POBI - Estruturas de Decisão e Loops - 08/04/24

Estruturas de decisão

If, Else if e Else

`if`

A estrutura `if` é uma instrução condicional que avalia uma expressão booleana. Se a expressão for verdadeira, o bloco de código dentro do `if` é executado. Caso contrário, o bloco é ignorado.

A sintaxe geral é:

```
if (condição) {
    // Bloco de código a ser executado se a condição for verd
}
```

- `condição`: Uma expressão que é avaliada como verdadeira (`true`) ou falsa (`false`).

Exemplo:

```
int x = 10;
if (x > 5) {
    cout << "x é maior do que 5." << endl;
}
```

`else if`

A estrutura `else if` é usada para avaliar múltiplas condições em sequência, depois de uma condição `if` inicial. Cada `else if` é avaliado somente se as condições anteriores forem falsas.

A sintaxe é:

```
if (condição1) {
    // Bloco de código se condição1 for verdadeira
} else if (condição2) {
    // Bloco de código se condição2 for verdadeira
} else {
    // Bloco de código se nenhuma das condições anteriores fo
}
```

- `condição1`, `condição2`, etc.: Expressões booleanas a serem avaliadas em ordem.

Exemplo:

```
int idade = 20;
if (idade >= 18) {
    cout << "Você é maior de idade." << endl;
} else if (idade >= 13) {
    cout << "Você é adolescente." << endl;
} else {
    cout << "Você é uma criança." << endl;
}
```

`else`

A estrutura `else` é associada a um `if` e é executada quando a condição do `if` é falsa. Ela permite definir um bloco de código alternativo a ser executado.

A sintaxe é:

```
if (condição) {
    // Bloco de código se a condição for verdadeira
} else {
    // Bloco de código se a condição for falsa
}
```

Exemplo:

```
int numero = 7;
if (numero % 2 == 0) {
    cout << "O número é par." << endl;
} else {
    cout << "O número é ímpar." << endl;
}
```

Ternário

O operador ternário é uma forma compacta de escrever uma estrutura `if-else` em uma única linha. Ele avalia uma expressão e retorna um valor com base nessa avaliação.

A sintaxe é:

```
(condição) ? expressão1 : expressão2
```

- `condição`: Uma expressão booleana a ser avaliada.
- `expressão1`: Valor retornado se a condição for verdadeira.
- `expressão2`: Valor retornado se a condição for falsa.

Exemplo:

```
int idade = 20;
string mensagem = (idade >= 18) ? "Maior de idade" : "Menor d
cout << mensagem << endl;
```

Switch

O `switch` é uma estrutura de controle de fluxo que permite selecionar um dos muitos blocos de código a serem executados. É uma alternativa ao encadeamento de `if-else if-else`, especialmente quando há várias condições possíveis a serem verificadas.

```
switch (expressão) {
    case valor1:
        // Bloco de código a ser executado se expressão == va
        break;
```

```
    case valor2:
        // Bloco de código a ser executado se expressão == va
        break;
    // Outros casos possíveis...
    default:
        // Bloco de código a ser executado se nenhum dos caso
}
```

- `expressão`: Uma expressão cujo valor é comparado com os valores de cada `case`.
- `case valor1`, `case valor2`, etc.: Possíveis valores que a `expressão` pode assumir.
- `break`: Indica o fim do bloco de código a ser executado para um determinado `case`. Se não for incluído, o fluxo de execução continuará para o próximo `case` sem interrupção.
- `default`: Opcional. Define um bloco de código a ser executado caso nenhum dos casos anteriores corresponda ao valor da `expressão`.

Exemplo:

```
int diaDaSemana = 3;
switch (diaDaSemana) {
    case 1:
        cout << "Domingo" << endl;
        break;
    case 2:
        cout << "Segunda-feira" << endl;
        break;
    case 3:
        cout << "Terça-feira" << endl;
        break;
    case 4:
        cout << "Quarta-feira" << endl;
        break;
    case 5:
        cout << "Quinta-feira" << endl;
        break;
    case 6:
```

```
            cout << "Sexta-feira" << endl;
            break;
        case 7:
            cout << "Sábado" << endl;
            break;
        default:
            cout << "Dia inválido" << endl;
    }
```

- A expressão dentro do `switch` é avaliada e comparada com os valores dos casos.
- Se uma correspondência for encontrada, o bloco de código associado a esse `case` é executado. O fluxo de execução continua até encontrar um `break` ou o final do `switch`.
- Se nenhum `case` corresponder ao valor da expressão, o bloco de código dentro do `default` (se existir) é executado.

▼ O `break` é importante para evitar a execução inadvertida de outros casos após uma correspondência.

A necessidade de usar o `break` dentro da estrutura `switch` em C++ está relacionada ao controle do fluxo de execução do programa. Sem o `break`, o programa continuaria executando os blocos de código de todos os casos subsequentes, mesmo depois de encontrar um caso correspondente. Aqui estão alguns pontos importantes sobre o uso do `break`:

1. **Terminação do Case Atual:** Quando um `break` é encontrado dentro de um caso, o controle é transferido para fora do `switch`, ignorando os blocos de código restantes. Isso impede que o programa execute os blocos de código de todos os casos subsequentes.
2. **Evitar Execução de Códigos Indesejados:** Sem o `break`, o programa continuaria executando os blocos de código de todos os casos subsequentes, mesmo depois de encontrar um caso correspondente. Isso pode levar a resultados indesejados ou comportamentos inesperados no programa.
3. **Garantir Comportamento Esperado:** O `break` garante que apenas o bloco de código associado ao caso correspondente seja executado. Isso ajuda a garantir que o programa se comporte conforme o esperado

e evita resultados inesperados devido a execuções indesejadas de códigos.

4. **Padrão de Codificação:** O uso do `break` é uma prática recomendada de programação e é considerado uma boa prática de codificação em C++. Isso ajuda a tornar o código mais legível e compreensível para outros programadores que possam revisá-lo posteriormente.

No entanto, é importante notar que o uso do `break` é opcional em algumas situações específicas. Por exemplo, em alguns casos, pode ser desejável que a execução continue em múltiplos casos, ou pode ser que o último caso de um `switch` não precise de um `break`, pois não há mais blocos de código a serem executados abaixo dele. No entanto, na maioria dos casos, é recomendável usar o `break` para garantir o comportamento esperado do programa.

Observações:

- A expressão no `switch` geralmente é uma variável inteira ou um tipo enumerado.
- Os casos podem ser especificados usando valores constantes (inteiros, caracteres, enumerações, etc.).
- É possível ter múltiplos casos com o mesmo bloco de código, desde que o `break` seja utilizado corretamente.

O `switch` é uma ferramenta poderosa para selecionar entre várias opções em um programa, tornando o código mais legível e organizado quando comparado a uma sequência de `if-else if-else`. No entanto, é importante usá-lo com moderação e considerar a legibilidade do código ao decidir entre `switch` e `if-else`.

Loops

For

O loop `for` é utilizado quando sabemos exatamente quantas vezes queremos executar um bloco de código. Ele consiste em três partes: inicialização, condição de continuação e atualização.

```
for (inicialização; condição; atualização) {
    // Bloco de código a ser executado repetidamente
```

```
}
```

- `inicialização`: Uma expressão que é executada uma vez no início do loop. É geralmente usada para inicializar uma variável de controle.
- `condição`: Uma expressão booleana que é verificada antes de cada iteração do loop. Se for verdadeira, o bloco de código é executado; se for falsa, o loop é encerrado.
- `atualização`: Uma expressão executada após cada iteração do loop. É geralmente usada para modificar a variável de controle.

Exemplo:

```
for (int i = 0; i < 5; i++) {
    cout << "O valor de i é: " << i << endl;
}
```

While

O loop `while` é utilizado quando queremos executar um bloco de código enquanto uma condição específica for verdadeira.

```
while (condição) {
    // Bloco de código a ser executado repetidamente
}
```

-

`condição`: Uma expressão booleana que é verificada antes de cada iteração do loop. Se for verdadeira, o bloco de código é executado; se for falsa, o loop é encerrado.

Exemplo:

```
int contador = 0;
while (contador < 3) {
    cout << "Contagem: " << contador << endl;
    contador++;
}
```

Do-while

O loop `do-while` é semelhante ao loop `while`, mas garante que o bloco de código seja executado **pelo menos uma vez**, mesmo que a condição seja falsa na primeira verificação.

```
do {
    // Bloco de código a ser executado repetidamente
} while (condição);
```

- `condição`: Uma expressão booleana que é verificada após cada iteração do loop. Se for verdadeira, o bloco de código continua a ser executado; se for falsa, o loop é encerrado.

Exemplo:

```
int numero;
do {
    cout << "Digite um número positivo: ";
    cin >> numero;
} while (numero <= 0);
```

Observações:

- Os loops `for`, `while` e `do-while` podem ser interrompidos com a palavra-chave `break` se uma condição específica for atendida durante a execução.
- A palavra-chave `continue` pode ser usada para pular a iteração atual do loop e passar para a próxima iteração.

Range based for loop

O loop baseado em intervalo é uma forma conveniente de iterar sobre todos os elementos de um contêiner (como um vetor, uma lista, um array, etc.) em C++. Ele foi introduzido no C++11 e é uma adição útil à linguagem, tornando a iteração sobre contêineres mais simples e legível.

Sintaxe:

```
for (tipo elemento : contêiner) {
    // Bloco de código a ser executado para cada elemento
}
```

- `tipo`: O tipo de dado dos elementos no contêiner. Normalmente, você usará `auto` aqui para deixar o compilador inferir o tipo automaticamente.
- `elemento`: Uma variável que representa cada elemento do contêiner em cada iteração do loop.
- `contêiner`: O contêiner sobre o qual você deseja iterar.

Como funciona:

1. O loop baseado em intervalo itera sobre todos os elementos no contêiner especificado.
2. Para cada iteração, o valor do elemento atual é atribuído à variável `elemento`.
3. O bloco de código dentro do loop é executado para cada elemento no contêiner.

Exemplo:

```
#include <iostream>#include <vector>int main() {
    std::vector<int> numeros = {1, 2, 3, 4, 5};

    // Loop baseado em intervalo para percorrer os elemento
s do vetor 'numeros'
    for (int numero : numeros) {
        std::cout << numero << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

Resultado:

```
1 2 3 4 5
```

Vantagens:

- Sintaxe simples e legível: A sintaxe do loop baseado em intervalo é mais simples e mais legível do que a sintaxe de loops tradicionais.
- Evita erros comuns: Ajuda a evitar erros comuns, como erros de índice fora dos limites, que podem ocorrer com loops tradicionais.
- Mais seguro: O loop baseado em intervalo é seguro contra a modificação acidental dos índices durante a iteração.

Limitações:

- Não é possível acessar os índices dos elementos diretamente no loop baseado em intervalo, como seria possível com um loop tradicional usando um contador.
- Não é possível modificar os elementos do contêiner durante a iteração, a menos que o elemento seja passado por referência.

Quando usar:

- O loop baseado em intervalo é especialmente útil quando você precisa percorrer todos os elementos de um contêiner, sem se preocupar com índices ou com a manipulação direta de ponteiros.
- É uma escolha preferível sempre que você precisa apenas dos valores dos elementos e não dos índices ou de uma iteração reversa.

Em resumo, o loop baseado em intervalo é uma adição valiosa ao C++, tornando a iteração sobre contêineres mais simples, segura e legível. É uma boa prática de programação usar o loop baseado em intervalo sempre que possível, especialmente quando você precisa apenas dos valores dos elementos e não dos índices.

Observações importantes

Incremento para o for

1. `i++` (pós-incremento):

A expressão `i++` realiza o incremento da variável `i`, mas retorna o valor original de `i` antes do incremento.

Exemplo:

```
int i = 0;
int valor = i++; // Primeiro, valor recebe o valor atual de
i (0), depois i é incrementado para 1
// Agora, i = 1 e valor = 0
```

2. `++i` (pré-incremento):

A expressão `++i` realiza o incremento da variável `i` e retorna o valor atualizado de `i` após o incremento.

Exemplo:

```
int i = 0;
int valor = ++i; // Primeiro, i é incrementado para 1 e dep
ois valor recebe o novo valor de i (1)
// Agora, i = 1 e valor = 1
```

Diferença:

A principal diferença entre `i++` e `++i` é o momento em que o incremento é aplicado e o valor retornado pela expressão:

- No pós-incremento (`i++`), o incremento é aplicado após o valor original de `i` ser usado na expressão. Isso significa que o valor retornado é o valor original de `i`.
- No pré-incremento (`++i`), o incremento é aplicado antes do valor de `i` ser usado na expressão. Isso significa que o valor retornado é o valor atualizado de `i` após o incremento.

Quando usar cada um:

- Use `i++` quando precisar usar o valor original de `i` na expressão e, em seguida, incrementar `i`.
- Use `++i` quando precisar usar o valor atualizado de `i` após o incremento na expressão.

Nos loops, a diferença entre `i++` e `++i` geralmente não é significativa, especialmente em loops simples. No entanto, em certos casos, como ao usar variáveis de controle em expressões complexas dentro de um loop, a escolha entre `i++` e `++i` pode afetar o comportamento do programa e até mesmo a

eficiência do código. Portanto, é importante entender essa distinção e usá-la de acordo com as necessidades do seu código.

`while (n--)`

A expressão `n--` é um exemplo de pós-decremento em C++. Ela decrementa o valor da variável `n` após usar o valor original da variável na expressão em que está inserida. Isso significa que o valor de `n` é decrementado após a expressão ser avaliada.

Aqui está uma explicação detalhada sobre como funciona o pós-decremento:

1. O valor atual de `n` é usado na expressão onde `n--` está inserido.
2. Após a avaliação da expressão, o valor de `n` é decrementado em 1.

Vamos ver um exemplo para ilustrar isso:

```
int n = 5;
while (n--) {
    cout << "O valor de n é: " << n << endl;
}
```

Neste exemplo, a expressão `n--` é usada como condição do loop `while`. Aqui está o que acontece em cada iteração do loop:

1. Na primeira iteração, o valor original de `n` (5) é usado na condição do loop. A expressão é avaliada como verdadeira, pois o valor original de `n` (5) é diferente de zero.
2. Após a avaliação da expressão, o valor de `n` é decrementado em 1, então agora `n` é igual a 4.

Esse processo continua até que `n` se torne zero. Na próxima iteração, `n` será igual a 0, e a expressão `n--` será avaliada como falsa, terminando o loop.

É importante observar que, como o pós-decremento ocorre após o uso do valor original de `n`, o valor 0 também será impresso no loop, pois o decremento ocorre após a avaliação da expressão.