Estruturas de Dados I

Revisão de Tipos e Módulos

Igor Machado Coelho

13/09/2020 - 03/04/2023

Section 1

Revisão de Tipos e Módulos

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 compiladores/IDEs ou uso de ferramentas de programação online

Ambiente de Programação

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 (nativamente) quanto para Windows (com a instalação do compilador MinGW).

Também é possível praticar diretamente em um navegador web com plataformas online: onlinegdb.com/online_c++_compiler. Neste caso, o aluno pode escolher o compilador de C ou da linguagem C++ (considerando padrão C++20).

Section 2

Tipos em C/C++

Conceitos de C/C++

Compreender a lógica da programação é a habilidade mais impor- tante 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 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.

```
int x = 5; // armazena o inteiro 5 na variável x
char y = 'A'; // armazena o caractere 'A' na variável y
float z = 3.7 ; // armazena o real 3.7 na variável z
```

Tipos de Variáveis

Pergunta/Resposta: Cuidado com tipos. Quais são os valores armazenados nas variáveis abaixo (C++)?

```
int     x1 = 5;          // => 5
int     x2 = x1 + 8;     // => 13
int     x3 = x2 / 2;     // => 6
float   x4 = x2 / 2;     // => 6.0
float   x5 = x2 / 2.0;   // => 6.5
auto    x6 = 13;         // => 13 (C warning: Wimplicit-int)
auto    x7 = x2 / 2;     // => ?  (C warning: Wimplicit-int)
auto    x8 = x2 / 2.0;   // => ?  (C warning: Wimplicit-int)
```

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=`).

```
int64_t x2  {20};  // long (ou long long)
int32_t x1  {10};  // int
int16_t x3  {30};  // short
int8_t  x4  {40};  // signed char
uint8_t x5  {50};  // unsigned char
std::byte b {60};  // unsigned char
```

Impressão de Saída Padrão

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

O C++20 traz o header `<format>`, que é suficiente para implementar o `print`, mas somente o C++23 traz o header `<print>` com método oficial `std::print`. Então utilizaremos o comando `fmt::print`, da biblioteca `<fmt/core.h>`, ao invés do `std::print`, ainda indisponível no C++20.

```
#include <fmt/core.h>
using fmt::print;

int main() {
    print("olá mundo!\n");
    return 0;
}
```

Impressão de Saída Padrão

Para imprimir na saída padrão utilizaremos o comando `fmt::print`. Este comando é dividido em duas partes, sendo que na primeira colocamos a mensagem formatada e, a seguir, colocamos as variáveis cujo conteúdo será impresso.

Pergunta: como podemos misturar um texto (também chamado de cadeia de caracteres ou string) com o conteúdo de variáveis?

Impressão de Saída Padrão

Para imprimir na saída padrão utilizaremos o comando `fmt::print`. Este comando é dividido em duas partes, sendo que na primeira colocamos a mensagem formatada e, a seguir, colocamos as variáveis cujo conteúdo será impresso.

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 `{}`.

```
int32_t x1 = 7;
print("x1 é {}", x1);  // x1 é 7
float x6 = x1 / 2.0;
print("metade de {} é {}", x1, x6);  // metade de 7 é 3.5
char b = 'L';
print("isto é uma {}etra", b);  // isto é uma Letra
print("Olá mundo! \n");  // Olá mundo! (quebra de linha)
```

Condicionais e Laços de Repetição

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

```
int x = 12;
if (x > 10)
    print("x maior de 10\n");
else
    print("x menor ou igual a 10\n")
```

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 (auto i=0; i < 10 ; i++) {
    print("i : {}\n" , i);
}
```

```
auto j=0;
while (j < 10) {
    print("j : {}\n", j);
    j++;
}
```

Tipos Compostos

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 vetores (arrays) ou agregados (structs, ...).

Tipos Compostos

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 vetores (arrays) ou agregados (structs, ...).

```
int32_t v[8]; // cria um vetor com 8 inteiros
v[0] = 3;     // atribui o valor 3 à primeira posição
v[7] = 5;     // atribui o valor 5 à última posição
```

Tipos Agregados I

Comparação C/C++:

```c
// Em C (tipo agregado P)
struct P
{
    int32_t 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)
class P
{
public:
    int32_t x;
    char y;
};
// declara variável tipo P
P p1;
// designated initializers
auto p2 = P{.x=10, .y='Y'};
```

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
print("{} {}\n", p1.x, p1.y);  // imprime '21 A'
```

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

Espaço de Memória

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:

```
int32_t 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).

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>
class G
{
public:
    T x;    // qual o tipo da variável x?
    char y;
};
// declara o agregado genérico G
G<float> g1 = {.x = 3.14, .y  = 'Y'};
G<int>   g2 = {.x = 3,    .y  = 'Y'};
```

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;
int y1 = (int) x;                  // em C
int y2 = int(x);                   // em C
int y3 = static_cast<int>(x); // em C++
const unsigned int z1 = x;         // OK
// unsigned int z2 = z1;           // ERRO
```

O `const` pode ser removido em algumas circustâncias através de um `const_cast`. Em C++, existe também o `constexpr`, que diferentemente do `const`, nunca pode ser removido, pois é de tempo de compilação. Em C, algo similar é possível com macros, mas permite reescrita, sendo inseguro.

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

Resumo até agora

Até agora, verificamos as seguinte estruturas:

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

Rotinas I

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'
int quadrado (int p) {
    return p*p;
}
// variável do tipo 'int', com valor 25
int x = quadrado(5);
```

Rotinas II

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

```
void imprime (int a , int b) {
    for (auto i=a ; i<b ; i++)
        print("{}\n", i) ;
}
```

Também é possível retornar múltiplos elementos (par ou tupla), através de um *structured binding* (requer `#include<tuple>`):

```
auto duplo(int p) {
    return std::make_tuple(p+3, p+6);
}
auto [x1,x2] = duplo(10); // x1=13 x2=16
```

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. **Exemplos:** `int* x; struct P* p1;`

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 {
    int32_t x;
    char y;
};

void imprimir(struct P* p1, struct P p2) {
    print("{} {}\n", p1->x, p2.x);
}
// ...
struct P p0 = {.x = 20, .y = 'Y'}; // cria variável 'p0'
imprimir(&p0, p0); // resulta em '20 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)
print("{}\n", vp->x);
// descarta a memória
free(vp);
```

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

Rotinas III

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

```
// o tipo da função 'quadrado' é: int(*)(int)
int quadrado(int p) {
    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) {
                                return p*p;
                            };
print("{}\n", quad(3)); // 9
```

Rotinas IV

A linguagem C++ permite a inclusão de funções e variáveis dentro de agregados (em C, funções devem ser externas). 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;
};

// imprime campo x
void imprimex(struct Z* this)
{
   print("{}\n", this->x);
}
```

```c
// Em C++ (tipo agregado Z)
class Z
{
public:
   int x;
   // imprime campo x
   void imprimex() {
      print("{}\n", this->x);
   }
};
```

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`.

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

```
// Aloca (C++) o 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`.

```
// 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");
```

```
// 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 = std::nullptr;
```

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* `TemImprimeX`, que exige que o agregado possua um método `imprimex()`:

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

Conceitos II

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

```
auto a1                 = Z{.x = 1};   // tipo automático
TemImprimeX auto a2 = Z{.x = 2};   // tipo conceitual
Z a3                    = Z{.x = 3};   // tipo explícito
```

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

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

Passagem de Parâmetros por Referência II

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

```cpp
// C++
void imprimex(P& vp) {
  // imprime x (valor 10)
  print("{}\n", 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.

Section 3

Uso da biblioteca padrão

O que é biblioteca padrão?

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. Atualmente, é necessário utilizar `#include<...>` para incluir esses componentes, mas em um futuro próximo (C++23) será possível através de `import std`, utilizando a estrutura moderna dos CXX Modules.

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

Veremos rapidamente exemplos de estruturas muito fundamentais como: `std::string`, `std::array` e `std::vector`.

Tipo std::string

O tipo `std::string` representa cadeias de caracteres, chamadas de *strings*. Ela substitui a necessidade de `char*`, `char[]` ou `const char*` em C. Para utilizar, basta fazer `#include <string>`. Exemplo:

```
std::string s1 = "abcd";
std::string s2 = "ef";
print("tamanho1={} tamanho2={}\n", s1.length(), s2.length());
// tamanho1=4 tamanho2=2
s1 = s1 + s2;
print("s1={} s2={}\n", s1, s2);
// s1=abcdef s2=ef
const char* cs = s1.c_str();
print("s1={} cs={}\n", s1, cs);
// s1=abcdef cs=abcdef
```

Tipo std::array

Assim como vetores nativos, exemplo `int[]`, o agregado `std::array<tipo, tamanho>` permite representar vetores de tamanho fixo. Para utilizar, basta fazer `#include <array>`. 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
```

Tipo std::vector

A popular estrutura `std::vector<tipo>` permite representar vetores com tamanho variável (através do método `push_back`). Para utilizar, basta fazer `#include <vector>`. 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::optional

O `std::optional<tipo>` representa um valor opcional, com alocação em *stack*, não em *heap* como smart pointers. Para utilizar, basta fazer `#include <optional>`. Exemplo:

```
std::optional<int> busca(char c, const std::vector<char>& v) {
  // busca char 'c' num vetor v e retorna posição
  for(int i=0; i<static_cast<int>(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) print("posicao={}", *op);
else   print("não encontrou");
```

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. Para utilizar, basta fazer #include <memory>. Exemplo:

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

auto u1 = std::make_unique<int>(10);
auto u2 = std::move(u1);
auto* p3 = u2.get();
print("*u2={} *p3={}\n", *u2, *p3);
// *u2=10 *p3=10
u2.reset(); // apaga ponteiro u2 manualmente
```

Tipo std::shared_ptr

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();
print("*s1={} *s2={} *s3={}\n", *s1, *s2, *s3);
// *s1=10 *s2=10 *s3=10
s1.reset(); // apaga ponteiro s1 manualmente
print("*s2={} *s3={} ainda existem!\n", *s2, *s3);
```

Section 4

Bibliotecas experimentais

Proposta para um std::scan

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:

```
#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

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
```

Section 5

C ou C++?

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 6

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)
class Z
{
public:
    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 7

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)