Tutorial: Generalizando Funções e Estruturas de Dados

Rodrigo Cezar Silveira

Introdução

Um ponteiro nulo é capaz de referenciar qualquer tipo de dados, primitivos ou compostos (struct), sendo um recurso poderoso quando se trata da construção de funções e estruturas de dados genéricas. Tendo domínio sobre estes princípios, bastaria a nós, construir uma única vez determinada função ou estrutura de dados, a qual seria capaz de manipular dados de qualquer tipo, inclusive, dados heterogêneos (variáveis de diferentes tipos de dados ao mesmo tempo), dessa forma poderíamos utilizá-la para os mais diversos propósitos.

Em contrapartida, esta maior flexibilidade, exige um esforço adicional: a desvantagem do uso de ponteiros de tipo void reside sobre a impossibilidade de os desreferenciarmos diretamente, fato que nos obriga a primeiramente realizar uma conversão de tipo (cast) para transformá-lo no tipo de dados desejado, para só então desreferenciá-lo.

Este documento não se trata de um guia completo sobre ponteiros para tipo void, mas apenas elucida brevemente os conceitos básicos para sua utilização na implementação de TADs genéricas. Embora este texto tenha sido escrito da forma mais “enxuta” o possível, maiores informações podem ser encontradas diretamente no código-fonte de exemplo.

Ponteiro Nulo, O Bêabá da Generalização

Um ponteiro nulo é declarado da seguinte forma:

```
1. void* p;
```

Este se comporta como um ponteiro ordinário, porém pode referenciar variáveis de qualquer tipo, inclusive tipos compostos. Abaixo encontramos um exemplo simples, utilizando uma variável inteira e uma variável alfanumérica:

```
1. void* p;
2. int i;
3. char c;
4. p = &i;
5. p = &c;
6. ...
```

Como dito anteriormente, ponteiros nulos não podem ser desreferenciados diretamente, em outras palavras, isto significa que não temos acesso direto à seu conteúdo, mas devemos primeiramente efetuar uma conversão de tipo, para então o desrefenciar.

Dessa forma, ao trabalharmos com ponteiros nulos devemos respeitar o seguinte procedimento antes de acessarmos seu conteúdo:

1- Converter o ponteiro nulo para um ponteiro de tipo desejado; por exemplo se desejamos armazenar o conteúdo de uma variável inteira devemos convertê-lo para um ponteiro para inteiros <<int*>>.

2- Uma vez convertido, o ponteiro nulo agora passa a ser um ponteiro de algum tipo específico e dessa forma podemos desreferenciá-lo para acessar ou definir seu conteúdo.

Exemplo:

```
1. char c;
2. void* p;
3. char* p_aux_char;
4. p = &c;
5. p_aux_char = (int*)p;
6. *p_aux_char = ‘m’;
```

Poderíamos ainda realizar o mesmo procedimento de uma forma mais resumida, realizando a conversão de tipo e o desreferenciamento em uma única etapa:

```
1. char c;
2. void* p;
3. p = &c;
4. *((int*)p)  = ‘m’;
```

Exemplo simples sobre uso de ponteiro nulo

Aliando Ponteiro Nulo, Funções e Alocação Dinâmica

Funções podem tanto retornar um ponteiro nulo como também recebê-los como argumento, não estando limitadas à trabalhar apenas com tipos específicos de dados.

```
1. void* func(void* a, void* b){
2.         …
3.         return a;
4. }
```

As funções de alocação dinâmica podem ser utilizadas seguindo os mesmos procedimentos aprendidos durante as disciplinas de Estruturas de Dados, de forma similar como quando trabalhamos com tipos de dados específicos. O seguinte exemplo aborda como poderíamos alocar um vetor de 100 posições ou para uma variável de 100 bits:

```
1. void* p;
2. p = malloc(100);
3. ...
4. free(p);
```

Com isso podemos nos beneficiar deste recurso para a construção de estruturas de dados genéricas, por exemplo, a estrutura abaixo poderia ser utilizada para criar uma árvore binária, a qual poderia armazenar variáveis de qualquer tipo em seu campo *data*.

```
1. typedef struct SNode{
2.         void *data;
3.         struct SNode *left;
4.         struct SNode *right;
5. } Node;
```

Exemplo de uma lista encadeada genérica

De uma forma ainda mais radical, poderíamos definir as sub-árvores à direita e à esquerda como também sendo um ponteiro para void, dessa forma possuiríamos uma árvore binária a qual de certa forma deixaria de ser uma árvore, mas poderia armazenar variáveis de qualquer tipo em seus nós, até mesmo outras estruturas de dados! Tal abordagem é desaconselhada visto o esforço necessário para realizar as devidas conversões de tipo, e aqui fica a dúvida se tal propriedade é apenas mais uma brecha para a criação de gambiarras ou se uma vez utilizada da maneira correta poderia se tornar um recurso realmente poderoso.

```
1. typedef struct SNode{
2.         void *data;
3.         void *left;
4.         void *right;
5. } Node;
```

E então chegamos ao fim deste tutorial, espero ter lhe ajudado!