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]** A função: `int* f(int *p, int v=10) {int *i; *i=v; i=p; return i;}` não está correcta porque …
- ☐ ... retorna o endereço de uma variável local.
- ✓ ... armazena um valor num local indeterminado.
- ☐ ... se o segundo parâmetro tem valor por omissão o primeiro também tem que ter.

b) **[1]** Para gerar um programa em C ou C++ com N módulos é necessário chamar …
- ☐ ... N vezes o compilador (para cada módulo) e depois N vezes o linker.
- ☐ ... uma só vez o linker e só se for compilado o módulo com a função `main()`.
- ✓ ... o compilador para cada um dos N módulos modificados e o linker para ligar os N módulos.

c) **[1]** Em C não existem referências como em C++, mas podemos ter funções com parâmetros de saída ...
- ✓ ... passando ponteiros para as variáveis a afectar.
- ☐ ... usando o `return` com mais do que um valor retornado.
- ☐ ... deixando o resultado em variáveis locais estáticas.

d) **[1]** No início de uma função, antes do prólogo, o valor do registo ESP aponta para ...
- ✓ ... o endereço de retorno do chamador.
- ☐ ... o anterior valor de EBP.
- ☐ ... os parâmetros passados à função.

e) **[1]** Numa classe, só os métodos que forem virtuais é que ...
- ☐ ... podem ser redefinidos nas classes derivadas.
- ☐ ... utilizam o ponteiro `this` para o objecto em causa.
- ✓ ... são chamados polimórficamente.

2. **[3]** Considerando arrays de inteiros, sem repetições, ordenados por ordem crescente e terminados por zero, implemente a função cardinal_intercept que retorna o número de valores que constam simultaneamente nos dois arrays passados como parâmetros. Privilegiam-se soluções com o mínimo de comparações e variáveis.
Exemplo: Dados os arrays: `int a[]={1,3,5,6,7,0}; int b[]={2,3,4,5,7,9,0};` a expressão `cardinal_intercept(a,b)` tem o valor 3.

```
int cardinal_intercept(int *a, int *b) {
  int res = 0;
  while( *a && *b ) {
    if ( *a < *b ) ++a;
    else if ( *a > *b ) ++b;
    else { ++a; ++b; ++res; }
  }
  return res;
}
```

3. **[6]** Para representar conjuntos de valores inteiros positivos, considere o tipo BitSet:

```
typedef struct BitSet {
  int numBits; // dimensão do BitSet (cardinalidade máxima do conjunto)
  int *bits; // ponteiro para elementos (alojamento dinâmico)
} BS;
```

Em `BitSet` a posição de cada bit corresponde ao valor de cada elemento do conjunto, onde **1** indica elemento presente e **0** elemento ausente. A implementação suporta-se num array de inteiros cuja dimensão depende do número de bits necessário (cardinalidade máxima do conjunto).

Para suportar as operações básicas considere as funções:

```
BS* newBitSet(int maxCard, int val); // iniciado com val (0 ou 1)
void putBit(BS *bs, int elem, int val); // val a pôr no bit (0 ou 1)
void deleteBitSet(BS *bs);
void printBitSet(BS *bs);
```

Implemente em C as funções:

a) **[1]** `newBitSet()`, que cria dinamicamente um `BitSet` com o array de inteiros cuja dimensão satisfaça o número de bits indicado (`maxCard`). Inicia a totalidade dos bits a **0** ou **1** como indicado em `val` .

```
BS* newBitSet(int maxCard, int val) {
  BS * bs; int numInts, i;
  bs = (BS*) malloc(sizeof(BS));
  if (!bs) return 0;
  bs->numBits = maxCard;
  numInts = maxCard / (sizeof(int)*8);
  if (maxCard % (sizeof(int)*8) != 0) ++numInts;
  bs->bits = (int*) malloc(sizeof(int)*numInts);
  if (!bs->bits) { free(bs); return 0; }
  if (val==1) val=-1;
  while(numInts--) bs->bits[numInts] = val;
  return bs;
}
```

b) **[1]** `deleteBitSet()`, que liberta o espaço ocupado pelo `BitSet` indicado.

```
void deleteBitSet(BS * bs) {
  free(bs->bits); free(bs);
}
```

c) **[2]** `putBit()`, que altera o bit da posição `elem` para o valor de `val`, no `BitSet` indicado. Os bits estão organizados por posição, desde 0 até `numBits-1`. O elemento 0 (zero) do conjunto corresponde ao bit de menor peso do primeiro inteiro.

```
void putBit(BS * bs, int elem, int val) {
  int idx = elem / (sizeof(int)*8);
  int mask = 1 << (elem % (sizeof(int)*8));
  if(val==1) bs->bits[ idx ] |= mask;
  else bs->bits[ idx ] &= ~mask;
}
```

d) **[2]** `printBitSet()`, que mostra na consola os elementos presentes no `BitSet` indicado. Os elementos a apresentar (posições dos bits com o valor 1) devem ficar separados por espaços.

Exº: para o BitSet **00000000000000011000000000000101** **10000000000000011000000000000101** o output deve ser **0 2 15 16 32 34 47 48 63**
inteiro no índice 0 inteiro no índice 1

```
void printBitSet(BS * bs) {
  int elem, idx, mask;
  for(elem=0; elem < bs->numBits ; ++elem) {
    idx = elem / (sizeof(int)*8);
    mask = 1 << (elem % (sizeof(int)*8));
    if ( bs->bits[ idx ] & mask )
      printf("%d ",elem);
  }
}
```

4. [6] Para implementar o modelo de frequência num ginásio, foi definida a classe `Freq` (classe abstracta base dos tipos de frequência) apresentada em seguida:

```
class Freq {
public:
    // retorna preço
    virtual float obterPreco() = 0;
    // mostra nome(s) e número de sessões
    virtual void mostrarInfo() = 0;
    void mostrarFreq() {
        this->mostrarInfo();
        cout << ':' << this->obterPreco();
    }
};
```

As frequências podem ser simples, tendo um preço fixo e um determinado número de sessões semanais. (exemplo: "Natação" uma vez por semana por 25€, "CardioFitness" duas vezes por semana por 45€).

As frequências também podem ser compostas por frequências simples, tendo desconto no preço total das frequências que a compõem (exemplo: frequentar "Natação" uma vez por semana e "CardioFitness" duas vezes por semana com um desconto de 25%).

a) [3] Defina as classes `FreqSimples` e `CardioFitness` de forma a que o seguinte troço de código tenha o comportamento indicado. Cada instância de `CardioFitness` é uma frequência simples em que o nome é "CardioFitness".

```
FreqSimples n("Natacao", 1, 35.0);
CardioFitness cf(2, 65.0);
n.mostrarFreq(); cout << '|';
cf.mostrarFreq();
```

```
Natacao x 1:35 | CardioFitness x 2:65
```

```
class FreqBase : public Freq {
  float preco;
protected:
  int sessoes;
  FreqBase(int s, float p) : preco(p), sessoes(s) {}
public:
  float obterPreco() { return preco; }
  void mostrarInfo() { cout<< "x "<<sessoes; }
};

class FreqSimples: public FreqBase {
  string nome;
public:
  FreqSimples(string n, int s, float p) : FreqSBase(s,p), nome(n) {}
  void mostrarInfo() { cout<<nome<<' '; FreqBase::mostrarInfo(); }
};

class CardioFitness: public FreqBase {
public:
  CardioFitness(int s, float p) : FreqSBase(s,p) {}
  void mostrarInfo() { cout<<"CardioFitness "; FreqBase::mostrarInfo(); }
};
```

b) [1,5] Defina a classe `FreqComposta` de forma a que o seguinte troço de código (continuação da alínea anterior) tenha o comportamento indicado.

```
FreqComposta fc(0.25);
fc.addFreq(&n); fc.addFreq(&cf);
fc.mostrarFreq();
```

```
Natacao x 1 + CardioFitness x 2:75
```

```
class FreqComposta : public Freq {
  typedef list< Freq* > Freqs;
  float factor;
  Freqs fs;
public:
  FreqComposta(float d) : factor(1-d) {}
  void addFreq(Freq *f) { fs.push_back(f); }
  float obterPreco() {
    Freqs::iterator i; float total=0;
    for( i= fs.begin(); i!=fs.end() ; ++i ) total+= (*i)->obterPreco();
    return total * factor;
  }
  void mostrarInfo() {
    Freqs::iterator i = fs.begin();
    if ( i != fs.end() ) {
      (*i)->mostrarInfo();
      for( ++i; i!=fs.end() ; ++i ) { cout<< " + "; (*i)->mostrarInfo(); }
    }
  }
};
```

c) [1,5] Com o objectivo de conter todas as frequências de um ginásio, alojadas dinamicamente, foi definida a classe `Ginasio`. Dada a função `g()`, acrescente o que for necessário (sem alterar a função) por forma a que esta termine libertando adequadamente a memória alojada dinamicamente.

```
class Ginasio {
  vector<Freq*> freqs;
public:
  void add(Freq* f)
  {
    freqs.push_back(f);
  }
};
```

```
void g() {
   Ginasio g; FreqComposta *fc;
   g.add( new FreqSimples("HidroGym",2,55.0) );
   g.add( new CardioFitness(2,65.0) );
   fc = new FreqComposta(0.30);
   fc->addFreq(new FreqSimples("Natacao",1,25.0));
   fc->addFreq(new FreqSimples("Judo",3,50.0));
   g.add( fc );
}
```

```
class Ginasio {
  ...
  ~Ginasio() {
    for( vector<Freq*>::iterator i = freqs.begin() ; i != freqs.end() ; ++i )
      delete *i;
  }
};

class Freq {
  ...
  virtual ~Freq() {}
};

class FreqComposta : public Freq {
  ...
  ~FreqComposta() {
    for (Freqs::iterator i = fs.begin(); i != fs.end() ; ++i )
      delete *i;
  }
};
```