Data Structures I

Introduction to Types in C/C++ and Algorithm Analysis

Igor Machado Coelho

13/09/2020 - 13/08/2026

1. Introduction to Types in C/C++ and Algorithm Analysis
2. Part 1: Primitive Types in C/C++
3. Part 2 - Practice: Programs in C/C++
4. Part 3: Introduction to Algorithm Analysis and Big-O Notation
5. Part 4: Trivial and Generic Aggregate Types
6. Part 4: Pointers, Routines and Dynamic Allocation in C/C++
7. Part 5: Abstract Types and Concepts
8. Part 6: Smart Pointers and `std::move` in C++
9. Part 7: Coroutines (advanced topic)
10. Part 8: References in C++ (advanced topic)
11. Part 9: Experimental and advanced libraries in C++
12. Modularization and Testing (to be revised!!)

13 Acknowledgements

Section 1

Introduction to Types in C/C++ and Algorithm Analysis

NOTE ON TRANSLATION

This document was AI Translated to English. For reference, see the original one, in Portuguese.

Prerequisites

The requirements for this class are knowledge of:

- Introduction/Fundamentals of Programming (in some programming language)
- **Interest** in learning C/C++
- Familiarity with using and installing programs via the command line (in your preferred operating system)
- Familiarity with using compilers/IDEs or using online programming tools

Programming Environment

Why program in C/C++?

- A consolidated, efficient and **low-level** language (close to the *hardware*)
- **Lots** of online material and books, etc.
- **Many** tools and ready-made code, works on any operating system

How to program in C/C++?

- Install a **modern** compiler for **C23/C++23**. General recommendation: **clang 22**
  - Download link: https://github.com/llvm/llvm-project/releases
- For Linux, you can use **GCC 15** or **clang**
- For Mac, use **clang**
- For Windows, **avoid** both native **MSVC** and ***clang-cl*** *for Windows*!
  - You **must** install **WSL2** on Windows and use the Linux support recommended above.
- If you are going to use an online compiler, use https://godbolt.org

Section 2

Part 1: Primitive Types in C/C++

Type Concepts in C/C++

Understanding programming logic is the most important skill for a programmer! With it, you can easily switch programming languages, knowing only a few basic commands.

The first concept to be reviewed is that of a variable. A variable consists of a valid identifier (the same as for other popular languages such as Python) and stores some kind of data in the computer's memory.

The C/C++ language is **strongly typed**, so the programmer must explicitly say which data type they wish to store in each variable, or let **auto** deduce it automatically.

```
int    x = 5;    // stores the integer 5 in variable x
char   y = 'A';  // stores the character 'A' in variable y
float  k = 3.7f; // stores the real number 3.7 in variable z
double z = 3.7;  // stores the real number 3.7 in variable z
bool v = true;   // stores the boolean true in variable v
auto b = 'B';    // type deduction with 'auto'... which type?
auto s = "abcd"; // character string, type still to be seen
```

Variable Types

Question/Answer: Be careful with types. What are the values stored in the variables below (C++23 and C23)?

```
int    x1 = 5;         // => 5
int    x2 = x1 + 10;   // => 15
int    x3 = x2 / 2;    // => 7
double x4 = x2 / 2;    // => 7.0
double x5 = x2 / 2.0;  // => 7.5
auto   x6 = 15;        // => 15
auto   x7 = x2 / 2;    // => 7
auto   x8 = x2 / 2.0;  // => 7.5
```

Check these variable operations by writing to standard output (the computer screen).

C/C++ Concepts (defined types)

Primitive types in C/C++ have a defined size, so it is good practice to use fixed sizes.

Prefer direct initialization with braces { }, rather than indirect initialization by assignment (operator=).

```
int     x0 {-1};  // direct initialization!
int64_t x1 = 10;  // long (or long long)
int32_t x2 = 20;  // int
int16_t x3 = 30;  // short
int8_t  x4 = 40;  // signed char
uint8_t x5 = 50;  // unsigned char
```

Introduction to Routines: Functions

Modularizing programs is very important, especially when code snippets are repeated many times.

In these cases, it is common to create routines, such as *functions and procedures*, which in turn may receive parameters.

Let us take as an example the function square, which returns the value passed raised to the square.

```
// function that returns an 'int',
//   with an integer parameter 'p'
int square (int p) {
   return p*p;
}
// which type?
auto x = square(5);
```

```
// function that returns an 'int',
//   with an integer parameter 'p'
auto square (int p) -> int {
   return p*p;
}
// which type?
auto x = square(5);
```

Important: the type deduction after the arrow -> is done automatically.

Standard Output Printing

In C, the `printf` command is typically used, but due to countless security flaws, using a safer alternative is recommended. Thus, in C++, to print to standard output we will use the `std::print` command.

C++23 officially brings `std::print` and `std::println` as part of the `std` standard library module. To use it, you just need `import std;`.

```
import std;

int main() {
  std::println("Hello World!");
  return 0;
}

// online GCC: https://godbolt.org/z/j3W938PP6
```

Question: What is the return value of the `main` function? What does it mean?

Standard Output Printing

To print to standard output we will use the `std::print` command. Note: it is possible to avoid the `std::` prefix with a `using namespace std;`.

Question: how can we mix a text (also called a character string, or string) with the content of variables? **Answer:** through the substitution pattern `{}`.

```
import std;
int main() {
  int x1 = 7;
  std::println("x1 is {}", x1);  /*  x1 is 7  */
  double x6 = x1 / 2.0;
  std::println("half of {} is {}", x1, x6);  // half of 7 is 3.5
  char b = 'L';
  std::println("this is a {}etter", b);  // this is a Letter
  std::print("Hello world! \n");   // Hello world! (line break)
  return 0;
}
```

Conditionals

Problem: given `x` and `y`, print the larger value.

Conditionals can be done through the if or if else commands.

```
int x = 15;
int y = 12;
if (x > y)
   println("x is greater than y");
else
   println("x is less than or equal to y");
```

Question: What is the result of the expression `if(x = y)`? And `if(x == y)`?

Loops (Part 1/2)

Loops can be done through while or for commands. A for command is divided into three parts: initialization, continuation condition and increment.

```
for (int i=0; i < 10; i++) {
   std::println("i : {}", i);
}
```

```
int j=0;
while (j < 10) {
   std::println("j : {}", j);
   j++;
}
```

Question: What is printed in both loops?

The void and std::monostate Types

We have seen some types with a larger number of values, for example, **int** and **char**. **int** takes up 4 bytes and is able to support up to 2^{32} distinct values (approx. -2 billion to +2 billion), while **char** supports up to 256 values, taking up only 1 byte.

In some cases, it is also useful to use types with a smaller number of values, such as **std::monostate**, which takes up 1 byte and has only a single value, and finally the **void** type, which is an *incomplete type* and *does not represent any value*. Since **void** does not represent values, it is not possible to create variables of type **void**!

```
int              i = 10;     // one out of 4 billion valid values
bool             b = true;   // one out of two valid values
// void          v;          // compilation error: void has no value
std::monostate   m;          // only one possible value
auto             n = m;      // monostate can be copied
```

Important: the `std::monostate` type only exists in C++, since in C there was never a need to define this type, due to the lack of *generic types* (we will see this later).

Introduction to Routines: Procedures

When no value is returned (in a procedure), we use the **void** type. Procedures are useful even when no value is returned. **Example**: (from a to b):

```
import std;

void print_range (int a, int b) {
   for (int i=a; i<b; i++)
      std::println("{}", i);
}

int main() {
   print_range(2, 5);
   return 0;
}
```

```
import std;

auto print_range (int a, int b) -> voic
   for (int i=a; i<b; i++)
      std::println("{}", i);
}

auto main() -> int {
   print_range(2, 5);
   return 0;
}
```

Question: Why does the `print_range` routine not need to return anything? What happens as a *side effect* when calling `print_range(2,5)`?

Composite Types: Arrays

Besides the primitive types presented earlier (int, float, char, ...), the C/C++ language allows us to create composite types.

Task: study the remaining primitive types such as double and long long, as well as the modifiers unsigned, signed, short and long.

Composite types can be homogeneous aggregates (vectors/arrays) or heterogeneous aggregates (structs, ...).

Composite Types: Arrays

Besides the primitive types presented earlier (int, float, char, ...), the C/C++ language allows us to create composite types.

Task: study the remaining primitive types such as double and long long, as well as the modifiers unsigned, signed, short and long.

Composite types can be homogeneous aggregates (vectors/arrays) or heterogeneous aggregates (structs, ...).

```
int v[10];  // creates an array with 10 integers  (40 bytes)
v[0] = 3;   // assigns the value 3 to the first position
v[9] = 5;   // assigns the value 5 to the last position
```

Example of an array v, from 0 to 9, from left to right:

```
v: | 3 |   |   |   |   |   |   |   |   | 5 |
     0   1   2   3   4   5   6   7   8   9
```

Flow control with `break` and `continue`

Flow control in loops can be done with `break` and `continue`. `break` ends the execution of the loop and `continue` restarts the loop.

Problem: Given an array B, find the first/last negative value, or print -1 if it does not exist.

```
int B[] = {4, -3, 5, -7, 8};
int z = -1;
for (int i=0; i < 5; i++)
   if (B[i] < 0) {
      z = i;
      break;
   }
println("z={}", z);
// z==1
```

```
int B[] = {4, -3, 5, -7, 8};
int z = -1;
for (int i=0; i < 5; i++){
   if (B[i] >= 0)
      continue;
   z = i;
}
println("z={}", z);
// z==3
```

Unconditional jumps with goto (advanced topic)

Unconditional jumps in the code can be done with `goto label;` and `label:`. A usual application is the "multiple break" of loops. Avoid using `goto` as much as possible and, whenever possible, prefer structured alternatives such as `for`, `while`, `if`, `else`, `break`, etc.

Count how many prints are executed (variable `z`):

```
  int z = 0;
  for (int i = 0; i < 10; i++) {
    if (i < 5) continue; int j = i;
    while (j < 10) {
      if (i > 6) goto end;
      println("z={} i={} j={}", z, i, j); z++; j++;
    }
  }
end:
  println("final z={}", z);
  // z==9: i=5 j=5..9 [5 steps]; i=6 j=6..9 [4 steps]
```

Section 3

Part 2 - Practice: Programs in C/C++

Programming Environment (details)

Examples will be given based on the GNU/Linux system and GCC compilers, but there are equivalent tools for Windows and other operating systems. The IDE Visual Studio Code supports the C++ language both for Linux and for Windows, requiring CMake 4.0 with Ninja. On Windows/WSL or Linux, install the Clang 19 compiler (on Windows, use Scoop with `scoop install main/llvm`).

It is also possible to practice directly in a web browser with online platforms: onlinegdb.com/online_c++_compiler or Godbolt (more recommended!). In this case, the student can choose the compiler for C or for the C++ language (considering the C23 and C++23 standards).

To set up an IDE, read the tutorial "Breve Introdução ao C/C++ com IDE de Desenvolvimento" (in Portuguese).

Example Program in Standard C

The GCC compiler supports C/C++, so it can compile both C and *C with C++*.

Code main.c:

```c
#include <stdio.h>

int main() {
    auto m = "World";
    printf("Hello %s!\n", m);
    return 0;
}
```

To compile manually (without CMake), run the following command with GCC 15:

```
g++ -std=c23 main.c -o example_c
```

Since C++ includes everything C has and still adds safer and simpler features, such as `import` and `print`, we will use the C++ standard in the compiler.

Example Program in C/C++ (Manual GCC)

See the example at https://godbolt.org/z/j3W938PP6:

Code main.cpp:

```
import std;

int main() {
  auto m = "World";
  std::println("Hello {}!", m);
  return 0;
}
```

To compile manually (without CMake), run the following command with GCC 15 or 16:

```
g++-15 -std=c++23 -fmodules -fsearch-include-path bits/std.cc main.cpp -o example
g++-16 -std=c++23 -fmodules --compile-std-module              main.cpp -o example
```

For larger and more complex programs (with many files), it is necessary to use some build system, such as CMake or Bazel. On the next slide, an example of a CMakeLists.txt.

Example Program in C/C++ with CMake 4.4 and GCC 16

See the example at https://godbolt.org/z/We6fdojj1 (cmake 4.4.2):

CMakeLists.txt code (you need to install CMake 4.4 and Ninja)

```
cmake_minimum_required(VERSION 4.4)

# https://github.com/Kitware/CMake/blob/master/Help/dev/experimental.rst
set(CMAKE_EXPERIMENTAL_CXX_IMPORT_STD "d0edc3af-4c50-42ea-a356-e2862fe7a444") # 4.1
# set(CMAKE_EXPERIMENTAL_CXX_IMPORT_STD "451f2fe2-a8a2-47c3-bc32-94786d8fc91b") # 4.3
set(CMAKE_CXX_MODULE_STD 1)

project(my_project VERSION 0.1.0 LANGUAGES CXX)
set(CMAKE_CXX_STANDARD 23)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_EXTENSIONS OFF)
set(CMAKE_EXPORT_COMPILE_COMMANDS ON)

add_executable(example src/main.cpp)
```

Example Program in C/C++ with CMake 4.4 and GCC 16

See the example at https://godbolt.org/z/We6fdojj1 (cmake 4.4.2):

To build, there are four commands:

```
mkdir -p build
cd build
cmake .. -GNinja
ninja
```

Another quite elegant solution is to use Bazel with the Clang compiler.

Tip for the VSCode IDE: use the clangd extension

To use a development IDE such as VSCode, it is necessary to use Clang with the clangd extension for correct visual processing of the code (it does not work properly with GCC nor with Microsoft's standard C/C++ extension).

Example Program in C/C++ with Bazel 9 and Clang

See the example in the tutorial "Local import std module on C++23 with Bazel". You only need to configure your MODULE.bazel and BUILD files (copy the `extensions.bzl` from the tutorial!).

```
# MODULE.bazel
module(name = "project")

bazel_dep(name = "rules_cc", version = "0.2.17")
std_modules = use_extension("//:extensions.bzl", "local_libcxx_extension")
use_repo(std_modules, "std_modules")

# BUILD
load("@rules_cc//cc:defs.bzl", "cc_binary")

cc_binary(
    name = "example",
    srcs = ["main.cpp"],
    deps = ["@std_modules"]
    features = ["cpp_modules"]
)
```

Example Program in C/C++ with Bazel 9 and Clang

See the example in the tutorial "Local import std module on C++23 with Bazel".

To build and run, there are two commands:

```
bazel build ...
bazel run :example
```

Remember to update your `.bazelrc` file with the correct compiler version:

```
# .bazelrc

build --repo_env=BAZEL_COMPILER=clang
build --repo_env=BAZEL_CXXOPTS=-stdlib=libc++
build --repo_env=BAZEL_LINKOPTS=-stdlib=libc++

build --repo_env=LIBCXX_MODULE_PATH=/usr/lib/llvm-21/share/libc++/v1
build --experimental_cpp_modules
build --cxxopt=-std=c++23
```

Section 4

Part 3: Introduction to Algorithm Analysis and Big-O Notation

Algorithm Analysis

- Algorithm analysis (or algorithm complexity analysis) deals with the efficiency of algorithms
  - it is also about determining the resources (time/space) needed to execute a given algorithm
- But how do we measure the efficiency of algorithms?
- Not to be confused with the field of Computational Complexity, which includes other topics on the analysis of the complexity of computational problems
- Term coined by Donald Knuth (*Analysis of Algorithms*), but historically already explored by other thinkers (next slide!)

Charles Babbage's Analytical Engine

- Charles Babbage was a computing thinker who tried to build a computer, mechanical at that time, called the *analytical engine*
  - it only became possible in the 2000s, with advances in gear production technology
  - See the Analytical Engine on Wikipedia

Charles Babbage (1864) says: > "As soon as analytic engine exists, it will necessarily guide the future course of science. Whenever any result is sought by its aid, the question will raise ? By what means of calculation can these results be arrived at by this machine in the shortest time?"

- Babbage figure see link

Efficiency Metric

- Consider the following efficiency metric: in any execution, the algorithm must respond in less than t time units.
- Example: Google's systems are notable for keeping a tolerable response time in any interaction.
- Is Google efficient? Are chat AIs efficient?

Efficiency Metric

- Consider the following efficiency metric: in any execution, the algorithm must respond in less than t time units.
- Example: Google's systems are notable for keeping a tolerable response time in any interaction.
- Is Google efficient? Are chat AIs efficient?

- Inadequate metric, since execution time varies with the hardware!
- For example, what would become of complexity analysis if the hardware were infinitely fast and with no restrictions on quantity?

Turing Machine - Alan Turing

- Alan Turing was another important thinker of computing, and even without a modern electronic machine as we have nowadays, he already raised theoretical aspects that, still today, are fundamental to the field
- Alan Turing (1947) says: > "It is convenient to have a measure of the amount of work involved in a computing process, even if it has to be a very crude one. We may count up the number of things that various times at various elementary operations are applied in the whole process."
- Turing photo see link

Counting Steps

- Let us therefore consider another metric: assuming that each instruction takes constant time, efficiency is given by the number of steps of an execution
- We can study efficiency for **worst-case**, **average**, **best-case** complexities, also **amortized analyses** or even for particular execution instances.

Random Access Machine (RAM) Model

- We therefore assume the Random-Access Machine model of computation with a single processor.
  - We adopt the Aho-Hopcroft-Ullman RAM, applied directly to C/C++: since the model has no registers, every variable resides in memory and every reference to it is an access.
- Memory hierarchy (cache, virtual memory, etc.) is not taken into account
- Instructions whose cost does not depend on the size of the data being processed count as one step; otherwise, they count as a function of that size.
- Naturally, a line of code may include several instructions, and not just one, for example, arithmetic ones, memory reads, memory writes, array index access, etc.
  - Literal constants are immediate operands and do not cost an access.
  - We consider the naive syntax-directed translation for each expression, without register allocation and without common subexpression elimination (typical optimizations of a compiler).

To be continued...

Section 5

Part 4: Trivial and Generic Aggregate Types

Aggregate Types I

C/C++ comparison: remember to use **struct** or **class public:**, otherwise it will not be recognized as an *aggregate type*, but rather as an *object*, which works in a completely different way in the C++ language.

```c
// In C (aggregate type P)
struct P {
   int  x;
   char y;
};

// declares a variable of type P
struct P p1;
// designated initializers
struct P p2 = {.x=10, .y='Y'};
```

```cpp
// In C++ (aggregate type P)
struct P {
   int  x;
   char y;
};

// declares a variable of type P
P p1;
// designated initializers
auto p2 = P{.x=10, .y='Y'};
```

Important: we will always use **struct** throughout this C/C++ course as a *trivial aggregate type*, never as a *class*.

Aggregate Types II

We return to the previous P struct example and ask ourselves, how do we access the internal variables of the aggregate P?

Just as in designated initialization, we can use the dot operator (.) to access fields of the aggregate. Example:

```
auto p1 = P{.y = 'A'};
p1.x = 20;              // assigns 20 to variable x of p1
p1.x = p1.x + 1;        // increments variable x of p1
println("{} {}", p1.x, p1.y);  // prints '21 A'
```

Example of struct p1, with p1.x and p1.y, from left to right:

```
p1:  |      21       | 'A' |
          p1.x          p1.y
```

Important: we will see later that (.) can also access *internal methods* of the aggregate type.

Memory Space and Methods in Aggregates

All variables of a program occupy a certain space in the computer's main memory. **We will assume** that the int (or float) type takes up 4 bytes, while a char takes up only 1 byte.

In the case of arrays, the space occupied in memory is multiplied by the number of elements. Let us calculate the space of the variables:

```
int   v[256];   // = 1024 bytes = 1 kibibyte = 1 KiB
char  x[1000];  // = 1000 bytes = 1 kilobyte = 1 kB
float y[5];     // = 20 bytes
```

In aggregates, we assume the space occupied as the sum of its internal variables (although in practice the size may be slightly larger, due to memory alignment).

Important: in C++, trivial aggregates *can also contain methods*.

Generic Types

C++ allows the definition of generic types, that is, types that allow some *other type* to be passed as a parameter.

Let us consider the aggregate P that carries an int and a char... how do we transform it into a generic aggregate with respect to the variable x?

```
template<typename T>
struct G {
   T x;   // which type does variable x have?
   char y;
};
// declares the generic aggregate G with type T=float or T=char
G<float> g1 = {.x = 3.14, .y  = 'Y'};
G<char>  g2 = {.x = 'A',  .y  = 'Y'};
```

Question: How much space (in bytes) does each of these variables take up?

Constant values and casts

In C/C+, we can define a value as constant, through the word `const`. A type change can be done with a *type cast*. In C++, use `static_cast<type>` instead of the C-style cast.

```
unsigned int x = 10;                        // 10
double y1 = x / -2;                         // 0
double y2 = (double)x / -2;                 // -5
double y3 = static_cast<double>(x) / -2; // -5 (C++)
const unsigned int z1 = x;                  // 10
// z1 = 20;                                 // ERROR
```

`const` can be removed through a `const_cast`, which is unsafe.

In C23 and C++23 there is `constexpr`, which unlike `const`, can never be removed or redefined (unlike macros), since it is resolved at compile time.

```
#define k1 10          // unsafe and allows redefinition
constexpr int k2 = 10; // safe, impossible to redefine
```

The std::string and std::string_view Types in the STL

The std::string type represents character strings, called *strings*. It replaces the need for char*, char[] or const char* in C.

If you need a lightweight "view" of a string, such as a substring, use std::string_view (it avoids the full copy of the string).

```
std::string s1 = "abcd";
std::string s2 = "ef";
println("length1={} length2={}", s1.length(), s2.length());
// length1=4 length2=2
s1 = s1 + s2;
std::string_view sv = s1;
std::string_view sub = sv.substr(3, 2);
println("s1={} s2={} sv={} sub={}", s1, s2, sv, sub);
// s1=abcdef s2=ef sv=abcdef sub=de
const char* cs = s1.c_str();
println("s1={} cs={}", s1, cs);
```

The std::vector Type in the STL

The popular std::vector<type> structure allows representing arrays with variable size (through the push_back method). Example:

```
int v1[10];
int v2[] = {1, 2, 3, 4};
std::vector<int> k1{};
std::vector<int> k2 = {1, 2, 3, 4};
k2.push_back(999);
//
print("v[0]={} v[3]={} size={}\n", v2[0], v2[3],
      sizeof(v2) / sizeof(v2[0]));
// v[0]=1 v[3]=4 size=4
print("k[0]={} k[4]={} size={}\n", k2[0], k2[4], k2.size());
// k[0]=1 k[4]=999 size=5
print("{}\n", std::is_aggregate<std::vector<int>>::value);
// false
```

The std::array Type in the STL

Just like native arrays, e.g. `int[]`, the aggregate `std::array<type, size>` allows representing arrays of fixed size. Example:

```
int v1[10];
int v2[] = {1, 2, 3, 4};
std::array<int, 10> a1{};
std::array<int, 4> a2 = {1, 2, 3, 4};
print("v[0]={} v[3]={} size={}\n", v2[0], v2[3],
      sizeof(v2) / sizeof(v2[0]));
// v[0]=1 v[3]=4 size=4
print("a[0]={} a[3]={} size={}\n", a2[0], a2[3], a2.size());
// a[0]=1 a[3]=4 size=4
print("{} {} {}\n", std::is_aggregate<int*>::value,
      std::is_aggregate<int[]>::value,
      std::is_aggregate<std::array<int, 4>>::value);
// false true true
```

Summary so far

So far, we have checked the following structures:

- primitive types (C)
- automatic type with **auto** (C)
- introduction to routines with **auto** return (C++)
- conditional structures and loops (C)
- arrays (C)
- aggregate types with **struct** or **class/public:** (C/C++)
- generic aggregates (C++)

Section 6

Part 4: Pointers, Routines and Dynamic Allocation in C/C++

Routines I

It is possible to return multiple elements (pair or tuple), through a *structured binding* with tuples:

```
auto double_it(int p) {
   return std::tuple{p+3, p+6.5};
}
auto [x1,x2] = double_it(10); // x1=13 x2=16.5
```

Q.: what is the return type of 'double_it'? **A:** `std::tuple<int, double>`.

Pointers I

Parameters are always copied (in C) when passed to a function or procedure. How do we pass complex types (structs and arrays with many elements) without losing time?

In these cases, the C language offers a special type called pointer. The pointer syntax simply includes an asterisk (*) after the type of the variable. An empty state is made with **nullptr** (or 0).

Examples: `int* x = nullptr; struct P* p1 = nullptr;`

A pointer simply stores **the location** (address) where a certain variable is stored in memory (basically, a number). So when a pointer is passed as a parameter, **the copy of the pointer** can be used to find the desired structure in memory.

The size of the pointer varies according to the architecture, but to address 64-bits, it takes up 8 bytes.

Pointers II

In pointers to aggregates, the access operator (`.`) is replaced by an arrow (`->`). The `&` operator takes the address of the variable:

```
struct P { int x; char y; };
// ...
P p0 = {.x = 20, .y = 'Y'};
```

Testing procedures `f` and `g`:

```
void f(P* p1) {
   println("{}", p1->x);
   p1->x = 1;
}
f(&p0);
println("{}", p0.x); // 1
```

```
void g(P p2) {
   println("{}", p2.x);
   p2.x = 1;
}
g(p0);
println("{}", p0.x); // 20
```

Dynamic Memory Allocation

Programs frequently need to allocate more memory for use, which is stored safely in a pointer to the type of the memory:

```c
// Allocates (C) the aggregate P
struct P* vp =
   malloc(1*sizeof(struct P));
// initializes the fields of P
vp->x = 10;
vp->y = 'Y';
// prints x (value 10)
printf("%d\n", vp->x);
// discards the memory
free(vp);
```

```cpp
// Allocates (C++) the aggregate P
auto vp = new P{
                  .x = 10,
                  .y = 'Y'
                };

// prints x (value 10)
println("{}", vp->x);
// discards the memory
delete vp;
```

Routines II

The type of a function is basically a pointer (address) of the location of this function in the computer's memory. For example:

```
// type: int(*)(int)
int square(int p) {
   return p*p;
}
```

```
// type: float(*)(int)
auto fsquare(int p) -> float {
   return p*p;
}
```

This fact can be useful to receive functions as parameters, as well as to store anonymous functions (*lambdas*):

```
// stores a lambda in the function pointer 'sq'
int(*sq)(int) = [](int p) -> int { return p*p; };
println("{}", sq(3)); // 9
// or, using 'auto' to deduce the type
auto func = [](int p) { return p*p; };
println("{}", func(3)); // 9
```

Routines III

The C++ language allows member methods (*member functions*) with the inclusion of functions and variables inside aggregates (in C, functions must be external/global). To access fields of the aggregate from within these functions, use the *pointer to the aggregate*, called **this**:

```c
// In C (aggregate type Z)
struct Z {
    int x;
};
void neg(struct Z* this) {
   printf("%d\n", -1*(this->x));
}

// C: using Z and neg
struct Z z = {.x = 10};
neg(&z);
```

```cpp
// In C++ (aggregate type Z)
struct Z {
   int x;
   void neg() {
      println("{}", -1*(this->x));
      // println("{}", -1*x);
   }
};
// C++: using Z and neg
auto z = Z{.x = 10};
z.neg();  // this = &z
```

Routines IV

Functions can call themselves again during their execution in a *recursive* process. The implementation of member functions can also occur outside the aggregate with the scope resolution notation (`::`):

```
struct Fact {
   int factorial(int n);       // declaration: ends with ;
};

int Fact::factorial(int n) {  // definition or implementation
   if (n < 2)
      return 1;
   else
      return n * factorial(n - 1);
}
// using inside main()...
Fact f;
println("{}", f.factorial(5)); // 120
```

The std::span Type in the STL

Like `std::string_view`, for other arrays `int[]`, `std::array` and `std::vector`, `std::span` supports *non-owning* data sequences.

```
import std;        // calling ./program 1 2 3
int main(int argc, char* argv[]) {
   int v2[] = {1, 2, 3, 4};
   std::span<int> s1{v2};
   for (auto i : s1) std::println("{}", i);
   // 1 2 3 4
   std::vector<int> vec = {1, 2, 3, 4};
   std::span<int> s2{vec};
   for (auto i : s2) std::println("{}", i);
   // 1 2 3 4
   std::span<char*> input{argv, argc};
   for (auto i : input) std::println("{}", i);
   // ./program 1 2 3
   return 0;
} // ============================================
```

The std::optional Type in the STL

`std::optional<type>` represents an optional value, with allocation on the *stack*, not on the *heap* like pointers (and *smart pointers*, which we will see next). Access is done with the (*) operator.

```
auto find(char c, std::span<char> v) -> std::optional<int> {
  // searches for char 'c' in an array v and returns the position
  for (int i = 0; i < v.size(); i++)
    if (v[i] == c) return i;  // found
  // not found
  return std::nullopt;
}
// ...
std::vector<char> v = {'a', 'b', 'c'};
auto op = find('x', v);
if(op) println("position={}", *op);
else   println("not found");
```

The std::expected Type in the STL

`std::expected<type, error_type>` represents an *expected* value, with allocation on the *stack*, not on the *heap* like pointers (and *smart pointers*, which we will see next). Access is done with the (*) operator.

```
auto find2(char c, std::span<char> v) -> std::expected<int, std::string> {
  // searches for char 'c' in an array v and returns the position
  for (int i = 0; i < v.size(); i++)
    if (v[i] == c) return i;  // found
  return std::unexpected{"not found"};
}
// ...
std::vector<char> v = {'a', 'b', 'c'};
auto exp = find2('x', v);
if(exp) println("position={}", *exp);
else    println("{}", exp.error());
```

Modularization of Routines and Aggregates (CXX Modules)

Routines and aggregates can be *exported* to other compilation units, through the word `export`. A module file has the extension `.cppm` and begins with `export module MODULE_NAME;`. Both conventional `.cpp` files and `.cppm` modules can import modules, with the keyword `import`.

```cpp
// file test.cppm
export module test;
import std;

export void hello() {std::println("hello!");}

export struct ABC {
   int zero() {
      return  0;
   }
};
```

```cpp
// file main.cpp
import test;
import std;

int main() {
   ABC abc;
   std::println("{}", abc.zero()); // 0
   hello();
   return 0;
}
```

Section 7

Part 5: Abstract Types and Concepts

Concepts I

C++20 brings the possibility of defining concepts (or *concepts*). This feature allows *generic definitions* about some type (including aggregate types with internal functions).

For example, we can create a *concept* `HasNeg`, which requires the aggregate to have a `neg()` method:

```
template <typename T>
concept HasNeg = requires(T a) {
  { a.neg() };
};
```

Example of an aggregate conforming to the `HasNeg` concept:

```
struct Z {
   int x;
   void neg() { std::println("{}", -1*x); }
};
```

Concepts II

Thus, we can use a more specific concept instead of an automatic type:

```
auto  a0        = Z{.x = 1};      // automatic type
auto  p0        = new Z{.x = 1}; // pointer type
auto* p1        = new Z{.x = 1}; // pointer type
HasNeg auto a2 = Z{.x = 2};      // conceptual type
Z     a3        = Z{.x = 3};      // explicit type
```

Another form of type validation at *compile time* is `static_assert`. For example, how do we guarantee that the aggregate Z conforms to the concept HasNeg?

```
static_assert(HasNeg<Z>);
```

Important: the notion of *concepts* is fundamental for understanding *abstract types*, central in the data structures course.

Section 8

Part 6: Smart Pointers and std::move in C++

Pointers III

Pointers are recognizedly problematic structures, therefore since the C++11 revision it is recommended to use *smart pointers* instead of native pointers. There are two kinds of smart pointers: `unique_ptr` and `shared_ptr`. Both avoid the user having to deallocate memory (*with the exception of cyclic structures, to be addressed in the future*). To use them, you just need to include the `<memory>` header, and replace `new` with `std::make_unique` or `std::make_shared`.

```cpp
// Allocates (C++) aggregate P
auto vp = new P{
                  .x = 10,
                  .y = 'Y'
                };
// prints x (value 10)
print("{}\n", vp->x);
// discards the memory
delete vp;
```

```cpp
// Allocates (C++) aggregate P
auto vp = std::make_unique<P>(
      P{.x = 10, .y = 'Y'});


// prints x (value 10)
print("{}\n", vp->x);
// discards the memory
// delete vp;
```

Pointers IV

Pointers can be used as markers of an invalid memory space, generally called *null*. In C, the `NULL` macro is generally defined as zero, so it is a better practice to use the number zero directly instead of `NULL`. The conditional can be used to check a pointer as a boolean, which is the safest option. In C++, there is `std::nullptr`, which can be used in specific situations (generally *smart pointers*), but in general avoid `NULL` and `std::nullptr`.

```cpp
// Allocates (C++) the aggregate P
auto vp = new P{
                  .x = 10,
                  .y = 'Y'
                };
if(vp)        print("success!\n");
if(!vp)       print("failure!\n");
if(vp==NULL)  print("failure!\n");
if(vp==0)     print("failure!\n");
if(vp)        delete vp;
```

```cpp
// Allocates (C++) the aggregate P
auto vp = std::make_unique<P>(
            P{.x = 10, .y = 'Y'}
          );

if(vp)  print("success!\n");
if(!vp) print("failure!\n");

// resets manually
vp = nullptr;
```

Parameter Passing by Reference I

In C, it is only possible to pass variables by copy, which demands the use of pointers to avoid bulky and unnecessary copies.

In C++, there are the concepts of *left-side reference* (`&`) and *right-side reference* (`&&`). In short, we use a `type&` to denote a *reference to live data*, and `type&&` for a *reference to data about to die* (or *data in movement*). This concept is fundamental to deal with `unique_ptr`, since they do not allow copies, being necessarily passed by reference.

To transform a *live variable* into a *variable in movement*, you just need to use the `std::move` command.

```
auto p1 = std::make_unique<P>(P{.x = 10, .y = 'Y'});
std::println("{}", p1->x); // prints x (value 10)
auto p2 = std::move(p1);
if(!p1) std::print("p1 does not exist anymore!\n");
std::println("{}", p2->x); // prints x (value 10)
```

Section 9

Part 7: Coroutines (advanced topic)

Coroutines I

In addition to the classic *routines*, which return (or do not return) values, there are also *coroutines*, with the capacity to *pause and resume* execution.

One example is the *fibonacci sequence*, which starts from 0, 1, and continues with the sum of the *last two elements*. This is an *infinite sequence*, and we can easily represent it with `co_yield` returns of a coroutine with `std::generator`:

```
auto fibonacci() -> std::generator<int> {
  int b = 1, a = 0;
  while (true) {
    co_yield b;
    // a, b <- b, b+a
    int b2 = a + b; a = b; b = b2;
  }
}
```

Coroutines II

To consume the values, just use the range for (all Fib less than 10):

```
for (int num : fibonacci()) {
    if (num > 10) break;
    else std::println("{}", num);  // 1 1 2 3 5 8
}
```

Challenge: how do we implement this same functionality without a coroutine?

Challenge 2: another use is `std::future` with coroutines connected to concurrent programming with threads. This is a bit outside the scope of this course, but check out other applications of coroutines and `co_await`.

Section 10

Part 8: References in C++ (advanced topic)

Parameter Passing by Reference II

```cpp
// C++
auto printx(P* vp) -> void {
  // prints x (value 10)
  println("{}", vp->x);
}
// ...
auto p = P{
          .x = 10,
          .y = 'Y'
          };
// pointer copy
printx(&p);
```

```cpp
// C++
auto printx(P& vp) -> void {
  // prints x (value 10)
  println("{}", vp.x);
}
// ...
auto p = P{
          .x = 10,
          .y = 'Y'
          };
// reference (lvalue)
printx(p);
```

Parameter Passing by Reference III

Left-side references (*lvalue*) complement right-side references (*rvalue*). Observe:

```
void test1(int x) {
   x = 10;
}
void test2(int* x) {
   *x = 10;
}
void test3(int& x) {
   x = 10;
}
void test4(int&& x) {
   x = 10;
}
```

```
int a = 20;
test1(a);      // a == 20
test2(&a);     // a <- 10
test3(a);      // a <- 10
// test4(a);   // ERROR
test4(std::move(a)); // OK
// supposedly a <- 10

test1(20);     // OK
// test2(20);  // ERROR
// test3(20);  // ERROR
test4(20);     // OK
```

Observation: there is also the `const type&` syntax which allows *lifetime extension*, something we will not explore in this brief review.

The std::unique_ptr Type

std::unique_ptr<type> represents a unique pointer to type (as if it were type*). A useful function is get, which returns a native C pointer to the data. The reset function deletes the pointer manually.

```
auto p1 = new int{10};
auto p2 = p1;
println("*p1={} *p2={}", *p1, *p2);
// *p1=10 *p2=10
delete p1;

auto u1 = std::make_unique<int>(10);
auto u2 = std::move(u1);
auto p3 = u2.get();
println("*u2={} *p3={}", *u2, *p3);
// *u2=10 *p3=10
u2.reset();   // deletes pointer u2 manually
u2 = nullptr; // deletes pointer u2 manually
```

Section 11

Part 9: Experimental and advanced libraries in C++

What is the STL standard library?

The language's standard library has components that are already tested and in common use, solving several basic programming problems. C++ has quite important implementations in its standard library, called the STL. In the past, it was necessary to use `#include<...>` to include these components, but since C++23 it is possible to do it all automatically with an `import std`, using the modern structure of CXX Modules.

We have already seen indirectly the use of some of these structures in the course, such as: tuples in `std::tuple`; smart pointers in `std::make_unique` or `std::make_shared`; among other things. We have also seen examples of very fundamental structures such as: `std::string` and `std::vector`. Generally, proposals are made by the community, and good implementations are incorporated into the standard library, in future revisions of the language.

The std::shared_ptr Type (advanced)

`std::shared_ptr<type>` represents a shared pointer to `type` (as if it were `type*`). A useful function is `get`, which returns a native C pointer to the data. The `reset` function deletes the pointer manually. The shared one allows copies and sharing, through *reference counting*. Be careful with cycles, since they can lead to memory leaks! For that, use `std::weak_ptr` or `cycles::relation_ptr` (next). To use it, you just need `#include <memory>`. Example:

```
auto s1 = std::make_shared<int>(10);
auto s2 = s1;
std::weak_ptr<int> w1 = s1;
auto s3 = w1.lock();
println("*s1={} *s2={} *s3={}", *s1, *s2, *s3);
// *s1=10 *s2=10 *s3=10
s1.reset(); // deletes pointer s1 manually
println("*s2={} *s3={} still exist!", *s2, *s3);
```

The std::function Type (advanced)

The `std::function<type>` structure allows storing functions, be it a lambda without capture (*captureless lambda*) or a capturing lambda, also called a *closure*. A *captureless lambda* can decay to a function pointer, while the others can only be encapsulated as `std::function`. You just need `#include <functional>`. Example:

```
// captureless lambda
int(*fsq1)(int) = [](int p) -> int { return p*p; };
std::function<int(int)> fsq2 = [](int p) { return p*p; };
// capturing variable x (by copy)
int x = 10;
int y = 20;
// closure x1 (returns x + 1)
std::function<int()> x1 = [x]() { return x+1; };
// capturing all local variables with =, y by reference
std::function<int()> fxy = [=, &y]() { y++; return x+y; };
int z = fxy(); // z==31  y==21
```

Deducing this with C++23 (advanced)

An interesting capability of C++23 is *deducing this*, which allows working with typing over the `this` variable in functions. This can be useful to capture `this` as a reference, instead of a pointer, and also to *name anonymous functions*.

```
auto factorial = [](this auto func, int n) {
    if (n < 2)
        return 1;
    else
        return n * func(n - 1);
};
//
println("{}", factorial(5)); // 120
```

Proposal for a `std::scan` (advanced/experimental)

Just like `std::print` (currently from `fmt`), there are proposals for a `std::scan`, currently in the `scnlib` project by `eliaskosunen`.

The experimental proposal for C++26 is called P1729 "Text Parsing", and seeks to create a `scn::scan` function that replaces `scanf` (by the same reasoning employed in the abolition of `printf`). Example:

```cpp
#include <scn/scn.h>
// remember to include the eliaskosunen/scnlib package in CMake
using scn::scan;
// ...
int x = 0;
int y = 0;
auto rest = scan("10 20", "{}", x);
scan(rest, "{}", y);
print("x={} y={}", x, y);
// x=10 y=20
```

The cycles::relation_ptr Pointer (advanced/experimental)

A smart pointer proposal to solve cyclic cases was created by prof. Igor Machado Coelho, called cycles::relation_ptr. This is an interesting project to understand the limitations of current smart pointers, and what could possibly be improved in a future C++. Example:

To use it, you just need #include <cycles/relation_ptr>. Example:

```
using cycles::relation_pool;
using cycles::relation_ptr;
// see instructions at: https://github.com/igormcoelho/cycles
relation_pool<> group;
auto r1 = group.make<int>(10);
auto r2 = std::move(r1);
print("*r2={}\n",*r2);
// *r2=10
r2.reset(); // deletes pointer r2 manually
```

Quick Discussion: C or C++?

We quote the C++ direction group, "DIRECTION FOR ISO C++" (2022-10-15), by H. Hinnant, R. Orr, B. Stroustrup, D. Vandevoorde, M. Wong (page 10):

C++ is seriously underrepresented in academia and often very poorly taught. It has been conventional to start teaching C++ by first introducing the lowest level and most error-prone facilities. Naturally, that discourages students and increases the time needed to get to what students consider meaningful computing (graphics, networking, mathematics, data analysis, etc.). Often, teachers even go to the extreme of insisting on using a C compiler. If the ultimate aim is to teach C++, that's like insisting people start learning English by reading Beowulf or the Canterbury Tales in their original early-English language versions. Those are great books, but Early English is incomprehensible to most native Modern-English speakers.

Quick Discussion: C or C++? (continued)

We quote the C++ direction group, "DIRECTION FOR ISO C++" (2022-10-15), by H. Hinnant, R. Orr, B. Stroustrup, D. Vandevoorde, M. Wong (page 10):

In addition to the linguistic difficulties, such ancient sources present cultural conventions and idioms that seem very peculiar today. Instead of C, someone could teach Simula to prepare for learning C++. Why don't people do that? Because the historical approach to teaching language (natural or programming language) complicates and detracts from the end goal: good code.

Why then do teachers use the C-first approach to teach C++? Part is tradition, curriculum inertia, and ignorance, but part of the reason is that C++ doesn't offer a smooth path to idiomatic, proper, modern use of C++. It is hard to bypass both the traps of low-level constructs and the complexities of advanced features and teach programming and proper C++ usage from the start.

Quick Discussion: C or C++? (summary)

In summary: modern C++ is already absolutely superior to C in safety and clarity, with equivalent performance, but historically it lacks good structures for doing the **basics** (such as printing to the screen, making arrays, etc.), forcing the use of unsafe structures, such as pointers. So, recent revisions have pursued this goal, of making basic usage easier (such as `std::print`, `std::array`, `std::string`, `std::vector`, smart pointers, ...) and avoiding the need for the C language to write basic programs.

Today (2023 with the C++20 standard) there are still problems, such as:

- the need to use external libraries (we are needing `fmt::print` and `scn::scan`)
- the need to do `#include` in basic code: the idea is that, starting from the implementation of `import std` in C++23, it will be unnecessary to include external libraries in basic code :)

Many will be solved in the next edition of C++ (but `scn::scan` will still be missing), always keeping an eye on good modern competitors such as Rust.

Section 12

Modularization and Testing (to be revised!!)

Motivation: Modularization and Testing

Any complex program needs division into parts, or modules, for greater control and verification of the correctness of the operations.

In this course, we will use a minimal modularization standard, so that it is possible to carry out tests on the code (in a systematic way).

Basic Modularization

A program starts at its "entry point" (or *entrypoint*), typically an `int main()` function:

```
#include<iostream> // includes an external file
int main() {
   return 0;       // 0 means: no error
}
```

The declaration of functions can be done before the definition:

```
int square(int p); // declares the 'square' function
int square(int p) {
   return p*p;     // implements the 'square' function
}
```

Declarations come in `.h` files, while the respective implementations come in a `.cpp` file (or together as `.hpp`).

Running `main.cpp`

When using GCC and an *entrypoint* in the `main.cpp` file:

To compile: `g++ -std=c++23 -O3 main.cpp -o appMain`

To run the code: `./appMain`

Important: we consider a GNU/Linux system, but in case it is Windows one can use the C/C++ compiler MinGW and run the generated application with a `.exe` extension (Windows executable standard).

File Organization I

Minimal modularization: 4 files.

- one entry point (entrypoint) - usually `main.cpp` (tip: put it in the `src/` folder)
- one (or more) file(s) with the other modules (tip: put them in the `src/` folder)
- one (or more) file(s) with your tests - usually `test.cpp` (tip: put them in the `tests/` folder)
- one file (at the root) with build information - usually a GNU `makefile` (with `all:` and `test:` rules)

File Organization II

It is also informative to have an extra file at the root with explanations about the code (typically `README.md` in the markdown language)

Important: the *entrypoint* file must contain exclusively the `int main()` function (and its respective `#includes`), to make code testing feasible.

Types in the C++ standard library

During the course we will study several data structures, but whenever possible use the existing ones in the standard library (STL). They are "more efficient" and "error-proof".

For example, it is easy to define an aggregate type `Pair`, which holds two internal elements (generic type). However, it is more advantageous to use the existing one in the STL, called `std::pair` (the `std::` prefix is called a *namespace* and avoids name collisions):

```
#include<iostream> // input/output functions
#include<tuple>    // pair and tuple aggregates
int main() {
   std::pair<int, char> p {5, 'C'}; // direct init.
   printf("%d %c\n", p.first, p.second); // 5 C
   // ...
}
```

Recalling (aggregate Z)

```
// In C++ (aggregate type Z)
struct Z
{
   int x;
   // prints field x
   void printx() {
      printf("%d\n", this->x);
   }
};
```

Recalling (concept HasPrintX)

```
template<typename Aggregate>
concept bool
HasPrintX = requires(Aggregate a) {
   {
      a.printx()
   }
};
```

Checks with assert

During development, it is useful to check parts of the code with simple tests that are necessary for its correctness (in real time). For that, we can use `assert()`. Example:

```
int x = 10;
x++;
assert(x == 11); // x should be 11
```

In the same way, we can check types, especially *concepts*, at compile time:

```
// checks whether aggregate type Z has the printx() method
static_assert(HasPrintX<Z>);
```

Testing with the Catch2 library

A practical way to test modularized code with `main.cpp` separated from `rest.hpp` is by using the Catch2 library.

You just need to create a test file, for example, `test.cpp`:

```
#include "rest.hpp"

#define CATCH_CONFIG_MAIN // catch2 main()
#include "catch.hpp"

TEST_CASE("Tests initialization of aggregate Z")
{
   auto z1 = Z{.x = 10};
   // checks whether z1.x is in fact 10
   REQUIRE(z1.x == 10);
}
```

Downloading Catch2 and running

To download the `catch2.hpp` file, just visit the project site: github.com/catchorg/Catch2. Direct link (August 2020):

github.com/catchorg/Catch2/releases/download/v2.13.1/catch.hpp

To compile: `g++ -fconcepts test.cpp -o appTests`

To run the tests: `./appTests -d yes`

```
0.000 s: Tests initialization of aggregate Z
=============================================
All tests passed (1 assertion in 1 test case)
```

Important: The options `-fsanitize=address` and `-g3` are recommended to avoid bugs during development using GCC.

Keep Learning

In this review about types, we sought not to go deep into any "advanced" characteristic of C/C++, although some concepts may seem new. Recommended topics (not covered in the course):

- Object Orientation (other courses cover this topic)
- frequent use of *references* (instead of pointers)
- frequent use of *move semantics* (instead of references)
- frequent use of *closures* (instead of functions and lambdas)
- frequent use of self-managing memory, such as `std::unique_ptr` and `std::shared_ptr` (does not require `delete`)
- use of C++20 *coroutines* (we only considered *routines* in the course), especially for building infinite iterators
- microbenchmark testing (we recommend the Google Benchmark library)

Recommended Bibliography

Besides the course bibliography, we recommend (for this topic):

- Book "Introdução a estruturas de dados" by W. Celes and J. L. Rangel
- Book "The C++ Programming Language" by Bjarne Stroustrup
- C++ tips and guidelines: https://github.com/isocpp/CppCoreGuidelines

Section 13

Acknowledgements

People

In particular, I thank the colleagues who elaborated good materials, such as prof. Fabiano Oliveira (IME-UERJ), and prof. Jayme Szwarcfiter whose concepts form the core of these slides.

I extend my thanks to the other colleagues who collaborated on the elaboration of the material of the Operations Research course, which paved the way for practical verification of this slide technology.

Software

This course material is only possible thanks to the countless open-source projects that are necessary to it, including:

- pandoc
- LaTeX
- GNU/Linux
- git
- markdown-preview-enhanced (github)
- visual studio code
- atom
- revealjs
- groomit-mpx (screen drawing tool)
- xournal (screen drawing tool)
- ...

Companies

Special thanks to companies that support free projects involved in this course:

- github
- gitlab
- microsoft
- google
- ...

Reproduction of the material

These slides were written using pandoc, according to the ilectures tutorial:

- https://igormcoelho.github.io/ilectures-pandoc/

Except where expressly mentioned (with the due caveats for the material provided by colleagues), the license will be Creative Commons.

Igor Machado Coelho

This Slide Is Intentionally Blank (for goomit-mpx)