6. VETORES, PONTEIROS E REFERÊNCIAS

É possível referir-se a objetos de dados pelos seus endereços ou seus nomes. Nesta seção são mostrados meios mais flexíveis de designar objetos de dados, o que envolve nomes, endereços e o relacionamento entre eles.

Como em qualquer outra linguagem de programação um ***array*** ou **vetor** é uma série de objetos de dados, todos eles possuindo o mesmo tipo; estes objetos de dados são chamados de elementos do vetor. Um vetor é declarado fornecendo-se o tipo dos seus elementos, o nome do vetor e o número de elementos. Por exemplo:

```
int a[5];
```

declara a como um vetor de cinco objetos de dado inteiro; isto é, como uma série de cinco locações de memória, cada uma capaz de armazenar um valor do tipo inteiro.

Pode-se inicializar um vetor quando este é declarado listando os valores iniciais para os seus elementos, da seguinte maneira:

```
int a[5] = {75, 25, 100, -45, 60};
```

Neste caso, se o número de valores iniciais fornecido é menor que o número de elementos do vetor, os elementos restantes são inicializados com zero. Assim,

```
int b[5] = {75, 25};
```

é equivalente a

```
int b[5] = {75, 25, 0, 0, 0};
```

Se não for especificado o número de elementos para um vetor inicializado, o número de valores iniciais fornecidos determina o número de elementos. Assim,

```
int c[ ] = {75, 25, 100, -45, 60};
```

declara um vetor de cinco elementos.

Os elementos de um vetor são designados, considerando o exemplo anterior, por *a[0]*, *a[1]*, *a[2]*, *a[3]* e *a[4]*. Os inteiros entre chaves são "índices" e variam de zero até o número máximo de elementos do vetor menos um. Os índices podem ser tanto expressões como constantes. Assim *a[i+j]* refere-se ao elemento designado pelo valor resultante da expressão *i+j*.

Quando se trabalha com *loops*, expressões como *c[i++]* e *c[i--]* são freqüentemente usadas. Neste caso, depois do valor corrente de *i* ser usado como índice, o valor de *i* é incrementado ou decrementado. Assim, na próxima vez que *i* é usado como índice ele vai referir-se ao próximo elemento (para ++) ou ao elemento anterior (para --) do vetor [3].

Apesar dos programadores em geral reconhecerem nos **ponteiros** o principal ponto de dificuldade no aprendizado da linguagem, sua utilidade é incontestável. Com ponteiros o programador pode otimizar o gerenciamento da memória, reutilizando espaços alocados para variáveis, compartilhando espaço entre variáveis e solicitando ou liberando memória dinamicamente. Além disso, muitas aplicações têm sua performance otimizada em termos de velocidade quando se usam ponteiros.

Um ponteiro é um endereço de um objeto de dado de um tipo particular. Em outras palavras, ponteiros nada mais são do que variáveis cujos valores são endereços de memória. A sintaxe de declaração de ponteiros é a seguinte:

```
<tipo_variavel> * <nome_ponteiro>;
```

Para guardar valores em variáveis ponteiro, deve-se utilizar o "operador de endereço" "&" à frente de uma variável e para ler o conteúdo que um ponteiro indica deve-se utilizar o operador unário "*" à frente do ponteiro. Em resumo [9]:

```
<ponteiro> = &<variavel>;
*<ponteiro> = <variavel>;
```

O tipo é essencial para a utilização de objetos de dados designados pelo endereço. O endereço simplesmente dá a posição de memória inicial que contém o objeto de dado. O tipo determina o tamanho do objeto de dado, e as funções e operadores que podem ser aplicados a ele. Assim, não se tem apenas um único tipo de ponteiro para todos os endereços. Tem-se um tipo de "ponteiro-para-*int*" para endereçar objetos de dados inteiros, um tipo "ponteiro-para-*double*" para endereçar objetos de dados *double* e assim por diante.

O operador unário "*" é chamado de "operador indireto", uma vez que ele permite que um objeto de dado seja referenciado indiretamente, através de um identificador nomeando uma variável ponteiro, ao invés de diretamente, através de um identificador nomeando o próprio objeto de dado. O termo ponteiro é aplicado tanto a variáveis ponteiro como aos endereços que elas contêm; o contexto no qual a palavra é usada determina quando significa uma variável ponteiro ou um valor de ponteiro (endereço).

A seguir são apresentado alguns exemplos de declarações de variáveis ponteiros:

```
int *p;          // ou int* p;
int *p, *r, *s;
double *q;       // ou double* q;
```

Neste caso, "p" é um ponteiro para um inteiro e "q" é um ponteiro para um *double*.

Para os ponteiros serem usados efetivamente, é necessário operadores e expressões que irão fornecer os endereços correntes dos objetos de dados. Tal operador é o "operador de endereço" representado por "&", que retorna um ponteiro para um objeto designado. Assim, por exemplo, se "cont" é uma variável inteira, "&cont" é um ponteiro que aponta para "cont" e possui o tipo "ponteiro-para-*int*". Pode-se usar o endereço de um operador para inicializar e atribuir valores para variáveis ponteiro. Por exemplo, depois da declaração

```
int n;
int* p = &n;
```

a variável ponteiro "p" aponta para variável inteira "n". Além disso, "*p" é um nome alternativo para "n". Uma atribuição a "*p" irá alterar o valor de "n", assim como uma atribuição a "n" irá alterar o valor de "*p".

Também pode-se fazer com que "*p" seja um nome alternativo para outra variável, simplesmente atribuindo outro valor a "p". Considerando que "m" seja uma variável do tipo inteira, então

```
p = &m;
```

atribui o endereço de "m" a "p".

Pode-se usar ponteiros para dar às funções acesso a variáveis passadas como argumentos. Por exemplo, suponha que queira-se escrever uma função que troque os valores de duas variáveis inteiras. Assim, depois do comando

```
swap (i, j);
```

ser executado, "i" terá o valor de "j", e "j" terá o valor anterior de "i". Em C e C++ argumentos são passados por valor somente se os valores atuais dos argumentos são apenas transmitidos para a função.

Por outro lado, ponteiros são valores e podem ser passados para funções. Se forem passados os ponteiros para a função "*swap*", esta poderá usá-los para acessar as variáveis e trocar seus valores. Assim, declara-se "*swap()*" para ter dois "ponteiro-para-*int*" como argumentos:

```
void swap (int* p, int* q);
```

Quando chama-se a função, deve-se passar para ela ponteiros para as variáveis cujos valores deseja-se trocar. Pode-se fazer isso com a ajuda do operador de endereço. O comando

```
swap (&i, &j);
```

chama a função e passa os endereços de "i" e "j" como argumentos. A definição de "*swap*" é apresentada a seguir.

```
void swap (int* p, int* q) {
        int temp = *p;      // salva o valor de p
        *p = *q;            // atribui q para p
        *q = temp;          // q recebe valor inicial de p    }
```

Variáveis passadas para a função desta maneira são chamadas de variáveis passadas por referência, porque referências para as variáveis (isto é, ponteiros para elas) são passadas ao invés de valores de variáveis.

Ponteiros também podem ser usados para "apontar" para estruturas. Por exemplo:

```
struct date {
        int day;
        int month;
        int year;
}
date dt;            // variável dt
date* pdt = &dt;    // ponteiro para dt
```

Neste caso, pode-se referir aos membros da variável "dt" diretamente, da seguinte maneira: "dt.day", "dt.month" e "dt.year". Também pode-se referir aos membros da variável indiretamente, através do ponteiro "pdt": "(*pdt).day", "(*pdt).month", "(*pdt).year". Os parênteses são necessários porque o operador "." tem maior precedência. Como estas expressões ocorrem freqüentemente, C++ fornece um operador de acesso indireto, "->", para simplificá-las. Assim, a expressão "pdt->day" é equivalente a "(*pdt).day" [3].

Torna-se interessante comentar que o C++ acrescentou algumas características à sintaxe de ponteiros. Basicamente, foi acrescentado o tipo *void* para ponteiros e ampliado o conceito *const*, associando-o aos ponteiros. Para declarar um ponteiro *void*, a sintaxe é a mesma da declaração de qualquer tipo, ou seja, o tipo seguido do operador "*":

```
void * <nome_ponteiro>;
```

Exemplo:

```
void * ptr;
```

A diferença para os demais ponteiros está no fato de que o ponteiro *void* pode apontar para qualquer tipo de dados, inclusive o NULL. O C++ permite que se acesse o conteúdo apontado por um ponteiro genérico somente após ele ser formatado, isto é, eles não podem ser referenciados sem um *casting* explícito, porque o compilador não pode determinar o tamanho do objeto para o qual ele está apontando. No entanto, pode-se atribuir outros ponteiros a um ponteiro *void* ou atribuir o valor dele a um ponteiro comum. A motivação para a criação de ponteiros *void* está no fato de se poder especificar funções de propósito geral cujo parâmetro é um ponteiro a formatar em tempo de execução. Baseando-se em critérios internos da função (provavelmente um *flag* especificando o tipo), este ponteiro pode ser modelado, de modo a apontar para o tipo adequado. Exemplo:

```
void Imprime (void *ptr, int flag);
void main () {
        int inteiro = 10;
        float real = 5.5;
        Imprime (&inteiro, 1);
        Imprime (&real, 2);
}
void Imprime (void *ptr, int flag)  {
        float soma = 22.2;
        switch (flag) {
                case 1: cout << "Numero Inteiro = " << (* (int *) ptr) << "\n";
                        break;
                case 2: cout << "Numero Real = " << (* (float *) ptr) << "\n";
        }
        ptr = &soma;
        cout << "Conteudo do ponteiro = " << (* (float *) ptr) << "\n";
        * (float *) ptr = 22.2 + soma;
        cout << "Conteudo do ponteiro = " << (* (float *) ptr) << "\n";
}
```

Em C++ pode-se definir ponteiros para constantes (apontadores cujo conteúdo é imutável), ponteiros constantes (apontadores que não mudam o endereço apontado) e ponteiros constantes para valores constantes. O primeiro tipo é usado quando deseja-se que o conteúdo de um ponteiro fique inalterável durante o processamento de um programa. Neste caso utiliza-se a seguinte sintaxe:

```
const <tipo_a_apontar> * <nome_ponteiro>;
```

Exemplo:

```
const char *string = "Eu sou uma constante!";
```

Ponteiros constantes são usados quando há interesse em que um determinado endereço de memória seja sempre localizado por um certo ponteiro, independente do valor nesse endereço. Nesses casos quem deve ser constante é o ponteiro, e não seu conteúdo. A sintaxe para ponteiro constante é a seguinte:

```
<tipo_a_apontar> *const <nome_ponteiro>;
```

Exemplo:

```
char * const msg = "Meu pontero eh constante!";
```

Neste caso, a *string* "Meu ponteiro eh constante!" pode ser modificada, mas qualquer nova frase colocada sobre ela terá sempre o ponteiro "msg" a indicá-la. O compilador não permitirá que se aponte "msg" para outro local de memória, mesmo que lá esteja um *char*.

Também é possível desejar-se a total segurança de determinados dados declarando seu valor como constante e ajustando um ponteiro constante para o endereço onde estará esse valor. Para tal, utiliza-se ponteiro constante para constante através da seguinte sintaxe:

```
const <tipo_a_apontar> *const <nome_ponteiro>;
```

Exemplo:

```
const char * const msg = "Todos somos constantes!";
```

Quando tanto o ponteiro quanto o valor apontado são constantes, o compilador "congela" ambos e não permite nem modificações no conteúdo do ponteiro, nem no local de memória para o qual ele aponta. No exemplo, a *string* "Todos somos constantes!" é inalterável e será sempre encontrada pelo igualmente imutável ponteiro "msg" [9].

Uma **referência** consiste em um apelido (*alias*) de uma variável, ou seja, apesar de possuírem nomes distintos, uma referência e sua variável de inicialização ocupam o mesmo endereço de memória. A modificação de uma afeta a outra. A principal utilidade de uma referência é melhorar a sintaxe de manipulação de parâmetros por referência. A sintaxe de declaração de referência é a seguinte:

```
<tipo> &refer = var;
```

onde a referência "refer" constitui uma segunda forma de chamar a variável "var". Por exemplo, "*int &*" é um tipo "referência-para-*int*", "*double &*" é um tipo "referência-para-*double*", e assim por diante. Como regra geral, uma referência deve ser sempre inicializada na sua declaração, depois ela serve como outro nome para aquele objeto de dado. Por exemplo, suponha a declaração de uma variável inteira "n":

```
int n = 10;
```

e a declaração da referência "r" através de

```
int& r = n;
```

Tanto "n" quanto "r" designam o mesmo objeto de dado, que pode ser referenciado pelos dois "nomes". Assim, a atribuição "*n = 20;*" altera o valor de "r" e "n" para 20, da mesma maneira que "*r = 15;*" altera o valor de "r" e "n" para 15.

É interessante observar que a inicialização e a atribuição são completamente diferentes para referências. A inicialização estabelece uma correspondência entre a referência e objeto de dado. A atribuição, que é igual para referência e outra variável/valor, somente atribui um novo valor para um determinado objeto de dado. Também deve-se ter em mente que uma referência não é uma cópia nem um ponteiro para uma variável. Trata-se da própria variável com um nome diferente. A principal vantagem é mesmo a passagem de parâmetros por referência (seção 8), daí, inclusive, o nome deste tipo de variável [3, 9].