



Estruturas de Dados
Aula 17:
Estruturas Genéricas

12/08/2013

Fontes Bibliográficas



- Livro:
 - Introdução a Estruturas de Dados (Celes, Cerqueira e Rangel): **Capítulo 14;**
- Slides baseados no material da PUC-Rio, disponível em <http://www.inf.puc-rio.br/~inf1620/>.

Estruturas Genéricas: Motivação



- Estruturas que vimos até agora são específicas para o tipo de informação que manipulam
- Por exemplo, vimos listas *de inteiros*, *de caracteres* e *de estruturas compostas*
- Para manipular cada um destes tipos, algumas funções do TAD devem ser reimplementadas
- Por exemplo, a função *Pertence*
 - Lista de caracteres (compara caracteres)
 - Lista de inteiros (compara inteiros)
- Função *Imprime*
 - Lista de caracteres (imprime caracter: "%c")
 - Lista de inteiros (imprime inteiro: "%d")

Estruturas Genéricas: Objetivos



- Uma estrutura genérica deve ser capaz de armazenar qualquer tipo de informação
- Para isso, um TAD genérico deve desconhecer o tipo da informação
- As funções do TAD genérico não podem manipular diretamente as informações
- As funções são responsáveis pela manutenção e encadeamento das informações

Cliente do TAD Genérico



- O cliente de um TAD Genérico fica responsável pelas operações que envolvem acesso direto à informação
- Por exemplo, o cliente do TAD lista genérica
 - Se o cliente deseja manipular inteiros, precisa implementar operações para manipular inteiros
 - Se o cliente deseja manipular caracteres, precisa implementar operações para manipular caracteres

Lista Genérica



- Uma célula da lista genérica guarda um ponteiro para informação que é genérico (void*). Por que?

```
struct listagen {  
    void* info;  
    struct listagen* prox;  
};  
typedef struct listagen ListaGen;
```

Lista Genérica (cont.)



- As funções do TAD lista que não manipulam informações (cria lista, verifica se está vazia) são implementadas da mesma maneira
- Funções com objeto opaco
 - Função que insere uma nova célula na lista
 - Cliente passa para função um ponteiro para a informação

```
ListaGen* lgen_inserere (ListaGen* l, void* p)
{
    ListaGen* n = (ListaGen*) malloc(sizeof(ListaGen));
    n->info = p;
    n->prox = l;
    return n;
}
```

Lista Genérica (cont.)



- Problema surge nas funções que precisam manipular as informações contidas em uma célula
 - Função libera? Cliente fica responsável por liberar as estruturas de informação
 - Função pertence? TAD não é capaz de comparar informações.
- Solução: TAD deve prover uma função genérica para percorrer todos os nós da estrutura.
- Precisamos ter um mecanismo que permita, a partir de uma função do TAD, chamar o cliente => **Callback** (*"chamada de volta"*)

Callback



- Função genérica do TAD lista é a função que percorre e visita as células
- A operação específica a ser executada na célula (comparação, impressão, etc) deve ser passada como parâmetro
 - Função como parâmetro? *Ponteiro para Função!*
- O nome de uma função representa o endereço dessa função

Callback (cont.)



- Exemplo

- Assinatura da função de callback

```
void callback (void* info) ;
```

- Declaração de variável ponteiro para armazenar o endereço da função

```
void (*cb) (void*) ;
```

- cb: variável do tipo ponteiro para funções com a mesma assinatura da função callback

Uso de callback



- Ex. Função genérica para percorrer as células da lista
 - Chama a função de callback para cada célula visitada

```
void lgen_percorre (ListaGen* l, void (*cb) (void*))  
{  
    ListaGen* p;  
    for (p=l; p!=NULL; p=p->prox) {  
        cb(p->info);  
    }  
}
```

Cliente:

```
...  
lgen_percorre (l, callback);  
...
```

Exemplo de Cliente



- Exemplo de aplicação cliente que armazena pontos (x,y)

```
struct ponto {  
    float x, y;  
};  
typedef struct ponto Ponto;
```

Exemplo de Cliente (cont.)



- Para inserir pontos na lista genérica
 - cliente aloca dinamicamente uma estrutura do tipo Ponto
 - passa seu ponteiro para a função de inserção
 - cliente implementa função auxiliar para inserir pontos (x,y) na estrutura da lista genérica

```
static ListaGen* insere_ponto (ListaGen* l, float x, float
    y)
{
    Ponto* p = (Ponto*) malloc(sizeof(Ponto));
    p->x = x;
    p->y = y;
    return lgen_inserere(l,p);
}
```

Exemplo de Cliente (cont.)



- Para imprimir os pontos de uma lista genérica
 - Cliente converte ponteiro genérico (void*) em Ponto (type cast)
 - Imprime informação

```
static void imprime (void* info)
{
    Ponto* p = (Ponto*)info;
    printf("%f %f", p->x, p->y);
}
```

Exemplo de Cliente (cont.)



- Se o programa cliente deseja imprimir os pontos que estão em uma lista
 - Usa a função genérica do TAD lista `lgen_percorre`

...

```
lgen_percorre (l, imprime)
```

...

Exemplo de Cliente (cont.)



- Callback para cálculo do centro geométrico dos pontos armazenados na lista:
 - atualiza variáveis globais a cada chamada da *callback*:
 - NP: tipo int representa o número de elementos visitados
 - CG: tipo Ponto representa o somatório das coordenadas

```
static void centro_geom (void* info)
{
    Ponto* p = (Ponto*)info;
    CG.x += p->x;
    CG.y += p->y;
    NP++;
}
```


Exemplo de Cliente (cont.)



- Cálculo do centro geométrico dos pontos pelo cliente:
 - Usa a função genérica *lgen_percorre* passando o endereço da função *centro_geom* como parâmetro

...

```
NP = 0;
```

```
CG.x = CG.y = 0.0f;
```

```
lgen_percorre (1, centro_geom);
```

```
CG.x /= NP;
```

```
CG.y /= NP;
```

...

Passando dados para o callback



- Devemos evitar variáveis globais
 - Pode tornar o programa difícil de ler e difícil de manter
- Para evitar o uso de variáveis globais, precisamos de mecanismos que permitam passagem de informações do cliente para a função de callback
 - utiliza parâmetros da callback:
 - informação do elemento sendo visitado
 - ponteiro genérico com um dado qualquer
 - cliente chama a função de percorrer passando como parâmetros
 - a função *callback*
 - o ponteiro a ser repassado para a *callback* a cada elemento visitado

Passando dados para o callback



- Função genérica para percorrer os elementos da lista
 - utiliza assinatura da função *callback* com dois parâmetros

```
void lgen_percorre(ListaGen* l,
    void(*cb)(void*,void*), void* dado)
{
    ListaGen* p;
    for (p=l; p!=NULL; p=p->prox) {
        cb(p->info,dado);
    }
}
```

Passando dados para o callback



- Modificando a função para calcular o centro geométrico dos pontos (não precisa de variáveis globais)
 - passo 1: criação de um tipo que agrupa os dados para calcular o centro geométrico:
 - número de pontos
 - coordenadas acumuladas

```
struct cg
{
    int n; // número de pontos analisados
    Ponto p; // "soma" dos pontos
};
typedef struct cg Cg;
```

Passando dados para o callback



- Modificando a função para calcular o centro geométrico dos pontos (não precisa de variáveis globais)
 - passo 2: re-definição da *callback* para receber um ponteiro para um tipo Cg que representa a estrutura

```
static void centro_geom (void* info, void* dado)
{
    Ponto* p = (Ponto*)info;
    Cg* cg = (Cg*)dado;
    cg->p.x += p->x;
    cg->p.y += p->y;
    cg->n++;
}
```

Passando dados para o callback



- Chamada do cliente

...

```
Cg cg = {0, {0.0f, 0.0f}};
```

```
lgen_percorre(1, centro_geom, &cg);
```

```
cg.p.x /= cg.n;
```

```
cg.p.y /= cg.n;
```

...

Retornando valores de callback



- Função pertence
 - Retorna 1 se o ponto existe na lista e 0 caso contrário
- Poderíamos implementar uma função callback que receberia como dado o ponto a ser pesquisado
 - Usando a função genérica *percorre* como implementada até agora, a lista inteira seria percorrida, mesmo já encontrando o elemento.
- Solução
 - Mecanismo para interromper o percurso
 - Valor de retorno!!!!
 - =0 função deve prosseguir visitando o próximo elemento
 - !=0 função deve interromper a visitação aos elementos

Retornando valores de callback



- Modificando a função percorre para permitir interrupção

```
int lgen_pcorre (ListaGen* l, int
    (*cb) (void*,void*), void* dado)
{
    ListaGen* p;
    for (p=l; p!=NULL; p=p->prox) {
        int r = cb(p->info,dado);
        if (r != 0)
            return r;
    }
    return 0;
}
```


Retornando valores de callback



- Função igualdade...

```
static int igualdade (void* info, void* dado)
{
    Ponto* p = (Ponto*)info;
    Ponto* q = (Ponto*)dado;
    if (fabs(p->x-q->x)<TOL && fabs(p->y-q->y)<TOL)
        return 1;
    else
        return 0;
}
```

Retornando valores de callback



- Função Pertence...

```
static int pertence (ListaGen* l, float x, float y)
{
    Ponto q;
    q.x = x;
    q.y = y;
    return lgen_percorre(l, igualdade, &q);
}
```

Concluindo...



- Exemplo de callback stdlib
- Callbacks permitem o uso de TAD's genéricos
- Callbacks são muito empregados em programação