Développement logiciel : Principes de base

par Thierry Vaira © v.1.00

Ce sujet comprend 35 questions pour un total de 45 points.
Aucun document autorisé
Durée : 2 h

Nom : _____

La partie "Développement logiciel : Principes de base" regroupe les savoirs suivants :
- Organisation des fichiers dans un projet logiciel ; chaîne de développement (préprocesseur, compilateur, éditeur de lien, chargeur, etc.)
- Représentation et codage des informations : bases de calcul (2,10,16), types scalaires, réels, caractères, etc.
- Gestion mémoire : adresse/valeur, pointeurs, variables statiques, allocations automatique et dynamique (pile/tas), etc.
- Variables ; durée de vie, visibilité
- Organisation des programmes : point d'entrée et arguments de la ligne de commande, prototypes, fonctions, paramètres, valeur de retour
- Flux d'entrée et de sortie de base : terminaux, fichiers, réseau, etc. (spécifications POSIX)

Organisation des fichiers et chaîne de développement

Une bibliothèque logicielle est fournie avec un ensemble de fichiers. Généralement, on trouve :
– Un répertoire `include`, qui contient des fichiers ayant une extension « `.h` »
– Un répertoire `lib`, qui contient des fichiers ayant comme extension « `.lib` »

Lors du développement d'un projet, un ensemble de fichiers est nécessaire à la génération de l'application (partie logicielle). On vous demande de préciser le contenu des fichiers manipulés (on demande de distinguer deux types de fichier, les fichiers textes et les fichiers binaires). On vous demande également d'indiquer la phase durant laquelle ces fichiers sont utilisés lors de la génération de l'application.

Question 1 (2 points)
Cocher les bonnes réponses concernant ces différents fichiers dans le tableau ci-dessous.

Type de fichier	Utilisé en entrée de la phase de **compilation séparée**	Utilisé en entrée de la phase **d'édition de lien**
*.cpp		
*.obj		
*.h		
*.lib		
*.exe		

Question 2 (1 point)
Une directive de préprocesseur est une ligne qui commence toujours par :
☐ `#` ☐ `{` ☐ `//` ☐ `;`

Représentation et codage des informations

On s'intéresse à un programme de gestion de codes postaux dans un système de tri d'objets. Ces codes postaux sont constitués de 5 caractères compris entre '`0`' et '`9`' et ils sont stockés dans un objet de type `string`. Le programme doit d'abord vérifier que le code postal est bien constitué de 5 caractères compris entre `0` et `9`. Pour réaliser cela, on a écrit une fonction `verifierCode()` qui reçoit en paramètre la chaîne de caractère représentant le code postal et qui retourne `true` si le code est valide sinon `false` :

Question 3 (4 points)
Compléter dans le corps de la fonction `verifierCode()` ci-dessous la déclaration des types des variables et leurs valeurs initiales.

```
bool verifierCode(..... code)
{
   ..... codeOK = .....;

   ..... c = .....;
```

```
   ....... i = .......;

   while (codeOK && i < 5)
   {
      c = code[i];
      if ((c < '0') || (c > '9'))
           codeOK = false;
      i++;
   }

   return codeOK;
}
```

Question 4 (1 point)
Si `a` et `b` sont définis par `int a = 0x5C;` et `int b = 0xD1;`, alors `(a & b)` vaut :
☐ `0x00` ☐ `0x01` ☐ `0x50` ☐ `0x5C` ☐ `0xDD` ☐ `0xFF`

Question 5 (1 point)
Si `a` et `b` sont définis par `int a = 0xC5;` et `int b = 0x6E;`, alors `(a | b)` vaut :
☐ `0x00` ☐ `0x01` ☐ `0x44` ☐ `0x133` ☐ `0xEF` ☐ `0xFF`

Question 6 (1 point)
Quelle est la valeur de `n` après l'exécution des instructions suivantes ?
`int n = 1; n = (n « 3) | 1;`
☐ `0` ☐ `3` ☐ `7` ☐ `1` ☐ `8` ☐ `9`

On réalise une capture d'un échange réseau d'une communication basée sur les protocoles TCP/IP :

```
00000: 00 E0 18 B9 6B 0B 00 0B DB 14 E0 6B 08 00 45 00
00010: 00 2B 6C 12 40 00 80 06 EB 49 C0 A8 11 16 C0 A8
00020: 11 0A 09 A3 11 49 A0 F1 CC A4 7A 69 8E 33 50 18
00030: FA F0 3D 42 00 00 43 06 00 00 00 00
```

Question 7 (1 point)
Dans la trame capturée ci-dessous, l'adresse IPv4 source est codée sur 32 bits par les valeurs hexadécimales : `C0 A8 11 16`. Quelle est alors la valeur de cette adresse IP en notation décimale pointée ?
☐ `192.168.11.16` ☐ `192.168.11.10` ☐ `192.168.17.10` ☐ `192.168.17.22`

Question 8 (1 point)
Quelles seraient les valeurs hexadécimales de l'adresse IPv4 `"10.0.1.12"` ?
☐ `10 00 01 12` ☐ `10 00 01 0C` ☐ `0A 00 01 0C` ☐ `0C 01 00 0A`

Question 9 (1 point)
Dans la trame capturée ci-dessous, le numéro de port source est codé sur 16 bits par les valeurs hexadécimales : `09 A3`. Sachant que les numéros de port sont représentés dans

l'ordre *big-endian* dans le protocole TCP, quelle sera alors la valeur du numéro de port source en décimal ?
☐ 80 ☐ 2467 ☐ 2617 ☐ 36922 ☐ 41737

Question 10 (1 point)
Quelle serait les valeurs hexadécimales du numéro de port 80 codé sur 16 bits ?
☐ 00 50 ☐ 50 00 ☐ 00 80 ☐ 80 00 ☐ 1F 40

Question 11 (1 point)
Dans la trame capturée ci-dessous, le champ *Flags* de l'en-tête TCP vaut en hexadécimal 0x18. Donner l'état des 6 *flags* ?

```
<-                        32 bits                                -> 
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|       |           |U|A|P|R|S|F|                                 |
|       | Reserved  |R|C|S|S|Y|I|                                 |
|       |           |G|K|H|T|N|N|                                 |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

☐ URG=0 ACK=0 PSH=0 RST=0 SYN=0 FIN=0
☐ URG=0 ACK=1 PSH=1 RST=0 SYN=0 FIN=0
☐ URG=0 ACK=0 PSH=0 RST=1 SYN=1 FIN=0
☐ URG=0 ACK=0 PSH=0 RST=0 SYN=1 FIN=0
☐ URG=0 ACK=0 PSH=0 RST=0 SYN=0 FIN=1
☐ URG=1 ACK=1 PSH=1 RST=1 SYN=1 FIN=1

Question 12 (1 point)
Donner la valeur en hexadécimal du champ *Flags* de l'en-tête TCP lorsque SYN=1 et ACK=1 ?

```
<-                        32 bits                                -> 
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|       |           |U|A|P|R|S|F|                                 |
|       | Reserved  |R|C|S|S|Y|I|                                 |
|       |           |G|K|H|T|N|N|                                 |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

☐ 0x18 ☐ 0x81 ☐ 0x11 ☐ 0x12 ☐ 0x21

Dans un système de mesures météorologiques, on utilise un capteur vent et un capteur de température. Le capteur de vent donne à la fois la direction et la vitesse du vent en communicant avec le protocole CIBus dont le format des réponses est le suivant :

Réponse transmise par le capteur, CIBus version 3

`<SOH>Raa A<STX>Cx VSS DDFF1 DDFF2 DDFF3 ddDD fffFFF DDFF4 VVVV<EOT><CRC>`

Raa Adresse physique du capteur dans le réseau CIBUS
A Identification de message de données minutes
Cx Identification logique de la mesure de vent
VSS Version du message du capteur vent/status public/status privé (V= 3)
DDFF1 Vent moyen 10 min
DDFF2 Vent instantané : maximum dans les 10 min précédentes
DDFF3 Vent moyen 2 min
ddDD Vent moyen 3 s : domaine de variation des directions dans les 10 min précédentes
fffFFF Vent moyen 3 s : domaine de variation des vitesses dans les 10 min précédentes
DDFF4 Vent instantané : maximum dans la minute précédente ;
VVVV Vent passé dans la minute précédente.

Question 13 (2 points)
À partir de la capture ci-dessous d'une trame réponse avec le protocole CIBus et sachant que « `DD` » représente la direction du vent en dizaine de degrés et « `FFi` » représente la force du vent en dixième de m/s, donnez la direction et la vitesse du vent pour les deux dernières minutes. Le dernier champ de cette trame représente la valeur du CRC.

<soh>	104	A	<stx>	C13	♥	30113	22165	31128	2136	101159	22165	0878	<eot>	6

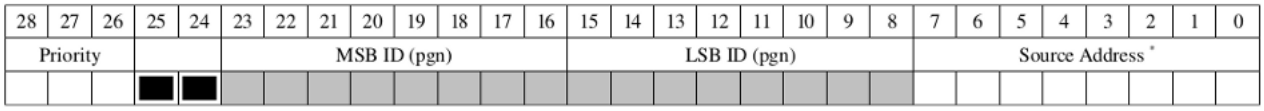
Remarque codage ASCII ♥ : 0x03
♦ : 0x04

La valeur du capteur de température est codée sur 16 bits en complément à deux et exprimée en dixième de degrés Celsius.

Question 14 (1 point)
Quelle est la valeur hexadécimale sur 16 bits pour une température de -19,3°C ?

La norme NMEA 2000, définie par la *National Marine Electronics Association* qui une association américaine de fabricants d'appareils électroniques maritimes, utilise le bus CAN en mode étendu (CAN 2.0B) comme liaison de communication entre appareils. Le bus CAN en mode étendu définit des identifiants sur 29 bits pour identifier les messages transmis.

Les messages NMEA 2000 sont identifés par un PGN (*Parameter Group Number*) qui apparairaît dans le champ ID (*IDentifier*) d'un message CAN. La structure des identifiants étendus (29 bits) des messages NMEA 2000 est la suivante :
- Priorité (3 bits)
- PGN (18 bits = 2 bits libres + 16 bits fixés par la norme)
- Adresse source (8 bits)



On réalise la capture suivante des messages NMEA 2000 :

Message	Length	Data	Period	Count
08FF2801h	8	41 9F FF FF FF FF 01 FF		1
08FF3C01h	8	41 9F 10 F9 FE F8 00 80	1000	20
09F10501h	8	83 33 FF FF FF FF FF FF	1	20
09F10D01h	8	FF FF FF 7F 19 02 FF FF	200	96
09F11201h	8	FF 50 33 FF 7F FF 7F FD	200	96
0DFF1801h	8	42 96 50 C0 FF FF FF FF	0	9
18EEFF01h	8	B8 3F 34 E8 00 96 50 C0		1
1CFF1601h	8	41 9F 0A 09 00 00 28 FF	135	2
1CFF2B01h	8	41 9F FF 08 01 FF 00 FF	500	8

Question 15 (1 point)
Quelle est la valeur en décimal du PGN du message NMEA 2000 d'identifiant `0x09F10D01` ?

Question 16 (1 point)
Quelle est la valeur en hexadécimal de l'identifiant pour un PGN `127250` émis par un appareil d'adresse `0x01` avec une priorité de `2` ?

Le message de PGN `127250` fournit le cap du navire (*Vessel Heading*) :

Cap du navire - *Vessel Heading* : PGN 127250

Field 1	**Field 2**		Field 3		Field 4		Field 5
FF	**50**	**33**	FF	7F	FF	7F	FD 1111 11**01**
SID	**Cap (*Heading*) Résolution : 0,0001 rad**		Deviation		Variation		Reference (2 bits) 0=True 1=Magnetic

Question 17 (1 point)
Déterminer la valeur du cap du bateau en degrés sachant que celle-ci est codée en *little endian* dans les données du message NMEA 2000 de PGN `127250` ?

Gestion mémoire

Question 18 (3 points)
Compléter dans le code source ci-dessous les valeurs des différentes variables ?

```
int main()
{
   int a = 1, b = 2, c = 3;
   int *p1 = &a, *p2 = &c;

   *p2 = *p1; /* 1 : a = ___ , b = ___ , c = ___ */

   *p1 = *p2 + b; /* 2 : a = ___ , b = ___ , c = ___ */

   p1 = p2; /* 3 : p1 = ___ et p2 = ___ */

   ++(*p1); /* 4 : a = ___ , b = ___ , c = ___ */

   b = (*p2)++; /* 5 : a = ___ , b = ___ , c = ___ */

   p2 = &b; /* 6 : p1 = ___ et p2 = ___ */

   return 0;
}
```

Question 19 (1 point)
L'allocation dynamique de mémoire est réalisée dans quelle zone réservée pour les programmes ?
☐ la pile (*stack*) ☐ le tas (*heap*)

Classiquement, les fonctions de la bibliothèque standard de `C` `malloc` et `free`, ainsi que les opérateurs du langage `C++` `new` et `delete` permettent, respectivement, d'allouer et désallouer de la mémoire dynamique.

Les fonctions de base d'allocation et libération dynamiques en `C` sont :

- `void * malloc (size_t size)` : La fonction `malloc()` alloue `size` octets, et renvoie un pointeur sur la mémoire allouée. Le contenu de la zone de mémoire n'est pas initialisé. Si `size` est nulle, malloc renvoie soit `NULL` ou un unique pointeur qui pourra être passé ultérieurement à `free()` avec succès.
- `void free(void *ptr)` : La fonction `free()` libère l'espace mémoire pointé par `ptr`, qui a été obtenu lors d'un appel antérieur à `malloc()`, `calloc()` ou `realloc()`. Si le pointeur `ptr` n'a pas été obtenu par l'un de ces appels, ou s'il a déjà été libéré avec `free(ptr)`, le comportement est indéterminé. Si `ptr` est `NULL`, aucune opération n'est effectuée.

Remarque : Le type `size_t` est equivalent au type `unsigned int`.

Dans les langages de programmation C et C++, l'opérateur `sizeof()` sert à calculer la taille de n'importe quel type de données ou d'un objet en mémoire (tel qu'une variable ou un tableau). La taille est mesurée en nombre d'octets. Le résultat de `sizeof` est de type `size_t` (un type entier non signé défini dans l'entête standard `stddef.h`).

```
printf("taille d'un char : %d octet\n", sizeof(char)); // 1 octet
printf("taille d'un short : %d octets\n", sizeof(short)); // 2 octets
printf("taille d'un int : %d octets\n", sizeof(int)); // 4 octets
printf("taille d'un float : %d octets\n", sizeof(float)); // 4 octets
printf("taille d'un double : %d octets\n", sizeof(double)); // 8 octets
```

Question 20 (2 points)
Écrire la fonction `allouerVecteur(int dimension, double val)` qui alloue la mémoire d'un vecteur de taille `dimension` et qui l'initialise à la valeur `val`. La fonction retourne le pointeur sur la mémoire allouée.

Question 21 (1 point)
Écrire la fonction `libererVecteur()` qui reçoit en paramètre le pointeur sur la mémoire à libérer et qui ne retourne rien.

Soit l'allocation dynamique suivante :

```
double *vecteur = NULL;

vecteur = allouerVecteur(3, 0.);
```

Question 22 (1 point)
Quelle est la taille en octets de la mémoire allouée pour le vecteur ?

Question 23 (1 point)
Écrire, en utilisant l'opérateur `new` en `C++`, l'allocation dynamique d'un tableau de 5 entiers (`int`).

Question 24 (1 point)
Écrire la libération de la mémoire allouée à la question précédente.

Variables

Question 25 (1 point)
Lequel de ces types de données permet de stocker la valeur `76.8` ?
☐ `char` ☐ `double` ☐ `long` ☐ `unsigned int`

Question 26 (1 point)
Lequel de ces types de données permet de stocker la valeur `-10000` ?
☐ `char` ☐ `unsigned double` ☐ `long` ☐ `unsigned int`

Soit le code source suivant :

```
void f(int a, int b)
{
   int c;
   c = a;   a = b;   b = c;
}

void main (void)
{
   int a = 3; int b = 5;
   f(a, b);
   printf("a = %d et b = %d\n", a, b);
}
```

Question 27 (1 point)
Qu'affiche ce programme ?
☐ `a = 3 et b = 5`
☐ `a = 5 et b = 3`
☐ `a = 5 et b = 5`
☐ `a = 0 et b = 0`

Soit le code source suivant :

```
int compte()
{
   static int cpt = 0;

   ++cpt;
   return cpt;
}

int main (void)
{
   int n = 1;

   n += compte();
   n += compte();

   printf("n = %d\n", n);

   return 0;
}
```

Question 28 (1 point)
Qu'affiche ce programme ?
☐ `n = 1`
☐ `n = 2`
☐ `n = 3`
☐ `n = 4`

Organisation des programmes

Question 29 (1 point)
Que se passe-t-il après un `return` dans une fonction ?
☐ Le programme s'arrête
☐ La fonction s'arrête et renvoie le résultat indiqué dans le `return`
☐ La fonction continue et ne renvoie pas de résultat

Question 30 (1 point)
Dans quel cas l'instruction `return` n'est pas obligatoire ?
☐ Quand la fonction ne prend aucun paramètre en entrée
☐ Quand la fonction est de type `void`
☐ Quand la fonction doit renvoyer la valeur `0`

Question 31 (1 point)
Que représentent les paramètres d'une fonction ?
☐ Des variables qu'on lui envoie pour qu'elle puisse travailler dessus
☐ Des indications complémentaires sur le rôle de la fonction
☐ Des indications sur la valeur qu'elle doit renvoyer

Question 32 (1 point)
Laquelle de ces affirmations est fausse ?
☐ Une fonction n'est pas obligée de renvoyer une valeur
☐ Une fonction peut renvoyer une valeur de n'importe quel type
☐ Une fonction peut renvoyer plusieurs type de valeurs

Soit le code source suivant :

```
int main (void)
{
   long r, x = 2;
   r = carre(x);
   return 0;
}
```

Question 33 (1 point)
Quel sera le prototype de la fonction `carre()` ?
☐ `void carre(long x);`
☐ `long carre(long x);`
☐ `double carre(double x);`
☐ `long carre(long x * long x);`
☐ `long carre(long x, long x);`

Lorsqu'on saisit des arguments après le nom d'un programme, ceux-ci sont passés en paramètres de la fonction principal `main()`. Le prototype de la fonction `main()` d'un programme C/C++ est la suivante :

```
int main(int argc, char *argv[]); // ou : int main(int argc, char **argv);

- Le paramètre argc contient le nombre d'arguments (%d pour l'afficher).

- Le paramètre argv contient l'ensemble des arguments sous la forme de chaînes
    de caractères : argv[0] contient le nom du programme, argv[1] contient le
   premier argument, etc ... (%s pour afficher un argv[x]).
```

Soit le code source du programme `testArg` :

```
#include <stdio.h>

int main(int argc, char *argv[])
{
   int i;

   printf("nb d'arguments = %d\n", argc);
   //printf("nom du programme argv[0] : %s\n", argv[0]);

   for(i=0;i<argc;i++)
   {
      printf("argv[%d] = %s\n", i, argv[i]);
   }
```

```
   return 0;
}
```

Question 34 (1 point)
Quel sera l'affichage obtenu lorsqu'on exécute la commande : `./testArg␣bts␣2016` ?
☐ `nb d'arguments = 1`
`argv[0] = ./testArg bts 2016`
☐ `nb d'arguments = 2`
`argv[0] = ./testArg`
`argv[1] = bts 2016`
☐ `nb d'arguments = 2`
`argv[0] = bts`
`argv[1] = 2016`
☐ `nb d'arguments = 3`
`argv[0] = ./testArg`
`argv[1] = bts`
`argv[2] = 2016`

Flux d'entrée et de sortie

On désire exporter des résultats vers un tableur par l'intermédiaire d'un fichier texte (au format CSV).

Le format CSV (*Comma-Separated Values*) est un format informatique ouvert représentant des données tabulaires sous forme de valeurs séparées par un délimiteur (initialement des virgules). Les séparateurs ne sont pas standardisés (virgules, points-virgules, etc...) et cela rend ce format peu pratique pour une utilisation autre que des échanges de données ponctuels. Il est toutefois très populaire parce qu'il est très facile à générer. Les fichiers CSV sont des fichiers texte essentiellement utilisés par des logiciels de type tableur (Microsoft Excel, LibreOffice/OpenOffice calc, ...). Chaque ligne du texte correspond à une ligne du tableau et les virgules correspondent aux séparations entre les colonnes. Les portions de texte séparées par une virgule correspondent ainsi aux contenus des cellules du tableau.

Le format d'enregistrement sera ici :
– retour à la ligne (`\n`) entre chaque enregistrement (mesure ;date ;heure)
– les trois valeurs (mesure ;date ;heure) sont séparés par des points-virgules (`';'`).

Les données à exporter sont accessibles à partir des tableaux suivants :

```
#include <iostream>
#include <fstream>
using namespace std;

int main ()
{
    const int nbMesures = 6;
    double pt100[nbMesures] = {35.23, 35.10, 34.45, 35.02, 35.50, 35.53 };
    string date = "12-01-2016";
```

```
    string heure[nbMesures] = {"11:00", "11:05", "11:10", "11:15", "11:20", "
        11:25" };

    //...

    return 0;
}
```

En C++, on utilisera la classe `ofstream` pour écrire dans un fichier :

public member function

std::ofstream::ofstream <fstream>

C++98 C++11

```
default (1)       ofstream();
initialization (2) explicit ofstream (const char* filename, ios_base::openmode mode = ios_base::out);
```

Construct object

Constructs an ofstream object:

(1) default constructor

Constructs an ofstream object that is not associated with any file.
Internally, its ostream base constructor is passed a pointer to a newly constructed filebuf object (the *internal file stream buffer*).

(2) initialization constructor

Constructs an ofstream object, initially associated with the file identified by its first argument (*filename*), open with the mode specified by *mode*.
Internally, its ostream base constructor is passed a pointer to a newly constructed filebuf object (the *internal file stream buffer*). Then, filebuf::open is called with *filename* and *mode* as arguments.
If the file cannot be opened, the stream's failbit flag is set.

(3) copy constructor (deleted)

Deleted (no copy constructor).

(4) move constructor

Acquires the contents of *x*.
First, the function move-constructs both its base ostream class from *x* and a filebuf object from *x*'s internal filebuf object, and then associates them by calling member set_rdbuf.
x is left in an unspecified but valid state.

The internal filebuf object has at least the same duration as the ofstream object.

Parameters

filename

A string representing the name of the file to be opened.
Specifics about its format and validity depend on the library implementation and running environment.

mode

Flags describing the requested input/output mode for the file.
This is an object of the bitmask member type openmode that consists of a combination of the following member constants:

member constant	stands for	access
in	**in**put	File open for reading: the *internal stream buffer* supports input operations.
out *	**out**put	File open for writing: the *internal stream buffer* supports output operations.
binary	**binary**	Operations are performed in binary mode rather than text.
ate	**at e**nd	The *output position* starts at the end of the file.
app	**app**end	All output operations happen at the end of the file, appending to its existing contents.
trunc	**trunc**ate	Any contents that existed in the file before it is open are discarded.

These flags can be combined with the bitwise OR operator (|).
* out is always set for ofstream objects (even if explicitly not set in argument *mode*).
Note that even though ofstream is an output stream, its internal filebuf object may be set to also support input operations.

Exemple :

```
#include <fstream> // for std::ofstream

int main ()
{
  std::ofstream ofs("test.txt", std::ofstream::out);

  ofs << "hello\n" ; // write string
```

```
  int n = 5;
  ofs << n << std::endl; // write int

  ofs.close();

  return 0;
}
```

Question 35 (3 points)
Ecrire le code C++ qui exporte les valeurs du tableau pt100 (capteur de températures) dans le fichier texte "export.txt". Celui-ci sera créé dans le répertoire courant.