Uma breve introdução a Progamação Orientada a Objetos com C++

Daniel Santana Rocha

Resumo

Nesta apostila pretendo introduzir em breves palavras os principais conceitos que rondam o paradigma de objeto orientação em progamação. Os codigos fontes de exemplos estão escritos em C++ com algumas versões em Java para ilustrar a diferença. No final da apostila.

Sumário

CAPíTULO 1

Introdução

1. Progamação como a arte da Abstração

Em minha opinião de progamador, e principalmente, de matemático acredito que a progamação, mais ainda, boa parte das áreas mais próximas da matemática abstrata giram em torno de um único conceito: **Abstração**. Mas, o que é abstração? A wikipedia descreveria abstração como

> *Abstração (do latim abstractio)[1] é uma operação intelectual que consiste em isolar, por exemplo num conceito, um elemento à exclusão de outros, do qual então se faz abstração.[2] Por exemplo, abstraindo uma bola de futebol de couro, por uma bola de futebol, retemos apenas a informação enxuta das propriedades e comportamentos da palavra.*

Palavras simples, retiradas de [**1**], porém o suficiente para começarmos a enteder este complexo, porém extremamente útil, conceito. Mas de ondem vem sua utilidade, podemos citar, por exemplo, na progamação, onde usamos o conceito com o objetivo de **isolar** os aspectos relevantes do progama. Mas como faze-lo, isto é, como abstrair se de conceitos menos relevante em certos momentos? Isto é uma discussão extremamente complexa que o escopo deste livro não almeja entrar. O que este livro objetiviza e apresentar a alguém com uma noção básica de progamação procedural uma maneira, dentre outras vale ressaltar, de atingir níveis de abstração conhecida como POO (Progamação Orientada a Objetos) ou do inglês OOP (Oriented Object Progamming.

2. Objetos

Correndo o risco de soar redundante, ressalta-se que POO é orientada a objetos, isto é, em linguagens de compução com este **paradigma** de progamação, o código se baseia em objetos e suas relações, muitas vezes hierárquicas, como veremos. Para sairmos do campo das ideias e irmos a um contexto u pouco mais concreto, em linguagens como C++ e Java permeia o conceito classico de **Classe**. No caso, nestas linguagens Classe é o Objeto.

3. Hierarquia

Nas linguagens de POO modernas, porém não em todas, junto ao conceito de Objetos temos um conceito de **Hierarquia** entre eles. Neste sentido, se declararmos um Objeto Animal por exemplo e um outro Objeto Gato, faz sentido que haja uma ordem natural pois, todo Gato é um Animal, porém a recíproca não é verdadeira. Por esta ideia simples de níveis de abstração, nasceu o conceito de extensão de clases, em C++ por exemplo, diriamos que a classe Gato estende a Classe Animal.

4. Porque C++?

Uma pergunta que o leitor pode indagar é porque foi escolhido C++ para uma linguagem Introdutória a POO? E não Java por exemplo? O escritor ressalta alguns pontos importantes sobre C++

(1) C++ diferencia objetos de suas referências e ponteiros. Isto quer dizer que ponteiros, presente em qualquer linguagem de progamação por ser algo built-in (hardware), são menos "escondidos"e acreditamos que manter o contato com esses conceitos, pelo menos nesta primeira fase do aprendizado é importante para melhor entedimento da progamação como um todo.
(2) C++ não possui um coletor de lixo e com isso o conceito de *memory leak* podera ser melhor trabalho e estudado nesta linguagem.
(3) Muitas outras linguages POO se baseiam em C++ e vice-versa, fazendo com que haja tremendas semelhanças entre elas facilitando a curva de aprendizado aos bons entededores de C++.

CAPíTULO 2

Progamando em C/C++: Conceitos Básicos

Antes de prosseguirmos para conceitos mais avançados (porém não mais complexos!) como Classes, hierarquia, polimorfismo entre outros e necessário uma base solida de progamação. Relembraremos estes conceitos usando codigos exemplos em C/C++.

1. Tipos Primitivos

Resumidamente, os tipos primitivos de C são char (8 bits), int (32 bits), float (32 bits), double (64 bits), void com possiveis modificadores como long (64 bits) e short (16 bits). Toda **estrutura** é uma composição em sequência destes tipos. Por exemplo, abaixo um código (em C) simples que declara um tipo primitivo int (inteiro) e imprime seu valor na tela:

```
#include <stdio.h>
int main(int argc,char* argv[]) {
        int x = 123;
        printf("%d",x);
        return 0;
}
```

[Código 1]

2. Escopos e Stack

2.1. Escopos e amarração de variável. No codigo 1 alguns elementos se destacam. Na primeira linha de código temos uma inclusão do arquivo stdio.h para usarmos a biblioteca padrão da GCC de entrada e saida de dados. Após isso temos a declaração da função main com alguns aspectos importantes, porém nesta seção iremos focar nos simbolos { e } que correspondem a abertura e fechamento de um escopo respectivamente. Primeiramente, o que é um escopo? Uma boa maneira de definir escopo é uma porção de código em um progama em que a amarração de variavéis é a mesma. Vamos usar alguns exemplos par explicar isso:

```
#include <stdio.h>
int main(int argc,char* argv[]) {

        {                               // Comeco do Primeiro Escopo
                int x = 123; // Declaracao de x no primeiro escopo
        }                               // Fim do Primeiro Escopo

        {                                       // Comeco do Segundo Escopo
                printf("%d",x); // Referencia a variavel x declarada em outro escopo.
                // ERRO de Compilacao
        }                                       /// Fim do Segundo Escopo
        return 0;
}
```

[Código 2.1].

Observamos que em 2.1 recebemos um erro de compilação, isso porque tentamos acessar uma variável que não está declada no escopo 2. Claro que existe uma definição dela no escopo 1, porém ao fim do escopo essa definição é destruída. Outro código-exemplo:

```
#include <stdio.h>
int main(int argc,char* argv[]) {

        {                               // Comeco do Primeiro Escopo
                int x = 123; // Declaracao de x no primeiro escopo
        }                               // Fim do Primeiro Escopo

        {                                       // Comeco do Segundo Escopo
                int x = 321; // Declaracao de x no segundo escopo
                printf("%d",x); // Referencia a variavel x.
        }                                       /// Fim do Segundo Escopo
        return 0;
}
```

[Código 2.1]

No código acima não recebemos um erro de compilção e conseguimos compilar. Mas qual é a saída deste progama? 123 ou 321. E a resposta é 321. A primeira vista podemos pensar que receberemos um erro de compilação devido a dupla

declaração de x. Mas não recebemos pois os dois x são x diferentes pois estão em escopo diferentes! E também assumem valores diferentes e a renferência no segundo escopo se refere ao x do segundo escopo. Mais um exemplo:

```
#include <stdio.h>
int main(int argc,char* argv[]) {

        int z = 555; // Declarando z no escopo mae
        {                                   // Comeco do Primeiro Escopo
                printf("%d",z); //Referencia a z
        }
        return 0;
}
```

[Código 2.2]

Nesse código não recebemos um erro de compilação e de fato, a saída do progama é 555. Isto pois, dentro de um escopo interno, ou escopo filho, declarado dentro de um escopo maior podemos acessar todos as variáveis declaradas no escopo anterir e assim sucessivamente. Outra observação importante que ao declararmos uma nova função o escopo é "zerado". E que comando como for() e while(), como notado pela posse dos simbolos , também são considerados novos escopos.

2.2. Organização Hierarquica de Escopos. Observe o código de exemplo abaixo:

```
#include <stdio.h>
int main(int argc,char* argv[]) {
        int x = 0;
        int entrada[20];
        printf("Entre 20 numeros:\n");

        int i = 0;
        while(i<20) {
                int x = 0;
                scanf("%d",&x);
                entrada[i] = x;
                i++;
        }

        for(i=0; i< 20; i++) {
                int j = 0;
                for(j = 0; j < 20; j++) {
                        if(entrada[j]>entrada[i]) {
                                int c = entrada[i];
                                entrada[i] = entrada[j];
                                entrada[j] = c;
                        }
                }
        }

        for(i = 0;i < 20; i++) {
                printf("%d ",entrada[i]);
        }
        return 0;
}
```

[Código 2.2] O código acima pede a entrada de 20 números inteiros pela cli e imprime esses números de volta na tela em ordem crescente. O progama foi deliberadamente complicado e mal-feito para fins didáticos da análise do escopo. Não focaremos no progama em si e sim na forma com que os escopos se organizam. Observe que temos o escopo principal, da função main, sempre o primeiro código a ser executado e então dele se deriva 3 novos escopos, um while e dois for, sendo que dentro do primeiro for, temos um novo for com um if dentro. Podemos organiza-lo em um grafo hierarquico da seguinte maneira

2.3. Stack Pointer e Base Pointer.

3. Tempo de Codificação, Tempo de Compilação e Tempo de Execução

4. Tempo de Amarramento de Funções (Linker)

5. Ponteiros

Ponteiros é um dos conceitos mais importante na progamação C, de fato, é um conceito fundamental em grande parte das linguagens de progamação dado que a maioria dos microprocessadores possui este conceito "hard-coded". Algumas linguagens de progamação, como Java, mantêm está distinção entre ponteiros e objetos menos clara ao progamador resolvendo ela em "tempo de compilação". A idéia de ponteiros e possibilitar que dois escopos, a priori não ligados hierarquicamente, se referenciem ao mesmo objeto permitindo leitura e acesso. Começaremos com um exemplo

```
#include <stdio.h>

void f(int* ptr) {
```

```
        *ptr = 3;
}

int main(int argc,char* argv[]) {
        int x = 10;
        f(&x);
        printf("%d",x);
}
```

[Código 5]

A priori, usando os conceitos já apresentados, o leitor pode se indagar qual será a saída deste progama. Esquecendo por um momento os caracteres $*$ e & a saída será 10, pois a atribuição ptr = 3 apenas altera um cópia de x que é criada ao passarmos x como argumento da função f. Porém, ao rodarmos esse progama obtemos que saída é surpreendentemente 3! O que ocorreu? Para explicamos esse fenômeno precisamos falar sobre

5.1. Os operadores $*$ e &. Até então apredemos uma maneira apenas de guardar objetos na memoria, a stack. Note então que todos os objetos declarados até agora estão contidos na stack e, por esta ser linear, todos os objetos possuem um início, fim e tamanho bem definidos. Além disso, para acessar esses valores podemos se utilizar do base pointer. Base Pointer é o que chamamos de registro e guarda em si um inteiro de tamanho igual a arquitetura do computador (32 bits ou 64 bits em geral). Então para acessar um certo objeto, base pointer assume um certo valor, este valor é o que chamamos de *ponteiro* associado a este objeto.

Na linguagem C, podemos utilizar o operador & para obter o ponteiro associado a um objeto. Por exemplo, se temos um inteiro x, &x é o ponteiro associado a x. Uma observação é cabida no momento: se estamos em um computador de arquitetura 64 bits, provavelmente o inteiro terá tamanho 32 bits e seu ponteiro associado terá o dobro do tamanho, 64 bits. Em geral, todo objeto (estando na Stack ou não!) possui um ponteiro associado que corresponde a que lugar da memoria ele é armazenado. Além deste operador possuimos sua inversa $*$ que a partir de um ponteiro recupera o valor associado a este ponteiro. Isto quer dizer que o código abaixo retorna o valor de x:

```
#include <stdio.h>

int main(int argc,char* argv[]) {
        int x = 10;
        int* y = &y;
        int z = *y;
        printf("%d",z);
}
```

Observe que para declararmos um ponteiro colocamos int*. Em geral para declaramos um ponteiro para um objeto do tipo T, escrevemos T*. A partir disto, podemos analizar melhor o código 5. Nota-se que f não toma como argumento x, e sim seu ponteiro associado isto quer dizer que a atribuição *ptr = 3 de fato acessa o mesmo x que no escopo principal pois o ponteiro possuem o mesmo valor!

6. Estruturas

Sumarizando, uma estrutura é uma forma conveniente de organizar informação. Mais detalhadamente, uma estrutura (em C) é um objeto que pode ser composto por mutiplos tipos primitivos organizados de maneira linear. Por exemplo, nó codigo abaixo declaramos uma class círculo (que corresponde as informações de um círculo no plano) e então instaciamos um tipo círculo e atribuimos seus valores

```
struct Circle {
        int posx;
        int posy;
        int radius;
};

int main(int argc, char* argv[]) {
        struct Circle c;
        c.posx = 1;
        c.posy = 1;
        c.radius = 10;
}
```

Uma observação interessante sobre estruturas e como elas se organizam na stack. Ao declaramos uma estrutura Circle, esta se posiciona no primeiro ponto livre da stack e ocupa os próximos 12 bytes (correspodendo aos 3 inteiros, cada um com 4 bytes que o compõe). Além disto, os elementos aparecem na ordem de crescimento na stack em relação a ordem de declaração, isto é, posx primeiro logo depois posy e por último radius.

7. Heap

CAPíTULO 3

Classes

Classe é um conceito, se não o mais, importante em qualquer linguagem orientada a objetos. Bem simplisticamente, uma classe é uma estrutura com metódos. Por exemplo, vamos declarar a mesma estrutura declarada na seção de estruturas com a notação de classes

```
class Circle {
public:
        int posx;
        int posy;
        int radius;
};

int main(int argc, char* argv[]) {
        Circle c;
        c.posx = 1;
        c.posy = 1;
        c.radius = 10;
}
```

Observe que nada, além de uma mudança tipográfica, distingue esse código do apresentado da declaração de estruturas. Porém,

Referências Bibliográficas

[1] Susanne K. Langer, *Feeling and Form: A Theory of Art Developed from Philosophy in a New Key*, Routledge & Kegan Paul [S.l.] p. 90.