Programação Imperativa em C e C++
Teste final, 12 de Janeiro de 2009

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** ao total do grupo.

a) [1] Em C podem existir mensagens de erro dadas pelo …
- ☐ ... *linker*, por estar escrito `retrun` em vez de `return` num dos ficheiros fonte.
- ☐ ... compilador, se uma função for definida mais do que uma vez em ficheiros fonte diferentes.
- ☒ ... *linker*, por existir a chamada a uma função declarada mas não definida.

b) [1] Nas arquitecturas em que `sizeof(int)==sizeof(int*)`, então …
- ☐ ... `sizeof(A) == sizeof(A*)` para qualquer tipo `A`.
- ☒ ... `sizeof(A*) == sizeof(B*)` para qualquer tipo `A` e qualquer tipo `B`.
- ☐ ... `sizeof(int) < sizeof(long int).`

c) [1] Dada a função: `char f(char * a, char b) { if (&b==a+2) a++; return a[1]; }` e o array: `char n[]={'P','I','C','C',0};` a instrução `putchar(f(n,n[2]));` escreve…

☐... **P** ☒... **I** ☐... **C**

d) [1] A tabela de métodos virtuais tem uma entrada para cada …
- ☐ ... método virtual, construtor virtual e destrutor se também for virtual.
- ☐ ... método virtual e construtor virtual.
- ☒ ... método virtual e destrutor se for virtual.

e) [1] Dadas as seguintes variáveis iniciadas: `char * m = "abc"; char * p = m;`

a expressão cujo valor lógico é considerado verdadeiro é …

☐... `++p==m++` ☒... `&m[2]==p+2` ☐... `*(p+3)`

2. [10] **Implemente ou corrija as funções em C.**

a) [3] Considerando *arrays* de inteiros positivos, sem repetições, ordenados crescentemente e terminados por zero, implemente a função array_interception que recebe dois *arrays* e preenche um terceiro, recebido também como parâmetro, com os valores que constam simultaneamente nos outros dois.
Valorizam-se soluções com o mínimo de variáveis locais.
Assuma que o terceiro *array* tem espaço suficiente para o resultado.
Exemplo: Dados os *arrays*: `int a[]={1,3,5,6,7,0}; int b[]={1,2,3,4,7,0}; int res[5];` a expressão `array_interception(a,b,res)` preenche `res` com os valores: `{1,3,7,0}`.

Resposta:

```
void array_interception(const int *a, const int *b, int *r) {
  while(*a && *b)
        if (*a < *b) ++a;
        else if (*a == *(b++)) *(r++) = *(a++);
  *r = 0;
}
```

b) [2] Considerando os *arrays* da alínea anterior, implemente uma função array_alloc que retorna um *array* alojado dinamicamente igual ao que for passado como parâmetro. A memória alojada deve ser apenas a absolutamente necessária para os valores contidos. Valorizam-se soluções que usem ponteiros em vez de indexação em *arrays*.

Resposta:

```
int* array_alloc(const int *a) {
   int *p, *i;
   for(p=a ; *p ; ++p);
   i = p = (int*) malloc(((p-a)+1)*sizeof(int));
   while( *(p++)=*(a++) );
   return i;
}
```

c) [3] Dada a declaração do tipo `BitCounters`, implemente uma função `count_bits` que retorna numa estrutura `BitCounters` o número de bits a 1 (um) e a 0 (zero) do valor inteiro sem sinal recebido como parâmetro.

```
typedef struct _bitCounter {
  int bits1;
  int bits0;
} BitCounters;
```

Resposta:

```
#include <limit.h>
BitCounters count_bits(unsigned int v) {
   BitCounters c = { 0 , sizeof(unsigned int)*CHAR_BIT };
   for( ; v ; v>>=1)
         c.bits1 += v&1;
    c.bits0 -= c.bits1;
   return c;
}
```

d) [2] A função apresentada tem como objectivo perguntar o nome do utilizador e retornar o apontador para uma *string* com o nome lido. Assume-se que o nome termina quando for introduzida uma mudança de linha e que nunca são introduzidos mais do que 255 caracteres. Descreva o erro grave desta função e uma possível solução sem alterar o objectivo.

```
#define MAX_NOME 256
char * read_name() {
  char nome[MAX_NOME];
  printf("nome ? ");
  scanf("%s",nome);
  return nome;
}
```

Resposta:

O erro grave é retornar um ponteiro para uma variável local. O espaço para o array está em memória automática que é libertada quando a função retorna.

Uma possível solução é alojar o array em memória estática. A declaração seria: static char nome[MAX_NOME];

3. [5] Considere a definição parcial da classe `Student` e a função `main` que usa essa classe assim como a função apresentada em 2 d) , sendo apresentado em comentário o respectivo *input* e *output*.

```
class Student {
  int num;
  char * name;
  ... acrescentar aqui se necessário
  void setName(const char *n) {
    name = new char[strlen(n)+1]; strcpy(name,n);
  }
public:
  virtual void print() { printf("%d: %s",num,name); }
  void changeName(char *n) { delete[] name; setName(n); }
  ... acrescentar aqui se necessário
};
```

```
void main() {
  Student a(read_name());                          // nome ? Pedro
  Student* b = new Student(read_name());           // nome ? Jorge
  a.print(); putchar('\n');                        // 1: Pedro
  b->print(); putchar('\n');                       // 2: Jorge
  b->changeName("Luis");
  b->print(); putchar('\n');                       // 2: Luis
  delete b;
}
```

a) [2] Acrescente em o que for necessário na classe **`Student`** de modo a que a função **`main`** funcione correctamente. Tenha em atenção que o número de aluno é atribuído automaticamente e o espaço de memória para o nome é alojado dinamicamente.

Resposta:

Acrescentar na parte privada um campo estático:

```
static int last_num;
```

Acrescentar na parte pública o construtor e o destrutor:

```
Student(const char *n) : num(++last_num) { setName(n); }
~Student() { delete[] name; }
```

Acrescentar num módulo cpp (provavelmente Student.cpp) a reserva de espaço do campo estático:

```
int Student::last_num = 0;
```

b) [2] Defina a classe derivada **`StudentAverage`** por forma a que seja possível acrescentar na função **`main`** as seguintes instruções com o respectivo *output*.

```
void main() {
  ...  linhas anteriores
  StudentAverage c("Rui",14.3);
  b = &c;
  c.print();                                       // 3: Rui - 14.3
  b->print();                                      // 3: Rui - 14.3
}
```

Resposta:

```
class StudentAverage: public Student {
  double average;
public:
  StudentAverage(const char*n, double avg) : Student(n), average(avg) {}
  void print() { Student::print(); printf(" - %f",average); }
};
```

c) [1] Para as instruções acrescentadas na função **`main`**, na alínea anterior, funcionarem correctamente era necessário o método **`print`** ter sido definido como virtual na classe **`Student`**?
Justifique a resposta.

Resposta:

Das duas chamadas ao método print(), só a segunda é polimorfica.
A primeira chamada executa sempre o código redefinido na classe derivada, seja o método virtual ou não.
Se o método não fosse virtual, a segunda chamada executava o código definido na classe Student porque o ponteiro é do tipo Student*