


Laboratório de Engenharia de Software

Aula 10

Implementação da programação modular 1

Arndt von Staa
LES/DI/PUC-Rio
Agosto 2008

Especificação



Laboratório de Engenharia de Software

- Objetivo dessa aula
 - Apresentar os conceitos relacionados com espaços de dados e tipagem de espaços de dados.
- Referência básica:
 - Capítulo 6 do livro texto


Ago 2008

Arndt von Staa © LES/DI/PUC-Rio

2 / 38

Laboratório de Engenharia de Software

Sumário



- Espaços de dados
- Nomes de espaços de dados, amarração
- Referências, funções de acesso
- Tipos de dados
- Tipagem de espaços
- Imposição de tipos
- Conversão de tipos


Ago 2008

Arndt von Staa © LES/DI/PUC-Rio

3 / 38

Laboratório de Engenharia de Software

Onde são armazenados os dados?



- São armazenados em **espaços de dados**
 - **dados**
 - **código**
 - não deixa de ser uma forma de dado


Ago 2008

Arndt von Staa © LES/DI/PUC-Rio

4 / 38

Laboratório de Engenharia de Software

Quais são os atributos de espaços de dados?




- Espaços de dados **contíguos**
 - são armazenados em um **meio de armazenamento**, exemplos:
 - **memória real** (principal)
 - **memória virtual**
 - para todos os processos um único grande arquivo contém todos os dados que estão em memória
 - » quando necessário os dados são transferidos para memória real
 - cada processo tem uma origem nesse arquivo
 - endereço é dado por `< idProcesso , inxPagina , Deslocamento >`
 - **memória virtual segmentada**
 - arquivos (segmentos) contém os dados que irão para (ou estão em) memória real
 - cada processo identifica $n \geq 1$ segmentos (arquivos)
 - endereço é dado por `< idprocesso , idSegmento , inxPagina , Deslocamento >`
 - **arquivos**
 - **máquinas remotas**

Ago 2008
Arndt von Staa © LES/DI/PUC-Rio
5 / 38

Laboratório de Engenharia de Software

Quais são os atributos de espaços de dados?



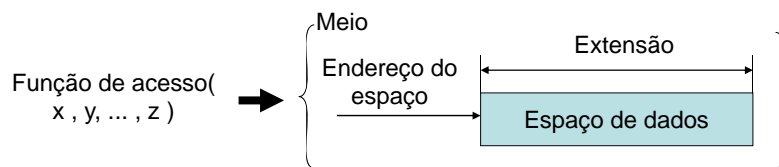
- Espaços de dados **contíguos**
 - possuem uma **extensão**
 - espaço ocupado em bytes
 - `sizeof(Espaco)` é a função que informa a extensão do espaço
 - `sizeof(int)` é igual a o que?
 - independe da máquina?
 - dado: `long vt[100] ;`
 - `sizeof(vt)` é igual a o que?
 - dado: `typedef struct { int a ; char b ; } tpX ;`
 - `sizeof(tpX)` é igual a o que?

Ago 2008
Arndt von Staa © LES/DI/PUC-Rio
6 / 38

Quais são os atributos de espaços de dados?



- Espaços de dados **contíguos**
 - possuem um **endereço real** (virtual) da sua origem
 - são acessados por meio de uma **função de acesso**
 - a forma mais simples dessa função é o **endereço físico do espaço** em memória real (virtual)
 - pode ser uma expressão computacional simples, exemplo
 - `vtAlgo[inxAlgo]`
 - pode ser uma expressão complexa, exemplo:
 - `TabSimb[ObterHash(Simbolo)]->ValorElemento`
 - acessa o espaço ocupado pelo valor do 1o. elemento da lista de colisão de `Simbolo` contido em um tabela de randomização (*hash table*) `TabSim`



Ago 2008

Arndt von Staa © LES/DI/PUC-Rio

7 / 38

Linguagens e espaços de dados



- Sempre que se utiliza uma **linguagem de programação simbólica**, espaços podem ser referenciados por um **nome**, exemplos:
 - nome de uma variável
 - nome de uma constante
 - nome de uma função
- Um nome é, portanto, um parâmetro para uma função de acesso

Ago 2008

Arndt von Staa © LES/DI/PUC-Rio

8 / 38

Linguagens e espaços de dados



- **Amarração** (*binding*) é a operação de associar um nome ou uma função de acesso a um espaço de dados
 - pode ser realizado pelo compilador, exemplos:
 - `int x ;`
 - `int f(int x) { return x * x ; }`
 - pode ser realizado por programa, exemplos:
 - `pX = malloc(dimX) ;`
 - `pV = &V ;`

O que são referências?



- Uma **referência** é formada por um conjunto de **parâmetros** a serem fornecidas a uma **função de acesso** para acessar um determinado espaço
 - exemplo: elemento de um vetor em C
 - < `origemVetor` , `dimElemento` , `inxElemento` >
 - exemplo: disco físico
 - < `unidade` , `cilindro` , `trilha` , `setor` >
 - exemplo: arquivo
 - < `pArq` , `inxByte` , `idOrigem` >
- Cada **classe de referência** possui uma função de acesso associada a ela
 - exemplo vetor: `tpX vtX[dimVtX] ; ... vtX[i] ...`
 - exemplo disco físico: um serviço do sistema operacional
 - exemplo arquivo: `fseek(, ,)`

Dereferenciação



- **Dereferenciar** é a operação de converter uma referência em um endereço real através de sua função de acesso, exemplos: (obs.: não são fragmentos de código em C)

– `A[j] :: &A + j * sizeof(tpA)` – tudo medido em bytes

– `*pX :: [pX]` ou por extenso: conteúdo de `pX`

– seja: `pElemTabSimb * ObterElemTabSimb(char * pszSimbolo)`

• `*(ObterElemTabSimb("um_simbolo"))`

– é o espaço de dados associado ao símbolo "um_simbolo"

operador de dereferenciação de ponteiro

expressão gera um ponteiro

- `::` operador "é definido por"

Ago 2008

Arndt von Staa © LES/DI/PUC-Rio

11 / 38

Dereferenciação



- A dereferenciação pode ser realizada até chegar ao valor

`expA = expB`

LHS

left hand side

RHS

right hand side

atribui-se o valor RHS ao endereço LHS

busca-se o valor referenciado pelo endereço RHS

i.e. **dereferencia-se** o endereço RHS

Ago 2008

Arndt von Staa © LES/DI/PUC-Rio

12 / 38

Dereferenciação composta



- Assumindo que os espaços de dados referenciados existam:

```
typedef struct tgElemLista
{
    char          szSimbolo[ DIM_SIMBOLO ] ;
    unsigned      IdSimbolo ;
    struct tgElemLista * pProx;
} tpElemLista ;
tpElemLista * Tabela[ DIM_TABELA ] ;

( Tabela[ ObterHash( szSimboloDado ) ]->pProx)->szSimbolo
OU
*(*( Tabela[ ObterHash( szSimboloDado ) ] ).pProx ).szSimbolo
```

- acessam o mesmo vetor de caracteres – o segundo elemento da lista

Ago 2008

Arndt von Staa © LES/DI/PUC-Rio

13 / 38

Ponteiros



- Ponteiros são **casos especiais** de referências
 - função de acesso é implícita: ***pX** :: conteúdo de **pX**
- Há quem prefira outra definição de modo que se caracterize o controle que se pode ter quando se usa referências, contrastando com a falta de controle quando se usa ponteiros, exemplo

```
- vtA[ inxA ] ::
    • if ( ( inxA < 0 ) || ( inxA >= dimA ) )
        ErroAcesso( __FILE__ , __LINE__ ) ;

- pA[ inxA ] :: ??? pois não se sabe o valor de dimA
- pA + inxA  :: ??? idem
```

Ago 2008

Arndt von Staa © LES/DI/PUC-Rio

14 / 38

Espaços aninhados



- Aninhamento homogêneo, vetores, matrizes

- Sejam:

```
typedef struct { ... } tpA ;  
tpA vtA[ dimA ]
```

- Qual é o `sizeof` de:

- `vtA[inxA]` ?
 - `sizeof(tpA)`
- `*vtA` ?
 - `sizeof(tpA)` corresponde a `vtA[0]`
- `vtA + inxA` ?
 - `sizeof(tpA)` corresponde a `vtA[inxA]`
- `vtA` ?
 - `sizeof(tpA) * dimA`

Ago 2008

Arndt von Staa © LES/DI/PUC-Rio

15 / 38

Espaços aninhados



- Aninhamento heterogêneo

- estruturas (`struct`) podem conter elementos que por sua vez também são estruturas

- Exemplo de `structs` aninhados em C

```
typedef struct  
{  
    char Nome[ DIM_NOME ] ;  
    char SobreNome[ DIM_NOME ] ;  
} tpNome ;
```

```
typedef struct  
{  
    tpNome Nome ;  
    tpEndereco Endereco ;  
} tpPessoa ;
```

Ago 2008

Arndt von Staa © LES/DI/PUC-Rio

16 / 38

Espaços de dados compostos



- Espaços de dados podem ser
 - **simples**: todo o valor está contido em um único espaço contíguo
 - variável
 - struct
 - vetor
 - lista contida em vetor
 - **compostos**: o valor completo requer vários espaços de dados, simples ou compostos, para ser armazenado
 - lista encadeada
 - árvore
 - base de dados


Espaços de dados compostos



- Espaços de dados compostos podem ser **tratados como um todo** através de um **nome**, exemplos
 - nome associado à **cabeça do espaço de dados composto**
 - nome simbólico que referencia um arquivo
 - nome ODBC que referencia uma base de dados como um todo
 - *handle* que identifica uma janela

Laboratório de Engenharia de Software

Espaços compostos com cabeça, exemplos




- Lista
- Árvore binária codificando árvore n-ária

Ago 2008
Arndt von Staa © LES/DI/PUC-Rio
19 / 38

Laboratório de Engenharia de Software

Espaços de dados compostos



- Considere uma árvore, qual a diferença entre uma árvore com cabeça e uma árvore identificada pela referência a sua raiz?
 - *Sem cabeça*: o projeto da estrutura estará visível para os clientes,
 - *com cabeça*: encapsula a implementação
 - *Sem cabeça*: se for permitido que várias funções compartilhem uma mesma árvore, esta não poderá ficar vazia, tampouco poderá receber uma nova raiz
 - *com cabeça*: todas as referências são para a cabeça
 - *Com cabeça*: dificulta a exploração
 - *sem cabeça*: percorrimento recursivo convencional
 - *Com cabeça*: facilita a implementação de operadores que atuem sobre a árvore como um todo, exemplos:
 - cópia
 - destruição
 - percorrimento usando um iterador de percorrimento

Ago 2008
Arndt von Staa © LES/DI/PUC-Rio
20 / 38

O que é um tipo de dados?



- Do ponto de vista de quase todos os computadores o conteúdo de um espaço de dados é amorfo
 - sequência de bytes, mais precisamente sequência de bits
- A interpretação do conteúdo do espaço implica o código a ser executado ao acessá-lo, exemplos
 - $1 + 5 \rightarrow$ integer add
 - $1.0 + 5 \rightarrow$ floating point add, após converter 5 para 5.0
- O tipo de um espaço de dados determina como interpretar o seu conteúdo:
 - a organização de baixo nível deste espaço
 - conjunto de elementos, caso existam (**struct**)
 - o tamanho em bytes do todo e dos elementos

Ago 2008

Arndt von Staa © LES/DI/PUC-Rio

21 / 38

Tipos de dados



- Tipos computacionais são definidos pela linguagem de programação
 - int
 - char
 - char *
- Tipos do usuário são declarados pelo programador. Em C:
 - enum
 - struct
 - union
 - typedef
 - **typedef** é na realidade uma espécie de **#define** que conhece a estrutura de uma declaração de tipo do usuário

Ago 2008

Arndt von Staa © LES/DI/PUC-Rio

22 / 38

Espaços de dados contíguos e tipos



- O **tipo** de um espaço de dados contíguo determina **como deve ser interpretado** o seu conteúdo
 - o tipo pode ser **declarado explicitamente** através de código de declaração
 - C, C++, Java
 - o tipo pode ser **declarado implicitamente** por contexto
 - Scheme, Lua
 - funções C para as quais não se tenha declarado um protótipo
 - o tipo pode ser **determinado através de um atributo** contido no espaço de dados
 - identificador do tipo
 - Lua
 - tipos dinâmicos em C++ e Java
 - o tipo pode ser estabelecido **através do código** usado
 - assembler

Ago 2008

Arndt von Staa © LES/DI/PUC-Rio

23 / 38

Dados do tipo union



- **union** define **tipos alternativos** para um mesmo espaço de dados
 - possibilita acessar uma mesma área de dados utilizando diferentes tipos de um conjunto explicitamente especificado

```
typedef union tagNome
{
    tipo1 NomeInterpretação1 ;
    tipo2 NomeInterpretação2 ;
    . . .
    tipon NomeInterpretaçãon ;
} tpNome ;
```
- ao acessar o espaço de dados usando **NomeInterpretação_i** será utilizado o tipo **tipo_i** para interpretar o espaço
 - o tamanho do espaço de dados da **union tagNome** será igual ao maior dos tamanhos dos tipos
 - o uso descuidado de **unions** pode trazer consequências desastrosas

Ago 2008

Arndt von Staa © LES/DI/PUC-Rio

24 / 38

Alinhamento de dados em memória



```
union
{
    char VetCh[ 9 ] ;           opção 1
    short int VetInt[ 3 ] ;     opção 2
    struct
    {
        char IdCh ;
        long IdLong ;
        short IdInt ;
    } Desalinhado ;           opção 3
    struct
    {
        long IdLong ;
        short IdInt ;
        char IdCh ;
    } Alinhado ;              opção 4
}
```

Alinhado a byte

	0	1	2	3	4	5	6	7	8
VetCh									
VetInt									
DesAlinhado									
Alinhado									

Alinhado a margem 4

	0	1	2	3	4	5	6	7	8	9	10	11
VetCh												
VetInt												
DesAlinhado												
Alinhado												

Ago 2008

Arndt von Staa © LES/DI/PUC-Rio

25 / 38

Imposição de tipos



- Podem-se forçar diferentes interpretações para um mesmo espaço de dados através da **imposição de tipos** (*type casts*)
- uma imposição de tipos é a uma **union** na qual o **conjunto dos possíveis tipos não é definido a priori**, exemplo

```
( short int * ) pEspaco
```

 - estabelece que o espaço de dados designado por **pEspaco** deve ser interpretado como um ponteiro para um **short int** independentemente do tipo com que foi declarado

Ago 2008

Arndt von Staa © LES/DI/PUC-Rio

26 / 38

Imposição de tipos



- A imposição de tipos em C **pode violar as regras de controle de tipos**, por exemplo
 - no código a seguir “vetor de 4 caracteres” recebe a interpretação “vírgula flutuante” o que provavelmente não faz sentido algum

```
char VetorChar[ 4 ] = "abcd" ;  
*((float *) VetorChar )
```

Ago 2008

Arndt von Staa © LES/DI/PUC-Rio

27 / 38

Imposição de tipos: tipo definido no dado



- Por **padrão de programação** o primeiro atributo de qualquer **struct** poderia ser do tipo **tpIdTipo** e ter o nome **idTipo**
 - O valor de **idTipo** corresponde ao tipo efetivo do espaço de dados
 - é atribuído ao criar uma instância da estrutura
- Ao acessar um espaço (**void ***) **pEspaco** executa-se

```
switch( *(( tpIdTipo * ) pEspaco ))  
{  
    case idTipoA : ...  
        break ;  
    case idTipoB : ...  
        break ;  
    default:  
        tratar erro de espaço com tipo desconhecido  
        break ;  
} /* switch */
```

Ago 2008

Arndt von Staa © LES/DI/PUC-Rio

28 / 38

Imposição de tipos



- Ao programar em C++ utilize as construções de imposição de tipo que a linguagem oferece ao invés da sintaxe C apresentada anteriormente:
 - `const_cast< tipo >(expressão)`
 - elimina a restrição `const` de um parâmetro
 - `static_cast< tipo >(expressão)`
 - pode ser usado para *up cast*, não é seguro no caso de *down cast*
 - `dynamic_cast< tipo >(expressão)`
 - pode ser utilizado de forma segura para impor tipos satisfazendo as restrições de herança existentes
 - `reinterpret_cast< tipo >(expressão)`
 - vale tudo, deve ser utilizado com muito cuidado

Imposição de tipos



- Em Java existe uma forma limitada e segura de imposição de tipos
 - impor a um objeto o tipo de uma de suas superclasses
 - sempre vale
 - impor a um objeto o tipo de uma sub-classe (herdeira)
 - neste caso Java verifica se o objeto foi construído a partir da classe herdeira, a imposição de tipo vale somente neste caso
 - similar ao `dynamic_cast` de C++

Imposição de tipos, exemplo



```
#include <stdio.h>
#include <memory.h>
void main( void)
{
    char Vet[ 5 ] ;
    int c0, c1, c2, c3 ;
    /* Definir o valor contido no espaço de dados */
    memset( Vet , 0 , sizeof( Vet ) ) ;
    memcpy( Vet , "ABCD" , 4 ) ;
    /* Exibir várias interpretações do espaço de dados */
    printf( "\nlong hexa  %lx" , *((long * ) Vet) ) ;
    printf( "\nshort hexa %4x %4x" ,
            *(( short int * ) Vet ) ,
            *(( short int * ) &( Vet[ 2 ])) ) ;

    c0 = Vet[ 0 ] ;
    c1 = Vet[ 1 ] ;
    c2 = Vet[ 2 ] ;
    c3 = Vet[ 3 ] ;
    printf( "\nchar hexa  %2x %2x %2x %2x" ,
            c0, c1, c2, c3 ) ;
    printf( "\nlong int   %li" , *((long * ) Vet) ) ;
    printf( "\nfloat      %f" , *((float * ) Vet) ) ;
    printf( "\nstring     %s" , Vet ) ;
}
```

Ago 2008

Arndt von Staa © LES/DI/PUC-Rio

31 / 38

Imposição de tipos, exemplo



- Resultados da execução (em máquina Intel):

```
long  hexa  44434241
short hexa  4241 4443
char   hexa  41 42 43 44
long  int   1145258561
float           781.035217
string        ABCD
```

Porque estas inversões de ordem?

- Dependendo da máquina (Intel, Motorola, etc.) o resultado será diferente
- Se o conteúdo de Vet fosse **executado** em uma máquina Intel, seriam executadas as 4 instruções:

```
INC CX
INC DX
INC BX
INC SP
```


Ago 2008

Arndt von Staa © LES/DI/PUC-Rio

32 / 38

Laboratório de Engenharia de Software

Quando usar imposição de tipos



- Justifica-se o uso de imposição de tipos irrestrita ao implementar funções de gerenciamento de memória


```
tpMeuTipo * ptMeuTipo ;


. . .

ptMeuTipo = ( tpMeuTipo * ) malloc(sizeof(tpMeuTipo)) ;
```
- Ao programar em C++ use sempre os operadores **new** e **delete** e jamais as funções **malloc** e **free** mesmo para **structs**

Ago 2008
Arndt von Staa © LES/DI/PUC-Rio
33 / 38

Laboratório de Engenharia de Software

Quando usar imposição de tipos



- Ao desenvolver módulos genéricos muitas vezes é necessário utilizar **referências** para dados **de tipos desconhecidos**
 - frequentemente isto requer ponteiros para o **tipo indefinido** (**void ***)
 - em Java o tipo indefinido seria **object**
 - em C++ pode-se criar uma estrutura de herança
- Existem mecanismos que permitem evitar o uso de **void ***
 - No módulo de definição **zzz.h** inclua exatamente:



```
typedef struct tagXXX * tpXXX ;
```
 - No módulo de implementação **zzz.c** inclua a implementação:


```
struct tagXXX
{
    . . .
} ;
```

Ago 2008
Arndt von Staa © LES/DI/PUC-Rio
34 / 38

Laboratório de Engenharia de Software

Imposição versus conversão de tipos




- Imposição de tipos e **conversão** de tipos são operações distintas.
 - a imposição de tipos **muda simplesmente a interpretação** do espaço de dados, sem alterar o seu conteúdo
 - consiste meramente em uma instrução para o compilador
- Na conversão, opera-se sobre o conteúdo do espaço de dados
 - transforma o valor no tipo origem para um valor no tipo destino
 - **preserva a semântica do valor**

Ago 2008
Arndt von Staa © LES/DI/PUC-Rio
35 / 38

Laboratório de Engenharia de Software

Imposição versus conversão de tipos



- Por exemplo ao converter um valor **inteiro** para um valor **vírgula flutuante**
 - usando aritmética de inteiros e manipulação de bits, computa-se uma representação flutuante com o mesmo valor do inteiro
 - ao terminar, atribui-se o valor resultante a uma variável flutuante usando imposição de **float** ao resultado
- Pode-se converter uma **árvore binária** para um **string**
 - percorrer a árvore em ordem pre-fixada pela esquerda
 - a cada nó visitado concatenar o valor do nó seguido de um separador
 - a cada referência **NULL** para sub-árvore concatenar **NIL** seguido do mesmo separador
- O **string** assim formado pode ser convertido de volta para a mesma árvore – façam o experimento ©

Ago 2008
Arndt von Staa © LES/DI/PUC-Rio
36 / 38

Cuidados com a imposição de tipos



- Evite o uso de imposições irrestritas de tipos
 - Quando utilizar
 - justifique muito bem
 - necessidade
 - corretude
 - inclua comentários no programa contendo esta justificativa
- Evite de todas as maneiras o uso de imposição de tipos **indiscriminada** ao programar **orientado a objetos**
 - vale impor para superclasse, é automático
 - vale impor para classe herdeira somente se tiver certeza que objeto havia sido criado na subclasse imposta ou em uma classe herdeira dela
 - use sempre `dynamic_cast` em C++



FIM