Lt . Info 4A

CONTROLE FINAL DE LANGAGE C ET C++

BITWISES (3 POINTS)

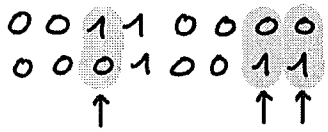
EXERCICE 1 (2 POINTS)

Donnez les affichages réalisés par la fonction suivante.

```
void test_bitwise()
{
	printf("%02x %02x %02x %02x\n", 1&2, 0x7&0x8, 0x7|0x8, 0x7<<3);
}
```

EXERCICE 2 (1 POINT)

On appelle **distance de Hamming** entre deux mots binaires le nombre de positions auxquelles ces deux mots ont des bits de valeurs différentes. Par exemple, la distance de Hamming entre 00110000 et 00010011 vaut 3. La figure ci-dessous montre les 3 positions dont les bits ont des valeurs différentes.



Complétez la fonction suivante de manière à ce qu'elle retourne la distance de Hamming entre les mots binaires représentés par ses deux paramètres.

```
int hamming(unsigned a, unsigned b)
{
	int d=0;
	while(a|b)
	{
	    if(----------------------) d++;
	    a = a>>1; b = b>>1;
	}
	return d;
}
```

TABLEAUX ET POINTEURS (3 POINTS)

EXERCICE 3 (1.5 POINTS)

Complétez la fonction suivante en faisant en sorte que son exécution ait pour effet de remplir les n premières cellules du tableau désigné par le paramètre tab avec la valeur du paramètre val.

16

```
void iniTab(int* tab, int n, int val)
{
	// À compléter
}
```

EXERCICE 4 (1.5 POINT)

Donner deux lignes de programme permettant de créer dans la pile un tableau de 10 cellules contentant chacune une valeur de type `int`, initialisée à 0, et de remplir les 5 dernières cellules de ce tableau avec des valeurs 1 grâce à un appel de la fonction `iniTab` de l'exercice précédent.

STRUCTURES MONOLITHIQUE, ET CHAINES DANS LA PILE (4 POINTS)

On définit la structure suivante.

```
#define S_MAX 100

typedef struct
{
	char data[S_MAX];
	int n;
}sbuf;
```

Une instance de sbuf encapsule une chaîne de caractères et sa longueur. Stocker la longueur dans l'attribut n permet de rendre plus efficace la concaténation d'une autre chaîne, puisqu'il n'est pas nécessaire de parcourir la chaîne depuis le début pour déterminer l'adresse du marqueur de fin. Mais il y tout de même un marqueur de fin 0 juste après le dernier caractère imprimable de la chaîne située dans le champ data.

La fonction `appendString` définie ci-dessous permet d'ajouter la chaîne au format C désignée par le paramètre `s` à la fin de la chaîne encapsulée dans l'instance de `sbuf` désignée par le paramètre `w`.

```
void appendString(sbuf* w, const char* s)
{
	while(*s != 0)
	{
		w->data[w->n] = *s;
		w->n++;
		s++;
	}
	w->data[w->n] = 0;
}
```

EXERCICE 5 (1.5 POINTS)

Complétez la fonction `appendString` de manière à ce qu'elle ajoute le caractère `c` à la fin de la chaîne encapsulée dans l'instance de `sbuf` désignée par le paramètre `w`. La nouvelle valeur de la chaîne située dans le champ `data` doit être au format C, donc comporter le marqueur de fin. Cette fonction doit être aussi efficace que possible : pas de boucle.

```
void appendChar(sbuf* w, char c)
{
	// À compléter
}
```

EXERCICE 6 (2.5 POINTS)

Donnez les lignes de code permettant successivement :

- De créer une instance de sbuf non initialisée située dans la pile.
- De placer la valeur 0 dans le champ n et dans la première cellule du tableau data de cette instance. La chaîne encapsulée est donc vide puisque le tableau data commence par la valeur 0.
- D'ajouter la chaîne "Tim" à la fin de la chaîne encapsulée, en utilisant la fonction appendString.
- D'ajouter le caractère 'X' à la fin de la chaîne encapsulée, en utilisant la fonction appendChar.
- D'afficher avec printf la chaîne encapsulée.

STRUCTURES RAMIFIEES ET TABLEAUX DYNAMIQUES (4 POINTS)

On définit la structure suivante permettant d'encapsuler une chaîne et sa longueur. À la différence de l'exercice précédent, la chaîne encapsulée est placée dans un tableau dynamique situé dans le tas.

```
typedef struct
{
	char* data;
	int n;
	int capa;
}sb;
```

Le champ n devra avoir pour valeur la longueur de la chaîne encapsulée, située dans le tableau désigné par le champ data. Cette chaîne est au format C, donc est terminée par le marqueur de fin 0. Le champ capa contient la longueur du tableau désigné par data, qui peut être plus grande que la taille de la chaîne encapsulée de manière à permettre de la prolonger sans avoir à agrandir le tableau.

EXERCICE 7 (2.5 POINTS)

Complétez la fonction creeSb de manière à ce qu'elle permette de créer dans le tas une instance de sb encapsulant une *copie* de la chaîne désignée par le paramètre s. Le tableau de stockage désigné par le champ data de l'instance créée doit avoir une taille égale à 100 *plus* la longueur de la chaîne désignées par s. Le champ capa doit être initialisé en conséquence.

```
sb* creeSb(const char* s)
{
	sb* r = ---------------------------------------------------- ;
	// À compléter
	return r;
}
```

EXERCICE 8 (1.5 POINT)

Compléter la fonction `freeSb` de manière à ce qu'elle libère toute la mémoire occupée par l'instance de `sb` désignée par le paramètre `w`. Cette instance est supposée être située entièrement dans le tas.

```
void freeSb(sb* w)
{
	// À compléter
}
```

CLASSES EN C++ (6 POINTS)

On définit la classe `Tache` qui représente une tâche ayant un nom et un statut. Le nom est une instance de `string` et le statut est un Booléen qui vaut `false` lorsque la tâche n'est pas encore réalisée et `true` lorsqu'elle est réalisée.

```
class Tache
{
private:
	string nom;
	bool status;
public:
	Tache();
	Tache(string& nom);
	string getNom();
	bool getStatus();
	void setStatus(bool stat);
};
```

Voici les définitions des deux constructeurs de la classe `Tache`.

```
Tache::Tache()
{
	nom = "";
	status = false;
}

Tache::Tache(string& nom)
{
	this->nom = nom;
	this->status = false;
}
```

Et voici les définitions de ses méthodes.

```
string Tache::getNom()
{
	return nom;
}
```

```
bool Tache::getStatus()
{
	return status;
}

void Tache::setStatus(bool stat)
{
	this->status = stat;
}
```

On définit une classe `Checklist` qui représente une liste de tâches. L'attribut `taches` désigne un tableau d'instances de Tache dont la taille est donnée par l'attribut n.

```
class Checklist
{
private:
	Tache* taches;
	int n;
public:
	Checklist(string* nomsTaches, int nTaches);
	~Checklist();
	Checklist& operator=(const Checklist& m);
	//...
};
```

EXERCICE 9 (2 POINTS)

Vous devez définir le constructeur de la classe `Checklist`. Il accepte deux paramètres : `nomTaches` qui désigne un tableau de chaînes au format C++ qui contient des noms de tâches, et `nTaches`, un entier qui a pour valeur la longueur du tableau. Ce constructeur crée une instance de `Checklist` représentant une liste de tâches non réalisées dont les noms sont dans le tableau passé en paramètre.

```
Checklist::Checklist(string* nomsTaches, int nTaches)
{
	this->n = ------------------ ;
	this->taches = ---------------------------------------- ;
	for(int i=0; i<this->n; i++)
	{
		------------------------------------------------------ ;
	}
}
```

EXERCICE 10 (2.5 POINTS)

Vous devez définir la méthode `operator=` de la classe `Checklist` de manière à ce qu'elle permette la copie d'une instance dans une autre en respectant les bonnes pratiques de programmation en C++.

```
Checklist& Checklist::operator=(const Checklist& m)
{
    // À compléter
    return *this;
}
```

EXERCICE 11 (1.5 POINTS)

Donnez les lignes de code permettant de créer dans le tas une instance de `Checklist` représentant une liste de 3 tâches non réalisées nommées "Menage", "Vaisselle" et "Courses".