3. CONSTRUTORES E DESTRUTORES

Em uma linguagem de programação qualquer, quando uma variável de um tipo existente, como um inteiro, é declarada, o compilador cria espaço para aquela variável. Além disso, a linguagem C permite que uma variável seja declarada e inicializada simultaneamente, evitando problemas de utilização de variáveis não inicializadas. Porém, quando uma variável de um tipo existente sai fora de escopo, o compilador faz com que o espaço para aquela variável seja liberado. Na verdade, o compilador "constrói" e "destrói" a variável.

A linguagem C++ é concebida para fazer tipos definidos pelo usuário (classes). Para permitir uma inicialização direta dos objetos, o compilador necessita de uma função para chamar quando a variável é criada, isto é, o compilador chama um **construtor**. Os construtores são muito úteis em tarefas ligadas à inicialização de classes. Nelas pode-se encontrar inicialização direta de membros, alocação dinâmica de memória, recuperação de dados em arquivos, etc.

Quando um objeto sai fora do seu escopo, um **destrutor** pode ser chamado. Embora a liberação de memória seja uma aplicação bastante freqüente para funções destrutoras, esta não é sua única utilidade. Uma segunda utilidade para funções destrutoras pode ser a finalização de dispositivos ou subsistemas que tenham sido ativados como classes. Por exemplo, ao terminar a impressão pode-se finalizar a impressora desativando eventuais modos de impressão ajustados pelo sistema. Outra utilidade para a função destrutora, às vezes associada ao gerenciamento dinâmico de memória, está na prática de gravar os dados ao se encerrar o escopo dos mesmos (ou antes de liberá-los da memória).

Se o programador não proporcionar construtores (pode haver mais de um) e um destrutor (só pode haver um) para uma classe, o compilador assume as ações mais simples. Em outras palavras, caso não seja definido um construtor e um destrutor para a classe, o compilador cria um construtor e um destrutor *default*, sem código e sem parâmetros, que são chamados, respectivamente, a cada declaração de instância, e cada vez que a instância sai fora do escopo.

Resumindo-se, pode-se dizer que:

- Construtor: função que é chamada sempre que é criado um objeto de uma classe - faz a inicialização;
- Destrutor: função que é chamada sempre que o escopo de duração do objeto de uma classe encerra-se - faz a "limpeza".

O construtor é uma função com o mesmo nome da classe e que não pode retornar nenhum valor. Exceto em casos não usuais, o construtor assume que o espaço foi alocado para todos os atributos na estrutura do objeto quando ele é chamado. O nome do destrutor é o da classe com um til (~) anexado no início. O destrutor nunca pega nenhum argumento; ele só é chamado pelo compilador e não pode ser chamado explicitamente pelo programador.

Como geralmente o desejado é fazer vários tipos de inicialização num objeto, o "destrutor padrão" (fazendo nada) é muitas vezes suficiente e não é necessário definir um destrutor. Entretanto, se um objeto inicializa algum *hardware* ou muda algum valor global, pode ser preciso desfazer o efeito do objeto quando este é destruído. Para isso precisa-se de um destrutor [1, 9].

O código abaixo exemplifica a utilização de construtores e destrutores [7].

```
// Criação da Classe
class vector {
        int sz;   // número de elementos
        int* v;   // ponteiro para inteiros
        public:
                vector(int);   // construtor
                ~vector();   // destrutor
};
```

```
// Construtor
vector::vector (int s) {
        if (s<=0) {
                cout << "Bad Vector Size";
                return;
        }
        sz = s;
        v = new int[s];   // aloca um vetor de inteiros
        cout << "\n Construtor";
}
// Destrutor
vector::~vector() {
        delete [] v;   // desaloca o vetor apontado por v
        cout << "\n Destrutor";
}

// Função Principal
void main()
{
        vector v(10);
}
```

Obs.: Já foram apresentadas duas opções para o "tempo de vida" de um objeto de dado: Um objeto de dado *static* existe durante toda a execução do programa; Um objeto de dado com classe de armazenamento "automática" é criado quando é declarado e destruído quando sai do escopo. Em contraste, os operadores ***new*** e ***delete*** dão ao programador liberdade para criar objetos de dados em qualquer momento e destruí-los quando eles não são mais necessários. O "tempo de vida" de tais objetos não é relacionado com a estrutura do programa. Um objeto de dado criado dentro de um bloco com *new* vai "existir" até ser explicitamente destruído com *delete*; assim, tal objeto de dado continua existindo depois que o processamento saiu do bloco onde ele foi criado e pode, então, ser manipulado por comandos fora do bloco.

O operador *new* pode ser usado para criar objetos de qualquer tipo de dado; *new* é aplicado ao tipo de objeto de dado a ser criado e retorna um ponteiro para o objeto de dado criado. Assim:

```
int* p = new int;
```

cria um novo objeto de dado inteiro e determina que "p" é um apontador para ele. O mesmo resultado pode ser obtido com:

```
int* p;
p  = new int;
```

Uma atribuição é feita da seguinte maneira:

```
*p  = 100;
```

Quando o objeto de dado não é mais necessário, ele é destruído para liberar o espaço de memória que ele ocupa para criação de novos objetos de dados. O objeto de dado, no caso do exemplo, apontado por "p" é destruído com o comando

```
delete p;
```

Se o valor de "p" é *null*, nenhuma ação é tomada [3].

É interessante comentar que **tanto construtores como destrutores não são herdados**. Entretanto, construtores *default* e construtores de cópia (usados quando deseja-se que o objeto declarado seja inicializado com os valores guardados em outro objeto existente; Ex: *Nome_Classe (Nome_Classe &obj);*) são gerados pelo compilador quando necessários. Tanto os construtores como os destrutores gerados são *public*. No caso dos destrutores, se uma superclasse ou um membro tem um destrutor e nenhum destrutor é declarado para sua subclasse, é gerado um destrutor *default*. Este chama os destrutores para as superclasses e membros da subclasse.

Alguns pontos importantes sobre destrutores que devem ser reforçados são: destrutores não possuem parâmetros e não possuem tipos de dados de retorno (nem mesmo *void*); são opcionais; são chamados automaticamente para desinicializarem a variável de classe; programas raramente chamam destrutores diretamente, isto é, se os objetos são globais, os destrutores são chamados de forma implícita quando seu escopo global deixa de existir.

Deve-se observar que a ordem que o processador segue ao chamar o destrutor implicitamente para cada objeto é inversa à ordem de chamada das funções construtoras, ou seja, o primeiro objeto a ser criado é o último a ser destruído. Também deve-se ressaltar que as classes derivadas não herdam os construtores e destrutores das superclasses. Portanto, devem ter suas próprias funções construtoras e destrutoras. Por outro lado, é importante saber como o construtor de uma subclasse se relaciona com um construtor da superclasse.

Quando uma instância da subclasse é criada, um construtor desta classe pode ser chamado. Esta chamada efetiva a relação preconcebida entre o construtor da base e o da derivada. Assim, na declaração de um objeto da subclasse um construtor da superclasse é acionado, mesmo quando não for desejável inicializar os atributos da superclasse. A relação explícita entre o construtor da subclasse e o construtor da superclasse é colocada na definição do construtor da subclasse. Para tal, deve-se usar a seguinte sintaxe:

Nome_Classe_Derivada (<parâmetros_classe_derivada>, <parâmetros_classe_base) :
Nome_Classe_Base (<parâmetros_classe_base>)
{
// definição da função
}

A ordem dos parâmetros, na realidade, não importa. Mas, deve-se tomar cuidado para não passar os valores errados na chamada do construtor da base. Torna-se importante ressaltar que independente da vontade do programador, o compilador faz um acesso ao construtor da superclasse. Quando a chamada não é explícita, o acesso ocorre no construtor *default* [2, 4, 9]. O exemplo a seguir exemplifica a relação entre os construtores da superclasse e das classes derivadas [4]:

```
# include <iostream>
using namespace std;
#define FALSE 0
#define TRUE  1
class animal  {
        private:
                char name[30];
        public:
                animal (const char *s);
                const char *getName (void) {
                        return name;
                }
};
animal::animal (const char *s) {
        strncpy (name, s, 29);
}
class mammal : public animal  {
        private:
                int offspring;
        public:
                mammal (const char *s, int nc);
                int numOffspring (void) {
                        return offspring;
                }
};
mammal::mammal (const char *s, int ncd) : animal (s) {
        offspring = ncd;
}
class bird : public animal  {
        private:
                int eggs;
                int nesting;
        public:
                bird (const char *s, int ne, int nests);
                int getEggs (void) {
                        return eggs;
                }
                const char *buildsNest() {
                        if (nesting)
                                return "TRUE";
                        else
                                return "FALSE";
                }
};
bird::bird (const char *s, int ne, int nests) : animal(s) {
```

```
        eggs = ne;
        nesting = nests;
}

void showMammal (mammal &m);
void showBird (bird &b);

main()
{
        mammal homoSapiens ("Homo Sapiens", 1);
        mammal gopher ("Gopher", 9);
        mammal armadillo ("Armadillo", 4);
        mammal houseMouse ("House Mouse", 12);
        bird woodDuck ("Wood Duck", 15, FALSE);
        bird sandhillCrane ("Sandhill Crane", 2, TRUE);
        bird loon ("Loon", 3, TRUE);
        cout << "\n \n Mammals: ";
        showMammal(homoSapiens);
        showMammal(gopher);
        showMammal(armadillo);
        showMammal(houseMouse);
        cout << "\n \n Birds: ";
        showBird (woodDuck);
        showBird (sandhillCrane);
        showBird (loon);
        return(0);
}

void showMammal (mammal &m) {
        cout << "\n Name ..... " << ( m.getName() );
        cout << "\n Avg offspring .. " << ( m.numOffspring() );
}

void showBird (bird &b) {
        cout << "\n Name ..... " << ( b.getName() );
        cout << "\n Avg no. eggs .. " << ( b.getEggs() );
        cout << "\n Builds a nest .. " << ( b.buildsNest() );
}
```

Também é interessante observar que:

- Os objetos de uma classe com construtores podem ser inicializado através de uma lista de parâmetros passada para o construtor, ou através do uso de um sinal de igual seguido de um valor. Por exemplo:

```
class X {
        int i;
    public:
        X();
        X(int x);
};
void main() {
    X one;              // construtor default é invocado
    X two(1);    // construtor X::X(int) é usado
    X three = 1;// construtor X::X(int) é usado
}
```

- O construtor pode atribuir valores para seus membros de duas maneiras:
  1. Pode aceitar os valores como parâmetros e atribuir para as variáveis dentro do corpo da função construtor, como mostra o exemplo a seguir.

```
class X {
        int a, b;
    public:
        X(int i, int j) {
            a = i; b = j
        }
};
```

  2. Uma lista de inicialização pode ser usada antes do corpo da função. Por exemplo:

```
class X {
        int a, b;
    public:
        X(int i, int j) : a(i), b(j) { }
};
```