

C++

André Furlan

1 Introdução

Neste guia iremos ao mesmo tempo construir um sistema e explicar os conceitos por detrás desta construção. A ideia é que ao final deste guia você tenha um sistema completo e funcional, no caso, uma pequena agenda de compromissos.

Esse material foi escrito tendo como alvo dispositivos de leitura móveis como celulares e leitores de ebook, mas você pode imprimir se quiser.

O compartilhamento desse material é livre contanto que seja feita de forma gratuita e com os devidos créditos (sim, eu gosto de ser estrelinha). Não é permitido comercializar de qualquer forma o mesmo! Se quiser fazê-lo entre em contato comigo e eu vejo se quebro o teu galho 😊: ensismoebius@gmail.com.

Ah! Doações são sempre bem vindas também!

Sem mais, Vamos ao trabalho.

2 A função principal

A função principal é onde o nosso programa dá os primeiros passos, na função principal é possível chamar outras funções, ou ainda, executar todas as instruções do programa inteiro, embora esta segunda opção não seja muito recomendada para programas grandes.

```
1  int main(int argc, char **argv)
2  {
3      //corpo da funcao
4  }
```

Código 1: Função principal

Funções são estruturas que tiveram suas origens na matemática, nessa disciplina temos várias funções, por exemplo, a função soma que recebe dois argumentos e retorna o valor correspondente, ou a função subtração que subtrai um argumento do outro e retorna o valor resultante, existem ainda

Em se tratando de funções parâmetros e argumentos são sinônimos.

outras funções matemáticas como, por exemplo, a função cosseno que recebe apenas um parâmetro e retorna o seu valor correspondente, representando isso usando a linguagem C++ teremos o seguinte código:

```
1 float soma(float a, float b){
2     return a + b;
3 }
4 float subtracao(float a, float b){
5     return a - b;
6 }
7 float cosseno(float a){
8     // Faz alguma coisa com a variavel 'a'
9     return resultado;
10 }
11
```

Código 2: Funções matemáticas em C++

Perceba que toda função tem:

- Um tipo de retorno (nos casos acima as funções retornam “float” que são números com casas decimais).
- Um nome (cosseno, soma, subtração).
- Parâmetros (as variáveis do tipo float **a** e **b**).
- E por fim dentro de chaves o código que será executado ao se chamar uma função.

Podemos interpretar a função como uma máquina que recebe várias coisas e, usando essas coisas recebidas, produz uma outra coisa, como por exemplo, uma máquina que recebe água, farinha, chocolate, fermento e nos retorna um bolo.

A função principal faz exatamente isso: Ela recebe os parâmetros **argc** e ****argv** (que serão explicados mais a frente) e retorna um valor inteiro (indicado por **int**). A função principal é especial dentro do C++, como já foi dito é por ela que o programa começa a executar, o papel da função principal é também informar ao sistema operacional se o programa terminou com sucesso ou se houve algum erro dentro do mesmo, para isso **a função principal pode retornar o valor zero, quando tudo deu certo e um valor diferente de zero quando algo deu errado**, esse é o padrão.

Agora finalmente vamos iniciar o nosso programa, abra seu editor de texto favorito, recomendo a IDE Dev C++ ou para os/as mais experientes a IDE eclipse com o plugin CDT ou ainda o Microsoft Visual Studio.

3 O início da agenda

```
1  int main(int argc, char **argv)
2  {
3      int dia = 0;
4      int mes = 0;
5      int ano = 0;
6  }
```

Código 3: Agenda parte 1

Nas linhas 3, 4 e 5 o que acabamos de fazer foi declarar 3 variáveis do tipo inteiro, ou seja, agora **podemos guardar valores inteiros na variável ano, mes, dia!** Elas servirão para guardar a data que será pedida ao usuário/usuária.

Mas o que é uma variável? **Uma variável é um espaço reservado na memória do computador dentro do qual podemos armazenar valores, a esse espaço por questão de praticidade damos um nome.**

Existem vários tipos de variáveis! Inclusive podemos criar os nossos próprios! Mas, para começar, vamos ver alguns dos tipos básicos de variáveis que a linguagem C++ nos fornece:

Caso você não entenda a notação abaixo: $2,2e-308$ é o mesmo que $2,2 \times 10^{-308}$

- **bool**: Contém o valor Verdadeiro (True) ou Falso (False).
- **char**: Permite armazenar um caractere.
- **unsigned short int** : Valores de 0 a 65.535.
- **short int**: Valores de -32.768 a 32.767.
- **unsigned long int**: Valores de 0 a 4.294.967.295.
- **long int**: Valores: de -2.147.483.648 a 2.147.483.647.

- **int (em sistemas de 16 bits)**: Valores de -32.768 a 32.767.
- **int (em sistemas de 32 bits)**: Valores de -2.147.483.648 a 2.147.483.647.
- **unsigned int (16 bits)**: Valores de 0 a 65.535.
- **unsigned int (32 bits)**: Valores de 0 a 4.294.967.295.
- **float**: Valores de $1,2e-38$ a $3,4e-38$.
- **double**: Valores: de $2,2e-308$ a $3,4e308$.

Saiba que esses valores podem variar de acordo com a implementação da linguagem, do sistema operacional ou da máquina que está rodando o programa.

No nosso caso, por enquanto, usaremos apenas variáveis inteiras pois, nesse caso, nos interessa valores inteiros como o mês, dia e o ano. Caso fosse necessário fazer um cálculo que resultasse em valores de ponto flutuante (ou seja com casas depois da vírgula) usaríamos o tipo float ou double dependendo do tamanho do número que se desejasse trabalhar.

4 Interrogando o usuário!

Programas que rodam sem que seja fornecida alguma informação raramente tem alguma utilidade, no nosso caso, precisamos que o usuário informe o dia, o mês e o ano do compromisso que quer armazenar na nossa agenda. Para que o usuário ou usuária consiga realizar tal tarefa nosso programa precisa, de alguma forma, ler os dados que ela ou ela digita. É aí que entra a **biblioteca iostream**!

De forma resumida **uma biblioteca é composta de uma série de códigos feitos por outra pessoa ou organização que tem como fim a realização de uma ou mais tarefas**, em outras palavras, **na maioria das vezes não precisamos reinventar a roda!** Pois, em boa parte das vezes, alguma biblioteca já implementou a funcionalidade que desejamos usar em nosso programa! Nesse caso a biblioteca iostream já implementou as rotinas necessárias para que possamos interrogar usuário ou usuária sobre alguma

informação que nosso programa precisa.

Nosso programa fica assim:

```
1 #include <iostream>
2
3 int main(int argc, char **argv)
4 {
5     int dia;
6     int mes;
7     int ano;
8
9     std::cout << "Informe o dia:" << std::endl;
10    std::cin >> dia;
11
12    std::cout << "Informe o mes:" << std::endl;
13    std::cin >> mes;
14
15    std::cout << "Informe o ano:" << std::endl;
16    std::cin >> ano;
17 }
18
```

Código 4: Agenda parte 2

Linha 1: Usamos o que se chama de “diretiva do compilador” que nada mais é do que uma instrução ao compilador para que o mesmo faça alguma coisa com o nosso código, nesse caso, **o compilador estará copiando e colando tudo o que estiver escrito no arquivo chamado iostream! É isso o que a diretiva include faz!**

Linha 9: O programa exibe uma mensagem pedindo para que o usuário informe o dia do compromisso. Isso é feito utilizando-se o **operador de fluxo** (<<) que direciona o texto para a **saída do programa dentro de um terminal (std::cout)**, ou seja, sempre que quisermos exibir uma mensagem para o usuário no terminal podemos direcionar essa mensagem usando o operador de fluxo para std::cout!

Ainda na linha 9 vemos que ao final da mesma se encontra a expressão std::endl, essa expressão indica que devemos ir para próxima linha no terminal (é como se, dentro de um editor de texto, apertássemos a tecla enter do teclado).

Já na linha 10 vemos o inverso: Agora o operador de fluxo aponta para outra direção! O fluxo agora vai de std::cin para a variável dia! **A expressão**

`std::cin` indica que devemos ler o que o usuário digita! Essa leitura termina no momento em que o usuário ou a usuária aperta a tecla **enter**, o valor digitado é armazenado, nesse caso específico, na variável `dia`. (Quase) Sempre que quisermos perguntar algo ao usuário usaremos `std::cin`.

As outras linhas, tu que é inteligente, já entendeu não é? Vamos em frente então!

5 Armazenando listas

Vetores, arrays e listas **nesse contexto** são tudo a mesma coisa!

Você há de concordar que de nada serve uma agenda se ela tem apenas a data do compromisso! Precisamos de alguma forma armazenar a descrição deste compromisso é aí (e em muitos outros lugares) que entram as listas na linguagem C++. Como já vimos o C++ oferece muitos tipos de variáveis e o único tipo capaz de armazenar texto é o tipo `char`. Porém, como já vimos, `char` apenas armazena um único caractere! Para que seja possível armazenar vários caracteres (ou seja um texto) precisamos criar uma **“lista de chars”**! Listas, em linguagens de programação, geralmente são conhecidas como **“arrays”** e é possível criar listas de todos os tipos disponíveis na linguagem C++, inclusive usando aqueles tipos criados pelo próprio programador ou programadora.

Listas de chars também são conhecidas no mundo da programação como **“strings”** nome esse que geralmente usaremos. Existe um tipo na linguagem C++ específico para armazenar Strings, no entanto, por questões didáticas e de demonstração das listas, usaremos a lista de chars para fazer o que precisamos. Mais adiante usaremos esse tipo específico para codificar nossos programas.

Nosso programa fica assim:

```
1 #include <iostream>
2
3 int main(int argc, char **argv)
4 {
5     int dia;
6     int mes;
7     int ano;
```

```

8 char* descricao = new char[100];
9
10 std::cout << "Informe o dia:" << std::endl;
11 std::cin >> dia;
12
13 std::cout << "Informe o mes:" << std::endl;
14 std::cin >> mes;
15
16 std::cout << "Informe o ano:" << std::endl;
17 std::cin >> ano;
18
19 std::cout << "Descreva o compromisso:" << std::endl;
20 std::cin >> descricao;
21 }
22

```

Código 5: Agenda parte 3

Na linha 8 você pode perceber que o tipo `char` vem acompanhado de um `*`, isso indica que **a variável (no caso a descrição) aponta para uma região na memória onde está o início de uma lista**, em outras palavras, a variável aponta para o início da nossa lista de letras, ou ainda de outra forma, a variável aponta para nossa string! **Em C++ sempre que quisermos fazer uma lista temos que informar que tipo de dados essa lista vai guardar**, no nosso caso será uma lista de `char`, mas poderia ser uma lista de `float`, `double`, `int`, etc!

Ainda na linha 8 perceba que após o símbolo de *igual* aparece a expressão `new char[100]`, isso significa que reservaremos um **novo** espaço na memória do computador capaz de guardar **100 chars!!**

Em C++ a palavra **new** serve para reservar espaços de memória para nossos programas usarem. Quando declaramos **variáveis simples** (ou seja: Que não são ponteiros) **não precisamos de new**.

Grave isso: Toda variável do tipo lista em C++ nada mais é do que um ponteiro apontando para o início da nossa lista! Frase redundante porém útil!

Mais à frente veremos formas mais fáceis de usar listas com a biblioteca ***vector***.

O resto do código tá de boas para entender! Em frente!!

6 Visualizando os valores armazenados

Pois bem! Não entraremos em como executar passo-a-passo o seu programa de forma que você possa depurá-lo pois isso depende do ambiente de programação que você está usando. Veremos agora como visualizar o conteúdo de suas variáveis na própria saída do seu programa, você verá que é muito fácil! Olha só:

```
1 #include <iostream>
2
3 int main(int argc, char **argv)
4 {
5     int dia = 0;
6     int mes = 0;
7     int ano = 0;
8     char* descricao = new char[100];
9
10    std::cout << "Informe o dia:" << std::endl;
11    std::cin >> dia;
12
13    std::cout << "Informe o mes:" << std::endl;
14    std::cin >> mes;
15
16    std::cout << "Informe o ano:" << std::endl;
17    std::cin >> ano;
18
19    std::cout << "Descreva o compromisso:" << std::endl;
20    std::cin >> descricao;
21
22    // Exibe apenas o dia
23    std::cout << dia << std::endl;
24
25    // Exibe o dia, mes e ano separados por "-"
26    std::cout << dia << "-" << mes << "-" << ano << std::endl;
27
28    // Exibe o dia, mes e ano separados por "-" mais a descricao
29    std::cout << dia << "-" << mes << "-" << ano << ": " <<
30    descricao << std::endl;
31 }
```

Código 6: Agenda parte 4

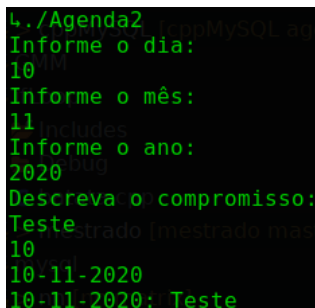
Mais coisas novas! Perceba aqui nas linhas 22, 25 e 28 temos textos comuns antecidos por duas barras (//), a isso damos o nome de comentários!

Comentários são linhas de código que serão ignorados quando o programa for compilado os **comentários servem para orientar e explicar as linhas de códigos digitadas para outros programadores ou programadoras facilitando assim a manutenção do código**. É importante que você comente bem os seus códigos pois este outro programador ou programadora que vai dar manutenção no código pode ser você mesmo no futuro! Outro fator importante, em se tratando de comentários, é que simplesmente você deve ser uma pessoa boa independente se será você ou outra pessoa a se beneficiar com isso no futuro!

Não seja um cuzão ou cuzona! 😊

Já nas linhas 23, 26 e 29 perceba que nosso código é composto de vários operadores de fluxo de forma que todas as strings são concatenadas para formar a saída que desejamos. Simples não? **A figura 1 mostra como o nosso programa está se comportando.**

Use um tempinho comparando os resultados da figura e o código e você entenderá o que está acontecendo.



```
./Agenda2
Informe o dia:
10
Informe o mês:
11
Informe o ano:
2020
Descreva o compromisso:
Teste
10-11-2020
10-11-2020: Teste
```

Figure 1: Comportamento do programa

7 Algo de errado com a descrição não está certo...

Legal! Agora o nosso sistema solicita dados do usuário! No entanto, se você tentou colocar uma descrição com mais de uma palavra percebeu que o sistema apenas considera a primeira palavra! E isso não é legal!

Precisamos que o sistema leia todas as palavras da descrição, e para que isso seja possível em vez de usarmos *cin* usaremos a função **std::getline**. Esta função em vez de considerar apenas a primeira palavra lê todo o conteúdo da linha digitada, veja como ficou o nosso código:

```
1 #include <string>
2 #include <iostream>
3
4 int main(int argc, char **argv)
5 {
6     int dia = 0;
7     int mes = 0;
8     int ano = 0;
9     std::string descricao;
10
11     std::cout << "Informe o dia:" << std::endl;
12     std::cin >> dia;
13
14     std::cout << "Informe o mes:" << std::endl;
15     std::cin >> mes;
16
17     std::cout << "Informe o ano:" << std::endl;
18     std::cin >> ano;
19
20     // Limpa a memoria de qualquer caractere restante
21     // Se isso nao for feito getline nao funciona
22     std::cin.ignore();
23
24     std::cout << "Descreva o compromisso:" << std::endl;
25     std::getline(std::cin, descricao);
26
27     // Exibe apenas o dia
28     std::cout << dia << std::endl;
29
30     // Exibe o dia, mes e ano separados por "-"
31     std::cout << dia << "-" << mes << "-" << ano << std::endl;
32
33     // Exibe o dia, mes e ano separados por "-" mais a descricao
34     std::cout << dia << "-" << mes << "-" << ano << ": " <<
        descricao << std::endl;
35 }
36
```

Código 7: Agenda parte 5

Vamos por partes: A primeira coisa a notar é que na linha 1 incluímos

a biblioteca **string**, ela nos permite manipular nossas strings de uma forma muito mais prática do que se usarmos um *array de char*. Perceba que agora, quando vamos criar a variável descrição basta colocar **std::string descricao** em vez de *char* descricao = new char[100]*: Muito mais fácil de ler e entender!!!

Na linha 22 temos mais uma coisa nova: O comando **std::cin::ignore()** **limpa a parte da memória recebe os dados que o usuário ou usuária digitou e que porventura tenha alguma “lixo” ou informação indesejada**, e pra que usamos essa desgraça aqui? Pois bem: Quando o comando *std::cin >> ano* lê o ano, ele o faz depois do pressionamento da tecla *enter*, no entanto, **a variável não armazena o dígito da tecla enter** deixando esse valor “sobrando” na memória. O comando **std::getline** é sensível a esse lixo na memória, sendo assim, para garantir que esse comando funcione precisamos limpar a memória com *std::cin::ignore()*.

E finalmente na linha 25 usamos o comando *std::getline*: O primeiro argumento dessa função é o próprio fluxo de entrada de dados (*std::cin*) o segundo é a variável, dentro da qual, queremos armazenar o valor.

8 Um ciclo infinito

Pois bem, agora armazenamos todos os nossos dados e conseguimos mostrar eles na tela! No entanto, nosso programa faz isso apenas uma vez e precisamos que ele não feche após fazer isso e sim que continue a rodar para que o mesmo possa nos avisar quando um compromisso vai ocorrer, ou ainda, para que possamos adicionar mais compromissos.

Para conseguirmos isso adicionaremos uma coisa incrível chamada de laço infinito! mais conhecido como “Loop Infinito”.

Adicionaremos agora em nosso programa uma parte que ficará rodando infinitamente, essa parte ficará esperando constantemente por dados do usuário ou usuária, assim poderemos implementar um esquema no qual poderemos enviar comandos para o nosso programa!

O **loop infinito** é um artifício usado por todos os programas que se mantêm rodando no computador! **Todo programa, inclusive jogos, que ficam ativos esperando alguma ação do usuário ou usuária usa esse recurso!**

```
1 #include <string>
2 #include <iostream>
3
```

```

4 int main(int argc, char **argv)
5 {
6     int dia = 0;
7     int mes = 0;
8     int ano = 0;
9     std::string descricao;
10
11
12     while (true)
13     {
14         std::cout << "Informe o dia:" << std::endl;
15         std::cin >> dia;
16
17         std::cout << "Informe o mes:" << std::endl;
18         std::cin >> mes;
19
20         std::cout << "Informe o ano:" << std::endl;
21         std::cin >> ano;
22
23         // Limpa a memoria de qualquer caractere restante
24         // Se isso nao for feito getline nao funciona
25         std::cin.ignore();
26
27         std::cout << "Descreva o compromisso:" << std::endl;
28         std::getline(std::cin, descricao);
29
30         // Exibe apenas o dia
31         std::cout << dia << std::endl;
32
33         // Exibe o dia, mes e ano separados por "-"
34         std::cout << dia << "-" << mes << "-" << ano << std::endl;
35
36         // Exibe o dia, mes e ano separados por "-" mais a
37         descricao
38         std::cout << dia << "-" << mes << "-" << ano << ": " <<
39         descricao << std::endl;
40     }
41 }

```

Código 8: Agenda parte 6

Agora, se você rodar esse programa perceberá que ele fica perguntando as mesmas questões o tempo todo e exibindo os dados digitados. Na linha 12 perceba que temos um laço do tipo "while" esse laço executa alguma coisa

enquanto uma condição for verdadeira, e se você der uma boa olhada perceberá que a condição desse laço é sempre verdadeira! Isso é um Loop Infinito!

9 Uma estrutura no sistema!

Agora sim! O nosso programa continua ativo o tempo todo! Mas a única coisa que ele faz é pedir os dados dos compromissos e nem ao menos os armazena! Como ousa?!

O próximo passo é criar uma estrutura de dados que comporte as informações manipuladas pela nossa agenda, para isso criaremos o que chamamos de “struct”. Uma “struct” é um tipo personalizado de dados! Uma forma de criar um tipo a partir de outros mais primitivos.

```
1 #include <string>
2 #include <iostream>
3
4 int main(int argc, char **argv)
5 {
6     struct itemDeAgenda
7     {
8         int dia = 0;
9         int mes = 0;
10        int ano = 0;
11        std::string descricao;
12    };
13
14    while (true)
15    {
16        struct itemDeAgenda compromisso;
17
18        std::cout << "Informe o dia:" << std::endl;
19        std::cin >> compromisso.dia;
20
21        std::cout << "Informe o mes:" << std::endl;
22        std::cin >> compromisso.mes;
23
24        std::cout << "Informe o ano:" << std::endl;
25        std::cin >> compromisso.ano;
26
27        // Limpa a memoria de qualquer caractere restante
28        // Se isso nao for feito getline nao funciona
29        std::cin.ignore();
```

```

30 std::cout << "Descreva o compromisso:" << std::endl;
31 std::getline(std::cin, compromisso.descricao);
32
33 // Exibe apenas o dia
34 std::cout << compromisso.dia << std::endl;
35
36 // Exibe o dia, mes e ano separados por "-"
37 std::cout << compromisso.dia << "-" << compromisso.mes << "-"
38 << compromisso.ano << std::endl;
39
40 // Exibe o dia, mes e ano separados por "-" mais a descricao
41 std::cout << compromisso.dia << "-" << compromisso.mes << "-"
42 << compromisso.ano << ": " << compromisso.descricao << std::
43 endl;
44 }

```

Código 9: Agenda parte 7

Vejam só! Nas linhas 7, 8, 9, 10, 11, 12 e 13 **o dia, mês, ano e a descrição estão dentro de um bloco de estrutura chamado ItemDeAgenda !** Essa forma de escrever nossos dados faz com que se possa acessar os dados da agenda em bloco (usando a estrutura) ou individualmente (usando a **sintaxe de ponto** como veremos adiante).

Perceba que a forma de acessar o dia, mês, ano e a descrição mudou! Se você olhar para a linha 18 Perceberá que podemos criar uma variável do tipo `itemDeAgenda` ! Um tipo que nós mesmos criamos!

Nas linhas 21, 24, 27 e 34 percebemos que a **forma de acessar os dados individualmente** mudou! Agora precisamos do nome da variável seguida de `."` para depois colocar o nome do que chamaremos a partir de agora **campo ou propriedade** dessa estrutura.

Mas espera um pouquinho! No final você está mais complicado do que facilitando! O código anterior fazia a mesma coisa e era muito mais simples!

Sim, de fato! No entanto, olhando o próximo código você perceberá a flexibilidade que a criação dessa estrutura nos trouxe

pois agora, poderemos usar um novo tipo de dados fornecido pela linguagem C++ chamado **vector**.

10 Uma lista de estruturas

O *vector* propicia a criação de listas (mais conhecidas como arrays) de forma fácil, ele nos permite gerenciar nossas listas de uma forma muito mais prática. O próximo código mostra como integrar uma estrutura ao *vector*:

```
1 #include <string>
2 #include <vector>
3 #include <iostream>
4
5 int main(int argc , char **argv)
6 {
7     struct itemDeAgenda
8     {
9         int dia = 0;
10        int mes = 0;
11        int ano = 0;
12        std::string descricao;
13    };
14
15    // Cria uma lista de itemDeAgenda
16    std::vector<struct itemDeAgenda> listaDeCompromissos;
17
18    while (true)
19    {
20        struct itemDeAgenda compromisso;
21
22        std::cout << "Informe o dia:" << std::endl;
23        std::cin >> compromisso.dia;
24
25        std::cout << "Informe o mes:" << std::endl;
26        std::cin >> compromisso.mes;
27
28        std::cout << "Informe o ano:" << std::endl;
29        std::cin >> compromisso.ano;
30
31        // Limpa a memoria de qualquer caractere restante
```



```

32 // Se isso nao for feito getline nao funciona
33 std::cin.ignore();
34
35 std::cout << "Descreva o compromisso:" << std::endl;
36 std::getline(std::cin, compromisso.descricao);
37
38 // Exibe apenas o dia
39 std::cout << compromisso.dia << std::endl;
40
41 // Exibe o dia, mes e ano separados por "-"
42 std::cout << compromisso.dia << "-" << compromisso.mes << "-"
43 << compromisso.ano << std::endl;
44
45 // Exibe o dia, mes e ano separados por "-" mais a descricao
46 std::cout << compromisso.dia << "-" << compromisso.mes << "-"
47 << compromisso.ano << ": " << compromisso.descricao << std::
48 endl;
49
50 // Adiciona esse item de agenda na listaDeCompromissos
51 listaDeCompromissos.push_back(compromisso);
52
53 }
54 }

```

Código 10: Agenda parte 8

Na linha 1 importamos uma nova biblioteca! A biblioteca **vector**! Com ela podemos fazer listas de qualquer coisa! Já na linha 17 criamos a lista para os nossos itens de agenda. A forma de criar essas listas está exemplificada no código [11](#).

Na linha 50 adicionamos o item da agenda à sua lista usando o comando do próprio tipo `vector`, o *push_back*, que adiciona algo ao final da lista.

Olha que legal! Agora, a cada iteração do Loop, adicionamos um novo item de agenda a nossa lista!

```

1 // Cria uma lista de inteiros
2 std::vector<int> lista1;
3
4 // Cria uma lista de strings
5 std::vector<std::string> lista2;
6
7 // Cria uma lista de floats
8 std::vector<float> lista3;
9
10 // Cria uma lista de cabras
11 std::vector<cabra> lista4;
12
13 // Cria uma lista de mandiocas
14 std::vector<mandioca> lista5;
15

```

Código 11: Exemplos de std::vector

11 Óh... Augusto Carlos de Almeida Costa Barros! Se você me ama, mostre-me sua lista!

De nada adianta armazenarmos tanta informação nas nossas listas se não houver alguma forma de usar essa informação! No nosso caso usar essa informação significa mostrá-la ao usuário ou usuária. Para que tal coisa seja possível teremos que usar um outro tipo de loop, **conhecido como o loop for, ou em português, o laço para.**

Esse tipo de laço **serve, geralmente, para percorrer listas ou para gerar números sequenciais** segundo alguma regra.

Esse laço **é constituído por uma variável interna que tem o seu valor modificado a cada iteração**, geralmente essa variável interna é usada de alguma forma dentro desse laço, mais comumente **como um indicador de posição dentro de uma lista**, por exemplo, se o valor da variável for 0 geralmente

estaremos acessando o primeiro item da lista, se o valor da variável for 1 estaremos acessando o segundo item da lista, se o valor for 2 estaremos acessando o terceiro item da lista e assim por diante.

Olha só o nosso código como ficou:

```
1 #include <string>
2 #include <vector>
3 #include <iostream>
4
5 int main(int argc, char **argv)
6 {
7     struct itemDaAgenda
8     {
9         int dia = 0;
10        int mes = 0;
11        int ano = 0;
12        std::string descricao;
13    };
14
15    std::vector<struct itemDaAgenda> listaDoItensDaAgenda;
16
17    while (true)
18    {
19        struct itemDaAgenda novoItem;
20
21        std::cout << "Informe o dia:" << std::endl;
22        std::cin >> novoItem.dia;
23
24        std::cout << "Informe o mes:" << std::endl;
25        std::cin >> novoItem.mes;
26
27        std::cout << "Informe o ano:" << std::endl;
28        std::cin >> novoItem.ano;
29
30        std::cin.ignore();
31
32        std::cout << "Descreva o compromisso:" << std::endl;
33        std::getline(std::cin, novoItem.descricao);
34
35        listaDoItensDaAgenda.push_back(novoItem);
36
37        for (int i = 0; i < listaDoItensDaAgenda.size(); i++)
38        {
39            // Exibe o dia, mes e ano separados por "-" mais a
```

descricao

```
std::cout << listaDoItensDaAgenda[i].dia << "-" <<
listaDoItensDaAgenda[i].mes << "-" << listaDoItensDaAgenda[i].
ano << ":" << listaDoItensDaAgenda[i].descricao << std::endl;
}
}
}
```

Código 12: Agenda parte 9

Antes de continuar é importante ressaltar que *vector* proporciona uma função interna chamada **size** que retorna a quantidade de itens armazenados em nossa lista.

Temos muita coisa para entender aqui! Começando pela linha 39 é possível ver que o laço **for** é constituído por uma variável que tem um valor inicial, no nosso caso 0, após a declaração desta variável com este valor inicial é necessário definir uma condição para que

nosso laço continue executando, neste caso enquanto a variável for menor do que a quantidade de itens dentro da nossa lista, finalmente é definida a **forma de modificação do valor da variável**, neste caso, “i++” significa que, a cada iteração, adicionaremos 1 ao valor atual de nossa variável.

Como se pode notar agora o comando que exibe os dados digitados pelo usuário ou usuária nesse momento se encontra dentro do nosso laço, ele também foi ligeiramente modificado como se pode notar na linha 42, olha só!

Agora Fazemos uma referência à nossa lista e depois colocamos colchetes, dentro desses colchetes colocamos a variável que con-

i+ + é igual a i=i+1
i- - é igual a i=i-1
i+=2 é igual a i=i+2
i*=2 é igual a i=i*2
i/=2 é igual a i=i/2
etc.

terá a posição que queremos acessar!

Neste caso o valor da variável i na primeira iteração será 0, depois se tornará 1, 2, etc. O valor de i **continuará a aumentar enquanto for menor do que o tamanho da nossa lista!** É importante dizer que **em computação geralmente começamos a contar a partir do número zero!** Mantenha isso sempre mente!

Esta é uma forma clássica de acessar itens dentro de uma lista! Não importando se você está usando *vector* ou um *array* tradicional!

É importante dizer que é possível acessar os itens de uma lista não só colocando uma variável entre os colchetes, mas também colocando números literais dentro dos mesmos.

Mais uns exemplinhos abaixo 🤪:

```
1
2 // Cria uma lista de inteiros
3 std::vector<int> lista;
4
5 // Acessando o terceiro item usando uma variavel
6 batata = 2;
7 std::cout << lista[batata];
8
9 // Acessando o segundo item usando um numero literal
10 std::cout << lista[1];
11
12 // Alterando o quarto item usando uma variavel!
13 k = 3
14 lista[k] = 50;
15
16 // Alterando o primeiro item usando um numero literal!
17 lista[0] = -2;
18
19 // Adicionando 1 a todos os valores!
20 for (int i = 0; i < lista.size(); i++)
21 {
```

```

22 // O mesmo que lista[i] = lista[i] + 1
23 lista[i]++;
24 }
25
26 // Multiplicando todos os valores por 5!
27 for (int i = 0; i < lista.size(); i++)
28 {
29     // O mesmo que lista[i] = lista[i] * 5
30     lista[i] *= 5;
31 }
32
33 // Zerando todos os valores!
34 for (int i = 0; i < lista.size(); i++)
35 {
36     lista[i] = 0;
37 }
38
39 // Fazendo mais coisas... Acho que ja deu pra entender...
40 for (int i = 0; i < lista.size(); i++)
41 {
42     lista[i] = lista[i] / lista[i] - 5;
43 }
44

```

Código 13: Exemplos de `std::vector` com laço `for`

Finalmente agora podemos inserir nossos compromissos e visualizá-los!

12 Brrrrrrr!!! Tá saindo da jaula a função!!

Está dando para ver que nosso código está ficando grande! Então, a partir de agora seguiremos mostrando apenas os trechos que são mais importantes, dessa forma otimizamos o espaço desse material escrito e, ao menos espero, facilitamos o entendimento.

Para que isso seja possível usaremos uma forma de modularizar o nosso código usando funções! **Funções são uma forma de separar o nosso código em pequenos blocos (o mesmo**

que modularizar 😊) de forma que o entendimento do significado da nossa lógica fique mais simples. Também possibilita reutilizar uma mesma funcionalidade sem precisar copiar e colar o mesmo código várias vezes, **isso faz com que seja necessário digitar menos linhas além de facilitar a manutenção do nosso programa** no futuro.

Eu sei que eu prometi que não ia mostrar o código todo novamente, mas prometo essa é a última vez! Preciso fazer isso para que você entenda como foi feita a modularização, sendo assim, vamos separar essa bagaça em funções agora!

```
1 #include <string>
2 #include <vector>
3 #include <iostream>
4
5 struct itemDaAgenda
6 {
7     int dia = 0;
8     int mes = 0;
9     int ano = 0;
10    std::string descricao;
11 };
12
13 std::vector<struct itemDaAgenda> listaDoItensDaAgenda;
14
15 void adicionarItemNaAgenda()
16 {
17     struct itemDaAgenda novoItem;
18
19     std::cout << "Informe o dia:" << std::endl;
20     std::cin >> novoItem.dia;
21
22     std::cout << "Informe o mes:" << std::endl;
23     std::cin >> novoItem.mes;
24
25     std::cout << "Informe o ano:" << std::endl;
26     std::cin >> novoItem.ano;
27
28     std::cin.ignore();
```

```

29
30 std::cout << "Descreva o compromisso:" << std::endl;
31 std::getline(std::cin, novoItem.descricao);
32
33 listaDoItensDaAgenda.push_back(novoItem);
34 }
35
36 void mostrarItensDaAgenda()
37 {
38     // i e a variavel que indica a posicao que queremos
39     // acessar dentro de nossa lista
40     for (int i = 0; i < listaDoItensDaAgenda.size(); i++)
41     {
42         // Exibe o dia, mes e ano separados por "-" mais a descricao
43         std::cout << listaDoItensDaAgenda[i].dia << "-" <<
44         listaDoItensDaAgenda[i].mes << "-" << listaDoItensDaAgenda[i].
45         ano << ": " << listaDoItensDaAgenda[i].descricao << std::endl;
46     }
47 }
48
49 int main(int argc, char **argv)
50 {
51     while (true)
52     {
53         adicionarItemNaAgenda();
54         mostrarItensDaAgenda();
55     }
56 }

```

Código 14: Agenda parte 10

Eita... Agora o negócio ficou tenso... Houve uma mudança radical no código!

Sim meu caro e minha cara estamos alcançando maiores vôos! A partir de agora nossos programas ficarão cada vez mais complexos e, conseqüentemente, vão requerer uma organização igualmente mais complexa!

Vamos por partes: Começemos pelas linhas 52 e 53 que se

localizam dentro do laço infinito: Com certeza você notou que agora temos apenas dois comandos: *adicionarItemNaAgenda()* e *mostrarItensDaAgenda()*. Esses dois comandos **chamam as funções de adição de itens na agenda e de exibição desses itens respectivamente**.

Ficou muito mais fácil de entender não é? O que complicou mesmo mesmo foi o código que está acima da função principal, agora a declaração da estrutura e da lista que usamos para armazenar nossos compromissos estão fora da função principal e também surgiram novas estruturas com uma palavra estranha chamada “void” que ainda não foi explicada, então sem mais delongas vamos a elas!

Na linha 15 declaramos a função de adição de itens na nossa agenda, **a palavra *void* significa que esta função não retornará valor algum (*void* em inglês significa “vazio”)**. Na linha 36 declaramos a função que mostrará todos os nossos compromissos, novamente, essa função não retornará valor algum. Perceba que ambas funções não tem parâmetros!

12.1 Além das funções o que mais está saindo da jaula?

Muito bem, no código anterior todas as estruturas e variáveis eram declaradas dentro da função principal, se você olhar agora, como já foi dito anteriormente, essas estruturas estão declaradas fora da função principal! **Quando uma variável ou estrutura é declarada dentro de uma função dizemos que seu escopo é local àquela função, quando ela é declarada fora das funções dizemos que seu escopo é global!**

Vixe... Não entendi nada!

Veja bem: **Escopo é o nome chique que damos à localização que alguma coisa pertence**, por exemplo, no código anterior a estrutura era válida e usável apenas dentro da função principal, se tentássemos, de alguma forma, usar aquela estrutura fora da função principal teríamos um erro de compilação, **portanto dizemos que o escopo da estrutura e da lista era a função principal**. Agora na versão atual do código o escopo da estrutura e da lista de itens de agenda é global! Em outras palavras quando o escopo de uma estrutura, função ou variável é global estas são acessáveis por qualquer outra parte do código dentro ou fora da função principal! No nosso caso intencionalmente declaramos a estrutura e a lista de itens da agenda de forma global para que, tanto a função de adição de itens como a função de leitura dos itens pudessem acessar e manipular seus valores.

Tirando o escopo global, um escopo é definido sempre que declaramos alguma coisa dentro de um bloco de código, lembrando que os blocos de código são determinados pela abertura e fechamento de chaves.

Mais exemplos para vossa senhoria:

```
1
2 #include <iostream>
3
4 // Aqui declaramos coisas que queremos
5 // que sejam acessíveis por todo mundo.
6 // Esse eh o escopo global
7 float a = 15;
8
9 // A funcao tambem tem variaveis, mas o
10 // escopo dessas variaveis eh o bloco da
11 // funcao, ou seja, um escopo local!
12 float funcaoDeTeste(float b, float c)
13 {
14 // "b" e "c" so existem aqui dentro!!!
```

```

15 // Eh possivel usar "a" aqui pois seu
16 // escopo eh mais amplo (global)
17 return a + b + c;
18 }
19
20 int main(int argc , char **argv)
21 {
22     std::cout << funcaoDeTeste(10, 20) << std::endl;;
23
24     // Nao e possivel fazer std::cout << b << std::endl
25     // ou std::cout << c << std::endl pois b e c apenas
26     // existem dentro do bloco da funcao
27 }
28

```

Código 15: Exemplo de escopo

13 A vida é sobre a ilusão de que podemos escolher

Legal! Agora modularizamos o nosso programa separando-o em funções! *Maaaaaasssss* nossa agenda sempre faz mesma coisa: Cadastra o compromisso e mostrar uma lista deles.

O que queremos na verdade é que nosso programa tenha um **menu** a partir do qual a usuária ou usuário possa escolher a funcionalidade que deseja executar. Então, sem mais delongas, vamos criar este menu!

A criação do menu se dá verificando um número ou letra que o usuário ou usuária digita e direcionando o fluxo do programa de acordo com o desejo do mesmo ou da mesma. Podemos fazer isso com um simples desvio condicional, mais conhecido como “if” e, para nos acostumarmos, criaremos uma função que exibirá este menu:

```

1 int exibeMenu()
2 {

```

```

3 std::cout << "Escolha o que deseja fazer:" << std::endl;
4 std::cout << "1 - Inserir um compromisso" << std::endl;
5 std::cout << "2 - Exibir compromissos" << std::endl;
6
7 int escolha = 0;
8
9 std::cin >> escolha;
10
11 return escolha;
12 }
13

```

Código 16: Menu da agenda

Como você pôde perceber esta é uma função simples: Ela exibe uma série de mensagens de forma que fiquem evidentes as escolhas que devem ser feitas e solicita que um dos valores seja escolhido, depois disso a mesma retorna o valor escolhido.

Em seguida, dentro da função principal, usaremos dois desvios condicionais, um para cada opção que pode ser escolhida:

```

1 int main(int argc, char **argv)
2 {
3     while (true)
4     {
5         int escolha = exibeMenu();
6
7         if (escolha == 1)
8         {
9             adicionarItemNaAgenda();
10        }
11        if (escolha == 2)
12        {
13            mostrarItensDaAgenda();
14        }
15    }
16 }
17

```

Código 17: Menu com desvios condicionais

Olha só! Na linha 6 criamos uma variável que vai guardar a escolha do usuário ou da usuária, na linha 8 criamos um desvio

condicional caso a escolha seja igual a 1, na linha 12 criamos outro desvio condicional caso a escolha seja igual a 2. Como é de se esperar caso a escolha seja igual a 1 adicionaremos um item na agenda, caso a escolha seja igual a 2 mostraremos os compromissos.

14 Na vida também precisamos de um pouco de estabilidade

Olhando o código anterior novamente surge um problema: Enquanto o programa está pequeno é fácil saber o que significa 1 e o que significa 2. Basta olhar no código não é? **Masssssss** quando o programa ficar maior e a quantidade de opções cresce começa a ficar confuso e muitas vezes impraticável controlar o que significa 1, 2, 3, etc. Seria muito legal se pudéssemos **dar um nome a esses valores**, E é exatamente isso que faremos ao criar o que chamamos de **constantes**!

A vantagem das constantes é que, geralmente, não precisamos nos preocupar com o valor delas! As constantes melhoram a legibilidade do código pois facilitam a interpretação do significado de cada linha do nosso programa. No exemplo seguinte criaremos duas constantes de escopo global que vão substituir os números 1 e 2 tanto dentro da função do menu quanto nos desvios condicionais:

```
1 const int INSERIR_COMPROMISSO = 1;
2 const int EXIBIR_COMPROMISSOS = 2;
3
4 int exibeMenu()
5 {
6     std::cout << "Escolha o que deseja fazer:" << std::endl;
7     std::cout << INSERIR_COMPROMISSO << " - Inserir um compromisso"
8     << std::endl;
9     std::cout << EXIBIR_COMPROMISSOS << " - Exibir compromissos" <<
10    std::endl;
```

```

9
10  int escolha = 0;
11
12  std::cin >> escolha;
13
14  return escolha;
15 }
16
17 int main(int argc, char **argv)
18 {
19     while (true)
20     {
21         int escolha = exibeMenu();
22
23         if (escolha == INSERIR_COMPROMISSO)
24         {
25             adicionarItemNaAgenda();
26         }
27         if (escolha == EXIBIR_COMPROMISSOS)
28         {
29             mostrarItensDaAgenda();
30         }
31     }
32 }
33

```

Código 18: Constantes

De cara podemos perceber que a leitura do código melhorou muito! Agora, em vez de compararmos o valor da escolha com o número 1 ou 2, comparamos com as constantes *INSERIR_COMPROMISSO* e com a *EXIBIR_COMPROMISSOS*, muito mais fácil de ler!

Perceba que as **constantes foram declaradas dentro do escopo global pois precisamos usa-las tanto dentro da função de menu quanto na função principal**. Se você comparar o código anterior com o atual vai perceber que os números 1 e 2 foram substituídos pelas suas respectivas constantes!

O padrão adotado para nomenclatura das constantes dentro

deste material será sempre o de palavras com letras maiúsculas separadas pelo caracter underline.

15 Múltiplas escolhas requerem melhores formas de escolher

Tá bonito! Tá formoso! **Maaaassssssssssss** imagine que nosso sistema tenha 10 opções, não preciso dizer que o código ficaria um pouco extenso demais e por vezes, até confuso. Geralmente **quando temos múltiplas comparações dentro de um programa em vez de usarmos seguidos desvios condicionais do jeito tradicional usamos outra estrutura que condensa todas as verificações de uma forma muito mais resumida.**

Essa estrutura é o **switch**!

```
1 int main(int argc , char **argv)
2 {
3     while (true)
4     {
5         int escolha = exibeMenu();
6
7         switch (escolha)
8         {
9             case INSERIR_COMPROMISSO:
10                 adicionarItemNaAgenda();
11                 break;
12             case EXIBIR_COMPROMISSOS:
13                 mostrarItensDaAgenda();
14                 break;
15             default:
16                 std::cout << "Opcao invalida" << std::endl;
17                 break;
18         }
19     }
20 }
21
```

Código 19: Substituindo vários ifs com switch

O switch é uma forma de substituir vários desvios condicionais, vemos na linha 7 que ele começa, como se fosse uma função, recebendo um valor representado, nesse caso, pela variável “escolha”, em seguida, ele compara esse valor recebido com os valores listados em cada linha contendo a palavra “case” (linhas 9 e 12), caso o valor coincida o switch executa o código que estiver depois dos dois pontos (linhas 10 e 13).

A palavra “**break**” significa, nesse contexto, “pare”, ou seja **encerra a execução do switch**, caso contrário, após a primeira coincidência o switch executaria todos os comandos contidos nos “cases” abaixo. Experimente fazer os “cases” sem os “breaks”!

Na linha 15 a palavra “default” significa “padrão”. Essa **instrução opcional** é executada toda vez que o valor recebido não coincidir com algum dos “cases”.

Podemos ler este switch como:

- **Caso** o valor for `INSERIR_COMPROMISSO` **faça** `adicionarItemNaAgenda()`
- **Caso** o valor for `EXIBIR_COMPROMISSOS` **faça** `mostrarItensDaAgenda()`
- **Caso** o valor for algum outro mostre “opção inválida”.

16 Onde estamos?

Até agora o código da agenda está assim:

```
1 #include <string>
2 #include <vector>
```



```

3 #include <iostream>
4
5 const int INSERIR_COMPROMISSO = 1;
6 const int EXIBIR_COMPROMISSOS = 2;
7
8 struct itemDaAgenda
9 {
10     int dia = 0;
11     int mes = 0;
12     int ano = 0;
13     std::string descricao;
14 };
15
16 std::vector<struct itemDaAgenda> listaDoItensDaAgenda;
17
18 void adicionarItemNaAgenda()
19 {
20     struct itemDaAgenda novoItem;
21
22     std::cout << "Informe o dia:" << std::endl;
23     std::cin >> novoItem.dia;
24
25     std::cout << "Informe o mes:" << std::endl;
26     std::cin >> novoItem.mes;
27
28     std::cout << "Informe o ano:" << std::endl;
29     std::cin >> novoItem.ano;
30
31     std::cin.ignore();
32
33     std::cout << "Descreva o compromisso:" << std::endl;
34     std::getline(std::cin, novoItem.descricao);
35
36     listaDoItensDaAgenda.push_back(novoItem);
37 }
38
39 void mostrarItensDaAgenda()
40 {
41     // i eh a variavel que indica a posicao que queremos
42     // acessar dentro de nossa lista
43     for (int i = 0; i < listaDoItensDaAgenda.size(); i++)
44     {
45         // Exibe o dia, mes e ano separados por "-" mais a descricao
46         std::cout << listaDoItensDaAgenda[i].dia << "-" <<
47         listaDoItensDaAgenda[i].mes << "-" << listaDoItensDaAgenda[i].

```

```

47     ano << ":" << listaDoItensDaAgenda[i].descricao << std::endl;
48 }
49
50 int exibeMenu()
51 {
52     std::cout << "Escolha o que deseja fazer:" << std::endl;
53     std::cout << INSERIR_COMPROMISSO << " – Inserir um compromisso"
54     << std::endl;
55     std::cout << EXIBIR_COMPROMISSOS << " – Exibir compromissos" <<
56     std::endl;
57
58     int escolha = 0;
59
60     std::cin >> escolha;
61
62     return escolha;
63 }
64
65 int main(int argc, char **argv)
66 {
67     while (true)
68     {
69         int escolha = exibeMenu();
70
71         switch (escolha)
72         {
73             case INSERIR_COMPROMISSO:
74                 adicionarItemNaAgenda();
75                 break;
76             case EXIBIR_COMPROMISSOS:
77                 mostrarItensDaAgenda();
78                 break;
79             default:
80                 std::cout << "Opcao invalida" << std::endl;
81                 break;
82         }
83     }
84 }

```

Código 20: Agenda parte 11

17 Dedo duro!

Muito bem, como você pode notar nosso código está ficando bem grande! Chegou a hora de separar-lo em arquivos! A esses arquivos daremos os nomes de módulos ou bibliotecas. Mas, temos um problema:

As nossas **funções estão acopladas com as variáveis** que declaramos, isso significa que, não podemos coloca-las em outro arquivo pois **o funcionamento delas depende da existência das variáveis que declaramos no arquivo principal**. E isso não é bom! Como regra geral **devemos evitar o máximo de acoplamento no nosso código** para que o mesmo seja flexível o suficiente para aceitar mudanças durante o desenvolvimento.

Mas como resolvemos isso? Se você olhar bem vai perceber que nenhuma das nossas funções tem parâmetros pois as mesmas usam diretamente as variáveis do programa! E é exatamente isso que gera o acoplamento! Para resolver isso as funções que lêem e manipulam a nossa lista de itens vão receber o **endereço de memória** dessa lista, a esse endereço **damos o nome de ponteiro!**

A passagem de parâmetros para uma função se dá por valor ou por referência, Quando a passagem é feita por valor o computador faz uma cópia do mesmo e o usa dentro da função, caso esse valor seja modificado essa mudança não será refletida na variável original. Quando a passagem é feita por referência qualquer modificação altera a variável original, isso acontece por quê, no caso da passagem por referência parâmetro passa a apontar para a mesma região da memória que a variável original, consequentemente, alterando-se o valor do parâmetro altera se o valor da

variável original.

```
1 #include <iostream>
2
3 // Passagem de parametros por valor
4 void funcaoComPassagemPorValor(int parametro)
5 {
6     // Nesse caso a variavel "parametro" eh uma COPIA
7     // da variavel original alterar ela NAO altera a
8     // variavel original
9     parametro = parametro + 5;
10 }
11
12 // Passagem de parametros por referencia
13 void funcaoComPassagemPorReferencia(int &parametro)
14 {
15     // Nesse caso a variavel "parametro" eh uma REFERENCIA
16     // para a variavel original alterar ela altera SIM a
17     // variavel original
18     parametro = parametro + 5;
19 }
20
21 int main(int argc, char **argv)
22 {
23     int minhaVariavel = 1;
24
25     // Essa funcao NAO altera "minhaVariavel"
26     funcaoComPassagemPorValor(variavel);
27     // Aqui vai exibir o valor "1"
28     std::cout << variavel << std::endl;
29
30     // Essa funcao altera SIM "minhaVariavel"
31     funcaoComPassagemPorReferencia(variavel);
32     // Aqui vai exibir o valor "6"
33     std::cout << variavel << std::endl;
34 }
35
```

Código 21: Exemplo de passagem por referência e valor

Na linha 13 a **assinatura da função** tem um parâmetro antecedido por “&”, esse símbolo indica que essa **variável vai fazer uma referência à variavel original**: Alterando ela, altera o valor original.

Na linha 4 a **assinatura da função** tem um parâmetro **sem o símbolo “&”**, nesse caso, alterando ela, **não altera** o valor original.

Sabendo disso agora podemos criar nossas queridas e estimadas bibliotecas! Para fazê-lo começaremos criando o arquivo com extensão **.cpp**, que conterá a implementação das funções e outro com extensão **.h** que conterá as assinaturas das funções.

Veja só como ficaram os arquivos do nosso programa; esse é o arquivo “agenda.h” que tem as assinaturas das funções:

```
1 #ifndef AGENDA_H
2 #define AGENDA_H
3
4 #include <string>
5 #include <vector>
6
7 const int INSERIR_COMPROMISSO = 1;
8 const int EXIBIR_COMPROMISSOS = 2;
9
10 struct itemDaAgenda
11 {
12     int dia = 0;
13     int mes = 0;
14     int ano = 0;
15     std::string descricao;
16 };
17
18 int exibeMenu();
19
20 void mostrarItensDaAgenda(std::vector<struct itemDaAgenda> &lista
    );
21
22 void adicionarItemNaAgenda(std::vector<struct itemDaAgenda> &
    lista);
23
24 #endif
25
```

Código 22: Arquivo agenda.h da biblioteca

Temos muitas coisas novas aqui! Pra começar na linha 1 temos

uma diretiva de compilação (`#ifndef`), que nada mais é do que uma instrução para o compilador fazer alguma coisa, essa diretiva verifica se o valor `AGENDA_H` **não** foi definido, seguido da **linha 2 que será executada** caso `AGENDA_H` **não tenha sido definido**. Isso é usado para garantir que os símbolos, estruturas, constantes e assinaturas de funções sejam definidas apenas uma vez, pois, uma vez que `AGENDA_H` foi definido o compilador não mais entrará neste “if”.

#ifndef vem do inglês “if not defined” que significa “se não estiver definido”.

#define é “defina”.

#endif é o mesmo que “fim do se”.

Em seguida, nas linhas 4 e 5, são incluídas as bibliotecas “string” e “vector” já explicadas anteriormente.

Se pode ver que na linha 10 e 16 estamos definindo a estrutura que vamos usar, na linha 18, 20 e 22 estão definidas as assinaturas das funções. Lembrando que o “.h” **deve conter apenas assinatura das funções!**

Finalmente na linha 24 fechamos a definição de nossa biblioteca com o “endif” do compilador.

Esse é o arquivo “agenda.cpp” que contém as implementações das funções:

```
1 #include "agenda.h"
2
3 #include <string>
4 #include <vector>
5 #include <iostream>
6
7 int exibeMenu()
8 {
9     std::cout << "Escolha o que deseja fazer:" << std::endl;
10    std::cout << INSERIR_COMPROMISSO << " - Inserir um compromisso"
        << std::endl;
11    std::cout << EXIBIR_COMPROMISSOS << " - Exibir compromissos" <<
        std::endl;
12
```

```

13  int escolha = 0;
14
15  std::cin >> escolha;
16
17  return escolha;
18 }
19
20 void adicionarItemNaAgenda(std::vector<struct itemDaAgenda> &
    lista)
21 {
22     struct itemDaAgenda novoItem;
23
24     std::cout << "Informe o dia:" << std::endl;
25     std::cin >> novoItem.dia;
26
27     std::cout << "Informe o mes:" << std::endl;
28     std::cin >> novoItem.mes;
29
30     std::cout << "Informe o ano:" << std::endl;
31     std::cin >> novoItem.ano;
32
33     std::cin.ignore();
34
35     std::cout << "Descreva o compromisso:" << std::endl;
36     std::getline(std::cin, novoItem.descricao);
37
38     lista.push_back(novoItem);
39 }
40
41 void mostrarItensDaAgenda(std::vector<struct itemDaAgenda> &lista
    )
42 {
43     // i eh a variavel que indica a posicao que queremos
44     // acessar dentro de nossa lista
45     for (int i = 0; i < lista.size(); i++)
46     {
47         // Exibe o dia, mes e ano separados por "-" mais a descricao
48         std::cout << lista[i].dia << "-" << lista[i].mes << "-" <<
            lista[i].ano << ": " << lista[i].descricao << std::endl;
49     }
50 }
51

```

Código 23: Arquivo agenda.cpp da biblioteca

Na linha 1 um do arquivo “agenda.cpp” incluímos o arquivo

“agenda.h” pois ele tem todas as definições de variáveis e estruturas que iremos usar nas nossas funções, em seguida implementamos todas as funções cujas assinaturas foram colocadas no arquivo “agenda.h”. Pronto! É só isso!

E é assim que ficou o arquivo principal:

```
1 #include "agenda.h"
2
3 #include <iostream>
4
5 std::vector<struct itemDaAgenda> listaDoItensDaAgenda;
6
7 int main(int argc, char **argv)
8 {
9     while (true)
10    {
11        int escolha = exibeMenu();
12
13        switch (escolha)
14        {
15            case INSERIR_COMPROMISSO:
16                adicionarItemNaAgenda(listaDoItensDaAgenda);
17                break;
18            case EXIBIR_COMPROMISSOS:
19                mostrarItensDaAgenda(listaDoItensDaAgenda);
20                break;
21            default:
22                std::cout << "Opcao invalida" << std::endl;
23                break;
24        }
25    }
26 }
27
```

Código 24: Arquivo principal

Perceba que, embora a complexidade geral do nosso sistema tem aumentado com a adição de novos arquivos, o nosso arquivo principal ficou muito mais simples de entender!

18 Tudo junto e misturado!

Beleza! Agora que nosso código ficou mais simples e mais fácil de ler, ainda existe, o que podemos chamar de um certo problema: A função “exibeMenu” retorna um valor inteiro e **não é** evidente o que esse inteiro significa!

É interessante deixar explícito o significado dos retornos dessa função! Isso facilita a leitura do código e sua manutenção. Esses retornos são representados pelas constantes que criamos, mas apenas batendo o olho na função não é possível dizer que ela retorna tais constantes! Para resolver esse problema devemos agrupar tais constantes, para fazê-lo vamos usar as enumerações ou, como chamamos mais comumente, os “enums”.

Enumerações ou “enums” são agrupamentos de constantes.

Como agora boa parte do nosso código está dentro de bibliotecas podemos mexer no mesmo de forma mais modularizada, no entanto, esta modificação em particular alterará tanto as bibliotecas quanto o arquivo principal.

Começaremos por agrupar as nossas constantes em uma enumeração. Então, no arquivo “agenda.h”, onde tinha isto:

```
1 const int INSERIR_COMPROMISSO = 1;  
2 const int EXIBIR_COMPROMISSOS = 2;  
3
```

Código 25: Constantes

Temos isto:

```
1 enum OPERACOES  
2 {  
3     INSERIR_COMPROMISSO, EXIBIR_COMPROMISSOS  
4 };
```

Código 26: Enumeração

Uma das **diferenças entre enumerações e constantes** é que **enumerações podem ter seus valores atribuídos automaticamente começando do zero**, nesse caso, `INSERIR_COMPROMISSO`, `EXIBIR_COMPROMISSOS` valem 0 e 1 respectivamente. Também é possível atribuir valores de forma explícita caso seja necessário:

```
1 enum EXEMPLO
2 {
3     CONSTANTE.COM.VALOR1 = 50, CONSTANTE.COM.VALOR2 = 9
4 };
5
```

Código 27: Enumeração com valores explícitos

No nosso caso a numeração automática bastará pois apenas nos interessa saber que tipo de operação estamos realizando e não, necessariamente, os valores das constantes.

Agora estamos prontos! Definir uma enumeração é como definir um novo tipo de variável! E assim como qualquer outro tipo podemos dizer que uma função recebe ou retorna esse tipo!

E agora que sabemos isso a função “`exibeMenu`” fica da seguinte forma:

```
1 OPERACOES  exhibeMenu();
2
```

Código 28: Protótipo da função no arquivo agenda.h

```
1 OPERACOES  exhibeMenu()
2 {
3     std::cout << "Escolha o que deseja fazer:" << std::endl;
4     std::cout << INERIR_COMPROMISSO << " – Inserir um compromisso"
5     << std::endl;
6     std::cout << EXIBIR_COMPROMISSOS << " – Exibir compromissos" <<
7     std::endl;
8 }
```

```

6
7  int  escolha = 0;
8
9  std::cin >> escolha;
10
11 return (OPERACOES) escolha;
12 }
13

```

Código 29: Implementação da função no arquivo agenda.cpp

Na hora de fazer o *casting* os valores precisam ser compatíveis: Não adianta (pelo menos em C++) tentar converter `std::string` para `float`, `double` ou `int`. Para fazer isso usamos funções disponíveis em bibliotecas da linguagem.

De saída já dá para ver que o retorno da função “`exibeMenu`” é do tipo `OPERACOES`, a partir da linha 3 até a 9 temos basicamente a mesma lógica que já vimos, mas na linha 11 existe uma coisa especial: A variável “`escolha`” recebe o que chamamos de **casting**. O **casting** é uma conversão entre tipos, nesse caso, estamos convertendo um valor do tipo inteiro para o valor do tipo `OPERACOES`!

Olhe alguns exemplos de casting:

```

1 float variavelFloat1 = 0.980000019;
2 float variavelFloat2 = 1.980000019;
3
4 double variavelDouble1 = 1.1234567891011121;
5 double variavelDouble2 = 0.1234567891011121;
6
7 int variavelInteira1 = 10;
8 int variavelInteira2 = 10;
9
10 std::cout << "Convertendo float para int:" << std::endl;
11 std::cout << (int) variavelFloat1 << std::endl;
12 std::cout << (int) variavelFloat2 << std::endl;
13
14 std::cout << "Convertendo double para int:" << std::endl;
15 std::cout << (int) variavelDouble1 << std::endl;

```

```

16 std::cout << (int) variavelDouble2 << std::endl;
17
18 std::cout << "Convertendo int para float:" << std::endl;
19 std::cout << (float) variavelInteira1 << std::endl;
20 std::cout << (float) variavelInteira2 << std::endl;
21
22 std::cout << "Convertendo int para double:" << std::endl;
23 std::cout << (double) variavelInteira1 << std::endl;
24 std::cout << (double) variavelInteira2 << std::endl;
25

```

Código 30: Alguns exemplos de casting

Apesar de complicar um pouquinho mais a nossa função agora podemos ter certeza que nosso menu retornará apenas valores contidos dentro da enumeração definida em OPERACOES, isso melhora a leitura de nossa lógica e também garante uma consistência maior em nossa programação.

Nosso arquivo principal fica assim:

```

1 #include <iostream>
2
3 #include "agenda.h"
4
5 std::vector<struct itemDaAgenda> listaDoItensDaAgenda;
6
7 int main(int argc, char **argv)
8 {
9     while (true)
10     {
11         OPERACOES escolha = exibeMenu();
12
13         switch (escolha)
14         {
15             case INSERIR_COMPROMISSO:
16                 adicionarItemNaAgenda(listaDoItensDaAgenda);
17                 break;
18             case EXIBIR_COMPROMISSOS:
19                 mostrarItensDaAgenda(listaDoItensDaAgenda);
20                 break;
21             default:
22                 std::cout << "Opcao invalida" << std::endl;
23                 break;
24         }
25     }
26 }

```

```
25 }  
26 }  
27
```

Código 31: Arquivo principal com as enumerações

Dá uma olhada na linha 11: A variável “escolha” é do tipo OPERACOES, ou seja, ela apenas admite como valores as constantes INSERIR_COMPROMISSO ou EXIBIR_COMPROMISSOS! Agora sabemos exatamente o que a função “exibeMenu” retorna!

19 Descobrimos a quarta dimensão: O tempo!

Finalmente podemos armazenar e exibir no nossos compromissos! Mas o que queremos mesmo, pelo menos nesse caso, é exibir os itens da agenda que pertencem ao dia atual, portanto, precisamos saber qual a data em que estamos e, baseando-se nessa informação, encontrar os compromissos que precisamos cumprir!

Essa parte vai ser um pouco complicadinha pois precisaremos manipular algumas estruturas que já vem prontas na linguagem, por isso, criaremos uma nova biblioteca que ajudará na modularização do nosso programa e deixará o código principal mais simples e fácil de dar manutenção.

Vamos criar os arquivos “tempo.h” e “tempo.cpp” dentro dos quais estarão os códigos que nos retornarão o ano, mês, dia, horas, minutos e por último os segundos caso necessário:

```
1 #ifndef TEMPO_H  
2 #define TEMPO_H  
3  
4 #include <ctime>  
5  
6 struct tm* recuperarDataAtual();  
7  
8 #endif
```

Código 32: Arquivo tempo.h

Iniciamos com a inclusão da biblioteca `ctime` na linha 2, essa é uma das bibliotecas que podem ser usadas para manipular tempo dentro do C++, é importante deixar claro que existe mais de uma forma de fazer o que queremos fazer, o jeito apresentado aqui pode não ser igual a outras formas que você venha ver ou já tenha visto.

Seguimos criando o protótipo da função “recuperarDataAtual”, ela retornará uma estrutura que já está definida dentro da biblioteca “`struct_tm.h`” que é usada por “`time.h`” que, por sua vez, é usada pela biblioteca “`ctime`”, essa estrutura separa os componentes da data e da hora de forma que fique fácil acessar cada um deles, abaixo você pode ver uma cópia simplificada e parcialmente traduzida de “`struct_tm.h`”:

Lembre-se que em programação costumamos contar a partir do zero, então janeiro será o mês 0, fevereiro 1, março 2 e assim em diante.

```

1 #ifndef __struct_tm_defined
2 #define __struct_tm_defined 1
3
4 #include <bits/types.h>
5
6 struct tm
7 {
8     int tm_sec;      /* Segundos.  [0-60] */
9     int tm_min;      /* Minutos.  [0-59] */
10    int tm_hour;      /* Horas.    [0-23] */
11    int tm_mday;      /* Dia.      [1-31] */
12    int tm_mon;       /* Mes.      [0-11] */
13    int tm_year;      /* Ano subtraído de 1900. */
14    int tm_wday;      /* Dia da semana. [0-6] */
15    int tm_yday;      /* Dia do ano.   [0-365] */
16    int tm_isdst;     /* Horario de verao.[
17                        0=ativado
18                        1=desativado

```

```

19         -1=indefinido
20     ]
21     */
22 };
23
24 #endif
25

```

Código 33: Estrutura tm

Agora vamos ver como é a implementação da função “recuperarDataAtual” dentro do arquivo “tempo.cpp”:

```

1 #include "tempo.h"
2
3 struct tm* recuperarDataAtual()
4 {
5     // Variavel usada para armazenar a
6     // quantidade de segundos para
7     // representar uma data
8     time_t tt;
9
10    // Armazena a quantidade segundos que
11    // se passaram desde as 0 horas de 1
12    // de janeiro de 1970 no horario de
13    // Greenwich
14    time(&tt);
15
16    // Transforma esses segundos em uma
17    // estrutura que separa o dia, mes,
18    // ano, para a data e hora local e
19    // a retorna.
20    return localtime(&tt);
21 }
22

```

Código 34: Arquivo tempo.cpp

Agora vai ser fácil recuperar a data hora minuto, etc, basta incluirmos “tempo.h” e chamarmos a função “recuperarDataAtual”! Veja o exemplo:

```

1 auto data = recuperarDataAtual();
2
3 std::cout << data->tm_year + 1900 << std::endl;
4 std::cout << data->tm_mon << std::endl;

```

```
5 std::cout << data->tm_mday << std::endl;  
6 std::cout << data->tm_hour << std::endl;  
7 std::cout << data->tm_min << std::endl;  
8
```

Código 35: Exemplo de uso da função recuperarDataAtual

Uma observação em relação ao código 35: Quando usamos a palavra **auto** em C++ significa que deixamos para o compilador a tarefa de descobrir de qual tipo é a variável que foi declarada, **use isso pouco**, pois o uso excessivo pode diminuir a legibilidade e interpretação do código.

20 Repetição é a chave para a solução

Existem muitas formas de se localizar informações dentro de uma lista que vão desde algoritmos de ordenação (bubblesort, merge sort, insertion sort, etc .) até formas super eficientes de encontrar alguma informação dentro de uma lista não ordenada. Mas, devido a natureza introdutória deste material, faremos o que intuitivamente uma pessoa comum faria: Olhar item-a-item da lista para ver se um ou mais deles pertencem a data atual.

Como vimos na seção anterior a primeira coisa a se fazer é descobrir a data atual, feito isso, precisamos comparar o ano, o mês e o dia dos nossos compromissos da agenda com essa data, a regra é simples: Caso essas três informações sejam iguais encontramos o nosso compromisso!

Novamente para garantir a modularização do nosso código a rotina para encontrar um compromisso será colocada em uma biblioteca, nesse caso específico, na biblioteca “agenda.cpp” e “agenda.h” pois essa função também faz parte das funcionalidades da nossa agenda.

Lembre-se que as bibliotecas devem agrupar as funcionalidades de acordo com um contexto, até agora temos dois contextos: Um é o das funcionalidades da agenda e o outro é o das funcionalidades do tempo.

20.1 O bixo vai pegar: Segura na minha mão e vem!

Sem mais enrolação comecemos pela linha adicionada no arquivo agenda.h:

```
1 std::vector<struct itemDaAgenda> recuperarCompromissosDeHoje(std  
  ::vector<struct itemDaAgenda> &lista);
```

Código 36: Protótipo da função recuperarCompromissosDeHoje

É uma única linha que precisa ser destrinchada:

Primeiramente temos **std::vector<struct itemDaAgenda>** indicando que função retornará uma lista de “struct itemDaAgenda”.

Seguido disso temos o nome da função: “recuperarCompromissosDeHoje”

E, entre parênteses, temos

std::vector<struct itemDaAgenda> &lista indicando que a função receberá uma **referência** para uma lista de “struct itemDaAgenda”.

Resumindo: Nossa função vai procurar os compromissos em uma lista de “struct itemDaAgenda” e retornar os resultados em outra lista “struct itemDaAgenda”! Ou ainda: Recebo uma lista, crio outra a partir da seleção de alguns itens e retorno essa lista de selecionados.

Vamos agora à implementação dessa função:

```
1 std::vector<struct itemDaAgenda> recuperarCompromissosDeHoje(std
  ::vector<struct itemDaAgenda> &lista)
2 {
3     auto data = recuperarDataAtual();
4
5     std::vector<struct itemDaAgenda> resultado;
6
7     for (unsigned int i = 0; i < lista.size(); i++)
8     {
9         if (data->tm_year + 1900 != lista[i].ano) continue;
10        if (data->tm_mon + 1 != lista[i].mes) continue;
11        if (data->tm_mday != lista[i].dia) continue;
12
13        resultado.push_back(lista[i]);
14    }
15
16    return resultado;
17 }
18
```

Código 37: Implementação da função recuperarCompromissosDeHoje

Talvez você tenha notado que nas linhas 9, 10 e 11 acessamos os elementos da estrutura usando “->” em vez de “.”

Isso se dá porque “data” é um ponteiro para uma estrutura, quando isso acontece devemos usar “->” em vez de “.”

Vamos lá: Na linha 3 chamamos a função “recuperarDataAtual” e guardamos a estrutura resultante dentro da variável “data”. Na linha 5 criamos a variável “resultado” que conterá a lista dos itens de agenda selecionados. Na linha 7 iniciamos um laço para (mais conhecido como loop for) cujo papel será a percorrer a lista fornecida em busca do dia, mês e ano que coincida com a data atual, ele faz isso criando a variável “i” e, a cada interação, soma 1 a mesma de forma que a variável “i” será usada como um indicador de posição dentro da lista fornecida começando pela posição 0, depois indo para posição 1, 2, 3, 4, 5

etc.

Na linha 9 acessamos o campo “tm_year” da estrutura e somamos a este 1900, pois, **de acordo com a documentação** o ano é representado pelo ano atual subtraído de 1900, o valor dessa soma é comparado com o ano do item da agenda, se os dois forem iguais esse desvio condicionado não é executado, caso sejam diferentes executamos o comando **continue**.

O comando **continue** faz com que **qualquer** laço pule para a próxima iteração.

Na Linha 10 verificamos se o mês da data atual é diferente do mês do item da agenda, se for diferente pulamos para a próxima iteração usando o comando **continue**.

Na Linha 11 testamos se o dia é diferente, caso seja verdade pulamos para a próxima iteração senão executamos a linha 13 cujo objetivo é armazenar o item da agenda para depois retorná-lo.

Quando temos um “if”, “for” ou “while” que executa **um único comando** podemos omitir as chaves, então:







```
if(teste){  
comando;  
}
```

é o mesmo que:

```
if(teste) comando;
```

Resumindo, este laço verifica se o dia, mês ou ano atuais são diferentes do dia, mês ou ano do item da agenda, se um destes for diferente significa que o item da agenda não pertence a data recuperada e que, portanto, não deve entrar na seleção, conseqüentemente, devemos pular aquela iteração para que o comando da linha 13 não seja executado!

Quando Laço é terminado vemos na linha 16 que a lista con-

tendo os resultados, ou seja, os itens selecionados, são retornados      ...ufa!

Em seguida vamos programar a exibição dessa seleção .

21 Depois da tempestade vem a calmaria

Legal! Agora temos todas as ferramentas necessárias para exibir nossos compromissos de hoje!

Começaremos por criar duas novas constantes na enumeração OPERACOES:

VER_COMPROMISSOS_DE_HOJE e SAIR, a primeira constante será usada para indicar que se quer mostrar os compromissos atuais e a segunda para indicar que desejamos fechar nosso programa.

Nossa enumeração ficará assim:

```
1 enum OPERACOES
2 {
3     INSERIR_COMPROMISSO, EXIBIR_COMPROMISSOS,
4     VER_COMPROMISSOS_DE_HOJE, SAIR
5 };
```

Código 38: As duas novas constantes

Voltando ao arquivo principal vamos alterar o nosso “switch” para que o mesmo reflita as novas opções disponíveis. Como já temos a função “mostrarItensDaAgenda” e a função “recuperarCompromissosDeHoje” podemos combiná-las para obter a funcionalidade que procuramos.

Para fechar nosso programa a única coisa que faremos é sair da função principal.

Nosso “switch” fica assim:

```

1 std::vector<itemDaAgenda> compromissosDeHoje;
2
3 while (true)
4 {
5     OPERACOES escolha = exibeMenu();
6
7     switch (escolha)
8     {
9         case INSERIR_COMPROMISSO:
10             adicionarItemNaAgenda(listaDoItensDaAgenda);
11             break;
12         case EXIBIR_COMPROMISSOS:
13             mostrarItensDaAgenda(listaDoItensDaAgenda);
14             break;
15         case VER_COMPROMISSOS_DE_HOJE:
16             // Recupera os compromissos de hoje
17             compromissosDeHoje = recuperarCompromissosDeHoje(
18                 listaDoItensDaAgenda);
19             // Mostra os compromissos
20             mostrarItensDaAgenda(compromissosDeHoje);
21             break;
22         case SAIR:
23             // Retorna ao sistema operacional
24             return 0;
25         default:
26             std::cout << "Opcao invalida" << std::endl;
27             break;
28     }
29 }

```

Código 39: Novo switch

A nossa modificação começa pela linha 1 onde declaramos a lista que conterà os compromissos de hoje, seguimos para a linha 15 onde fazemos uma referência a constante `VER_COMPROMISSOS_DE_HOJE`. Na linha 17 recuperamos os compromissos cadastrados para hoje e criamos uma nova lista chamada de “compromissos”, perceba o uso da palavra **auto** que, **nesse caso**, melhora a leitura do código, seguimos para a linha 19 onde os compromissos recuperados são passados para a função “mostrarItensDaAgenda” que finalmente mostra os compromissos

na tela.

Na linha 21 Fazemos uma referência a constante SAIR para na linha 23 retornarmos 0 para o sistema operacional fazendo com que nosso programa pare e termine sem problemas.

Não podemos esquecer de alterar a função “exibeMenu”:

```
1 OPERACOES  exibeMenu()  
2 {  
3     std::cout << "Escolha o que deseja fazer:" << std::endl;  
4     std::cout << INSERIR_COMPROMISSO << " – Inserir um  
5     compromisso" << std::endl;  
6     std::cout << EXIBIR_COMPROMISSOS << " – Exibir todos os  
7     compromissos" << std::endl;  
8     std::cout << VER_COMPROMISSOS_DE_HOJE << " – Exibir  
9     compromissos de hoje" << std::endl;  
10    std::cout << SAIR << " – Sair do programa" << std::endl;  
11  
12    int escolha = 0;  
13  
14    std::cin >> escolha;  
15  
16    return (OPERACOES) escolha;  
17 }
```

Código 40: O novo menu

Perceba as novas opções colocadas nas linhas 6 e 7!

22 Eis onde chegamos!

Abaixo vão os códigos que fizemos:

22.1 Arquivo principal

```
1 #include <iostream>  
2 #include "agenda.h"  
3  
4 std::vector<struct itemDaAgenda> listaDoItensDaAgenda;  
5
```

```

6 int main(int argc, char **argv)
7 {
8     std::vector<itemDaAgenda> compromissosDeHoje;
9
10    while (true)
11    {
12        OPERACOES escolha = exibeMenu();
13
14        switch (escolha)
15        {
16            case INSERIR_COMPROMISSO:
17                adicionarItemNaAgenda(listaDoItensDaAgenda);
18                break;
19            case EXIBIR_COMPROMISSOS:
20                mostrarItensDaAgenda(listaDoItensDaAgenda);
21                break;
22            case VER_COMPROMISSOS_DE_HOJE:
23                // Recupera os compromissos de hoje
24                compromissosDeHoje = recuperarCompromissosDeHoje(
25                    listaDoItensDaAgenda);
26                // Mostra os compromissos
27                mostrarItensDaAgenda(compromissosDeHoje);
28                break;
29            case SAIR:
30                // Retorna ao sistema operacional
31                return 0;
32            default:
33                std::cout << "Opcao invalida" << std::endl;
34                break;
35        }
36    }
37 }

```

Código 41: Arquivo principal

22.2 Os cabeçalhos da biblioteca agenda “agenda.h”

```

1 #ifndef AGENDA_H
2 #define AGENDA_H
3
4 #include <string>
5 #include <vector>
6
7 enum OPERACOES

```

```

8 {
9     INSERIR_COMPROMISSO, EXIBIR_COMPROMISSOS,
        VER_COMPROMISSOS_DE_HOJE, SAIR
10 };
11
12 struct itemDaAgenda
13 {
14     int dia = 0;
15     int mes = 0;
16     int ano = 0;
17     std::string descricao;
18 };
19
20 OPERACOES exhibeMenu();
21
22 void mostrarItensDaAgenda(std::vector<struct itemDaAgenda> &lista
    );
23
24 void adicionarItemNaAgenda(std::vector<struct itemDaAgenda> &
    lista);
25
26 std::vector<struct itemDaAgenda> recuperarCompromissosDeHoje(std
    ::vector<struct itemDaAgenda> &lista);
27
28 #endif
29

```

Código 42: Os cabeçalhos da biblioteca da agenda “agenda.h”

22.3 As implementações da biblioteca agenda “agenda.cpp”

```

1 #include "agenda.h"
2 #include "tempo.h"
3
4 #include <string>
5 #include <vector>
6 #include <iostream>
7
8 OPERACOES exhibeMenu()
9 {
10     std::cout << "Escolha o que deseja fazer:" << std::endl;
11     std::cout << INSERIR_COMPROMISSO << " – Inserir um compromisso"
        << std::endl;
12     std::cout << EXIBIR_COMPROMISSOS << " – Exibir todos os
        compromissos" << std::endl;

```



```

13 std::cout << VER_COMPROMISSOS_DE_HOJE << " - Exibir
    compromissos de hoje" << std::endl;
14 std::cout << SAIR << " - Sair do programa" << std::endl;
15
16 int escolha = 0;
17
18 std::cin >> escolha;
19
20 return (OPERACOES) escolha;
21 }
22
23 void adicionarItemNaAgenda(std::vector<struct itemDaAgenda> &
    lista)
24 {
25     struct itemDaAgenda novoItem;
26
27     std::cout << "Informe o dia:" << std::endl;
28     std::cin >> novoItem.dia;
29
30     std::cout << "Informe o mes:" << std::endl;
31     std::cin >> novoItem.mes;
32
33     std::cout << "Informe o ano:" << std::endl;
34     std::cin >> novoItem.ano;
35
36     std::cin.ignore();
37
38     std::cout << "Descreva o compromisso:" << std::endl;
39     std::getline(std::cin, novoItem.descricao);
40
41     lista.push_back(novoItem);
42 }
43
44 void mostrarItensDaAgenda(std::vector<struct itemDaAgenda> &lista)
45 {
46     // i eh a variavel que indica a posicao que queremos
47     // acessar dentro de nossa lista
48     for (unsigned int i = 0; i < lista.size(); i++)
49     {
50         // Exibe o dia, mes e ano separados por "-" mais a descricao
51         std::cout << lista[i].dia << "-" << lista[i].mes << "-" <<
            lista[i].ano << ": " << lista[i].descricao << std::endl;
52     }
53 }

```

```

54
55 std::vector<struct itemDaAgenda> recuperarCompromissosDeHoje(std
    ::vector<struct itemDaAgenda> &lista)
56 {
57     auto data = recuperarDataAtual();
58
59     std::vector<struct itemDaAgenda> resultado;
60
61     for (unsigned int i = 0; i < lista.size(); i++)
62     {
63         if (data->tm_year + 1900 != lista[i].ano) continue;
64         if (data->tm_mon + 1 != lista[i].mes) continue;
65         if (data->tm_mday != lista[i].dia) continue;
66
67         resultado.push_back(lista[i]);
68     }
69
70     return resultado;
71 }
72

```

Código 43: As implementações da biblioteca da agenda “agenda.cpp”

22.4 Os cabeçalhos da biblioteca tempo “tempo.h”

```

1 #ifndef TEMPO_H
2 #define TEMPO_H
3
4 #include <ctime>
5
6 struct tm* recuperarDataAtual();
7
8 #endif
9

```

Código 44: O cabeçalho da biblioteca da tempo “tempo.h”

22.5 As implementações da biblioteca da tempo “tempo.c”

```

1 #include "tempo.h"
2
3 struct tm* recuperarDataAtual()
4 {
5     // Variavel usada para armazenar a
6     // quantidade de segundos para

```

```

7 // representar uma data
8 time_t tt;
9
10 // Armazena a quantidade segundos que
11 // se passaram desde as 0 horas de 1
12 // de janeiro de 1970 no horario de
13 // Greenwich
14 time(&tt);
15
16 // Transforma esses segundos em uma
17 // estrutura que separa o dia, mes,
18 // ano, etc. e a retorna.
19 return localtime(&tt);
20 }
21

```

Código 45: As implementações da biblioteca da tempo “tempo.cpp”

23 Bibliotecas externas... bom... digamos que teremos alguns problemas.

Em C++ temos várias bibliotecas disponíveis por padrão: vector, iostream, etc. Mas e se precisarmos de alguma funcionalidade que as bibliotecas padrão não fornecem? Neste caso precisaremos usar uma biblioteca externa. Uma biblioteca externa é um conjunto de funcionalidades pré-programadas que não estão disponíveis por padrão na linguagem.

O tópico de bibliotecas externas, na verdade, não estava previsto para ser incluído neste material pois a forma de usá-las varia muito de acordo com o sistema que você está usando.

Referenciar uma biblioteca externa no seu projeto pode ser muito simples ou complicado variando entre os ambientes de programação. Por isso neste capítulo não entraremos em detalhes sobre como usar essas bibliotecas e sim como elas são organizadas.

A primeira coisa que temos nas bibliotecas em C++ são os arquivos de cabeçalho ou, como já vimos, os arquivos com extensão “.h”, novamente, os arquivos “.h” contém as assinaturas dos métodos, constantes ou outras estruturas que serão usadas na programação.

Os conteúdos dos arquivos “.h” podem ser chamado de símbolos. Os arquivos “.h” tem a responsabilidade de externar todos as funcionalidades disponíveis naquela biblioteca.

Mas nem só de “.h” viverá a pessoa! Bibliotecas também são constituídas por outros arquivos! A extensão desses pode depender para qual sistema operacional a biblioteca foi compilada.

Além disso existem dois tipos de bibliotecas que temos que considerar:

- A biblioteca estática que será incorporada ao executável no momento da compilação e vinculação (linkagem).
- A biblioteca dinâmica que se manterá como um arquivo separado do executável e cuja as funcionalidades serão carregadas no momento da execução do programa de acordo com a demanda.

A vantagem das bibliotecas estáticas é que, geralmente, o executável resultante tende a ser mais rápido, no entanto, o tamanho dele fica maior e ocupa mais memória.

A vantagem das bibliotecas dinâmicas é que vários programas podem carregar a mesma biblioteca diminuindo assim a necessidade de armazenamento, no entanto, os programas gerados para usar esse tipo de biblioteca tendem a ser ligeiramente mais lentos justamente, entre outras coisas, por causa deste tempo a mais

para carregamento.

Dito isso Vamos aos formatos de arquivos para os respectivos sistemas operacionais GNU/Linux, Unix, Mac e Windows

No GNU/Linux, Unix ou Mac a extensão das bibliotecas:

- Estáticas é “.a”: Exemplo: ugauga.**a**
- Dinâmicas é “.so”: Exemplo: ugauga.**so**

No Windows a extensão das bibliotecas:

- Estáticas é “.lib”: Exemplo: ugauga.**lib**
- Dinâmicas é “.dll”: Exemplo: ugauga.**dll**

Em síntese para referenciar uma biblioteca em C++ são necessários o arquivo “.h” e o arquivo “.a” ou “.so” para GNU/Linux, Unix ou Mac ou o arquivo “.lib” ou “.dll” para Windows.

Algumas pessoas gostam de incluir tais arquivos dentro de uma pasta no seu projeto, muitas vezes esta pasta se chama “dependências” ou “lib” ou ainda o nome que o desenvolvedor ou a desenvolvedora preferir.

É importante lembrar que não basta ter esses arquivos dentro do seu projeto, é necessário configurar o seu ambiente de programação ou de compilação para encontrá-los!