Giovane Avancini

Introdução à linguagem de programação C++

Sumário

Apresentação

Este texto foi elaborado com o intuito de auxiliar na introdução da linguagem de programação C++ para aplicações práticas de engenharia. Por ser uma linguagem bastante completa e abrangente, é recomendado que os interessados busquem outros materiais para aprofundarem seus conhecimentos.

Algumas referências sugeridas são:

- Introdução à Programação Orientada a Objetos com C++. Antonio Mendes da Silva Filho;
- Accelerated C++: Practical Programming by Example. Andrew Koenig e Barbara E. Moo;
- C++ Series. TheChernoProject, disponível em: https://www.youtube.com/playlist?list=PLlrATfBNZ98dudnM48yfGUldqGD0S4FFb;
- C++ Reference. Disponível em: http://www.cplusplus.com/reference/.

O primeiro livro, em portugês, é recomendado para aqueles alunos que não tiveram um contato prévio com o paradigma de orientação a objetos, mostrando detalhadamente conceitos como objeto, herança, poliformismo, encapsulamento, etc.

Já o segundo, em inglês, aborda o assunto de maneira mais acelerada, através de códigos já escritos, sendo muito bom para aqueles que já possuem um certo conhecimento sobre o assunto e buscam apenas aprender sobre a linguagem C++.

A terceira referência é uma série de vídeos feita pelo engenheiro de softwares da EA Games Ian Chernkov, a qual acredito ser a forma mais rápida de aprender tópicos específicos da linguagem.

A quarta referência é um site/manual da linguagem extremamente útil e prático. É ideal para sanar dúvidas pontuais sobre diferentes tópicos, como variáveis, funções, bibliotecas, etc.

Capítulo 1

Como funciona um programa em C++

Antes de falarmos especificamente sobre a linguagem de programação, é importante entendermos como saímos de um arquivo de texto (*source file*) e chegamos em um binário utilizando C++. Esse binário pode ser uma biblioteca (stática ou dinâmica) ou um binário executável, que é o objeto de interesse nesse curso. O exemplo a seguir mostra um programa bastante simples mas muito utilizado para exemplificar cada componente da linguagem, implementado em um único arquivo "HelloWorld.cpp":

```
1  #include <iostream>
2
3  int main()
4  {
5    std::cout << "Hello Word!" << std::endl;
6    std::cin.get();
7  }
8
```

Código 1.1: Hello World

Na linha 1 temos uma diretiva começada pelo caractere #. Em C++, tudo o que começa com esse caractere é entendido como uma diretiva de pré processador, e todo esse código será processado imediatamente antes do processo de compilação. No nosso caso, a diretiva é o comando *include*, que tem como finalidade copiar todo o conteúdo de um arquivo externo, geralmente um arquivo .h (*header file*) e colar no arquivo onde está se trabalhando. No nosso caso, todo o conteúdo do arquivo "iostream" está sendo colado no arquivo de texto "HelloWorld.cpp".

Na linha 3 é declarada a função main. Essa é uma função especial e obrigatória em todos os programas escritos em C++, e funciona como um ponto de partida. Isso quer dizer que quando o programa é executado, o processador irá executar a partir da função main, linha por linha do nosso código, em ordem, até que se chegue no final do escopo da função, onde a execução é terminada.

Outro conceito importante que vemos aqui é o de escopo. Em C++ utiliza-se o caractere { para denotar o início do escopo de uma função e } para indicar onde seu escopo termina. Assim, as linhas 5 e 6 fazem parte do escopo da função main. É importante entender a definição de escopo pois está diretamente ligado ao tempo de vida das variáveis declaradas em C++, e será visto com mais detalhe a frente.

As linhas 5 e 6 são efetivamente os comandos que serão executados no nosso programa. A razaão para termos incluído o arquivo "iostream" é que a função *std::cout* está definida nele, e o que ela faz é imprimir os argumentos desejados para o terminal do computador. O operador << nesse caso

atua exatamente como uma função que informa os argumentos que serão imprimidos, e podem ser usados sucessivamente quando deseja-se imprimir mais de um argumento. Assim, podemos pensar que a função da linha 5 poderia ser hipoteticamente substituída por:

```
1 std::cout.print("Hello Word!").print(std::endl);
2
```

Código 1.2: Operador <<

O comando *std::endl* informa ao terminal para avançar para a próxima linha. Pode ser substituído pelo caractere de quebra de linha "\n". Por outro lado, na linha 6 temos uma função que pausa a executação do programa até que a tecla "ENTER" seja pressionada.

1.1 Compilação

Vimos anteriormente que, basicamente, existem dois tipos de arquivos de texto: os *source files* (.cpp) e os *header files* (.h). Quando criamos um projeto, todos os arquivos .cpp são compilados, os .h não. Eles são arquivos que são incluídos em arquivos .cpp via comando *include*, e então, esse arquivo .cpp é compilado. Ainda, cada arquivo .cpp será compilado individualmente em um arquivo chamado *object* (.o ou .obj, a depender do compilador). Uma vez compilados em arquivos .o, eles devem ser combinados em um único arquivo executável, dando origem assim ao nosso programa. O responsável por realizar essa tarefa é o *linker*. A Figura 1.1 tenta resumir o processo de como os arquivos textos são transformados em um binário executável.

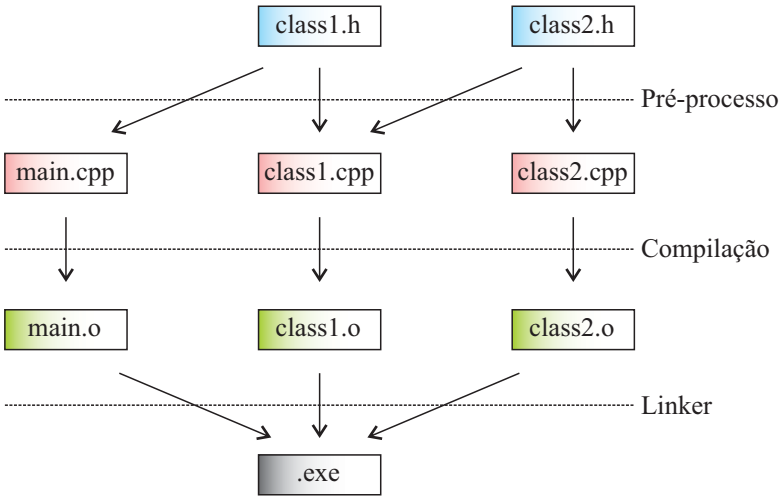

Figura 1.1: Processo de compilação

Explicar como o compilador transforma um arquivo de texto em código de máquina é um processo complicado, e aos interessados, sugiro dar uma olhada no seguinte vídeo. Para saber mais sobre como o linker funciona, veja este vídeo.

Capítulo 2

Elementos da linguagem

Uma vez entendido como um programa em C++ é estruturado e compilado, vamos aos elementos da linguagem.

2.1 Variáveis

Existem diversos tipos de variáveis em C++, e basicamente, o que diferencia elas não é o seu uso, e sim a quantidade de memória que elas ocupam. O tipo da variável é apenas uma forma de dizer ao compilador e à memória qual o espaço que deverá ser reservado para guardar aquele dado. As variáveis que não precisam de uma biblioteca externa ou de um *header file* para serem usadas são chamadas de primitivas. A Tabela 2.1 mostra as mais usuais, assim como a quantidade de memória que cada uma ocupa.

Tipo	Sintaxe	Tamanho (byte)
Inteiro	int	4
Real	float	4
Real	double	8
Caractere	char	1
Booleano	bool	1

Tabela 2.1: Variáveis primitivas

Há muitas outras variáveis que são criadas a partir dessas apresentadas, e são chamadas de variáveis customizadas. Um exemplo que será visto mais a frente é a variável *std::string*, criada a partir da variável *char*, mas que possuem diversos métodos atrelados à ela.

2.2 Funções

Funções são blocos de código que executam uma certa tarefa. Na maioria das vezes, essa tarefa é repetida diversas vezes na execução do programa, assim, separando o código em funções o deixa mais organizado. Em programação orientada a objetos, quando uma função está atrelada a uma classe, dizemos que essa função é um método. Por enquanto, vamos nos ater a funções que não são definidas em um classe.

Suponhamos um código em que seja necessário multiplicar dois números inteiros diversas vezes. Podemos definir então uma função que execute essa tarefa:

```
1  #include <iostream>
2
3  int multiplicar(int a, int b)
4  {
5    return a * b;
6  }
7
8  int main()
9  {
10   int resultado = multiplicar(2, 3);
11
12   std::cout << resultado << std::endl;
13   std::cin.get();
14 }
15
```

Código 2.1: Função de multiplicação

Note-se então que antes de usarmos a função *multiplicar* na linha 10 devemos primeiro declarar e implementar essa função. A sintaxe para declarar uma função é mostrada na linha 3. Temos que informar o tipo de retorno da função, seguido pelo seu nome e entre parênteses seus argumentos ou parâmetros. Vimos anteriormente que os caracteres { } indicam o escopo da função, assim, todo código que estiver entre esses caracteres compreendem a definição ou implementação de uma função, ou seja, a tarefa que será executada. A última coisa a se fazer é informar o dado que será retornado da função. Vejamos que, neste exemplo, estamos retornando uma expressão que resultará em um inteiro, e portanto, o compilador não irá ter problemas.

Uma vez declarada e definida a função, é possível utiliza-la como na linha 10. Vemos que, por retornar um dado do tipo inteiro, essa função deve ser atribuída a uma variável de mesmo tipo.

Há funções em que deseja-se executar uma tarefa mas sem, necessariamente, retornar algum valor. Nesses casos declara-se a função como sendo do tipo *void*, que em C++ significa não ter um tipo definido. Ainda, os argumentos das funções não são elementos obrigatórios, podendo ser omitidos quando a tarefa a ser executada não depende de valores externos ou definidos pelo usuário. Segue um exemplo:

```
1  #include <iostream>
2
3  void multiplicar()
4  {
5    std::cout << 5 * 3 << std::endl;
6  }
7
8  int main()
9  {
10   multiplicar();
11   std::cin.get();
12 }
13
```

Código 2.2: Função sem retorno

Nesse caso, ao chamarmos a função multiplicar, o valor da multiplicação entre os inteiros 5 e 3 irá ser imprimido no terminal, ou seja, sempre que for chamada teremos o valor 15 impresso na tela. Diferente do código anterior, nesse caso a função irá apenas executar uma determinada tarefa, sem retornar um valor para ser usado posteriormente no código.

2.3 Condições e controle de fluxo

Ao programar códigos mais complexos, muitas vezes é necessário executar algumas tarefas ou mudar o fluxo de execução se algumas condições acontecerem ou forem satisfeitas. Nesses casos utilizamos os chamados *if statements* que têm por finalidade criar novas ramificações no código e direcionar o fluxo em tempo de execução. Vejamos o seguinte bloco de código:

```
1  #include <iostream>
2
3  int main()
4  {
5    int x = 5;
6
7    if (x > 0)
8    {
9      std::cout << "Hello World!" << std::endl;
10   }
11
12   std::cin.get();
13 }
14
```

Código 2.3: Comando if

Criamos uma variável do tipo *int* e atribuímos a ela o valor 5. Suponhamos que, caso seu valor seja maior que 0, iremos imprimir "Hello World!" no terminal. Assim, na linha 6 usamos o comando de condição *if*, seguido pela condição que será analisada entre parênteses. Caso essa condição seja satisfeita, ou seja, retorne *true*, o bloco de código entre os caracteres { } será executado. Por outro lado, se a condição retornar *false*, o fluxo do programa seguirá normalmente e, no nosso caso, não irá imprimir o valor de x no terminal.

Logicamente que, quando quisermos executar outro bloco de código quando a condição analisada não seja satisfeita, podemos usar o comando *else*, da seguinte maneira:

```
1  #include <iostream>
2
3  int main()
4  {
5    int x = -5;
6
7    if (x > 0)
8    {
9      std::cout << "Hello World!" << std::endl;
10   }
11   else
12   {
13     std::cout << "Bye World!" << std::endl;
14   }
15
16   std::cin.get();
17 }
18
```

Código 2.4: Comando else

Nesse caso iremos obter como resultado o texto "Bye World!" impresso no terminal, visto que a condição x > 0 não será satisfeita, forçando o programa a executar o bloco de código que foi

implementado dentro do escopo do comando *else*. Ainda, é possível utilizar o comando *else if* para impor múltiplas condições que, só serão checadas, caso a condição imediatamente anterior a ela não for satisfeita. Vejamos:

```
#include <iostream>

int main()
{
  int x = -5;

  if (x > 0)
  {
    std::cout << "x e positivo" << std::endl;
  }
  else if (x < 0)
  {
    std::cout << "x e negativo" << std::endl;
  }
  else
  {
    std::cout << "x e nulo" << std::endl;
  }

  std::cin.get();
}

```

Código 2.5: Comando else if

Uma vez que x = -5, a primeira condição não irá ser satisfeita, e então, a segunda condição será analisada para sabermos se x < 0.

Considerando que as instruções de um programa são guardadas na memória, utilizar *if statements* significa, basicamente, pular de um trecho da memória para outro, a depender das condições estabelecidas, e esse salto na memória, em geral, resulta em perda de desempenho. Sendo assim, é importante utilizar esse tipo de comando com cuidado e apenas em casos necessários.

Ainda, na Tabela 2.2 encontram-se os operadores booleanos para comparação de dados:

Operador	Sintaxe
E	&&
Ou	||
Igual	==
Diferente	!=
Maior	>
Maior igual	>=
Menor	<
Menor igual	<=

Tabela 2.2: Operadores booleanos

2.4 Loops de repetições

Os chamados loops de repetição são comandos que permitem executar um mesmo bloco de código quantas vezes for necessário, até que um determinado critério de parado seja satisfeito. Em C++ há basicamente duas maneiras de atingir esse objetivo, através dos comandos *for* e *while*. Vamos supor que queremos executar uma mesma linha de comando 5 vezes em sequência. Para isso, é óbvio que poderíamos repeti-la manualmente 5 vezes. Porém, para códigos mais extensos e repetições maiores, isso se torna inviável e nessas situações devemos usar os *loops* de depetições. Vejamos o exemplo abaixo:

```
1  #include <iostream>
2
3  int main()
4  {
5    for (int i = 0; i < 5; i++)
6    {
7      std::cout << "Hello World!" << std::endl;
8    }
9
10   std::cin.get();
11 }
12
```

Código 2.6: Comando for

O comando *for*, como podemos observar na linha 5, possui 3 argumentos separados pelo caractere ;. O primeiro é a declaração da variável que será utilizada como iterador ou contador de repetições, seguido pelo critério de parada e por fim o código que será executado ao final de cada iteração. No nosso caso declaramos uma variável do tipo *int* e a inicializamos com o valor 0. O critério de parada funciona como uma condição que determina se haverá ou não uma próxima repetição, ou seja, nosso bloco de código relacionado ao *for loop* será executado até que a condição estipulada deixe de ser verdadeira. Nesse caso, quanto i for igual a 6, automaticamente o loop se encerra, uma vez que 5 é igual a 5 e portanto, a condição torna-se falsa. Por último, queremos que nosso iterador seja incrementado ao final de cada repetição, e em C++ é possível realizar essa operação por meio do operador unário ++. É importante atentarmos ao fato de que se utilizarmos uma condição que é sempre verdadeira como critério de parada, iremos obter um loop infinito.

Outro jeito de conseguir o mesmo resultado é usando o comando *while*:

```
1  #include <iostream>
2
3  int main()
4  {
5    int i = 0;
6    while (i < 5)
7    {
8      std::cout << "Hello World!" << std::endl;
9      i++;
10   }
11
12   std::cin.get();
13 }
14
```

Código 2.7: Comando while

Esse tipo de looping pode ser usado nas mesmas circunstâncias que o comando *for*, acaba sendo mais uma questão de preferência do usuário. Convencionalmente o *for loop* pode ser usado sem que haja a necessidade de ter uma variável previamente declarada, e é mais adequado para situações em que se conhece o número de iterações desejadas ou quando pretende-se iterar sobre um *array* ou vetor.

Da mesma forma que tínhamos maneiras de controlar o fluxo de execução do programa através dos *if statementes*, temos comandos que auxiliam a controlar o fluxo em laços de repetições, são eles:

- *continue*: quando usado sob uma determinada condição, o comando *continue* irá pular para a próxima iteração do *loop*, não executando os comandos que estão abaixo dele;
- *break*: automaticamente interrompe o processo iterativo e move a execução do programa para fora do *loop*.

2.5 Ponteiros

Talvez ponteiros seja o conteúdo mais impotante de todo o curso, e é um assunto onde a maioria das pessoas se confunde e larga o C++. Visto que todas as instruções que passamos ao compilador, como declarar uma variável, uma função, criar uma nova classe, são salvas na memória, os ponteiros nos permite manipular e controlar esse endereço de memória que está sendo usado. Ao contrário do que parece, **ponteiros são apenas inteiros que guardam um endereço de memória**.

Imaginem uma cidade em que só exista uma grande rua, com um início e um fim. Para simplicar, essa rua possui uma grande quantidade de casas em apenas um de seus lados e cada casa possui um número de identificação, um endereço. A memória do computador funciona exatamente assim, é um grande bloco linear com diversos endereços onde são salvas as informações do nosso programa.

Agora supondo que cada casa dessa rua, que tem um endereço único, representa 1 byte de dados. Caso algum morador de uma das casas peça, digamos, algo por delivery, deve haver uma maneira da encomenda ser entregue no endereço correto. Assim, os ponteiros exercem justamente essa função de armazenar o endereço de memória a qual a informação deve ser entregue ou recebida. Vejamos um exemplo:

```
1  #include <iostream>
2
3  int main()
4  {
5    int var = 8;
6    int* ptr = &var;
7
8    std::cout << ptr << std::endl;
9    std::cin.get();
10 }
11
```

Código 2.8: Exemplo com ponteiros

Na linha 6 temos a declaração de um ponteiro *ptr* do tipo *void* apontando para o endereço de memória da variável *var*. Repare que a forma como declaramos um ponteiro é parecida com a declaração de uma variável qualquer, exceto pelo caractere * que colocamos a frente do tipo, indicando que aquilo é, na verdade, um ponteiro. Como foi falado, um ponteiro guarda um endereço

de memória e por isso seria um erro atribuir o valor da variável *var* ao ponteiro *ptr*. Mas em C++ há um operador unário chamado endereço (&) que quando aplicado sobre uma variável retorna o endereço de memória que está sendo usado por aquela variável. De maneira semelhante podemos acessar, de fato, o dado armazenado no endereço de memória salvo em um ponteiro por meio do operador unário desreferência (*), aplicando antes do operando (ponteiro).

```
#include <iostream>

int main()
{
  int var = 8;
  int* ptr = &var;

  std::cout << *ptr << std::endl;
  std::cin.get();
}

```

Código 2.9: Desreferência

Assim, podemos alterar o valor inicial da variável *var* simplesmente alterando o valor desreferenciado de um ponteiro.

```
#include <iostream>

int main()
{
  int var = 8;
  int* ptr = &var;
  *ptr = 10;

  std::cout << var << std::endl;
  std::cin.get();
}

```

Código 2.10: Alterando uma variável por meio de um ponteiro

Outra coisa legal sobre ponteiros é que, se eles são variáveis que guardam o endereço de memória de um certo dado, podemos criar ponteiros que apontam para outro ponteiro, e assim sucessivamente. Vejamos:

```
#include <iostream>

int main()
{
int var = 8;
int* ptr = &var;
int** ptr2 = &ptr;

std::cout << var << std::endl;
std::cin.get();
}

```

Código 2.11: Ponteiros duplos

Agora, temos o ponteiro *ptr2* apontando para o endereço de memória do ponteiro *ptr*, que por sua vez aponta para a variável *val*. Assim, faz sentido ter que desreferenciar duas vezes o ponteiro *ptr2* para obtermos o valor guardado no endereço de memória da variável *val*:

```
#include <iostream>

int main()
{
int var = 8;
int* ptr = &var;
int** ptr2 = &ptr;

std::cout << *ptr2 << std::endl;
std::cin.get();
}

```

Código 2.12: Dupla desreferência

Outro papel fundamental que os ponteiros exercem está relacionado ao tempo de vida da variável. Mais adianta iremos ver formas de declarar um objeto, que basicamente pode ser no *stack* ou no *heap* da memória.

2.6 Referências

Ponteiros e referências são quase a mesma coisa em C++, exceto pelo fato de que um ponteiro possui mais funcionalidades e versatilidade do que uma referência. O compilador irá trata-los exatamente da mesma forma, entretanto, há diferença na sintaxe entre um e outro. Como o próprio nome ja diz, referência é uma "variável"que se refere a outra já existente. O fato da palavra variável estar escrito entre aspas é porque, nesse caso, uma referência não ocupa espaço na memória, diferente de um ponteiro. Assim, uma referência é uma espécie de "apelido"que damos a uma variável já existente. Vejamos o exemplo:

```
#include <iostream>

int main()
{
  int a = 1;
  int& b = a;
  std::cout << b << std::endl;
  b+=2;
  std::cout << a << std::endl;

  std::cin.get();
}

```

Código 2.13: Referência

O código acima, se tudo estiver correto, irá escrever no terminal os números 1 e 3. Já vimos na seção anterior que o operador (&) retorna o endereço da variável operando. Porém, quando o mesmo símbolo é utilizado junto ao tipo da variável a ser declarada, isso indica na verdade que aquela variável é uma referência. Ou seja, teoricamente, a referência b atua da mesma forma que um ponteiro que aponta para a variável a, exceto por um detalhe: uma referência é um elemento

not assinable. Isso significa que uma vez declarada e linkada com uma variável, não se pode mudar essa variável a faz-se referência. Por isso, também não se pode declarar uma referência e deixa-la sem referenciar uma variável pré existente, ou seja:

```
1  int a = 1;
2  int& b;
3  b = a;
4
```

Código 2.14: Limitação de uma referência

O código acima não poderá ser compilado uma vez que na linha 2, a referência b não foi atribuída a uma variável já existente, no caso a. Sendo assim, qual a usabilidade de referências então? Sua funcionalidade é melhor compreendida quando utilizamos funções. Suponha-se então que vamos implementar uma função para increment uma dada variável pré existente:

```
1  #include <iostream>
2
3  void increment(int value)
4  {
5    value++;
6  }
7
8  int main()
9  {
10   int a = 2;
11   increment(a);
12   std::cout << a << std::endl;
13   std::cin.get();
14 }
15
```

Código 2.15: Argumentos por valor

A primeira vista podemos pensar que o código acima não apresenta problemas. De fato, se compilarmos não ocorrerá nenhuma falha, porém, ao executarmos temos que o valor de a impresso na tela será 2. Isso ocorre porque passamos a variável a para a função increment como "valor". Por padrão, sempre que passamos um argumento para uma função por valor, o que ocorre é que uma cópia desse argumento é temporariamente criada no escopo da função, e automaticamente destruída quando a função termina sua execução. Ou seja, estaríamos incrementando uma cópia da variável a que está sendo temporariamente criada, e não a variável já existente. Nesse caso temos duas alternativas: passar o argumento da função como um ponteiro ou como referência, e nesse caso, utiliza-se mais a segunda opção por deixar o código mais limpo e fácil de compreender. Nesse caso, o código acima poderia ser reescrito para que a variável a fosse incrementada da seguinte forma:

```
1  void increment(int& value)
2  {
3    value++;
4  }
5
```

Código 2.16: Argumentos por referência

2.7 Arrays

Primeiramente, o conteúdo que vimos em ponteiros será a base para o entendimento dos *arrays*. Um *array* nada mais é que um arranjo de variáveis de mesmo tipo que segue uma determinada ordem. É apenas uma forma de facilitar o processo de criação e agrupamento de variáveis que terão o mesmo propósito no nosso programa. O motivo de ser necessário compreender ponteiros para entender o funcionamento de um *array* é porque, basicamente, um array é um ponteiro que aponta para o endereço de memória do primeiro elemento do arranjo. O exemplo abaixo mostra como é feita a declaração de um array:

```
1  #include <iostream>
2
3  int main()
4  {
5    int exemplo[4];
6    exemplo[0] = 1;
7    exemplo[1] = 2;
8    exemplo[2] = 3;
9    exemplo[3] = 4;
10
11   for (int i = 0; i < 4; i++)
12   {
13     std::cout << exemplo[i] << std::endl;
14   }
15   std::cin.get();
16 }
17
```

Código 2.17: Declaração de arrays

Na linha 5 vemos que a sintaxe para a declaração de um *array* é semelhante a declaração de uma variável comum, exceto que agora devemos especificar quantos elementos terá esse arranjo, ou seja, seu tamanho. Após declarado, uma forma de acessar e atribuir valores para cada elemento desse *array* é por meio do operador [ ]. Vejamos então o porquê de um array ser exatamente um ponteiro que aponta para o endereço de memória de seu primeiro elemento:

```
1  #include <iostream>
2  int main()
3  {
4    int exemplo[4];
5    exemplo[0] = 1;
6    exemplo[1] = 2;
7    exemplo[2] = 3;
8    exemplo[3] = 4;
9
10   int* ptr = array;
11
12   for (int i = 0; i < 4; i++)
13   {
14     std::cout << *(ptr+i) << std::endl;
15   }
16   std::cin.get();
17
18   *(ptr) = 10;
19   *(ptr + 1) = 20;
20
```

```
21      for (int i = 0; i < 4; i++)
22      {
23        std::cout << exemplo[i] << std::endl;
24      }
25      std::cin.get();
26    }
27
```

Código 2.18: Array é um ponteiro que aponta para o primeiro elemento

Claramente pode-se manipular um *array* da mesma forma que manipula-se um ponteiro, pois são exatamente a mesma coisa. Ao declararmos o ponteiro *ptr* e apontarmos para o *exemplo* na linha 10, vemos que na verdade um *array* guarda um endereço de memória, ou seja, é um ponteiro.

2.8 Struct

As variáveis que temos conhecimento até o momento são ditas primitivas pois, à partir delas, podemos criar novas variáveis compostas. Se pensarmos nos objetos reais, faz sentido criarmos um tipo de dado que consegue representar, da forma mais próxima possível, os atributos e funções que operam sobre esse objeto. Por exemplo, se pensarmos em uma pessoa, faz sentido atribuirmos um nome para ela, altura, idade, logo, podemos pensar como a pessoa sendo um objeto com seus respectivos atributos. Isso é possível em C++ utilizando o comando *struct* ou *class*.

A diferença básica entre eles é que as classes foram incorporadas ao C++ com o intuito de substituir as structs, já existentes em C, de uma forma a abordar o conceito de encapsulamento. Em outras palavras, a única diferença consiste no fato de que os atributos e métodos de uma classe são privados por default, enquanto que nas *structs* eles são públicos. Vamos ao exemplo mencionado acima:

```
1   #include <iostream>
2   #include <string>
3
4   struct Pessoa
5   {
6     std::string nome_;
7     int idade_;
8     float altura_;
9     std::string pais_;
10    std::string cidade_;
11  };
12
13  int main()
14  {
15    Pessoa p1;
16    p1.nome_ = "Maria";
17    p1.idade_ = 30;
18    p1.altura_ = 1.6f;
19    p1.pais_ = "Brasil";
20    p1.cidade_ = "Sao Carlos";
21    std::cin.get();
22  }
23
```

Código 2.19: Exemplo de struct

2.9 Life time das variáveis

Antes de entendermos como funciona a vida de uma variável em C++, é necessário compreendermos como a memória do computador se encontra dividida: em uma parte chamada *stack* e uma outra parte chamada *heap*.

- stack: Compreende uma pequena parte do total da memória disponível, cerca de 1Mb em Windows e 8Mb em Linux. É a porção da memória destinada às operações voláteis, ou seja, de curto tempo de armazenamento. Por ser destinada ao armazenamento temporário de dados, sua velocidade de processamento é alta, sendo uma boa forma de alocar objetos e variáveis sempre que possível;
- heap: Compreende a maior parte do total da memória disponível e é destinada ao armazenamento de dados muito grandes ou de longa duração. Seu desempenho é bem menor comparado ao stack.

Por padrão, temos que o C++ aloca as variáveis e objetos no *stack*. Então surge a pergunta: se o *stack* é mais rápido que o *heap*, para que se preocupar com isso? E a resposta para essa pergunta está relacionada ao tempo de vida da variável alocada no stack. Esse tipo de variável ou objeto tem sua vida definida pelo escopo em que está inserida, ou seja, uma vez que a linha de execução do programa sair desse escopo, o espaço da memória alocado para essa variável será liberado. Esse conceito é melhor entendido depois que se aprende a trabalhar com classes, porém, pode ser extendido pra variáveis primitivas também. Um erro bastante comum cometidos por iniciantes em C++ é mostrado a seguir:

```
1  #include <iostream>
2
3  int* criarArray()
4  {
5    int a[10];
6    return a;
7  }
8
9  int main()
10 {
11   int* b = criarArray()
12
13   std::cin.get();
14 }
15
```

Código 2.20: Exemplo de um código que irá falhar

Sintaticamente falando, o código acima não apresenta algum erro. De fato, se tentarmos compilar, é provável que consigamos sem problemas, com exceção de alguns compiladores que irão nos mostrar o seguinte aviso: “warning C4172: returning address of local variable or temporary: a”. O que acontece nesse caso é que, na função criarArray, estamos declarando uma variável no stack da memória, e assim, ao término da execução da função, a prória variável a ser retornada será apagada. Isso irá fazer com que o ponteiro *b* aponte para um endereço de memória que não mais contêm dados ou está reservado, e sim para um endereço livre e que possivelmente poderá ser ocupado por outra variável, acarretando em *memory leackage*.

Ou seja, embora alocar uma variável ou objeto no *stack* seja mais rápido, na maioria das vezes teremos que utilizar uma alocação dinâmica (*heap*) quando: a variável a ser alocada consumir muito espaço na memória ou quando essa variável necessitar existir em escopos diferentes de onde foi declarada. Apenas ponteiros podem ser alocados no *heap*, e para isso utilizamos o comando *new* da seguinte maneira:

```
1  int* criarArray()
2  {
3    int* a = new int[10];
4    return a;
5  }
6
```

Código 2.21: Alocação no heap

2.10 Overload

Overload é o termo usado quando queremos atribuir uma outra função a um operador ou realizar outra tarefa em uma função a depender dos argumentos fornecidos. Algumas linguagens não permitem esse tipo de manipulação, outras permitem parcialmente, e em C++ o programador é livre para usar como bem entender. Basicamente é utilizado o conceito de *overload* em operadores e em funções que, de alguma forma, podem executar tarefas diferentes a depender dos operandos ou parâmetros associados.

Vejamos um exemplo simples de overload em uma função inicialmente utilizada para imprimirmos uma variável do tipo double no terminal:

```
1  #include <iostream>
2
3  void print(const double& valor)
4  {
5    std::cout.precision(17);
6    std::cout << valor << std::endl;
7  }
8
9  int main()
10 {
11   double numero = 2.3333333334444445;
12   print(numero);
13   std::cin.get();
14 }
15
```

Código 2.22: Função que imprime double

Nada de novo até o momento, mas e se agora quisermos utilizar uma função para imprimir uma variável do tipo *int* outra do tipo *char*? Devemos supostamente mudar o nome da função atual para *printDouble* e definir outras funçõe com os nomes *printInt* e *printChar*? Nesses casos podemos utilizar simplesmente um *overload* na função print para que ela execute uma outra tarefa caso receba um argumento do tipo *int* ou *char*:

```
1  #include <iostream>
2
3  void print(const double& valor)
```

```
4   {
5     std::cout.precision(17);
6     std::cout << valor << std::endl;
7   }
8
9   void print(const int& valor)
10  {
11    std::cout << valor << std::endl;
12  }
13
14  void print(const char* texto)
15  {
16  std::cout << texto << std::endl;
17  }
18
19  int main()
20  {
21    double numero = 2.3333333334444445;
22    print(numero);
23
24    double numero2 = 1236;
25    print(numero2);
26
27    const char* msg = "Hello World!";
28    print(msg);
29    std::cin.get();
30  }
31
```

Código 2.23: Overload em função

O mesmo conceito extende-se para os operadores. Como dito anteriormente, em C++ existem diversos operadores, como por exemplo +, -, <<, >>, [ ], entre outros, sendo que uma lista completa dos operadores pode ser encontrada aqui. Cada um deles possui funções pré-definidas a depender de onde será usado e *overload* nesse caso é simplesmente adicionar outra função para um deles que não a pré-definida. É bastante utilizado, por exemplo, para definir operações com structs e classes que, até então, não existiam. Digamos que queremos representar um vetor no espaço tridimensional por meio de uma struct, com componentes x, y e z. Como fazemos para somar esse vetor com outro vetor? Vejamos:

```
1   #include <iostream>
2
3   struct Vetor3D
4   {
5     double x_;
6     double y_;
7     double z_;
8
9     Vetor3D()
10    {
11      x_ = 0.0;
12      y_ = 0.0;
13      z_ = 0.0;
14    }
15
16    Vetor3D(const double& x, const double& y, const double& z)
17    {
```

```
18      x_ = x;
19      y_ = y;
20      z_ = z;
21    }
22
23    Vetor3D somar(const Vetor3D& vec)
24    {
25      return Vetor3D(x_ + vec.x_, y_ + vec.y_, z_ + vec.z_);
26    }
27
28    Vetor3D operator+(const Vetor3D& vec)
29    {
30      return somar(vec);
31    }
32
33  };
34
35  int main()
36  {
37  Vetor3D v1(4.0, 8.0, 12.0);
38  Vetor3D v2(2.0, 4.0, 6.0);
39  Vetor3D v3 = v1 + v2;
40
41  std::cout << v3.x_ << ", " << v3.y_ << ", " << v3.z_ << std::endl;
42
43  std::cin.get();
44  }
45
```

Código 2.24: Overload em operador

2.11 Templates

Template é um assunto bastante avançado em C++ e o intuito de apresentar tal assunto nesse curso não é aprendermos a usar todas as suas funcionalidades, mas sim saber para o que serve e onde ele está presente. De uma forma simples, os *templates* permitem fazer com que o compilador escreva o código para nós baseado nas diretrizes que damos a ele. Isso implica que, na verdade, todo o "código"implementado dentro de um *template* não é entendido pelo compilador como um código de verdade, mas sim uma diretriz que, quando invocamos aquele *template*, o próprio compilador irá escrever esse código baseado nessas informações. Um exemplo bastante simples para entendermos como ele funciona é retormarmos o caso visto na seção 2.10 da função que recebe diferentes tipos de argumentos e os imprime no terminal. Vimos anteriormente que é possível realizar *overload* na função para que a depender do tipo da variável a ser impressa, ela execute uma tarefa diferente, mas a desvantagem desse processo é a duplicação de código toda vez que queremos definir uma nova tarefa.

Podemos usar o template nessa situação, fazendo com que o compilador escreva uma função para nós a depender do tipo de argumento fornecido:

```
1  #include <iostream>
2
3  template<typename T>
4  void print(T valor)
5  {
```

```
6     std::cout << valor << std::endl;
7   }
8
9   int main()
10  {
11    print(5);
12    print(5.5f);
13    print("Hello World!");
14    print(true);
15
16    std::cin.get();
17  }
18
```

Código 2.25: Overload em operador

2.12 Bibliotecas

Para nossa felicidade, o C++ conta com diversas bibliotecas oficiais ou externas que tem como finalidade tornar mais fácil a programação e oferecer recursos extras. Até o momento trabalhamos com tipos primitivos de variáveis e também aprendemos como criar novos tipos.

2.12.1 Standard

A biblioca *standard* (*std*) é uma biblioteca que já está disponível em qualquer aplicação escrita em C++ e, embora não pareça, usamos ela ao longo do curso. Ao incluirmos o *headerfile iostream* no nosso código para usarmos a função std::cout, estávamos na verdade informando ao compilador para buscar as informações da biblioteca *std* referentes às operações de *input/output stream*. Note que ao utilizarmos qualquer coisa referente a essa biblioteca, escrevemos o nome da biblioteca, em seguida o operador duplo dois pontos (::) e o nome da variável ou função. Em C++ o operador :: é usado para indicar *namespace*, ou seja, a biblioteca qual contêm a definição daquilo que queremos usar. Nessa biblioteca encontramos suporte para diversos tipos de dados e operações, e a seguir veremos os mais usuais.

Fstream

Nesse arquivo encontram-se definidos alguns tipos de variáveis e funções que nos ajudam a realizar leituras e escritas em *streams* que não o terminal (std::cin e std::cout). Para isso, criamos uma variável do tipo *std::ifstream*, caso deseja-se um arquivo de *input* (leitura), ou *std::ofstream* para arquivos de *output* (escrita):

```
1   #include <iostream>
2   #include <fstream>
3   #include <string>
4
5   int main()
6   {
7     int a, b, c;
8     std::ifstream file("entrada.txt");
9     std::string line;
10    std::getline(file, line);
```

```
11      file >> a >> b >> c;
12      file.close();
13
14      std::ofstream file2("saida.txt");
15      file2 << a << ", " << b << ", " << c << std::endl;
16      file2.close();
17
18      std::cin.get();
19  }
20
```

String

O tipo de variável *string* nos permite salvar uma sucessão de caracteres e a manipular esse arranjo de uma forma mais fácil. Vimos anteriormente que para salvarmos esse tipo de dado criávamos um *array* do tipo *char*, então qual seria a vantagem de usar *string*? A vantagem é que as strings nos permitem executar diversas manipulações de texto mais facilmente, como cortar um pedaço, saber a quantidade de caracteres, espelhar, substituirum caractere específico, acrescentar outros caractéres, etc. Isso ocorre pois existem métodos que operam sobre as *strings* permitindo tais manipulações, algo que deveria ser programado para tornar-se possível utilizando *char*. Vejamos algumas funcionalidades:

```
1   #include <iostream>
2   #include <string>
3
4   int main()
5   {
6     std::string texto = "Hello World!";
7     std::string texto2 = "Bye World!";
8
9     std::string texto3 = texto + " " + texto2;
10    std::cout << texto3 << std::endl;
11
12    texto2.erase(3);
13    std::cout << texto2 << std::endl;
14
15    size_t position = texto.find("World");
16    std::cout << position << std::endl;
17
18    std::cin.get();
19  }
20
```

Código 2.26: Exemplo com string

Vector

Vector é um container utilizado para guardar objetos e variáveis quaisquer. Por ser um cointainer, podemos ir adicionando ou removendo objetos sempre que quisermos, o que é bastante vantajoso por não necessitar especificarmos seu tamanho a priói. Como nas strings, temos diversos métodos que operam sobre o std::vector, abaixo vemos os mais usuais também:

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

```
3
4  int main()
5  {
6    std::vector<double> vec;
7    std::cout << vec.size() << std::endl;
8
9    vec.push_back(2.0);
10   std::cout << vec.size() << std::endl;
11
12   vec.push_back(7.55);
13   std::cout << vec.size() << std::endl;
14
15   std::cout << std::fixed << vec[0] << ", " << vec[1] << std::endl;
16
17   vec.pop_back();
18   std::cout << vec.size() << std::endl;
19
20   std::cin.get();
21 }
22
```

Código 2.27: Exemplo com vector

Pair

Pair é uma estrutura que nos permite agrupar duas variáveis de tipos diferentes, formando um "par". A princípio não parece algo muito útil, mas pensemos na seguinte situação: precisamos implementar uma função que retorne duas variáveis de tipos diferentes, como fazer?

```
1  #include <iostream>
2  #include <utility>
3
4  std::pair<int, double> exemplo()
5  {
6    return std::make_pair(10, 30.4736);
7  }
8
9  int main()
10 {
11   std::pair<int, double> ex = exemplo();
12   std::cout << ex.first << ", " << ex.second << std::endl;
13   std::cin.get();
14 }
15
```

Código 2.28: Exemplo com pair

Map

Este tipo de container nos permite inserir objetos e "nomea-los"como quisermos. Aos usuários da linguagem Python, é exatamente igual à estrutura de dados chamada dicionário. Pensemos na estrutura std::vector por exemplo, ao inserirmos um objeto nesse tipo de estrutura, podemos acessa-lo através do operador [ ] e informando o índice do objeto, ou seja, um número inteiro. No std::map podemos, então, associar dois tipos de objetos, e então acessar um deles por meio do seu "nome":

```
#include <iostream>
#include <map>
#include <string>

int main()
{
  std::map<std::string, int> vertices;
  vertices["reta"] = 2;
  vertices["triangulo"] = 3;
  vertices["quadrado"] = 4;

  std::cout << vertices["reta"] << ", " << vertices["triangulo"] << ", " <<
  vertices["quadrado"] << "\n";
  std::cin.get();
}

```

Código 2.29: Exemplo com map

2.12.2 Boost

Boost é uma biblioteca numérica de acesso livre, que pode ser baixada aqui. Utilizamos ela basicamente para realizar operações matemáticas relacionadas a álgebra linear, mas ela tem uma gamma de aplicações bem maior, podendo ser utilizada para praticamente tudo. Como dito anteriormente, o C++ não disponibiliza uma biblioteca nativa para operações matemáticas, tanto é que seus *arrays* (unidimensionais e bidimensionais) e *vectors* não possuem funções matemáticas. Assim, o boost nos fornece variáveis tais como vetores e matrizes com funções matemáticas que operam sobre eles, como por exemplo cálculo de norma, produto interno, multiplicação de matrizes, número de condição, etc.

Capítulo 3

Orientação ao objeto

A seguir veremos alguns conceitos de programação orientada ao objeto, afinal, é pra isso que C++ serve.

3.1 Classes

As classes, exatamente como as *structs*, são comandos que permitem criar novos tipos de dados agrupando diversas variáveis. A única diferença entre elas é a visibilidade dos seus atributos e métodos, que por padrão, são privados na classe. Por essa razão e por questão de conveniência, utiliza-se sempre as classes na programação orientada ao objeto, seguindo o preceito de encapsulamento de dados. Embora a visibilidade de seus atributos e métodos seja privada, é possível alterar isso facilmente por meio dos modificadores de acesso *public*, *private* e *protected*, explicados abaixo:

- public: um membro público é acessível de qualquer parte do código fora da classe. É possível modificar e pegar dados de variáveis públicas sem que haja a necessidade de utilizar uma função para isso;
- private: um membro privado não pode ser acessado, visto ou modificado por nenhuma parte do código que não seja dentro do escopo da classe e por funções declaradas como friend. Esse é o tipo de acesso padrão para atributos e métodos de uma classe. Usualmente definimos os atributos de uma classe como sendo privados, e os métodos como sendo públicos;
- protected: um membro protegido de uma classe funciona igual a um membro privado, com a exceção de que esse dado pode ser acessado por classes filhas derivadas da classe mãe qual possui o dado.

Como nosso interesse é programar códigos que resolvam problemas de engenharia, vamos analisar uma classe simples que poderia ser usada pra representar um Nó de um elemento finito. Como se sabe, um Nó de elementos finitos possui basicamente um índice global, sua coordenada cartesiana inicial, seus deslocamentos e alguns outros atributos a depender da análise. Assim, segue um exemplo de como ficaria:

```
1  #include <iostream>
2  #include "boost\numeric\ublas\vector.hpp"
3
4  using namespace boost::numeric::ublas;
5
6  class Node
```

```
7   {
8     int index_;
9     bounded_vector<double, 2> coord_;
10    bounded_vector<double, 2> disp_;
11  };
12
13  int main()
14  {
15    Node n;
16    std::cin.get();
17  }
18
```

Código 3.1: Exemplo de classe

No exemplo acima criamos um novo tipo de dado para os Nós de uma malha de elementos finitos. Porém, se tentarmos modificar ou imprimir um membro dessa classe por exemplo, o código não irá compilar pois o acesso a esses membros é privado. O que comumente se faz então é implementar métodos públicos nessa classe para acessar ou modificar seus atributos.

```
1   #include <iostream>
2   #include "boost\numeric\ublas\vector.hpp"
3
4   using namespace boost::numeric::ublas;
5
6   class Node
7   {
8   public:
9     int getIndex()
10    {
11      return index_;
12    }
13
14    void setIndex(const int& index)
15    {
16      index_ = index;
17    }
18
19  private:
20    int index_;
21    bounded_vector<double, 2> coord_;
22    bounded_vector<double, 2> disp_;
23  };
24
25  int main()
26  {
27    Node n;
28    n.setIndex(15);
29    std::cout << n.getIndex() << "\n";
30    std::cin.get();
31
32    return 0;
33  }
34
```

Código 3.2: Implementação de métodos

Outra convenção em programação orientada ao objeto é criar métodos chamados *set* e *get* para modificar e acessar, respectivamente, os atributos privados das classes, exatamente como

fizemos para o *index*. Quando criamos de fato uma variável definida por uma classe, dizemos que instanciamos um objeto daquela classe. No nosso caso, na linha 27 instanciamos um objeto do tipo *Node*, e após isso alteramos o valor de seu *index*. Porém, ainda há o que melhorar nesse código, visto que seria trabalhoso ter que alterar manualmente todos os atributos de uma classe sempre que instanciamos um objeto.

Para isso existem os chamados construtores, métodos especiais de uma classe que são executados automaticamente sempre que instanciamos um objeto. Nesses construtores é comum implementarmos tarefa que inicialize os atributos de uma variável de acordo com o valor que especificarmos. Podemos implementar quantos construtores quisermos, sendo válido o conceito de *overload* em funções visto anteriormente. Vejamos como ficaria no nosso caso:

```
1  #include <iostream>
2  #include "boost\numeric\ublas\vector.hpp"
3
4  using namespace boost::numeric::ublas;
5
6  class Node
7  {
8  public:
9    Node(const int& index)
10   {
11     index_ = index;
12     coord_(0) = 0.0; coord_(1) = 0.0;
13     disp_(0) = 0.0; disp_(1) = 0.0;
14   }
15
16   Node(const int& index, const bounded_vector<double, 2>& coord)
17   {
18     index_ = index;
19     coord_ = coord;
20     disp_(0) = 0.0; disp_(1) = 0.0;
21   }
22
23   Node(const int& index, const bounded_vector<double, 2>& coord, const
     bounded_vector<double, 2>& disp)
24   {
25     index_ = index;
26     coord_ = coord;
27     disp_ = disp;
28   }
29
30   int getIndex()
31   {
32     return index_;
33   }
34
35   void setIndex(const int& index)
36   {
37     index_ = index;
38   }
39
40 private:
41   int index_;
42   bounded_vector<double, 2> coord_;
43   bounded_vector<double, 2> disp_;
44 };
```

```
45
46  int main()
47  {
48    Node n(0);
49    bounded_vector<double, 2> coord; coord(0) = 1.0; coord(1) = 2.0;
50    Node n2(1, coord);
51    bounded_vector<double, 2> disp; disp(0) = 5.0; disp(1) = 7.0;
52    Node n3(2, coord, disp);
53    std::cin.get();
54    return 0;
55  }
56
```

Código 3.3: Implementação de construtores

O mesmo pode ser feito para destrutores, ou seja, são métodos especiais que são automaticamente executados quando o objeto é destruído.

3.2 Herança

Herança é uma ferramenta muito poderosa em linguagens orientadas ao objeto porque nos permite evitar respetições desnecessárias no código e ainda permitir certas manipulações, como o polimorfismo, que será visto na próxima seção. Basicamente, herança é a possibilidade de uma classe "filha"herdar todos os atributos e métodos de uma classe "mãe". Vejamos o exemplo:

```
1  #include <iostream>
2  #include <string>
3
4  class Mae
5  {
6  public:
7    double x_;
8    int y_;
9    std::string nome_;
10
11   void print()
12   {
13     std::cout << "Classe mae!" << std::endl;
14   }
15 };
16
17 class Filha : public Mae
18 {
19 public:
20   float z_;
21
22   void print()
23   {
24     std::cout << "Classe filha!" << std::endl;
25   }
26 };
27
28 int main()
29 {
30   Filha exemplo;
31   exemplo.print();
```

```
32      exemplo.x_ = 1.0;
33      exemplo.y_ = 12;
34      exemplo.nome_ = "filha";
35      std::cin.get();
36      return 0;
37  }
38
```

Código 3.4: Herança

Observa-se pela linha 17 que para utilizarmos o conceito de harança, basta acrescentarmos *: acesso classe Mãe* à declaração da classe Filha. Esse acesso que especificamos define a visibilidade dos membros da classe Mãe quando acessamos pela classe Filha:

```
1   class A
2   {
3     public:
4     int x;
5     protected:
6     int y;
7     private:
8     int z;
9   };
10
11  class B : public A
12  {
13    // x e publico
14    // y e protegido
15    // z nao e acessivel de B
16  };
17
18  class C : protected A
19  {
20    // x e protegido
21    // y e protegido
22    // z nao e acessivel de C
23  };
24
25  class D : private A
26  {
27    // x e privado
28    // y e privado
29    // z nao e acessivel de D
30  };
31
```

Código 3.5: Visibilidade entre classes herdeiras

3.3 Polimorfismo

Poliformismo surge naturalmente do conceito de herança, e basicamente é a ideia de termos múltiplos tipos para um mesmo objeto. Se pensarmos que uma classe Filha herda todos os atributos e métodos de uma classe Mãe, faz sentido podermos usar um objeto do tipo Filha em quaisquer lugares do meu código onde, teoricamente, eu usuaria um objeto Mãe. De fato, se a classe Filha tem tudo o que a classe Mãe tem, e possivelmente mais, não há problema algum em usar um objeto Filha como se fosse um objeto Mãe. Vejamos como funciona na prática:

```
1  #include <iostream>
2
3  class A
4  {
5  public:
6    int x_, y_;
7    A() {}
8    A(const int& x, const int& y)
9    {
10     x_ = x;
11     y_ = y;
12   }
13 };
14
15 class B : public A
16 {
17 public:
18   int z_;
19   B() {}
20   B(const int& x, const int& y, const int& z)
21   {
22     x_ = x;
23     y_ = y;
24     z_ = z;
25   }
26 };
27
28 void printXY(A* a)
29 {
30   std::cout << a->x_ << ", " << a->y_ << std::endl;
31 }
32
33 int main()
34 {
35   A* objetoA = new A(1, 2);
36   B* objetoB = new B(4, 5, 6);
37
38   printXY(objetoB);
39
40   std::cin.get();
41   return 0;
42 }
43
```

Código 3.6: Polimorfismo

Declaramos duas classes A e B, sendo que B deriva de A, e definimos também uma função *printXY()* que recebe um ponteiro para um objeto do tipo A como argumento, e executa a impressão dos seus atributos x e y no terminal. O polimorfismo nesse caso é a garantia de poder passar um objeto do tipo B para essa função, uma vez que B é derivado de A, e assim, também possui atributos x e y. Na linha 39 chamamos a função *printXY()* e passamos como argumento um objeto do tipo B, e se compilarmos e executarmos o programa, veremos os valores 4 e 5 impressos na tela.

Funções virtuais

Uma última coisa que devemos nos atentar ao utilizar polimorfismo é com a chamada dos métodos nas classes derivadas. O código anterior é perfeitamente compilável e executável, entretanto, as coisas podem dar errado se adicionarmos, por exemplo, um método em cada classe para imprimir o tipo da classe, vejamos:

```
#include <iostream>

class A
{
public:
  void printTipo()
  {
    std::cout << "Classe do tipo A." << std::endl;
  }
};

class B : public A
{
public:
  void printTipo()
  {
    std::cout << "Classe do tipo B." << std::endl;
  }
};

void print(A* a)
{
  a->printTipo();
}

int main()
{
  A* objetoA = new A;
  B* objetoB = new B;

  print(objetoA);
  print(objetoB);

  std::cin.get();
  return 0;
}


```

Código 3.7: Exemplo de um problema com polimorfismo

Ao utilizarmos o *objetoB* como argumento da função *print()* esperávamos que o texto impresso no terminal fosse "Classe do tipo B", afinal, o argumento nesse caso é um objeto da classe B. Entretanto, não é o que acontece. Isso ocorre pois o compilador sempre irá entender que, ao utilizarmos polimorfismo em situações em que métodos são chamados, esses métodos são referentes a classe mãe. Quando não queremos que isso aconteça, como é o nosso caso, devemos explicitar ao compilador que o método a ser utilizado será o definido na classe filha. Fazemos isso transformando o método da classe mãe em uma função *virtual*. É de boa prática também incluir o termo *override* no método que está sobrescrevendo o anterior para prevenir bugs. Finalmente fica assim:

```
1  #include <iostream>
2
3  class A
4  {
5  public:
6    virtual void printTipo()
7    {
8      std::cout << "Classe do tipo A." << std::endl;
9    }
10 };
11
12 class B : public A
13 {
14 public:
15   void printTipo() override
16   {
17     std::cout << "Classe do tipo B." << std::endl;
18   }
19 };
20
21 void print(A* a)
22 {
23   a->printTipo();
24 }
25
26 int main()
27 {
28   A* objetoA = new A;
29   B* objetoB = new B;
30
31   print(objetoA);
32   print(objetoB);
33
34   std::cin.get();
35   return 0;
36 }
37
38
```

Código 3.8: Função virtual

Na linha 6 acrescentamos o termo *virtual* antes do tipo de retorno da função que será substituída, e na linha 15, a palavra *override* após a declaração da função que irá sobrescrever.