Estruturas de controle de fluxo

Gonzalo Travieso

2018

1 Estado de um programa

Dentro do paradigma de programação imperativo, no qual se classifica a linguagem C++, a execução de um programa consiste na realização de um conjunto de operações. Como vimos, os dados sobre os quais essas operações são executadas ficam armazenados na memória do computador. Al[em disso, o computador executa operações de entrada e saída de dados como, principalmente, mostra ou recebimento de dados do usuário do programa ou leitura e escrita em arquivos.

1.1 Estado

No momento, vamos nos ocupar apenas de operações na memória. Neste sentido, o efeito principal da execução de um comando no programa é a alteração do valor de dados na memória.

O conjunto dos valores de dados do programa na memória (isto é, os valores das suas variáveis ou de quaisquer outras posições de memória acessíveis através delas) é denominado seu *estado*.

Dizemos então que um comando executa a alteração do estado do programa.

Note que, na maioria dos casos, para analisar um pedaço de código não precisamos nos preocupar com todo o estado do programa, mas apenas com a parte do estado que está sendo manipulada por esse pedaço.

Por exemplo, supondo que o seguinte trecho de código aparece num programa.

```
int a{2}; // 1
int b{3}; // 2
a += b; // 3
```

Após a execução da linha marcada como 1, o estado (que nos interessa) é aumentado com uma variável `a` e ela tem o valor de 2. Após a execução da linha 2, o estado continua tendo 2 na variável `a`, mas agora tem também uma variável `b` com valor 3. Por fim, após a execução da linha 3, o estado é tal que a variável `a` tem o valor 5 enquanto `b` continua com o valor 3. Como esse trecho de código não faz referência a outras variáveis, não precisamos considerá-las ao analisar o efeito do código no estado.

1.2 Predicados sobre um estado

Para descrever um estado utilizamos *predicados*, que sáo afirmações lógicas sobre esse estado. A idéia é que o predicado define um conjunto de estados, entre todos

os estados possíveis, que é o conjunto dos estados que satisfazem o predicado (fazem com que ele seja verdadeiro).

No exemplo anterior, após a execução do comando 1, podemos formar, entre muitos outros, os seguintes predicados:

- $a = 2$
- $a >= 0$
- $a < 5$

Note que uma grande número de predicados pode ser satisfeito pelo estado atual. Geralmente, procuramos usar os predicamos mais restritivos que descrevem corretamente o intuito do código. No nosso caso, seria o primeiro predicado acima.

Após a execução do comando 2, podemos indicar o estado pelo predicado $a = 2 \wedge b = 3$ e após a execução de 3 por $a = 5 \wedge b = 3$. Neste caso, o predicado se refere apenas às variáveis `a` e `b`, pois elas são as que nos interessam na análise do código atual. Geralmente, para indicar os predicados, usaremos a notação de C++ para operações lógicas; no exemplo, após o comando 3 o predicado seria indicado por `a == 5 && b == 3`.

1.2.1 Pré-condições e pós-condições

Quando estamos analisando um pedaço de código, para a execução correta do mesmo precisamos garantir certas condições. Essas condições se expressam como um predicado sobre o estado do sistema, que é denominado uma **pré-condição**.

Da mesma forma, o que é atingido pela execução do trecho de código pode ser expresso por um predicado sobre o estado (lembre-se, no momento estamos desconsiderando efeitos colaterias que não sejam na memória), que é denominado uma **pós-condição**.

Isto é:

Pré-condição indica um predicado sobre o estado do sistema que deve ser satisfeito para que a execução do código possa ser correta.

Pós-condição indica um predicado que descreve o estado resultante da execução do código, e portanto indica o objetivo desse código.

Podemos então garantir que um trecho de código está correto se pudermos provar o seguinte: se a pré-condição é satisfeita antes da execução desse código, então a pós-condição será satisfeita no fim dessa execução.

Veremos alguns exemplos a seguir.

2 Execução condicional

Em diversas situações, queremos executar alguns comandos apenas para alguns dos possíveis estados do sistema.

Por exemplo, suponha que temos uma variável `int` de nome `x`, que queremos colocar em zerar *apenas no caso de ela ser negativa*. Como já vimos, é fácil de colocar `x` em zero com um código da forma

```
x = 0;
```

No entanto, se fazemos apenas isso, ela será zerada também se tinha antes um valor positivo. Este é um caso em que o comando em questão deve ser executado *condicionalmente ao estado corrente do sistema.*

2.1 Guardas

O caso mais simples de execução condicional é como no exemplo anterior, onde um conjunto de comandos deve ser executado apenas em algumas situações. Para isso precisamos por um sentinela ou guarda, na execução do comando. Esse guarda é representado por uma *estrutura de controle de fluxo de execução* que permite a execução dos comandos apenas se certo predicado (chamado *condição*) sobre o estado for satisfeito.

A sintaxe é como indicada abaixo

```
if (condicao) {
    Comando1;
    Comando2;
    ...
    ComandoN
}
```

onde `condicao` deve ser uma expressão que retorne um valor booleano (por exemplo, expressões lógicas e de comparação), e os diversos `Comandox` representam o trecho de código que deve ser executado apenas se `condicao` resultar em valor `true`.

É importante notar que os parêntesis `( )` são parte obrigatória da sintaxe dessa estrutra de controle; já as chaves `{ }` são opcionais, e necessárias apenas quando há mais do que um comando a executar. De qualquer modo, é recomendado sempre usar as chaves, para tornar o código de mais fácil leitura e evitar erros ao fazer alterações posteriores no código (como adição de novos comandos).

Para o nosso exemplo anterior, teremos então

```
if (x < 0) {
    x = 0;
}
```

No caso geral, temos

```
// P1
if (C) {
    // P2
    S;
    // Q2
}
// Q1
```

onde `C` indica a condição a ser testada e `S` o camando (ou conjunto de comandos) a ser executado caso ela seja verdadeira. Anotamos acima com `P1` e `Q1` para respectivamente a pré-condição e a pós-condição do condicional como um todo; `P2` e `Q2` são pré e pós-condição do(s) comando(s) executado(s) caso `C` seja veradeira. Podemos ver então que, para a construção ser válida, devemos ter:

- `P1 && C` deve implicar `P2` (isto é, se a condição para a execução do comando todo for satisfeita e a condição do teste for verdadeira, então devemos satisfazer a condição para e execução de `S`);
- `Q2` deve ser válida após a execução de `S` se `P2` for válida antes (esta é a relação tradicional entre pré e pós-condições);
- `Q2` deve implicar `Q1`.
- `P1 && !C` deve implicar `Q1` (isto é, se `C` for falsa, a pós-condição do código já deve estar satisfeita, ou então estaria faltando algum código).

  No caso do nosso exemplo de zerar o valor de `x` se ele for negativo, o código pode ser executado desde que `x` seja do tipo apropriado (numérico), mas isto é testado explicitamente pelo compilador, e já está expresso no código na declaração da variável `x` (por exemplo, se declaramos `int x`, então não é possível que `x` tenha algum valor para o qual o código não pode ser executado). *Esta é uma das vantagens de trabalhar com linguagens com verificação estática de tipos: os tipos já expressam restrições sobre o código, essas restrições evitam certos erros e são testadas pelo compilador.* Portanto, neste exemplo, `P1` é vazia, isto é, `P1` vale `true`.

  A pós-condição `Q1` por outro lado é o que desejamos atingir com o código (zerar a variável se ela for negativa, manter o valor caso contrário), que pode ser expresso como `x' == x` se `x >= 0` e `x' == 0` se `x < 0` (onde usei um apóstrofe para indicar o valor alterado de uma variável após a execução do código, e distingui-lo do valor anterior).

  Como queremos colocar `x` em zero apenas se ele for negativo, então a pré-condição `P2` é `x < 0`, que neste caso é trivialmente igual a `P1 && C` (pois `true && x > 0` é o mesmo que `x > 0`). A pós-condição `Q2` é `x < 0 && x' == 0`, o que é garantido pela atribuição e por P2.

  Para ver que `Q2` implica `Q1` basta ver que para o subconjunto de todos os estados (neste caso importa apenas o valor de `x`) que satisfazem `Q2` (isto é, com `x < 0`) `Q1` requer que `x' == 0`, o que é garantido por `Q2`.

  Da mesma forma `P1 && !C` vale `true && x >= 0` e para estes estados, como nenhum código é executado, temos `x' == x`, garantindo `Q1`.

2.2 Escolha

No caso da execução condicional usada como um guarda, como descrito acima, a pós-condição do código já é satisfeita se a condição do `if` não for satisfeita. Em muitos casos, se a condição não for satisfeita devemos executar outro código, isto é, queremos escolher entre dois códigos possíveis para executar, dependendo de uma condição. Por exemplo, temos um número inteiro na variável `n` e queremos calcular a metade arredondada para cima; isto é, se o número for par, calculamos sua metade, se for ímpar, calculamos a metade de `n+1`. Podemos fazer isso seguindo o comando `if` de um `else`, como no código abaixo:

```
if (m % 2 == 0) {
    // m par
    metade_mais = m / 2;
}
```

```
else {
    // m impar
    metade_mais = (m + 1) / 2;
}
```

Note como o `else` vem logo em seguida do bloco de comandos executados caso a condição seja verdadeira; o bloco de comandos que segue o `else` sejá executado apenas se a condição for falsa.

No caso geral temos:

```
// P1
if (C) {
    // P2
    S1;
    // Q2
}
else {
    // P3
    S2;
    // Q3
}
// Q1
```

E as regras para correção são:

- `P1 && C` implica `P2`
- `P1 && !C` implica `P3`
- Se `P2` é garantido antes da execução de `S1`, então temos `Q2`
- Se `P3` é garantido antes da execução de `S2`, então temos `Q3`
- `Q2` implica `Q1`
- `Q3` implica `Q1`

Você pode verificar que isso é válido para o nosso exemplo, após escolha das pré-condições e pós-condições adequadas para o código.

2.3 Escolha múltipla

Em algumas situações, temos vários ramos possíveis de código a executar, dependendo de um conjunto de condições. Infelizmente, o C++ não tem uma construção específica para esse caso (a não ser para uma situação especial a ser discutida depois), e precisamos usar vários `if/else` encadeados testando cada uma das condições.

Por exemplo, suponhamos que temos uma variável inteira `x` e queremos colocar na variável inteira `s` o valor da função sinal de `x` (isto é, se `x` é negativo, `s` deve receber `-1`, se `x` é positivo, `s` deve receber `1` e se `x` é zero, `s` deve receber zero). Neste caso temos três possíveis códigos a executar, e precisamos encader mais do que um `if/else`:

```
if (x < 0) {
    // x < 0
    s = -1;
}
else {
    // x >= 0
    if (x > 0) {
        // x > 0
        s = 1;
    }
    else {
        // x == 0
        s = 0;
    }
}
// s == sinal(x)
```

Note que quando fazemos `s=0` sabemos que `x == 0`, pois estamos dentro do `else` da condição `x > 0` e portanto `x` não é positivo; mas ele também não é negativo, pois essa condição só é testada quando `x >= 0`.

É tradicional e recomendado formatar essas cadeias de `if/else` um dentro do outro que fazem escolha entre várias opções da seguinte forma:

```
if (x < 0) {
    // x < 0
    s = -1;
}
else if (x > 0) {
    // x > 0
    s = 1;
}
else {
    // x == 0
    s = 0;
}
// s == sinal(x)
```

Isto é possível pois, como dito, os `{ }` são opcionais e podem ser omitidos quando há apenas um comando a ser executado. Esta é a única situação em que eu recomendo a omissão das chaves.

3 Repetição

Uma outra forma de controle de execução frequentemente necessária é a execução repetitiva de um mesmo conjunto de comandos. C++ possui diversos comandos que permitem realizar repetições.

3.1 Repetição simples

A forma mais simples é quando repetimos um conjunto de comandos enquanto uma condição for veradeira. Isto é parecido com a execução condicional, mas

ao invés de executar os comandos apenas uma vez, eles são repetidos até que a condição indicada seja falsa. A construção é similar à da execução condicional, mas substituindo o `if` por um `while`. Cada uma das execuções dos comandos repetidos é chamada de uma **iteração** da repetição.

Por exemplo, suponha que queremos encontrar o primeiro número da sequência de Fibonacci que seja maior do que 100. Como vocês sabem essa sequência começa com $f_0 = 1$ e $f_1 = 1$ e é definda de forma que cada número é a soma dos dois números anteriores na sequência ($f_{i+1}=f_i+f_{i-1}$). Podemos encontrar o número que desejamos calculando os números consecutivos da sequência até achar um que seja maior do que 100. Isto implica que teremos que repetir os comandos que calculam um número da sequência até achar um número maior que 100. O código pode ser como abaixo:

```
int anterior{1}, atual{1};
while (atual <= 100) {
    std::swap(anterior, atual);
    atual += anterior;
}
// atual tem o valor desejado
```

A operação `std::swap` troca o valor nas duas variáveis passadas (`anterior` e `atual`). A execução do código será: primeiro as variáveis `anterior` e `atual` são criadas e inicializadas em 1; em seguida, a condição `atual <= 100` é testada; como ela é verdadeira, então os dois comandos entre `{ }` sáo executados e volta-se a testar a condição. Isso se repete até que a condição seja falsa.

Para compreender e repetição em termos de alterações de estado e predicados sobre o estado, precisamos considerar o seguinte:

- Queremos repetir várias vezes os comandos;
- Queremos que essa repetição pare em algum momento.

Como para executar um conjunto de comandos precisamos garantir uma pré-condição para eles, e como esses comandos voltarão a executar, fica claro que essa condição deve ser mantida sem mudança enquanto a repetição estiver ocorrento. Porisso, denominamos esse predicado de **invariante da repetição**. O invariante deve ser válido antes da execução dos comandos da repetição toda vez que esses comandos forem repetidos. Para garantir que a repetição irá terminar em algum momento, precisamos garantir que o código está fazendo progresso em direção ao término, cada vez que os códigos repetidos são executados. Para garantir isso, especificamos um **variante da repetição**, que é um número inteiro não-negativo que possamos mostrar que sempre decresce quando os comandos da repetição são executados.

Ficamos então com a seguinte estrutura geral da repetição (onde `I` é o invariante de `V` é o variante):

```
// P
while (C) {
    // I
    // V
    S;
    // V'
```

```
    // Q1
}
// Q
```

A correção da repetição depende então do seguinte:

- A pós-condição `Q` especifica corretamente o que o código pretende atingir.
- A pré-condição `P` especifica corretamente o estado do sistema necessário para a execução correta do código.
- `P && C` implica `I` (isto é, se a pré-condição for satisfeita e a condição da repetição for verdadeira, então o invariante está estabelecido).
- Se `I` é válido e `S` é executado, então temos `Q1`.
- `Q1 && C` implica `I` (isto é, se ao voltar de uma execução da repetição a condição de repetição ainda for verdadeira, então o invariante continua verdadeiro).
- `Q1 & !C` implica `Q` (isto é, ao fim da execução de uma repetição, se a condição de repetição for falsa, então temos a pós-condição garantida).
- `P & !C` implica `Q` (isto é necessário pois se a condição `C` for inicialmente falsa, então os códigos da repetição nunca serão executados, e portanto a pós-condição tem que já ser verdadeira).
- `V' < V'` (isto é, o variante é decrementado pela execução de `S` nas situações apropriadas).

Vejamos como fica isso para nosso exemplo de cálculo do primeiro número da sequência de Fibonacci maior que 100:

```
int anterior{1}, atual{1};
// P: f[i] == atual && f[i-1] == anterior
while (atual <= 100) {
    // I: f[i] == atual && f[i-1] == anterior && atual <= 100
    // V: f[n] - atual
    std::swap(anterior, atual);
    // AUX: anterior == f[i] && atual == f[i-1]
    atual += anterior;
    // Q1: anterior == f[i] && atual == f[i] + f[i-1] == f[i+1]
}
// atual == f[n] && atual > 100 && f[n-1] <= 100
```

Coloquei os predicados anotados no código para facilitar, usando a notação `f[i]` para o i-ésimo número da sequência de Fibonacci e `f[n]` para o número que desejamos encontrar. Vejamos agora que os predicados anotados são corretos e que eles descrevem adequadamente o código:

- `Q` diz que `atual` é um número da sequência de Fibonacci, maior do que 100 e que o anterior a ele é menor ou igual a 100. Portanto ele é o valor procurado.
- `P` é válido para os dois números iniciais da sequência, devido a como as variáveis `atual` e anterior~ foram inicializadas.

- `P && C` implica `I` trivialmente.
- Se `I` é válido e executamos a troca de `atual` com `anterior` então `AUX` é trivialmente válido. Por outro lado, se temos `AUX` e executamos o incremento em `anterior`, `Q1` é válido por causa de como a sequência de Fibonacci é construida.
- `Q1 && C` implica `I`, simplesmente para um valor de `i` maior.
- `Q1 && !C` é o mesmo que `Q1 && atual > 100`, o que implica automaticamente `atual == f[n] && atual > 100` para $n = i+1~; por outro lado, se tivéssemos `f[n-1] > 100` a repetição teria terminado antes, portanto `f[n-1] <= 100` e vemos que `Q1 && !C` implica `Q`.[1]
- `P && !C` nunca acontece em nosso código (pois `1 <= 100`) e portanto não precisamos provar que isso garante a pós-condição (logicamente falando, se algo é falso ele implica qualquer coisa).
- O nosso variante é a diferença entre o valor do número desejado e a variável `atual`. Como `atual` passa sucessivamente por todos os números da sequência de Fibonacci e pára assim que encontrar um maior do que 100, então teremos sempre `atual<=f[n]` o que garante que `V` é maior ou igual a zero. Por outro lado, como os números da sequência são todos positivos, a execução de `atual+=anterior` incrementa o valor de `atual` e portanto decrementa o valor de `V`, garantindo que a repetição vai terminar.

3.2 Repetição regular

É frequente que seja necessária uma repetição com um padrão regular, isto é, onde as variações de uma iteração para outra são regulares. Vejamos um exemplo típico. Suponhamos que queremos criar um vetor de 100 números `double` tal que `v[i]` vale a metade de `i`. Isto pode ser feito com um `while` da seguinte forma:

```
std::vector<double> v(100);
int i{0};
while (i < 100) {
    v[i] = i / 2.0;
    ++i;
}
```

Note que temos a inicialização de uma variável auxiliar `i`, depois temos uma repetição que testa o valor dessa variável e incrementa o valor a cada iteração. Essa estrutura de repetição é muito frequente (com variações), e possui uma sintaxe especial em C++, usando o comando `for`.

```
std::vector<double> v(100);
int i;
for (i = 0; i < 100; ++i) {
    v[i] = i / 2.0;
}
```

[1] Para provar isso, precisamos provar que o código gera os números de Fibonacci na sequência correta. Não é difícil de fazer isso, mas não é nosso interesse.

Veja como dentro dos parêntesis do `for` temos não apenas a condição de repetição, mas três coisas separadas por `;` que são, respectivamente, a *inicialização*, a *condição* e o *incremento*. O código funciona da seguinte forma:

- A primeira coisa executada é o código de inicialização.
- Depois disso, testa-se a condição.
- Se a condição for falsa, a repetição termina.
- Se a condição for verdadeira, executam-se os comandos no bloco.
- Após executar os comandos do bloco, o incremento é executado.
- Depois do incremento, volta-se ao teste de condição.

No caso do nosso exemplo, vemos então que o `for` apresentado é equivalente ao código com `while` anterior. Isto é sempre válido: um código escrito com `for` sempre pode ser re-escrito com `while` (e vice-versa). A recomendação é usar `for` para situações onde o processo de repetição é controlado por ajustes regulares em uma variável (chamada *variável de indução*) a cada iteração. No nosso exemplo, a variável de indução é `i`, e o ajuste em cada iteração é o seu incremento de 1.

Em muitos casos, a variável de indução é necessária apenas durante a execução da repetição. Nesse caso, o C++ permite que você declare uma variável no próprio `for`, que será válida apenas durante a sua execução:

```
std::vector<double> v(100);
for (int i = 0; i < 100; ++i) {
    v[i] = i / 2.0;
}
```

Para a análise em termos de predicados, não há nenhuma novidade em relação à repetição com `while`, visto que eles são equivalentes.

3.3 Varredura de conteiner

Uma forma bastante comum de repetição é quando queremos fazer uma operação com todos os elementos de um dado conteiner. Um conteiner é uma estrutura que guarda vários elementos (normalmente todos do mesmo tipo). Um conteiner que já foi apresentado é o `vector`. Se queremos acessar todos os elementos em um conteiner, podemos usar um `for` como já discutido (ou o `while` equivalente) fazendo a variável de indução percorrer cada um dos índices do vetor:

```
for (int i = 0; i < 100; ++i) {
    std::cout << v[i];
}
```

Como esse tipo de operação é muito comum, o C++ tem uma versão simplificada de `for` que já se encarrega dos detalhes:

```
for (auto x: v) {
    std::cout << x;
}
```

Note que este `for` não tem `;` dentro dos parêntesis, mas sim a palvra-chave `auto` seguida de um nome de variável (no caso `x`), seguida de `:` seguido do nome de um conteiner (no caso o vetor de `double` que criamos antes). Com essa sintaxe, o C++ gera código para percorrer cada um dos elementos do vetor e colocar seu valor na variável `x`, e então executar os comandos do bloco com esse valor de `x`. As vantagens dessa sintaxe são:

- O código é mais abreviado.
- Não precisamos nos preocupar com terminação (a terminação é garantida)
- Não precisamos nos preocupar com controle do valor da variável de indução.
- A intenção do código fica clara mais facilmente, pois ao ver um `for` desses sabemos que estamos varrendo todos os elementos do conteiner, enquanto que no caso do outro `for` (ou de um `while`), precisamos analisas as instruções de início, término e incremento para garantir que é isso que está ocorrendo. Por exemplo, no código com o `for` usando a variável de indução `i` acima, como você sabe que todos os elementos do vetor estão sendo mostrados na tela? E se o vetor tiver mais do que 100 elementos?

Devido a isso, quando usado em situações apropriadas, esta sintaxe tende a reduzir a incidência de erros no código.

No exemplo anterior, os valores dos elementos do vetor estão apenas sendo lidos. Se quisermos alterar esses valores (ou ler e alterar), então a sintaxe é um pouco diferente:

```
for (auto &x: v) {
    x = 0;
}
```

Explicaremos mais tarde o significado do `&` usado neste código. Por enquanto, basta lembrar que o `&` precisa ser colocado se queremos alterar o valor do elemento acessado.

3.4 Repetição com teste posterior

Em algumas situações (não muito comuns), queremos executar os comandos a serem repetidos antes de testar a condição de repetição. Isto pode ocorrer por duas razões:

- Sabemos que pelo menos uma execução dos comandos é necessária; ou
- (mais comum) não conseguimos fazer o teste antes de executar os comandos.

Nessas situações, usamos o comando `do/while`, com a seguinte sintaxe:

```
do {
    S;
} while (C);
```

onde `S` são os comandos a repetir, e `C` é a condição de continuar na repetição.

Por exemplo, se queremos dar para o usuários escolher sim (s) ou não (n), deixando o usuário tentar novamente enquanto não for um desses caracteres, só podemos testar o término da repetição depois de ler pelo menos uma vez o caracter do usuário. Podemos fazer um código (ruim) da seguinte forma:

```
char escolha;
do {
    std::cout << "Escolha s ou n:";
    std::cin >> escolha;
} while (escolha != 's' && escolha != 'n');
```

Note como o teste só pode ser realizado depois de fazer a leitura. Este código poderia também ser feito com um `while` simplesmente garantindo que a condição fosse válida na primeira vez, por exemplo:

```
char escolha{'x'};
while (escolha != 's' && escolha != 'n') {
    std::cout << "Escolha s ou n:";
    std::cin >> escolha;
}
```

Muitos dos casos que requerem `do/while` podem ser adaptados para `while` de forma igualmente simples, e nesses casos você pode optar por usar um `while`.

Em termos de predicados teremos:

```
// P
do {
    // I
    // V
    S;
} while (C);
// Q
```

A grande diferença para o caso do `while` é que agora `P` deve implicar `I`, enquanto para o `while` era `P && C` implicando `I`.

4 Escolha baseada em valor

Uma última construção de controle de fluxo de execução é usada num caso especial de múltipla escolha de comandos em que a escolha é feita de acordo com um valor integral (inteiro ou algo parecido).

Por exemplo, ao invés de escrevermos um código do tipo:

```
if (opcao == 1) {
    sufixo = ".txt"s;
}
else if (opcao == 2) {
    sufixo = ".dat"s;
}
else if (opcao == 3) {
    sufixo = ".doc"s;
}
```

```
else {
    sufixo = ""s;
}
```

podemos usar a construção `switch` da seguinte forma:

```
switch (opcao) {
case 1:
    sufixo = ".txt"s;
    break;
case 2:
    sufixo = ".dat"s;
    break;
case 3:
    sufixo = ".doc"s;
    break;
default:
    sufixo = ""s;
}
```

Entre parêntesis no `switch` deve estar uma expressão que resulte em um valor integral. A execução será desviada para os comandos que seguem o `case` correspondente ao valor da expressão; os comandos seguintes são executados até encontrar um `break`. Se o valor da expressão não corresponde ao valor de nenhum `case`, então os comandos após o `default` são executados. As vantagens de usar `switch` ao invés de diversos `if/else` são:

- É mais fácil de ver que se está fazendo escolha simplesmente de acordo com o valor de uma variável. Nos conjuntos `if/else` teríamos que verificar todas as condições para chegar a essa conclusão.
- O código gerado pelo compilador é normalemente mais eficiente, pois nos `if/else` o código tem que testar uma das condições depois da outra, enquanto no `switch` ele pode desviar para o local correto assim que souber o valor da expressão.

Em contrapartida, conjuntos de `if/else` são mais flexíveis em como escolher o código a executar, e portanto nem todos os casos de múltipla escolha podem ser resolvidos por `switch`.