Ponteiros

Gonzalo Travieso

2019

1 Dados na memória

Como vimos, o computador usa bits para a representação de todos os valores sendo processados. Durante a execução do programa, esses bits estão armazenados na memória,[1] organizados em *bytes* de 8 bits. Num computador com 8 Gbytes de memória existem 2^{33} bytes.[2] Ao acessar esses bytes, o computador precisa distinguir uns dos outros. Para isso, é usado um esquema parecido com a indexação de vetores: damos um "índice", começando em 0, para cada um dos bytes; no exemplo da memória de 8 Gbytes, teríamos "índices" de 0 a $2^{33}-1$, ou em hexadecimal de `0x000000000` a `0x1fffffffff`. Os "índices" dos bytes são denominados seu *endereço*.

Quando declaramos uma variável, o compilador verifica o tipo da variável para saber quantos bytes ela precisa e então reserva um espaço na memória para essa variável. Por exemplo, um `int` é representado (nos sistemas que estamos usando) por 32 bits, ou 4 bytes. Quando declaramos uma variável do tipo `int`, o compilador irá reservar espaço na memória de 4 bytes consecutivos. O endereço do primeiro byte reservado é denominado o *endereço da variável*.

É claro que não apenas variáveis, mas qualquer dado presente no programa tem algum endereço associado, pois deve estar guardado em alguma posição de memória. Por exemplo, cada elemento individual de um vetor possui um endereço.

2 Endereços e ponteiros (brutos)

Em algumas situações (veremos exemplos em seguida) queremos saber o endereço de um elemento na memória. Para lidar com endereços em códigos C++, precisamos de variáveis de um tipo denominado *ponteiro*. Uma variável ponteiro é capaz de armazenar um endereço de memória. Ao especificar um ponteiro, precisamos especificar qual o tipo de elemento que deve estar no endereço que será armazenado nessa variável. Isto é necessário para podermos saber como operador com um endereço, visto que a posição de memória nesse endereço poderia conter qualquer tipo de dados. A especificação de variáveis ponteiros é como segue:

```
int *p;
```

[1] Na prática, o assunto é bem mais complicado se levarmos em conta e existência de registradores, caches e memória virtual, mas a descrição dada é apropriada para a compreensão dos programas.

[2] $1K = 2^{10}$, $1M = 2^{20}$, $1G = 2^{30}$, $1T = 2^{40}$.

Neste código, estamos declarando uma variável denominada `p` que é do tipo *ponteiro para* `int`, isto é, ela pode guardar o endereço onde um `int` está armazenado na memória. O operador `*` aqui se lê como "ponteiro", e usando a regra da leitura da direita para a esquerda, lemos: `p` é um ponteiro para `int`.

3 Usando ponteiros

3.1 Operador de endereço `&`

Uma forma de conseguir um ponteiro quando já temos acesso a um elemento na memória é usar o operador `&`, ou *operador de endereçamento*. Podemos conseguir o endereço de uma posição de memória colocando esse operador à esquerda do elemento para o qual queremos o endereço. Vejamos alguns exemplos.

```
int a{2};
int *p; // Esta variável é um ponteiro para int

p = &a; // p tem o endereço da variável a
```

Após a execução do código acima, dizemos que `p` *aponta* para `a`.

Não precisamos apontar para uma variável específica, mas podemos apontar para outros elementos aos quais temos acesso, por exemplo, os elementos de um vetor:

```
std::vector<int> alguns_primos{2, 3, 5, 7, 11, 13, 17, 19, 23, 29};
p = &alguns_primos[5]; // p agora tem o endereço do 13 na memória.
```

Podemos ter ponteiros para qualquer tipo de dados, por exemplo:

```
double x{1.5};
std::string nome{"Justiniano"};

double *pd;
std::string *ps;

pd = &x;
ps = &nome;
```

Mas não é permitido colocar o endereço de um tipo diferente do ponteiro:

```
pd = &a; // ERRO: a é um int e pd é double*
ps = pi; // ERRO: pi é int*, enquanto ps é std::srring*
```

Para qualquer tipo, o endereço do elemento na memória é, como dito, o endereço de memória do primeiro byte que o representa. Neste sentido, todos os tipos de ponteiros são semelhantes: eles precisam armazenar um endereço de memória de acordo com a arquitetura do processador; em uma máquina de 64 bits um ponteiro precisa então de 8 bytes para sua representação.

3.2 Operador de acesso por ponteiro `*`

Uma vez que temos um ponteiro apontando para um elemento na memória, certamente iremos querer fazer algum acesso a esse elemento. Isso pode ser

feito com o uso do operador unário * (não confundir com o operador binário * usado para multiplicação):

```
std::vector<int> alguns_primos{2, 3, 5, 7, 11, 13, 17, 19, 23, 29};
int *p = &alguns_primos[2];
int dez = *p * 2;
```

Também podemos alterar o valor apontado:

```
double x{0.0};
double *px{&x};

*px = 1.3; // Agora x vale 1.3
```

3.3 Ponteiros nulos

Uma característica importante de ponteiros é que, como eles são variáveis e podem ter seu valor alterado, em um dado ponto eles podem *não estar apontando para um elemento válido*. Para distinguir um ponteiro que não aponta para algo de um ponteiro que aponta, temos o conceito de *ponteiro nulo*, representado em C++ pela constante literal `nullptr`. Se um ponteiro tem o valor `nullptr` então ele não aponta para nada. Obviamente, não podemos usar o operador de acesso em um ponteiro com valor `nullptr`.

Podemos então distinguir três casos para uma variável ponteiro:

1. O ponteiro está apontando para um elemento válido na memória.
2. O ponteiro tem o valor `nullptr`, indicando que ele não aponta para nada.
3. O ponteiro tem um valor diferente de `nullptr`, mas não aponta para um elemento válido.

Exemplos em que esses três casos ocorrem:

```
int rodas{4};
int *p_bom{&rodas}; // Este ponteiro aponta para um int válido.
int *p_nulo{nullptr}; // Este ponteiro não aponta para nada.
int *p_undef; // Este pode ou não apontar para algo (veja abaixo)
```

O terceiro caso é devido à falta de inicialização de variáveis em C++ se um valor inicial não for especificado. Como não especificamos um valor inicial para `p_undef`, ele pode ter qualquer valor: se dermos sorte, ele será `nullptr`, mas pode ser um valor diferente desse, e inválido.

Em nosso código, podemos distinguir entre o segundo caso e os outros dois (comparando o valor do ponteiro com `nullptr`), mas não entre o primeiro e o terceiro casos. Por essa razão, não é boa idéia declarar variáveis do tipo ponteiro sem inicializá-las (inicializamos com `nullptr` se não sabemos na declaração qual o valor inicial a usar).

3.4 Ponteiros de ponteiros

Um ponteiro é uma variável como qualquer outra, e precisa ser armazenado na memória em algum lugar. Isto significa que ele também possue um endereço, e portanto podemos ter um ponteiro para ele:

```
int a{2};
int *p{&a}; // p é ponteiro para um int
int **pp{&p}; // pp é ponteiro para um pnnteiro para int
int ***ppp{&pp}; // ppp é um ponteiro para um ponteiro para um
                 // ponteiro para int
// etc.
```

Esse exemplo também reforça a idéia que precisamos manter em mente ao usar ponteiros: quando lidamos com ponteiros, estamos sempre lidando com dois objetos: o ponteiro e o objeto apontado por ele. Esses dois objetos podem ser trabalhados independentemente:

```
int a{2}, b{4};
int *p{&a};
int **pp{&p};

*p = 3; // Muda o valor de a para 3
*pp = &b; // Faz p apontar para b
**pp = 5; // Faz b valer 5.
(*p)++; // Agora b vale 6.
```

3.5 Acesso a membro por ponteiro

Também podemos ter ponteiros para objetos de tipo `struct` (na verdade, isso é bastante comum):

```
struct Point {
    double x, y;
};

Point position{0.0, 0.0};
Point *current{&position};

position.x = 1.0; // Muda o ponto para (1, 0)

(*current).y = (*current).x / 2; // Muda o ponto para (1, 0.5)
```

Quando usamos ponteiros para `struct`, código do tipo `(*p).campo` como usado acima é bastante comum: queremos acessar o objeto apontado pelo ponteiro e então acessar um de seus membros. Por essa razão, o C++ tem uma abreviação para isso com o operador de acesso a membro por ponteiro `->`. Com esse operador, podemos substituir a última linha do código acima por:

```
current->y = current->x / 2;
```

Isto é mais econômico e mais claro.

4 Alocação dinâmica de memória

Quando declaramos uma variável, o compilador se encarrega do código de reserva do espaço de memória para ela, bem como da liberação do espaço quando a variável sair do escopo.

```
int div_by_smaller(int a, int b) {
    int result;
    if (a < b) {
        result = b / a;
    }
    else {
        result = a / b;
    }
    return result;
}
```

Neste código, as variáveis `a`, `b` e `result` são criadas (tem memória reservada para elas) quando a função for chamada. Ao terminar a execução da função, o espaço ocupado por elas na memória é liberado.

A alocação de memória pode também ser feita *dinamicamente* (isto é, em tempo de execução sob controle do programador) usando o operador `new`:

```
struct Point {
    double x, y;
};

Point *p_point = new Point;

p_point->x = 1.0;
p_point->y = 0.5;
```

O operador `new` deve ser seguido pelo nome do tipo do dado a ser alocado e ele retorna um **ponteiro** com o endereço do dado alocado.

Neste exemplo, o compilador cuida da variável `p_point`, mas não do objeto do tipo `Point` criado pelo `new`. Portanto o usuário deve tomar cuidado para liberar a memória reservada a ele quando não precisar mais do objeto. Isso é feito pelo operador `delete`, que deve receber um ponteiro para o objeto a ser liberado.

```
delete p_point;
```

É responsabilidade do programador determinar quando a liberação deve ocorrer, e é um erro tentar acessar o objeto após realizar essa liberação, ou deixar de realizar a liberação.

Uma possibilidade adicional é alocar diversos objetos do mesmo tipo de uma vez. A forma de fazer isso é usar o operador de indexação `[]` em conjunto com o `new`, como exemplificado abaixo:

```
Point *points = new Point[100];
```

Este código aloca 100 objetos do tipo `Point` na memória (organizados um após o outro), e retorna um ponteiro com um endereço do primeiro byte do primeiro desses objetos. Quando realizamos uma alocação dessa forma, precisamos realizar a liberação através do operador `delete []`:

```
delete [] points;
```

Se não usarmos o `[]` a memória não será corretamente liberada.

A questão agora é: como acessar os diversos objetos alocados dessa forma? Isso é feito através da chamada *aritmética de ponteiros*. Como um ponteiro é um endereço de memória, se temos diversos objetos consecutivamente na memória, seus endereços serão consecutivos, com a diferença entre o endereço de um objeto e do seguinte dada pelo tamanho do objeto. Para exemplificar, consideremos o caso de diversos objeto `int` consecutivos, numa máquina onde um `int` ocupa 4 bytes. Se temos diversos `int` consecutivos e o endereço de um deles (no meio do conjunto) for `0x40000808` então o próximo objeto terá endereço `0x4000080c` e o anterior terá endereço `0x40000804`. Em nosso programa, os endereços estão em ponteiros, portanto basta fazer operações aritméticas sobre esse ponteiros. Para facilitar, como o compilador sabe o tamanho dos objetos apontados (pois ele sabe o tipo do ponteiro), então indicamos o deslocamento através do *número de objetos* a deslocar, ao invés do número de bytes. Se `p` é um ponteiro para um `int`, então `p+1` é um ponteiro para o `int` seguinte, e `p-1` aponta para o `int` anterior. No caso do nosso vetor de `Point` do exemplo anterior, podemos então fazer:

```
for (int i = 0; i < 100; ++i) {
    (points + i)->x = 0.1 * i;
    (points + i)->y = 0.2 * i - 0.1;
}
```

Aqui `points + i` calcula um ponteiro para o i-ésimo `Point` consecutivo entre os alocados pelo `new[]`, e então acessamos os campos `x` e `y` desse ponto através do operador `->`. Para facilitar esse tipo de código, quando `p` é um ponteiro, o compilador aceita o uso de indexação em `p`, transformando `p[i]` em `*(p + i)`, o que permite simplificar o código anterior para (lembrando que `p->a` é o mesmo que `(*p).a`):

```
for (int i = 0; i < 100; ++i) {
    points[i].x = 0.1 * i;
    points[i].y = 0.2 * i - 0.1;
}
```

Note como desta forma ponteiros funcionam como *arrays*. Na verdade, em C e C++ antigo esta era a única forma de usar *arrays* com tamanho determinado durante a execução do código (não conhecido durante a compilação). O uso de ponteiros para representar *arrays* **não é recomendado em C++ moderno**.

5 Ponteiros inteligentes

5.1 O problema da posse

Vejam as defições de ponteiros abaixo:

```
int *p1, *p2, *p3;
int a;

p1 = &a;
p2 = new int;
```

Note como os três ponteiros agora são diferentes: `p1` está apontando para um objeto associado a uma variável alocada pelo compilador; `p2` está apontando para um objeto alocado dinamicamente; `p3` não tem valor válido. Se os ponteiros `p1`, `p2` e `p3` sairem do escopo neste ponto, o que deve ser feito? Como o objeto apontado por `p1` foi alocado automaticamente pelo compilador, nada precisa ser feito; no caso de `p3`, por outro lado, a alocação foi feita com `new` e portanto precisamos fazer `delete`; já sobre `p3` não podemos fazer `delete`. Suponha agora que o código fosse o seguinte:

```
int *p1, *p2, *p3;
int a;

p1 = &a;
p2 = new int;
p3 = p2;
```

Aqui `p3` também está apontando para um objeto alocado dinamicamente: o mesmo apontado por `p2`. Devemos fazer `delete` sobre `p2` ou sobre `p3`? No código do exemplo, a solução é simples: vemos que nenhum `delete` é necessário sobre `p1` e que um `delete` é necessário sobre ou `p2` ou `p3` **mas não sobre ambos** (pois estaríamos tentando liberar um objeto já liberado, o que é um erro). Já num código mais complexo, as relações de posse podem ser muito difíceis de serem seguidas.

Descrevemos o problema apresentado pelo conceito de *posse*: um ponteiro, ao apontar para um objeto, pode ou não ser "dono" desse objeto, e portanto responsável por liberar a memória quando necessário. No nosso exemplo, `p1` não é dono do objeto que aponta; já entre `p2` e `p3`, precisamos designar um como dono, e o outro não será dono.

Considerando posse, o C++ faz distinção entre 4 tipos de ponteiros:

1. Ponteiros de posse única.
2. Ponteiros de posse compartilhada.
3. Ponteiros fracos.
4. Ponteiros brutos.

Esses tipos de ponteiros são descritos brevemente a seguir.

5.2 Ponteiros de posse única

Um ponteiro de posse única é responsável isoladamente pelo objeto ao qual aponta, isto é, ele deve liberar o objeto quando não mais necessário. Estes ponteiros são representados pelo tipo `std::unique_ptr<T>` (onde `T` é o tipo do objeto apontado), definido no header `<memory>`. Estes ponteiros são usualmente criados através da função auxiliar `std::make_unique<T>`, como no exemplo:

```
std::unique_ptr<int> p1 = std::make_unique<int>(2);
```

Este código aloca novo espaço na memória para um `int`, inicializa esse objeto com o valor 2 e retorna um `std::unique_ptr<int>` para ele. O código fica mais simples usando dedução de tipo:

```
auto p1 = std::make_unique<int>(2);
```

Como esse ponteiro tem responsabilidade única pelo objeto apontado, não podemos fazer cópias dele (pois senão teríamos outro ponteiro com responsabilidade "única" sobre o mesmo objeto, o que não faz sentido):

```
auto p1 = std::make_unique<int>(2);
std::unique_ptr<int> p2;
p2 = p1; // ERRO!
```

O que podemos fazer é **transferir a responsabilidade** pelo objeto de `p1` para `p2`, se desejarmos:

```
auto p1 = std::make_unique<int>(2);
std::unique_ptr<int> p2;
p2 = std::move(p1); // OK.
```

Neste código, agora a responsabilidade pelo objeto alocado dinamicamente está com o ponteiro `p2`, e o ponteiro `p1` está "vazio" (na verdade, em um estado que não permite seu uso, a não ser para colocar um novo objeto a ser apontado por ele).

5.3 Ponteiros de posse compartilhada

Quando queremos indicar posse, o caso normal é que a posse seja única, e usamos um `std::unique_ptr`, mas em alguns casos desejamos uma posse compartilhada. Nestas situações, temos vários ponteiros apontando para o objeto, e o objeto deve permanecer alocado enquanto pelo menos um dos ponteiros precisar dele. O objeto somente pode ser desalocado quanto todos os ponteiros que têm posse compartilhada sobre ele não mais precisam do objeto. Isto é indicado pelo uso de `std::shared_ptr<T>`, também definido em `<memory>`. Os objetos são geralmente alocados através da função `std::make_shared<T>`

```
auto p1 = std::make_shared<int>(2);
std::shared_ptr<int> p2;
p2 = p1; // OK.
```

Neste código, a responsabilidade pelo objeto `int` alocado é compartilhada pelos ponteiros `p1` e `p2`.

5.4 Ponteiros fracos

Ponteiros fracos, definidos com o tipo `std::weak_ptr<t>` (também em `<memory>`) são um caso especial, necessário quando trabalhamos com posse compartilhada mas o uso exclusivo de `std::shared_ptr` poderia levar a referências circulares. Não discutiremos em mais detalhes aqui este ponto.

5.5 Ponteiros brutos

Ponteiros brutos (ou ponteiros ao natural; *raw pointers* em inglês) são aqueles definidos usando `T *`, onde `T` é um tipo (por exemplo, `int *`). Conforme já discutido, estes ponteiros não carregam informação de posse.

5.6 Como escolher?

Quando usamos um ponteiro, precisamos escolher qual deles usar, e para isso devemos nos guiar pela posse:

- Se estamos alocando um objeto na memória, em geral queremos já determinar um ponteiro para ser responsável pelo seu gerenciamento. O caso mais comum é que tenhamos um ponteiro único para isso, e portanto usamos `std::unique_ptr`.
- Se estamos em uma situação (mais rara) em que a posse não pode ser atribuída a um único ponteiro, então usamos um conjunto de `std::shared_ptr`.
- Se apenas precisamos apontar para um objeto, mas sem indicar que tomamos posse dele, usamos um ponteiro bruto.
- Usamos um `std::weak_ptr` nos casos raros em que eles são necessários.

Estes fatores ficam mais claros através de exemplos, então o melhor é observar como esses diversos tipos de ponteiros são usado em códigos reais bem escritos.

5.7 Exemplos

Vejamos agora alguns exemplos de como isso funciona:

```
#include <memory>

void f(int *rp) {
    int *rp_f = new int; // Cria novo int
    *rp_f = 7;
    *rp = *rp_f - *rp;

    // Ponteiro rp_f para o int criado vai sair de escopo.  Precisamos
    // lembrar de liberar a memória.
    // Já o rp não podemos liberar, pois ele foi criado no main.
    delete rp_f;
}

void f(std::unique_ptr<int> up) {
    // up tem responsabilidade única pelo que ele aponta.
    auto up_f = std::make_unique<int>(8); // Cria novo int
    *up = *up_f - *up;
    // Novo int apontado por up_f é automaticamente liberado.  int
    // apontado por up tambem é liberado (a variável up é unica
    // responsável por ele).
}

void f(std::shared_ptr<int> sp) {
    // sp tem responsabilidade compartilhada pelo que aponta.
    auto sp_f = std::make_shared<int>(9); // Novo int
    *sp = *sp_f - *sp;
```

```
    // sp_f sai de escopo e libera responsabilidade pelo int. Como era
    // o único responsável, a memoria associada é liberada.  sp tambem
    // sai de escopo. Mas como ele tem responsabilidade compartilhada
    // com o shared_ptr do main, a memória não é liberada.
}

int main(int, char const *[]) {
    int *rp1 = new int; // Novo int
    *rp1 = 1;
    auto up1 = std::make_unique<int>(2); // Movo int, responsabilidade única
    auto sp1 = std::make_shared<int>(3); // Novo int, responsabilidade compartilhada
    {
        int *rp2 = new int; // Novo int
        *rp2 = 4;
        auto up2 = std::make_unique<int>(5); // Novo int, responsabilidade única
        auto sp2 = std::make_shared<int>(6); // Novo int, responsabilidade compartilhada
        int *rp3 = rp1; // Outro ponteiro para um int já existente.
        std::shared_ptr<int> sp3;
        sp3 = sp1; // Outro ponteiro com responsabilidade
                   // compartilhada para int já existente

        // rp2 sai de escopo. Precisa liberar memoria.
        delete rp2;
        // up2 sai de escopo. Liberação de memória automática.  sp2
        // sai de escopo: também libera memoria, pois é único
        // responsável.  sp3 sai do escopo, mas é responsável
        // compartilhado com sp1 pelo mesmo int. Portanto a memória
        // não é liberada.
    }

    // Ao passar um int* para função, não informamos posse. Neste
    // código, a posse fica com o maini.
    f(rp1);
    // Não podemos copiar um unique_ptr para a função. Aqui movemos o
    // seu conteúdo, isto é, deixamos explícito que a posse do
    // ponteiro para para a função.
    f(std::move(up1));
    // Ao passar um shared_ptr, fazermos uma cópia, criando dois
    // shared_ptr compartilhando o mesmo objeto, um no main e um na
    // função.
    f(sp1);

    // rp1 sai de escopo e está apontando para um int que precisa ser
    // liberado.
    delete rp1;
    // up1 sai de escopo: int liberado automaticamente.  sp1 sai de
    // escopo, e com isso não temos mais nenhum shared_ptr responsável
    // pelo objeto alocado, e portanto a memória é liberada.
}
```