C

Partie I – Concepts basiques

1. Compiler : cc -c monprog.c ou gcc -c monprog.c
2. void = pas de data
3. Commentaires multi-lignes /* …. */. Bien utiliser les commentaires>
4. <X.h> standard header file. "X.h" user header file crée par le user
5. scanf() pour lire l'entrée d'un user
6. Pour scanf() je spéécifie le type de variable que je veux. Si c'est une variable basique (int, double, float, etc), je mets & (l'adresse) avant la variable dans l'argument. Si c'est pour un string, je mets rien. Exemple : scanf("%d %s", &chiffre, string);
7. scanf() est l'inverse de printf(). Output → input et vice-versa

Partie II – Variables et types de data

1/Types de data basiques

8. Une variable doit commencer par une lettre (majuscule ou minuscule) ou un _ et peut etre suivie de chiffres ou de _ (contrairement à java myData VS my_data en C)
9. Obligation de déclarer les variables avant le programme. Possibilités de déclarer plusieurs variables de meme types ensemble : int x =1, y=2, z=3;
10. float 1.5e4 = 1.5 x 10 puissance 4
11. expression de boolean type : _Bool avec 0 false et 1 true. Ou bool tout court. Ca depend de la version de C utilisée.
12. Short, long et unsigned (valeurs non négatives) int sont utilisés pour specifier la taille de memoire à utiliser pour les int avec un L ou S à la fin de la valeur. Cela rend le programme plus efficient.
13. Génerer un random aléatoire : #include <stdlib.h> et #include<time.h>
    1. Créer une variable time : time_t t;
    2. Initialiser le génerateur de nombre alératoir : srand((unsigned)time(&t));
    3. Storer le nombre aléatoire (0-20) dans une variable int : int nombreAleatoire = rand() % 21;

2/ Enums et chars

14. Les enums sont des types de data crées par le programmer : c'est un type de data qui n'acceptera que les valeurs permises par le programmeur

    1. utiliser le mot-clé enum
    2. suivi du nom de variable
    3. suivi par la liste de variantes PERMISES. Tout autre valeur sera bloquée par le compiler

    Exemple : enum maCouleur {bleu, blanc, rouge};
    Pour l'utiliser en tant que variable :

    1. on récrit enum maCouleur
    2. suivi du nom de la variable
    3. assigné d'une valeur permise

    Exemple : enum maCouleur laCouleurDeFrance;
    laCouleurDeFrance = bleu;

15. Les enums sont associés à des valeurs int. Le premier dans la liste = 0, le deuxième = 1, etc. Mais si on assigne une valeur int à une des variantes, celle d'après prend la valeur int + 1.
    Exemple : enum direction {up, down, droite = 10, gauche} equivaut aux valeurs int {0,1,10,11}

16. Char est représente un seul caractère. Il doit etre mis entre des guillements simples 'a', '6' etc. Pour déclarer : char caractere; caractere = 'r';

17. Marqueur de format
    %c – character
    %d – signed decimal int
    %i – int et bool
    %f – float et double
    %s – string ou array de String
    %u – unsigned int (pas de valeur négative)

    Exemple : printf("Ma variable a pour valeur %i\n",
    nomDeLaVariableOuLaValeurEst);

18. L'operateur de largeur est utilisé pour etre plus précis. Si je veux qu'un resultat soit printed à 2 decimal près : printf("%.2f", monFloat); etc

3/Arguments des lignes de commandes

19. Quand le main est appelé, deux arguments sont passés à la fonction.
    Le premier argument : argc (argument count) qui est un int qui spécifie le nombre d'arguments passés dans la ligne de commande.
    Le deuxieme argument : argv (argument vector) qui est une array de pointeurs de caractere (strings).

20. Exemple du main : int main (int argc, char *argv[]) {
    int nombreDArgument = argc;
    char *argument1 = argv[0];

21. Pour passer du data à un programme qui ne demande pas à un utilisateur d'en entrer : Surligner le projet / Project / Set program arguments / En entrer dans la case "'program arguments".

22. Pour y accéder il faut chercher l'argument à son index dans l'array. Un argument par index.

23. Pour avoir un saut de ligne : printf("caca : %s\n", nomDeVariable)
24. Pour créer des constantes : #define CONSTANTEMAJUSCULE xxxx avec xxxx etant la valeur de la constante. Il est intéressant d'utiliser les constantes au lieu des "nombres magiques" au cas ou on voudra changer de valeur.l

Partie III – Opérateurs

1/Overview

25. Un statement n'est pas une expréssion : il se finit avec un semi-colon. Il existep plusieurs type de déclarations.

    Déclaration : int Nombre;
    Déclaration d'attribution : Nombre = 5;
    Déclaration de type Appel de fonction : printf("Ream");
    Déclaration de type Structure : while (….);
    Déclaration de type Return : return 0;
    Déclaration composée : two ou plus déclarations groupées ensemble dans un block {….}

2/ Opérateurs basiques

26. La position (post-fix ou su-fix) de l'opérateur ++/-- à un role important.
    Avant la variable (++a) : La variable sera incrémentée +1 AVANT l'éxécution du code.
    Après la variable (a++) : La variable sera incrémentée de +1 APRES l'éxécution du code.

    Exemple :
    int a = 2;
    printf(a++) // ce qui sera printé : 2;

    int a = 2;

printf(++a) // ce qui sera printé : 3;

27. x += y // x = x+y
    x -= y // x = x – y
    x *= y // x = x *y

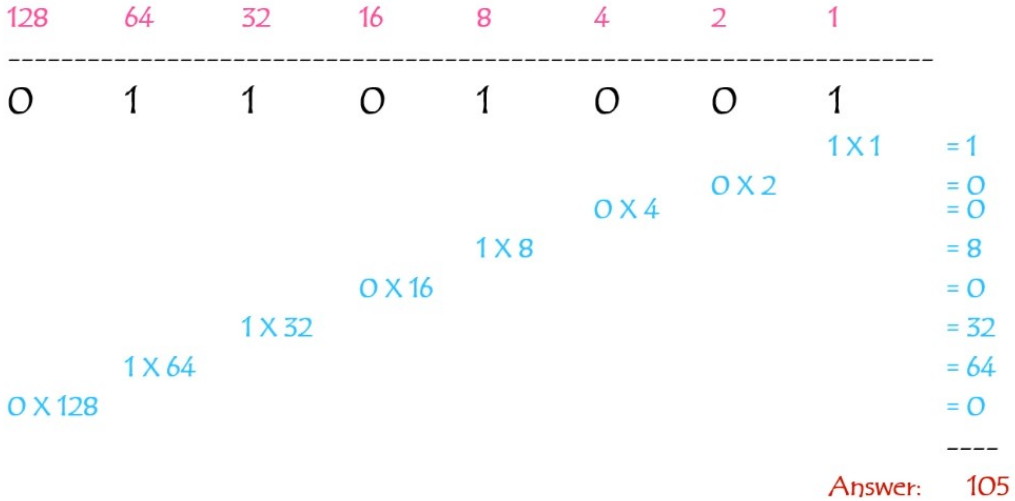
3/ Opérateurs Bits

28. Utile pour storer beaucoup de data dans quelques bits. 8 bit = 1 byte VS 4 byte = 1 int.
    1 bit pour savoir si un user est féminini ou masculin. Un autre bit pour savoir si la personne parle une certaine langue. Etc

29. Il faut connaitre l'équivalent du nombre binaire en son équivalent décimal pour utiliser le bit : pour chaque chiffre, multiplier par sa position (chaque position est multipliée par 2 en commencant par le 1er chiffre)
    Exemple :
    1 = 1
    10 = 2
    11 = 3
    1010111 = 87.

Les opérateurs fonctionnent qu'avec le 1 (true). C'est à dire que 0 & 0 donneront 0 (false) et non pas 1 (true) comme un opérateur logique.

- **The value of binary 01101001 is decimal 105. This is worked out below:**



Bitwise Operators (tutorials point)

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) = 12, i.e., 0000 1100
|	Binary OR Operator copies a bit if it exists in either operand.	(A | B) = 61, i.e., 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) = 49, i.e., 0011 0001
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) = -61, i.e,. 1100 0011 in 2's complement form.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 = 240 i.e., 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 = 15 i.e., 0000 1111

30.

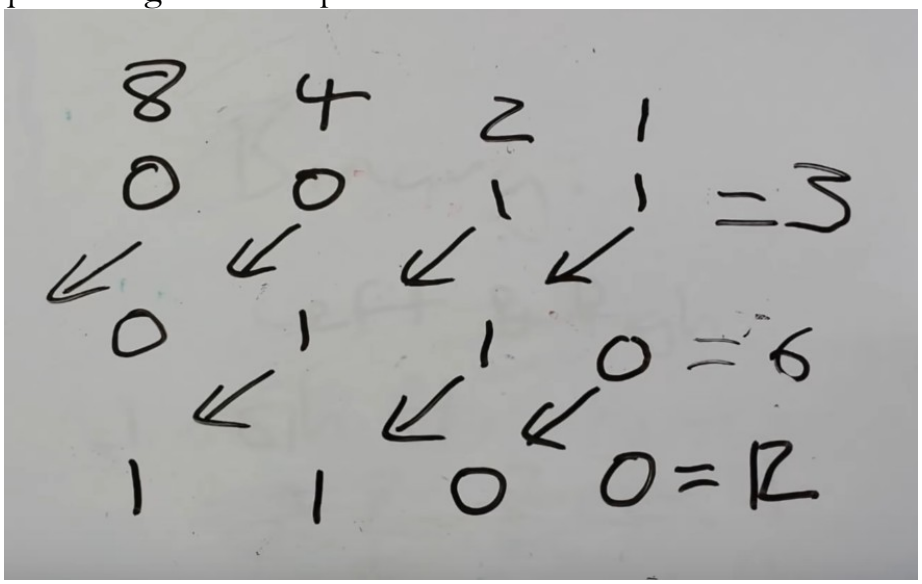
Truth Table

p	q	p & q	p | q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

31. Exemple :

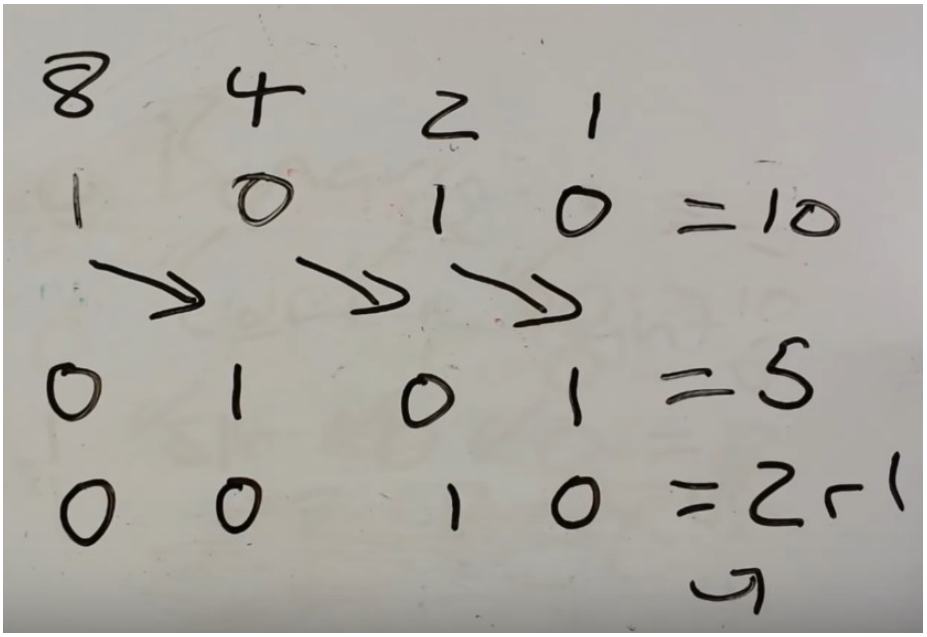
unsigned int a = 13 // 0000 1101
unsigned int b = 60 // 0011 1100
result = a &b
Il faut que les deux bits des deux valeurs soient égaux à 1 (: les premiers bits (0) valent 0, les 2eme 0, les 3eme 0, les 4eme (0 & 1) 0, les 5eme 1 (1 & 1), les 6emes 1, les 7eme 0, les 8eme 0. Cela donne le bit result : 0000 1100. Ce bit sera converti à l'éxcution du code est donnera : 12.

32. Binary Left Shift – Multiplication :
    1. On prend le bit
    2. On écrit notre table de deux en haut
    3. On pousse à gauche chaque chiffre



33. Binary right Shift –

Division :



4/ Opérateurs The Cast et sizeof

34. Lors d'une division de deux entiers qui donne une décimale, le résultat sera tronqué meme si le résultat est storé dans un float.

35. Pour convertir, mettre le type désiré entre parenthèses. Exemple : (int) 21.23 + (int) 23.21. Cela donne 21 + 23.

36. sizeof donne le nombre de bytes occupés dans la mémoire. C'est un opérateur et non pas une fonction. Il prend un argument qui peut etre une variable, le nom d'une array, le nom d'un data type ou une expression.

Exemple : sizeof(int).

37. Utiliser l'opérateur sizeof autant que possible

38. L'asterisque est un opérateur qui représente un pointeur vers une variable

39. ? est l'opérateur ternaire utilisé pour les comparaisons.
Exemple : If Condition is true ? Then value X : otherwise value Y
(a>b) ? printf("a est sup à a") : printf("b est sup à a);

Partie IV - Controle du flux

1/ Overview

40. Prise de décision : if then, if then else, switch, goto
Boucles : for, while, do while
Ramification : break, continue, return

41. Variables locales dans une boucles sont détruites à la sortie de la boucle

42. Boucle while : répète une déclaration tant que la condition est TRUE. Elle test la condition avant d'éxecuter le corps de la boucle.
Do while : Comme la boucle while mais la condition à tester se trouve à la fin. Cela nous permet d'etre sur que la boucle sera exécuter au moins une fois.

2/ IF

43. Opérateur conditionnel ternaire :
condition ? expression 1 : expression 2
Exemple : x = y > 7 ? 25 : 50;

Equivaut à :

```
if (y>7)
  x = 25;
else
  x = 50;
```

3/ SWITCH

44. Utile quand la valeur d'une variable est successivement comparée à différentes valeurs

45. switch (expression) {
case 'valeur1' :

```
        déclaration …
        break;

    case 'valeur2' :
        déclaration ..
        break;

     default :
        déclaration …
        break;
}
```

46. Les cases doivent etre des constantes ou expressions de constantes.

47. On peut retirer les break si on veut vérifier plusieurs cases simultanément et avoir différentes déclarations

3/FOR

48. for (int I = 1; I <= 10; ++i) { printf("%d, I)

49. On peut implémenter plusieurs variables.
    Exemple : for (int j = 2, x = 1; j < 5; ++j; x = x+2) { printf("%5d, j*x)

4/WHILE et DO-WHILE

50. While : condition au début de la boucle.

51. NE PAS OUBLIER d'incrémenter la boucle sinon la boucle est infinie.

52. While VS For

```
initialise:
while (test)
{
 corps;
update;
}

for (initialise; test; update)
corps;
```

53. Do while : condition à la fin de la boucle

```
do {
déclaration
{ while (expression);
```

54. Boucle while : controle par logique
    Boucle for : controle par décompte

5/Boucles nichées

55. continue; pour sauter l'iteration mais qu'on ne veut pas finir la boucle sans rien faire comme déclaration.

56. Break; est utilisé pour quitter une boucle quand il y a 2 raisons de quitter. La condition normale pour gérer la boucle et une seconde condition à l'intérieur de la boucle.

57. Return; est utilisé quand on veut sortir d'une boucle mais qu'on ne veut pas print une déclaration en dehors d'une boucle qui vient avec la loop. Exemple : for (int w = 0; w < 4; w++) { condition remplie : return }

Partie V – Arrays

1/Créer et accéder les arrays

58. Stocker les valeurs dans une array au lieu de créer une variable pour chaque valeur.

59. On ne peut pas changer la dimension et le type d'une array

60. typeDeData nomDeLaVariable [numero] / long numbers [10] équivaut à dire que l'array contient 10 élèments.

61. Pour accéder à une valeur : nomArray[numero] ou une expression qui résulte en un entier

62. Pour stocker une valeur dans une array : nomArray[numero] = valeurAStocker
    Exemple : note[100] = 95 équivaut à dire : la valeur 95 est stocké dans le 100eme élèment de l'array note

2/Initialisation

63. typeDeData nomArray[dimension] = {valeur1, valeur2, valeurX}
    Exemple:
    int entiers[5] = {0,1,2,3,4}

64. Si certains élèments d'une array ne sont pas initialisés, ils seront égaux à 0 par défaut

65. Pour initialiser tous les élèments d'une array à une meme valeur : utiliser une boucle

3/ Array multidimensionnelle

66. Array multidimensionnelle est déclarée de la meme manière qu'une simple : typeDeData nomArray[rangée][colonne].
Exemple : int matricce[4][5] contient 4 rangées et 5 colonnes, pour un total de 20 élèments.

67. Les valeurs sont initialisées par rangées.
Exemple :

```
int nombre[3][4] = {
    {10,20,30,40},          //valeurs de la première rangée
    {15,25,35,45},          //valeurs de la deuxième rangée
    {1,2,3,4}               // valeurs de la troisième rangée
};
```

68. Il n'est pas requis d'initialiser les arrays en entier

```
int numbers[2][3][4] = {
                        {               // First block of 3 rows
                          { 10, 20, 30, 40 },
                          { 15, 25, 35, 45 },
                          { 47, 48, 49, 50 }
                        },
                        {               // Second block of 3 rows
                          { 10, 20, 30, 40 },
                          { 15, 25, 35, 45 },
                          { 47, 48, 49, 50 }
                        }
                      };
```

Arraymultidimensionnelle 2+

69. Necéssité d'une boucle nichée pour accéder à tous les elements d'une array. Le niveau de niche sera le nombre de dimension de l'array. Chaque loop itère une dimension.
Exemple:

```
Int somme = 0;
For (int a = 0; a <2; a++){
    for (int b = 0; b<3;b++){
        for (int c=0; c<4; c++) {
         somme +=nombre[a][b][c]:
```

4/Array de longueur variable

70. Ne veut pas dire que sa longueur changera. Elle permet juste d'utiliser une variable pour initialiser une array
    Exemple : int a = 3;
    int array[a];

Partie VI – Fonctions

1/ Définir une fonction

71. Divide et conquer c'est créer des taches divisées en plusieurs sous-taches indépendantes

72. L'entete de la fonction définit le nom de la fonction + parametre + type de valeur que la fonction renvoit

73. TypeDeDataRenvoyé Nom_de_fonction(parametres – séparés par une virgule)
    { corps }

74. Elles doivent etre petites et spécifiques

75. S'il n'y a aucun corps dans une fonciton, le type de data renvoyé doit etre void. C'est interessant de l'utiliser comme ca lors de programmes compliqués. Ca permet de tester nos autres fonctions. Au fur et à mesure, on ajoute les détails des fonctions, en testant à chaque étape jusqu'à ce que tout soit implémenté.

76. Un prototype de fonction définit une fonction, son nom, le type renvoyé et le type de chacun de ses paramètres. On l'écrit de la meme manière qu'une fonction sauf en ajoutant un point-virgule à la fin.
    Exemple :
    void printMessage (void);

Cela permet au compiler de génerer les instructions appropriées à chaque point d'appel quand on ne définit pas les fonctions avant le main.
Pas nécessaire de le faire si on écrit bien nos fonctions avant le main.

2/Arguments et parametres

77. Les arguments sont la data passés aux paramètres d'une fonction. Valeurs véritables passées aux fonctions

78. Un parametre est inclut avec son type et se trouve entre les parathenses suivant le nom de la fonctionnent

79. Une fonction sans parametres prend void

80. Les noms de parametres sont locaux à la fonctionnent

81. Le corps de la fonction utilise les parametres définis plus haut

82. Le corps d'une fonction peut contenir des variables locales nécéssaires à son implementation

83. Lors de l'utilisation d'une array en tant qu'argument, il faut spécifier sa dimension

84. Ajouter des commentaires avant chaque fonction

85. Exemple

```
void multiplieDeuxNumeros(int x, int y) {
int resultat = x * y;
printf("résutat est %d", resultat);
}
int main (void) {
multiplieDeuxNumeros(10,20);
multiplieDeuxNumeros(20,30);

return 0;
}
```

***3/* Retourner de la data**

86. Parfois on ne voudra pas que nos fonctions retournent quoi que ce soit car on veut garder une certaine fléxibilité. C'est ici qu'on utilise return.

87. Return; fait sortir d'une fonction. L'équivalent de break dans une boucle. On l'utiliser pour les fonctions void

88. Génralement return est suivi d'une expression.
Exemple :

```
int add(int a, int b)
int resultat = a + b:
return resultat;

//main
int add = 0;
add = add(10,30);
printf("%d", add);
```

89. Une fonction qui a un corps mais qui ne retourne aucune valeur doit avoir "return void;" à la fin sinon il s'affichera une erreur pendant la compilation

90. Lors de l'appel d'une fonction, il faut suivre l'ordre de la declaration des parametres.

91. Une fonction peut etre passée à une variable.
Exemple :
int x = maMultiplication();

4/Variables locales et globales

92. Une variable locale d'une fonction n'est accessible que par la fonction et est "créer" à chaque fois que la fonction est appelée. Sa valeur est locale dans la fonction

93. Les variables locales sont applicables à n'importe quel code dans le meme bloc (boucle, declaration if etc)

94. Variables locales sont accessibles par tous et vit autant que le programme vit. Elles sont déclarées en dehors de fonctions.

95. La variable locale est prioritaire sur la variable gloable (pour les memes noms)

96.

```
int myglobal = 0;       // global variable

int main ()
{
  int myLocalMain = 0;       // local variable
 // can access my global and myLocal
 return 0;
}

void myFunction()
{
 int x;         // local variable
// can access myGlobal and x, cannot access myLocal
}
```

97. Eviter les variables globales car cela créer le jumeglage entre les fonctions (dépendance). Difficile de trouver le bug et quand il l'est, changer la variable peut engendrer des problemes dans les autres fonctions ou elle est utillisée.

98. system("cls"); pour "nettoyer" l'ecran entre différentes parties du jeu

99. getch() (get character) est utile pour demander une certaine entrée ou pour pauser un jeu jusqu'à ce que l'user tape sur une touche

100. condition ternaire pour changer entre 2 player : int player;
player = (player %2) ? 1: 2;
Pour assigner une move à un player :
move = (player == 1) ? 'X' : 'O';

Partie VII – Strings de caractères

1/Définir une string

101. La taille d'une array de char vaut toujours element + 1 car le char ,u y null est inclus.

102. Char null : terminateur de string
NULL : symbole qui représente une adresse qui ne réference rien

103. Ajout de \0 à la fin d'une string crée 2 strings

104. Pour ouvrir des guillements dans une string : obligation d'utiliser la sequence escape \ avant les guillemets.
Exemple :
char string[] = "je m'apelle Ream";
printf("La string \"%s\" est cool", string);
Ouput : La string "je m'apelle ream" est cool.

105. Déclaration d'une string : char myString[]. L'array stockera 19 caracteres car le compiler ajoute automatiquement le char null à la fin.

106. Ne pas spécifier une array de string. Laisser le compiler le faire seul pour éviter les fautes d'oubli

107. On ne peut pas initialiser une array après l'avoir déclarer. Il faut le faire en meme temps. Si on veut ajouter de la data à l'array, il faut utiliser la fonction strncpy()
Esxemple :
char s[100] ; // déclaration
s = "hello" // initialisation – NE MARCHE PAS

Possible de faire : s[0] = 'h' etc …

108. Afficher une string : on utilise direcrement le nom de l'array: ("Le message est : %s", nomDeArray)

109. scanf ne lit du input que jusqu'un 'space'.
Exemple :
Ream Sadaoui => il n'y a que Ream qui sera lu

110. la constante de string "x" n'est pas la meme que la constante caractere 'x'
'x' est un type basique (char)
"x" est un type dérivé, une array de char. Il consiste de 2 char : 'x' et '\0'.

2/ Constantes de strings

111. Constantes symboliques sont meilleures que des nombres magiques. Si jamais il faut changer le nombre, il est plus simple de changer la constante.
Exemple :
circonference = 3.14 * diametre
circonference = PI * diametre

112. Utiliser #define avant le main, tout en haut du programme (préprocesseur : il s'active avant le compiler)

113. Define n'est pas une variable, on ne peut pas lui assigner de valeurs. Pas besoin de = ou de point-virgule. Il faut juste un espace entre le nom et la valeur
Exemple : #define TAX 0.015

114. #define peut etre assigné des char/string :
#define CHAR 'c'
#define STRING "t trop belle wow!"

115. Deuxieme manière de définir une constante sous la forme d'une variable (CE N'EST PAS UNE VARIABLE CAR ON NE PEUT PAS CHANGER SA VALEUR) : utilisation du keyword const
Exemple :
const int MOIS = 12; //MOIS est une constante symbolique pour 12

116. Plus interessant qu'un DEFINE : flexible, possibilité de déclarer un type et controle quelle partie d'une programme peut utiliser la constante

117. Troisime manière de définir une constante : enums

118. Utiliser une array const est pour les messages standards.
Exemple :
const char message[] = "Bienvenu sur le site de la plus belle";

3/ Fonctions communes de strings

119. strlen spour la longueur d'une string
strcpy() et strncpy() pour copier une string à une autre
strcat() et strncat() pour combiner 2 strings ensemble
strcmp() et strncmp() pour voir si 2 strings sont égales

120. Ces fonctions se trouve dans le header file string.h

121. Longueur d'une string:
char maString[] = "my string";
printf("Sa longeur vaut %d", strlen(maString));

122. On ne peut pas copier une string à une autre qu'avec strcpy (meme soucis qu'avec l'initialisation des array)
Exemple :
char source[50], destination[50];
strcpy(source, "this is source);
strcpy(destination, "this is destination")

123. Le probleme est que strcpy() ne check pas si la string de source rentre dans la string qu'on veut. Il faut utiliser strncpy et non pas strcpy.

124. Strncpy() prend un 3ème argument : le nombre maximum de caracteres à copier
Exemple :

```
char source[40];
char destination[12]:

strncpy(destination, source, sizeof(destination) – 1) // -1 pour etre sur qu'on ne compte pas \0
```

125. Ne pas utiliser strcat() et plutot strncat() qui combine la 2eme string dans la premiere. La 1ere string est altérée, la 2eme non. Il retourne la valeur de la premiere string. Le 3eme argument est le chiffre d'élèments à ne pas dépasser
Exemple:

```
char source[50], destination[50]

strcpy(source, "la caca")
strcpy(destination, "tu pue")

strncat(destination, source, 15)
printf("Final est : %s,", destination)
```

126. Impossibilité de comparer des caracteres avec strcmp(). Utilisation des arguments “x” mais pas ‘x’

127. Fonctionnement similaire aux opérateurs relationnels
0 si les 2 sont les meme
-1 si str1 est inférieur à str2
1 si str2 est inférieur à str1

Exemple :

```
printf(“On compare\”A\” et \”A\” : “)
printf(“%d”, strcmp(“A”, “A”))          // retourne 0

printf(“strcmp(\”pommes\”, \”pomme”\) est”)
printf(“%d”, strcmp(“pommes”,”pomme”);   // retourne 1
```

128. la fonction strncmp() compare les strings jusqu’à ce qu’elles diffèrent ou qu’elle atteint le chiffre spécifié par le dev. A utiliser pour comparer deux strings ou recherche des mots

Exemple :

On recherche une string qui commence par “astro”, donc limitation aux 5 premiers caracteres

```
if (strncmp(“astronomie”,”astro”) == 0) // comparaison beaucou utilisée avec les if
{
  printf(“Trouvé : astronomie”)
}

if (strncmp(“astoum”,”astro”,5) == 0)
{
   printf(“Trouvé: astoum”)
}
```

4/Chercher, tokenizer et analyser des strings

129. Fonctions pour chercher un character ou une sous-string dans une string : strchr() et strstr()
130. Un token est une séquence de caracteres délimitée par un ‘délimeteur’ (espace, virgule, point etc) : strtok()
131. Analyse : islower(), isupper(), isdigit() etc
132. Utilisation de strchr() : le premier argument sera la string à chercher (qui sera l’adresse de l’array de char). Le second argument sera le caractere à chercher.

La fonction retourne un pointer à la premiere position ou le caractere a été trouvé (l'adresse de position en mémoire). Elle sera de type char* (qui se lit : le pointeur du caractere). Pour retourner la valeure trouvée, il faut créer une variable qui puisse socker l'adresse du caractere

133. Si caractere n'est pas trouvé, la fonction retourne NULL

134. Exemple :

```
char string[] = "tu pues";  //La string à chercher
char caractere = 'p';          // La lettre à chercher
char *pGet_caractere = NULL; //Initialisée à null car il ne pointe à rien
pGet_caractere = strchr(string, p);
```

135. strstr() : premier argument est la string à cherché et le deuxieme argument est le mot à chercher. L'adresse qui sera retournée sera le mot + tout le reste de la phrase qui suit.

Exemple :

```
char text[] = "tu pues toujours";
char mot[] = "pue";
char *pTrouve = NULL;
pTrouve = strstr(text, mot);
```

136. Utilisé un délimiteur unique et non pas une lettre :

strtok(stringàchercher, délimiteur)

Exemple :

```
char string[] = "pourquoi ? tu pues ?";
const char delimiteur[] = "?";
char *token;
token = strtok(string, delimiteur); // ca va retourner : pourquoi
```

Pour chercher TOUS les tokens, utilser une boucle while :

```
while (token != NULL){
```

137.

```
printf("%s\n", token);
token = strtok(NULL, delimiteur);
}
```

138. Analyser des strings : l'argument de toutes ces fonctions est le caractere à chercher. Elles retournent un int qui n'est pas égal à 0 si le caractere se trouve dans la string

Analyzing strings

Function	Tests for
`islower()`	Lowercase letter
`isupper()`	Uppercase letter
`isalpha()`	Uppercase or lowercase letter
`isalnum()`	Uppercase or lowercase letter or a digit
`iscntrl()`	Control character
`isprint()`	Any printing character including space
`isgraph()`	Any printing character except space
`isdigit()`	Decimal digit (`'0'` to `'9'`)
`isxdigit()`	Hexadecimal digit (`'0'` to `'9'`, `'A'` to `'F'`, `'a'` to `'f'`)
`isblank()`	Standard blank characters (space, `'\t'`)
`isspace()`	Whitespace character (space, `'\n'`, `'\t'`, `'\v'`, `'\r'`, `'\f'`)
`ispunct()`	Printing character for which `isspace()` and `isalnum()` return `false`

5/Convertir des strings

139. toupper() convertit des miniscules aux majuscules
tolower() convertit de majuscules aux miniscules

140. Exemple pour tout mettre en majuscules :
for (int i = 0; tableau[i] = (char)toupper(tableau[i])) != '\0'; i++)
(char) est là car toupper() retourne un type int

141. Exemple de conversion pour voir si une string est présente dans une autre :

```
char phrase[100]
char sous_phrase[40]

printf("Entrer une phrase à moins de 100 caracteres)
scanf("%s", phrase)

printf("Entrez la sous-phrase à chercher dans la premiere")
scanf("%s", sous_phrase)
```

// La je convertit les deux en majuscule pour pouvoir les comparer
for (int x = 0; (phrase[x] = (char)toupper(phrase[x])) != '\0'; x++)
for (int x = 0; (sous_phrase[x] = (char)toupper(sous_phrase[x]) != '\0'; x++)

//La j'utilise ma fonction strstr() avec une condition ternaire
printf("la sous-phrase %s présente dans la premiere", strstr(phrase, sous_phrase) == NULL ? "n'était pas" : "était"));

142. Convertir des strings en nombre

Function	Returns
`atof()`	A value of type `double` that is produced from the string argument. Infinity as a `double` value is recognized from the strings `"INF"` or `"INFINITY"` where any character can be in uppercase or lowercase and 'not a number' is recognized from the string `"NAN"` in uppercase or lowercase.
`atoi()`	A value of type `int` that is produced from the string argument.
`atol()`	A value of type `long` that is produced from the string argument.
`atoll()`	A value of type `long long` that is produced from the string argument.

143.

Partie VIII – Pointers

1/Définition d'un pointer

144. Déclaration d'un pointer : typeDeData *pnomDeVariable (le 'p' n'est pas nécessaire). Toujours initialiser un pointer. Au pire initialiser à NULL. On utilise p en préfixe pour etre clair sur le fait que c'est un pointer
Exemple :
int *pi; //int.
float *pf; //float

145. L'espace entre l'* et le nom du pointeur est optionnel. Les dev utillisent l'espace lors de la déclaration et l'omette quand ils veulent dereferencer une variable

146. La valeur d'un pointer est une adresse
147. %p est le spécifieur de format d'un pointer
148. Il faut toujours déclarer la variable qu'on veut accéder avant le pointeur pour le compiler ai le temps d'allouer de la mémoire et ainsi de créer une adresse qu'on peut accéder

149. Exemple de la déclaration d'un pointer : typeDeData *pnomDeVariable = &nomdevariablequ'onveutavoir

2/Accéder aux pointers

150. Les pointers sont utiles pour faire des changements sur des variables. Pour voir la valeur qu'un pointer pointe : "%d",*pointer
Exemple :
int nombre = 10;
int resultat = 0;
int *pointer = &nombre;
resultat = *pointer + 5 // soit 10 + 5 = 15

151. Pour printer la valeur de l'adresse du pointer : "%p",(void*)&pointer
On utilise (void*) pour éviter une erreur du compiler. %p expecte une valeure de type pointer alors que &pointer est un pointer d'un pointer d'un int/char/etc

152. Pour printer l'adresse d'une variable : "%p",&nomDeVariable

153. Printer la taille d'un pointer : (int)sizeof(pointer). On convertit en int car size_t est de type int

3/Utiliser des pointers

154. On peut assigner à un pointer le nom nom d'une array, une variable précedée par & ou un autre pointer

155. 2 types de soustractions :
soustraire un pointer d'un autre pour avoir un int
soustraire un int d'un pointer pour avoir un pointer

156. scanf("%zd", pointer) car le 2eme argument de scanf est une adresse et le pointer contient déjà l'adresse de la variable
Pour printer la valeur, on utilise en argument le nom de la variable et non pas le pointer

157. NE JAMAIS DEREFERENCER UN POINTER NON INITIALISE. Soit on lui donne l'adresse d'une variable ou on lui alloue un espace de mémoire

Exemple :
int * pt;
*pt = 5
NE JAMAIS FAIRE CA

158. Avant de déreferencer un pointer, on check s'il est NULL (ou le faire explicitement ave == NULL) Souvent quand on passe les pointers aux fonctions
Exemple :
if(!pointeur) …

159. pas de deferencer un pointeur quand on utilise %s car le compiler le fait automatiquement

4/Pointers et const

160. Quand on utilise const sur une variable, ca veut dire que le contenu de cette dernière sera inchangée

161. Quand on applique const à un pointer, il faut savoir si :

le pointer va changer
si le valeur à laquelle le pointer pointe changera

162. Définir un pointer à une constante :
int valeur = 0
const int *pvaleur = &valeur //la valeur 0 sera bloquée
Donc écrire : *pvaleur = 9 renverra une erreur
Si on veut changer la valeur de la variable, il faut la changer directement à la source. Soit écrire : valeur = 9. ON a juste pas utiliser le pointer pour la changer

163. On peut changer l'adresse d'un pointer const (Ce n'est pas lui qui est constant)

164. Pour bloquer l'adresse stockée dans un pointeur on utilise l'* avant const et non pas avant le pointeur:
int compte = 0
int *const pcompte = &compte
pcompte =&autrechose // NON FAISABLE
On est sur que l'adresse stockée comme valeur ne changera pas et que pcompte pointera toujours à l'adresse de compte et pas autre part. Mais comme en haut, on peut changer la valeur à laquelle le pointeur pointe
*pcompte = 45 //TOTALEMENT FAISABLE

165. Tout dépend d'ou on met le mot-clé 'const':
const int *p : valeure ne peut pas etre changée

int *const p : l'adresse ne peut pas etre changée

166. On peut créer un pointer constat qui pointe à une valeur constante. On pourra ni changer la valeur ni l'adresse. On pourra changer la valeur directement SAUF si on déclare la variable avec const :
int compte = 0; //On peut changer la valeur directement
const int compte = 0; //On pourra JAMAIS RIEN changer
const int *const pcompte = &compte

5/Pointers void

167. Un pointer de type void* peut contenir l'adresse d'une variable de n'importe quel type

168. Il faut convertir implicitement le void* avant de le déreferencer
Exemple :
int x = 0
char car = 'r'
void * pointer = &x //un pointer peut pointer à plusieurs endroits
printf("valeur de %d", *(int *)pointer)
void * pointer = &car
printf("valeur de %c", *(char *)pointer;

6/Pointers et arrays

169. C'est presque les meme car les arrays sont des types de pointers elles-memes

170. Pointer à des arrays résulte dans un code plus rapide

171. On utilise pas &

172. Quand on pointe un pointer à une array, on ne le pointe pas à l'array entière mais au type d'element contenu dans l'array.
Exemple :
int valeurs[100]
int *pointerValeur = valeurs; //il pointera au 1er element de l'array
int *pointerValeur = &valeur[0] équivaut à la ligne dessus
*(p+0) equivaut à valeur[0]
p+0 équivaut à &valeur[0]

173. pointer++ pour passer d'un element à un autre dans une array

7/Expressions mathématiques

174. Pour accéder à un élément de l'array : *(pointerValeur + j)
175. Pour appliquer une valeur à un élément : *(pointerValeur + j) = 20;

176. incrementation et decrementation sont utiles ici
Attention avece - -pointerValeur si on est au début d'une array
Exemple :

```
int somme (int array[], const int n){
int somme = 0;
int * pointer =  NULL;
int * const finArray = array + n

for (pointer = array; pointer<finArray; pointer++)
somme += pointer,
return somme
```

MAIN

```
int valeurs[3] = {1,2,4}
printf("la somme est %d", somme(valeurs,10))
```

177. on peut déclarer une array dans une fonction comme étant des pointers :

```
int somme (int *array, int n)
int somme = 0;
int * const finArray = array + n

for (; array<finArray; ++array) //pas besoin d'initialiser à 0 car array est un
pointer qui commence au 1er element
somme += array,
return somme
```

178.

```
int urn[3];
int * ptr1, * ptr2;
```

Valid	Invalid
`ptr1++;`	`urn++;`
`ptr2 = ptr1 + 2;`	`ptr2 = ptr2 + ptr1;`
`ptr2 = urn + 1;`	`ptr2 = urn * ptr1;`

8/Pointers et arrays - Exemples

This program demonstrates the effect of adding an integer value to a pointer.

```
#include <stdio.h>
#include <string.h>

int main(void)
{
 char multiple[] = "a string";
 char *p = multiple;

 for(int i = 0 ; i < strnlen(multiple, sizeof(multiple)) ; ++i)
 printf("multiple[%d] = %c  *(p+%d) = %c  &multiple[%d] = %p  p+%d = %p\n",
                i, multiple[i], i, *(p+i), i, &multiple[i], i, p+i);
 return 0;
}
```

179. **UN AUTRE EXEMPLE**

```
#include <stdio.h>

int main(void)
{
 long multiple[] = {15L, 25L, 35L, 45L};
 long *p = multiple;

 for(int i = 0 ; i < sizeof(multiple)/sizeof(multiple[0]) ; ++i)
  printf("address p+%d (&multiple[%d]): %llu       *(p+%d)   value: %d\n",
                    i, i, (unsigned long long)(p+i),  i,  *(p+i));

 printf("\n    Type long occupies: %d bytes\n", (int)sizeof(long));
 return 0;
}
```

180.

9/Pointers et strings

181. Vaut mieux utiliser un char pointer qu'une array de char

```
void copyString (char  to[], char  from[])
{
    int  i;

    for ( i = 0;  from[i] != '\0';  ++i )
        to[i] = from[i];

    to[i] = '\0';
}

void copyString (char  *to, char  *from)
{
    for (  ;  *from != '\0';  ++from, ++to )
        *to = *from;
    *to = '\0';
}
```

182. ++from et ++to car on peut faire des opérations arithmétiques sur des pointers

183. pointers de char : char *pointerText. Pour accéder aux elements : + +pointerText

184. optimisation de codes avec pointers
void copy(char * to, char * from)
while (*from) //le caractere null est égal à 0 donc la condition sautera automatiquement
*to++ = *from++;

MAIN
char phrase1[] = "tu pues"
char prhase2[50];
copy(phrase2,phrase1)
On a pas besoin de convertir les arrays en pointer car c'est des pointers eux-memes

10/Pass by reference

185. On peut passer de la data à une fonction par valeur ou par reference

186. Passer par valeur c'est quand une fonction copie la valeur actuelle d'un argument dans le parametre formel de la fonction

187. Passer par réference copie l'adresse d'un argument dans le parametre formel et les changements faitx aux parametres seront appliqués aux arguments passés. Contrairement à une passe par valeur

188. Les changements à l'intérieur d'une fonction sont accédés par l'exterieur sans avoir besoin de déclarer des variables globales, d'ou l'utilite !

189. Exemple entre une passe par valeur (les valeurs restent les memes) et une passe par reference (les valeurs changent)

Par valeur :
void inverse(int x, int y) {

int temp;
temp = x;
x = y;
y=temp;

On print inverse(a,b) : les valeurs inchangées

Par reference :

```
void inverse(int * x, int * y) {

int temp;
temp = *x;
*x = *y;
*y = temp;
```

On print inverse(&x, &y) : les valeurs changent

190. On peut passer à une fonction un pointer. Quand on veut l'appeler, on invoque l'adresse du pointer
191. En retournant un pointer depuis une fonction permet de retourner tout un set de valeurs contrairement à avant ou on retournait une seule valeur
192. Pour ca, il faut declarer une fonction qui retourne un pointer :
int * maFonction()
193. Faire attention aux variables : en créer des locales pour éviter que ca interfere avec la variable à laquelle l'argument pointe

11/Allocation de mémoire dynamique

194. En définissant une variable ou un pointer, le compiler alloue automatiquement de la mémoire
195. Allouer de la mémoire quand on le veut pendant runtime est préferable
196. Avec la mémoire dynamique, je peux avoir autant de stockage que je veux quand j'en ai besoin. On évite donc d'utiliser de la mémoire pour rien
197. Heap vs Stack :
toute la mémoire dynamique reserve de l'espace dans une zome de mémoire qui est le heap. Elle reste durant tout le temps du programme. On controle quand en créer et quand en libérer.
La zone de mémoire stack est là ou sont alloués les arguments et variables de fonctions. Quand l'execution d'une fonction finit, l'espace alloué pour stocker les arguments et variables locales est libéré. On a pas de controle ici.

12/ Malloc, calloc and realloc (stdlib.h)

198. malloc retourne l'adresse du premier byte de mémoire allouée. C'est pour ca qu'on y met un pointer. On convertit en int car malloc retourne un pointer de type void
TypeDeData ***pointer =***
(typededata*)malloc(nombreDeValeurs*sizeof(typeDeData))*
int * pointer = (int*)malloc(25*sizeof(int))

199. malloc retourne NULL quand il n'y a pas de place ou que la mémoire ne peut pas etre allouée. Toujours checker que malloc ne renvoie pas NULL
Exemple :
int *pointer = (int*)malloc(25*sizeof(int))
if(!pointer){ code pour arranger l'echec d'allocation de mémoire}
En temps normal, sortir du program, break ou exit car c'est bourbier sinon. Printer un message et terminer le program, c'est mieux que de crasher le program et de chercher la faille dans le stack

200. Toujours libérer la mémoire du heap quand on finit de l'utiliser.

201. Une fuite de mémoire est quand tu alloues de la mémoire mais que tu ne retiens pas sa reference donc tu ne peux pas libérer la memoire. Récurrent avec les boucles : à chaque itération on occupe de la mémoire jusqu'à tout occuper

202. free(pointer);
pointer = NULL;
la fonction free() prend un pointer de type void.

203. Calloc initialise la mémoire necessaire pour que tous les bytes soient égaux à 0 (avantage)
typeDeData *pointer = (int *)calloc(nombreDeValeur, sizeof(typeDeData))
int *pointer = (int *)calloc(75, sizeof(int))

204. realloc permet de réutiliser ou d'etendre de la mémoire allouée avec malloc ou calloc. Elle prend 2 arguments : un pointer qui contient l'adresse de ce qui était retourné par malloc/calloc et la taille en bytes de la nouvelle mémoire à allouer
Exemple:
char *string = (char *)malloc(15);
strcpy(string,"jason")

realloc : string = (char*)realloc(string,25*sizeof(char))
free(string)

205. Eviter d'allouer des petites zones de mémoire

206. Toujours etre sur qu'on peut acceder à la mémoire qu'on a allouée : s'il faut la rendre globale on le fait (hors des boucles ou de blocks etc). On pense donc en avance à quand on va liberer la mémoire

Partie IX – Recursivite

207. La recursivite c'est quand une fonctions s'apelle elle-meme.
Exemple :

208. void min_max_som(int tab[], int taille, int *min, int *max, int *som) {
if (taille>1) {
min_max_som(tab+1,taille-1,min,max,som);
*som+=tab[0];

```
*min=*min>tab[0]?tab[0]:*min;
*max=*max<tab[0]?tab[0]:*max;

 else {
 *min=tab[0];
*max=tab[0];
*som=tab[0]; } }

void min_max_moy(int tab[], int taille, int *min, int *max, float *moy) {
int som;
min_max_som(tab,taille,min,max,&som);
 *moy=(float)som/(float)taille; }
```

209. Lorsqu'on veut appeler une fonction dans un main, il faut re-initialiser les arguments de cette derniere avant de les passer à la fonction. Exemple :
On initialise dans la déclaration de la fonction ses parametres :
void min_max_moy(int tab[], int taille, int *min, int *max, float *moy)
*min = etc
ET DANS LE MAIN, quand on veut appeler la fonction :
int t[TAILLE]={1,2,3,4,0,-1};
int min,max;
float moy;
unsigned int i;
min_max_moy(t,TAILLE,&min,&max,&moy);

210. fabs() ou abs() retourne la valeur absolue d'un nombre