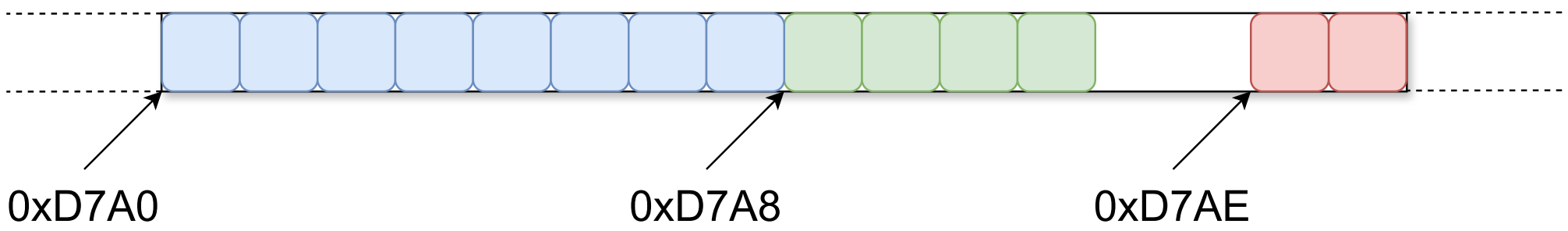
Algoritmos e Programação II

https://evandro-crr.github.io/alg2

Endereço de Memória

```cpp
int main() {
    short var1;
    int var2;
    double var3;

    std::cout << &var1 << "\n"
              << &var2 << "\n"
              << &var3 << "\n";

    return 0;
}
```

- Podemos usar o operador `&` para retornar o endereço de memória de uma variável.
- O valor de toda variável está armazenado na memória.
- Cada byte da memória possui um endereço único.

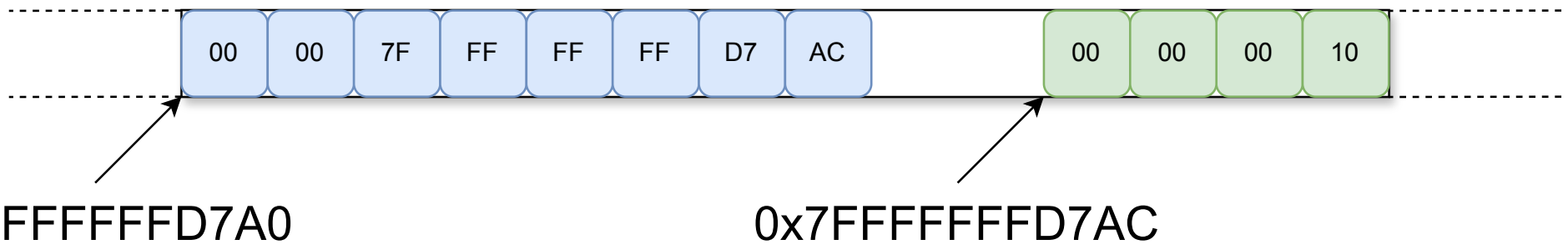


Ponteiros

Variáveis que armazenam a posição de memória de outra

```
int main() {
    int numero = 10;
    int *ptr = nullptr;

    ptr = &numero;

    cout << "O valor de numero é "
         << numero << "\n";
    cout << "O endereço de numero é "
         << ptr << "\n";
    cout << "O endereço de ponteiro é "
         << &ptr << "\n";
}
```

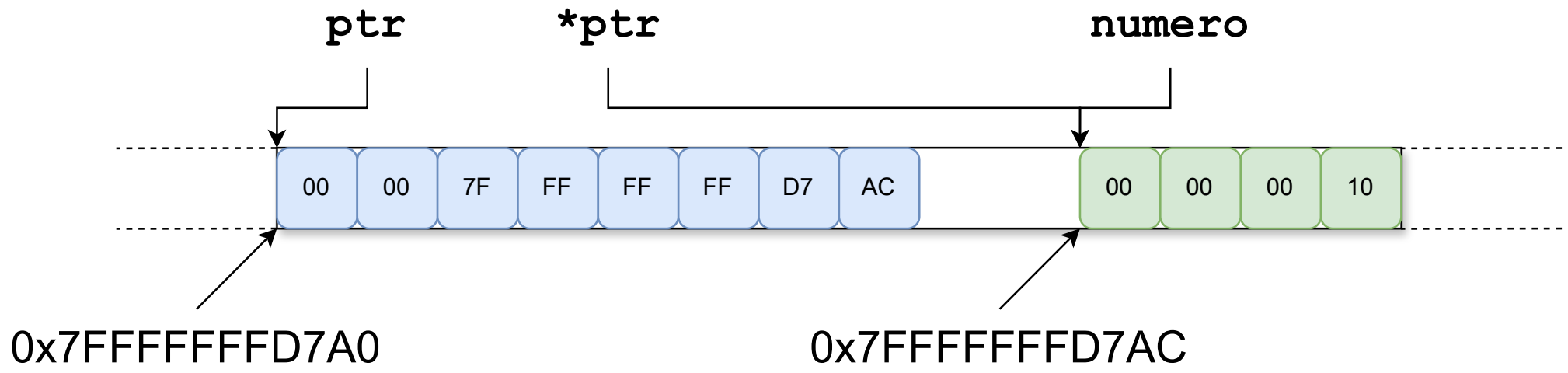
- `ptr` é um ponteiro e aponta para um valor do tipo `int`.
- `nullptr` é usado para inicializar o ponteiro com valor nulo.
- ⚠️ Inicializar o ponteiro não é obrigatório, mas pode facilitar a detecção de erros.



Operador de Indireção (*)

```
int main() {
    int numero = 10, *ptr = &numero;
    *ptr = *ptr + 20;
    std::cout << "O valor armazenado em "
              << ptr << " é " << *ptr << "\n";
    std::cout << "O valor de numero é "
              << numero << "\n";
}
```

- `*ptr` refere-se ao valor armazenado na posição de memória apontada por `ptr`.
- A operação `*ptr` é chamada de desreferenciação.



Já Vimos 3 Maneiras Diferentes de Usar o `*`

- Operador de multiplicação

```
area = altura * largura;
```

- Definição de variável do tipo ponteiro

```
int *ptr = nullptr;
```

- Operador de indireção

```
*ptr = 100;
```

Ponteiro como Argumento de Função

```
void ler_numero_0_100(int *var) {
    do {
        cout << "Número de 0 a 100: ";
        cin >> *var;
    } while (*var < 0 || *var > 100);
}

int main() {
    int num1, num2;
    ler_numero_0_100(&num1);
    ler_numero_0_100(&num2);
    cout << "Os números são: "
         << num1 << " "
         << num2 << "\n";
}
```

- Comportamento similar à passagem de argumento por referência.
- A passagem por referência é mais prática.
- A linguagem **C** não oferece passagem de argumento por referência.

Relação Entre Array e Ponteiro

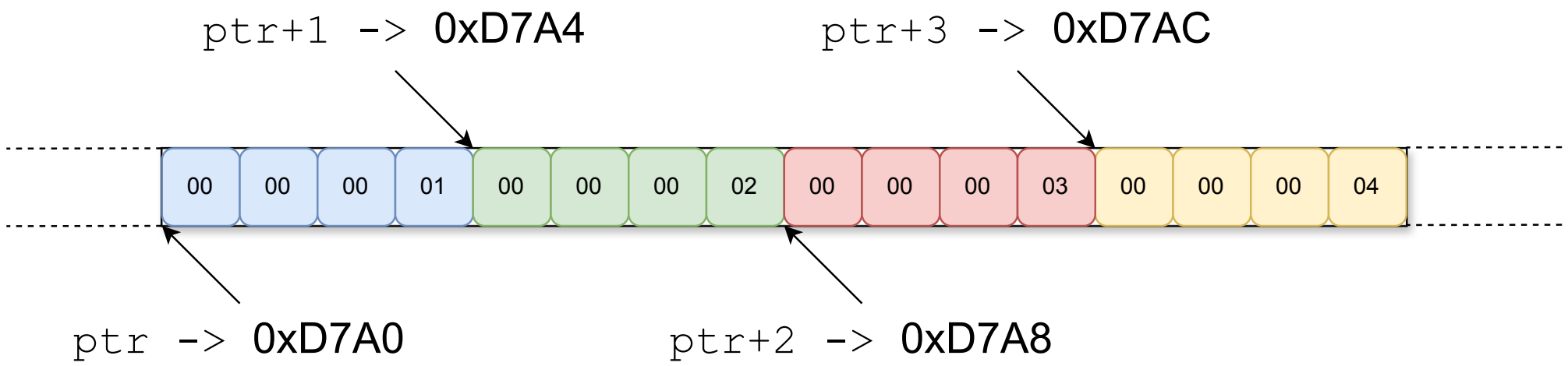
```
int main() {
    int numeros[4] = {1, 2, 3, 4};
    int *ptr = numeros;

    for (int i = 0; i < 4; i++)
        std::cout << ptr[i] << " ";

    std::cout << "\n";
}
```

- Variáveis de array podem ser atribuídas a ponteiros.
- Ponteiros podem ser tratados como arrays.

Aritmética de Ponteiros

```
int main() {
    int numeros[4] = {1, 2, 3, 4};
    int *ptr = numeros;
    for (int i = 0; i < 4; i++)
        std::cout << *(ptr + i) << " ";
    std::cout << "\n";
}
```

- `*(ptr + i)` é equivalente a `ptr[i]`.
- `ptr + i` retorna `ptr + i * sizeof(int)`.



⚙ Exercícios

1. Qual é a saída do programa?

```
int main() {
    int x = 50, y = 60, z = 70;
    int *ptr = nullptr;
    cout << x << " " << y << " " << z << "\n";
    ptr = &x;
    *ptr *= 10;
    ptr = &y;
    *ptr *= 5;
    ptr = &z;
    *ptr *= 2;
    cout << x << " " << y << " " << z << "\n";
}
```

2. Reescreva o código usando aritmética de ponteiro

```
for (int x = 0; x < 100; x++)
    std::cout << arr[x] << std::endl;
```

3. Quais definições são válidas?

1.
```
int ivar;
int *iptr = &ivar;
```

2.
```
int ivar, *iptr = &ivar;
```

3.
```
float fvar;
int *iptr = &fvar;
```

4.
```
int nums[50], *iptr = nums;
```

5.
```
int *iptr = &ivar;
int ivar;
```

Alocação Dinâmica de Memória

```
int menor_valor(const int*, int);
void print_array(const int*, int);

void ordenar(int lista[], int tamanho) {
    int *tmp = new int[tamanho];
    for (int i = 0; i < tamanho; i++)
        tmp[i] = lista[i];
    for (int i = 0; i < tamanho; i++) {
        int min_i = menor_valor(tmp, tamanho);
        lista[i] = tmp[min_i];
        *(tmp + min_i) = INT_MAX;
    }
    delete[] tmp;
}

int main() {
    int lista[] = {10, 2, 5, 44, 23};
    const int tamanho =
        sizeof(lista) / sizeof(int);
    print_array(lista, tamanho);
    ordenar(lista, tamanho);
    print_array(lista, tamanho);
}
```

- Nem sempre é possível saber a quantidade de memória necessária em tempo de compilação.
- Podemos usar a instrução `new` para alocar memória.
- É necessário usar a instrução `delete` para liberar a memória.

Vazamento de Memória

Memory leak

⛔ Este código vai travar o PC ⛔

```
void func() {
    int *ptr = new int;
}

int main() {
    for (;;) {
        func();
    }
}
```

- Toda memória alocada com `new` **DEVE** ser desalocada com `delete`.
- Se alocarmos memória sem desalocar, eventualmente, faltará memória.

Memory leak é um bug comum em diversos softwares, incluindo jogos. Existem ferramentas para detectar memory leaks e o C++ oferece maneiras de evitar esse problema.

Ponteiro para Estrutura

```
struct Pessoa {
    string nome;
    int idade;
};

int main() {
    Pessoa pessoa = {"Luiza", 32};
    Pessoa *ptr = &pessoa;

    cout << ptr->nome << "\n"
         << (*ptr).idade << "\n";
}
```

Podemos acessar os membros de um ponteiro para `struct`:

1. Usando o operador de ponteiro `->`.
2. Desreferenciando o ponteiro seguido do operador ponto.

O operador ponto (`.`) tem maior precedência do que o operador de indireção (`*`).

Alocação dinâmica de Estrutura

```
int main() {
    Pessoa *pessoa1 = new Pessoa;
    Pessoa *pessoa2 = new Pessoa{"Luiza", 20};

    cout << pessoa1->nome << "\n"
         << (*pessoa1).idade << "\n";
    cout << pessoa2->nome << "\n"
         << (*pessoa2).idade << "\n";

    delete pessoa1;
    delete pessoa2;
}
```

- O operador `new` sempre inicializa a memória.
- É possível passar uma lista de inicialização ao alocar a estrutura dinamicamente.

Função que retorna ponteiro

Uma função pode retornar um ponteiro,
mas é preciso ter cuidado com os seguintes casos:

✅ Casos válidos

- Retornar um ponteiro passado como argumento.
- Retornar um ponteiro para memória alocada dinamicamente dentro da função.

❌ Casos inválidos

- Retornar a referência de variáveis locais.
- Retornar um ponteiro para uma memória que já foi desalocada.

Função que retorna ponteiro

```
struct Pessoa {
    string nome;
    int idade;
};

Pessoa *mais_velho(Pessoa *a, Pessoa *b) {
    if (a->idade > b->idade) {
        return a;
    }
    return b;
}

int main() {
    Pessoa leo = {"Leo", 32};
    Pessoa bob = {"Bob", 19};

    Pessoa *pessoa = mais_velho(&leo, &bob);

    cout << pessoa->nome << " é mais velho\n";
}
```

Receber e retornar um ponteiro pode evitar a necessidade de novas alocações de memória, otimizando o uso de recursos.

Função que retorna ponteiro

```
int *clonar_lista(const int lista[], int tamanho) {
    int *nova_lista = new int[tamanho];
    for (int i = 0; i < tamanho; i++) {
        nova_lista[i] = lista[i];
    }
    return nova_lista;
}

int main() {
    int numeros[5] = {9, 8, 7, 6};
    int *clone = clonar_lista(numeros, 5);
    for (int i = 0; i < 5; i++) {
        cout << i << ":" << clone[i] << ", ";
    }
    cout << "\n";

    delete[] clone;
}
```

Funções podem retornar memória alocada dinamicamente. É importante garantir que essa memória seja desalocada posteriormente.

Como NÃO retornar ponteiro ❌

```
struct Pessoa {
    string nome;
    int idade;
};

Pessoa *mais_velho(Pessoa &a, Pessoa &b) {
    Pessoa p;
    if (a.idade > b.idade) {
        p = a;
    }
    p = b;
    return &p;
}

int main() {
    Pessoa leo = {"Leo", 32};
    Pessoa bob = {"Bob", 19};

    Pessoa *pessoa = mais_velho(leo, bob);

    cout << pessoa->nome << " é mais velho\n";
}
```

Não retorne o endereço de memória de variáveis locais, pois elas são destruídas quando a função retorna. O compilador geralmente alerta sobre esse problema.

Como NÃO retornar ponteiro ❌

```
int *clonar_lista(const int lista[], int tamanho) {
    int *nova_lista = new int[tamanho];
    for (int i = 0; i < tamanho; i++) {
        nova_lista[i] = lista[i];
    }
    delete[] nova_lista;
    return nova_lista;
}

int main() {
    int numeros[5] = {9, 8, 7, 6};
    int *clone = clonar_lista(numeros, 5);
    for (int i = 0; i < 5; i++) {
        cout << i << ":" << clone[i] << ", ";
    }
    cout << "\n";

    delete[] clone;
}
```

Não retorne um ponteiro para um endereço de memória que já foi desalocado. Não é possível desalocar a mesma posição de memória mais de uma vez.

⚙️ Exercício (continua ➡️)

Desenvolva um sistema para gerenciar o cadastro de produtos.

Informações do produto

- Código do produto
- Nome do produto
- Quantidade em estoque
- Preço unitário

1. Crie uma estrutura para armazenar as informações.
2. Crie uma função para receber os dados do produto.
3. Crie uma função que exiba as informações de um produto.
4. Crie uma lista usando alocação dinâmica e exiba as informações dos produtos.

Código Base:

```cpp
#include <iostream>

struct Produto;

Produto registrar_produto();
void imprimir_relatorio(...);

int main() {
    int quantidade = 3;
    Produto *produtos;
    // alocar produtos

    // Registre os produtos
    for (int i = 0; i < quantidade; i++) {
        ...
    }

    std::cout << "Relatório de Produtos\n";
    for (int i = 0; i < quantidade; i++) {
        imprimir_relatorio(...);
    }

    // desalocar produtos
    return 0;
}
```

Exemplo de Saída Esperada:

```
Código do produto: 101
Nome do produto: Caneta Azul
Quantidade em estoque: 200
Preço unitário: 1.50

Código do produto: 102
Nome do produto: Caderno
Quantidade em estoque: 150
Preço unitário: 5.75

...

Relatório de Produtos:
Código: 101 | Nome: Caneta Azul | Quantidade: 200 | Preço: 1.5 | Valor Total: 300
Código: 102 | Nome: Caderno | Quantidade: 150 | Preço: 5.75 | Valor Total: 862.5
...
```

Algoritmos e Programação II

https://evandro-crr.github.io/alg2

Evandro Chagas Ribeiro da Rosa