Revisão Final

Tópicos abordados:

- **Linguagem C++**
- **Variáveis e Ponteiros**
- **Passagem por valor e passagem por referência**
- **Alocação de memória**

Introdução:

O presente texto aborda alguns tópicos da disciplina Estrutura de Dados. A maioria dos tópicos abordados nesta revisão está baseada na consulta realizada junto aos alunos para levantamento de dúvidas. As principais dúvidas estavam focadas em uma introdução à linguagem C++ e conceitos básicos de programação, como variáveis e ponteiros, passagem por valor e passagem por referência em funções, e alocação de memória. Tais assuntos foram cobertos nas seções de 1 a 4. Apesar de não reportado nas dúvidas enviadas, também foram cobertos assuntos específicos da disciplina, como tipo abstrato de dados e listas e pilhas nas seções 5 e 6, respectivamente. Para cada tópico da revisão, uma recomendação de leitura é feita, visando complementar os estudos para realização da prova. A recomendação de leitura também contextualiza os trechos transcritos nesta revisão.

1. Linguagem C++ -

Recomendação de leitura:
Material de apoio - C++ Programming Tutorial | Chua Hock Chuan
Material de apoio - C++: como programar (Leia os capítulos 1 a 7) | Harvey M. Deitel e Paul J. C. Deitel
Videoaula 1 - Introdução ao C++

A linguagem C++ caracteriza-se por ser compilada, apresentar tipagem estática e manipular explicitamente a memória. Dessa forma, programas gerados usando C++ são mais rápidos. Relambrando alguns aspectos de sintaxe da linguagem C++:

- Comandos cin e cout para entrada e saída:

```
std::cout << "Digite o primeiro número: ";
std::cin >> number1;
std::cout << "Digite o segundo número: ";
std::cin >> number2;
```

- Operações aritméticas:

```
int   sum  = number1 + number2;
int   sub  = number1 - number2;
int   mul  = number1 * number2;
int   div  = number1 / number2;
float fdiv = (float)number1 / (float)number2;
int   res  = number1 % number2;
```

- Estruturas condicionais:

```
if (is_true())
{
    cout << "b is true!\n";  // executed
}
else
{
    cout << "b is false!\n";
}
```

- Estruturas de repetição:

```
for (int number = 2; number <= 20; number += 2)
  total += number;
```

```
while (counter < 10) {

  cout << "Digite um número (" << counter << ")" << endl;
  std::cin >> number1;

  if (number1 < 5) {
    amount++;
  }
  counter++;
}
```

```
do {
  cout << "Insira um novo número: " << endl;
  cin >> number;

  sum += number;
  count++;
}  while (number != 0);
```

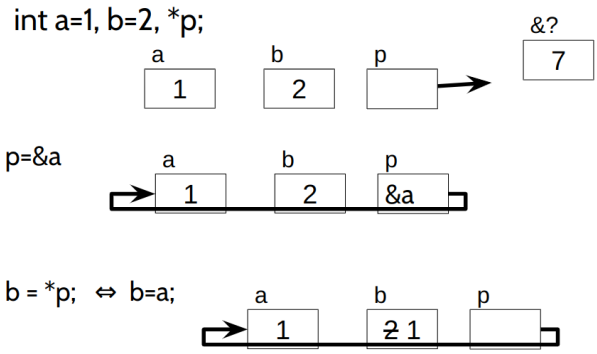
2. Variáveis e ponteiros

Recomendação de leitura:
Material de apoio - C++ Programming Tutorial | Chua Hock Chuan
Material de apoio - C++: como programar (Leia os capítulos 1 a 7) | Harvey M. Deitel e Paul J. C. Deitel
Videoaula 1 - Introdução ao C++

Uma variável armazena na memória um valor de um determinado tipo de dado, sendo reconhecia por um nome e um tipo associado. O nome da variável identifica e a referencia exclusivamente, permitindo recuperar o valor armazenado (referenciado) por ela na memória do computador. Os tipos em linguagem C++ são int para inteiros, double para ponto flutuante ou reais, char para caracters, bool para boleanos (verdadeiro ou falso). Durante a execução do programa, os dados estão sendo manipulados, logo, para que o computador não esqueça das informações contidas em um dado, é necessário guardá-las em sua memória. As variáveis guardam informações sobre os dados (o seu conteúdo) que estão sendo manipulados. O conteúdo da variável é substituído por outro que lhe será atribuído.

O uso de uma variável em uma expressão representa o seu conteúdo naquele momento. O uso não muda o seu conteúdo. Uma variável é armazenada em um certo número de bytes em uma determinada posição de memória. Um ponteiro é uma variável que contém o endereço de outra variável. Os ponteiros são usados para

acessar e manipular conteúdos em determinado endereço de memória conforme exemplo abaixo.



3. **Passagem por valor e passagem por referência**

   Recomendação de leitura:
   Material de apoio - C++ Programming Tutorial | Chua Hock Chuan
   Material de apoio - C++: como programar (Leia os capítulos 1 a 7) | Harvey M. Deitel e Paul J. C. Deitel
   Videoaula 1 - Introdução ao C++

Uma função numa linguagem de programação recebe e retorna valores (argumentos). Esses valores podem estar armazenados em variáveis e há duas formas de serem repassados às funções: passagem por valor e passagem por referência. Na passagem por valor, uma cópia da variável inteira é instanciada no escopo da função. Logo, a variável na função que recebe tal cópia do valor poderá usar e alterar tal valor copiado sem alterar a variável a partir do qual tal valor foi copiado.

```
#include <iostream>
using namespace std;

void troca_por_valor(int a, int b){
        int temp;
        temp=a;
        a=b;
        b=temp;
}
int main(){
        int a=2, b=3;
        cout<<"Antes: a = "<<a<<" b = " << b<<endl;
        troca_por_valor(a,b);
        cout<<"Depois: a = "<<a<<" b = "<< b << endl;
        return 0;
}
```

```
codigos$ g++ ex8_parametro_por_valor.cpp -o valor
codigos$ ./valor
Antes: a = 2 b = 3
Depois: a = 2 b = 3
```

Na passagem por referência, apenas o endereço da variável é repassado a uma variável da função. Logo, tal variável é um ponteiro que armazena o endereço da variável original. Assim, a função, ao alterar o valor da variável passada por referência, altera o valor naquele endereço, ou seja, na variável original. Na passagem por referência, os argumentos precisam ser declarados como tipos ponteiros para permitir o acesso e alterar as variáveis fora da função.

```
#include <iostream>
using namespace std;

void troca_por_referencia(int &a, int &b){
        int temp;
        temp=a;
        a=b;
        b=temp;
}
int main(){
        int a=2, b=3;
        cout<<"Antes: a = "<<a<<" b = " << b<<endl;
        troca_por_referencia(a,b);
        cout<<"Depois: a = "<<a<<" b = "<< b << endl;
        return 0;
}
```

```
codigos$ g++ ex9_parametro_por_referencia.cpp -o referencia
codigos$ ./referencia
Antes: a = 2 b = 3
Depois: a = 3 b = 2
```

Resumindo, na passagem por valor é feita uma cópia do argumento (ou variável), podendo ser alterado pela função sem alterar a variável original com aquele valor. Na passagem de parâmetros por referência em funções, manipula-se o endereço de memória da variável fornecida à função, usando ponteiro, e não uma cópia do valor da variável. Por isso, alterações realizadas pela função são efetivadas naquele

endereço de memória, fazendo com que as alterações permaneçam quando a função termina sua execução.

4. **Alocação de memória em C++**

   Recomendação de leitura:
   Material de apoio - C++ Programming Tutorial | Chua Hock Chuan
   Material de apoio - C++: como programar (Leia os capítulos 1 a 7) | Harvey M. Deitel e Paul J. C. Deitel

A linguagem C++ aloca memória em tempo de execução, ou seja, permite a chamada alocação dinâmica de memória. Isso é feito através dos operadores new para alocar e delete para desalocar. Lembre-se de que a memória deve ser desalocada manualmente na linguagem C++, diferente de Java ou Python em que há um gerenciamento automático de memória alocada. A sintaxe para alocação dinâmica de memória é <nomeVariávelPonteiro> = new <tipo_de_dado>. A sintaxe para desalocação de memória é delete <nomeVariávelPonteiro>. O trecho de código abaixo exemplifica o usou de new e delete.

```
int* x; // declara um ponteiro int

x = new int; // aloca memória dinamicamente

*x = 45; // atribui valor a memoria alocada
```

Até aqui, alocamos memória dinamicamente para a variável int x com new. O ponteiro x é declarado para a alocação da memória de forma dinâmica uma vez que o operador new retorna um endereço de memória. Se estivéssemos usando um vetor, o operador new retorna o endereço do primeiro elemento do vetor. Agora, utilizamos abaixo o operador delete para descartar a memória alocada ao encerrar o uso da variável alocada no código. Essa memória fica liberada para uso do sistema operacional.

```
delete x; //desalocação de memória
```

5. Tipo Abstrato de Dados

Recomendação de leitura:
Texto-base - Estruturas de Dados (Leia as páginas 34 e 35) | Nina Edelweiss e Renata Galante

Tipo Abstrato de Dados (TAD) trata-se de uma especificação para dados, assim como para operações executadas nesses mesmos dados. Logo, há uma separação ou divisão de um TAD em dados e operações. A ideia de ser abstrato está no fato de que se pode utilizar um TAD sem detalhes de sua utilização, ou seja, abstraindo no seu uso os detalhes de implementação. Logo, TADs podem representar ou estabelecer tipos de dados que não foram previamente definidos em linguagens de programação.

O TAD pode ser definido de forma mais elaborada como um par (v,o), v para valores e o para operações executadas sobre tais valores. Por exemplo, podemos ter o TAD (v,o) para números complexos, em que:

- v = parte real, parte imaginária.
- o = operações de inicialização, soma, produto etc.

Usando a notação do texto-base, temos para valores:

```
Complexo = Registro
             re: real
             img: real
           Fim registro
```

Associamos as operações:

- Função iniciaComplexo
  Entrada: x, y (real)
  Saída: c (complexo)
  Recebe dois valores reais e retorna um número c Complexo, em que os valores x e y são atribuídos para re e img, respectivamente, em c do tipo Complexo.

- Função soma
  Entrada: c1, c2 (complexo)
  Saída: c (complexo)
  Recebe valores armazenados em variáveis do tipo Complexo e retorna um número c Complexo, em que re e im recebem a soma da parte real e imaginária de c1 e c2, respectivamente.

Define-se a estrutura de dados e as operações aplicáveis de um TAD criado, no qual os detalhes de implementação fica restrito às operações. O usuário do TAD conhece a especificação da estrutura de dados e das operações, mas não precisa saber detalhes de implementação.

Resumindo, a especificação das operações em um TAD e os valores sobre as quais atuam definem um tipo abstrato, a abstração em um TAD. A implementação efetiva do conjunto de operações define o chamado tipo concreto. As vantagens no uso de TADs são (1) uso em diferentes contextos ou aplicações no desenvolvimento de códigos; (2) mudar o tipo abstrato sem precisar alterar todas as aplicações usuais desse tipo.

6. Listas e Pilhas

Recomendação de leitura:
Material de apoio - Estruturas de Dados: Algoritmos, Análise da Complexidade e Implementações em Java e C/C++ (Leia o cap. 4, p. 183-185) | Ana Fernanda Gomes Ascencio e Graziela Santos Araújo
Videoaula 7 - Listas Encadeadas

As listas lineares são estrutura de dados nas quais cada elemento possui um predecessor e um sucessor. As únicas exceções são o primeiro e o último elementos, que não possuem predecessor e sucessor, respectivamente. Esse tipo de estrutura estabelece uma ordem entre os elementos através dos sucessores e predecessores. Uma ordem pode ser dada pela inclusão de um elemento. Tanto as pilhas quanto as filas são exemplos de listas lineares. As chamadas listas lineares sequenciais se

caracterizam pelo fato de os elementos estarem em posições contíguas de memória. Logo, a ordem física e a ordem lógica dos elementos são iguais. Um vetor é um exemplo. A lista linear sequencial permite acesso em tempo constante a um elemento, através do índice que localiza a posição de memória do elemento. Assim, por exemplo, podemos ordenar um vetor usando os algoritmos de busca binária em O(log(n)). A desvantagem está na alocação de espaço em memória suficiente para todos os elementos, uma vez que há um custo computacional em mover ou deslocar elementos para uma nova posição de memória caso falte algum espaço na alocação. A inserção de ou a remoção de um elemento exigiria deslocamentos para reordenar os elementos em um vetor ordenado.

Nas chamadas listas encadeadas, a ordem lógica dos elementos não segue necessariamente a ordem física. As relações de precedência e sucessão entre os elementos é estabelecida em função dos endereços de memória para manter a ordem lógica. Assim, podemos ter elementos distribuídos pela memória, otimizando seu uso na distribuição dos elementos. A desvantagem está no acesso aos elementos que deixa de ser em tempo constante. Porém, temos como vantagem o aumento ou a redução no número de elementos durante a execução sem comprometer a ordem lógica.

A pilha é uma estrutura linear em que o primeiro elemento a entrar será o último a sair, enquanto o último elemento a entrar será o primeiro a sair. Logo, as remoções e inserções nessa estrutura de dados ocorrem sempre no topo. Essa estrutura é relevante para garantir alinhamento de componentes em processos, sendo exemplos do seu uso mecanismos de fazer e desfazer nos editores de texto, navegação entre páginas web, as funções recursivas usada em compiladores; verificação de alinhamento de parênteses em strings, dentre outros.

```
class Stack {
 public:
  Stack();   // Construtor
  ~Stack();   // Destrutor
  bool isEmpty() const;
  bool isFull() const;
  void print() const;

  void push(ItemType);
  ItemType pop();
 private:
  int length;
  ItemType* structure;
};
```