Memória e passagem de parâmetros

Marco A L Barbosa
malbarbo.pro.br

Departamento de Informática
Universidade Estadual de Maringá

Até o momento, quais eram as nossas preocupações no projeto de programas?

- Identificar o problema
- Resolver o problema
- Escrever bom código

Agora vamos discutir outro aspecto importante no projeto de programas: o uso de recursos, especificamente o uso de memória.

Memória

A memória é um recurso compartilhado entre os diversos programas que estão em execução em um computador. O sistema operacional faz a gerência de memória entre os diversos processos e garante que cada processo só tenha acesso a memória destinada a ele.

Cada processo também precisa gerenciar a sua própria memória.

Algumas linguagens como Python, Java e Go, fazem a gerência automática da memória. A parte do C++ que vimos até agora também faz a gerência automática da memória. Algumas outras linguagens, como C, requerem que o programador faça a gerência da memória de forma manual (explícita).

Cada forma de fazer a gerência de memória tem diversas vantagens e desvantagem, mas o ponto principal é a facilidade de programação vs o controle. Vocês vão aprender mais sobre isso ao longo do curso!

A gerência de memória requer basicamente duas operações: a alocação e a desalocação de memória.

O que significa alocar memória? É reservar um espaço de memória para ser usada de uma determinada forma.

O que significa desalocar memória? É devolver para o sistema um espaço de memória que havia sido alocada previamente para que ela possa ser usada de outra forma.

Desafio

Considere um caderno com 100 linhas, onde as linhas estão enumeradas em sequência de 0 a 99, e cada linha representa uma célula de memória. Simule no caderno como a memória é usada durante a execução do seguinte programa

```
int i = 10;
vector<int> v = {1, 2, 3};
if (i > 4) {
    int j = i + 1;
    v.push_back(j);
}
vector<int> w = {10};
for (int t : v) {
    w.push_back(t);
}
```

Que conclusão podemos tirar dessa atividade? A gerência de memória **pode** ser complicada!

Para facilitar a gerência de memória o compilador do C++ (os compiladores em geral), costumam dividir a memória em duas regiões: a pilha e o heap.

A pilha é geralmente utilizada para alocar as variáveis de tamanho fixo e o heap é usado para alocar as variáveis de tamanho dinâmico.

A pilha é gerenciada com uma estratégia simples:

- A memória é sempre alocada após a última memória alocada ainda em uso
- A memória é desalocada sempre na ordem inversa da alocação

O heap é gerenciado com estratégias mais elaboradas e permite a desalocação em qualquer ordem.

No modelo de execução linha a linha, quando uma linha que contém uma declaração de variável é executada, o C++ reserva automaticamente uma memória para aquela variável.

Quando uma variável sai do escopo, o C++ desaloca automaticamente a memória daquela variável.

E as atribuições de variáveis?

Quando as variáveis tem tamanho fixo, o valor que está sendo atribuído é armazenado diretamente na memória da variável.

Quando as variáveis tem tamanho dinâmico, a memória alocada para a variável de destino é desalocada e uma nova memória de tamanho adequado é alocada, então o valor que está sendo atribuído é armazenado nessa nova memória.

```cpp
int a = 10;
int x = 123;
// Armazena 20 e depos 123 na memória previamente alocada para a
a = 20;
a = x;
vector<int> v = {1, 6, 10};
vector<int> w = {10, 1, 7, 5};
// Explicação simplificada
// - Desaloca a memória de v
// - Aloca uma outra porção de memória suficiente para armazenar os valores de w
// - Copia os valores de w para a nova memória de v
v = w;
// Mesma coisa que antes, mas o vetor que está sendo copiado é criado implicitamente
v = {1, 3, 1};
```

E a passagem de parâmetros?

Por padrão, o C++ usa a mesma estratégia de atribuição de variáveis, a cópia dos valores.

Em algumas situações essa estratégia pode ser inconveniente.

Palíndromo

```
bool palindromo(vector<int> valores)
{
    bool palindromo = true;
    for (int i = 0, j = valores.size() - 1; i < j && palindromo; i = i + 1, j = j - 1) {
        if (valores[i] != valores[j]) {
            palindromo = false;
        }
    }
    return palindromo;
}
```

O que acontece na chamada a seguir?

```
vector<int> v = {};
for (int i = 0; i < 1000000; i = i + 1) {
    v.push_back(123);
}
palindromo(v);
```

Uma memória para 1000000 de elementos é alocada para `valores` e `v` é copiado para essa memória. Após a execução de palíndromo a memória de `valores` é desalocada.

Parece um desperdício! E é! Podemo melhorar? Sim!

Ao invés de copiar os valores de `v` para `valores` podemos instruir o C++ a compartilhar esses valores. Fazemos isso especificando que `valores` é uma referência constante com `const &`.

Palíndromo

```
bool palindromo(const vector<int> &valores)
{
    bool palindromo = true;
    for (int i = 0, j = valores.size() - 1; i < j && palindromo; i = i + 1, j = j - 1) {
        if (valores[i] != valores[j]) {
            palindromo = false;
        }
    }
    return palindromo;
}

palindromo(v);
```

Quando `palindromo` é chamado, o valor de `v` é compartilhado com `valores`, isto é, tanto a variável `v` quanto a variável `valores` referenciam a mesma memória (são **apelidos** para a mesma memória). Como a referência é `const`, então não é possível alterar `valores` (ex: `valores.push_back(1)` gera um erro de compilação).

Quando devemos usar referências constantes?

Quando vamos apenas ler os valores de uma variável e queremos evitar que o seu conteúdo seja copiado.

Note que evitar que o conteúdo seja copiado é uma preocupação apenas para variáveis que ocupam muita memória (arranjos, conjuntos e strings com muitos elementos).

Existem referências que não são constantes? Sim!

Uma variável que é uma referência constante não pode ser modificada, já uma variável que é apenas uma referência, pode ser modificada.

Como uma referência é um apelido para a memória de outra variável, então quando uma memória é modificada através de uma referência, de fato, a memória da outra variável é que é modificada.

```
void soma_1_no_primeiro(vector<int> &v)
{
    v[0] = v[0] + 1;
}

examples
{
    vector<int> x = {5, 1, 8};
    soma_1_no_primeiro(x);
    check_expect(x, (vector<int> {6, 1, 8}));
}
```

O que está acontecendo aqui?

Quando `soma_1_no_primeiro` é chamada, `v` passa a referenciar a mesma memória de `x`, então, quando `v` é alterado, de fato é `x` que está sendo alterado.

```
void soma_1_no_primeiro(vector<int> &v)
{
    v[0] = v[0] + 1;
}

examples
{
    vector<int> x = {5, 1, 8};
    soma_1_no_primeiro(x);
    check_expect(x, (vector<int> {6, 1, 8}));
}
```

Lendo o código do exemplo com cuidado, podemos entender a intenção do código, sem se preocupar com como ele funciona. Cria um arranjo `x` com os elementos `{5, 1, 8}`, soma 1 no primeiro elemento de `x`, verifica que a soma foi feita corretamente.

```
void soma_1_no_primeiro(vector<int> &v)
{
    v[0] = v[0] + 1;
}

examples
{
    vector<int> x = {5, 1, 8};
    soma_1_no_primeiro(x);
    check_expect(x, (vector<int> {6, 1, 8}));
}
```

Note que a função `soma_1_no_primeiro` não tem retorno.

Como uma função sem retorno produz algo de útil? Através de **efeitos colaterais**, no caso de `soma_1_no_primeiro`, o efeito colateral é alterar o valor do parâmetro `v`.

Como a função não tem retorno, temos que testá-la em três etapas, inicializar as variáveis, chamar a função e verificar se as variáveis foram alteradas corretamente.

Quando usar referências?

Quando queremos alterar variáveis e não precisamos manter os seus valores antigos.

Quando não usar referências?

Quando precisamos dos valores atuais e dos novos valores das variáveis.

Como projetar funções que modificam os seu argumentos?

Generalizando exemplos concretos.

Exemplo

Dado um arranjo ordenado em ordem não decrescente e um valor v, projete uma função que modifique o arranjo inserindo o valor v de maneira que o arranjo continue em ordem.

Especificação

```
// Insere v em valores de maneira que valores continue em ordem.
// Requer que valores esteja ordenado em ordem não decrescente.
void insere_ordenado(vector<int> &valores, int v) { }

examples {
    vector<int> valores = {};
    insere_ordenado(valores, 10);
    check_expect(valores, (vector<int> {10}));

    insere_ordenado(valores, 1);
    check_expect(valores, (vector<int> {1, 10}));

    insere_ordenado(valores, 2);
    check_expect(valores, (vector<int> {1, 2, 10}));

    insere_ordenado(valores, 12);
    check_expect(valores, (vector<int> {1, 2, 10, 12}));

    insere_ordenado(valores, 10);
    check_expect(valores, (vector<int> {1, 2, 10, 10, 12})); }
```

Exemplo

```
vector<int> valores = {1, 2, 10, 12};
int v = 9;

valores.push_back(v);
// valores = {1, 2, 10, 12, 9}

int t = valores[4];
valores[4] = valores[3];
valores[3] = t;
// valores = {1, 2, 10, 9, 12}

int t = valores[3];
valores[3] = valores[2];
valores[2] = t;
// valores = {1, 2, 9, 10, 12}
```

Quais variáveis vamos precisar para o laço? O índice `i` do elemento que está fora de ordem.

Como as variáveis são inicializadas? `i = valores.size() - 1`.

Qual a condição para o laço continuar a execução? `i > 0` e `valores[i - 1] > valores[i]`.

O que é feito a cada iteração? O elemento na posição `i` é trocado de lugar com o elemento na posição `i - 1` e `i` é decrementado.

Implementação

```
void insere_ordenado(vector<int> &valores, int v) {
    valores.push_back(v);
    int i = valores.size() - 1;
    while (i > 0 && valores[i - 1] > valores[i]) {
        int t = valores[i];
        valores[i] = valores[i - 1];
        valores[i - 1] = t;
        i = i - 1;
    }
}
```

Podemos melhorar?

```
void insere_ordenado(vector<int> &valores, int v) {
    valores.push_back(v);
    for (int i = valores.size() - 1; i > 0 && valores[i - 1] > valores[i]; i = i - 1) {
        int t = valores[i];
        valores[i] = valores[i - 1];
        valores[i - 1] = t;
    }
}
```