Estruturas de Dados I

Introdução a Tipos em C/C++ e Análise de Algoritmos

Igor Machado Coelho

13/09/2020 - 13/08/2026

Section 1

Introdução a Tipos em C/C++ e Análise de Algoritmos

Pré-Requisitos

São requisitos para essa aula o conhecimento de:

- Introdução/Fundamentos de Programação (em alguma linguagem de programação)
- **Interesse** em aprender C/C++
- Familiaridade com uso e instalação de programas via linha de comando (no seu sistema operacional preferido)
- Familiaridade com uso de compiladores/IDEs ou uso de ferramentas de programação online

Ambiente de Programação

Por que programar em C/C++?

- Linguagem consolidada, eficiente e de **baixo nível** (próxima ao *hardware*)
- **Muito** material online e livros, etc
- **Muitas** ferramentas e códigos prontos, funciona em qualquer sistema operacional

Como programar em C/C++?

- Instalar compilador **moderno** para **C23/C++23**. Recomendação geral: **clang 22**
  - Link de download: https://github.com/llvm/llvm-project/releases
- Para Linux, pode usar o **GCC 15** ou **clang**
- Para Mac, use o **clang**
- Para Windows, **evite** tanto o **MSVC** nativo quanto o ***clang-cl*** *para Windows*!
  - **Deve** instalar o **WSL2** no Windows e usar o suporte de linux recomendado acima.
- Se for usar um compilador online, use o https://godbolt.org

Section 2

Parte 1: Tipos Primitivos, Compostos e Genéricos em C/C++

Conceitos de Tipos em C/C++

Compreender a lógica da programação é a habilidade mais importante para um programador! Com ela, você pode facilmente trocar de linguagem de programação, conhecendo apenas alguns comandos básicos.

O primeiro conceito a ser revisado é de variável. Uma variável consiste de um identificador válido (mesmo para outras linguagens populares como Python) e armazena algum tipo de dado da memória do computador.

A linguagem C/C++ é **fortemente tipada**, portando o programador deve dizer explicitamente qual o tipo de dado deseja armazenar em cada variável ou deixar **auto** deduzir automaticamente.

```
int    x = 5;    // armazena o inteiro 5 na variável x
char   y = 'A';  // armazena o caractere 'A' na variável y
float  k = 3.7f; // armazena o real 3.7 na variável z
double z = 3.7;  // armazena o real 3.7 na variável z
bool v = true;   // armazena o booleano true na variável v
auto b = 'B';    // dedução de tipo com 'auto'... qual tipo?
auto s = "abcd"; // cadeia de caracteres, ainda veremos tipo
```

Tipos de Variáveis

Pergunta/Resposta: Cuidado com tipos. Quais são os valores armazenados nas variáveis abaixo (C++23 e 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
```

Verifiquem essas operações de variáveis, escrevendo na saída padrão (tela do computador).

Conceitos de C/C++ (tipos definidos)

Tipos primitivos em C/C++ tem um tamanho definido, então é uma boa prática utilizar tamanhos fixos.

Dê preferência a inicialização direta com chaves `{ }`, ao invés de indireta por atribuição (`operator=`).

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

Introdução a Rotinas: Funções

A modularização de programas é muito importante, principalmente quando trechos de código são repetidos muitas vezes.

Nesses casos, é comum criar rotinas, como *funções e procedimentos*, que podem por sua vez receber parâmetros.

Tomemos por exemplo a função quadrado que retorna o valor passado elevado ao quadrado.

```
// função que retorna um 'int',
//   com parâmetro 'p' inteiro
int quadrado (int p) {
   return p*p;
}
// qual o tipo?
auto x = quadrado(5);
```

```
// função que retorna um 'int',
//   com parâmetro 'p' inteiro
auto quadrado (int p) -> int {
   return p*p;
}
// qual o tipo?
auto x = quadrado(5);
```

Importante: a dedução do tipo após a seta -> é feita automaticamente.

Impressão de Saída Padrão

Em C, tipicamente é utilizado o comando `printf`, mas devido a inúmeras falhas de segurança, é recomendado o uso de uma alternativa mais segura. Assim, em C++, para imprimir na saída padrão utilizaremos o comando `std::print`.

O C++23 traz oficialmente `std::print` e `std::println` como parte do módulo de biblioteca padrão `std`. Para utilizá-lo, basta fazer `import std;`.

```
import std;

int main() {
  std::println("Olá Mundo!");
  return 0;
}

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

Pergunta: Qual o valor de retorno da função `main`? O que ele significa?

Impressão de Saída Padrão

Para imprimir na saída padrão utilizaremos o comando `std::print`. Obs: é possível evitar o prefixo `std::` com um `using namespace std;`.

Pergunta: como podemos misturar um texto (também chamado de cadeia de caracteres ou string) com o conteúdo de variáveis? **Resposta:** através do padrão de substituição {}.

```
import std;
int main() {
  int x1 = 7;
  std::println("x1 é {}", x1);  /*  x1 é 7  */
  double x6 = x1 / 2.0;
  std::println("metade de {} é {}", x1, x6);  // metade de 7 é 3.5
  char b = 'L';
  std::println("isto é uma {}etra", b);  // isto é uma Letra
  std::print("Olá mundo! \n");   // Olá mundo! (quebra de linha)
  return 0;
}
```

Condicionais

Problema: dados `x` e `y`, imprima o maior valor.

Condicionais podem ser feitos através dos comandos if ou if else.

```
int x = 15;
int y = 12;
if (x > y)
   println("x é maior que y");
else
   println("x menor ou igual a y");
```

Pergunta: Qual resultado da expressão `if(x = y)`? E `if(x == y)`?

Laços de Repetição (Parte 1/2)

Laços de repetição podem ser feitos através de comandos while ou for. Um comando for é dividido em três partes: inicialização, condição de continuação e incremento.

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

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

Pergunta: O que é impresso em ambos laços?

Tipos `void` e `std::monostate`

Vimos alguns tipos com número maior de valores, por exemplo, o **int** e o **char**. O **int** ocupa 4 bytes e é capaz de suportar até 2^{32} valores distintos (aprox. -2 bilhões até +2 bilhões), enquanto o **char** suporta até 256 valores ocupando apenas 1 byte.

Em alguns casos, também é util o uso de tipos com menor número de valores, como o **std::monostate**, que ocupa 1 byte e só tem um único valor, e finalmente o tipo **void**, que é um *tipo incompleto* e *não representa nenhum valor*. Como o **void** não representa valores, não é possível criar variáveis do tipo **void**!

```
int              i = 10;     // um dentre 4 bilhões de valores válidos
bool             b = true;   // um dentre dois valores válidos
// void          v;          // erro de compilação: void não tem valor
std::monostate   m;          // somente um valor possível
auto             n = m;      // monostate pode ser copiado
```

Importante: o tipo `std::monostate` só existe em C++, pois em C nunca houve a necessidade de definir esse tipo, por falta de *tipos genéricos* (veremos mais adiante).

Introdução a Rotinas: Procedimentos

Quando nenhum valor é retornado (em um procedimento), utilizamos o tipo **void**. Procedimentos são úteis mesmo quando nenhum valor é retornado. **Exemplo**: (de a até b):

```
import std;

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

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

```
import std;

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

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

Pergunta: Por que a rotina `imprime` não precisa retornar nada? O que acontece de *efeito colateral* ao invocar `imprime(2,5)`?

Tipos Compostos: Vetores

Além dos tipos primitivos apresentados anteriormente (int, float, char, ...), a linguagem C/C++ nos permite criar tipos compostos.

Tarefa: estude demais tipos primitivos como double e long long, bem como os modificadores unsigned, signed, short e long.

Os tipos compostos podem ser agregados homogêneos (vetores/arrays) ou agregados heterogêneos (structs, ...).

Tipos Compostos: Vetores

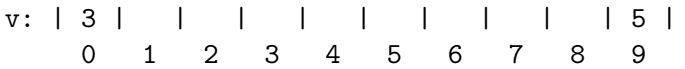
Além dos tipos primitivos apresentados anteriormente (int, float, char, ...), a linguagem C/C++ nos permite criar tipos compostos.

Tarefa: estude demais tipos primitivos como double e long long, bem como os modificadores unsigned, signed, short e long.

Os tipos compostos podem ser agregados homogêneos (vetores/arrays) ou agregados heterogêneos (structs, ...).

```
int v[10];  // cria um vetor com 10 inteiros  (40 bytes)
v[0] = 3;   // atribui o valor 3 à primeira posição
v[9] = 5;   // atribui o valor 5 à última posição
```

Exemplo de um vetor v, de 0 a 9, da esquerda para direita:



Controle de fluxo com `break` e `continue`

Controles de fluxo em laços de repetição podem ser efetuados com `break` e `continue`. O `break` finaliza a execução do laço e o `continue` recomeça o laço.

Problema: Dado um vetor B, encontre o primeiro/último valor negativo, ou imprima -1 caso não exista.

```
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
```

Saltos incondicionais com goto (tópico avançado)

Saltos incondicionais no código podem ser feitos com `goto label;` e `label:`. Uma aplicação usual é a "quebra múltipla" de laços de repetição. Evite ao máximo o uso de `goto` e, sempre que for possível, prefira alternativas estruturadas como `for`, `while`, `if`, `else`, `break`, etc.

Contabilize quantos prints são executados (variável `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 fim;
      println("z={} i={} j={}", z, i, j); z++; j++;
    }
  }
fim:
  println("final z={}", z);
  // z==9: i=5 j=5..9 [5 passos]; i=6 j=6..9 [4 passos]
```

Section 3

Prática: Programas em C/C++

Ambiente de Programação (detalhes)

Exemplos serão dados com base no sistema GNU/Linux e compiladores GCC, mas existem ferramentas equivalentes para Windows e demais sistemas operacionais. A IDE Visual Studio Code suporta a linguagem C++ tanto para Linux quanto para Windows, sendo necessário o CMake 4.0 com Ninja. No Windows/WSL ou Linux, instale o compilador Clang 19 (no Windows, use o Scoop com `scoop install main/llvm`).

Também é possível praticar diretamente em um navegador web com plataformas online: onlinegdb.com/online_c++_compiler ou Godbolt (mais recomendada!). Neste caso, o aluno pode escolher o compilador de C ou da linguagem C++ (considerando padrões C23 e C++23).

Para configurar IDE, leia o tutorial "Breve Introdução ao C/C++ com IDE de Desenvolvimento".

Exemplo de Programa em Padrão C

O Compilador GCC suporta C/C++, então consegue compilar tanto C quanto *C com C++*.

Código main.c:

```c
#include <stdio.h>

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

Para compilar manualmente (sem CMake), execute o seguinte comando com GCC 15:

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

Como o C++ inclui tudo que o C tem e ainda adiciona recursos mais seguros e simples, como `import` e `print`, utilizaremos o padrão C++ no compilador.

Exemplo de Programa em C/C++ (GCC Manual)

Veja exemplo no https://godbolt.org/z/j3W938PP6:

Código main.cpp:

```
import std;

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

Para compilar manualmente (sem CMake), execute o seguinte comando com GCC 15 ou 16:

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

Para programas maiores e mais complexos (com muitos arquivos), é necessário usar algum sistema de construção, como CMake ou Bazel. No próximo slide, um exemplo de CMakeLists.txt.

Exemplo de Programa em C/C++ com CMake 4.4 e GCC 16

Veja exemplo no https://godbolt.org/z/We6fdojj1 (cmake 4.4.2):

Código CMakeLists.txt (precisa instalar CMake 4.4 e 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(exemplo src/main.cpp)
```

Exemplo de Programa em C/C++ com CMake 4.4 e GCC 16

Veja exemplo no https://godbolt.org/z/We6fdojj1 (cmake 4.4.2):

Para construir, são quatro comandos:

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

Outra solução bem elegante é utilizar o Bazel com o compilador Clang.

Dica para IDE VSCode: use extensão clangd

Para utilizar IDE de desenvolvimento como o VSCode, é necessário usar Clang com extensão clangd para um correto processamento visual do código (não funciona direito com GCC nem com a extensão padrão C/C++ da Microsoft).

Exemplo de Programa em C/C++ com Bazel 9 e Clang

Veja exemplo no tutorial "Local import std module on C++23 with Bazel". Só precisa configurar os seus arquivos MODULE.bazel e BUILD (copie o `extensions.bzl` do tutorial!).

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

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 = "exemplo",
    srcs = ["main.cpp"],
    deps = ["@std_modules"]
    features = ["cpp_modules"]
)
```

Exemplo de Programa em C/C++ com Bazel 9 e Clang

Veja exemplo no tutorial "Local import std module on C++23 with Bazel".

Para construir e executar, são dois comandos:

```
bazel build ...
bazel run :exemplo
```

Lembre-se de atualizar seu arquivo `.bazelrc` com versão correta do compilador:

```
# .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

Tipos Agregados Triviais

Tipos Agregados I

Comparação C/C++: lembre-se de usar **struct** ou **class public:**, caso contrário não será reconhecido como um *tipo agregado*, mas sim um *objeto*, que funciona de forma completamente diferente na linguagem C++.

```c
// Em C (tipo agregado P)
struct P {
   int  x;
   char y;
};

// declara variável tipo P
struct P p1;
// designated initializers
struct P p2 = {.x=10, .y='Y'};
```

```cpp
// Em C++ (tipo agregado P)
struct P {
   int  x;
   char y;
};

// declara variável tipo P
P p1;
// designated initializers
auto p2 = P{.x=10, .y='Y'};
```

Importante: sempre utilizaremos **struct** no decorrer desse curso de C/C++ como um *tipo agregado trivial*, nunca como uma *classe*.

Tipos Agregados II

Retomamos o exemplo da estrutura P anterior e nos perguntamos, como acessar as variáveis internas do agregado P?

Assim como na inicialização designada, podemos utilizar o operador ponto (.) para acessar campos do agregado. Exemplo:

```
auto p1 = P{.y = 'A'};
p1.x = 20;              // atribui 20 à variável x de p1
p1.x = p1.x + 1;        // incrementa a variável x de p1
println("{} {}", p1.x, p1.y);  // imprime '21 A'
```

Exemplo de estrutura p1, com p1.x e p1.y, da esquerda para direita:

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

Importante: veremos à frente que o (.) também pode acessar *métodos internos* do tipo agregado.

Espaço de Memória e Métodos em Agregados

Todas variáveis de um programa ocupam determinado espaço na memória principal do computador. **Assumiremos** que o tipo int (ou float) ocupa 4 bytes, enquanto um char ocupa apenas 1 byte.

No caso de vetores, o espaço ocupado na memória é multiplicado pelo número de elementos. Vamos calcular o espaço das variáveis:

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

Já nos agregados, assumimos o espaço ocupado como a soma de suas variáveis internas (embora na prática o tamanho possa ser ligeiramente superior, devido a alinhamentos de memória).

Importante: em C++, agregados triviais *também podem conter métodos*.

Tipos Genéricos

C++ permite a definição de tipos genéricos, ou seja, tipos que permitem que algum *outro tipo* seja passado como parâmetro.

Consideremos o agregado P que carrega um int e um char... como transformá-lo em um agregado genérico em relação à variável x?

```
template<typename T>
struct G {
   T x;   // qual o tipo da variável x?
   char y;
};
// declara o agregado genérico G com tipo T=float ou T=char
G<float> g1 = {.x = 3.14, .y  = 'Y'};
G<char>  g2 = {.x = 'A',  .y  = 'Y'};
```

Pergunta: Quanto espaço (em bytes) cada variável dessa ocupa?

Valores constantes e casts

Em C/C+, podemos definir um valor como constante, através da palavra `const`. Uma mudança de tipos pode ser feita com *type cast*. Em C++, utilize `static_cast<tipo>` ao invés do padrão C de 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;                                 // ERRO
```

O `const` pode ser removido através de um `const_cast`, sendo inseguro.

Em C23 e C++23 existe o `constexpr`, que diferentemente do `const`, nunca pode ser removido ou redefinido (diferentemente de macros), pois é resolvido em tempo de compilação.

```
#define k1 10           // inseguro e permite redefinição
constexpr int k2 = 10; // seguro, impossível redefinir
```

Tipo std::string e std::string_view na STL

O tipo `std::string` representa cadeias de caracteres, chamadas de *strings*. Ela substitui a necessidade de `char*`, `char[]` ou `const char*` em C.

Caso precise de uma "vista" leve de uma string, como um substring, utilize `std::string_view` (evita a cópia completa do `string`).

```
std::string s1 = "abcd";
std::string s2 = "ef";
println("tamanho1={} tamanho2={}", s1.length(), s2.length());
// tamanho1=4 tamanho2=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);
```

Tipo std::vector na STL

A popular estrutura std::vector<tipo> permite representar vetores com tamanho variável (através do método push_back). Exemplo:

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

Tipo std::array na STL

Assim como vetores nativos, exemplo `int[]`, o agregado `std::array<tipo, tamanho>` permite representar vetores de tamanho fixo. Exemplo:

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

Resumo até agora

Até agora, verificamos as seguinte estruturas:

- tipos primitivos (C)
- tipo automático com **auto** (C)
- introdução a rotinas com retorno **auto** (C++)
- estruturas condicionais e laços de repetição (C)
- vetores (C)
- tipos agregados com **struct** ou **class/public:** (C/C++)
- agregados genéricos (C++)

Section 5

Parte 2: Rotinas, Ponteiros e Conceitos em C/C++

Rotinas I

É possível retornar múltiplos elementos (par ou tupla), através de um *structured binding* com tuplas:

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

Perg.: qual tipo de retorno de 'duplo'? **R:** `std::tuple<int, double>`.

Ponteiros I

Os parâmetros são sempre copiados (em C) ao serem passados para uma função ou procedimento. Como passar tipos complexos (estruturas e vetores de muitos elementos) sem perder tempo?

Nestes casos, a linguagem C oferece um tipo especial denominado ponteiro. A sintaxe do ponteiro simplesmente inclui um asterisco (*) após o tipo da variável. Um estado vazio se faz com **nullptr** (ou 0).

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

Um ponteiro simplesmente armazena **o local** (endereço) onde determinada variável está armazenada na memória (basicamente, um número). Então quando um ponteiro é passado como parâmetro, **a cópia do ponteiro** pode ser utilizada para encontrar na memória a estrutura desejada.

O tamanho do ponteiro varia de acordo com a arquitetura, mas para endereçar 64-bits, ele ocupa 8 bytes.

Ponteiros II

Em ponteiros para agregados, o operador de acesso (`.`) é substituído por uma seta (`->`). O operador `&` toma o endereço da variável:

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

Testando procedimentos `f` e `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
```

Alocação Dinâmica de Memória

Programas frequentemente necessitam de alocar mais memória para uso, o que é armazenado de forma segura em um ponteiro para o tipo da memória:

```c
// Aloca (C) o agregado P
struct P* vp =
   malloc(1*sizeof(struct P));
// inicializa campos de P
vp->x = 10;
vp->y = 'Y';
// imprime x (valor 10)
printf("%d\n", vp->x);
// descarta a memória
free(vp);
```

```cpp
// Aloca (C++) o agregado P
auto vp = new P{
                  .x = 10,
                  .y = 'Y'
                };

// imprime x (valor 10)
println("{}", vp->x);
// descarta a memória
delete vp;
```

Rotinas II

O tipo de uma função é basicamente um ponteiro (endereço) da localização desta função na memória do computador. Por exemplo:

```
// tipo: int(*)(int)
int quadrado(int p) {
   return p*p;
}
```

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

Este fato pode ser útil para receber funções como parâmetro, bem como armazenar funções anônimas (*lambdas*):

```
// armazena lambda no ponteiro de função 'quad'
int(*quad)(int) = [](int p) -> int { return p*p; };
println("{}", quad(3)); // 9
// ou, utilizando 'auto' para deduzir o tipo
auto func = [](int p) { return p*p; };
println("{}", func(3)); // 9
```

Rotinas III

A linguagem C++ permite métodos membros (*member functions*) com a inclusão de funções e variáveis dentro de agregados (em C, funções devem ser externas/globais). Para acessar campos do agregado de dentro dessas funções, utilize o *ponteiro para o agregado*, chamado **this**:

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

// C: uso de Z e neg
struct Z z = {.x = 10};
neg(&z);
```

```cpp
// Em C++ (tipo agregado Z)
struct Z {
   int x;
   void neg() {
      println("{}", -1*(this->x));
      // println("{}", -1*x);
   }
};
// C++: uso de Z e neg
auto z = Z{.x = 10};
z.neg();  // this = &z
```

Rotinas IV

Funções podem se chamar novamente durante sua execução em um processo *recursivo*. A implementação de funções membro pode ocorrer também fora do agregado com a notação do resolutor de escopo (`::`):

```
struct Fat {
   int fatorial(int n);       // declaração: termina em ;
};

int Fat::fatorial(int n) {    // definição ou implementação
   if (n < 2)
      return 1;
   else
      return n * fatorial(n - 1);
}
// utilizando dentro do main()...
Fat f;
println("{}", f.fatorial(5)); // 120
```

Tipo std::span na STL

Como `std::string_view`, para demais vetores `int[]`, `std::array` e `std::vector`, o `std::span` suporta sequências de dados *sem posse*.

```
import std;        // invocando ./programa 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*> entrada{argv, argc};
   for (auto i : entrada) std::println("{}", i);
   // ./programa 1 2 3
   return 0;
} // ============================================
```

Tipo std::optional na STL

O std::optional<tipo> representa um valor opcional, com alocação em *stack*, não em *heap* como ponteiros (e *smart pointers*, que veremos a seguir). O acesso se faz com operador (*).

```
auto busca(char c, std::span<char> v) -> std::optional<int> {
  // busca char 'c' num vetor v e retorna posição
  for (int i = 0; i < v.size(); i++)
    if (v[i] == c) return i;  // encontrou
  // não encontrou
  return std::nullopt;
}
// ...
std::vector<char> v = {'a', 'b', 'c'};
auto op = busca('x', v);
if(op) println("posicao={}", *op);
else   println("não encontrou");
```

Tipo std::expected na STL

O `std::expected<tipo, tipo_erro>` representa um valor *esperado*, com alocação em *stack*, não em *heap* como ponteiros (e *smart pointers*, que veremos a seguir). O acesso se faz com operador (`*`).

```
auto busca2(char c, std::span<char> v) -> std::expected<int, std::string> {
  // busca char 'c' num vetor v e retorna posição
  for (int i = 0; i < v.size(); i++)
    if (v[i] == c) return i;  // encontrou
  return std::unexpected{"não encontrou"};
}
// ...
std::vector<char> v = {'a', 'b', 'c'};
auto exp = busca2('x', v);
if(exp) println("posicao={}", *exp);
else    println("{}", exp.error());
```

Modularização de Rotinas e Agregados (CXX Modules)

Rotinas e agregados podem ser *exportados* para outras unidades de compilação, através da palavra `export`. Um arquivo de módulo tem extensão `.cppm` e começa com `export module NOME_DO_MODULO;`. Tanto arquivos convencionais `.cpp` ou módulos `.cppm` podem importar módulos, com a palavra-chave `import`.

```cpp
// arquivo teste.cppm
export module teste;
import std;

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

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

```cpp
// arquivo main.cpp
import teste;
import std;

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

Section 6

Parte 3: Tipos Abstratos e Conceitos

Conceitos I

C++20 traz a possibilidade de definir conceitos (ou *concepts*). Esse recurso permite *definições genéricas* sobre algum tipo (inclusive tipos agregados com funções internas).

Por exemplo, podemos criar um *conceito* `TemNeg`, que exige que o agregado possua um método `neg()`:

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

Exemplo de agregado de acordo com conceito `TemNeg`:

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

Conceitos II

Assim, podemos utilizar um conceito mais específico ao invés de um tipo automático:

```
auto  a0        = Z{.x = 1};      // tipo automático
auto  p0        = new Z{.x = 1}; // tipo ponteiro
auto* p1        = new Z{.x = 1}; // tipo ponteiro
TemNeg auto a2 = Z{.x = 2};      // tipo conceitual
Z     a3        = Z{.x = 3};      // tipo explícito
```

Outra forma de validação de tipos em *tempo de compilação* é o `static_assert`. Por exemplo, como garantir que o agregado Z está de acordo com o conceito TemNeg?

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

Importante: a noção de *conceitos* é fundamental para a compreensão de *tipos abstratos*, central no curso de estruturas de dados.

Section 7

Parte 4: Ponteiros Inteligentes e std::move em C++

Ponteiros III

Ponteiros são estruturas reconhecidamente problemáticas, portanto desde a revisão C++11 é recomendado que se use *ponteiros inteligentes* (ou *smart pointers*) ao invés de ponteiros nativos. Existem dois tipos de smart pointers: `unique_ptr` e `shared_ptr`. Ambos evitam que o usuário precise de desalocar memória (*com exceção de estruturas cíclicas, a serem abordadas no futuro*). Para utilizá-los, basta incluir o cabeçalho `<memory>`, e substituir o `new` por `std::make_unique` ou `std::make_shared`.

```cpp
// Aloca (C++) agregado P
auto vp = new P{
                  .x = 10,
                  .y = 'Y'
                };
// imprime x (valor 10)
print("{}\n", vp->x);
// descarta a memória
delete vp;
```

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


// imprime x (valor 10)
print("{}\n", vp->x);
// descarta a memória
// delete vp;
```

Ponteiros IV

Ponteiros podem ser utilizados como marcadores de um espaço de memória inválido, geralmente chamado de *nulo*. Em C, a macro `NULL` é geralmente definida como zero, sendo então uma melhor prática usar o número zero diretamente ao invés de `NULL`. O condicional pode ser usado para verificar um ponteiro como booleano, que é a opção mais segura. Em C++, existe o `std::nullptr`, que pode ser utilizado em situações específicas (geralmente *smart pointers*), mas geralmente evite `NULL` e `std::nullptr`.

```cpp
// Aloca (C++) o agregado P
auto vp = new P{
                  .x = 10,
                  .y = 'Y'
                };
if(vp)        print("sucesso!\n");
if(!vp)       print("falha!\n");
if(vp==NULL)  print("falha!\n");
if(vp==0)     print("falha!\n");
if(vp)        delete vp;
```

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

if(vp)  print("sucesso!\n");
if(!vp) print("falha!\n");

// reseta manualmente
vp = nullptr;
```

Passagem de Parâmetros por Referência I

Em C, só é possível passar variáveis por cópia, o que demanda uso de ponteiros para evitar cópias volumosas e desnecessárias.

Em C++, existem os conceitos de *referência de lado esquerdo* (`&`) e *referência de lado direito* (`&&`). Em resumo, utilizamos um `tipo&` para denotar uma *referência a um dado vivo*, e `tipo&&` para uma *referência a um dado prestes a morrer* (ou *dado em movimento*). Esse conceito é fundamental para lidar com `unique_ptr`, pois eles não permitem cópias, sendo obrigatoriamente passados por referência.

Para transformar uma *variável viva* para uma *variável em movimento*, basta usar o comando `std::move`.

```
auto p1 = std::make_unique<P>(P{.x = 10, .y = 'Y'});
std::println("{}", p1->x); // imprime x (valor 10)
auto p2 = std::move(p1);
if(!p1) std::print("p1 não existe mais!\n");
std::println("{}", p2->x); // imprime x (valor 10)
```

Section 8

Parte 5: Corrotinas (tópico avançado)

Corrotinas I (Coroutines)

Além das clássicas *rotinas*, que retornam (ou não) valores, existem também *corrotinas*, com capacidade de *paralisar e retomar* a execução.

Um exemplo é a *sequência de fibonacci*, que começa de 0, 1, e segue com a soma dos *dois últimos elementos*. Essa é uma *sequência infinita*, e podemos facilmente representá-la assim que retornos `co_yield` de corrotina com `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;
  }
}
```

Corrotinas II (Coroutines)

Para consumir os valores, basta usar o for range (todos Fib menores que 10):

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

Desafio: como implementar essa mesma funcionalidade sem corrotina?

Desafio 2: outro uso é o `std::future` com corrotinas conectadas a programação concorrente com threads. Isso foge um pouco do escopo desse curso, mas verifique outras aplicações de corrotinas e `co_await`.

Section 9

Parte 6: Referências em C++ (tópico avançado)

Passagem de Parâmetros por Referência II

```cpp
// C++
auto imprimex(P* vp) -> void {
  // imprime x (valor 10)
  println("{}", vp->x);
}
// ...
auto p = P{
          .x = 10,
          .y = 'Y'
          };
// cópia de ponteiro
imprimex(&p);
```

```cpp
// C++
auto imprimex(P& vp) -> void {
  // imprime x (valor 10)
  println("{}", vp.x);
}
// ...
auto p = P{
          .x = 10,
          .y = 'Y'
          };
// referência (lvalue)
imprimex(p);
```

Passagem de Parâmetros por Referência III

Referências de lado esquerdo (*lvalue*) complementam referências de lado direito (*rvalue*). Observe:

```
void teste1(int x) {
   x = 10;
}
void teste2(int* x) {
   *x = 10;
}
void teste3(int& x) {
   x = 10;
}
void teste4(int&& x) {
   x = 10;
}
```

```
int a = 20;
teste1(a);      // a == 20
teste2(&a);     // a <- 10
teste3(a);      // a <- 10
// teste4(a);  // ERRO
teste4(std::move(a)); // OK
// supostamente a <- 10

teste1(20);     // OK
// teste2(20);  // ERRO
// teste3(20);  // ERRO
teste4(20);     // OK
```

Observação: existe também a sintaxe `const tipo&` que permite *lifetime extension*, algo que não exploraremos nessa breve revisão.

Tipo std::unique_ptr

O std::unique_ptr<tipo> representa um ponteiro único para o tipo (como se fosse tipo*). Uma função útil é o get, que retorna um ponteiro nativo C para o dado. A função reset apaga o ponteiro manualmente.

```
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();   // apaga ponteiro u2 manualmente
u2 = nullptr; // apaga ponteiro u2 manualmente
```

Section 10

Parte 7: Bibliotecas experimentais e avançadas em C++

O que é biblioteca padrão STL?

A biblioteca padrão da linguagem tem componentes já testados e de uso comum, resolvendo diversos problemas básicos de programação. C++ possui implementações bastante importantes em sua biblioteca padrão, chamada STL. Antigamente, era necessário utilizar `#include<...>` para incluir esses componentes, mas desde o C++23 é possível fazer tudo automaticamente com um `import std`, utilizando a estrutura moderna dos CXX Modules.

Já vimos indiretamente o uso de algumas dessas estruturas no curso, como: tuplas em `std::tuple`; ponteiros inteligentes em `std::make_unique` ou `std::make_shared`; entre outras coisas. Também vimos exemplos de estruturas muito fundamentais como: `std::string` e `std::vector`. Geralmente, propostas são feitas pela comunidade, e boas implementações são incorporadas à biblioteca padrão, em revisões futuras da linguagem.

Tipo std::shared_ptr (avançado)

O std::shared_ptr<tipo> representa um ponteiro compartilhado para o tipo (como se fosse tipo*). Uma função útil é o get, que retorna um ponteiro nativo C para o dado. A função reset apaga o ponteiro manualmente. O shared permite cópias e compartilhamento, através de *reference counting*. Tome cuidado com ciclos, pois podem acarretar vazamento de memória! Para isso, utilize std::weak_ptr ou cycles::relation_ptr (a seguir). Para utilizar, basta fazer #include <memory>. Exemplo:

```
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(); // apaga ponteiro s1 manualmente
println("*s2={} *s3={} ainda existem!", *s2, *s3);
```

Tipo std::function (avançado)

A estrutura std::function<tipo> permite armazenar funções, seja ela uma lambda sem captura (*captureless lambda*) ou uma lambda de captura, também chamada de *closure*. Uma *captureless lambda* pode decair para ponteiro de função, enquanto as demais só podem ser encapsuladas como std::function. Basta fazer #include <functional>. Exemplo:

```
// captureless lambda
int(*fquad1)(int) = [](int p) -> int { return p*p; };
std::function<int(int)> fquad2 = [](int p) { return p*p; };
// capturando variável x (por cópia)
int x = 10;
int y = 20;
// closure x1 (retorna x + 1)
std::function<int()> x1 = [x]() { return x+1; };
// capturando todas variáveis locais com =, y por referência
std::function<int()> fxy = [=, &y]() { y++; return x+y; };
int z = fxy(); // z==31  y==21
```

Dedução de `this` com C++23 (avançado)

Uma capacidade interessante do C++23 é o *deducing this*, que permite trabalhar com tipagem sobre a variável `this` em funções. Isso pode ser útil para capturar o `this` como referência, ao invés de ponteiro, e também para *nomear funções anônimas*.

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

Proposta para um std::scan (avançado/experimental)

Assim como o std::print (atualmente da fmt), existem propostas para um std::scan, atualmente no projeto scnlib de eliaskosunen.

A proposta experimental para o C++26 se chama P1729 "Text Parsing", e busca criar uma função scn::scan que substitua a scanf (pelo mesmo raciocínio empregado na abolição do printf). Exemplo:

```cpp
#include <scn/scn.h>
// lembre-se de incluir o pacote eliaskosunen/scnlib no CMake
using scn::scan;
// ...
int x = 0;
int y = 0;
auto resto = scan("10 20", "{}", x);
scan(resto, "{}", y);
print("x={} y={}", x, y);
// x=10 y=20
```

Ponteiro cycles::relation_ptr (avançado/experimental)

Uma proposta de ponteiro inteligente para resolver casos cíclicos foi criado pelo prof. Igor Machado Coelho, chamado `cycles::relation_ptr`. Este é um projeto interessante para compreender as limitações dos ponteiros inteligentes atuais, e o que pode ser possivelmente melhorado em um C++ futuro. Exemplo:

Para utilizar, basta fazer `#include <cycles/relation_ptr>`. Exemplo:

```
using cycles::relation_pool;
using cycles::relation_ptr;
// veja instruções em: https://github.com/igormcoelho/cycles
relation_pool<> grupo;
auto r1 = grupo.make<int>(10);
auto r2 = std::move(r1);
print("*r2={}\n",*r2);
// *r2=10
r2.reset(); // apaga ponteiro r2 manualmente
```

Discussão Rápida: C ou C++?

Citamos o comitê diretor do C++, "DIRECTION FOR ISO C++" (2022-10-15), de H. Hinnant, R. Orr, B. Stroustrup, D. Vandevoorde, M. Wong (página 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.

Discussão Rápida: C ou C++? (continuação)

Citamos o comitê diretor do C++, "DIRECTION FOR ISO C++" (2022-10-15), de H. Hinnant, R. Orr, B. Stroustrup, D. Vandevoorde, M. Wong (página 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.

Discussão Rápida: C ou C++? (resumo)

Em resumo: C++ moderno já é absolutamente superior a C em segurança e clareza, com desempenho equivalente, mas historicamente carece de boas estruturas para fazer o **básico** (como imprimir em tela, fazer vetores, etc), obrigando o uso de estruturas inseguras, como ponteiros. Então, as revisões recentes tem buscado esse fim, de facilitar o uso básico (como `std::print`, `std::array`, `std::string`, `std::vector`, smart pointers, ...) e evitar que a linguagem C seja necessária para a escrita de programas básicos.

Hoje (2023) ainda existem problemas, como:

- necessidade de usar bibliotecas externas (estamos precisando do `fmt::print` e `scn::scan`)
- necessidade de fazer `#include` em um código básico: a ideia é que, a partir da implementação de `import std` no C++23, será desnecessário incluir bibliotecas externas em códigos básicos :)

Muitos serão resolvidos na próxima edição do C++ (mas ainda faltará o `scn::scan`), sempre de olho em bons concorrentes modernos como Rust.

Section 11

Modularização e Testes (a revisar!!)

Motivação: Modularização e Testes

Qualquer programa complexo necessita de divisão em partes, ou módulos, para maior controle e verificação da corretude das operações.

Nesse curso, vamos utilizar um padrão mínimo de modularização, para que seja possível efetuar testes no código (de forma sistemática).

Modularização Básica

Um programa começa pelo seu "ponto de entrada" (ou *entrypoint*), tipicamente uma função `int main()`:

```
#include<iostream> // inclui arquivo externo
int main() {
   return 0;       // 0 significa: nenhum erro
}
```

A declaração de funções pode ser feita antes da definição:

```
int quadrado(int p); // declara a função 'quadrado'
int quadrado(int p) {
   return p*p;       // implementa a função 'quadrado'
}
```

Declarações vem em arquivos `.h`, enquanto as respectivas implementações em arquivo `.cpp` (ou juntas como `.hpp`).

Executando o `main.cpp`

Quando utilizando o GCC e um *entrypoint* no arquivo `main.cpp`:

Para compilar: `g++ -fconcepts -O3 main.cpp -o appMain`

Para executar código: `./appMain`

Importante: consideramos um sistema GNU/Linux, mas caso seja Windows pode-se usar o compilador C/C++ MinGW e executar o aplicativo gerado com uma extensão `.exe` (padrão executável Windows).

Organização em Arquivos I

Modularização mínima: 4 arquivos.

- um ponto de entrada (entrypoint) - geralmente `main.cpp` (dica: colocar na pasta `src/`)
- um (ou mais) arquivo(s) com demais módulos (dica: colocar na pasta `src/`)
- um (ou mais) arquivo(s) com seus testes - geralmente `teste.cpp` (dica: colocar na pasta `tests/`)
- um arquivo (na raiz) com informações de construção - geralmente `makefile` do GNU (com regras `all:` e `test:`)

Organização em Arquivos II

Também é informativo um arquivo extra na raiz com explicações sobre o código (tipicamente `README.md` na linguagem markdown)

Importante: o arquivo do *entrypoint* deverá conter exclusivamente a função `int main()` (e seus respectivos `#include`), para viabilizar testes de código.

Tipos na biblioteca padrão C++

Durante o curso estudaremos várias estruturas de dados, mas sempre que possível utilize as existentes na biblioteca padrão (STL). São "mais eficientes" e "à prova de erros".

Por exemplo, é fácil definir um tipo agregado `Par`, que comporta dois elementos internos (tipo genérico). Porém, é mais vantajoso usar o existente na STL, chamado `std::pair` (o prefixo `std::` é chamado *namespace* e evita colisões de nomes):

```
#include<iostream> // funções de entrada/saída
#include<tuple>    // agregados de par e tupla
int main() {
   std::pair<int, char> p {5, 'C'}; // direct init.
   printf("%d %c\n", p.first, p.second); // 5 C
   // ...
}
```

Relembrando (agregado Z)

```
// Em C++ (tipo agregado Z)
struct Z
{
   int x;
   // imprime campo x
   void imprimex() {
      printf("%d\n", this->x);
   }
};
```

Relembrando (conceito TemImprimeX)

```
template<typename Agregado>
concept bool
TemImprimeX = requires(Agregado a) {
   {
      a.imprimex()
   }
};
```

Verificações com assert

Durante o desenvolvimento, é útil verificar partes do código com testes simples e necessários para a corretude do mesmo (em tempo real). Para isso, podemos utilizar o `assert()`. Exemplo:

```
int x = 10;
x++;
assert(x == 11); // x deveria ser 11
```

Da mesma forma, podemos verificar tipos, especialmente *conceitos*, em tempo de compilação:

```
// verifica se tipo agregado Z tem método imprimex()
static_assert(TemImprimeX<Z>);
```

Testes com a biblioteca Catch2

Uma forma prática de testar um código modularizado com `main.cpp` separado do `resto.hpp`, é utilizando a biblioteca Catch2.

Basta criar um arquivo de teste, por exemplo, `teste.cpp`:

```
#include "resto.hpp"

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

TEST_CASE("Testa inicializacao do agregado Z")
{
   auto z1 = Z{.x = 10};
   // verifica se, de fato, z1.x vale 10
   REQUIRE(z1.x == 10);
}
```

Baixando o Catch2 e executando

Para baixar o arquivo `catch2.hpp`, basta acessar o site do projeto: github.com/catchorg/Catch2. Link direto (Agosto 2020):

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

Para compilar: `g++ -fconcepts teste.cpp -o appTestes`

Para executar testes: `./appTestes -d yes`

```
0.000 s: Testa inicializacao do agregado Z
===============================================
All tests passed (1 assertion in 1 test case)
```

Importante: Recomenda-se a opção `-fsanitize=address` e `-g3` para evitar bugs durante o desenvolvimento usando GCC.

Continue Aprendendo

Nessa revisão sobre tipos, buscamos não aprofundar em nenhuma característica "avançada" de C/C++, embora alguns conceitos possam parecer novos. Tópicos recomendados (não cobertos no curso):

- Orientação a Objetos (outras disciplinas cobrem esse tópico)
- uso frequente de *referências* (ao invés de ponteiros)
- uso frequente de *move semantics* (ao invés de referências)
- uso frequente de *closures* (ao invés de funções e lambdas)
- uso frequente de memórias auto-gerenciáveis, como `std::unique_ptr` e `std::shared_ptr` (não requer `delete`)
- uso de *corrotinas* do C++20 (somente consideramos *rotinas* no curso), especialmente para elaboração de iteradores infinitos
- teste de microbenchmarks (recomendamos a biblioteca Google Benchmark)

Bibliografia Recomendada

Além da bibliografia do curso, recomendamos (para esse tópico):

- Livro "Introdução a estruturas de dados" de W. Celes e J. L. Rangel
- Livro "The C++ Programming Language" de Bjarne Stroustrup
- Dicas e normas C++: https://github.com/isocpp/CppCoreGuidelines

Section 12

Agradecimentos

Pessoas

Em especial, agradeço aos colegas que elaboraram bons materiais, como o prof. Fabiano Oliveira (IME-UERJ), e o prof. Jayme Szwarcfiter cujos conceitos formam o cerne desses slides.

Estendo os agradecimentos aos demais colegas que colaboraram com a elaboração do material do curso de Pesquisa Operacional, que abriu caminho para verificação prática dessa tecnologia de slides.

Software

Esse material de curso só é possível graças aos inúmeros projetos de código-aberto que são necessários a ele, incluindo:

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

Empresas

Agradecimento especial a empresas que suportam projetos livres envolvidos nesse curso:

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

Reprodução do material

Esses slides foram escritos utilizando pandoc, segundo o tutorial ilectures:

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

Exceto expressamente mencionado (com as devidas ressalvas ao material cedido por colegas), a licença será Creative Commons.

Igor Machado Coelho

This Slide Is Intentionally Blank (for goomit-mpx)