Programação Imperativa em C e C++

Teste final, 8 de Janeiro de 2007

Nome: _____

Número: _____ **Turma:** _____

1. [5] Assinale a única alternativa correcta que completa cada uma das frases. Cada frase assinalada com uma **alternativa incorrecta desconta metade da cotação da frase** ao total do grupo.

a) [1] As variáveis locais (não estáticas) das funções são alojadas ...
- ☐ ... automaticamente no *stack*, só se a função for recursiva.
- ✓ ... no mesmo tipo de memória que os parâmetros.
- ☐ ... em memória dinâmica iniciada com zeros, libertada no final da função.

b) [1] As regras usadas pelos compiladores de C e C++ na geração de símbolos das funções são …
- ☐ ... iguais, para ser possível a ligação de módulos já compilados em C e C++.
- ☐ ... iguais, excepto para as funções **`inline`** em C++.
- ✓ ... diferentes devido à sobrecarga de funções em C++.

c) [1] Em C não existe o tipo **`bool`**, mas ...
- ☐ ... a avaliação de uma expressão é considerada falsa se o valor for diferente de 1.
- ✓ ... pode ser feita a declaração **`typedef enum { false, true } bool;`**.
- ☐ ... um valor de qualquer tipo pode ser convertido no tipo primitivo **`bit`**.

d) [1] As referências e os ponteiros têm ...
- ✓ ... a mesma representação em memória.
- ☐ ... dimensões dependentes do tipo do elemento referido/apontado.
- ☐ ... as mesmas restrições na atribuição do valor inicial.

e) [1] Dada a classe A e a classe B, que deriva de A e acrescenta um método virtual. A expressão **`sizeof(B)>sizeof(A)`** é verdadeira ...
- ☐ ... porque existe mais uma entrada na tabela de métodos virtuais.
- ☐ ... se a classe B acrescentar novos campos estáticos.
- ✓ ... se a classe A não tiver métodos virtuais.

2. [4] Para a realização de uma lista simplesmente ligada de valores inteiros em que os nós são alojados dinamicamente, considere as declarações:

```
struct N { int value; struct N * next; };
typedef struct N Node;

Node * insertAfter(int value, Node * prev);
int removeAfter(Node * prev);
```

a) [2] Implemente em C a função **`insertAfter`**, que insere na lista um novo nó com o valor **`value`** a seguir ao nó apontado por **`prev`** e retorna o ponteiro para o novo inserido.

```
Node * insertAfter(int value, Node * prev) {
      Node * n = (Node*) malloc(sizeof(Node));  // Aloja memória dinamica para o nó.
      n->next = prev->next;                     // Liga com o nó seguinte.
      prev->next = n;                           // Liga com o nó anterior.
      n->value = value;                         // Copia o valor para o nó.
      return n;                                 // Returna o ponteiro para o nó.
}
```

b) [2] Implemente a função **removeAfter**, em C. A função remove da lista o nó seguinte ao apontado por **prev** e retorna o valor inteiro do nó removido.

```
int removeAfter(Node * prev) {
      int v;                    // Para guardar o valor.
      Node * n = prev->next;    // Aponta o nó a remover.
      prev->next = n->next;     // Liga nó anterior com o seguinte.
      v = n->value;             // Copia o valor do nó.
      free(n);                  // Liberta a memória diamica.
      return v;                 // Retorna o valor.
}
```

3. [5] Considere a função:

`int copy_if(void * dest, void * orig, int nelems, int dim, int (*eval)(void *));`

que recebe em **orig** um ponteiro para um *array* de **nelems** elementos, cada um com **dim** bytes, e copia para o espaço apontado pelo ponteiro **dest** aqueles para os quais a execução da função apontada por **eval** retorne o valor 1. É retornado o número de elementos copiados de **orig** para **dest**.

Considerando as classes da 2ª série de exercícios, a função **f** demonstra uma utilização de **copy_if**:

```
int isSmallShape(void *p) {
  int area; // ... Calcula a área usando ((Shape *)p)->getBounds(…)
  return area>100 ? 1 : 0;
}

void f() {
  Line s1[N_SHAPES] = { Line(1,1,200,200), Line(10,20,20,10), /*…*/ };
  Line s2[N_SHAPES];
  int ns
  ns = copy_if(s2, s1, N_SHAPES, sizeof(Line), isSmallShape);
  // ...
}
```

a) [3] Implemente a função **copy_if**, em C. Para realizar a cópia de cada elemento utilize a função **`void* memcpy(void *d, const void *s, unsigned n)`** que copia **n** bytes de **s** para **d** e retorna **d**.

```
int copy_if(void * dest, void * orig, int nelems, int dim, int (*eval)(void *)) {
      int n = 0;
      for( ; nelems > 0 ; --nelems ) {
            if ( (*eval)(orig) == 1 ) {
                  memcpy(dest,orig,dim);
                  ((char*)dest) += dim;
                  ++n;
            }
            ((char*)orig) += dim;
      }
      return n;
}
```

b) [2] Quais as consequências da utilização de **copy_if**, tal como apresentado na função **f**, mas considerando que os elementos dos *arrays* **s1** e **s2** eram do tipo **Drawing**.

No cenário proposto, a utilização do copy_if tem um efeito equivalente ao da execução do operador afectação gerado por omissão pelo compilador de C++. Contendo a classe Drawing um array de ponteiros para Shape (m_vShapes), a utilização da função copy_if, e em particular da função memcpy, resulta na cópia directa dos ponteiros presentes no array, ficando várias instâncias de Drawing a partilhar os mesmos Shapes. Sendo assim, qualquer alteração aos Shapes de um Drawing será vista pelos Drawings que partilham os mesmos Shapes. Para além disso, uma vez que a destruição de um Drawing implica a destruição de todos os Shapes referenciados por este, existirão múltiplas tentativas de libertação sobre os mesmos endereços de memória.

4. [6] Considere a necessidade de representar dois tipos de entidades: Pessoas e Grupos de entidades. Cada entidade tem um nome. As entidades têm operações para escrever a informação que lhe está associada (**`print`**) e para obter o número de pessoas (**`persons`**) que a constituem.
A classe **`Entity`** será a classe base das classes **`Person`** e **`Group`**.

```
class Entity {
   char* name;                              // Nome da entidade.
protected:
   Entity(const char * n) { name = new char[ strlen(n)+1 ]; strcpy(name,n); }
   virtual ~Entity()      { delete name; }
public:
   virtual int persons() = 0;               // Retorna o número de pessoas.
   virtual void print()   { cout<<name; } // Escreve informação associada.
   void printAll()        { print(); cout <<" - "<< persons() << endl; }
};
```

Considere a função **`main`** que utiliza as classes descritas:

```
#define PB(g,n) (g)->push_back(new Person(n))
typedef list<Entity*> Elems;

int main() {
   Elems all;                                  // Lista de todas as entidades
   PB(&all,"JC");                              // Acrescenta uma pessoa à lista
   Group* p= new Group("A"); PB(p,"Paulo");  PB(p,"Jorge");          // grupo A
   Group* q= new Group("B"); PB(q,"Miguel"); PB(q,"Nuno");           // grupo B
   all.push_back(p);                           // Acrescenta o grupo A à lista
   p->push_back(q);                            // Grupo B como entidade do grupo A
   q->printAll();          // Escreve informação e num. de pessoas do grupo B
   for(Elems::iterator i=all.begin() ; i != all.end() ; ++i )
     (*i)->printAll();     // Escreve inf. e num. de pessoas de cada entidade
}
```

Em que o resultado será:

```
B=[ Miguel Nuno ] - 2
JC - 1
A=[ Paulo Jorge B=[ Miguel Nuno ] ] - 4
```

a) [2] Defina a classe **`Person`** de forma a cumprir o resultado esperado.

```
class Person : public Entity {
public:
      Person(const char* n) : Entity(n) {}
      int persons() { return 1; }

};
```

b) [3] Defina a classe **`Group`** de forma a cumprir o resultado esperado.

```
class Group : public Entity {
      typedef list<Entity*> Ents;
      Ents elems;
      int total;
public:
      Group(const char* n) : Entity(n) { total=0; }
      int persons() { return total; }
      void push_back(Entity *e) {
            elems.push_back(e);
            total += e->persons();
      }
      void print() {
            Entity::print(); cout << "=[ ";
            for(Ents::iterator i = elems.begin() ; i!=elems.end() ; ++i )
              { (*i)->print(); cout<<' '; }
            cout << ']';
      }
```

```
        ~Group() {
              for(Ents::iterator i = elems.begin() ; i!=elems.end() ; ++i )
                delete (*i);
        }
};
```

c) [1] Qual a necessidade do destrutor de **`Entity`** ser virtual?

A classe Group contém vários ponteiros para Entity cujos objectos apontados podem ser Person ou Group.
Como assume que as entidades foram alojadas dinamicamente com new, o destrutor faz delete de cada uma.
O operador delete chama o destrutor do objecto apontado e liberta a memória ocupada.
Para o delete chamar o destrutor adequado ao objecto apontado (Person ou Group) e não o do tipo do ponteiro (Entity), o destrutor tem que ser virtual.