Estruturas de Dados I

Igor M. Coelho

30/08/2020

Section 1

Revisão/Tipos

Linguagem Adotada

O conteúdo do curso será focado na análise de algoritmos, então as técnicas apresentadas independem da linguagem de programação adotada (valendo inclusive para pseudocódigo).

Porém, para fornecer um conteúdo dinâmico que pode ser testado e experimentado pelos alunos em uma das linguagens mais utilizadas atualmente no mundo, adotaremos as linguagens C/C++ (ambas serão aceitas).

A linguagem C++ é uma extensão da linguagem C, adicionando dois principais recursos interessantes: orientação a objetos e programação genérica. *Não entraremos com profundidade em nenhum destes dois assuntos, mas não se preocupe! Eles serão aprofundados em disciplinas futuras.* Somente recursos básicos das linguagens C/C++ serão utilizados neste curso (e iremos revisá-los em breve!).

Ambiente de Programação

De forma a praticar o conteúdo do curso, podem ser utilizadas diversas ferramentas para Linux (preferencialmente) ou Windows. A IDE Eclipse 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++17).

Section 2

Conceitos Básicos de C/C++

Conceitos de 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 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).

Impressão de Saída Padrão

Para imprimir na saída padrão utilizaremos o comando printf. 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 printf. 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 dos padrões de formatação (%d, %f, %c, . . .).

```
int x1 = 7;
printf("x1 é %d", x1); // x1 é 7
float x6 = x1 / 2.0;
printf("metade de %d é %f", x1, x6); // metade de 7 é 3.5
char b = 'L';
printf("isto é uma %cetra", b) ; // isto é uma Letra
printf("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)
   printf("x maior de 10\n");
else
   printf("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 (int i=0; i < 10 ; i++) {
   printf("i : %d\n" , i);
}
```

```
int j=0;
while (j < 10) {
   printf("j : %d\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, ...).

```
int 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
{
    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)
class P
{
public:
    int 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'}; // compilador GCC8 (no mínimo)

p1.x = 20;                  // atribui 20 à variável x de p1
p1.x = p1.x + 1;            // incrementa a variável y de p1
printf("%d %c\n", p1.x, p1.y);  // imprime '21 A'
```

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:

```
int v [200];
char x [1000];
float y [5];
```

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:

```
int v [200];
char x [1000];
float y [5];
```

```
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)

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'};
```

Modularização: 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);
```

Modularização: 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 (int i=a ; i<b ; i++)
      printf("%d\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. Mas como passar tipos complexos (como estruturas e vetores de milhares de elementos) sem desperdiçar 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 {
    int x;
    char y;
};

void imprimir(struct P* p1, struct P p2) {
    printf("%d %d\n", p1->x, p2.x);
}
// ...
struct P p0 = {.x = 20, .y = 'Y'}; // cria variável 'p0'
imprimir(&p0, p0); // resulta em '20 20'
```

Modularização: 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;
                               };
// ...
printf("%d\n", quad(3));     // 9
```

Modularização: 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 P)
struct P {
    int x;
};

// imprime campo x
void imprimex(struct P* this)
{
   printf("%d\n", this->x);
}
```

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

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`).

Organização em Arquivos

Modularização mínima: 4 arquivos.

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

Também é informativo um arquivo extra na raiz com explicações sobre o código (geralmente `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
   // ...
}
```

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 um 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);
```

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

Continue Aprendendo

Nessa revisão sobre tipos, buscamos não aprofundar em nenhuma característica "avançada" de C/C++, embora alguns conceitos possam ser novos para alguns.

Para uma programação mais avançada em C++ é recomendado (tópicos não cobertos nesse 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 de *corrotinas* do C++20 (somente consideramos *rotinas* no curso), especialmente para elaboração de iteradores infinitos
- teste unitário de cada componente desenvolvido (recomendamos a biblioteca `catch2.hpp` ou Google Tests, para esse fim)

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 3

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)
- ...

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/

Igor Machado Coelho

This Slide Is Intentionally Blank (for goomit-mpx)