Resolução dos Exercícios do Capítulo III

1. Ponteiros são causadores potenciais de erros em programação. Dê exemplos, com trechos de código em C, de erros causados por ponteiros que provocam violação dos sistemas de tipos da linguagem, ocorrência de objetos pendentes e ocorrência de referências pendentes.

Violação dos sistemas de tipos da linguagem:

```
int a = 33;
float b = 3.12;
int *t = &a;
t++;    // t não aponta mais para um int
*t = 10;          // vai alterar o valor de b
```

Ocorrência de objetos pendentes:

```
int *d, *p;
d = (int *)malloc(20*sizeof(int));
p =(int *)malloc(10*sizeof(int));
d=p;    // área alocada inicialemente para d é perdida
```

Ocorrência de referência pendentes:

```
int *a, *b;
a =(int *)malloc(20*sizeof(int));
b = a;
free(a);  // b continua apontando para a área de memória desalocada.
```

2. Uma diferença significativa entre a definição de tipos primitivos cm C++ e JAVA se refere ao intervalo de valores de cada tipo. Enquanto em JAVA os intervalos foram fixados na definição da LP, em C++ é a implementação do compilador que define esses intervalos. Compare estas duas abordagens, justificando a opção de cada uma dessas linguagens.

A linguagem C deixa para os implementadores dos compiladores a definição dos intervalos dos tipos. Essa postura diminui a portabilidade dos programas, porém aumenta a sua eficiência. A eficiência é maior porque permite aos implementadores dos compiladores selecionar os intervalos de tipos de modo a utilizar melhor os recursos de hardware. Já na linguagem JAVA os intervalos foram fixados na definição da LP pois ela prioriza a portabilidade de seus programas em detrimento da eficiência de execução. Isso ocorre porque a definição dos intervalos de valores dos tipos no projeto da linguagem implica na necessidade de emulação em software das operações sobre esses tipos em alguns tipos de computadores.

3. Em geral, o uso de índice fora dos limites do vetor só pode ser verificado em tempo de execução. Algumas LPs, como JAVA, PASCAL e MODULA-2 fazem a verificação dinâmica dos índices. Outras, como C, C++ e FORTRAN não fazem essa verificação. Justifique porque essas LPs adotaram posturas opostas. Uma terceira postura, intermediária, seria gerar código com verificação dinâmica na fase de desenvolvimento e sem verificação dinâmica para a fase de uso. Discuta

essa opção em termos dos conceitos usados para justificar as opções das LPs mencionadas.

A verificação dinâmica dos índices garante que a execução do programa não continuará normalmente caso haja um acesso indevido, conferindo-lhe maior confiabilidade. Por outro lado, a necessidade de testar em tempo de execução se todo acesso ao vetor é apropriado implica em uma perda de desempenho na execução do programa.
Logo, as LPs que optam pela verificação dinâmica dos índices de vetores em tempo de execução, priorizam a confiabilidade do programa em detrimento do desempenho de execução. As LPs que não adotam essa postura, ou seja, que não verificam os índices de vetores dinamicamente, priorizam o desempenho dos programas em detrimento da confiabilidade.
A postura intermediária garante uma eficiência equivalente a das LPs que não fazem verificação dinâmica dos índices e apresentam uma maior confiabilidade do que essas LPs, uma vez que erros de acesso poderão ser mais facilmente identificados na fase de desenvolvimento. Por outro lado, a postura intermediária garante um desempenho de execução dos programas superior ao das LPs que verificam os índices dinamicamente, mas os programas se tornam menos confiáveis do que nessas LPs, uma vez que alguns erros de acesso podem não ser identificados na fase de desenvolvimento.

4. *Arrays* podem ser estáticos, semi-estáticos, semi-dinâmicos e dinâmicos. Enquanto a criação de *arrays* estáticos e semi-estáticos pode ser feita facilmente em C, a construção de *arrays* semi-dinâmicos e dinâmicos envolve um maior esforço de programação. Responda como os mecanismos de C permitem a criação destes tipos de *arrays*. Ilustre com exemplos.

Semi-dinâmico

C permite a criação desse tipo de vetores através do mecanismo de alocação dinâmica de memória associado ao uso de ponteiros.

```
char * p;
p = (char*) malloc(sizeof(char));
// uso normal de p como vetor
```

OBSERVAÇÃO: No padrão ISO mais recente de C (C99), vetores semi-dinâmicos podem ser alocados na pilha, como no exemplo seguinte:

```
void f (int t) {
  char p [n];
  // uso normal do vetor p com n elementos
```

Dinâmico

Como não é possível alterar o tamanho do vetor em tempo de execução em C, deve-se alocar uma outra área com o tamanho desejado e fazer a cópia

```
char * p,*q;
int i;
```

```
p = (int*) malloc (3*sizeof(int));
// uso normal de p como vetor
// necessario alocar mais memoria quando o vetor precisa aumentar de tamanho
q = (int*)malloc(6*sizeof(int));
// necessario copiar conteudo de p no inicio de q
for(i=0; i < sizeof(c); i++)  { *q++ = *p++; }
// necessario desalocar area de p apos a copia
free(p);
```

5. Produtos cartesianos, uniões, mapeamentos e tipos recursivos são categorias de tipos compostos de dados. Ilustre, com exemplos em C, cada um desses conceitos. Crie ainda um novo tipo de dados que combine três desses conceitos e diga qual a sua cardinalidade.

Produto cartesiano

```
struct produto{
        int codigo;
        char nome[20];
        float valor;
};
```

União

```
union temperatura{
        int kelvin;
        float celsius;
};
```

Mapeamento

```
int v[10];
```

Tipo recursivo

```
struct lista{
        int info;
        struct lista* prox;
}
```

Combinação

```
struct lista_produto{
        int codigo;
        char nome[20];
        union temperatura t ;
};
```

Cardinalidade

#int * (#char)^{20} * (#float + #int)

6. Determine a cardinalidade de cada um dos tipos abaixo, usando os conceitos de produto cartesiano, uniões e mapeamentos para explicar a cardinalidade dos tipos compostos:

```
enum sexo {masculino, feminino};
enum estado_civil {solteiro, casado, divorciado};
enum classe {baixa, media, alta};
enum instrucao {primario, secundario, superior};
union cidadania {
  enum classe c;
  enum instrucao i;
};
struct pessoa {
  enum sexo s;
  enum estado_civil e;
  union cidadania c;
};
struct amostra {
  int n;
  struct pessoa p[10];
};
```

A cardinalidade de cada tipo é dada na tabela a seguir:

Tipo	Cardinalidade
enum sexo	$\#S_1 = 2$
enum estado_civil	$\#S_2 = 3$
enum classe	$\#S_3 = 3$
enum instrução	$\#S_4 = 3$
union cidadania	$\#S_5 = (\#S_3 + \#S_4) = 6$
struct pessoa	$\#S_6 = (\#S_1 \times \#S_2 \times \#S_5) = 36$
struct amostra	$\#S_7 = \#int \times (\#S_6)^{10} = \#int \times (36)^{10}$

A cardinalidade de *union cidadania* foi calculada considerando que os valores das enumerações classe e instrução são distintos. Uma outra interpretação leva em conta que, na verdade, os valores de tipos enumerados são de fato valores inteiros em C. Nesse caso, haveria interseção nos valores desses dois tipos e a cardinalidade de *union cidadania* seria calculada por $\#S_5 = (\#S_3 + \#S_4 - \#(S_3 \cap S_4))$. Esse valor poderia ser igual a 3 (se considerarmos apenas o uso apropriado das enumerações ou #int se considerarmos que o programador pode atribuir valores inteiros às variáveis desses tipos).

7. Considere o seguinte programa escrito em C++:

```
#include <iostream>
int& xpto (int sinal) {
        int p = 4;
        if (!sinal) {
                p*=sinal;
   } else {
                p++;
```

```
        }
        return p;
}
void ypto () {
        int c[1000];
        int aux;
        for (aux = 0; aux < 1000; aux++) {
          c[aux] = aux;
        }
}
main() {
        int a = 1;
        int& b = xpto(a);
        ypto();
        cout << b;
}
```

Determine quais serão as saídas possíveis do programa acima. Explique sua resposta.

As saídas desse programa dependerão da forma como o código é gerado pelo compilador. Isso acontece porque no programa há ocorrência de referência pendente. Após *xpto* ser chamada, *b* fica referenciando a área de memória alocada para *p* em *xpto*, que a essa altura já foi desalocada. Quando *ypto* é chamada, a área referenciada por *b* é alocada pelo registro de ativação de *ypto*. Essa área poderá ser usada por um elemento do vetor, pela variável local *aux* ou por algum campo de controle do registro. Quando se vai gerar a saída, *b* continuará referenciando àquela área de memória, a qual conterá um dos valores mencionados anteriormente.

8. Considere o seguinte programa escrito em C:

```
#include <stdio.h>
int* calcula(int a){
        int p;
   p = a;
   if (a) {
                p*=3;
   } else {
               p++;
        };
        return &p;
}
main() {
    int x = 1;
    int* b = calcula(x);
    int* c = calcula (0);
    printf("%d\n", *b);
}
```

Descreva o que ocorre nesse programa. Justifique sua resposta.

Nesse programa também há ocorrência de referência pendente. Após *calcula* ser chamada pela primeira vez, *b* fica referenciando a área de memória alocada para *p* em *calcula*, que a essa altura já foi desalocada. Quando *calcula* é chamada de novo, a área referenciada por *b* é alocada para *p* novamente. Essa área tem seu conteúdo modificado para o valor *1*. Assim, após o término da segunda chamada de *calcula* tanto *b* quanto *c* estarão apontando para a mesma área com o valor *1*, o qual será o valor impresso.

9. Caracterize a diferença entre uniões livres e uniões disjuntas em termos de cardinalidade e segurança quanto ao sistema de tipos da linguagem. Discuta e exemplifique como as uniões de C podem ser utilizadas para criar estruturas de dados heterogêneas (isto é, estruturas de dados capazes de armazenar no seu campo de informação valores de tipos distintos), destacando como o programador (ou a linguagem, se for o caso) deve proceder para garantir o uso desse tipo de dado sem que haja violações do sistema de tipos. Discuta ainda quão genérica pode ser uma estrutura de dados heterogênea que se baseia no mecanismo de uniões.

A cardinalidade da união livre é a soma das cardinalidades de seus componentes para componentes disjuntos. Quando há interseção entre o conjunto de valores dos componentes a cardinalidade é dependente das possíveis interseções existentes. Já na união disjunta, como os conjuntos são disjuntos, a cardinalidade é a soma da cardinalidade de seus componentes.
O uso da união livre deve ser feito com atenção, pois não é muito seguro. O programador pode atribuir o valor de um tipo da união e tentar acessar a variável referenciando o outro tipo. Esse tipo de uso de uniões livres não pode ser verificado pelo compilador e têm conseqüências imprevisíveis para o resultado do programa.
Na união disjunta, o uso obrigatório do campo de "tag" na definição da união e no acesso aos seus valores torna esse tipo de união mais seguro. Porém a LP teria que testar se o tipo indicado confere com o valor a ser referenciado. Isso garante mais confiabilidade, porém há perda de eficiência ao fazer essa verificação de tipos em toda referência à união.
As uniões de C fornecem um meio de manipular tipos de dados heterogêneos em uma única área de memória. Assim, a estrutura de dados heterogênea deve possuir como elemento uma união que permita abrigar qualquer dos tipos que podem fazer parte da estrutura de dados heterogênea.
Como as uniões de C são livres, para garantir o uso desse tipo de dado sem que haja violações do sistema de tipos, o programador deve incluir no elemento um campo de "tag" para indicar o tipo do dado a ser armazenado e consultar seu valor sempre antes de acessar o valor do dado na união. Desta forma, o elemento da estrutura de dados passa a ser uma *struct* composta pelo campo "tag" e pela união. O exemplo a seguir mostra esse tipo de uso.

```
struct baleia {
  char especie [30];
  int peso;
};
struct tigre {
  char local_onde_vive [30];
  int numero_dentes;
```

```
  int tamanho_ninhada;
};
union mamifero {
  struct baleia b;
  struct tigre t;
};
struct elemento {
  int tag;                  // 0 se baleia e 1 se tigre
  union mamifero m;
};
struct no_lista_het {
  struct elemento elem;
  struct no_lista_het* prox;
};
struct no_lista_heterogena n;
struct baleia bal;
struct tigre tig;
// falta preencher conteudo de bal e tig ...
n.elem.tag = 0;
n.elem.m.b = bal;
// para acessar sem violação de tipo é preciso testar o tag antes
if (n.elem.tag == 0) {
  bal = n.elem.m.b;
}else{
  tig = n.elem.m.t;
}
```

A generalidade de uma estrutura de dados heterogênea baseada em uniões é limitada ao conjunto de tipos que fazem parte da união. Para adicionar um novo tipo de dados à estrutura heterogênea é necessário criar um código específico do campo "tag" para ele e acrescentar o novo tipo à união. Para usar a estrutura de dados heterogênea com um conjunto completamente novo de tipos é necessário refazer a *struct elemento*, substituindo a *union mamifero* pela nova *union*.

10. Uma forma de implementar listas heterogêneas em C é através do uso de uniões. Outra forma é através do uso de ponteiros para *void*. Mostre, através de exemplos de código em C, como se pode fazer para definir listas heterogêneas usando essas duas abordagens (use um campo tag adicional no nó da lista para indicar o tipo do elemento). Compare e discuta essas soluções em termos de redigibilidade (das operações da lista) e flexibilidade (em termos de necessidade de recompilação do código da lista quando for necessário alterar ou incluir um novo tipo de dado no campo informação).

O exercício 9 mostra como uniões podem ser utilizadas para construir estruturas de dados heterogêneas. O exemplo seguinte mostra como criar estruturas de dados heterogêneas usando a abordagem de ponteiros para *void*.

```
struct baleia {
  char especie [30];
  int peso;
```

```
};
struct tigre {
  char local_onde_vive [30];
  int numero_dentes;
  int tamanho_ninhada;
};
struct elemento {
  int tag;
  void* elem;
};
struct no_lista_het {
  struct elemento elem;
  struct no_lista_het* prox;
};
```

Em termos de redigibilidade das operações das listas, todas as operações podem ser implementadas usando apenas a *struct elemento* e o *no_lista_het* (sem fazer qualquer referência aos elementos *struct baleia* e *struct tigre)* em ambas abordagens. Se as operações da lista forem implementadas dessa forma, a redigibilidade das duas abordagens será a mesma.
No caso da abordagem usando *union* há necessidade de recompilação do código das operações da lista quando for preciso alterar ou incluir um novo tipo de dado no campo informação. Isso não ocorre com a abordagem de *void**.

11. Ao se retirar um elemento de uma estrutura de dados heterogênea é necessário identificar quais operações são válidas sobre o elemento removido de modo a evitar erros no sistema de tipos da LP. Como visto no exercício 10, isso pode ser feito através da consulta a um campo de "tag" indicador do tipo do elemento. Qual o impacto no código usuário quando da alteração ou inclusão de um novo tipo de elemento na lista.

O impacto é significativo no código usuário, uma vez que todo acesso ao elemento da estrutura terá de ser modificado para levar em conta a alteração do tipo ou o novo tipo incluído. A inclusão de um novo tipo é ainda mais complexa porque vai demandar a inclusão de um novo teste do campo de "tag" antes de qualquer acesso ao elemento.

12. Muito embora JAVA seja fortemente influenciada por C, os projetistas dessa LP resolveram incluir o tipo *boolean*, o qual não existe em C. Explique porque essa decisão foi tomada. Dê exemplo de situação na qual a postura de C traz alguma vantagem. Faça o mesmo em relação a postura de JAVA. Justifique suas respostas.

A decisão de incluir o tipo *boolean* em JAVA teve como principais motivações aumentar a legibilidade e a confiabilidade dos programas. A legibilidade é aumentada pois torna claro que uma variável será usada como "flag", isto é, como indicador de um estado bipolar. Em C, variáveis desse tipo podem assumir todos os valores do tipo inteiro utilizado e não apenas *true* ou *false*. A confiabilidade é aumentada por evitar erros em condições de comandos como *if*, *while*, etc.
Uma situação na qual a postura de C é vantajosa é mostrada a seguir:

```
char q[10],
char *p = "teste";
while (*q++ = *p++);
```

O uso desse comando *while* para copiar uma "string" só é possível porque C não requer que a condição desse comando retorne um tipo booleano.

Uma situação na qual a postura de JAVA é vantajosa é mostrada a seguir:

```
while (c = 1) {
```

No exemplo acima, o programador não digitou por engano um dos = do operador ==. Enquanto em C esse comando é sintaticamente válido, em JAVA daria erro de compilação.