**Programação Imperativa em C e C++**
Teste final, 15 de Janeiro de 2008

**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]** Considere a classe A e a classe B que deriva de A. Segundo o modelo de objectos da linguagem C++, a dimensão da tabela de métodos virtuais da classe B depende do...
- ☐ ... número total de métodos de B.
- ☐ ... número de métodos virtuais de A redefinidos em B.
- ☒ ... número total de métodos virtuais de B.

b) **[1]** A definição de destrutores …
- ☐ ... serve para libertar de memória dinâmica o espaço ocupado pelos campos primitivos.
- ☒ ... é insuficiente para garantir que a mémória alocada dinamicamente, ao longo da execução do programa, é recuperada.
- ☐ ... apenas é necessária quando algum campo é um ponteiro.

c) **[1]** Considere o seguinte *array*: `int a[4]`. As instruções `a[3]=1;` e `(&(a[1]))[2]=1;` ...
- ☒ ... são equivalentes.
- ☐ ... escrevem em posições diferentes do *array* a.
- ☐ ... dão erro de compilação devido à utilização de referências.

d) **[1]** Para que a função `f` do módulo A possa usar a função `g` do módulo B, o **compilador** tem de encontrar o código fonte com a …
- ☒ ... declaração da função `g`, normalmente no ficheiro B.h.
- ☐ ... declaração e definição da função `g`.
- ☐ ... definição da função `g`, normalmente no ficheiro B.c.

e) **[1]** Considere a função `void f(char *s) { while (*s) ++s; }` e as instruções `char a[]="picc"; f(a); printf(a);`. A chamada `f(a)` ...
- ☐ ... dá erro de compilação porque a função recebe um ponteiro e não um *array*.
- ☒ ... não produz qualquer efeito sobre o *array* `a`.
- ☐ ... altera o *array* provocando um erro de execução na chamada à função `printf` .

**2.** **[8]** Uma autarquia resolveu realizar uma sondagem para conhecer o grau de satisfação relativamente aos serviços por ela prestados: estado das ruas, iluminação pública, qualidade da água, etc. A resposta a cada questão é dada escolhendo um dos oito níveis disponíveis (desde "mau" até "muito bom"). O tipo `Resposta` abaixo apresentado é usado para registar as respostas a cada questão, onde cada bit se relaciona directamente com um nível: bit0=>"mau" até bit7=>"muito bom".

```
typedef unsigned char Resposta;   /* Apenas um bit com valor 1 */
#define MAX_NIVEIS 8
/* Nível | Resposta
       1   00000001
       2   00000010
       ...
       8   10000000 */
```

a) **[1]** Implemente a função `Resposta nivelParaResposta(int nivel)`, que recebe o nível de uma resposta (valor de 1 a 8) e retorna a resposta correspondente ao nível indicado.

Resposta:

```
Resposta nivelParaResposta(int nivel) {
       return 1 << (nivel-1);
}
```

b) [1] Implemente a função `int respostaParaNivel(Resposta resp)`, que retorna o nível da resposta indicada.

Resposta:

```
int respostaParaNivel(Resposta r) {
        int l;
        for(l=1; r>1 ; ++l, r>>=1);
        return l;

}
```

Considere o tipo `Inquerito` que reúne as respostas dadas por cada pessoa, o tipo `Sondagem` que reúne vários inquéritos, e a função `createSondagem` que aloja espaço para todos os inquéritos a recolher.

```
#define NUM_QUESTOES 10 /* um valor conhecido à prior */
typedef Resposta Inquerito[NUM_QUESTOES];
typedef struct Sondagem_t {
  int maxInqs;       // número máximo de inquéritos a guardar
  int numInqs;       // número de inquéritos já recolhidos
  Inquerito *inqs;  // ponteiro para inqueritos (alojamento dinâmico).
} Sondagem;

Sondagem *createSondagem(int numMaxInqs){
       Sondagem *s=( Sondagem *)malloc(sizeof(Sondagem));
       s->numInqs=0;
       s->maxInqs= numMaxInqs;
       s->inqs=( Inquerito *)malloc(sizeof(Inquerito)*numMaxInqs);
       return s;
}
```

Implemente em C as funções:

c) [2] `int recolherInquerito(Sondagem *s, int resps[])`, que acrescenta à sondagem indicada o inquérito representado pelos níveis das respostas dadas a cada questão e retorna 1 se a sondagem ainda tinha espaço para o inquérito.

Resposta:

```
int recolherInquerito(Sondagem *s, int resps[]) {
        Resposta *inq;
        int i;
        if (s->numInqs >= s->maxInqs) return 0;
        inq = s->inqs[ s->numInqs++ ];
        for(i=0 ; i<NUM_QUESTOES ; ++i)
                inq[i] = nivelParaResposta( resps[i] );
        return 1;

}
```

d) [2] `void contagemQuestao(Sondagem *s, int questao, int contadores[])`, que preenche o array de oito contadores com o número de respostas em cada nível na questão indicada em todos os inquéritos da sondagem.

Resposta:

```
void contagemQuestao(Sondagem *s, int questao, int contadores[]) {
        int i;
        for(i=0 ; i<NUM_QUESTOES ; ++i) contadores[i]=0;
        for(i=0 ; i< s->numInqs ; ++i)
                ++contadores[ respostaParaNivel(s->inqs[i][questao]) ];
}
```

e) [2] `int escreveInquerito(Inquerito *iq, char * filename)`, que escreve no ficheiro de texto, cujo nome é indicado, as respostas de um inquérito. Cada resposta é apresentada numa linha com 7 caracteres `'-'` e um `'X'` no nível correspondente. Por exemplo, o nível 1 fica: `x-------`.

Resposta:

```
int escreveInquerito(Inquerito *iq, char * filename) {
        int nivel,i,v;
        FILE *f;
        if (!(f = fopen(filename,"w"))) return 0;
        for(i= 0; i<NUM_QUESTOES ; ++i) {
                nivel = respostaParaNivel((*iq)[i]);
                for(v=0 ;  v<MAX_NIVEIS ; ++v) fputc((v==nivel)?'X':'-',f);
                fputc('\n',f);
        }
        fclose(f);
        return 1;
}
```

**3.** [7] Para guardar a estrutura de um sistema de ficheiros foram definidas as classes `Ficheiro`, `Pasta` e `Entrada`. Cada `Pasta` contem várias `Entradas` que podem ser `Ficheiros` ou `Pastas`. Para simplificar a solução cada `Pasta` não pode conter mais de 32 `Entradas`. Cada `Entrada` tem um nome. Cada `Ficheiro`, além do nome, tem a dimensão em bytes do conteúdo.
O método `add()` adiciona a entrada indicada à `Pasta`. O método `print()` escreve no *standard output* a informação completa da `Entrada`, do `Ficheiro` ou da `Pasta`.
Considere o seguinte troço de código que utiliza as classes descritas, a definição parcial da classe `Entrada` e a definição da classe `Ficheiro`:

```
Pasta r("root"), p1("PICC");
Ficheiro a("a.txt",64), b("b.txt",512),
         n("notas.dat",1048);
r.add(&a); r.add(&b); r.add(&p1);
p1.add(&n);
r.print();
```

```
class Entrada {
  char * nome; // Espaço alojado dinamicamente.
...
};
```

```
class Ficheiro : public Entrada {
  size_t dim;
public:
  Ficheiro(char *nome, size_t dimensao)
    : Entrada(nome) { dim=dimensao; }
  void print() const {
    Entrada::print(); cout<<" DIM="<<dim;
  }
};
```

No troço de código apresentado, a instrução `r.print();` escreve no *standard output*:
**`root{ a.txt DIM=64, b.txt DIM=512, PICC{ notas.dat DIM=1048 } }`**

**a)** [2] Complete a definição da classe `Entrada` justificando a necessidade de cada construtor e destrutor definido.

Resposta:

```
class Entrada {
        char * nome;
public:
        Entrada(char *n) { nome = new char[ strlen(n)+1 ]; strcpy( nome, n ); }
        ~Entrada() { delete[] nome; }
        virtual void print() const { cout << nome; }
        bool igualNome( Entrada *e ) const { return strcmp(e->nome,nome)==0; }
};

O construtor com um parâmetro do tipo char* é necessário porque é chamado explicitamente no construtor de
Ficheiro. Como o construtor faz alojamento dinâmico, é necessário o destrutor para libertar a memória.
```

**b)** [3] Dada a seguinte definição do método `add()` da classe `Pasta`, defina a classe `Pasta`, declarando apenas o método `add()`, e acrescente o que for necessário à classe `Entrada`, justificando a necessidade do método `print()` ser `virtual`.

```
bool Pasta::add(Entrada *e) {
  if (size==32) return false;
  for(size_t i=0; i<size ; ++i ) if (entradas[i]->igualNome(e)) return false;
  entradas[size++]=e; return true;
}
```

Resposta:

```
class Pasta : public Entrada {
        static const size_t MAX_ENTRADAS = 20;
        Entrada *entradas[MAX_ENTRADAS];
        size_t size;
public:
        Pasta(char *nome) : Entrada(nome) { size=0; }
        void print() const;
        bool add(Entrada *e);
};

void Pasta::print() const {
        Entrada::print(); cout<<"{ ";
        if (size>0) {
                entradas[0]->print();
                for(size_t i=1 ; i<size ; ++i) { cout<<", "; entradas[i]->print(); }
        }
        cout<<" }";
}

O método print() foi definido como virtual na classe Entrada porque o troço de código sublinhado terá que
fazer uma chamada polimórfica, ou seja, dependendo do tipo do objecto apontado executa o print() de
Ficheiro ou de Pasta.
```

**c)** [2] Admitindo que as instâncias de `Ficheiro` e `Pasta` são sempre alojadas dinamicamente com `new`, como mostra o seguinte troço de código, indique as alterações necessárias em todas as classes para que só seja necessário fazer o `delete` da pasta raiz para destruir e desalojar todas as instâncias.

```
Pasta *r = new Pasta("root"), *p1 = new Pasta("PICC");
r->add(new Ficheiro("a.txt",64)); r->add(new Ficheiro("b.txt",512)); r->add(p1);
p1->add(new Ficheiro("notas.dat",1048));
r->print();
delete r; // Destroi e desaloja todas as entradas
```

Resposta:

```
O destrutor da classe Pasta deve aplicar delete a cada entrada adicionada à pasta. Para que a operação
delete possa invocar polimorficamente o destrutor, é necessário declarar o destrutor da classe Entrada
como virtual.

class Entrada {
        ...
        virtual ~Entrada() { delete[] nome; }
};
class Pasta : public Entrada {
        ...
        ~Pasta() { for(size_t i=0; i<size ; ++i) delete entradas[i]; }
};
```