**Langage de programmation avancé**

**HAUTE ÉCOLE DE NAMUR-LIÈGE-LUXEMBOURG**

**DA-IA – bloc 1**

Atelier 2 : énumérations et opérateurs bit à bit

Objectifs

* Utiliser judicieusement le codage des informations en binaires
* Connaître et utiliser les opérateurs permettant de travailler directement au niveau binaire
* Apprendre à construire et utiliser des énumérations et des énumérations binaires à bon escient

[Introduction 2](#_Toc159494398)

[A. Énumération 3](#_Toc159494399)

[Définition d'un type énuméré 3](#_Toc159494400)

[Déclaration d'une variable de type énuméré 4](#_Toc159494401)

[Utilisation d'une variable de type énuméré 5](#_Toc159494402)

[Utilisation "directe" d'une constante de type énuméré 5](#_Toc159494403)

[B. Opérateurs bit à bit 6](#_Toc159494404)

[Opérateurs 6](#_Toc159494405)

[Utilisations du OU 7](#_Toc159494406)

[Forcer un bit à 1, en laissant les autres inchangés 7](#_Toc159494407)

[Combiner des parties différentes de mots binaires 7](#_Toc159494408)

[Utilisation du AND 8](#_Toc159494409)

[Forcer un bit à 0, en laissant les autres inchangés 8](#_Toc159494410)

[Interroger la valeur d’un bit 8](#_Toc159494411)

[Récupérer une partie d’un mot, en mettant les autres à 0 8](#_Toc159494412)

[C. Vue d’ensemble 9](#_Toc159494413)

[D. Et le binaire dans tout ça ? 11](#_Toc159494414)

[Exemple 11](#_Toc159494415)

# Introduction

Cet atelier a pour objectif de vous présenter un nouveau type, appelé énumération, et de découvrir quelques utilisations des opérateurs sur les bits.

Une **énumération** permet d’associer un nom à des valeurs constantes entières qui correspondent à un même thème. Le but est de pouvoir nommer des valeurs afin de rendre le code plus lisible.

Les **opérateurs sur les bits** permettent la manipulation de données numériques en travaillant au niveau binaire, bit à bit.

Dans ce document, plusieurs conventions sont utilisées :

* les mots gras désignent des termes de vocabulaire liés à l’**informatique en général**.
* les mots soulignés et gras désignent des termes de vocabulaire directement liés aux cours de **programmation**.
* le logo signifie que vous avez quelque chose à réaliser.
* Une image contenant texte, symbole, Police, ligne

  Description générée automatiquementle logo est associé aux cadres présentant certaines conventions.
* le logo est associé aux cadres présentant les éléments liés à la propreté/lisibilité du code (*clean code*).

# Énumération

Une énumération ou type énuméré permet de créer un nouveau type à partir d’une liste de valeurs. Par exemple :

enum couleur {  
 ROUGE,   
 ORANGE,   
 JAUNE,   
 VERT,   
 BLEU,   
 VIOLET   
};

Il s'agit donc d'un type au même titre que int, float… mais qui est "créé" par le programmeur via un mécanisme qui ressemble un peu à une liste de constantes symboliques (directive #define) correspondant aux valeurs possibles.

Une fois le type défini, il sera possible de l’utiliser pour déclarer des variables qui pourront contenir une des valeurs citées. Par exemple :

enum couleur couleurCiel = BLEU;

enum couleur couleurHerbe;

couleurHerbe = VERT;

if (couleurCiel == ROUGE) printf("Panique !");

Même si tous les libellés sont remplacés par des entiers par le compilateur, leur utilisation permet d’obtenir un programme bien plus facile à lire sans obliger le lecteur (ou le programmeur) à mémoriser les codes entiers correspondant à chacun d’eux.

Pour effectuer les exercices ci-dessous, commencez par créer un projet (et une solution) sous VS2022. Donnez-lui un nom vous permettant de vous y retrouver par la suite, par exemple, Module2Atelier.

Ajoutez-lui un fichier source appelé enumEtBinaire.c qui contient la fonction main.

## Définition d'un type énuméré

Un type énuméré regroupe donc plusieurs éléments appelés énumérateurs. Chaque énumérateur associe un **identificateur** à une **valeur** qui est une constante entière.

Pour définir un type énuméré, il faut respecter la syntaxe suivante :

enum <tag> {

<enumerator\_name>,

<enumerator\_name>,

…

}**;**

Il faut y remplacer

* <tag> par l’étiquette de l’énumération, c’est-à-dire le nom donné à l’ensemble des valeurs énumérées ou énumérateurs,
* <enumerator\_name> par le nom de l’énumérateur.

**Remarques**

L'espace mémoire associé à une variable de type énuméré est celui requis pour une valeur de type int. Un énumérateur ou une valeur de type énuméré peut être utilisé partout où le langage C autorise une expression de type entier.

Par défaut, le premier énumérateur a la valeur 0. Les autres énumérateurs ont la valeur de leur prédécesseur incrémentée de 1.

On peut forcer la valeur d'un énumérateur en le faisant suivre d'un = et de la constante en question. Cependant, sauf dans certains cas, il est inutile de gérer les valeurs des énumérateurs soi-même.

Dans votre fichier source, après les définitions de constantes symboliques (directives #define) et avant la fonction principale (voire les prototypes s’il y en a), définissez l’énumération qui permet d’associer un nom de constante à chacun des jours de la semaine.

Voici ce à quoi vous devriez arriver.

enum jour {  
 LUNDI,   
 MARDI,   
 MERCREDI,   
 JEUDI,   
 VENDREDI,   
 SAMEDI,   
 DIMANCHE  
};

## Déclaration d'une variable de type énuméré

Une variable de type énuméré permet de mémoriser une des constantes faisant partie des énumérateurs de ce type.

En fait, syntaxiquement parlant, elle peut se voir affecter n'importe quelle valeur entière, il n’y a pas de validation. C’est donc à vous de faire en sorte de l’utiliser correctement.

Comme pour toute déclaration de variable, la déclaration d’une variable de type énuméré suit la syntaxe suivante :

<var\_type> <var\_name>;

Il faut y remplacer

* <var\_type> par le type que la variable doit avoir,
* <var\_name> par le nom de la variable.

Dans la fonction principale, déclarez une variable permettant de mémoriser un jour.

Vous devriez avoir une instruction similaire à la suivante.

enum jour unJour;

Faites bien attention à mettre le mot enum, c’est lui qui permet au compilateur de savoir qu’il s’agit d’un type énuméré et d’adapter le traitement des lexèmes lors des étapes d’analyse lexicale.

Comme pour les structures, il est possible de définir un synonyme au moyen de la syntaxe suivante :

typedef <var\_type> <synonym>;

Il faut y remplacer

* <var\_type> par le type à redéfinir, et
* <synonym> par le synonyme.

Juste avant la définition de l’énumération, écrivez l’instruction permettant de redéfinir le type énuméré que vous avez défini ci-dessus pour représenter un jour. Ensuite dans la fonction principale, adaptez le type de la variable unJour.

## Utilisation d'une variable de type énuméré

Une variable d'un type énuméré est utilisée comme les autres variables à la différence que sa valeur est supposée faire partie de l'ensemble des énumérateurs du type en question.

Dans la fonction principale, affectez à la variable unJour le jour correspondant à « dimanche », en utilisant la constante nommée adéquate du type énuméré Jour.

Affichez ensuite cette valeur au format numérique (format d’affichage %d).

Vérifiez que la valeur affichée correspond au jour « dimanche ».

Les instructions que vous avez écrites doivent ressembler à ceci.

unJour = DIMANCHE;

printf("%d", unJour);

Dans cet exemple, la valeur affichée est 6 puisque LUNDI correspond à la valeur 0…

## Utilisation "directe" d'une constante de type énuméré

Il est également possible de travailler avec les constantes de ce type sans passer par une variable.

Dans la fonction principale, écrivez les instructions (en choisissant judicieusement la structure de contrôle) qui permettent d’afficher le libellé correspondant à la valeur de la variable unJour. Si elle contient la constante LUNDI, vous affichez « lundi », si elle contient la constante MARDI, vous affichez « mardi »…

Une solution est d’utiliser un switch, comme ci-dessous.

switch (unJour) {

case LUNDI : printf("Lundi"); break;

case MARDI : printf("Mardi"); break;

case MERCREDI : printf("Mercredi"); break;

case JEUDI : printf("Jeudi"); break;

case VENDREDI : printf("Vendredi"); break;

case SAMEDI : printf("Samedi"); break;

case DIMANCHE : printf("Dimanche");

}

Testez votre code en affectant différentes valeurs à la variable unJour.

Pour information, si on déclare un tableau des libellés de jour, on peut même faire comme ceci :

char libellesJours[][9] = { "lundi", "mardi", "mercredi",   
 "jeudi", "vendredi"} ;

puts(libellesJours[unJour]);

# Opérateurs bit à bit

En programmation, en plus des opérateurs arithmétiques usuels, il est possible de travailler au niveau de leur représentation binaire. Les langages de programmation offrent donc des opérateurs travaillant sur les bits, appelés opérateurs *bitwise*. Certains de ces opérateurs permettent de travailler bit à bit sur des valeurs, d'autres permettent de décaler les bits vers la droite ou vers la gauche.

Les premiers sont équivalents à ceux de la logique booléenne (ou encore aux connecteurs de la logique des propositions). Tout comme dans le système binaire, la logique booléenne possède deux valeurs : 0 et 1. L'algèbre de Boole fournit 4 opérateurs qui ont chacun leur équivalent en programmation, notamment dans les langages "reposant" sur la syntaxe du C, comme montré dans le Tableau 1.

|  |  |
| --- | --- |
| Opérateurs booléens | Opérateurs bit à bit (en C) |
| NON | ~ |
| OU | | |
| OU exclusif | ^ |
| ET | & |

Tableau 1 - opérateurs booléens et bit à bit

Les seconds permettent d'effectuer des décalages. Dans les langages de programmation tels que le C,

* le >> permet de décaler vers la droite et
* le << permet de décaler vers la gauche.

*Ces deux derniers ne sont pas vus en détails dans ce module.*

Il est à noter que les opérateurs *bitwise* ne peuvent **pas** être utilisés **avec des données réelles** !

## Opérateurs

Dans cette partie, seuls les opérateurs | et & sont présentés. Le Tableau 2 est un rappel de leurs tables de vérité.

|  |  |  |  |
| --- | --- | --- | --- |
| a | b | a | b | a & b |
| **0** | **0** | 0 | 0 |
| **0** | **1** | 1 | 0 |
| **1** | **0** | 1 | 0 |
| **1** | **1** | 1 | 1 |

Tableau 2 - tables de vérité

Ces deux opérateurs sont des opérateurs dits *binaires*, dans le sens où ils nécessitent deux opérandes : <operand1> <operator> <operand2>.

C'est notamment pour cela que cet atelier est appelé "opérateurs bit à bit" et non "opérateurs binaires", ce qui n'a rien à voir !

Quand on utilise ces opérateurs sur des entiers, le résultat est calculé en appliquant l’opération correspondante sur les bits des représentations binaires de ces entiers.

Prenons par exemple les entiers 3 et 5, dont les représentations binaires sont 0011 et 0101 (ce n’est pas un hasard s’ils s’agit des nombres binaires qu’on obtient en lisant les deux premières colonnes du tableau ci-dessus ; cela permettra de couvrir toutes les combinaisons possibles).

Ajoutez le code suivant à votre fichier source et observez les résultats des deux opérations.

int bin0011 = 3;

int bin0101 = 5;

printf("%d - %d", bin0011 | bin0101, bin0011 & bin0101);

Complétez le tableau suivant avec la représentation binaire de ces résultats, puis examinez le tout colonne par colonne pour bien comprendre comment ces résultats sont obtenus.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **Valeur** | **Représentation binaire (4 derniers bits)** | | | |
| bin0011 | 3 | 0 | 0 | 1 | 1 |
| bin0101 | 5 | 0 | 1 | 0 | 1 |
| bin0011 | bin0101 |  |  |  |  |  |
| bin0011 & bin0101 |  |  |  |  |  |

## Utilisations du OU

Parmi les nombreuses utilisations possibles de cet opérateur, en voici deux.

### Forcer un bit à 1, en laissant les autres inchangés

Une image contenant texte, capture d’écran, nombre, Police

Description générée automatiquementPour forcer la valeur d’un bit à 1, on utilise l’opérateur | avec le mot et un masque (dont la valeur correspond au poids du bit à forcer) comme opérandes.

Un **masque** désigne des données utilisées, entre autres, pour des opérations bit à bit. Il doit permettre de manipuler la (ou les) valeur(s) d'un bit (ou d’un groupe de bits) en une seule opération sans modifier les autres bits.

Dans la fonction principale, écrivez la déclaration suivante : unsigned char mot. Affectez la valeur A816 à mot. Affichez la valeur de mot en hexadécimal. Écrivez ensuite l’instruction qui permet de forcer le bit de rang 2 à 1. Affichez à nouveau la valeur de mot en hexadécimal.

Pour exprimer un nombre hexadécimal en C, n’oubliez pas de le faire précéder de 0x. Par exemple 2A16 devient 0x2A en langage C. Pour obtenir ou sortir des nombres en hexadécimal, utilisez le format "%X".

Avant de tester bêtement votre programme, déterminez la valeur que mot doit avoir à la fin de son exécution. Vous pouvez vérifier votre réponse au moyen de la calculatrice de Microsoft comme montré ci-contre.

Vous devriez arriver aux instructions suivantes.

unsigned char mot = 0xA8;

printf("%X\n", mot);

mot = mot | 0x4;

printf("%X\n", mot);

Exécutez ces instructions et vérifier que vous aviez trouvé la bonne valeur pour mot.

### Combiner des parties différentes de mots binaires

Pour combiner des parties de mots binaires, par exemple 1010 00002 et 0000 10112, on utilise l’opérateur | avec les mots en question comme opérandes.

Dans la fonction principale, comme dans l’exemple précédent, déclarez les variables mot1 et mot2. Affectez-leur respectivement les valeur A016 et B16. Affichez ces deux variables en hexadécimal. Écrivez ensuite l’instruction qui permet de combiner ces deux mots en un seul et de le mémoriser dans la variable mot. Affichez la valeur de mot en hexadécimal.

Vous devriez arriver aux instructions suivantes.

unsigned char mot1 = 0xA0;

unsigned char mot2 = 0xB;

printf("%X et %X\n", mot1, mot2);

mot = mot1 | mot2;

printf("%X\n ", mot);

Vérifiez à nouveau votre réponse !

## Utilisation du AND

### Interroger la valeur d’un bit

Pour interroger la valeur d’un bit, on utilise l’opérateur & avec le mot et un masque (dont tous les bits sont à 0 sauf celui à interroger qui est à 1) comme opérandes.  
Dans la fonction principale, affectez la valeur A316 à mot. Affichez la valeur de mot en hexadécimal. Écrivez ensuite l’instruction qui permet d’interroger le bit de rang 0 et mémorisez le résultat dans la variable estImpair. Affichez « oui » si le mot contient une valeur paire et « non » si non.

Vous devriez arriver aux instructions suivantes.

bool estImpair;

mot = 0xA3;

printf("%X\n", mot);

estImpair = mot & 0x1;

printf("%s\n", estImpair ? "oui" : "non");

Vérifiez encore votre réponse…

### Récupérer une partie d’un mot, en mettant le reste à 0

Pour récupérer une partie de mot, on utilise l’opérateur & avec le mot et un masque (dont tous les bits sont à 0 sauf ceux qui correspondent aux bits à récupérer) comme opérandes.  
Dans la fonction principale, affectez la valeur AF16 à mot. Affichez la valeur de mot en hexadécimal. Écrivez ensuite l’instruction qui permet de récupérer les 4 bits de gauche dans partieGauche. Affichez la valeur de partieGauche en hexadécimal.

Vous devriez arriver aux instructions suivantes.

mot = 0xAF;

printf("%X\n", mot);

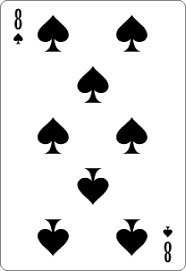
unsigned char partieGauche = mot & 0xF0;

printf("%X\n", partieGauche);

Vérifiez votre réponse…

# Vue d’ensemble

Pour réaliser l’exercice qui suit, mettez en commentaire le code du fichier source appelé enumEtBinaire.c, ajouter un nouveau fichier source appelé cartesAJouer.c à votre projet afin d’y écrire le code de l’exercice qui suit.

Le but du programme suivant est de permettre d’afficher une carte (à jouer) en sachant que chaque carte est représentée sous la forme d’un champ de bits.

Pour pouvoir représenter une carte, il faut mémoriser ses trois caractéristiques :

* sa couleur : rouge ou noir ;
* sa famille : cœur, carreau, trèfle et pique ;
* sa valeur : as, deux, trois, …, dix, valet, dame, roi.

La couleur a deux valeurs possibles. On peut donc l’encoder avec un seul bit de façon que

* la valeur 0 représente la couleur « noir », et
* la valeur 1 représente la couleur « rouge ».

La famille a quatre valeurs possibles. On a donc besoin de deux bits de façon à coder les 4 valeurs comme suit :

* la valeur 0 représente la famille « cœur »,
* la valeur 1 représente la famille « carreau »,
* la valeur 2 représente la famille « trèfle », et
* la valeur 3 représente la famille « pique ».

Enfin, en considérant que le 11 est le valet, le 12 est la dame et le 13 est le roi, on a besoin de 13 valeurs. En binaire, 1310 est représenté par 11012. Il est donc nécessaire d’utiliser quatre bits pour encoder la valeur d’une carte allant de 1 à 13.

En tout, il faut 7 bits pour représenter une carte à jouer. Plusieurs agencements sont possibles pour représenter une carte à jouer sous la forme d’un mot binaire de 7 bits. Il faut donc faire un choix.

Dans cet exercice, on décide de représenter une carte à l’aide du champ de bit suivant : CFF VVVV, avec C pour la couleur, F pour la famille et V pour la valeur.

Comme un char permet d’encoder des mots de 8 bits, le type unsigned char est celui qui correspond le mieux pour ne pas occuper de la place mémoire inutilement.

Il est intéressant de noter que si on avait décidé de mémoriser les caractéristiques d’une carte sous la forme de 2 chaines de caractères et un entier ou même de 3 entiers, on aurait occupé bien plus de place mémoire qu’en utilisant un champ de bits.

Le type unsigned char occupant 8 bits, le bit de poids fort n’est pas utilisé dans cet exemple. Une carte est donc représentée à l’aide du champ de bit 0CFF VVVV, où le bit de poids fort est toujours à 0.

Voici quelques exemples de mots binaires correspondant à des cartes précises :

* 0000 0001 🡪 As de cœur
* 0111 1011 🡪 Valet de trefle
* 0001 0111 🡪 7 de carreau

Commencez par trouver la valeur en hexadécimal correspondant à ces 3 cartes. Déclarez 3 variables (carte1, carte2 et carte3) dans lesquelles vous mémorisez ces valeurs et affichez-les.

Maintenant que vous avez des valeurs correspondant à des cartes, il est temps d’écrire le programme qui permet de déterminer de quelles cartes il s’agit et de l’afficher.

Au lieu d’utiliser des littéraux correspondant aux valeurs citées ci-dessus, que ce soit pour la couleur ou pour la famille, vous allez définir les énumérations (ou type énumérés) qui permettent d’associer une constante nommée à chacune de ces valeurs.

Définissez deux énumérations : une pour les constantes nommées liées à la couleur (NOIR et ROUGE), avec comme synonyme Couleur, et une autre pour celles liées à la famille (COEUR, CARREAU, TREFLE et PIQUE) avec comme synonyme Famille.

Vous devriez obtenir un code similaire à ce qui suit.

typedef enum couleur Couleur;

enum couleur {ROUGE, NOIR};

typedef enum famille Famille;

enum famille {COEUR, CARREAU, TREFLE, PIQUE };

En vous basant sur les exemples d’utilisation d’opérateurs bit à bit, écrivez les instructions qui permettent de récupérer dans 3 variables (couleur, famille et valeur) les 3 caractéristiques de la carte1.

Vous devriez arriver aux instructions suivantes.

unsigned char couleur = carte1 & 0x40;

unsigned char famille = carte1 & 0x30;

unsigned char valeur = carte1 & 0xF;

Maintenant que vous avez les manipulations de bases en tête, écrivez une fonction qui, recevant une carte à jouer en paramètre, affiche ses caractéristiques. Pour ce faire, il faut commencer par récupérer les 3 valeurs de caractéristiques comme ci-dessus.

Pour simplifier le traitement, copier les deux instructions qui suivent juste après.

couleur >>= 6;

famille >>= 4;

Elles permettent de décaler à droite les valeurs récupérées dans ces variables afin d’aligner celles-ci du côté du bit de poids faible.

Il faut ensuite s’occuper de l’affichage :

* La valeur est affichée sous la forme d’un entier, sauf pour les 3 images. On affiche « Valet » pour le 11, « Dame » pour le 12, « Roi » pour le 13 et « As » pour le 1.
* La famille de la carte est affichée à la suite de la valeur comme suit : « de trèfle », « de cœur »… grâce à la structure de contrôle la plus adéquate.
* La couleur de la carte est ensuite affichée entre parenthèses.

Pour les trois exemples de cartes ci-dessus, vous devriez avoir l’affichage suivant :

As de coeur (rouge)

Valet de trefle (noir)

7 de carreau (rouge)

# Et le binaire dans tout ça ?

Dans certaines applications, il est utile, voire nécessaire, d'utiliser ce qu'on appelle des *flags* ou drapeaux. Il s'agit d'un ensemble de bits fournissant une information contextuelle. En effet, comme le registre d'état du processeur qui permet de savoir s'il y a eu un dépassement de mémoire ou un report lors d'une opération d'addition, on peut utiliser un ensemble de bits pour fournir diverses informations sur le contexte.

Une technique couramment utilisée consiste à interroger un groupe de drapeaux en une seule instruction bit à bit via un masque. Mais d'autres applications sont possibles…

Le but de l’exercice qui suit est de vérifier qu’un étudiant ne joue que le week-end et pas la semaine, et d’afficher le libellé des jours de la semaine où l’étudiant a « craqué ». Une solution pour cela, on doit utiliser un mot contenant un drapeau pour chacun des jours de la semaine.

Le mot en question contient donc 7 bits, dont chacun est associé à un jour de la semaine. Le type le plus proche en termes de place mémoire est unsigned char qui contient 8 bits. Le dernier, le bit de poids fort, est inutile dans ce cas. On vous propose de travailler avec les drapeaux suivants : -dsv jeal avec '-' pour le bit non utilisé.

Chaque bit correspond à un jour, le bit de rang 0 représenté par la lettre 'l' est à 0 si l’étudiant n’a pas joué le lundi et à 1 s’il a craqué, le bit de rang 1 représenté par la lettre 'a' (mardi et mercredi commençant par la même lettre, c’est la deuxième lettre des mots qui est prise comme nom du drapeau dans cette représentation du mot) est à 0 si l’étudiant n’a pas joué le mardi et à 1 s’il a craqué, et ainsi de suite pour chacun des bits utilisés.

Si le mot vaut 01101010, cela signifie qu’il a joué le WE, mais aussi le mardi et le jeudi.

C’est la variable aJoueQuelJour qui va permettre de mémoriser les jours de la semaine où l’étudiant a joué.

unsigned char aJoueQuelJour; // drapeaux

Il est donc question de vérifier que les bits correspondants aux jours de la semaine (du lundi au vendredi) sont bien à 0 et si ce n’est pas le cas, d’afficher les libellés du/des jour(s) qui correspond(ent).

Pour cela, une énumération est à nouveau utile, mais cette fois il se peut que plusieurs jours soient à considérer en même-temps… donc en utilisant une énumération comme ci-dessous, l’information n’est pas suffisamment précise.

enum jour { LUNDI, MARDI, MERCREDI, JEUDI, VENDREDI, SAMEDI, DIMANCHE};

En effet, lorsque aJoueQuelJour vaut 5, on ne sait pas si c’est pour

* 5, c’est-à-dire SAMEDI
* 0 + 5, c’est-à-dire LUNDI et SAMEDI
* 1 + 4, c’est-à-dire MARDI et JEUDI
* 2 + 3, c’est-à-dire MERCREDI et JEUDI

Il faut donc pouvoir associer une valeur qui permet d’être interprétée même si elle est la somme de plusieurs constantes. C’est pour cette raison qu’on utilise des énumérations dites « binaire » comme décrite ci-après. On aurait pu faire la somme de 1, 10, 100… Mais en binaire, ça prend moins de place.

Voici l’énumération en question :

enum jour {

LUNDI = 1, // 0000 0001

MARDI = 2, // 0000 0010

MERCREDI = 4, // 0000 0100

JEUDI = 8, // 0000 1000

VENDREDI = 16, // 0001 0000

SAMEDI = 32, // 0010 0000

DIMANCHE = 64 // 0100 0000

};

Ainsi, pour savoir si le drapeau est levé un certain jour, par exemple mercredi, il suffit de voir le résultat de aJoueQuelJour & MERCREDI. Si le résultat est 1, l’étudiant a joué et si c’est 0, il a tenu le coup… Et comme on a un masque pour chaque jour, on peut savoir quels sont les jours où le drapeau est « levé » ou « baissé ».

Lisez le code de la page suivante et assurez-vous de comprendre ce qu’il fait.

Ensuite testez ce programme en mettant en commentaire le code du fichier source appelé cartesAJouer.c, ajoutez un autre fichier source appelé journeesDeJeu.c et copiez le code ci-dessus dans ce dernier. Quel est le jour de la semaine où il a craqué ?

Changez la valeur de aJoueQuelJour de façon à faire en sorte que l’étudiant n’ait joué aucun jour de la semaine, et ensuite de façon qu’il ait joué plusieurs jours de la semaine. Assurez-vous de comprendre ce qu’il se passe !

#include <stdio.h>

enum jour {

LUNDI = 1, // 0000 0001

MARDI = 2, // 0000 0010

MERCREDI = 4, // 0000 0100

JEUDI = 8, // 0000 1000

VENDREDI = 16, // 0001 0000

SAMEDI = 32, // 0010 0000

DIMANCHE = 64 // 0100 0000

};

void main(void) { // -dsv jeal

unsigned char aJoueQuelJour = 0x64; // 0110 0100

unsigned char masqueJoursSemaine = 0x1F; // 0001 1111

unsigned char aJoueQuelJourSemaine;

// récupérer la partie jour de la semaine du mot

aJoueQuelJourSemaine = aJoueQuelJour & masqueJoursSemaine;

// si aucun des jours de la semaine n’est à 1, sa valeur est 0

if (aJoueQuelJourSemaine == 0) {

printf("Bravo ! Quelle force...");

} else {

printf("Tu as craque ");

if (aJoueQuelJourSemaine & LUNDI)

printf("lundi ");

if (aJoueQuelJourSemaine & MARDI)

printf("mardi ");

if (aJoueQuelJourSemaine & MERCREDI)

printf("mercredi ");

if (aJoueQuelJourSemaine & JEUDI)

printf("jeudi ");

if (aJoueQuelJourSemaine & VENDREDI)

printf("vendredi ");

}

}