Seleção, enumerações e estruturas

Marco A L Barbosa
malbarbo.pro.br

Departamento de Informática
Universidade Estadual de Maringá

Antes de estudarmos instruções de seleção, vamos analisar como um computador “executa” um programa.

```
1 int n = 10;
2 int a = n + 2;
3 cout << a;
```

Qual o valor exibido pelo programa? 12.

Nesse exemplo, o computador executa uma linha após a outra, por isso chamamos essa "estrutura" de **código sequencial** (sequenciação).

Qual é a ordem que as linhas são executadas? 1, 2, 3.

```
1 int n;
2 int a = 10;
3 n = a;
4 cout << n;
```

Qual o valor exibido pelo programa? 10.

```
1 int n = 5;
2 int a = 10;
3 n = a;
4 cout << n;
```

Qual o valor exibido pelo programa? 10.

No primeiro caso, a variável **n** é declarada mas não é inicializada na linha 1. Quando a linha 3 é executada, o **n** é inicializado com o valor que está armazenado em **a**, que é 10. Na linha 4 o valor de **n** é exibido.

Já no segundo caso, a variável **n** é declarada e inicializada na linha 1 com o valor 5. Na linha 3 o valor de **a** é copiado para **n** (célula de memória com o nome **n**), substituindo o valor 5 por 10. Na linha 4 o valor de **n** é exibido.

```
1 int n = 1;
2 int a = n + 1;
3 n = a + 3;
4 cout << n;
```

Qual o valor exibido pelo programa? 5.

Para entender esse resultado devemos executar o programa como um computador!

- Na linha 1 alocamos uma célula de memória e nomeamos ela de **n** e armazenamos o valor 1 nessa célula;
- Na linha 2 alocamos uma célula de memória e nomeamos ela de **a**. Lemos o valor de **n**, somamos 1 e armazenamos o resultado (2) em **a**;
- Na linha 3 lemos o valor de **a** e somamos com 3, depois armazenamos o resultado (5) em **n**;
- Na linha 4 lemos e exibimos o valor de **n**.

```
1 int n = 1;
2 n = n + 2;
3 n = n + 5;
4 cout << n;
```

Qual o valor exibido pelo programa? 8.

Executando como um computador:

- Na linha 1 alocamos uma célula de memória e nomeamos ela de **n** e armazenamos o valor 1 nessa célula;
- Na linha 2 lemos o valor de **n** e somamos com 2, depois armazenamos o resultado (3) em **n**;
- Na linha 3 lemos o valor de **n** e somamos com 5, depois armazenamos o resultado (8) em **n**;
- Na linha 4 lemos e exibimos o valor de **n**.

Pergunta

Em todos os programas que escrevemos até agora as linhas são sempre executadas na sequência? Não!

Quando chamamos uma função, o fluxo de execução é desviado para o início do corpo da função. Quando uma função executa uma instrução **`return`**, o fluxo de execução volta para onde ele estava antes da chamada da função.

Execução de funções

```
1  int dobro_mais_um(int n) {
2      int a = 2 * n;
3      return a + 1;
4  }
5
6  int main() {
7     int b = 5;
8     int c = dobro_mais_um(b + 4) + 1;
9     cout << c;
10 }
```

Qual o valor exibido pelo programa? 20.

Qual é a ordem que as linhas são executadas? (Feito em sala)

- 7
- 8 (`b + 4`)
- 2
- 3
- 8 (soma do resultado da função (19) com 1 e atribuição do valor a `c`)
- 9

```cpp
int soma_1(int n) {
    int a = n + 1;
    return a;
}

int main() {
   int n = 2;
   int a = 3;
   n = soma_1(a);
   cout << n << " " << a;
}
```

Qual o valor exibido pelo programa?

Não tente “executar” a chamada da função, pense apenas no seu propósito, sem olhar para o seu corpo.

Então, qual o valor exibido pelo programa? 4 e 3.

Além das chamadas de funções, temos outras instruções que alteram o fluxo de execução do programa.

Uma dessas instruções é o **`if else`** (se e senão em inglês).

O **`if else`** é uma **instrução de seleção** e sua forma geral é:

```
if (condição) {
    instruções então;
} else {
    instruções senão;
}
```

Como a instrução **`if else`** é executada? O computador avalia a condição e verifica o resultado. Se o resultado for **`true`**, então as instruções do bloco “instruções então” são executadas, senão (o resultado é **`false`**), as instruções do bloco “instruções senão” são executadas.

Exemplo

```
1  int a = 10;
2  int b = 20;
3  int m;
4  if (a > b) {
5      m = a;
6  } else {
7      m = b;
8  }
9  cout << m;
```

Qual o valor exibido pelo programa? 20.

Em que ordem as linhas são executas para gerar esse resultado? 1, 2, 3, 4, 7, 9.

```
1  int a = 15;
2  int b = 8;
3  int m;
4  if (a > b) {
5      m = a;
6  } else {
7      m = b;
8  }
9  cout << m;
```

Qual o valor exibido pelo programa? 15.

Em que ordem as linhas são executas para gerar esse resultado? 1, 2, 3, 4, 5, 9.

Valor máximo

Vamos voltar ao exemplo do valor máximo.

Projete uma função que encontre o valor máximo entre dois números dados.

Especificação e implementação

Especificação

```
// Encontra o valor máximo entre a e b.
int maximo(int a, int b)
{
    return 0;
}

examples
{
    // a é máximo
    check_expect(maximo(20, 10), 20);
    // b é máximo
    check_expect(maximo(5, 10), 10);
    check_expect(maximo(5, 5), 5);
}
```

Implementação

Até agora, todas as funções que projetamos tinham apenas uma “forma” de gerar o resultado.

Na função máximo, existem duas “formas” para a resposta, ou a resposta é **a**, ou a resposta é **b**.

Como escolher a resposta para a função? Avaliando um condição: se o valor de **a** for maior do que o valor de **b**, então a resposta é o valor de **a**, senão a resposta é o valor de **b**.

Quando a resposta depende de uma ou mais condições, usamos uma instrução de seleção!

```
1  int maximo(int a, int b) {
2      int max;
3      if (a > b) {
4          max = a;
5      } else {
6          max = b;
7      }
8      return max;
9  }
10
11 int main() {
12     int a = maximo(10, 8);
13     int b = maximo(14, 20);
14     cout << a << " " << b;
15 }
```

Qual é a saída exibida pelo programa? 10 e 20.

Em que ordem as linhas são executas para gerar esse resultado?

12, 2, 3, 4, 8, 12,

13, 2, 3, 6, 8, 13,

14

Máximo 3

Projete uma função que encontre o valor máximo entre três números.

Análise

- Encontrar o valor máximo entre três número dados

Tipos de dados

- Os valores serão números inteiros

Especificação

Especificação

```
// Encontra o valor máximo entre a, b e c.
int maximo3(int a, int b, int c) {
   return 0;
}
examples {
    // a é máximo
    check_expect(maximo3(20, 10, 12), 20);
    check_expect(maximo3(20, 12, 10), 20);
    check_expect(maximo3(20, 12, 12), 20);
    check_expect(maximo3(20, 20, 20), 20);
    // b é máximo
    check_expect(maximo3(5, 12, 3), 12);
    check_expect(maximo3(3, 12, 5), 12);
    check_expect(maximo3(5, 12, 5), 12);
    // c é máximo
    check_expect(maximo3(4, 8, 18), 18);
    check_expect(maximo3(8, 4, 18), 18);
    check_expect(maximo3(8, 8, 18), 18);
}
```

Implementação

Quantas “formas” de reposta temos? 3. Ou a resposta é **a**, ou a resposta é **b**, ou a resposta é **c**.

Se temos respostas diferentes, então a resposta depende de uma ou mais condições. Então, usamos instruções de seleção.

Qual é a condição para a resposta ser **a**?

`a >= b && a >= c`

Qual é a condição para a resposta ser **b**?

`b >= a && b >= c`

Qual é a condição para a resposta ser **c**?

`c >= a && c >= b`

Implementação

Especificação

```
// Encontra o valor máximo entre a, b e c.
int maximo3(int a, int b, int c) {
   return 0;
}
examples {
    // a é máximo
    check_expect(maximo3(20, 10, 12), 20);
    check_expect(maximo3(20, 12, 10), 20);
    check_expect(maximo3(20, 12, 12), 20);
    check_expect(maximo3(20, 20, 20), 20);
    // b é máximo
    check_expect(maximo3(5, 12, 3), 12);
    check_expect(maximo3(3, 12, 5), 12);
    check_expect(maximo3(5, 12, 5), 12);
    // c é máximo
    check_expect(maximo3(4, 8, 18), 18);
    check_expect(maximo3(8, 4, 18), 18);
    check_expect(maximo3(8, 8, 18), 18);
}
```

Implementação

```
// Encontra o valor máximo entre a, b e c.
int maximo3(int a, int b, int c) {
    int max;
    if (a >= b && a >= c) {
        max = a;
    } else if (b >= a && b >= c) {
        max = b;
    } else if (c >= a && c >= b) {
        max = c;
    }
    return max;
}
```

Verificação: ok.

Revisão

Dependendo do compilador, podemos obter o aviso

```
x.cpp:9:16: warning: variable 'max' is used uninitialized whenever 'if'
condition is false [-Wsometimes-uninitialized]
```

Por que? Se nenhuma das três condições for verdadeira, `m` não será inicializado. Mas nós sabemos que sempre pelo menos uma das condições é verdadeira! Então vamos ajustar o código!

```c
// Encontra o valor máximo entre a, b e c.
int maximo3(int a, int b, int c) {
    int max;
    if (a >= b && a >= c) {
        max = a;
    } else if (b >= a && b >= c) {
        max = b;
    } else { // c >= a && c >= b
        max = c;
    }
    return max;
}
```

Podemos fazer uma implementação diferente? Sim.

Ao invés de “perguntar” duas coisas por vez, podemos perguntar apenas uma coisa por vez e fazer uma “sequência” de decisões.

Se `a >= b` é **`true`**, quais valores podem ser o máximo? Os valores de `a` e `c`. E como descobrimos quem é o máximo entre `a` e `c`? Fazendo outra seleção.

Se `a >= b` é **`false`**, quais valores podem ser o máximo? Os valores de `b` e `c`. E como descobrimos quem é o máximo entre `b` e `c`? Fazendo outra seleção.

Solução alternativa

```c
int maximo3(int a, int b, int c) {
    int max;
    if (a >= b) {
        if (a >= c) {
            max = a;
        } else {
            max = c;
        }
    } else {
        if (b >= c) {
            max = b;
        } else {
            max = c;
        }
    }
    return max; }
```

```c
int maximo3(int a, int b, int c) {
    int max;
    if (a >= b && a >= c) {
        max = a;
    } else if (b >= a && b >= c) {
        max = b;
    } else {
        max = c;
    }
    return m;
}
```

Qual versão é mais fácil de entender? A primeira...

Podemos melhorar ainda? Sim!

```
1  int maximo3(int a, int b, int c) {
2      int max;
3      if (a >= b) {
4          if (a >= c) {
5              max = a;
6          } else {
7              max = c;
8          }
9      } else {
10         if (b >= c) {
11             max = b;
12         } else {
13             max = c;
14         }
15     }
16     return max;
17 }
```

Qual o propósito do bloco das linhas 4 a 8?

Encontrar o máximo entre **a** e **c**.

Qual o propósito do bloco das linhas 10 a 14?

Encontrar o máximo entre **b** e **c**.

Já temos uma função para encontrar o máximo entre dois números? Sim! A função `maximo` que fizemos anteriormente.

```
1  int maximo3(int a, int b, int c) {
2      int max;
3      if (a >= b) {
4          max = maximo(a, c);
5      } else {
6          max = maximo(b, c);
7      }
8      return max;
9  }
```

Qual o propósito da estrutura de seleção da linha 3? Encontrar o máximo entre **a** e **b**... Nós já temos uma função para fazer isso!

```c
int maximo3(int a, int b, int c) {
    return maximo(maximo(a, b), c);
}
```

Estratégia de implementação

Encontrar o máximo entre **a** e **b** e depois o máximo entre o resultado e **c**.

Ponto final

Em um determinado programa é necessário que o texto digitado pelo usuário termine com um ponto. Projete uma função que coloque um ponto final em um texto se ele ainda não terminar com ponto.

Análise

- Colocar um ponto final em um texto caso ele ainda não termine com ponto.

Definição dos tipos de dados

- O texto é representado por uma string.

Ponto final - Especificação

```
// Produz s se s é vazia ou termina com um ponto
// final, caso contrário produz s seguido de um
// ponto final.
string coloca_ponto_se_necessario(string s) {
    return s;
}
examples {
    // Coloca o ponto
    // s.substr(s.length() - 1, 1) == "." é false
    // s + "."
    check_expect(coloca_ponto_se_necessario("casa"), "casa.");
    // s.substr(s.length() - 1, 1) == "." é false
    check_expect(coloca_ponto_se_necessario("eu tambem"), "eu tambem.");
    // Não coloca o ponto
    // s.subsr(s.length() - 1, 1) == "." é true
    // s
    check_expect(coloca_ponto_se_necessario("casa."), "casa.");
    // s == "" é true
    check_expect(coloca_ponto_se_necessario(""), "");
}
```

Ponto final - Implementação

```
string coloca_ponto_se_necessario(string s) {
    string r;
    // s é vazia ou o último caractere é "."
    if (s == "" || s.substr(s.length() - 1, 1) == ".") {
        r = s;
    } else {
        r = s + ".";
    }
    return r;
}
```

Verificação: ok.

Revisão: ok.

Álcool ou Gasolina?

Depois que você fez o programa para o André, a Márcia, amiga em comum de vocês, soube que você está oferecendo serviços desse tipo e também quer a sua ajuda. O problema da Márcia é que ela sempre tem que fazer a conta manualmente para saber se deve abastecer o carro com álcool ou gasolina. A conta que ela faz é verificar se o preço do álcool é até 70% do preço da gasolina, se sim, ela abastece o carro com álcool, senão ela abastece o carro com gasolina. Você pode ajudar a Márcia também?

Análise

- Determinar o combustível que será utilizado. Se o preço do álcool for até 70% do preço da gasolina, então deve-se usar álcool, senão gasolina.

Definição de tipos de dados

- O preço do litro do combustível será representado por um número positivo com três casas decimais;
- O tipo de combustível será representado por uma string.

Discutimos em sala o projeto desse programa.

```
// Combustivel é "alcool" ou "gasolina".

// Indica o combustível que deve ser utilizado no abastecimento. Produz
// "alcool" se preco_alcool for menor ou igual a 70% do preco_gasolina,
// caso contrário produz "gasolina".
string indica_combustivel(double preco_alcool, double preco_gasolina) {
    return "alcool";
}

examples {
    // Combustível é alcool
    // 4.000 <= 0.7 * 6.000 é true
    check_expect(indica_combustivel(4.000, 6.000), "alcool");
    // 3.500 <= 0.7 * 5.000 é true
    checK_expect(indica_combustivel(4.200, 6.000), "alcool");
    // Combustível é gasolina
    // 4.000 <= 0.7 * 5.000 é false
    check_expect(indica_combustivel(4.000, 5.000), "gasolina");
}
```

O resultado depende de uma condição? Ou seja, existe mais que uma forma para a resposta? Sim! Então usamos seleção.

```
// Indica o combustível que deve ser utilizado no abastecimento. Produz
// "alcool" se preco_alcool for menor ou igual a 70% do preco_gasolina,
// caso contrário produz "gasolina".
string indica_combustivel(double preco_alcool, double preco_gasolina) {
    string combustivel;
    if (preco_alcool <= 0.7 * preco_gasolina) {
        combustivel = "alcool";
    } else {
        combustivel = "gasolina";
    }
    return combustivel;
}
```

Verificação: ok.

Revisão: string não parece ser o tipo apropriado. Pela assinatura da função, “qualquer” string pode ser dada como resposta, mas de fato apenas dois valores são possível: “alcool” e “gasolina”. Podemos melhorar? Sim.

Em um **tipo enumerado** todos os valores válidos para o tipo são enumerados explicitamente.

A forma geral para definir tipos enumerados é

```
enum NomeDoTipo {
    Valor1,
    Valor2,
    ...,
};
```

```
// O tipo do combustível utilizado no abastecimento.
enum Combustivel {
    Alcool,
    Gasolina,
};
```

Observações

- Sempre vamos adicionar um comentário sobre o propósito do tipo;
- É necessário finalizar a declaração com um ponto e vírgula;

Uma variável do tipo `Combustivel` só pode armazenar o valor `Alcool` ou `Gasolina`, se tentarmos atribuir um valor diferente, o compilador indicará um erro.

```
Combustivel c = "Alcool";
```

Erro

```
x.cpp:13:21: error: cannot convert ‘const char*’ to ‘Combustivel’ in initialization
   13 |     Combustivel c = "Alcool";
      |                     ^~~~~~~~
```

Quando usar tipos enumerados?

Quando todos os valores válidos para o tipo podem ser nomeados.

Por que utilizar tipos enumerados?

Para expressar mais claramente o propósito do código e evitar a utilização de valores inválidos (como `"alcoo"` em uma variável string que representa o tipo do combustível).

Revisão do exemplo

```
Combustivel indica_combustivel(double preco_alcool, double preco_gasolina)
{
    Combustivel combustivel;
    if (preco_alcool <= 0.7 * preco_gasolina) {
        combustivel = Alcool;
    } else {
        combustivel = Gasolina;
    }
    return combustivel;
}
examples
{
    check_expect(indica_combustivel(4.000, 5.000), Gasolina);
    check_expect(indica_combustivel(4.000, 6.000), Alcool);
    check_expect(indica_combustivel(3.500, 5.000), Alcool);
}
```

Exemplo

Projete uma função que receba como entrada a cor atual de um semáforo de trânsito e devolva a próxima cor que será ativada (considere um semáforo com três cores: verde, amarelo e vermelho).

Fizemos o projeto desse programa durante a aula obtivemos o seguinte resultado:

```
// Representa a cor de um semáforo.
enum Cor {
    Verde,
    Vermelho,
    Amarelo,
};
// Produz a próximo cor de um semáforo
// que está com a cor c.
Cor proxima_cor(Cor c) {
    Cor proxima;
    if (c == Verde) {
        proxima = Amarelo;
    } else if (c == Vermelho) {
        proxima = Verde;
    } else { // c == Amarelo
        proxima = Vermelho;
    }
    return proxima;
}
```

```
examples
{
    check_expect(proxima_cor(Verde), Amarelo);
    check_expect(proxima_cor(Vermelho), Verde);
    check_expect(proxima_cor(Amarelo), Vermelho);
}
```

Verificação: Ok.

Revisão: vamos usar uma construção nova, o **`switch`/`case`**.

A sintaxe simplificada do **switch/case** é

```
switch (expressão) {
    // zero ou mais casos
    case caso1:
        instruções;
        break;
    // ...
    case cason:
        instruções;
        break;
    // opcional
    default:
        instruções;
        break;
}
```

Onde `expressão` precisa gerar um valor do tipo inteiro ou enumerado.

A instrução **switch/case** funciona da seguinte forma:

A `expressão` é avaliada e seu valor é comparado com cada caso na sequência.

Quando um caso que tem o mesmo valor do resultado da expressão é encontrado, as `instruções` daquele caso são executadas até encontrar um `break`, quando então a instrução **switch/case** termina e o programa continua a execução com a próxima instrução após o **switch/case**.

Se o valor da expressão não é igual a nenhum caso, então as instruções da cláusula **default** são executadas.

Quando utilizar o **switch**/**case**?

Quando precisamos analisar o valor de um tipo enumerado ou quando precisamos analisar um conjunto de valores inteiros específicos (ex 1, 3, 4, 5).

Qual é a vantagem de utilizar **switch**/**case** ao invés de uma sequência de **if**s?

O código fica mais fácil de ler.

O compilador (usando a opção `-Wall`) gera um aviso se esquecermos de algum valor na análise de valores enumerados.

Escrevendo a função `proxima_cor` para utilizar o **`switch`**/**`case`** obtemos

```
Cor proxima_cor(Cor c) {
    Cor proxima;
    switch (c) {
        case Verde:
            proxima = Amarelo;
            break;
        case Vermelho:
            proxima = Verde;
            break;
        case Amarelo:
            proxima = Vermelho;
            break;
    }
    return proxima;
}
```

Exemplo

Em um determinado programa é necessário exibir para o usuário o tempo que uma operação demorou. Esse tempo está disponível em segundos, mas exibir essa informação em segundos para o usuário pode não ser interessante, afinal, ter uma noção razoável de tempo para 14678 segundos é difícil!

a) Projete uma função que converta uma quantidade de segundos para uma quantidade de horas, minutos e segundos equivalentes.

b) Projete uma função que converta uma quantidade de horas, minutos e segundos em uma string amigável para o usuário. A string não deve conter informações sobre tempo que são zeros (por exemplo, não deve informar 0 minutos).

Análise

- Converter uma quantidade de segundos em horas, minutos e segundos.

Definição de tipos de dados

- Os segundos da entrada serão representados com números inteiros positivos
- A saída são três números inteiros positivos... As funções em C++ só podem produzir um valor de saída, como proceder? Vamos criar um novo tipo de dado que agrupa esses três valores.

Tipos de dados

Vamos relembrar alguns tipos de dados que utilizamos até agora:

- Tipos atômicos pré-definidos (primitivos) na linguagem: `int`, `double`, `bool`
- Tipos atômicos definidos em bibliotecas: `string`
- Tipos atômicos enumerados definidos pelo usuário: `enum`

Esses tipos são chamados atômicos porque não são compostos por partes.

Podemos criar novos tipos de dados a partir de tipos existentes.

Uma forma de fazer isso é através de tipos estruturas.

Um **tipo estrutura** é um tipo de dado composto por um conjunto fixo de campos com nome e tipo.

A forma geral para definir um tipo estrutura é

```
struct NomeDoTipo {
    Tipo1 campo1;
    Tipo2 campo2;
    ...
};
```

Tipos estruturas

Podemos definir um novo tipo para representar um tempo da seguinte forma

```
// Representa o tempo de duração de um evento.
// horas, minutos e segundos devem ser positivos.
// minutos e segundos devem ser menores que 60.
struct Tempo {
    int horas;
    int minutos;
    int segundos;
};
```

Observações

- Assim como para definição de tipos enumerados, sempre vamos adicionar um comentário sobre o propósito do tipo;
- É necessário finalizar a declaração com um ponto e vírgula.

Para inicializar uma variável de um tipo estrutura, especificamos os valores dos campos (na ordem que eles foram declarados) entre chaves e separados por vírgula

```
Tempo t1 = {0, 20, 10};
Tempo t2 = {4, 0, 20};
```

Para exibir um valor de um tipo estrutura podemos utilizar a função `repr` da biblioteca `bscpp`

```
cout << repr(t1) << endl; // Tempo {0, 20, 10}
```

Como valores do tipo `Tempo` são compostos de outros valores (partes), podemos acessar e alterar cada valor de forma separada.

```
Tempo t1 = {0, 20, 10};

cout << t1.minutos << endl; // exibe 20

t1.horas = 3; // muda a quantidade de horas para 3
```

Agora podemos voltar para o nosso problema.

Especificação

```
// Converte a quantidade segundos para o tempo equivalente em
// horas, minutos e segundos. A quantidade de segundos e
// minutos da resposta é sempre menor que 60.
Tempo segundos_para_tempo(int segundos) {
    return Tempo {0, 0, 0};
}

examples {
    // 160 / 60 -> 2
    // 160 % 60 -> 40
    check_expect(segundos_para_tempo(160), (Tempo {0, 2, 40}));
    // 3760 / 3600 -> 1
    // 3760 % 3600 -> 160 segundos que sobraram
    // 160 / 60 -> 2
    // 160 % 60 -> 40
    check_expect(segundos_para_tempo(3760), (Tempo {1, 2, 40}));
}
```

Implementação

```
Tempo segundos_para_tempo(int segundos)
{
    int h = segundos / 3600;
    // segundos que não foram
    // convertidos para hora
    int restantes = segundos % 3600;
    int m = restantes / 60;
    int s = restantes % 60;
    return Tempo {h, m, s};
}
```

Verificação: ok

Revisão: ok

Quando utilizamos estruturas?

Quando a informação consiste de dois ou mais itens que juntos descrevem uma entidade.

Exemplo

O novo problema inicial era:

Em um determinado programa é necessário exibir para o usuário o tempo que uma operação demorou. Esse tempo está disponível em segundos, mas exibir essa informação em segundos para o usuário pode não ser interessante, afinal, ter uma noção razoável de tempo para 14678 segundos é difícil!

a) Projete uma função que converta uma quantidade de segundos para uma quantidade de horas, minutos e segundos equivalentes.

b) Projete uma função que converta uma quantidade de horas, minutos e segundos em uma string amigável para o usuário. A string não deve conter informações sobre o tempo que são zeros (por exemplo, não deve informar 0 minutos).

Agora vamos fazer o item b. (Projeto desenvolvido em aula.)

```
// Converte t em uma mensagem para o usuário. Cada componente de t aparece
// com a sua unidade, mas se o valor do componente for 0, ele não aparece na
// mensagem. Os componentes são separados com " e " ou ", " respeitando as
// regras do Português. Se t for {0, 0, 0}, devolve "0 segundo(s)".
string tempo_para_string(Tempo t)
{
   return "";
}
```

Especificação

```
examples {
    // horas == 0 && minutos == 0
    check_expect(tempo_para_string(Tempo {0, 0, 0}), "0 segundo(s)");
    check_expect(tempo_para_string(Tempo {0, 0, 1}), "1 segundo(s)");
    check_expect(tempo_para_string(Tempo {0, 0, 10}), "10 segundo(s)");
    // horas == 0 && minutos != 0 && segundos != 0
    check_expect(tempo_para_string(Tempo {0, 1, 20}), "1 minuto(s) e 20 segundo(s)");
    // horas == 0 && minutos != 0 && segundos == 0
    check_expect(tempo_para_string(Tempo {0, 2, 0}), "2 minuto(s)");
    // horas != 0 && minutos != 0 && segundos != 0
    check_expect(tempo_para_string(Tempo {1, 2, 1}), "1 hora(s), 2 minuto(s) e 1 segundo(s)");
    // horas != 0 && minutos == 0 && segundos != 0
    check_expect(tempo_para_string(Tempo {4, 0, 25}), "4 hora(s) e 25 segundo(s)");
    // horas != 0 && minutos != 0 && segundos == 0
    check_expect(tempo_para_string(Tempo {2, 4, 0}), "2 hora(s) e 4 minuto(s)");
    // horas != 0 && minutos == 0 && segundos == 0
    check_expect(tempo_para_string(Tempo {3, 0, 0}), "3 hora(s)");
}
```

A implementação direta usando as condições combinadas fica com exercício.

A implementação a seguir usando condições aninhadas foi desenvolvida em sala.

Implementação

```
string tempo_para_string(Tempo t) {
    string h = to_string(t.horas) + " hora(s)";
    string m = to_string(t.minutos) + " minuto(s)";
    string s = to_string(t.segundos) + " segundo(s)";
    string msg = "";
    // nos exemplos são 7 casos distintos
    if (t.horas > 0) {
        if (t.minutos > 0) {
            if (t.segundos > 0) {
                msg = h + ", " + m + " e " + s;
            } else {
                msg = h + " e " + m;
            }
        } else if (t.segundos > 0) {
            msg = h + " e " + s;
        } else {
            msg = h;
        }
    } else if (t.minutos > 0) {
        if (t.segundos > 0) {
            msg = m + " e " + s;
        } else {
            msg = m;
        }
    } else {
        msg = s;
    };
    return msg; }
```

Implementação alternativa

```
string tempo_para_string(Tempo t) {
    // usado para separar cada componente de t
    string sep = "";
    string msg = "";
    if (t.segundos > 0) {
        sep = " e ";
        msg = to_string(t.segundos) + " segundo(s)";
    }

    if (t.minutos > 0) {
        msg = to_string(t.minutos) + " minuto(s)" + sep + msg;
        if (t.segundos > 0) {
            sep = ", ";
        } else {
            sep = " e ";
        }
    }

    if (t.horas > 0) {
        msg = to_string(t.horas) + " hora(s)" + sep + msg;
    }

    if (msg == "") {
        msg = "0 segundo(s)";
    }

    return msg;
}
```

Exercício

Modifique a especificação e implementação da função anterior para que o plural dos componentes fique de acordo com o Português.

Exemplo

Segundo a Wikipédia, um pixel é o menor elemento de um dispositivo de exibição, como por exemplo, um monitor, ao qual é possível atribuir uma cor. Nos monitores atuais, os pixels são organizados em linhas e colunas, de maneira a formar a imagem exibida. Cada pixel pode ser referenciado por uma coordenada, que é o número da linha e coluna que ele aparece. Por exemplo, em um monitor de 1080 linhas e 1920 colunas, o pixel no canto superior esquerdo está na posição (0, 0), enquanto o pixel no canto inferior direito está na posição (1079, 1919).

Em um ambiente gráfico com janelas, quando um usuário faz um clique com o mouse é necessário identificar em qual janela ocorreu o clique. Considerando que o espaço que uma janela ocupa pode ser representada pela coordenada do canto superior esquerdo e pela quantidade de pixels da largura e da altura da janela

a) Projete uma função que receba como parâmetros as informações sobre uma janela e um clique do mouse e determine se o clique aconteceu sobre a janela.

b) Projete uma função que verifique se os espaços de duas janelas se sobrepõem.

Projeto desenvolvido em aula.

Definição de tipos de dados

```c
// Representa o espaço que uma janela ocupa em um ambiente gráfico.
//
// A coordenada (x, y) descreve a posição do canto superior esquerdo.
// A largura representa a quantidade de pixels à direita de (x, y)
// e a altura representa a quantidade de pixels abaixo de (x, y).
//
// Os valores da largura e altura devem ser maiores que zero.
struct Janela {
    int x;
    int y;
    int largura;
    int altura;
};

// Representa a posição de um clique em um ambiente gráfico.
// Os valores de x e y devem ser maiores que 0 e menores do que as dimensões do
// ambiente.
struct Clique {
    int x;
    int y;
};
```

Especificação

```
// Devolve true se o clique c está dentro do espaço da janela j, false contrário.
bool dentro_janela(Janela j, Clique c)
{
   return false;
}
```

Especificação

```
examples {
    //  x = 100, y = 100, largura = 300, altura = 200
    //
    //        p5
    //      +------------+
    //  p4  | p1         | p2
    //      |            |
    //      +------------+
    //        p3
    Janela janela = { 100, 100, 300, 200 };
    // p1 - dentro da janela
    check_expect(dentro_janela(janela, { 150, 150 }), true);
    // p2 - dentro do espaço da altura e depois do espaço da largura
    check_expect(dentro_janela(janela, { 600, 150 }), false);
    // p3 - depois do espaço da altura e dentro do espaço da largura
    check_expect(dentro_janela(janela, { 150, 300 }), false);
    // p4 - dentro do espaço da altura e antes do espaço da largura
    check_expect(dentro_janela(janela, { 150, 50 }), false);
    // p5 - antes do espaço da altura e dentro do espaço da largura
    check_expect(dentro_janela(janela, { 150, 50 }), false);
```

```
    // canto superior esquerdo
    check_expect(dentro_janela(janela, { 100, 100 }), true);
    // canto superior direito
    check_expect(dentro_janela(janela, { 399, 100 }), true);
    check_expect(dentro_janela(janela, { 400, 100 }), false);
    // canto inferior direito
    check_expect(dentro_janela(janela, { 399, 299 }), true);
    check_expect(dentro_janela(janela, { 400, 299 }), false);
    check_expect(dentro_janela(janela, { 399, 300 }), false);
    check_expect(dentro_janela(janela, { 400, 300 }), false);
    // canto inferior esquerdo
    check_expect(dentro_janela(janela, { 100, 299 }), true);
    check_expect(dentro_janela(janela, { 100, 300 }), false);
}
```

```
// Devolve true se o clique c está dentro do espaço da janela j, false contrário.
bool dentro_janela(Janela j, Clique c)
{
    // c.x está dentro do espaço da largura e c.y dentro do espaço da altura
    return j.x <= c.x && c.x < (j.x + j.largura) && j.y <= c.y && c.y < (j.y + j.altura);
}
```

Verificação: ok

Revisão: ok

Especificação

```
// Produz true se o espaço das janelas a e b se soprepõem, false caso contrário.
bool janelas_soprepoem(Janela a, Janela b)
{
    return false;
}

examples
{
    // fixa (eixo y): a janela a vem antes da janela b
    // variável: posição da borda direita de a
    check_expect(janelas_soprepoem({  10, 20, 100, 200 }, { 300, 400, 50, 100 }), false);
    check_expect(janelas_soprepoem({ 210, 20, 100, 200 }, { 300, 400, 50, 100 }), false);
    check_expect(janelas_soprepoem({ 310, 20, 100, 200 }, { 300, 400, 50, 100 }), false);
    check_expect(janelas_soprepoem({ 410, 20, 100, 200 }, { 300, 400, 50, 100 }), false);
    // fixa: (eixo y) interseção da parte de baixo de a com a parte de cima de b
    // variável: posição da borda direita de a
    check_expect(janelas_soprepoem({  10, 250, 100, 200 }, { 300, 400, 50, 100 }), false);
    check_expect(janelas_soprepoem({ 210, 250, 100, 200 }, { 300, 400, 50, 100 }), true);
    check_expect(janelas_soprepoem({ 310, 250, 100, 200 }, { 300, 400, 50, 100 }), true);
    check_expect(janelas_soprepoem({ 410, 250, 100, 200 }, { 300, 400, 50, 100 }), false);
```

```
    // fixa: (eixo y) interseção da parte de cima de a com a parte de baixo de b
    // variável: posição da borda direita de a
    check_expect(janelas_soprepoem({  10, 450, 100, 200 }, { 300, 400, 50, 100 }), false);
    check_expect(janelas_soprepoem({ 210, 450, 100, 200 }, { 300, 400, 50, 100 }), true);
    check_expect(janelas_soprepoem({ 310, 450, 100, 200 }, { 300, 400, 50, 100 }), true);
    check_expect(janelas_soprepoem({ 410, 450, 100, 200 }, { 300, 400, 50, 100 }), false);
    // fixa: (eixo y) a janela a vem depois da janela b
    // variável: posição da borda direita de a
    check_expect(janelas_soprepoem({  10, 550, 100, 200 }, { 300, 400, 50, 100 }), false);
    check_expect(janelas_soprepoem({ 210, 550, 100, 200 }, { 300, 400, 50, 100 }), false);
    check_expect(janelas_soprepoem({ 310, 550, 100, 200 }, { 300, 400, 50, 100 }), false);
    check_expect(janelas_soprepoem({ 410, 550, 100, 200 }, { 300, 400, 50, 100 }), false);
}
```

```
// Produz true se o espaço das janelas a e b se soprepõem, false caso contrário.
bool janelas_soprepoem(Janela a, Janela b)
{
    return a.x < (b.x + b.largura) &&
           // borda direta de a vem antes da borda esquerda de b
           b.x < (a.x + a.largura) &&
           // borda direta de b vem antes da borda esquerda de a
           a.y < (b.y + b.altura) &&
           // borda superior de a vem antes da borda inferior de b
           b.y < (a.y + a.altura);
           // borda superior de b vem antes da borda inferior de a
}
```

Exemplo

Em um jogo de loteria os apostadores fazem apostas escolhendo 6 números distintos entre 1 e 60. No sorteio são sorteados 6 números de forma aleatória. Os apostadores que acertam 4, 5 ou 6 números são contemplados com prêmios. Projete uma função que conte quantos números uma determinada aposta acertou.

Projeto desenvolvido em sala.

```c
// Coleção de 6 números distintos entre 1 e 60.
struct SeisNumeros {
    int a;
    int b;
    int c;
    int d;
    int e;
    int f;
};
```

As apostas e os números sorteados serão representados pela estrutura `SeisNumeros`.

```
// Calcula quantos números da aposta estão em sorteados.
int numero_acertos(SeisNumeros aposta, SeisNumeros sorteados)
{
    return 0;
}

examples
{
    check_expect(numero_acertos({1, 2, 3, 4, 5, 6}, {8, 12, 20, 41, 52, 57}), 0);
    check_expect(numero_acertos({8, 2, 3, 4, 5, 6}, {8, 12, 20, 41, 52, 57}), 1);
    check_expect(numero_acertos({8, 12, 3, 4, 5, 6}, {8, 12, 20, 41, 52, 57}), 2);
    check_expect(numero_acertos({8, 12, 20, 4, 5, 6}, {8, 12, 20, 41, 52, 57}), 3);
    check_expect(numero_acertos({8, 12, 20, 41, 5, 6}, {8, 12, 20, 41, 52, 57}), 4);
    check_expect(numero_acertos({8, 12, 20, 41, 52, 6}, {8, 12, 20, 41, 52, 57}), 5);
    check_expect(numero_acertos({8, 12, 20, 41, 52, 57}, {8, 12, 20, 41, 52, 57}), 6);
}
```

Qual o processo que utilizamos para determinar as repostas?

Começamos o número de acertos com zero.

Depois verificamos se o primeiro número está entre os sorteados, se sim, aumentamentos os acertos em 1.

Depois verificamos se o segundo número está entre os sorteados, se sim, aumentamentos os acertos em 1.

E assim com o restante dos números. No final, temos a quantidade de acertos.

Este processo não parece muito detalhado... como verificamos se um número está entre os sorteados? Vamos deixar esse problema para depois.

Como expressar esse processo em C++? Com uma sequência de etapas, como em muitos exercícios anteriores.

Implementação

```
int acertos = 0;
if (sorteado(aposta.a, sorteados)) {
    acertos = acertos + 1;
}
if (sorteado(aposta.b, sorteados)) {
    acertos = acertos + 1;
}
if (sorteado(aposta.c, sorteados)) {
    acertos = acertos + 1;
}
if (sorteado(aposta.d, sorteados)) {
    acertos = acertos + 1;
}
if (sorteado(aposta.e, sorteados)) {
    acertos = acertos + 1;
}
if (sorteado(aposta.f, sorteados)) {
    acertos = acertos + 1;
}
return acertos;
```

Aqui utilizamos o pensamento desejoso e desejamos que a função `sorteado` exista e funcione de acordo com a seguinte especificação:

```
// Produz true se n é um dos números em sorteados,
// false caso contrário.
bool sorteado(int n, SeisNumeros sorteados);
```

Todas as funções que desejamos durante o projeto de outra função são colocas em uma lista de pendência. Após o término da implementação da função em questão, precisamos implementar as funções da lista de pendências.

Agora precisamos terminar o projeto da função `sorteado`.

```
// Produz true se n é um dos números em sorteados, false caso contrário.
bool sorteado(int n, SeisNumeros sorteados)
{
    return false;
}

examples {
    SeisNumeros sorteados = {1, 7, 10, 40, 41, 60};
    check_expect(sorteado(1, sorteados), true);
    check_expect(sorteado(7, sorteados), true);
    check_expect(sorteado(10, sorteados), true);
    check_expect(sorteado(40, sorteados), true);
    check_expect(sorteado(41, sorteados), true);
    check_expect(sorteado(60, sorteados), true);
    check_expect(sorteado(2, sorteados), false);
    check_expect(sorteado(15, sorteados), false);
    check_expect(sorteado(49, sorteados), false);
}
```

Qual o processo que utilizamos para determinar as repostas?

Verificamos se n é igual ao primeiro valor sorteado, se sim, guardamos a resposta **`true`**.

Senão, verificamos se n é igual ao segundo valor sorteado, se sim, guardamos a resposta **`true`**.

Senão, verificamos se n é igual ao terceiro valor... Senão guardamos a resposta **`false`**.

Como expressar esse processo em C++? Com uma sequência de etapas! Aqui vamos fazer uma simplificação para evitar o aninhamento demasiado de ifs. Vamos colocar um if após o outro (você consegue ver que este processo alternativo continua correto?)

```
// Produz true se n é um dos números
// em sorteados, false caso contrário.
bool sorteado(int n, SeisNumeros sorteados)
{
    bool sorteado = false;
    if (n == sorteados.a) {
        sorteado = true;
    }
    if (n == sorteados.b) {
        sorteado = true;
    }
    if (n == sorteados.c) {
        sorteado = true;
    }
    if (n == sorteados.d) {
        sorteado = true;
    }
    if (n == sorteados.e) {
        sorteado = true;
    }
    if (n == sorteados.f) {
        sorteado = true;
    }
    return sorteado;
}
```

Implementação alternativa.

```
// Produz true se n é um dos números
// em sorteados, false caso contrário.
bool sorteado(int n, SeisNumeros sorteados)
{
    return n == sorteados.a ||
        n == sorteados.b ||
        n == sorteados.c ||
        n == sorteados.d ||
        n == sorteados.e ||
        n == sorteados.f;
}
```

Verificação: ok

Revisão: o código parece repetitivo... Vamos ver novas construções!