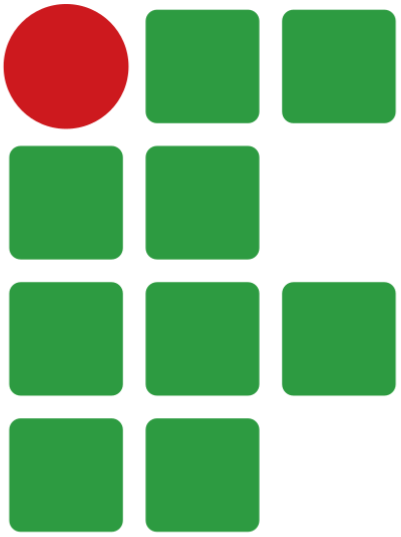
Material de apoio 4

Introdução ao paradigma de programação estruturada de Computadores



INSTITUTO FEDERAL
São Paulo

Autor: Gabriel de Azevedo Camargo

Sumário

1 Paradigma de Programação Estruturada: Fundamentos e Princípios

2 Introdução

O paradigma de programação estruturada é um modelo fundamental que revolucionou a forma como os programas de computador são desenvolvidos e mantidos. Desde sua concepção na década de 1960, a programação estruturada tem sido uma pedra angular da engenharia de software, proporcionando organização, clareza e eficiência no desenvolvimento de sistemas complexos.

Neste texto, exploraremos os fundamentos e princípios da programação estruturada, desde sua origem até sua aplicação prática nos dias de hoje. Discutiremos as estruturas de controle básicas, os conceitos de modularidade e reutilização de código, bem como os benefícios que esse paradigma traz para a construção de software robusto e escalável.

3 História e Contexto

A programação estruturada surgiu como uma resposta aos desafios enfrentados pelos programadores na década de 1960, quando os programas eram escritos de forma desorganizada e desestruturada, resultando em códigos difíceis de entender e manter. Naquela época, o principal paradigma de programação era o de programação não estruturada, que permitia o uso indiscriminado de instruções de salto, como o infame comando "goto".

No entanto, o uso excessivo de "gotos" levava a programas confusos, propensos a erros e de difícil manutenção. Em 1966, o famoso artigo de Edsger Dijkstra intitulado "Go To Statement Considered Harmful" criticou veementemente o uso indiscriminado de instruções de salto, argumentando que elas tornavam o controle de fluxo do programa difícil de entender e seguir.

Como alternativa à programação não estruturada, surgiu a programação estruturada, que propunha uma abordagem mais organizada e disciplinada para o desenvolvimento de software. Influenciado por trabalhos anteriores de cientistas da computação como Alan Perlis e Christopher Strachey, o conceito de programação estruturada foi formalizado por Dijkstra, bem como por outros pioneiros da computação, como Niklaus Wirth e Tony Hoare.

4 Princípios da Programação Estruturada

A programação estruturada é baseada em uma série de princípios fundamentais que visam promover a clareza, a modularidade e a eficiência no desenvolvimento de software. Alguns dos princípios-chave da programação estruturada incluem:

1. Decomposição: Divide um programa em partes menores e mais gerenciáveis, conhecidas como módulos ou funções, cada uma responsável por uma única tarefa específica.
2. Abstração: Esconde os detalhes de implementação de um módulo, fornecendo uma interface clara e simplificada para sua utilização.
3. Encapsulamento: Protege os dados e funcionalidades de um módulo, permitindo que sejam acessados apenas por meio de interfaces bem definidas, o que evita o acesso direto e não autorizado.
4. Modularidade: Favorece a construção de programas compostos por módulos independentes e reutilizáveis, que podem ser combinados de forma flexível para criar sistemas complexos.
5. Hierarquia: Organiza os módulos em uma estrutura hierárquica, onde os módulos de nível superior coordenam e controlam os módulos de nível inferior, promovendo uma separação clara de responsabilidades.

Estes princípios formam a base da programação estruturada e orientam a forma como os programas são projetados, implementados e mantidos.

4.1 Decomposição

A decomposição é um princípio fundamental da programação estruturada que envolve dividir um programa em partes menores e mais gerenciáveis, conhecidas como módulos ou funções.

Cada módulo é responsável por realizar uma única tarefa específica dentro do programa. Ao dividir o programa em módulos menores, torna-se mais fácil entender, manter e modificar o código. Além disso, a decomposição permite reutilizar o código em diferentes partes do programa, promovendo a modularidade e reduzindo a redundância.

Exemplo:

```
#include <stdio.h>

// Função para calcular a soma de dois números
int soma(int a, int b) {
    return a + b;
}

// Função principal
int main() {
    int resultado = soma(5, 3);
    printf("Resultado da soma: %d\n", resultado);
    return 0;
}
```

4.2 Abstração

A abstração é outro princípio chave da programação estruturada, que envolve esconder os detalhes de implementação de um módulo e fornecer uma interface clara e simplificada para sua utilização.

Em outras palavras, os detalhes internos de como uma função ou módulo realiza sua tarefa são ocultados do usuário, que só precisa conhecer sua interface pública. Isso facilita o uso do módulo e promove a reutilização do código, uma vez que os usuários podem se concentrar apenas na funcionalidade fornecida pela interface pública.

Exemplo:

```
#include <stdio.h>

// Função para calcular a média de três números
float calcularMedia(float a, float b, float c) {
    return (a + b + c) / 3;
}

// Função principal
int main() {
    float nota1 = 7.5, nota2 = 8.0, nota3 = 6.5;
    float media = calcularMedia(nota1, nota2, nota3);
    printf("Média das notas: %.2f\n", media);
    return 0;
}
```

4.3 Encapsulamento

O encapsulamento é um princípio que visa proteger os dados e funcionalidades de um módulo, permitindo que sejam acessados apenas por meio de interfaces bem definidas. Isso significa que os detalhes internos de um módulo são protegidos contra acesso direto e não autorizado.

Em vez disso, os usuários interagem com o módulo apenas através de suas funções ou métodos públicos, garantindo que as operações sejam realizadas de maneira controlada e segura. O encapsulamento promove a segurança e a integridade dos dados, além de facilitar a manutenção e a evolução do código.

Exemplo:

```
#include <stdio.h>
// Definição da estrutura Ponto
struct Ponto {
    float x;
    float y;
};
// Função para calcular a distância entre dois pontos
float calcularDistancia(struct Ponto p1, struct Ponto p2) {
    float distancia = sqrt(pow(p2.x - p1.x, 2) + pow(p2.y - p1.y, 2));
    return distancia;
}
// Função principal
int main() {
    // Pontos
    struct Ponto ponto1 = {3, 4};
    struct Ponto ponto2 = {6, 8};
```

```
    // Cálculo da distância entre os pontos
    float distancia = calcularDistancia(ponto1, ponto2);
    printf("Distância entre os pontos: %.2f\n", distancia);
    return 0;
}
```

4.4 Modularidade

A modularidade é um princípio fundamental da programação estruturada que favorece a construção de programas compostos por módulos independentes e reutilizáveis. Cada módulo é projetado para realizar uma única tarefa específica e pode ser desenvolvido, testado e mantido separadamente dos outros módulos.

Isso permite que os desenvolvedores trabalhem de forma mais eficiente e colaborativa, além de facilitar a compreensão e a manutenção do código. A modularidade também promove a reutilização do código, uma vez que os módulos podem ser combinados de forma flexível para criar sistemas complexos.

Exemplo:

```
#include <stdio.h>

// Função para calcular o quadrado de um número
int quadrado(int x) {
    return x * x;
}

// Função principal
int main() {
```

```
    int numero = 5;

    int resultado = quadrado(numero);

    printf("O quadrado de %d é: %d\n", numero, resultado);

    return 0;

}
```

4.5 Hierarquia

Na prática da programação estruturada, o conceito de hierarquia desempenha um papel crucial ao organizar os diferentes elementos de um programa em uma estrutura em camadas. Nessa abordagem, os módulos de nível superior atuam como coordenadores, supervisionando e controlando as operações dos módulos de nível inferior. Essa divisão hierárquica não apenas promove uma separação clara de responsabilidades, mas também facilita a compreensão e a manutenção do código.

Ao adotar a hierarquia na programação, cada camada do sistema assume funções específicas e bem definidas. Os módulos de nível superior lidam com tarefas mais abrangentes e globais, enquanto os módulos de nível inferior tratam de operações mais detalhadas e específicas. Essa divisão permite que o código seja modular, modular e fácil de escalar, pois cada camada pode ser desenvolvida, testada e aprimorada de forma independente.

Além disso, a hierarquia na programação estruturada promove a reutilização de código, uma vez que módulos de nível inferior podem ser compartilhados e invocados por diferentes partes do programa. Isso resulta em uma base de código mais coesa e eficiente, com menos redundância e uma melhor organização geral.

Em resumo, a hierarquia na programação estruturada é essencial para criar sistemas de software robustos e escaláveis. Ao dividir o programa em camadas bem definidas e gerenciáveis, os desenvolvedores podem promover uma arquitetura limpa e modular, facilitando a manutenção e a evolução do código ao longo do tempo.

Exemplo:

```
#include <stdio.h>

// Função para calcular a área de um retângulo
float calcularAreaRetangulo(float comprimento, float largura) {
    return comprimento * largura;
}
```

```
// Função para exibir o resultado
void exibirResultado(float area) {
	printf("A área do retângulo é: %.2f\n", area);
}

// Função principal
int main() {
	float comprimento = 5.0;
	float largura = 3.0;

	// Chamada das funções para calcular e exibir a área
	float area = calcularAreaRetangulo(comprimento, largura);
	exibirResultado(area);

	return 0;
}
```

5 Estruturas de Controle na Programação Estruturada

Uma das características mais distintivas da programação estruturada é o uso de estruturas de controle bem definidas, que permitem a execução de instruções de forma organizada e previsível. As principais estruturas de controle na programação estruturada são:

5.1 Sequência

A sequência é a estrutura de controle mais simples e direta, na qual as instruções são executadas em uma ordem linear, uma após a outra. Isso significa que a execução do programa segue um fluxo claro e definido, sem desvios ou ramificações. A sequência é a base sobre a qual todas as outras estruturas de controle são construídas.

Exemplo na linguagem c:

```
int soma = 0;

for (int i = 1; i <= 10; i++) {

    soma += i;

}

printf("A soma dos números de 1 a 10 é: %d\n", soma);
```

Neste exemplo, as instruções dentro do loop "for" são executadas em sequência, resultando na soma dos números de 1 a 10.

5.2 Seleção

A estrutura de seleção permite que o programa tome decisões com base em condições específicas. Isso é feito usando instruções de "if", "else if" e "else", que avaliam uma expressão booleana e executam o código correspondente, dependendo se a condição é verdadeira ou falsa. O "if" é ideal para avaliar uma condição simples e executar um bloco de código se essa condição for verdadeira. Já o "else if" é usado para avaliar condições adicionais se a condição do "if" for falsa. O "else" é opcional e é executado se nenhuma das condições anteriores for verdadeira. Essa estrutura de seleção é muito flexível e permite lidar com uma variedade de cenários de decisão.

Por outro lado, a estrutura switch-case é uma alternativa à estrutura if-else if-else quando se trata de avaliar uma expressão com múltiplas possibilidades. Em vez de avaliar uma série de condições, como o if-else if-else, o switch-case avalia uma única expressão e direciona a execução para um dos muitos blocos de código, com base no valor dessa expressão.

Uma das principais diferenças entre switch-case e if-else if-else é que o switch-case só pode ser usado para comparar valores inteiros ou caracteres. Isso significa que a expressão avaliada no switch-case deve resultar em um valor inteiro ou um caractere. Por outro lado, o if-else if-else pode avaliar qualquer expressão booleana, tornando-o mais flexível em comparação com o switch-case.

Outra diferença importante é que o switch-case só permite a comparação de igualdade, enquanto o if-else if-else pode lidar com uma variedade de operadores de comparação, como maior

que, menor que, maior ou igual a, etc. Isso significa que o switch-case é mais limitado em termos de condições que pode avaliar.

Apesar de suas diferenças, tanto o switch-case quanto o if-else if-else são úteis em diferentes situações. O switch-case é especialmente útil quando se trata de avaliar uma expressão com várias opções claras e distintas, enquanto o if-else if-else é mais adequado para avaliar condições mais complexas e variáveis.

Exemplo de if, else if e else na linguagem c:

```
int x = 10;

if (x > 0) {

    printf("x é positivo\n");

} else if (x < 0) {

    printf("x é negativo\n");

} else {

    printf("x é zero\n");

}
```

Exemplo de switch case na linguagem c:

```
int dia = 3;

switch (dia) {

case 1:

   printf("Domingo\n");

   break;

case 2:
```

```
    printf("Segunda-feira\n");
    break;
case 3:
    printf("Terça-feira\n");
    break;
case 4:
    printf("Quarta-feira\n");
    break;
case 5:
    printf("Quinta-feira\n");
    break;
case 6:
    printf("Sexta-feira\n");
    break;
case 7:
    printf("Sábado\n");
    break;
default:
    printf("Dia inválido\n");
    break;
}
```

Neste exemplo, o programa imprime uma mensagem dependendo do valor da variável "x".

5.3 Repetição

A estrutura de repetição, também conhecida como loop, permite que o programa execute um bloco de código repetidamente enquanto uma condição específica for verdadeira. Existem três tipos principais de loops na programação estruturada: "while", "do-while" e "for".

- **while:** O loop "while" é a estrutura de repetição mais simples. Ele executa um bloco de código enquanto uma condição específica for verdadeira. A condição é avaliada antes da execução do bloco de código, o que significa que o bloco pode não ser executado nenhuma vez se a condição inicialmente for falsa.

- **do-while:** O loop "do-while" é semelhante ao loop "while", mas a diferença fundamental é que a condição é avaliada após a execução do bloco de código. Isso garante que o bloco de código seja executado pelo menos uma vez, independentemente da condição inicial. O loop "do-while" é útil quando você precisa garantir a execução do bloco pelo menos uma vez, mesmo se a condição inicial for falsa.

- **for:** O loop "for" é uma estrutura de repetição mais complexa, mas também mais flexível. Ele consiste em três partes: inicialização, condição e atualização. A inicialização é executada uma vez antes do loop começar. A condição é avaliada antes de cada iteração do loop e determina se o loop deve continuar executando. A atualização é executada após cada iteração do loop. O loop "for" é comumente usado quando você sabe exatamente quantas vezes deseja repetir um bloco de código.

Exemplo de while:

c

```c
int i = 1;

while (i <= 5) {

    printf("%d\n", i);
```

```
    i++;
}
```

Exemplo de do-while:

```
int i = 1;
do {
    printf("%d\n", i);
    i++;
} while (i <= 5);
```

Neste exemplo, o programa imprime os números de 1 a 5 usando um loop "while". Estes são apenas alguns exemplos das estruturas de controle na programação estruturada. Essas estruturas fornecem uma maneira organizada e eficiente de controlar o fluxo de execução do programa, facilitando a implementação de algoritmos complexos e a manutenção de código. Continuarei expandindo o texto para abranger mais aspectos da programação estruturada.

6 Exercícios

1. **Decomposição:** Escreva uma função em C chamada `soma` que recebe dois números inteiros como parâmetros e retorna a soma deles.

2. **Abstração:** Crie uma função em C chamada `calcularMedia` que calcula a média de três números de ponto flutuante e retorna o resultado.

3. **Encapsulamento:** Defina uma estrutura em C chamada `Pessoa` que possui os campos `nome` (string) e `idade` (int). Em seguida, crie funções `setNome` e `setIdade` para definir o nome e a idade de uma pessoa, respectivamente.

4. **Modularidade:** Implemente uma função em C chamada `verificarPrimo` que recebe um número inteiro como parâmetro e retorna verdadeiro se o número for primo, caso contrário, retorna falso.

5. **Hierarquia:** Escreva um programa em C que utiliza as funções definidas nos exercícios anteriores para calcular a média de idade de um grupo de pessoas e determinar quantos dos números de 1 a 100 são primos.

7 Respostas

1)
```
int soma(int a, int b) {
    return a + b;
}
```

2)
```
float calcularMedia(float a, float b, float c) {
    return (a + b + c) / 3;
}
```

3)
```
#include <stdio.h>
#include <string.h>

struct Pessoa {
    char nome[50];
    int idade;
};

void setNome(struct Pessoa *p, const char *nome) {
    strcpy(p->nome, nome);
}

void setIdade(struct Pessoa *p, int idade) {
    p->idade = idade;
}
```

4)
```
#include <stdbool.h>

bool verificarPrimo(int numero) {
    if (numero <= 1)
```

```
        return false;
    for (int i = 2; i * i <= numero; i++) {
        if (numero % i == 0)
            return false;
    }
    return true;
}
```

5)

```
#include <stdio.h>


int main() {
    // Cálculo da média de idade
    struct Pessoa pessoa1, pessoa2, pessoa3;
    setNome(&pessoa1, "João");
    setIdade(&pessoa1, 25);
    setNome(&pessoa2, "Maria");
    setIdade(&pessoa2, 30);
    setNome(&pessoa3, "Pedro");
    setIdade(&pessoa3, 35);
    float mediaIdade = calcularMedia(pessoa1.idade, pessoa2.idade,
pessoa3.idade);
    printf("Média de idade: %.2f\n", mediaIdade);

    // Contagem de números primos de 1 a 100
    int contadorPrimos = 0;
    for (int i = 1; i <= 100; i++) {
        if (verificarPrimo(i))
            contadorPrimos++;
    }
    printf("Quantidade de números primos de 1 a 100: %d\n",
contadorPrimos);

    return 0;
}
```