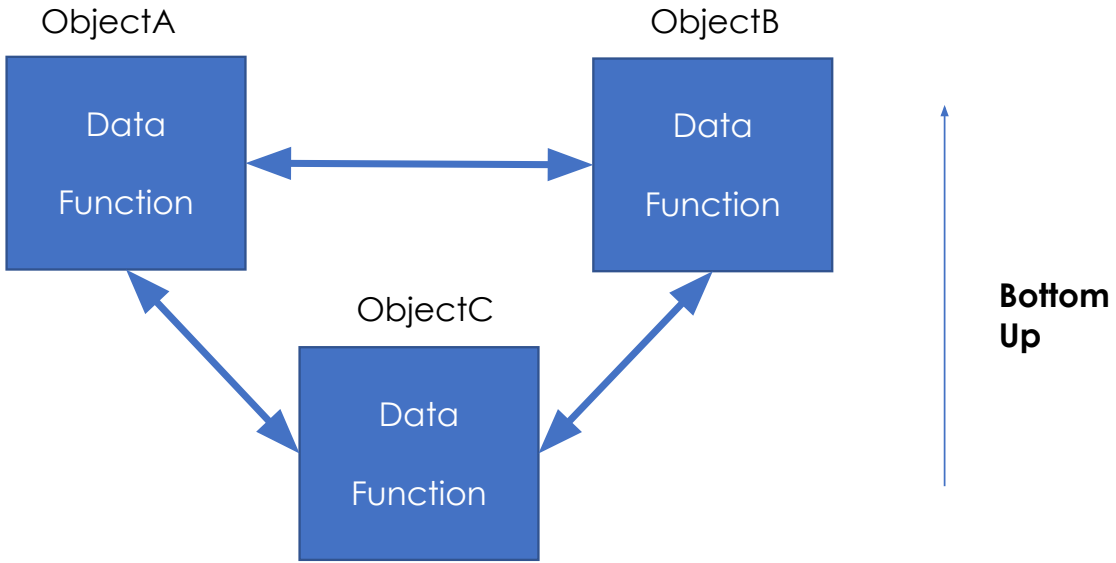
C: Top-down approach

C adopts a **procedural** programming paradigm:
the problem is decomposed into a series of elementary operations



C++: Bottom-up approach

C++ adds an **object-oriented** programming paradigm, identification with abstract entities and relationships among these



Although, C++ implements also other paradigms, for example functional

Comparing C and C++
Input/output

printf()

- Used in C and C++
- Function

```
#include <stdio.h>
printf("value: %d", val);
```

cout

- Used only in C++
- Object using output stream

```
#include <iostream>
std::cout << "value: " << val << std::endl;
```

First example of C++: I/O

```
/* roots of a second degree equation with real coefficients */
#include <cmath>        // provides sqrt() function
#include <iostream>     // input-output stream library

int main() {
    double a, b, c;
    std::cout << "Solving ax^2+bx+c=0, enter a: ";
    std::cin >> a;
    std::cout << std::endl << " enter b: ";
    std::cin >> b;
    std::cout << std::endl << " enter c: ";
    std::cin >> c;
    double delta = b * b - 4.0 * a * c;
    delta = sqrt (delta);
    double x1, x2;
    x1 = (-b + delta) / (2.0 * a); x2 = (-b - delta) / (2.0 * a);
    std::cout << "Real roots: " << x1 << ", " << x2 << std::endl;
    return 0;
}
```

<< Insertion operator

>> extraction operator

Road to the executable

main.cpp

```cpp
#include <iostream>
int main() {
    int x = 2;
    std::cout << "Your lucky number is " << x
<< "\n";
    return 0;
}
```

Source code (text)

"Compilation"

```
gcc main.cpp
```

a.out

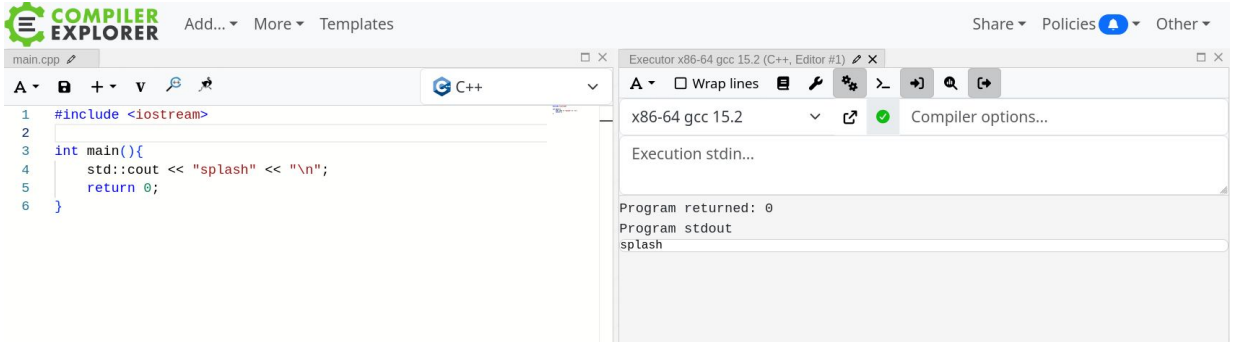
```
400468: f3 0f 1e fa
40046c: 48 83 ec 08
400470: 48 8b 05 79 0b 20 00
400477: 48 85
...
```

Executable (machine code)

[1] From CINECA's course on building systems, Tommaso Gorni

Compiler explorer

For the most part of the course, we will use an external service that compiles remotely our code



Features:

- explore intermediate compilation steps (e.g. assembly produced)
- compile and execute code for rapid prototyping
- multi-language support
- …

You can access a simple C++ template here: https://godbolt.org/z/6nKa3K4zx

Variables definition

Variables definition in C++

An object is used to store a *value* in memory. A **variable** is an object that has a name (identifier).

```
int main() {
    int x; // define a variable named x (of type int)
    int a, b; // define multiple int variable at once

    double width; // define a double variable named width
    width = 5; // assignment of value 5 into variable width

    double width { 5 }; // define variable width and initialize with initial value 5
    // {5} is the notation to call the initializer
}
```

Initialize a variable in C++

There are multiple ways to initialize a variable/object in C++

```
int main() {
    int a;          // default-initialization (no initializer)

    // “Traditional” initialization forms:
    int b = 5;      // copy-initialization (“copy” the r.h.s value into b)
    int c ( 6 );    // direct-initialization (calls the constructor directly)

    // “Modern” initialization forms:
    int d { 7 };    // direct-list-initialization
    int e {};       // value-initialization (empty braces)
}
```

Data types

Comparing C and C++

Data types

Built-in (float, int, double, char, long)[1]
Derived (array, pointer)
Structured (enum, union, struct

Built-in (float, int, double, char, long) + **Boolean & String**
Derived (array, pointer)
Structured (enum, union, struct)
User-defined : **Classes**

[1] Pre-C23

Built-in/fundamental types

Built-in: double (8 bytes), float (4 bytes), long (*i.e.* long int 4 bytes), int (2 bytes), char (1 byte), **bool** (1 byte, true/false)

Built-in: int

Keyword: **`int`**

Typically[1], requires 4 bytes of memory and range from -2147483648 to 2147483647

Type	Size (bytes, Unix 64 bit)	Range (2^(N-1), N=bits)
Short int	2	-32768 to 32767
int	4	-2147483648 to 2147483647
long int	8	-9223372036854775808 to 9223372036854775807
long long int	8	-(2^63) to (2^63)-1

[1] system-dependent: e.g. `int` might be 2 bytes in embedded systems

Built-in: int

Keywords:

- **`signed int`** (negative and positive integers values): from - 2^{N-1} to 2^{N-1} -1
- **`unsigned int`** (only positive integers values): from 0 to 2^{N} -1

Guarantee from C++20

Ranges signed int	Ranges unsigned int
from -2^{31} (-2 147 483 648) to 2^{31} – 1 (2 147 483 647)	from 0 to 2^{32} -1 (4 294 967 295)

Built-in: int

Unsigned integer **overflow**

```
#include <iostream>

int main() {
      unsigned int x{ 2 };
      unsigned int y{ 3 };
      std::cout << x - y << '\n';
}
```

Output: 4294967295

Wrong!

```
#include <iostream>

int main() {
      unsigned int u{ 2 };
      signed int s{ 3 };
      std::cout << u - s << '\n';
}
```

Output: 4294967295

Wrong!

Note that overflows occur when the value that you want to store is bigger than the highest representable by the type, hence it can occur in floats too

Built-in: floating points

Keywords: **`float`**, **`double`**
It's type variable that can hold a number with a **fractional** component.
Each type has a *range* and *precision* of representation

Type	Size (bytes)	Range
`float`	4	-3.4E+38 to +3.4E+38
`double`	8	-1.79769E+308 to 1.79769E+308
`long double`	8, 12, 16 (implementation-dependent)	-1.18E+4932 to 1.79769E+308

Special values:

- Not-a-number (NaN)
- Infinity (+ and -)
- negative 0: it equals to positive zero, used for special cases such as `1.0/-0 = -Infinity`

Built-in: float vs double

The *precision* of a floating point type defines **how many significant digits** it can represent without information loss. This depends on the order of magnitude of the value to represent:

- Floats have 6 to 9 digits precision
- Double have 15 to 18 digits of precision

```
#include <iomanip> // for output manipulator std::setprecision()
#include <iostream>

int main() {
      std::cout << std::setprecision(17); // show 17 digits of precision
      std::cout << 3.33333333333333333333333333333333333333f <<'\n'; // f suffix means float
      std::cout << 3.33333333333333333333333333333333333333 << '\n'; // no suffix means double
      return 0;
}
```

```
3.3333332538604736
3.3333333333333335
```

Built-in: boolean

Keyword: **bool**

A Boolean can be initialized with two values: *true* or *false* (1 and 0 respectively)

```
#include <iostream>

int main() {
      std::cout << true << '\n'; // true evaluates to 1
      std::cout << !true << '\n'; // !true evaluates to 0
      std::cout << std::boolalpha; // print bools as true or false
      std::cout << true << '\n’;

      return 0;
}
```

```
1
0
true
```

Built-in: char

Keyword: **char**

A **character** can be a single letter, number, symbol, or whitespace.
It is stored as integer interpreted as an ASCII character.

```
char ch = 5; // initialize with integer 5 (stored as integer 5)
char ch = '5'; // initialize with code point for '5' (stored as integer 53)
```

```
#include <iostream>
int main() {
	std::cout << "Input a keyboard character: "; // assume the user enters "abcd" (without quotes)
	char ch{}; std::cin >> ch; // ch = 'a', "bcd" is left queued.
	std::cout << "You entered: " << ch << '\n'; // Note: The following cin doesn't ask the user for input, it
grabs queued input!
	std::cin >> ch; // ch = 'b', "cd" is left queued.
	std::cout << "You entered: " << ch << '\n';
}
```

ESCAPE SEQUENCES: sequences of characters that have special meaning and starts with a '\' (backslash) character, and then a following letter or number.

Built-in: void

Keyword: **void**

void means “no type”

The compiler knows about the existence of such types, but does not have enough information to determine how much memory to allocate

```
void writeValue(int x) {
     std::cout << "The value of x is: " << x << '\n’;
}
```

No return statement, because this function doesn't return a value

Duration, Scope, and Linkage

Duration

A variable's duration determines when it is created and destroyed

- **Automatic duration**: objects are created at the point of definition, and destroyed when the block they are part of is exited:
  - local variables
  - function parameters
- **Static duration**: objects are created when the program begins and destroyed when the program ends:
  - global variables
  - static local variables
- **Dynamic duration**: objects are created and destroyed by programmer request.

Scope

An identifier's scope determines where the identifier can be accessed within the source code

- **Local scope**: Variables with local scope can only be accessed from the point of declaration until the end of the block in which they are declared
  - Local variables
  - Function parameters
  - Program-defined type definitions (such as enums and classes) declared inside a block
- **Global scope**: can be accessed from the point of declaration until the end of the file
  - Global variables
  - Functions
  - Program-defined type definitions (such as enums and classes) declared inside a namespace or in the global scope

Linkage

For *global* variable:

- **Internal linkage**: An identifier implementing internal linkage *is not* accessible outside the translation unit[1] it is declared in.

  It is implemented by the keyword `static`[2] for global variable

```
static int internalVariable = 5;
```

- **External linkage**: An identifier implementing external linkage *is* visible to every translation unit. Externally linked identifiers are shared between translation units.

  It is implemented by the keyword `extern`

```
extern int externalVariable = 5;
```

If the linkage is not specified, **default** linkage is extern (external linkage) for non-const symbols and static (internal linkage) for const symbols.

[1] Source file output of the preprocessor (file inclusion, macros expansion etc.) used by the compiler to generate the object file
[2] This keyword has a different meaning when used within functions and classes

Intermezzo: the const keyword

The `const` keyword declare a variable[1] as **immutable**, hence its value cannot change during the variable lifetime

```
int main(){
    const double pi {3.1415};
    pi = 3; // Compiler error!
}
```

[1] Functions, methods and other expressions can be labeled as immutable (see next lectures)

`const` vs `#define`

- a constant variable has a type
- a constant variable can be defined in a restricted scope (not global)
- the constness can be cast away

Memory allocations

Memory allocation

C++ support three basic types of memory allocation:

- **Static memory allocation** happens for static and global variables. Memory for these types of variables is allocated once when your program is run and persists throughout the life of your program.
  The size of the variable/array must be *known at compile time*.

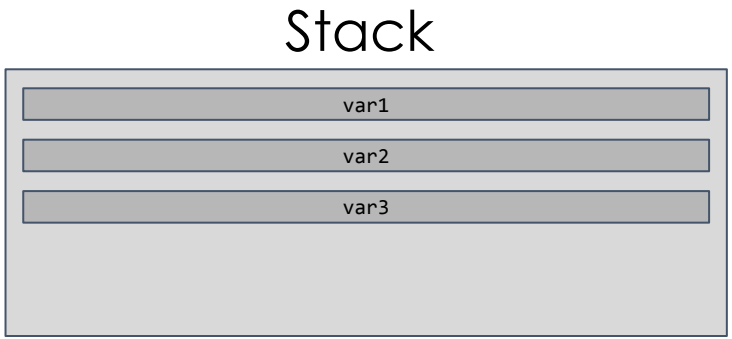
- **Automatic memory allocation** happens for function parameters and local variables. Memory for these types of variables is allocated when the relevant block is entered, and freed when the block is exited, as many times as necessary.
  Also in this case the size of the variable/array must be *known at compile time*.

- **Dynamic memory allocation** is a way for running programs to request memory from the operating system when needed, enabling objects to be *allocated memory at runtime*.

Stack and heap

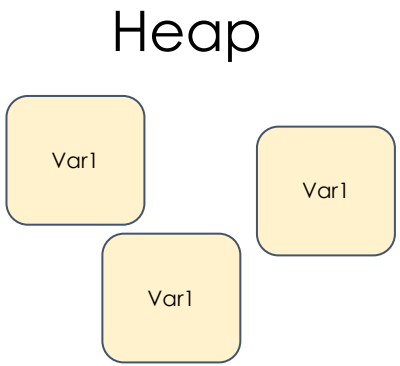
Automatic memory **allocation** uses the Stack
Properties:

- *Last in first out* principle
- Managed by the OS

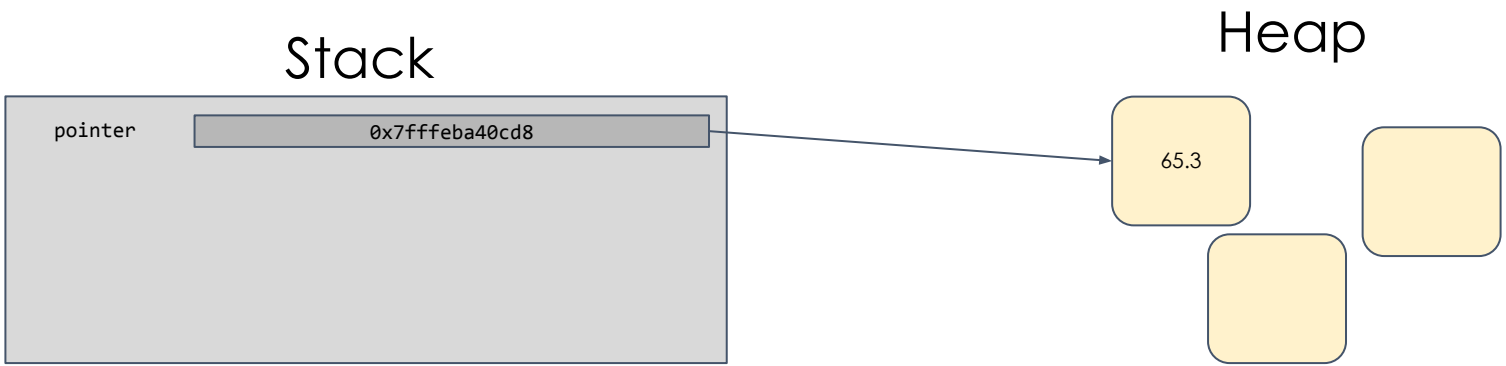


Dynamic memory **allocation** uses the Heap
Properties:

- Managed by the program at runtime
- Slower than stack
- More expensive on memory than stack



Dynamic allocation principle

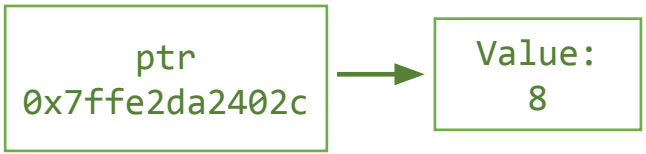


Pointers

Pointers

A **pointer** is a variable that holds a *memory address* (typically of another variable) as its value.

```
#include <iostream>

int main() {
      int *ptr;
      int a = 8;
      ptr = &a; // & is the address-of operator
      std::cout <<  ptr << std::endl; //output: 0x7ffe2da2402c
      std::cout << *ptr << std::endl; //output: 8
      return 0;
}
```



- **indirection/dereference operator *:** access pointee
- **address-of operator &**: return pointer address

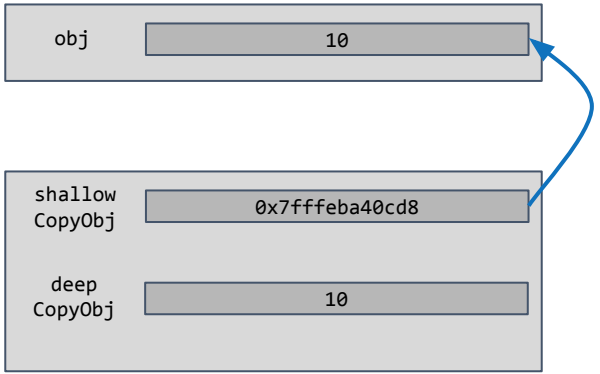
Pointers

We can change both what the pointer is pointing at (by assigning the pointer a new address to hold) or change the value at the address being held (by assigning a new value to the dereferenced pointer).

```
#include <iostream>

int main() {
     int *ptr;
     int a = 8;
     ptr = &a;
     int b = 7;
     double c = 8.6;
     ptr = &b; // ok we can change the address held
     ptr = &c; // error: cannot convert ‘double*’ to ‘int*’ in assignment
     return 0;
}
```

Intermezzo: shallow vs deep copy

```cpp
#include <iostream>

int main() {
    int obj = 10;

    int *shallowCopyObj = &obj;     // shallow copy of address
    *shallowCopyObj = 99;

    std::cout << obj << std::endl; // prints 99

    obj = 10;                      // back to initial value
    int deepCopyObj {obj};         // deep copy of the whole object
    deepCopyObj = 30;

    std::cout << obj << std::endl; // prints 10
}
```



Pointers to const

When the data type being pointed to is `const`, the value being pointed to can't be changed. To declare a pointer to a `const` value, use the const keyword **before** the pointer's data type:

```
int main() {
      const int x = 5;
      const int *ptr  =  &x;      // okay: ptr is pointing to a "const int"
      *ptr = 6;                   // not allowed: we can't change a const value
```

A pointer to `const` is not `const` itself (it just points to a `const` value), we can change what the pointer is pointing at by assigning the pointer a new address:

```
      const int y = 6;
      int z = 10;
      ptr = &y;  // okay: ptr now points at const int y
      ptr = &z   // The pointers will treat z as const
      return 0;
}
```

const pointer

A **const pointer** is a pointer whose address *cannot* be changed after initialization.
To declare a `const` pointer, use the `const` keyword **after** the asterisk in the pointer declaration:

```
int main() {
      int x = 5;
      int* const ptr = &x;  // ptr is initialized to the address of x
      int y = 10;
      ptr = &y;             // error: once initialized, a const pointer can not be changed.
```

Because the value being pointed to is non-`const`, it is possible to change the value being pointed to via dereferencing the `const` pointer

```
      *ptr = 10;            // The value of x now is 10
      return 0;
}
```

Null Pointers

A **null value** (often shortened to null) is a special value that means something has no value.
When a pointer is holding a null value, it means the pointer is not pointing at anything.
Such a pointer is called a **null pointer**.

```
#include <iostream>

int main() {
      int *ptr {};                // Create a null pointer
      ptr = nullptr;              // Or you can assign nullptr to it
      std::cout << *ptr << '\n'; // Dereference: program terminated with signal: SIGSEGV
      return 0;
}
```

Null Pointers

A pointer should either hold the address of a valid object, or be set to `nullptr`. That way we only need to test pointers for null, and can assume any non-null pointer is valid.

```
// Assume ptr is some pointer that may or may not be a null pointer
if (ptr) // if ptr is not a null pointer
    std::cout << *ptr << '\n'; // okay to dereference
else
    // do something else that doesn't involve dereferencing ptr (print an error message, do nothing at
all, etc...)
```

WARN: Dereferencing a null or dangling pointers it is undefined behaviour

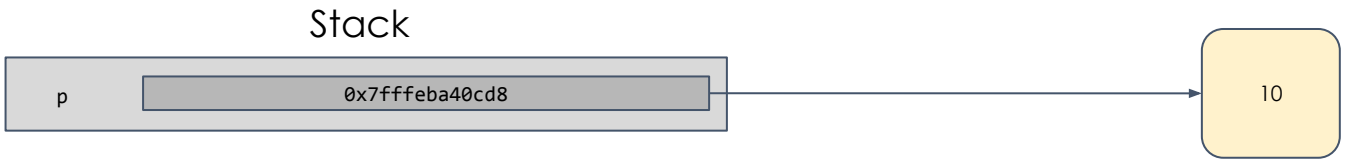
Dynamic memory allocation

C++ supports dynamic allocation and deallocation of memory using `new` and `delete` operators.

```cpp
// C++ allocation/deallocation
int *p  = new int{10};
delete p;
```

```cpp
// C allocation/deallocation
int *p  = (int*)malloc(sizeof(int));
free(p);
```

- **`new`**: creates the object and returns a pointer containing the address of the memory that has been allocated
- **`delete`**: returns the memory being pointed to back to the operating system, however it does not delete data. The operating system is then free to reassign that memory to another application.



WARN: A pointer that is pointing to deallocated memory is called a dangling pointer.

Dereferencing or deleting a dangling pointer will lead to undefined behaviour.

Smart pointers in a nutshell

Objects that behave like pointers and manage the lifetime of the pointee

- Resource (memory) is acquired when allocated
- Resource (memory) is released at the end of the scope
- Cannot be copied

Guaranteed no leak nor double release, even in presence of exceptions

```
#include <iostream>
#include <memory> // for std::unique_ptr

int main() {
    std::unique_ptr<int> num(new int); // Dynamic allocation in the heap
    *num = 10;
    std::cout << *num << std::endl;
} // automatic deallocation once the program leaves the scope
```

References

Value categories

All *expressions* in C++ have two properties: a **type** and a **value** category

```
int main() {
    int x;
    x = 5;        // valid: we can assign 5 to x
    5 = x;        // error: can not assign value of x to literal value 5
}
```

Why the second result into an error? The compiler checks if the expression resolves into a value (category).

Values can be:

- **named** (**lvalue**, as in left-side) or
- **unnamed/temporary** (**rvalue**, as in right-side)[1]

[1] In the post-C++11 era, values can also be glvalue, prvalue and xvalue; however, these are for more niche applications

References

In C++, a **reference** is an alias for an existing object. Once a reference has been defined, any operation on the reference is applied to the object being referenced.

The ampersand character, &, is used to define a reference

```
int main() {
    int x { 5 };
    int& ref { x };   // ref is an reference, a.k.a. alias, to x

    ref += 5;         // Now x stores the value 10
}
```

lvalue reference

An **lvalue reference** (commonly just called a "reference" since prior to C++11 there was only one type of reference) acts as an alias for an existing lvalue (such as a variable).

```
#include <iostream>

int main() {
    int x { 5 };
    int& ref { x };   // ref is an lvalue reference variable
                      // that can now be used as an alias for variable x

    std::cout << x << '\n';      // print the value of x (5)
    std::cout << ref << '\n';    // print the value of x via ref (5)
}
```

rvalue reference

An expression that evaluates to an identifiable object (named) is an lvalue, if it evaluates to a (n *unnamed*) value is an **rvalue**[1]

```
#include <iostream>

int main() {
    const int d{ 12 };          // 12 is an rvalue expression
    int y { d };                // d is a modificable lvalue expression
    int w { d + 1 };            // d + 1 is an rvalue expression
    w++;                        // w++ is an rvalue, while ++w is an lvalue
    std::cout << 2 + 2 << "\n"; // 2+2 is an rvalue expression

    int&& ref2r = w++;          // rvalue reference binds to rvalue (try with ++w, error!)
                                // Useful for move semantics (see next lessons)
}
```

Another example

[1] Strictly true in C++03, C++11 introduces more value categories

References can’t be reseated

Once initialized, a reference in C++ cannot be reseated, meaning it cannot be changed to reference another object.

```
#include <iostream>

int main() {
      int x = 5;
      int y = 6;
      int& ref = x;    // ref is now an alias (lvalue reference) for x
      ref = y;         // does NOT change what is ref referencing to,
                       // but it assigns the value of y to x
      std::cout << x << '\n'; // print 6
      return 0;
}
```

WARN: When an object being referenced is destroyed before its reference, the reference is left referencing an object that no longer exists. Such a reference is called a **dangling reference**. Accessing a dangling reference leads to undefined behaviour.

Reference implicit conversions

A reference to `const` can even bind to values of a different type, so long as those values can be *implicitly converted* to the reference type:

```
#include <iostream>

int main() {
	char c { 'a' };
	const int& r2 { c };
	std::cout << r2 << '\n'; // prints 97 (since r2 is a reference to int)
	return 0;
}
```

If you try to bind a `const` lvalue reference to a value of a different type, the compiler will create a temporary object of the same type as the reference, initialize it using the value, and then bind the reference to the temporary.

Reference to const

We saw that lvalue reference can only bind to a *modifiable* lvalue

To bind to a `const` lvalue, we need to define the reference as such:

```
#include <iostream>

int main() {
      const int x = 5;
      const int& refx = x;
      refx = 6; // error: we can not modify an object through a const reference
}
```

Recap on lvalue references

Lvalue references to `const` can bind to:

- modifiable lvalues — `int y = 6; int& ref {y};`
- non-modifiable lvalues — `char c { 'a' }; const int& ref { c };`
- rvalues — `int& ref = y++;`

This makes them a much **more flexible type of reference**.

Reference vs pointer

Generally: use references when you can, and pointers when you have to

References are usually preferred over pointers whenever you don't need "reseating". The exception is where a function's parameter or return value needs a "sentinel" reference — a reference that does not refer to an object (`nullptr`)

Exercise

Complete the code and answer the questions (Q): https://godbolt.org/z/ehfq3zo7Y

If the solution is working, you should see an output in the right lower part of Godbolt, after `Program stdout`

Solution: https://godbolt.org/z/rxnKW81e7

Type conversion

C++ allows converting values
from one fundamental type to another

Implicit conversion

```
#include <iostream>

int main() {
	int x = 10;
	float y = 5.5;
	char ch = ‘a’;

	x = x + ch;			// y implicitly converted to int. ASCII value of 'a' is 97
	float z = x + y;		// x implicitly converted to float (promotion)
	x = x + y;			// y implicitly converted to int, the fractional component is dropped

	double division = 4.0 / 3; // int value 3 implicitly converted to type double
}
```

C++ defines a collection of conversion rules.
The **standard conversions** specify how various fundamental types (and certain compound types, including arrays, references, pointers, and enumerations) convert to other types within that same group.

Explicit conversion

Cast	Description
`static_cast`	Performs compile-time type conversions between related types
`dynamic_cast`	Performs runtime type conversions on pointers or references in a polymorphic hierarchy (inheritance)
`const_cast`	Adds or removes const
`reinterpret_cast`	Reinterprets the bit-level representation of one type as if it were another type
C-style casts	Performs some combination of static_cast, const_cast, or reinterpret_cast

Explicit conversion: static

`static_cast<`*`dest_type`*`>  expression`

It is a *compile-time cast*. It converts between types, such as `int` to `float`, or pointer to `void*`

```
#include <iostream>

int main() {
      int x = 10;
      char y = ‘a’;
      y = static_cast<int> (y); // y explicitly converted to int. ASCII value of 'a' is 97

      x = x + y;
}
```

Best practice: favor `static_cast` when you need to convert a value from one type to another type.

Explicit conversion: dynamic

`dynamic_cast<`*`dest_type`*`>  expression`

It is a *run-time cast*.

```
#include <iostream>

// Base class declaration
class Base {
    void print(){
        std::cout << "Base" << std::endl;
    }
};

// Derived Class 1 declaration
class Derived1 : public Base {
    void print(){
        std::cout << "Derived1" << std::endl;
    }
};

int main() {
    Derived1 d1;
    Base* bp = dynamic_cast<Base*>(&d1);
    Derived1* dp1 = dynamic_cast<Derived1*>(bp);
}
```

Explicit conversion: const

`const_cast<`*`dest_type`*`>  expression`

`const_cast` is used to cast away the constness of variables.
It can be used to pass const data to a function that does not accept const arguments

To use only when needed

```
#include <iostream>

int sum(int* ptr) {
      return (*ptr + 10);
}

int main() {
      const int val = 5;
      const int *ptr = &val;
      int *ptr1 = const_cast<int *>(ptr);
      std::cout << sum(ptr1);
}
```

Typedef and alias

keywords to create an alias for an existing data type

Typedefs and alias

```
using Distance = double; // Distance is now an alias for double type
typedef double Distance; // the same alias created using typedef keyword

Distance milesToDestination = 3.4; // defines a variable using the alias created as type
```

Best practice: prefer **`using`** over **`typedef`**

When is it useful to use alias?

Because **`char`**, **`short`**, **`int`**, and **`long`** give no indication of their size, it is fairly common for cross-platform programs to use type aliases to define aliases that include the type's size in bits

```
#ifdef INT_2_BYTES
using int8_t = char;
using int16_t = int;
using int32_t = long;
#else
using int8_t = char;
using int16_t = short;
using int32_t = int;
#endif
```

Type deduction

Use the auto keyword to deduce the type of an object

auto (C++11)

The keyword `auto` instructs the compiler to deduce the type of an object from the object's initializer

```
auto i = 0;           // int
auto u = 0U;          // unsigned int
auto p = &i;          // int*
auto d = 1.;          // double
auto c = 'a';         // char
auto s = "a";         // char const*
```

```
int add(int x, int y) {
     return  x + y;
}
int main() {
    auto sum = add(5, 6);
}
```

auto (C++11)

`auto` never deduces a reference, one needs to add `&` explicitly

```
int v;
auto v1 = v;      // int – v1 is a copy of v
auto& v2 = v;     // int& – v2 is an alias of v
auto v3 = v2;     // int – v2 is a copy of v
```

`auto` makes a *mutable* copy, hence it drops the `const`

```
int main() {
  const int x = 5;
  auto y1 = x;          // y1 is a mutable copy of x (const is dropped!)
  auto const y2 = x;    // y2 is non-mutable copy of x
  auto& y3 =x;          // y3 is a non-mutable alias of x (!!)
}
```

auto (C++11): use and abuse

When to **use** it:

- when it helps readability, e.g., by decluttering a template definition

When to **avoid** it:

- when it reduces the readability of the code by obscuring the type (for the compiler is trivial, for the programmer it is not)
  - `for (const auto& user : users) { // smt } // what is user?`
- generally, when intent is not explicit

Exercises

What type of conversion happens in each of the following cases?

https://godbolt.org/z/9oEbfGz4j

Control flows

if/else statements

```
if (condition)
    true_statement;
else
    false_statement;
```

```
#include <iostream>
int main(){
    std::cout << "Enter a number: ";
    int x{};
    std::cin >> x;
    if (x > 10)
        std::cout << x << " is greater than 10\n";
    else
        std::cout << x << " is not greater than 10\n";
}
```

Conditional loops

while loops

```
while (condition)
    statement;
```

```
int main() {
    int count{1};
    while (count <= 10) {
        ++count;
    }
}
```

Do-while loops

```
do
    statement; // can be a single statement or a compound statement
while (condition);
```

for loops

```
for (init-statement; condition; end-expression)
   statement;
```

```
#include <iostream>

int main(){
    for (int i{ 1 }; i <= 10; i++)
        std::cout << i << ' ';
}
```

Other control flows

- `switch`: multiple conditional branches
- `goto`: jump to a specific line
- `break`: exit from loops
- `continue`: skip an iteration inside loops

Functions

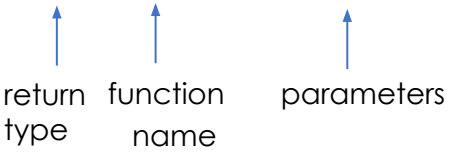
Functions

- A function is a block of code that it is executed when called
- Reuse: functions are used to perform certain operations and can be called multiple times
- You can pass variables/objects (a.k.a. parameters) to a function
- Function can be internal (user-defined) and external
- External functions are usually grouped into specialized libraries (e.g `iostream`, `stdlib`, `math`,etc..)

Functions

Function declaration

```
int sum(int x, int y);
```



Function definition

```
int sum(int x, int y){
    int z = x + y;
    return z;
}
```

Body function

- Function declaration is optional, however is mandatory when multiple files use the same function
- Function **must** be defined/declared before its call

Functions parameters and arguments

Parameters are specified after the function declaration, inside the brackets. These are comma-separated and have no limit in number

```
int sum(int x, int y);
```

Arguments are the actual values which are passed by the function call

```
int sum(int x, int y) {
  int z = x + y;
  return z;
}

int main() {
    int x = 4;
    int y = 6;
    sum(x, y);
}
```

Default parameters

If function is called with no arguments, the default arguments will be used.

```
double gaussian(double x, double mu=0.0, double sigma=1.0) {
    double pi2 = 2.0 * acos(-1.0);
    double m = x - mu;
    return exp(-m*m/(2.0*sigma*sigma)) / (sigma*sqrt(pi2));
}

int main() {
  std::cout << gaussian(3.5) << std::endl;             // x=3.5, mu=0,   sigma=1
  std::cout << gaussian(3.5,0.5) << std::endl;         // x=3.5, mu=0.5, sigma=1
  std::cout << gaussian(3.5,0.5,1.2) << std::endl;     // x=3.5, mu=0.5, sigma=1.2
  return 0;
}
```

Functions arguments

How we can pass arguments?

Call by Value
This method **copies the actual value** of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.

Call by Pointers (C)
This method **copies the address** of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

Call by Reference (C++)
This method **copies the reference** of an argument into the formal parameter. Inside the function, the reference is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

Call by Pointer

```
#include <iostream>
#include <math.h>
const double PI = 3.14159265359;

void calcArea(double *radius, double *area){
     *area = PI * pow(*radius, 2);
     std::cout << "The result is: " << *area << std::endl;
}

int main(){
     double radius;
     double area;
     std::cout << "Enter the radius of the circle: " << std::endl;
     std::cin >> radius;

     double *ptr = &radius;
     calcArea(ptr, &area); // use address-of operator (&) to get pointer holding
address of area
}
```

Call by Reference

```
#include <iostream>
#include <math.h>
const double PI = 3.14159265359;

void calcArea(double &radius, double &area){
     area = PI * pow(radius, 2);
     std::cout << "The result is: " << area <<std::endl;
}

int main(){
     double radius;
     double area;
     std::cout << "Enter the radius of the circle: " << std::endl;
     std::cin >> radius;

     calcArea(radius, area);
}
```

Functions

	Pass by Pointers	**Pass by reference**
Passing Arguments	We pass the arguments *addresses* in the function call	We pass the arguments in the function call
Accessing Values	The value of the arguments is accessed via the *dereferencing operator* *	The reference name can be used to implicitly reference a value
Reassignment	Passed parameters *can* be moved/reassigned to a different memory location	Parameters *can't* be moved/reassigned to a different memory address
Allowed Values	Pointers can contain a `nullptr` value	References *cannot* contain a `nullptr` value

Value-returning function

When we declare a function we need to specify the type of return of the function

```
double calcArea(double &radius){
    double area = PI * pow(radius, 2);
    return area;
```

A value-returning function that does not return a value will produce undefined behavior

Void functions are used when it is not required to return a value to the caller

```
void printRadiusValue(double &radius){
    std::cout << radius << std::endl;
}
```

no return statement

Functions overloading

Function defined with the *same name* but with parameters that can differ in names and types

```
// Greatest Common Divisor for int
int gcd(int a, int b) {
      a = abs(a); b = abs(b);
      if (a == 0)      return b;
      if (b == 0)      return a;
      do {
            int t = a % b;
            a = b;  b = t;
      } while (b != 0);
      return a;
}

// Least Common Multiple for int
int lcm(int a, int b)
{
    if (a == 0 || b == 0) return 0;
    return a*(b/gcd(a,b));
}
```

```
// Greatest Common Divisor for long
long gcd(long int a, long int b) {
      a = labs(a); b = labs(b);
      if (a == 0)      return b;
      if (b == 0)      return a;
      do {
            long int t = a % b;
            a = b;  b = t;
      } while (b != 0);
      return a;
}

// Least Common Multiple for long
long lcm(long int a, long int b)
{
    if (a == 0 || b == 0) return 0;
    return a*(b/gcd(a,b));
}
```

Functions overloading

```
void test_number_theory(int a, int b, long c, long d,
                        short s, unsigned u) {

cout << " gcd= " << gcd(a,b) << endl;    // calls gcd(int, int)
cout << " gcd= " << gcd(c,d) << endl;    // calls gcd(long, long)
cout << " gcd= " << gcd(a,d) << endl;    // error: gcd(int,int) or // gcd(long, long) ???
cout << " gcd= " << gcd(c,0) << endl;    // error: gcd(int,int) or // gcd(long, long) ???
cout << " gcd= " << gcd(c,0.0) << endl; // conversion: calls // gcd(long, long)
cout << " gcd= " << gcd(a,s) << endl;    // promotion: calls gcd(int, int)
cout << " gcd= " << gcd(c,u) << endl;    // conversion: calls // gcd(long, long)

}
```

Exercises

2. Write a function named `sort2` which allows the caller to pass 2 int variables as arguments. When the function returns, the first argument should hold the lesser of the two values, and the second argument should hold the greater of the two values.

Solution: https://godbolt.org/z/dezYK1Kbc

2 bis. Using functions overloading write the same function of ex. 2 in order to pass 2 double variables.

Exercises

3. Write a single-file program (named `main.cpp`) that reads two separate integers from the user, adds them together, and then outputs the answer. The program should use three functions:

- A function named "readNumber" should be used to get (and return) a single integer from the user.
- A function named "writeAnswer" should be used to output the answer. This function should take a single parameter and have no return value.
- A `main()` function should be used to glue the above functions together.

Solution: https://godbolt.org/z/Tv3Gbesh1

4. Write the following program: the user is asked to enter 2 floating point numbers (use doubles). The user is then asked to enter one of the following mathematical symbols: +, -, *, or /. The program computes the answer on the two numbers the user entered and prints the results. If the user enters an invalid symbol, the program should print nothing.

Solution: https://godbolt.org/z/ocsPbnf7W

Exercises

5. Upgrade the following program using type aliases: https://godbolt.org/z/Wb1h8fGn6

Solution: https://godbolt.org/z/dofWjYsvv

Strings

Intermezzo: C-style arrays

How to define a C-style array, a data container that handles variables of the same type

```
int main(){
    int a[5];
    int b[3] = {10, 20, 30}; // note: this is copy-initialization!
    int c[] = {1, 2, 3, 4, 5}; // implicitly: size 5

    int d[]{1, 2, 3};
}
```

String

C-style string

```c
#include <iostream>

int main() {
	char str[] = "worddd";
	std::cout << str << "\n";
}
```

C++-style string

```cpp
#include <iostream>
#include <string>     // relevant std library

int main() {
	std::cout << "Enter your full name: ";
	std::string name;
	std::cin >> name;
	std::cout << "Your name is " << name << std::endl;
}
```

string_view (C++17)

`std::string_view` provides **read-only access** to an existing string.
Hence, it avoids expensive copies

```
#include <iostream>
#include <string>
#include <string_view>

// C++17 // std provides read-only access to whatever argument is passed in
void printSV(std::string_view str){
	std::cout << str << "\n";
}

int main() {
	std::string s{"Hello, world!"};
	std::string_view sv{s}; // now a std::string_view
	printSV(sv);
}
```

WARN: `string_view` can be used as return value of a function. However, since local variables are destroyed at the end of the function scope, returning a `std::string_view` to a local variable will result in the returned `std::string_view` being invalid

string_view (C++17)

Modifying a `std::string` invalidates all views into that `std::string`.

```
#include <iostream>
#include <string>
#include <string_view>

int main() {
      std::string s = "Hello, world!";
      std::string_view sv = s;    // sv is now viewing s
      s = "Hello, universe!";     // modifies s, which invalidates sv (s is still valid)
      std::cout << sv << '\n';    // undefined behavior (try in godbolt!)
      return 0;
}
```

The core idea: `string_view` is a **viewer**, `string` an **owner** (generally meaning that it is the sole responsible for the data

Exercises

6.Write a program that asks for the name and age of two people, then prints which person is older.

Here is the sample output from one run of the program:

```
Enter the name of person #1: John Bacon
Enter the age of John Bacon: 37
Enter the name of person #2: David Jenkins
Enter the age of David Jenkins: 44
David Jenkins (age 44) is older than John Bacon (age 37).
```

Solution: https://godbolt.org/z/xM3jrfzYa

const, constexpr and consteval

Keywords to evaluate expressions at compile time or runtime

const

Keyword: **const**

- The expression initializing the variable may be evaluated at compile or run time
- A constant variable *cannot* be changed after initialization

```
int main() {
     int x = 4 ; // x is a non-constant variable
     x = 5; // change value of x to 5

     // Try example using const
     const int y = 4;   // x is a constant value now (compile time)
     y = 5;             // error: const variables can not be changed
     const int y;       // error: const variables must be initialized
     const int y = x;   // ok if initialized from non-const var (eval at runtime)
     return 0;
}
```

constexpr

Keyword: **constexpr**

- The expression initializing the variable **must** be known at compile time
- The resulting object, can be used in a constant expression

```
int main() {
	constexpr double gravity = 9.8;	// initialized with constant expression
	constexpr int sum =  4 + 5;		// same as above
	constexpr int something = sum;	// sum is guaranteed to evaluate to a constant expression
}
```

```
#include <iostream>

int main(){
	std::cout << "Enter your age: ";
	int age;
	std::cin >> age;
	constexpr int myAge { age };		// COMPILE ERROR: not a constant expression (only known at
runtime)
}
```

constexpr

The idea is that **constexpr** *are eligible* (i.e. not guaranteed) for compile-time evaluation, meaning they are more likely to be optimized at compile-time.

```
#include <iostream>

constexpr int greater(int x, int y) { // this is a constexpr function
     return (x > y ? x : y);
}

int main() {
     // case 1: always evaluated at compile-time
     constexpr int g { greater(5, 6)}; // because arguments are constants

     // case 2: may be evaluated at either runtime or compile-time
     greater(5, 6)

     // case 3: always evaluated at runtime
     std::cin >> x;
     std::cout << greater(x, 6) << " is greater!\n";
}
```

consteval (C++20)

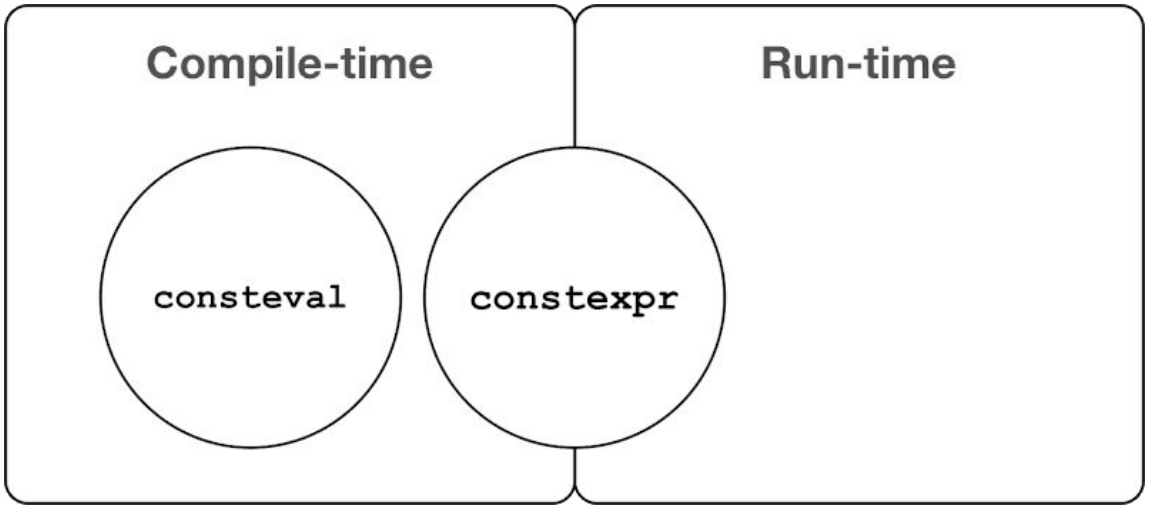
How to be sure that constexpr is evaluated at compile-time?

C++20 introduces the keyword **`consteval`**, which is used to indicate that a function *must* evaluate at compile-time.

```
#include <iostream>

consteval int greater(int x, int y) { // function is now consteval
     return (x > y ? x : y);
}

int main() {
     constexpr int g =  greater(5, 6); // ok: will evaluate at compile-time
     std::cout << g << '\n’;
}
```

The downside of **`consteval`** functions is that such functions can't evaluate at runtime, making them less flexible than **`constexpr`** functions, which can do either

constexpr vs consteval (C++20)



constexpr and consteval (C++20)

Within a **constexpr** or **consteval** function, we can use local variables that are not **constexpr**, and the value of these variables can be changed

```
#include <iostream>

consteval int doSomething(int x, int y) {
      x = x + 2;            // we can modify the value of non-const function parameters
      int sum =  x + y;     // we can instantiate non-const local variables

      if (x > y) sum = sum - 1; // and then modify their values

     return sum;
}

int main() {
      constexpr int g  = doSomething(5, 6) ;
      std::cout << g << "\n";
      return 0;
}
```

output: 12

Exception handling

Exception handling: `throw` and `if` statements

Exceptions are used to separate application logic from error management

The basic way of *throwing* or *raising* an exception is through the **`throw`** keyword

The most simple approach is to handle exceptions using `if` blocks:

```
#include <iostream>

double divide(double num, double den) {
    if(den == 0.0) throw "Division by 0!"; // raise this error and
print it in the stderr
    return num/den;
}
```

In C++ anything can be thrown (like the string in the example)

Exception handling: `try` and `catch` blocks

The **`try`** block acts as an *observer*, looking for any exceptions that are thrown by any of the statements within the try block.

The **`catch`** keyword is used to define a block of code that handles exceptions for a single data type.

```
#include <cmath>          // for sqrt() function
#include <iostream>

int main() {
	std::cout << "Enter a number: ";
	double x;
	std::cin >> x;

	try {
		// If the user entered a negative number, this is an error condition
		if (x < 0.0) throw "Can not take sqrt of negative number";	// explicit throw
		// Otherwise
		std::cout << "The sqrt of " << x << " is " << std::sqrt(x) << '\n';
	}
	// catch exceptions of type const char* (NOTE: no implicit conversions!)
	catch (const char* exception){
		std::cerr << "Error: " << exception << '\n';
	}
}
```

Exception handling: `try` and `catch` blocks

- **Exceptions propagates** up in the stack of functions calls
- If no suitable catch clause is found, the program terminates abnormally
  - We can define which exception we want to handle. We can use ***catch-all block***, written as `catch(...)`, that can be used to catch all types of exceptions

Exception handling: asserts

Asserts **evaluate at runtime** an expression.

If this expression is *true*, the assertion statement does nothing. If the conditional expression evaluates to *false*, an *error* message is displayed and the program is terminated (via `std::abort`).

Exception handling: assert

The *preprocessor macro*: **assert**

```
#include <cassert>            // for assert()
#include <cmath>              // for std::sqrt
#include <iostream>

double calculateTime (double initialHeight, double gravity) {
      assert(gravity > 0.0);  // The object won't reach the ground unless there is positive gravity
      if (initialHeight <= 0.0) {
            return 0.0;
      }
      return std::sqrt((2.0 * initialHeight) / gravity);
}

int main() {
      std::cout << calculateTime(100.0, -9.8) << " second(s)\n";
      return 0;
}
```

```
src/main.cpp:5: double calculateTime(double,
 double): Assertion 'gravity > 0.0' failed.
```

Asserts are used to detect errors while developing and debugging. Hence, they can be deactivated in production codes using the `-DNDEBUG` flag when compiling

Exception handling: `static_assert`

A **`static_assert`** is an assertion that is checked at compile-time. Hence, the expressions evaluated have to be **`constexpr`**.

A failing **`static_assert`** causes a compile error

Unlike **`assert`**, which is declared in the **`<cassert>`** header, **`static_assert`** is a keyword of the C++ language.

```
static_assert(condition, diagnostic_message)
```

```
static_assert(sizeof(long) == 8, "long must be 8 bytes");
static_assert(sizeof(int) >= 4, "int must be at least 4 bytes");

int main() {
      return 0;
}
```

Best practice: prefer compile-time checking to run-time checking

Exercises

7. Write a code which verifies that an inserted number is included between a minimum and a maximum value. Use two exceptions to verify if this is the case.

Solution: https://godbolt.org/z/Wz4x1fEEq

8. A prime number is a natural number greater than 1 that is evenly divisible (with no remainder) only by 1 and itself. Complete the following program by writing the `isPrime()` function using a for-loop. When successful, the program will print "Success!".

Solution: https://godbolt.org/z/j7E4eMMM4

Namespace

Namespaces

Namespaces **groups** variable, functions, and classes **under the same name**, avoiding name collision that occurs when our code base includes multiple libraries.

To define it use the namespace keyword:

```
namespace namespace_name
```

To use everything within the namespace use the scope operator `::` or a `using` declaration:

```
#include <vector>
#include <string>

std::vector<std::string> vec;
```

```
#include <vector>
#include <string>

using namespace std;
vector<string> vec;
```

Not recommended

Namespaces

```
#include <iostream>
// definition of a namespace for basic geometry formulas
namespace BasicGeometryFormulas {
	double circle(double radius){
		double pi = 3.1415;
		double area = pi * (radius * radius);
		return area;
	};
	double square(double side){
		double area = side * side;
		return area;
	}
}

int main() {
	double areacircle = BasicGeometryFormulas::circle(3.4);
	double areasquare = BasicGeometryFormulas::square(5.2);
	std::cout << areacircle << std::endl;
	std::cout << areasquare << std::endl;
	return 0;
}
```

Namespaces

A namespace can be declared in a header file and then used in a `.cpp` file including the header.

Example with a local header:

In *namespace.hpp*

```
namespace BasicGeometryFormulas {
	double circle(double radius){
		double pi = 3.1415;
		double area = pi * (radius * radius);
		return area;
	};
	double square(double side){
		double area = side * side;
		return area;
	}
}
```

In *namespace.cpp*

```
#include "namespace.hpp"
#include <iostream>

int main() {
	double areacircle =
BasicGeometryFormulas::circle(3.4);
	double areasquare =
BasicGeometryFormulas::square(5.2);
	std::cout << areacircle << std::endl;
	std::cout << areasquare << std::endl;
	return 0;
}
```

Nested namespaces

Namespaces can be nested is and its resolution of namespace variables is hierarchical.

In *namespace.hpp*

```
#include <iostream>
namespace BasicGeometryFormulas {
	namespace computearea{  // nested namespace
		double circle(double radius){
			double pi = 3.1415;
			double area = pi * (radius * radius);
			return area;
		}
	}
}
```

In *namespace.cpp*

```
#include "namespace.hpp"
int main() {
	double areacircle = BasicGeometryFormulas::computearea::circle(3.4);
}
```

Namespace alias

In C++, you can use an alias name for your namespace name, for ease of use.

In *namespace.cpp*

```
#include "namespace.hpp"

namespace area = BasicGeometryFormulas::computearea;

int main() {
      double areacircle = area::circle(3.4);
      double areasquare = area::square(5.2);
      std::cout << areacircle << std::endl;
      std::cout << areasquare << std::endl;
      return 0;
}
```

Final exercise!

9. Write a short program to simulate a ball being dropped off of a tower.

- To start, the user should be asked for the height of the tower in meters.
- Assume normal gravity (`9.81` m/s2), and that the ball has no initial velocity (the ball is not moving at the start).
- Have the program output the height of the ball above the ground after 0, 1, 2, … seconds until it reaches the ground.
- The ball should not go under the ground (height 0).
- Use a function to calculate the height of the ball after x seconds. The function can calculate how far the ball has fallen after `x` seconds using the following formula:

```
distance fallen = gravity_constant * x_seconds^2 / 2
```

Expected output:

```
Enter the height of the tower in meters: 100
At 0 seconds, the ball is at height: 100 meters
At 1 seconds, the ball is at height: 95.1 meters
At 2 seconds, the ball is at height: 80.4 meters
At 3 seconds, the ball is at height: 55.9 meters
At 4 seconds, the ball is at height: 21.6 meters
At 5 seconds, the ball is on the ground.
```

Solution #1: https://godbolt.org/z/YYM8GsWd3 **Solution #2**: https://godbolt.org/z/rGchhMzTe