7. PASSAGEM DE PARÂMETROS

É interessante ressaltar que a criação de novas classes de objetos é uma atividade primária da programação orientada a objetos, e a complexidade do programa é geralmente escondida nestes objetos. Sendo assim, escrever a função *main()* consiste em gerenciar objetos. Um tópico mais complexo sobre criar um tipo definido pelo usuário (ou classe de objetos) é a passagem de objetos para, e retorno de objetos de funções. Nesta seção é descrita a passagem de parâmetros, sendo que é dada especial atenção à construção de classes complexas que podem ser passadas e retornadas corretamente.

Uma função comunica-se com o "mundo externo" de duas maneiras: mudando os valores que são globais à função ou através de passagem e retorno de valores. A informação pode ser movida da/para função de duas maneiras:

- Uma cópia da variável inteira é feita;
- Somente o endereço da variável é transferido pelos limites.

O primeiro método é chamado **passagem por valor**. Já que é feita uma cópia do argumento (ou variável), esta pode ser usada e alterada dentro da função sem afetar a variável da qual ela foi gerada. Entretanto, quando uma função precisa ser capaz de alterar os valores das variáveis usadas como argumentos, os parâmetros precisam ser explicitamente declarados como tipos ponteiros. Assim, utiliza-se o segundo método que é chamado de **passagem por nome**, quando um ponteiro é passado, ou de **passagem por referência**, quando referências são usadas. Em ambos os casos um endereço é manipulado.

Os endereços são passados por duas razões. A primeira é simplesmente pela eficiência; sabendo-se que a variável será somente lida, então, geralmente é mais rápido passar o endereço quando se usam objetos maiores do que um ponteiro. Existe também um benefício adicional em se passar um endereço. No caso de objetos, passa-se o endereço para um objeto inicializado e o tamanho do endereço é sempre o mesmo, assim não há problema envolvendo cópia e inicialização.

A segunda razão para passar um endereço é manipular fisicamente o objeto ou variável para o qual o endereço aponta, isto é, pode-se alterar uma variável ou objeto que é externo a uma função. Normalmente, os resultados de uma chamada de função devem ser expressos pelo valor de retorno da função, mas existem situações em que isso não é satisfatório. Por exemplo, a função pode precisar alterar mais de um objeto. Se os endereços do objeto são passados, a função pode afetar diretamente qualquer número de objetos na função chamadora.

Na passagem por nome, são explicitamente declarados ponteiros e passados endereços. Já passar por referência é o mesmo que declarar um argumento para ser um ponteiro e, então, passar um endereço. Com referências, entretanto, o compilador faz todo o trabalho para o usuário: ele força o endereço a ser passado dentro de uma função, ainda que o usuário não pegue o endereço explicitamente. Dentro da função, pode-se manipular os itens como se fossem objetos. Isso é um grande avanço, uma vez que muitos erros quando se usam bibliotecas da linguagem C, podem ser atribuídos à confusão sobre como passar um endereço. Como o compilador da linguagem C++ manipula os detalhes, quando se passa argumentos por referência, muito poucos erros ocorrem. Também é muito mais eficiente passar um ponteiro do que copiar o objeto inteiro dentro da função chamada.

Entretanto, existem casos onde é preciso uma cópia de um objeto a ser criado dentro da janela da função. Pode-se necessitar modificar o objeto sem querer que o original seja tocado. Nesses casos, deve-se passar o objeto por valor. Se o compilador passa um *int* ou *float*, por exemplo, ele sabe o que fazer: uma cópia

do dado é empurrada na pilha antes que a função seja chamada. Quando se passa um tipo definido pelo usuário, entretanto, o compilador não tem idéia de como manipulá-lo.

No caso de uma classe "simples" (nenhum dos dados-membros são ponteiros) onde não foi definido um inicializador de cópia, o compilador recorre a uma cópia *bit* a *bit*, que copia todos os *bytes* da *struct* do objeto antigo na *struct* do novo objeto. Porém, se alguns dos membros da classe são ponteiros, uma "cópia mais complexa" é usualmente requerida para alocar memória livre e copiar os itens que são apontados [1].

O exemplo abaixo ilustra a passagem por valor e por referência:

```
void f  (int val, int& ref)
{
        val ++;
        ref++;
}

void g ()
{
        int i = 1,
            j = 1;
        f(i, j);
}
```

Neste caso, quando "f()" é chamada, "val++" incrementa uma cópia local do primeiro argumento, enquanto "ref++" incrementa o segundo argumento. Sendo assim, na função "g()", "j" é incrementado, mas "i" não, pois o primeiro é passado por referência e o segundo por valor.

Se um vetor é usado como argumento, um ponteiro para o seu elemento inicial é passado. Por exemplo:

```
:
int strlen(const char*);
void f()
{
        char v[] = "an array";
        int i = strlen(v);
        int j = strlen("Nicholas");
}
:
```

Vetores diferem de outros tipos nos quais um vetor não é (e não pode ser) passado por valor. O tamanho do vetor não é disponível para a função chamada [8].

Torna-se importante salientar que se uma expressão é especificada em uma declaração de argumentos, essa expressão é usada como um **argumento *default***. Todos os argumentos subseqüentes devem ter argumentos *default* fornecidos nessa ou em declarações prévias desta função. Um argumento *default* não pode ser redefinido por uma declaração posterior (ainda que para o mesmo valor). Porém, uma declaração pode acrescentar argumentos *default* não fornecidos em declarações anteriores.

Na declaração:

```
ponto (int = 3,  int = 4);
```

tem-se uma função que pode ser chamada com zero, um ou dois argumentos do tipo *int*. Ela pode ser chamada de qualquer dessas formas:

```
ponto (1, 2);    ponto (1);    ponto ();
```

As duas últimas chamadas são equivalentes a "ponto(1,4)" e "ponto(3,4)", respectivamente [2].

Ponteiros também podem ser argumentos de funções, apesar disto não ser muito usual em C++, uma vez que existem os parâmetros de referência. Assim sendo, para passar um argumento para uma função para uso direto, uma instrução deve passar um "endereço" desse argumento para um parâmetro do tipo ponteiro.

A seguir será apresentado um exemplo que mostra a diferença entre usar os parâmetros por referência e passar um ponteiro como argumento. Supondo que foi criada a seguinte estrutura para acompanhar as temperaturas baixas e altas:

```
struct temp
{
        float alta;
        float baixa;
}
```

Esta estrutura é usada em um programa que faz o acompanhamento das leituras altas e baixas de um termômetro de algum modo ligado ao computador. Uma função-chave neste programa lê a temperatura corrente e modifica o campo apropriado em uma estrutura do tipo "temp" passada por referência para uma função. O código fica da seguinte maneira:

```
:
temp atemp;
resgistratemp(atemp);
:
void registratemp (temp &t)
{
        float atual;

        pegatempatual(atual);

        if (atual > t.alta)
                t.alta = atual;
        else if (atual < t.baixa)
                t.baixa = atual;
}
:
```

Neste exemplo, como o parâmetro de "registratemp" é passado por referência, a função opera diretamente na variável "atemp". Se o parâmetro não tivesse sido declarado com &, a função operaria apenas sobre uma cópia do valor desse argumento. Com exceção do caracter &, entretanto, as instruções da função são idênticas, tenha sido o parâmetro passado por valor ou por referência.

No código abaixo, as instruções da função consideram que "t" é um ponteiro e não uma variável:

```
:
resgistratemp(&atemp);
:
void registratemp (temp *t)
{
        float atual;

        pegatempatual(&atual);

        if (atual > t->alta)
                t->alta = atual;
        else if (atual < t->baixa)
                t->baixa = atual;
}
:
```

Neste caso, em vez de usar expressões como "t.alta" e "t.baixa" para fazer referências aos campos da estrutura, as instruções devem usar o símbolo "->" [4].

É importante salientar que quando um objeto é passado como um argumento para uma função, uma cópia desse objeto é feita. Além disso, quando a cópia é feita, a função construtora desse objeto não é chamada. No entanto, quando a função termina, a destrutora da cópia é chamada. Se, por alguma razão, o usuário não desejar fazer uma cópia ou não quiser que a função destrutora seja chamada, basta passar o objeto por referência. Quando um objeto é passado por referência, nenhuma cópia dele é feita. Isso significa que nenhum objeto usado como parâmetro e destruído quando a função termina, e a destrutora do parâmetro não é chamada.