2. ELEMENTOS DE C++

Na seção anterior foi apresentado um exemplo simples de um programa em C++ para mostrar alguns dos seus principais elementos. Nesta seção a estruturação de um programa em C++ será mais detalhada. Também serão apresentados exemplos de programas simples que trabalham com herança e polimorfismo de função. A partir de agora, então, alguns dos principais elementos de C++ são destacados.

- **Comentários e Arquivos Fonte**: Descritos na seção anterior.

- **Arquivos *Header* e Diretivas *#include***: Cada implementação de C++ vem com uma biblioteca de funções predefinidas, operadores e outras entidades. Programadores são estimulados a usar estas entidades pré-definidas, mas precisam primeiro declará-las em cada arquivo fonte no qual são utilizadas. Para evitar que os programadores tenham que memorizar as declarações necessárias e escrevê-las repetidamente, são fornecidos vários arquivos *header*, cada um contendo as declarações para uma certa parte da biblioteca. Um arquivo *header* é inserido em um programa através da diretiva *#include*, a qual o pré-processador troca pelo conteúdo do arquivo *header*. No exemplo anterior, o programa "*Hello, World!*" usa as facilidades da biblioteca para o fluxo de saída, onde as declarações necessárias estão contidas no arquivo *header iostream.h*, que o programa inclui com a diretiva *#include* <*iostream*>.

- **Funções e a *main()***: Os blocos básicos de construção dos programas C++ são funções, que correspondem a funções, procedimentos e subrotinas em outras linguagens. Cada função implementa um conjunto de procedimentos logicamente relacionados para execução de uma operação bem definida. Uma função é chamada ou invocada sempre que a operação é necessária. Funções podem ser definidas no programa ou podem ser funções pré-definidas da biblioteca. Quando as funções de um biblioteca são utilizadas, os arquivos *header* contendo as declarações necessárias devem ser incluídos. As funções recebem uma lista de parâmetros, que pode ser vazia. Neste caso, parênteses são associados com funções e sempre aparecem nas declarações, definições e chamadas de funções. Quando um nome de função aparece no código, é convenção do C e do C++ que um par de parênteses seja colocado após o nome da função para indicar que o nome diz respeito a uma função. Se não tiver nada entre os parênteses significa que a função não possui argumentos. Funções também retornam apenas um ou nenhum valor (quando o retorno é do tipo *void*). Como já comentado, quando um programa é executado a primeira função que ele chama é a *main()*.

- ***Input/Output***: No C++, a entrada é lida de, e a saída é escrita em *streams*, sendo que o tipo da variável determina o tipo de valor de entrada e/ou saída. Quando *iostream.h* é incluída em um programa, vários *streams* padrão são definidos automaticamente. O *stream* "*cin*" é usado para entrada, que normalmente consiste em uma leitura do teclado. O *stream* "*cout*" é usado para saída, que normalmente é enviada para o monitor. O operador de inserção "<<" insere dados em um *stream*, e o operador de extração ">>" retira dados de um *stream* e armazena-os em variáveis. Quando um *string* (série de caracteres que aparecem entre aspas) é inserido em *cout*, o seu conteúdo é imprimido. Se *cin* não for explicitamente redirecionado, a entrada será feita através do teclado. C++ usa seqüências *escape* para representar caracteres que não são representados por símbolos tradicionais como a, b e c. Uma seqüência *escape* consiste no caracter \ seguido de uma letra ou número. Como já comentado anteriormente, toda seqüência *escape* representa um único caracter. Algumas seqüências comuns são: \n (nova linha), \a (alerta), \t (tab) e \" (insere aspas em um *string*).

- **Identificadores**: Um identificador consiste em um conjunto de letras, dígitos ou _, sendo que o primeiro caracter deve, obrigatoriamente, ser uma letra (incluindo _). Um identificador não pode ser igual a uma palavra reservada. C++ diferencia letras maiúsculas de minúsculas, portanto "nome", "Nome" e "NOME" são três identificadores diferentes. Deve-se evitar a criação de identificadores que comecem ou terminem com _,

a menos que eles já sejam definidos pela implementação de C++, pois este usa tais identificadores para evitar conflitos com identificadores criados pelo programador.

- **Valores, Tipos e Constantes**: Como a maioria das outras linguagens de programação, C++ classifica valores de dados em tipos, de acordo como eles são armazenados na memória e quais operações podem ser executadas por eles. Tipos cujos valores representam números podem ser chamados de tipos aritméticos, que são divididos em tipos inteiros (*char*, *short*, *int*, *long*, que podem ser *signed* ou *unsigned*) e tipos de ponto flutuante (*float*, *double*, *long double*) [3]. Tipos *booleanos* (*bool*) podem receber apenas *true* ou *false*. Constantes, que são usadas para fazer com que não se altere o valor de uma variável, podem ser inteiras, escritas como em aritmética ordinária, ou caracteres, representados pelos seus códigos numéricos. Neste caso, o modificador *const* é usado para atribuir um valor inicial para uma variável que não poderá ser alterado pelo programa. Qualquer atribuição futura para variável do tipo *const* resultará em erro de compilação. O C++ também estende o uso de *const* para incluir classes e funções membros. Na definição de uma classe em C++, utiliza-se *const* seguido da declaração de uma função membro, que é impedida de modificar qualquer dado na classe.

- **Variáveis enumeradas**: O tipo *enum* foi elaborado para que o programador possa tornar mais legível seu código. Com ele, por exemplo, pode-se escrever laços de uma forma mais próxima de como o problema é abstraído. Na declaração de tipo enumerado coloca-se o nome do tipo, os valores que uma variável deste tipo receberá e os valores com que o programa tratará essas variáveis. Em outras palavras, enumerações permitem definir tipos inteiros cujos valores são representados por identificadores. Por exemplo [3, 9]:

```
enum dia_semana {
      domingo = 1,        // tratará como 1
      segunda,            // tratará como 2
      terça,              // tratará como 3
      quarta,             // tratará como 4
      quinta,             // tratará como 5
      sexta,              // tratará como 6
      sabado              // tratará como 7
};
```

  Em C++ o nome da enumeração é tratado como um tipo conhecido a partir da sua definição. Assim, para declarar-se variáveis deste tipo deve-se usar seu rótulo específico, como por exemplo:

```
dia_semana dia;
```

- **Definição e declaração de estruturas**: A definição de variáveis estruturadas em C/C++ é obtida através do uso da palavra reservada *struct*, seguida de um rótulo especificando o nome da estrutura.

```
struct Funcionario {
      char nome [50];
      char endereco [80];
      float salario;
};
```

  Uma vez definida a estrutura, pode-se definir variáveis para ela. Por exemplo,

```
Funcionario  empresa [40];          // 40 funcionários em uma empresa
```

  é um vetor, onde em cada posição é guardado um nome, endereço e telefone. Para acessar um membro de uma variável estruturada, deve-se usar o nome da variável, um ponto e o nome do membro. Seguindo o exemplo anterior, o acesso aos membros é feito da seguinte maneira:

```
strcpy (empresa [3].nome, "Joao Alfredo");
strcpy (empresa [3].endereco, "Rua A, numero 10");
empresa [3].salario = 530.00;
```

  É importante ressaltar que o total de espaço ocupado por uma estrutura é equivalente à soma em *bytes* dos tipos que a compõem [9].

- **Classes e Objetos**: Na programação orientada a objetos, um programa executando em um computador é feito de componentes que interagem, chamados objetos. Conforme o programa é executado, os objetos interagem enviando mensagens uns para os outros e recebendo respostas. Como já comentado anteriormente, uma classe contém uma descrição completa de um tipo de objeto, e um objeto é uma instância de uma classe. Em C++ as classes são tipos de dados definidos pelo usuário, e são utilizadas da

mesma maneira que um tipo *int* ou *double*. As instâncias da classes em C++ são aqueles objetos (variáveis) que são declarados com a classe assim como seus tipos de dados. Uma classe é declarada usando a palavra reservada *class*. A menos que seja especificado o contrário, em C++, todos os membros, por *default*, são considerados *private*[3]. Uma classe possui um nome que é colocado após a palavra reservada *class*, isto é, uma palavra reservada no interior do escopo da classe. Uma classe em C++ é delimitada por "{" e "};". Classes podem conter uma lista de membros, que pode declarar dados, funções, classes, enumerações, campos de *bits*, *friends* (função que não é um membro da classe, mas tem permissão para usar os nomes de membros privados e protegidos da classe), e nomes de tipos. Os membros de uma classe podem ser declarados como *private*, *public* ou *protected*. Uma função membro é declarada da seguinte maneira: <*tipo*> <*nome_da_classe*>::<*nome_da_função*> (<*parâmetros*>), onde tipo identifica, por exemplo, se função retorna um valor inteiro, *float* ou não retorna nada (*void*). Torna-se importante comentar que classes podem ser declaradas com *struct* ou *union*. Neste caso seus membros e classes básicas são públicos por *default* [2, 3].

- **Herança**: A herança permite que uma nova classe seja definida através da extensão ou modificação de uma ou mais classes existentes. A nova classe é chamada de subclasse e a classe existente é chamada de superclasse. Uma subclasse pode servir como superclasse, e assim por diante, permitindo a criação de uma hierarquia de classes relacionadas através da herança. Fala-se em herança simples quando uma subclasse tem apenas uma superclasse, e herança múltipla quando uma subclasse possui várias superclasses. A herança múltipla permite combinar as propriedades das classes existentes. A sintaxe para declaração de uma subclasse é: *class* <*nome_classe_derivada*> : <*especif_acesso*> <*nome_classe_base*>, onde <*especif_acesso*> pode ser *private*, *public* ou *protected*. Torna-se importante atentar para o fato de que pode haver várias <*especif_acesso*> <*nome_classe_base*>, no caso de herança múltipla [3].

  A seguir é apresentado um programa em C++ que exemplifica a utilização de herança [10].

```
#include <iostream>
using namespace std;

enum tipo {carro, van, perua};


class Veiculo_Estrada
{
      int rodas;
      int passageiros;
  public:
      void define_rodas(int num);
      int pega_rodas();
      void define_passa(int num);
      int pega_passa();
};

class Caminhao : public Veiculo_Estrada
{
      int carga;
  public:
      void define_carga(int tamanho);
      int pega_carga();
      void mostrar();
};

class Automovel : public Veiculo_Estrada
{
      enum tipo carro_tipo;
  public:
      void define_tipo(enum tipo t);
      enum tipo pega_tipo();
      void mostrar();
};


// Funções da superclasse ("Veiculo_Estrada")
void Veiculo_Estrada::define_rodas(int num) {
  rodas = num;
}
```

```
int Veiculo_Estrada::pega_rodas() {
  return rodas;
}
void Veiculo_Estrada::define_passa(int num) {
  passageiros = num;
}
int Veiculo_Estrada::pega_passa() {
  return passageiros;
}


// Funções da classe "Caminhao"
void Caminhao::define_carga(int num) {
  carga = num;
}
int Caminhao::pega_carga() {
  return carga;
}
void Caminhao::mostrar() {
  cout << "Rodas: " << pega_rodas() << "\n";
  cout << "Passageiros: " << pega_passa() << "\n";
  cout << "Capacidade da carga em metros cubicos: " << carga << "\n \n";
}


// Funções da classe "Automovel"
void Automovel::define_tipo(enum tipo t) {
  carro_tipo = t;
}
enum tipo Automovel::pega_tipo() {
  return carro_tipo;
}
void Automovel::mostrar() {
  cout << "Rodas: " << pega_rodas() << "\n";
  cout << "Passageiros: " << pega_passa() << "\n";
  cout << "Tipo: ";
  switch(pega_tipo()) {
       case van: cout << "Van\n \n";
                 break;
       case carro: cout << "Carro\n \n";
                 break;
       case perua: cout << "Perua\n \n";
  }
}

// Função Principal
main()
{
  Caminhao t1, t2;
  Automovel c;

  t1.define_rodas(18);
  t1.define_passa(2);
  t1.define_carga(3200);

  t2.define_rodas(6);
  t2.define_passa(3);
  t2.define_carga(1200);

  t1.mostrar();
  t2.mostrar();

  c.define_rodas(4);
  c.define_passa(6);
  c.define_tipo(van);
  c.mostrar();
}
```

Obs.: O tipo *void* especifica um conjunto vazio de valores; ele é usado como o tipo de retorno para funções que não retornam um valor.

- **Polimorfismo**: Literalmente, polimorfismo significa a habilidade de ter “muitas formas”, ou seja, na programação orientada a objetos é caracterizado pela expressão "uma interface, diversos métodos". O

polimorfismo ocorre quando os objetos com estruturas internas muito diferentes podem compartilhar a mesma *interface* externa e podem ser usados da mesma maneira. Em outras palavras, polimorfismo refere-se a situação na qual objetos pertencendo a classes diferentes podem responder a mesma mensagem, usualmente de diferentes maneiras. Por exemplo, supondo uma superclasse "Figura" usada para armazenar as dimensões de vários objetos bidimensionais e para calcular suas áreas, e três classes derivadas "Triangulo", "Quadrado" e "Circulo", cujos objetos representam as figuras geométricas correspondentes. Os objetos destas classes podem entender um método "Mostra_Area" da superclasse, que faz com que um objeto exiba a área da figura geométrica correspondente na tela. Entretanto, a resposta para o método "Mostra_Area" é claramente diferente para os objetos "Triangulo", "Quadrado" e "Circulo". Neste caso, pode-se usar a mesma interface para as classes derivadas, embora estas forneçam seus próprios métodos para calcular a área de seus objetos. O polimorfismo ajuda a reduzir a complexidade, permitindo que a mesma *interface* seja usada para especificar uma classe geral de ações. É trabalho do compilador selecionar a ação específica (isto é, o método) que se aplica a cada situação. O programador não precisa fazer essa seleção manualmente, basta apenas se lembrar da interface geral e utilizá-la.

Em C++, o polimorfismo pode ser implementado através de funções virtuais, sendo que os únicos tipos que suportam polimorfismo são *class* e *struct*. A palavra reservada virtual permite que classes derivadas forneçam versões diferentes de uma função da superclasse. Uma vez que uma função é declarada como virtual em uma superclasse, ela pode ser redefinida em qualquer subclasse, porém os protótipos das funções virtuais nas classes derivadas devem ser **idênticos** ao protótipo na superclasse. As funções virtuais são especiais porque quando uma é chamada por meio de um ponteiro da superclasse (ou referência) para um objeto de uma subclasse, o C++ determina qual função chamar durante a execução, com base no tipo de objeto apontado. Assim, quando objetos diferentes são apontados, diferentes versões da função virtual são executadas.

Uma função virtual é declarada como virtual dentro da superclasse precedendo-se sua declaração com a palavra-chave virtual. No entanto, quando uma função virtual é redefinida por uma subclasse, a palavra-chave virtual não precisa ser repetida. O próximo exemplo ilustra o conceito de polimorfismo [3, 10].

```
#include <iostream>
using namespace std;
class Base {
      public:
                  virtual void quem()  {              // especifica uma função virtual
                              cout << "Base\n";
                  }
};
class primeira_d : public Base {
      public:
                  void quem()   {                     // define quem() relativa a primeira_d
                              cout << "Primeira derivacao\n";
                  }
};
class segunda_d : public Base {
      public:
                  void quem()    {                    // define quem() relativo a segunda_d
                              cout << "Segunda derivacao\n";
                  }
};
int main() {
  Base base_obj;
  Base *p;
  primeira_d primeira_obj;
  segunda_d segunda_obj;
  p = &base_obj;
  p->quem();  // acessa quem de Base
  p = &primeira_obj;
  p->quem(); // acessa quem de primeira_d
  p = &segunda_obj;
  p->quem();  // acessa quem de segunda_d
  return 0;
}
```

O ponto-chave de usar funções virtuais para obter polimorfismo durante a execução é que necessita-se acessar essas funções por meio de um ponteiro ou referência da superclasse. Embora seja correto chamar uma função virtual exatamente como se chama qualquer outra função "normal" (aplicando o operador ponto

para um objeto), é somente quando uma função virtual é chamada por meio de um ponteiro de superclasse (ou referência) que o polimorfismo durante a execução é alcançado. O polimorfismo é essencial para a programação orientada a objetos, pois permite que uma classe genérica especifique as funções que serão comuns a qualquer derivativo dessa classe, ao mesmo tempo em que permite a uma subclasse especificar a implementação exata de alguma ou de todas aquelas funções. Em outras palavras, a superclasse determina a *interface* geral que qualquer objeto derivado desta classe terá, mas permite que a subclasse defina o método real.

Parte do segredo para aplicar o polimorfismo com sucesso é compreender que as subclasses e as superclasses formam uma hierarquia que se move da maior para a menor generalização (da base para a derivada). Assim, quando usada corretamente, a superclasse fornece todos os elementos que uma subclasse pode usar diretamente, mais a base para aquelas funções que a subclasse precisa implementar em si mesma [10].

- **Ponteiro *this***: Resumidamente, *this* consiste numa "auto-referência" a instância de uma classe. Quando uma instância de uma classe (objeto) é criada, ela tem sua própria cópia dos membros. Para efetivar a ligação função membro-objeto, o C++ possui um ponteiro implicitamente criado para cada função de classe, denominado *this*. Sua finalidade é apontar para o objeto que chamou a função, isto é, em uma função membro, *this* é um ponteiro para o objeto que chamou a função. A maior utilidade do ponteiro *this* é o fato de ele sempre informar qual objeto está sendo referenciado no momento. Torna-se importante ressaltar que pode-se usar *this* apenas dentro de uma função membro; o ponteiro não existe em qualquer outra posição [9]. O código a seguir exemplifica sua utilização.

```
class X   {
            int a;
    public:
      X(int a) {
                this->a = a;          // membro da classe pode ser referenciado por this
      }                               // (mais usado para manipular listas encadeadas)
};
```