UNIVERSITÉ DE NANTES

POLYTECH NANTES

Langage procédural (C)
Partie 1

Support de Cours
Dpt Informatique Polytech'Nantes
Fabien Picarougne

UNIVERSITÉ DE NANTES

POLYTECH NANTES

Partie I

Bases de la programmation en C (Accueil)

Bases de la programmation en C : Plan

- Introduction / historique
- Éléments fondamentaux
  - Pourquoi un langage ? Compilation et étapes de compilation
  - Structure d'un programme en C
- Variables, types de données
  - Élémentaires, constantes
  - Tableaux et chaînes de caractères
- Les fonctions 1
  - Déclaration et définition, appel de fonction
  - Passage de paramètres, paramètres de ligne de commande
- Entrées/Sorties 1
  - printf, scanf
- Expressions et opérateurs
  - Arithmétiques, comparaison, affectation, logiques, opérateurs de bits, …
  - Dualité opérateur-expression, tableau des priorités et parenthésage
- Structures de contrôle de flux
  - Instructions alternatives, répétitives, de branchement
- Structures
  - Structure classique, union, champ de bits, …

Introduction

- Historique
  - Création du langage : 1969-1973
    - Dennis M. Ritchie et Brian W. Kernighan (Bell labs.)
    - Successeur des langages B et BCPL
    - Langage système : codage de Unix
      - Proche de la machine
      - Rapide
  - À partir de 1980 : C devient populaire -> plusieurs compilateurs
  - Proche de la machine
    - Standard informel dicté par le K&R
    - ANSI-C (American National Standards Institute) : 1988 (début de la normalisation en 1983)
    - ANSI/ISO Standard : ISO/IEC 9899:1990
    - Actuellement : ANSI/ISO C99

Introduction

- Raisons d'apprendre le C
  - Langage puissant
  - Programmes rapides
  - Meilleure connaissance de l'ordinateur
  - Ancêtre d'autres *(beaucoup)* langages (syntaxe/sémantique proches) :
    - C++
    - Java
  - Base installée très importante (trans-discipline)
  - Présent sur (presque) toutes les machines
    - Langage très portable

- Mais mise en garde
  - C = langage puissant
  - Mauvaise utilisation aisée
  - Syntaxe régulière et grande expressivité
  - Écriture de code illisible facile

Introduction

- Mise en garde
  - Tetris sur VT100

```
long h[4];t(){h[3]-=h[3]/3000;setitimer(0,h,0);}c,d,l,v[]={(int)t,0,2},w,s,I,K=0,i=276,j,k,q[276],Q[276],*n=q,*m,x=17,f[]={7,-13,-12,1,8,-11,-12,-1,9,-1,1,12,3,-13,-12,-1,12,-1,11,1,15,-1,13,1,18,-1,1,2,0,-12,-1,11,1,-12,1,13,10,-12,1,12,11,-12,-1,1,2,-12,-1,12,13,-12,12,13,14,-11,-1,1,4,-13,-12,12,16,-11,-12,12,17,-13,1,-1,5,-12,12,11,6,-12,12,24};u(){for(i=11;++i<264;)if((k=q[i])-Q[i]){Q[i]=k;if(i-++I||i%12<1)printf("\033[%d;%dH",(I=i)/12,i%12*2+28);printf("\033[%dm  "+(K-k?0:5),k);K=k;}Q[263]=c=getchar();}G(b){for(i=4;i--;)if(q[i?b+n[i]:b])return 0;return 1;}g(b){for(i=4;i--;q[i?x+n[i]:x]=b);}main(C,V,a)char**V,*a;{h[3]=1000000/(l=C>1?atoi(V[1]):2);for(a=C>2?V[2]:"jkl pq";i;i--)*n++=i<25||i%12<2?7:0;srand(getpid());system("stty cbreak -echo stop u");sigvec(14,v,0);t();puts("\033[H\033[J");for(n=f+rand()%7*4;;g(7),u(),g(0)){if(c<0){if(G(x+12))x+=12;else{g(7);++w;for(j=0;j<252;j=12*(j/12+1))for(;q[++j];)if(j%12==10){for(;j%12;q[j--]=0);u();for(;--j;q[j+12]=q[j]);u();}n=f+rand()%7*4;G(x=17)||(c=a[5]);}}if(c==*a)G(--x)||++x;if(c==a[1])n=f+4**(m=n),G(x)||(n=m);if(c==a[2])G(++x)||--x;if(c==a[3])for(;G(x+12);++w)x+=12;if(c==a[4]||c==a[5]){s=sigblock(8192);printf("\033[H\033[J\033[0m%d\n",w);if(c==a[5])break;for(j=264;j--;Q[j]=0);while(getchar()-a[4]);puts("\033[H\033[J\033[7m");sigsetmask(s);}}d=popen("stty -cbreak echo stop \023;sort -mnr -o HI - HI;cat HI","w");fprintf(d,"%4d from level %1d by %s\n",w,l,getlogin());pclose(d);}
```

  - Portabilité nulle
  - Maintenance difficile

Bases de la programmation en C : Plan

- Introduction / historique
- Éléments fondamentaux
  - Pourquoi un langage ? Compilation et étapes de compilation
  - Structure d'un programme en C
- Variables, types de données
  - Élémentaires, constantes
  - Tableaux et chaînes de caractères
- Les fonctions 1
  - Déclaration et définition, appel de fonction
  - Passage de paramètres, paramètres de ligne de commande
- Entrées/Sorties 1
  - printf, scanf
- Expressions et opérateurs
  - Arithmétiques, comparaison, affectation, logiques, opérateurs de bits, …
  - Dualité opérateur-expression, tableau des priorités et parenthésage
- Structures de contrôle de flux
  - Instructions alternatives, répétitives, de branchement
- Structures
  - Structure classique, union, champ de bits, …

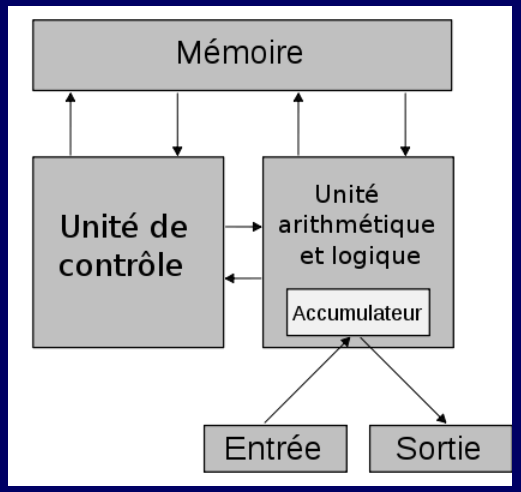
Éléments fondamentaux

- Machine de Von Neumann *(1945)*
  - Modèle de machine universelle possédant
    - Une mémoire *(contient instructions et données)*
    - Une unité arithmétique et logique (ALU) *(effectue les calculs)*
    - Une unité d'entrées/sorties (I/O) *(échange d'informations avec les périphériques)*
    - Unité de commande (UC) *(contrôle)*

  - Ces dispositifs permettent la mise en œuvre des fonctions de base d'un ordinateur
    - Le stockage de données
    - Le traitement des données
    - Le mouvement des données
    - Le contrôle des données

Éléments fondamentaux

- Machine Von Neumann : fonctionnement simplifié
  - L'UC
    - extrait une instruction de la mémoire,
    - analyse l'instruction,
    - recherche dans la mémoire les données concernées par l'instruction,
    - déclenche l'opération adéquate sur l'ALU ou l'E/S,
    - range au besoin le résultat dans la mémoire



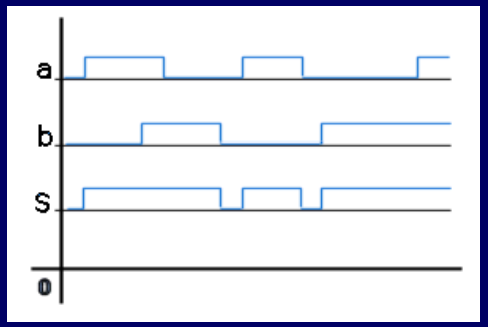
Source Wikipedia

- La plupart des machines actuelles s'appuient sur le modèle Von Neumann

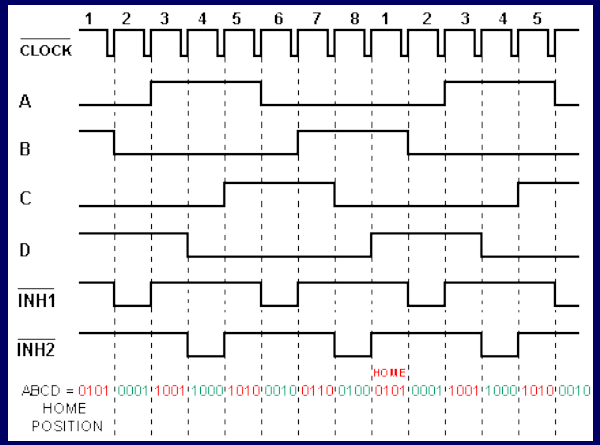
- Implémentation physique
  - Signal et chronogramme
  - Horloge
  - Registres
  - Bus

Éléments fondamentaux

- Signal et chronogramme
  - Un signal est une grandeur discrète appartenant à [0,1]
  - un chronogramme est la représentation graphique d'un signal évoluant dans le temps



- Horloge
  - Utilisée pour synchroniser l'ensemble des dispositifs logiques d'un ordinateur
    - Cadencement des instructions à fréquence constante
    - L'horloge divise le temps en battements de même durée appelés cycles
      - Ex : horloge 2GHz = cycles 0,5 nano sec



Éléments fondamentaux

- Registres
  - Éléments de mémoire rapide internes au CPU
  - En nombre très limité
    - Plus petit est l'espace de recherche, plus rapide est l'accès à l'information.
  - Mémorisation d'un élément de la mémoire centrale en vue d'une utilisation par l'UC
    - commandée par un signal de chargement

- Bus
  - Ensemble de fils électriques sur lesquels transitent les informations entre les unités
  - Largeur du bus
    - nombre de fils constituant le chemin
    - nombre d'impulsions électriques pouvant être envoyés en parallèle (en même temps)

Éléments fondamentaux

- Fonctionnement d'un processeur
  - Série d'opérations (instructions) codées dans le silicium (ALU)
  - UC lit une séquence d'instructions placées dans la mémoire centrale
  - Communication UC – ALU via les registres
- Registres processeur *(UC)*
  - Compteur ordinal (PC)
    - Registre contenant l'adresse mémoire de l'instruction à exécuter
  - Registre d'instruction (RI)
    - Mémorise l'instruction (une instruction est composée de plusieurs parties, ou champs)
- Jeu d'instructions
  - Ensemble des instructions exécutables sur une machine
  - Plusieurs familles (x86, x64, MMX, SSE, …)

Éléments fondamentaux

- Qu'est-ce qu'un programme ?
  - Série d'instructions et de données (variables) à transformer
  - Codés en binaire dans la mémoire centrale
    - Appelé code machine
    - Ex : 00100101 01111001 11001001 10010001 00010010 11000111

  - Difficile de créer un programme
    - Langage binaire loin d'être un langage courant pour un humain

  - Création d'un langage intermédiaire plus compréhensible : langage assembleur
    - Conversion directe du langage assembleur en code machine
    - Utilisation d'un outil de transformation langage assembleur -> code binaire : l'assembleur
    - Ex : a = b+c+d+e

```
add a, b, c # la somme de b et c est placée dans a
add a, a, d # la somme de b, c et d est dans a
add a, a, e # la somme de b, c, d et e est dans a
```

Éléments fondamentaux

- Problèmes
  - Difficultés de programmation en langage assembleur
  - Programmes peu lisibles
  - Réutilisabilité de code difficile
  - Nécessité d'écrire un nouveau programme pour chaque type de processeur
    - Jeu d'instruction différent
    - Codage mémoire différent *(little endian, big endian, …)*

- Solution
  - Créer un langage de plus haut niveau
    - Indépendant de l'architecture processeur
    - Plus proche du langage naturel humain
  - Nécessite un traducteur du nouveau langage vers le langage assembleur (ou directement vers le code machine)
    - Rôle du compilateur

Éléments fondamentaux

- Exemple
  - Code en langage C

```
int main(void)
{
    int a,b,c,d,e;
    b=1;c=2;d=3;e=4;
    a=b+c+d+e;
    return 0;
}
```

  - Traduction en assembleur par le compilateur C

```
mov b, 1
mov c, 2
mov d, 3
mov e, 4
add a, b, c
add a, a, d
add a, a, e
```

Éléments fondamentaux

- Étapes de compilation
  - Programme en C = suite de fichiers textes *(ex : prgm.c)*
    - Il est pratique de diviser l'écriture d'un programme en plusieurs fichiers sources
    - Réutilisabilité de portion de programmes
  - Compilation de chaque fichier source en fichier contenant le code machine (fichier objet)
    - Appel au compilateur C
    - Ex : `gcc -c source.c` donne en sortie *source.o*
  - Edition de lien entre tous les fichiers objet pour combiner les parties d'un programme en un seul exécutable
    - Appel à l'éditeur de lien *(linker)*
    - Ex : `gcc source.o -o prgm` donne en sortie le programme *prgm*
  - Il est possible de réaliser toutes les opérations en une seule commande
    - Ex : `gcc source1.c source2.c -o prgm` donne en sortie le programme *prgm*
  - Le compilateur possède de nombreuses options d'optimisation

Éléments fondamentaux

- Exécution d'un programme en C
  - Un programme est exécuté par le système d'exploitation (SE)
  - Le processeur a besoin de connaitre l'adresse de la 1ère instruction
  - Le SE charge un programme en mémoire
    - Recopie directement dans la mémoire centrale le contenu du fichier contenant le code machine (le programme)
    - Le SE charge le compteur ordinal (PC) avec l'adresse de la 1ère instruction
      - » Besoin de connaitre quelle est la 1ère instruction
      - » Notion de point d'entrée
  - Dans un exécutable il n'existe qu'un seul point d'entrée
    - *Plusieurs dans le cas d'une librairie*
    - En C, point d'entrée sous forme d'une fonction : *main*

Éléments fondamentaux

- Structure d'un programme en C
  - Exemple

```
#include <stdio.h>

#define ARRET '0'

/* Reconnaissance des voyelles/consonnes
en minuscules. Note : on suppose que
l'utilisateur ne rentre que des lettres*/

int main(void)
{
    char lettre;
    do
    {
        printf("Donnez une lettre : ");

        // Lecture sur l'entrée courante
        lettre = getchar();

        if (lettre == 'a' || lettre == 'e' ||
            lettre == 'i' || lettre == 'o' ||
            lettre == 'u' || lettre == 'y')
        {
            printf("C'est une voyelle\n");
        }
        else
        { // Tous les autres cas
            printf("C'est une consonne\n");
        }
    } while (lettre != ARRET);

    return 0;
}
```

- Déclaration de fonctions (`main` = réservée)
- Définition de macros
- Commentaires
- Déclaration de variables
- Liste d'instructions séparées par ;
- Boucle
- Entrées/sorties *(chaînes et caractères)*

Éléments fondamentaux

- Remarques
  - La rapidité des programmes écrits en C est en grande partie due au fait que le compilateur présuppose que le programmeur sait ce qu'il fait
    - Il génère un code ne contenant pas de vérifications sur la validité des pointeurs, l'espace d'adressage, etc
  - Ainsi, les programmes en C sont très compacts
  - C est un langage « faiblement typé »
    - Les types de données qu'il manipule sont très restreints et proches de la représentation interne du processeur
    - Ex : le type « chaîne de caractères » n'existe pas en C

Bases de la programmation en C : Plan

- Introduction / historique
- Éléments fondamentaux
  - Pourquoi un langage ? Compilation et étapes de compilation
  - Structure d'un programme en C
- Variables, types de données
  - Élémentaires, constantes
  - Tableaux et chaînes de caractères
- Les fonctions 1
  - Déclaration et définition, appel de fonction
  - Passage de paramètres, paramètres de ligne de commande
- Entrées/Sorties 1
  - printf, scanf
- Expressions et opérateurs
  - Arithmétiques, comparaison, affectation, logiques, opérateurs de bits, …
  - Dualité opérateur-expression, tableau des priorités et parenthésage
- Structures de contrôle de flux
  - Instructions alternatives, répétitives, de branchement
- Structures
  - Structure classique, union, champ de bits, …

Variables, types de données

- On appelle variable une mémoire de l'ordinateur à laquelle on a donné un nom
  - Le programmeur peut stocker une valeur dans la variable et la modifier au cours du programme

- Il existe plusieurs types de variables stockant différentes données
  - Nombre entier
  - Nombre réel
  - Caractère
  - Pointeurs
  - Tableaux
  - Structures

- La notion de type permet
  - De différencier l'occupation mémoire nécessaire au stockage des données
  - Décide du format de représentation de la variable
  - Permet au compilateur de vérifier que le programmeur ne se trompe pas
    - Dans la syntaxe, mais pas dans la fonction du programme

Variables, types de données

- Règles de nommage d'une variable
  - Peut être formé de lettres (A à Z ; a à z), de chiffres et du caractère _ (souligné)
  - Le 1er caractère est nécessairement une lettre ou _
    - Pas de chiffre
    - Ex : `valeur1` ou `_valeur` sont possibles, mais pas `1ere_valeur`
  - Le C tient compte de la casse des lettres
    - Les minuscules sont considérés différentes des majuscules
    - Ex : `valeur1` est différent de `Valeur1`
  - Il est interdit d'utiliser un caractère blanc (espace, tabulation, retour chariot, un commentaire) dans un nom de variable
  - La plupart des compilateurs acceptent n'importe quelle longueur d'identificateurs (tout en restant sur la même ligne) mais seuls les 32 premiers caractères sont significatifs

Variables, types de données

- Mots clés réservés
  - Il est interdit de nommer en C une variable avec un des noms suivants

```
auto        enum        restrict    unsigned
break       extern      return      void
case        float       short       volatile
char        for         signed      while
const       goto        sizeof
continue    if          static
default     inline      struct
do          int         switch
double      long        typedef
else        register    union
```

  - Ces noms sont réservés pour le fonctionnement du langage C

Variables, types de données

- On peut les catégoriser de la manière suivante
  - les spécificateurs de stockage

    `auto register restrict static extern typedef`

  - les spécificateurs de type

    `char double enum float int long short signed struct union unsigned void`

  - les qualificateurs de type

    `const volatile`

  - les instructions de contrôle

    `break case continue default do else for goto if switch while`

  - Divers

    `return sizeof`

Variables, types de données

- Déclaration d'une variable
  - La construction d'une variable se fait en 2 étapes
    - Déclaration : indication de l'existence du nom, de sa qualité (variable, fonction, …), de son type, …
    - Définition : pour une variable, allocation de l'espace mémoire nécessaire ; pour une fonction, l'algorithme
    - *Un identificateur peut être déclaré plusieurs fois, mais ne peut être défini qu'une seule fois*
- Tout identificateur doit être déclaré avant son utilisation
- L'endroit où le compilateur a choisi de mettre la variable est appelé adresse de la variable
  - Il est possible de stocker cette adresse dans une variable de type pointeur

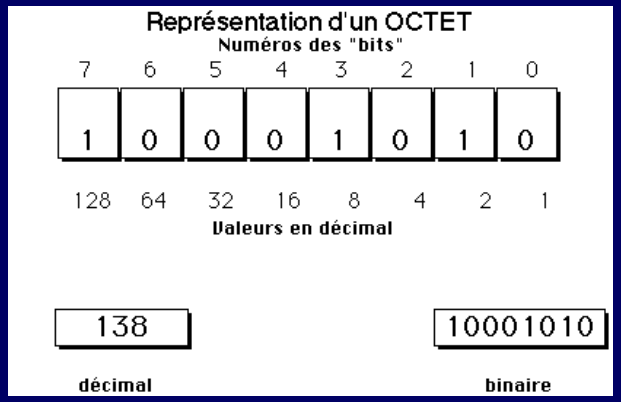
Variables, types de données

- Types de variables
  - On divise les types de variables en 2 groupes élémentaires
    - Représentation des nombres entiers : types entiers
    - Représentation des nombres réels : types à virgule flottante

- Types entiers
  - En C on trouve les types entiers `char`, `int`, `short` et `long`
  - La différence entre ces types réside dans leur occupation mémoire
  - Ces types peuvent être préfixés pour être signés (`signed`) ou non signés (`unsigned`)
    - Sans spécification, le compilateur les suppose signés
    - `signed` seul signifie `signed int`
    - `unsigned` seul signifie `unsigned int`

Variables, types de données

- Type `char`
  - Permet de représenter un nombre sur 1 octet
    - Mais occupe généralement dans la mémoire une place de 4 octets due à l'alignement sur 32 bits
    - Si le type utilisé est signé (`signed char` ou `char`), le bit de poids fort représente le signe du nombre
      - » représentation de nombres variant de -128 à 127
    - Si le type utilisé est non signé (`unsigned char`), pas de bit de signe
      - » Représentation de nombres variant de 0 à 255



  - Utilisé pour représenter un caractère
    - Utilisation de la table ASCII pour associer un nombre à un caractère
    - En informatique, un caractère est avant tout un nombre

Variables, types de données

- Table ASCII et ASCII étendue *(pour le latin1)*

REGULAR ASCII CHART (character codes 0 – 127)

000d	00h		(nul)	016d	10h	►	(dle)	032d	20h	␣	048d	30h	0	064d	40h	@	080d	50h	P	096d	60h	‘	112d	70h	p
001d	01h	☺	(soh)	017d	11h	◄	(dc1)	033d	21h	!	049d	31h	1	065d	41h	A	081d	51h	Q	097d	61h	a	113d	71h	q
002d	02h	☻	(stx)	018d	12h	↕	(dc2)	034d	22h	"	050d	32h	2	066d	42h	B	082d	52h	R	098d	62h	b	114d	72h	r
003d	03h	♥	(etx)	019d	13h	‼	(dc3)	035d	23h	#	051d	33h	3	067d	43h	C	083d	53h	S	099d	63h	c	115d	73h	s
004d	04h	♦	(eot)	020d	14h	¶	(dc4)	036d	24h	$	052d	34h	4	068d	44h	D	084d	54h	T	100d	64h	d	116d	74h	t
005d	05h	♣	(enq)	021d	15h	§	(nak)	037d	25h	%	053d	35h	5	069d	45h	E	085d	55h	U	101d	65h	e	117d	75h	u
006d	06h	♠	(ack)	022d	16h	▬	(syn)	038d	26h	&	054d	36h	6	070d	46h	F	086d	56h	V	102d	66h	f	118d	76h	v
007d	07h	•	(bel)	023d	17h	↨	(etb)	039d	27h	'	055d	37h	7	071d	47h	G	087d	57h	W	103d	67h	g	119d	77h	w
008d	08h	◘	(bs)	024d	18h	↑	(can)	040d	28h	(	056d	38h	8	072d	48h	H	088d	58h	X	104d	68h	h	120d	78h	x
009d	09h		(tab)	025d	19h	↓	(em)	041d	29h	)	057d	39h	9	073d	49h	I	089d	59h	Y	105d	69h	i	121d	79h	y
010d	0Ah	◙	(lf)	026d	1Ah		(eof)	042d	2Ah	*	058d	3Ah	:	074d	4Ah	J	090d	5Ah	Z	106d	6Ah	j	122d	7Ah	z
011d	0Bh	♂	(vt)	027d	1Bh	←	(esc)	043d	2Bh	+	059d	3Bh	;	075d	4Bh	K	091d	5Bh	[	107d	6Bh	k	123d	7Bh	{
012d	0Ch		(np)	028d	1Ch	└	(fs)	044d	2Ch	’	060d	3Ch	<	076d	4Ch	L	092d	5Ch	\	108d	6Ch	l	124d	7Ch	|
013d	0Dh	♪	(cr)	029d	1Dh	↔	(gs)	045d	2Dh	-	061d	3Dh	=	077d	4Dh	M	093d	5Dh	]	109d	6Dh	m	125d	7Dh	}
014d	0Eh	♫	(so)	030d	1Eh	▲	(rs)	046d	2Eh	.	062d	3Eh	>	078d	4Eh	N	094d	5Eh	^	110d	6Eh	n	126d	7Eh	~
015d	0Fh	☼	(si)	031d	1Fh	▼	(us)	047d	2Fh	/	063d	3Fh	?	079d	4Fh	O	095d	5Fh	_	111d	6Fh	o	127d	7Fh	⌂

EXTENDED ASCII CHART (character codes 128 – 255) LATIN1/CP1252

128d	80h	€		144d	90h			160d	A0h		176d	B0h	°	192d	C0h	À	208d	D0h	Ð	224d	E0h	à	240d	F0h	ð
129d	81h			145d	91h	‘		161d	A1h	¡	177d	B1h	±	193d	C1h	Á	209d	D1h	Ñ	225d	E1h	á	241d	F1h	ñ
130d	82h	‚		146d	92h	’		162d	A2h	¢	178d	B2h	²	194d	C2h	Â	210d	D2h	Ò	226d	E2h	â	242d	F2h	ò
131d	83h	ƒ		147d	93h	“		163d	A3h	£	179d	B3h	³	195d	C3h	Ã	211d	D3h	Ó	227d	E3h	ã	243d	F3h	ó
132d	84h	„		148d	94h	”		164d	A4h	¤	180d	B4h	´	196d	C4h	Ä	212d	D4h	Ô	228d	E4h	ä	244d	F4h	ô
133d	85h	…		149d	95h	•		165d	A5h	¥	181d	B5h	µ	197d	C5h	Å	213d	D5h	Õ	229d	E5h	å	245d	F5h	õ
134d	86h	†		150d	96h	–		166d	A6h	¦	182d	B6h	¶	198d	C6h	Æ	214d	D6h	Ö	230d	E6h	æ	246d	F6h	ö
135d	87h	‡		151d	97h	—		167d	A7h	§	183d	B7h	·	199d	C7h	Ç	215d	D7h	×	231d	E7h	ç	247d	F7h	÷
136d	88h	ˆ		152d	98h	˜		168d	A8h	¨	184d	B8h	¸	200d	C8h	È	216d	D8h	Ø	232d	E8h	è	248d	F8h	ø
137d	89h	‰		153d	99h	™		169d	A9h	©	185d	B9h	¹	201d	C9h	É	217d	D9h	Ù	233d	E9h	é	249d	F9h	ù
138d	8Ah	Š		154d	9Ah	š		170d	AAh	ª	186d	BAh	º	202d	CAh	Ê	218d	DAh	Ú	234d	EAh	ê	250d	FAh	ú
139d	8Bh	‹		155d	9Bh	›		171d	ABh	«	187d	BBh	»	203d	CBh	Ë	219d	DBh	Û	235d	EBh	ë	251d	FBh	û
140d	8Ch	Œ		156d	9Ch	œ		172d	ACh	¬	188d	BCh	¼	204d	CCh	Ì	220d	DCh	Ü	236d	ECh	ì	252d	FCh	ü
141d	8Dh			157d	9Dh			173d	ADh		189d	BDh	½	205d	CDh	Í	221d	DDh	Ý	237d	EDh	í	253d	FDh	ý
142d	8Eh	Ž		158d	9Eh	ž		174d	AEh	®	190d	BEh	¾	206d	CEh	Î	222d	DEh	Þ	238d	EEh	î	254d	FEh	þ
143d	8Fh			159d	9Fh	Ÿ		175d	AFh	¯	191d	BFh	¿	207d	CFh	Ï	223d	DFh	ß	239d	EFh	ï	255d	FFh	ÿ

Hexadecimal to Binary

0	0000	4	0100	8	1000	C	1100
1	0001	5	0101	9	1001	D	1101
2	0010	6	0110	A	1010	E	1110
3	0011	7	0111	B	1011	F	1111

Groups of ASCII-Code in Binary

Bit 6	Bit 5	Group
0	0	Control Characters
0	1	Digits and Punctuation
1	0	Upper Case and Special
1	1	Lower Case and Special

Variables, types de données

- Types `short`, `int`, `long`
  - Ces 3 types de variables désignent des entiers avec une précision plus grande que le type `char`
    - Stockage sur 2 ou 4 octets sur un processeur 32 bits
  - Le type `short` stocke les nombres sur 2 octets
    - Valeurs comprises entre -32 768 et 32 767 pour un `short`
    - Valeurs comprises entre 0 et 65 535 pour un `unsigned short`
  - Le type `int` stocke les nombres sur 4 octets sur un système 32 bits
    - Valeurs comprises entre -2 147 483 648 et 2 147 483 647 pour un `int`
    - Valeurs comprises entre 0 et 4 294 967 296 pour un `unsigned int`
  - Le type `long` est équivalent à un type `int` sur un système 32 bits
    - Codage sur 64 bits sur un système 64 bits
    - Codage sur 64 bits du type `long long` sur système 32 bits

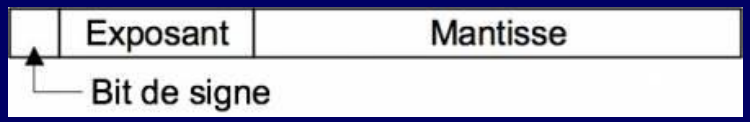
Variables, types de données

- Types réels
  - En C on trouve les types réels `float` et `double`
  - Les nombres décimaux nécessite un codage particulier
    - Stockage dit à virgule flottante
  - La position de la virgule en tant que séparateur de la partie entière et décimale n'est pas figée
  - La grandeur d'un tel nombre est donné par un exposant
    - Ex : 13,5 peut être représenté de plusieurs manières

      $1{,}35 * 10^{1}$

      $0{,}135 * 10^{2}$

      $135{,}0 * 10^{-1}$
  - Les nombres à virgule flottante se décomposent en 3 parties
    - Une mantisse codant la valeur du nombre
    - Un exposant
    - Un signe

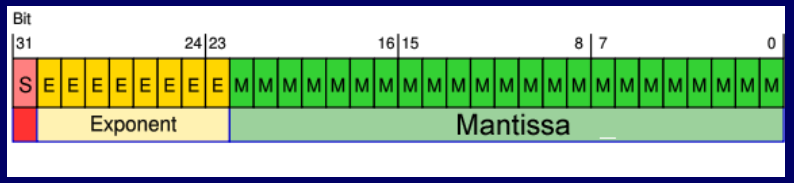


    - L'interprétation du nombre est

$valeur = signe \times 1,mantisse \times 2^{exposant - (2^{(e-1)}-1)}$, avec $signe = \pm 1$ (e représente le nombre de bits de l'exposant)

Variables, types de données

- Type `float`
  - Codé sur 4 octets
    - Mantisse sur 23 bits
    - Exposant sur 8 bits
    - Signe sur 1 bit



  - Attention, la précision est donc inférieure à celle d'un entier `int`

- Type `double`
  - Codé sur 8 octets
    - Mantisse sur 52 bits
    - Exposant sur 11 bits
    - Signe sur 1 bit
  - Même interprétation que pour un `float`

- Les nombres à virgule flottante sont des nombres approchés
  - Attention à l'utilisation de `float`
  - Ne jamais comparer 2 flottants (`float a,b;`)
    - `a==b` donnera un résultat aléatoire, dépendant des arrondis
    - Préférer `fabs(a-b)<EPS  // EPS : epsilone dépendant de la précision du système`

Variables, types de données

- Types de variables
  - Occupation mémoire et domaine de valeur

Type	Signification	Taille	Plage de variation
char	caractère	1	-2^7 à $2^7 - 1$
unsigned char	caractère non signé	1	0 à $2^8 - 1$
short [int]	entier court	2	-2^8 à $2^8 - 1$
unsigned short [int]	entier court non signé	2	0 à 2^{16}
int	entier	4	-2^{31} à $2^{31} - 1$
unsigned [int]	entier non signé	4	0 à $2^{32} - 1$
long [int]	entier long	4	-2^{-31} à $2^{31} - 1$
unsigned long [int]	entier long non signé	4	0 à 2^{32}
float	réel	4	3.4×10^{-38} à 3.4×10^{38}
double	réel double précision	8	1.7×10^{-308} à 1.7×10^{308}
long double	réel long double précision	10	3.4×10^{-4932} à 3.4×10^{4932}

Variables, types de données

- Typage des constantes numériques
  - Types entiers

100	`int`
100u	`unsigned int` ou `unsigned long`
100l	`long`
100lu	`unsigned long`

  - Indications préfixées :
    - Nombre précédé de `0` : octal
    - Nombre précédé de `0x` : hexadécimal
    - Sinon : décimal

    - Ex :

```
0100   // Octal. Valeur décimale = 64
0x12   // Hexadécimal. Valeur décimale = 18
14     // Décimal. Valeur décimale = 14
```

  - Types réels

1.35	`double`
1.35f	`float`
13.5e-1	`double`

Variables, types de données

- Typage des constantes caractère
  - Une constante caractère est un symbole appartenant à l'ensemble des caractères représentables
    - Symbole inclut entre des apostrophes
    - Ex : `'a','?','_','1'`
    - Chaque symbole représente un nombre correspondant à la valeur ASCII du caractère utilisé
    - Pour coder le caractère `'`, il faut utiliser le caractère d'échappement `\`
    - Ex : `'\'','\\'`
  - Une constante de type chaîne (string) est une suite de caractères placée entre guillemets
    - Ex : `"Ceci est une chaîne de caractère"`
    - Pour coder le caractère `"`, il faut utiliser le caractère d'échappement `\`
    - Ex : `"Chaîne de caractère avec \"échappement\""`

Variables, types de données

- Un certain nombre de caractères particulier sont accessibles via le caractère d'échappement
  - Tabulation : `'\t'`
  - Retour chariot : `'\n'`
  - Retour au début de ligne : `'\r'`
  - Trait oblique : `'\\'`
  - Saut de page (imprimante) : `'\f'`
  - Curseur arrière : `'\b'`
  - Fin d'une chaine de caractère : `'\0'`
  - Point d'interrogation : `'\?'`
  - Guillemets : `'\"'`
  - Tabulation verticale : `'\v'`

Variables, types de données

- Utilisation des variables
  - Toute variable doit être déclarée avant d'être utilisée
  - Syntaxe : `Type nomVariable;`
    - Dans le cas standard, la déclaration et l'instantiation sont effectuées par cette instruction
    - Ex : `double rayon;`
  - Une variable peut être initialisée lors de la déclaration
  - Syntaxe : `Type nomVariable = valeur;`
    - Ex : `int val = 34;`
  - Une variable non initialisée a une valeur indéfinie
  - Il est possible de déclarer plusieurs variables à la fois
    - Ex : `double x = 23.5, y, z = 12.7*x, t;`
  - L'opérateur = (affectation) permet d'affecter une nouvelle valeur à une variable
    - Ex : `x = 10.33;`
    - Attention, il ne s'agit pas ici d'une comparaison mais d'une affectation

Variables, types de données

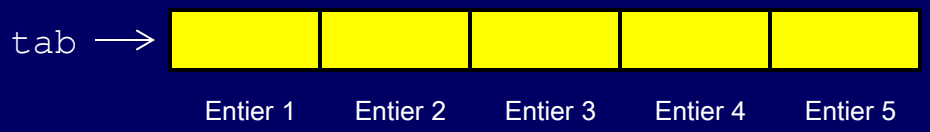
- Les tableaux
  - Variable qui se compose de plusieurs données élémentaires de même type
  - Chaque donnée élémentaire est elle-même une variable
  - Le type peut-être n'importe quel type du langage C

  - Tableau unidimensionnel
    - Composé d'éléments qui ne sont pas des tableaux
    - Déclaration

```
TypeDeBase NomTableau[NbCases];
```

    - Le nombre d'élément est fixé à la création du tableau
      - » ne pourra pas être modifié par la suite
    - Le nom du tableau est la variable tableau
    - Les données sont écrites de manière continues en mémoire
      - » Ex :

```
int tab[5];        // déclare un tableau de 5 entiers
```



Variables, types de données

- Les tableaux
  - Accès à un élément du tableau
    - Opérateur d'indexation : [index]
    - index peut être variable
      - » Ex : `tab[i+1];`
  - Les tableaux sont indicés à 0
    - 1er élément `tab[0]`
    - Dernier élément `tab[NbCases-1]`
  - Opérateur d'indexation retourne une l-value
    - Possibilité d'accès en lecture à la variable du tableau
      - » Ex : `val = tab[i+1];`
    - Possibilité d'accès en écriture à la variable du tableau
      - » Ex : `tab[i+1] = val;`
  - Attention à ne pas dépasser les bornes du tableau !
    - Aucune vérification par le compilateur
  - Ex : initialisation d'un tableau

```
for (i=0;i<5;i++)
   tab[i]=0;
```

équivaut à

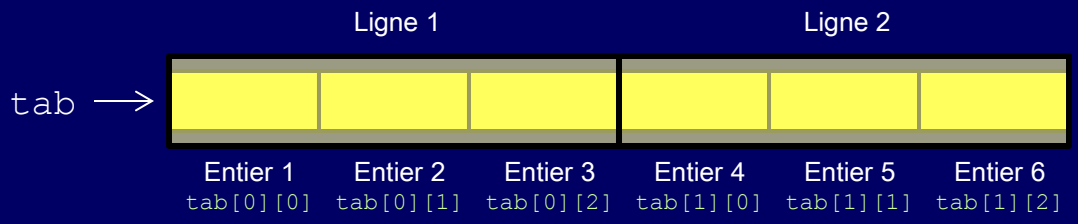
```
tab[0]=tab[1]=tab[2]=tab[3]=tab[4]=0
```

Variables, types de données

- Tableau multidimensionnels
  - Composé d'éléments qui sont des tableaux
  - Déclaration
    - `TypeDeBase NomTableau[NbCasesDim1][NbCasesDim2];`

  - Il est possible de créer autant de dimensions que nécessaire
    - Limité seulement par la capacité mémoire
    - La mémoire est un tableau unidimensionnel
      - Stockage d'un tableau multidimensionnel en *collant* les dimensions
      - Ex :

```
int tab[2][3];  // déclare une matrice de 2 lignes de 3 colonnes
```



Variables, types de données

- Initialisation d'un tableau

```
// Tableau unidimensionnel
int tab[5] = {1, 4, 5, 3, 8};

// Tableau multidimensionnel
int tab[2][3] = {{3, 4, 2},{6, 5, 1}};
int tab[2][3] = {3, 4, 2, 6, 5, 1};

// Calcul automatique de la taille
char V[] = {'i','n','f','2','2'};
char *V = {'i','n','f','2','2'};
```

Variables, types de données

- Les chaînes de caractères
  - Il n'existe pas en C de type chaîne de caractère
    - Il n'existe pas de type chaîne dans le processeur
  - Une chaîne de caractère est une suite de caractères
    - Implémentation dans un tableau de caractères
    - Indication de la fin de chaîne par un caractère spécial `'\0'` (code ASCII 0)
      - » Ex :

```
char str[7] = {'C', 'h', 'a', 'i', 'n', 'e' '\0'};
char str[] = {'C', 'h', 'a', 'i', 'n', 'e' '\0'};
char str[] = "Chaine";
```

    - Terminateur `'\0'` rajouté automatiquement avec "

Variables, types de données

- Fonctions sur les chaînes

```
unsigned int strlen(const char *s)
// Retourne le nombre de caractères de la chaîne s (non compris
   le terminateur '\0')

int strcmp(const char *s1, const char* s2)
// Compare les chaînes s1 et s2. Retourne 0 si les deux chaînes sont
   égales et un nombre différent de 0 sinon

char *strcpy(char *dest, const char *src)
// Recopie la chaîne src dans le tableau pointé par dest

char *strcat(char *dest, const char *src)
// Concatène src à dest
```

Variables, types de données

- Portée d'une variable
  - Il est possible de définir en C un groupe d'instructions
    - Forment un block d'instructions
    - Block délimité par { }
    - Un bloc est une (macro-)instruction simple
    - Un bloc prend la place d'une instruction et du ; qui suit

  - Les variables sont définies dans un bloc donné
    - Validité d'une variable
      - » Commence à la ligne de sa déclaration
      - » Termine à l'accolade fermante du block de définition de la variable
        - » Ex :

```
int main()
{
    int i,j=0;    // i est indéfini, j=0

    for (i=0;i<3;i++)
    {
      int j,k;    // nouvelle variable j et k

      j=i+4;      // j=4, puis 5, puis 6
    }

    // j et k n'existent plus
}
```

Variables, types de données

- Portée d'une variable
  - Possibilité de définir des variables globales
    - Durée de vie : tout le programme
    - Se déclare avant toute définition de fonction (haut du fichier source)
    - A éviter si possible
      - » Nuit à la lisibilité du code
      - » Ex:

```
int a=10;          /* Déclaration et définition de la variable
                      globale a */

int main()
{
    int b;

    b=a+4;         // La variable a est connue et a pour valeur 10
}
```

Variables, types de données

- Définition de types personnalisées
  - Possibilité de redéfinir des types
  - Instruction `typedef`
    - Ex :

```
#define VRAI 1
#define FAUX 0
typedef int BOOLEAN;

// On pourra par la suite déclarer des « booléens »
BOOLEAN b1,b2;

// et les utiliser
b1 = VRAI;
if (b2 == FAUX) …;
```

Variables, types de données

- Création d'énumération
  - Il n'est pas rare qu'une variable ne doive prendre qu'un nombre limité de valeur
  - Le compilateur peut les vérifier si le programmeur les spécifie
  - Définition d'énumération : `enum`

    - Ex :

```
enum saison
{
    Printemps,          // Séparation par des ,
    Été,
    Automne=4,          // Possibilité d'affecter une valeur
    Hiver
}; // Ne pas oublier le ;

enum saison ma_saison=Automne;
```

Bases de la programmation en C : Plan

- Introduction / historique
- Éléments fondamentaux
  - Pourquoi un langage ? Compilation et étapes de compilation
  - Structure d'un programme en C
- Variables, types de données
  - Élémentaires, constantes
  - Tableaux et chaînes de caractères
- Les fonctions 1
  - Déclaration et définition, appel de fonction
  - Passage de paramètres, paramètres de ligne de commande
- Entrées/Sorties 1
  - printf, scanf
- Expressions et opérateurs
  - Arithmétiques, comparaison, affectation, logiques, opérateurs de bits, …
  - Dualité opérateur-expression, tableau des priorités et parenthésage
- Structures de contrôle de flux
  - Instructions alternatives, répétitives, de branchement
- Structures
  - Structure classique, union, champ de bits, …

Les fonctions (1)

- Rappel
  - Un programme C est composé de :
    - Déclarations/définitions de types
    - Déclarations/définitions de constantes globales
    - Déclarations/définitions de variables globales
    - Déclarations/définitions de fonctions
  - Un programme principal
    - La fonction `main()`
  - Le C ne possède que très peu de commandes de base
    - `break`, `case`, `continue`, `default`, `do`, `else`, `for`, `goto`, `if`, `return`, `switch` et `while`
    - Pour beaucoup d'actions (ex : affichage), on ne dispose pas de commande directe
    - Ajout de fonctionnalité via des fonctions
      - » Appartenant au compilateur (librairies)
      - » Définis par l'utilisateur

Les fonctions (1)

- Pas de distinction entre procédure et fonction en C
  - Une procédure (au sens algorithmique) est une fonction qui ne renvoi rien (terme `void` en C)
- Le codage par fonction
  - Permet une simplicité du code
  - Permet une taille de programme minimale
- Un programme C peut être considéré comme une suite de fonctions
  - Disposées dans un ordre quelconque
  - Mais attention, le compilateur doit toujours pouvoir vérifier la validité du code
    - Lecture séquentielle des fichiers par le compilateur
    - Notion de déclaration de fonction

Les fonctions (1)

- Syntaxe d'une fonction

```
TypeRetour NomFonction(Type NomParam1,Type NomParam2 […])
{
   // Corps de la fonction
   return(Resultat);
}
```

- Exemple

```
double Distance(double x1,double y1,double x2,double y2)
{
   double diff_X=x2-x1;
   double diff_Y=y2-y1;
   return(sqrt(diff_X*diff_X+diff_Y*diff_Y));
}
```

Les fonctions (1)

- On sépare une fonction en 2 parties
  - Sa déclaration (le prototype)
    - Permet au compilateur de connaitre le nom, les paramètres et la valeur de retour de la fonction
    - Permet de vérifier la syntaxe d'appel de la fonction
    - Ex :

```
double Distance(double x1,double y1,double x2,double y2);
```

  - Son corps
    - Contient le code à exécuter lors de l'appel d'une fonction
      - » Séquence d'instructions
    - Doit être connu au moment de l'appel à l'éditeur de lien (*linker*)
    - Ex :

```
double Distance(double x1,double y1,double x2,double y2)
{
   double diff_X=x2-x1;
   double diff_Y=y2-y1;
   return(sqrt(diff_X*diff_X+diff_Y*diff_Y));
}
```

Les fonctions (1)

- Appel d'une fonction
  - Utilisation du nom de la fonction
  - Ex
    ```
    double resultat=Distance(1.0,1.2,5.3,-2.12);
    ```
  - Lors de l'appel d'une fonction, le compilateur doit pouvoir vérifier que la syntaxe est correcte
    - Vérification du nom de la fonction
    - Vérification des paramètres
      - » Nombre
      - » Types
  - Nécessité de connaître le prototype de la fonction
    - Soit en déclarant la fonction avant son utilisation
    - Soir en définissant la fonction avant son utilisation

- Le compilateur lit les fichiers sources (*.c) indépendamment et séquentiellement
  - A éviter : placer le corps des fonctions dans leur ordre d'utilisation
  - Bon usage : placer les déclarations de fonctions en haut d'un fichier source
  - Meilleur usage
    - Placer les déclarations de fonction dans un fichier d'entête (header *.h)
    - Inclure les entêtes dans les fichiers sources (*.c) qui utilisent les fonctions
      ```
      #include "MonHeader.h"
      ```

Les fonctions (1)

- Une fonction particulière : `main`
  - Contient le programme principal
  - Signature (prototype)

```
int main(void);
int main(int argc, char *argv[]);
int main(int argc, char *argv[], char *envp[])
    // argc : nombre de paramètres de la ligne de commande
    /* argv : tableau de chaine de caractères contenant les
       arguments de la ligne de commande */
    /* envp : tableau de chaine de caractères contenant les
       variables d'environnement système */
```

  - Point d'entrée d'un exécutable
    - Appelée automatiquement par le système d'exploitation au chargement du programme
  - Valeur de retour
    - Code de retour d'état d'exécution du programme
    - Usage standard
      - » Valeur >=0 exécution correcte
      - » Valeur <0 erreur du programme

Les fonctions (1)

- Passage de paramètres
  - Si pas de paramètres
    - Argument void `void MaFonction(void);`
    - Pas d'argument `void MaFonction();`

  - Passage de paramètre par valeur
    - Les modifications sur la variable restent locales à la fonction
    - Ex :

```
void echanger(int x, int y)
{
    int z=x;
    x=y;
    y=z;
}

int a=1, b=2;
echanger(a,b);
// a==1 et b==2;
```

Les fonctions (1)

- Passage de paramètres
  - Passage de paramètre par adresse
    - Les modifications sur la variable se propagent en dehors de la fonction
    - Ex :

```
void echanger(int *x, int *y)
{
    int z=*x;
    *x=*y;
    *y=z;
}

int a=1, b=2;
echanger(&a,&b);
// a==2 et b==1;
```

  - Utilisation de pointeur
    - Déclaration *
    - Manipulation
      - » Opérateur * : récupération de la valeur pointée par le pointeur
      - » Opérateur & : récupération de l'adresse (pointeur) d'une variable

Bases de la programmation en C : Plan

- Introduction / historique
- Éléments fondamentaux
  - Pourquoi un langage ? Compilation et étapes de compilation
  - Structure d'un programme en C
- Variables, types de données
  - Élémentaires, constantes
  - Tableaux et chaînes de caractères
- Les fonctions 1
  - Déclaration et définition, appel de fonction
  - Passage de paramètres, paramètres de ligne de commande
- Entrées/Sorties 1
  - printf, scanf
- Expressions et opérateurs
  - Arithmétiques, comparaison, affectation, logiques, opérateurs de bits, …
  - Dualité opérateur-expression, tableau des priorités et parenthésage
- Structures de contrôle de flux
  - Instructions alternatives, répétitives, de branchement
- Structures
  - Structure classique, union, champ de bits, …

Entrées/Sorties (1)

- Permet d'envoyer/récupérer une série de caractères sur l'entrée/sortie standard
  - Bibliothèque stdio.h
    ```
    #include <stdio.h>
    ```

- Première fonction d'écriture sur la sortie standard : `printf`
  - Permet l'écriture formatée sur le flux standard de sortie *stdout* (l'écran par défaut)
  - Prototype `int printf(const char *fmt,...);`
  - Paramètres
    - Valeur de retour : nombre de caractères écrits
    - fmt : chaîne décrivant le format d'affichage pour *n* éléments
    - Ensuite viennent les *n* éléments
      - » Ex :
        ```
        int val=12;
        printf("la valeur de la variable %s est %d", "val", val);
        // affiche « la valeur de la variable val est 12 »
        ```

  - Quelques formats
    - %d : entier signé
    - %c : caractère
    - %s : chaîne de caractères
    - %o, %u, %x : entier non signé affiché en octal, décimal, ou hexadécimal
    - %f, %g : nombre flottant
    - %p : pointeur en hexadécimal
    - %% : affiche un %

Entrées/Sorties (1)

- Première fonction de lecture sur la sortie standard : `scanf`
  - Permet la lecture formatée du flux standard d'entrée ***stdin*** (le clavier par défaut)
  - Prototype `int scanf(const char *fmt,...);`
  - Paramètres
    - Valeur de retour : nombre de valeurs convenablement introduites
    - fmt : chaîne décrivant le format de lecture pour *n* éléments
    - Ensuite viennent les *n* éléments
      - » Ex :

```
int x, y;
float q;
scanf("Le quotient %d/%d vaut %f", &x, &y, &q);
printf("%g %d %d\n",q,x,y);
```

- Autres fonctions de lecture sur la sortie standard
  - `char* gets(char *str);`
    - Lit une ligne sur la sortie standard (jusqu'à trouver un '\n')
  - `char getchar();`
    - Lit un caractère sur la sortie standard

Bases de la programmation en C : Plan

- Introduction / historique
- Éléments fondamentaux
  - Pourquoi un langage ? Compilation et étapes de compilation
  - Structure d'un programme en C
- Variables, types de données
  - Élémentaires, constantes
  - Tableaux et chaînes de caractères
- Les fonctions 1
  - Déclaration et définition, appel de fonction
  - Passage de paramètres, paramètres de ligne de commande
- Entrées/Sorties 1
  - printf, scanf
- Expressions et opérateurs
  - Arithmétiques, comparaison, affectation, logiques, opérateurs de bits, …
  - Dualité opérateur-expression, tableau des priorités et parenthésage
- Structures de contrôle de flux
  - Instructions alternatives, répétitives, de branchement
- Structures
  - Structure classique, union, champ de bits, …

Expressions et opérateurs

- Un opérateur indique la nature des opérations à effectuer sur des opérandes
  - Opérateurs : `+`, `-`, `*`, `=`, …
  - Il existe un nombre défini d'opérateurs (plus de 40)
- En C, toute représentation de valeur est une expression
  - Une constante, une variable, un opérateur, une fonction sont des expressions
  - Ex

```
35
2+3
i=2
MaFonction()
```

  - Conséquence : toute expression est une valeur
  - `i=2` a une valeur : 2
    - Il est donc possible de réutiliser cette expression comme une opérande
    - `j=i=2` est équivalent à `j=(i=2)` qui est équivalent à `j=2` (pour j)
    - Le type de la valeur de l'expression dépends du type des opérandes

Expressions et opérateurs

- Types d'opérateurs
  - Unaire : admettent une seule opérande
    - Ex : `&  // &x : adresse de la variable x`
  - Binaires : admettent deux opérandes
    - Ex : `+  // a+b : somme de a et b`
  - Ternaires : admettent trois opérandes
    - Il n'en existe qu'un seul, l'opérateur conditionnel
    - Ex : `? :  // (x>2 ? 3 : 4) : si x>2, alors 3 sinon 4`

- Famille d'opérateurs
  - Arithmétiques
  - Comparaison
  - Logique
  - Bits
  - Affectation
  - Autres opérateurs (conditionnel, séquentiel, taille, adressage, …)

- Il existe des priorités sur les opérateurs
  - Utiliser des parenthèses pour fixer la priorité
    - Ex : `a+b*c` est différent de `(a+b)*c`

Expressions et opérateurs

- Opérateurs arithmétiques
  - Procède à des opérations arithmétiques sur les opérandes
    - Addition (+) : `X+Y`
    - Soustraction (-) : `X-Y`
    - Multiplication (*) : `X*Y`
    - Division (/) : `X/Y`
      - Division sur des entiers = division entière
      - Division sur des flottants = division réelle
    - Modulo (%) : `X%Y`
    - Négation (-) : `-X`

- Opérateurs de comparaison
  - Comparent les valeurs des opérandes
    - `==` : `X==Y` `// X est égal à Y ?`
    - `!=` : `X!=Y` `// X est différend de Y ?`
    - `<=` : `X<=Y` `// X est inférieur ou égal à Y ?`
    - `>=` : `X>=Y` `// X est supérieur ou égal à Y ?`
    - `<` : `X<Y` `// X est inférieur à Y ?`
    - `>` : `X>Y` `// X est supérieur à Y ?`

Expressions et opérateurs

- Opérateurs logiques
  - Procède à des opérations de la logique des prédicats sur la valeur des variables
    - `&& : X&&Y  // X AND Y`
    - `|| : X||Y  // X OR Y`
    - `!  : !X    // NOT X`
  - Une expression évaluée FAUX a pour valeur 0
  - Une expression évaluée VRAI a pour valeur 1
  - Une valeur 0 indique une évaluation FAUX
  - Une valeur différente de 0 indique une évaluation VRAI
- Opérateurs de bits
  - Procède à des opérations de la logique des prédicats sur les bits des variables
    - `& : X&Y  // AND binaire`
    - `| : X|Y  // OR binaire`
    - `^ : X^Y  // XOR binaire`
    - `~ : ~X  // NOT binaire`
    - `>> : X>>Y  // opérateur de décalage de bit à droite`
      - X décalé de Y bits vers la droite
    - `<< : X<<Y  // opérateur de décalage de bit à gauche`
      - X décalé de Y bits vers la gauche

Expressions et opérateurs

- Opérateurs d'affectation
  - Mettent dans l'opérande de gauche (*l-value*) la valeur de l'opérande de droite (*r-value*)
    - `++ : X++  // incrémente X (X=X+1)`
    - `-- : X--  // décrémente X (X=X-1)`
      - Variante : `++X et --X`

        ```
        X=Y++ // X=Y PUIS Y++
        X=++Y // Y++ PUIS X=Y
        ```
    - `= : X=Y  // affectation simple`
  - L'affectation existe sous des formes contractées avec une opération arithmétique ou logique de bit
    - `+= : X+=Y  // X=X+Y`
    - `-= : X-=Y  // X=X-Y`
    - `*= : X*=Y  // X=X*Y`
    - `/= : X/=Y  // X=X/Y`
    - `%= : X%=Y  // X=X%Y`
    - `>>= : X>>=Y  // X=X>>Y`
    - `<<= : X<<=Y  // X=X<<Y`
    - `&= : X&=Y  // X=X&Y`
    - `|= : X|=Y  // X=X|Y`
    - `^= : X^=Y  // X=X^Y`

Expressions et opérateurs

- Autres opérateurs
  - Opérateur conditionnel
    - `? :` : `X>Y ? W : T` `// si X>Y alors W sinon T`
  - Opérateur séquentiel
    - `,` : `++x,++y` `// rassemble plusieurs expressions en une seule`
      - Ex : déclaration `int x,y;`
  - Opérateur de dimension
    - `sizeof()` : `sizeof(X)` `// taille en octet de la variable X`
  - Opérateur d'adressage
    - `&` : `&X` `// adresse de la variable X`
  - Opérateur de cast
    - `(type)` : `(double)X` `// convertit X en une valeur double`
  - Opérateur de parenthésage
    - `()` : `(X+Y)*Z` `// permet de maitriser les priorités des opérations`
  - Opérateur de champ d'indirection
    - `.` et `->` : `// sélectionner une composante d'une structure`

Expressions et opérateurs

- Tableau des priorités des opérateurs

prior	*opérateurs*											*sens de l'associativité*
15	()	[]	.	->								→
14	!	~	++	--	$-_{un}$	$*_{un}$	$\&_{un}$	sizeof	(type)			←
13	$*_{bin}$	/	%									→
12	+	$-_{bin}$										→
11	<<	>>										→
10	<	<=	>	>=								→
9	==	!=										→
8	$\&_{bin}$											→
7	^											→
6	|											→
5	&&											→
4	||											→
3	? :											←
2	=	*=	/=	%=	+=	-=	<<=	>>=	&=	^=	|=	←
1	,											→

Bases de la programmation en C : Plan

- Introduction / historique
- Éléments fondamentaux
  - Pourquoi un langage ? Compilation et étapes de compilation
  - Structure d'un programme en C
- Variables, types de données
  - Élémentaires, constantes
  - Tableaux et chaînes de caractères
- Les fonctions 1
  - Déclaration et définition, appel de fonction
  - Passage de paramètres, paramètres de ligne de commande
- Entrées/Sorties 1
  - printf, scanf
- Expressions et opérateurs
  - Arithmétiques, comparaison, affectation, logiques, opérateurs de bits, …
  - Dualité opérateur-expression, tableau des priorités et parenthésage
- Structures de contrôle de flux
  - Instructions alternatives, répétitives, de branchement
- Structures
  - Structure classique, union, champ de bits, …

Structures de contrôle de flux

- Instructions alternatives
  - Test if
    - Ex :

```
if (expression) instruction;
```

  - Test avec alternative
    - Ex :

```
if (expression) instruction1;
else instruction2;
```

  - Attention à la séparation des blocs
    - Ex :

```
if (expression2)
   if (expression2) instruction1;
   else instruction2;
instruction3;
```

Structures de contrôle de flux

- Test multiples (switch)
  - Ex :

```
switch (expression)
{
case cst1: instruction1;
   break;

case cst2: instruction2;
   break;

default:
}
```

- cst1, cst2, … doivent être des expressions constantes

Structures de contrôle de flux

- Instructions répétitives
  - Instruction tant que (while)
    - Ex :

```
while (expression) instruction;
```

    - Exécute instruction tant que expression est vrai

  - Alternative : instruction faire jusqu'à (do while)
    - Ex :

```
do instruction; while (expression);
```

    - Exécute l'instruction au moins une fois puis tant que l'expression est vrai
    - Attention à ne pas oublier le ;

Structures de contrôle de flux

- Instructions répétitives
  - Instruction pour (for)
    - Ex :

```
for (expression1; expression2; expression3) instruction;
```

    - expression1 est exécuté au premier tour de boucle
    - expression2 conditionne le passage dans la boucle
    - expression3 est exécuté après chaque passage dans la boucle
    - Ex :

```
for (i=0; i<10; i++)
{
   // ...
}
```

    - Possibilité de chaîner plusieurs instructions par expression via l'opérateur ,
    - Ex :

```
for (i=0,j=0; i<10 && j>1; i++,j--) ...
```

  - Instructions de contrôle de boucle
    - `continue` : branche à la fin du bloc
    - `break` : sort de la boucle

Bases de la programmation en C : Plan

- Introduction / historique
- Éléments fondamentaux
  - Pourquoi un langage ? Compilation et étapes de compilation
  - Structure d'un programme en C
- Variables, types de données
  - Élémentaires, constantes
  - Tableaux et chaînes de caractères
- Les fonctions 1
  - Déclaration et définition, appel de fonction
  - Passage de paramètres, paramètres de ligne de commande
- Entrées/Sorties 1
  - printf, scanf
- Expressions et opérateurs
  - Arithmétiques, comparaison, affectation, logiques, opérateurs de bits, …
  - Dualité opérateur-expression, tableau des priorités et parenthésage
- Structures de contrôle de flux
  - Instructions alternatives, répétitives, de branchement
- Structures
  - Structure classique, union, champ de bits, …

Structures

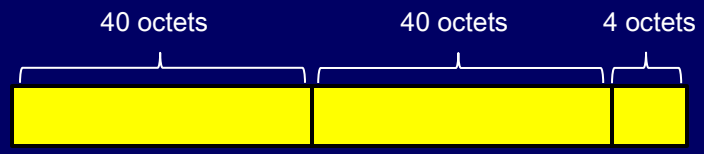
- Structure
  - Un tableau ne peut contenir que des éléments du même type
  - En C pas de possibilité de stocker n'importe quel type (non pointeur)
  - Solution intermédiaire : structure
    - Permet de regrouper des éléments de différents types
    - Stockage des éléments de manière continue en mémoire
      - Séquence de variables disposées de manière continue en mémoire

- Syntaxe

```
struct Identifiant
{
    Déclarations
};
```

- Ex

```
struct personne
{
    char nom[40],prenom[40];
    int age;
};
```



Structures

- Déclaration de variables de type structure
  - À la déclaration de la structure

```
struct Identifiant
{
    Déclarations
} Variable;

struct
{
    Déclarations
} Variable;
```

  - Après la déclaration

```
struct Identifiant Variable;
```

  - Ex

```
struct personne UnePersonne;
struct personne UnePersonne={"Nom", "Prenom", 30};
```

Structures

- Accès aux champs
  - Opérateur `.` pour une variable de type `struct`
  - Opérateur `->` pour un pointeur de type `struct`

  - Ex

```
struct personne Jean, *JeanPtr;

Jean.age=30;
JeanPtr->age=30;        // == (*JeanPtr).age=30;
```

  - Attention à la priorité des opérateurs !

    `*JeanPtr.age` équivaut à `*(JeanPtr.age)` et non à `(*JeanPtr).age`

Structures

- Les unions : Union
  - Permet de stocker un type différent dans un même espace mémoire
  - Syntaxe

```
union Identifiant
{
    Déclarations
};
```

  - Ex

```
union int_double
{
    unsigned int i;
    double d;
};

// i et d partagent le même espace mémoire
// seul un des deux éléments est utilisable au même instant
```

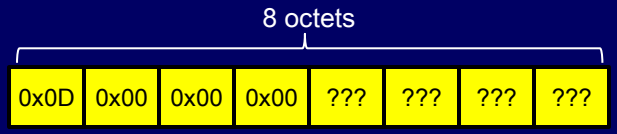
Structures

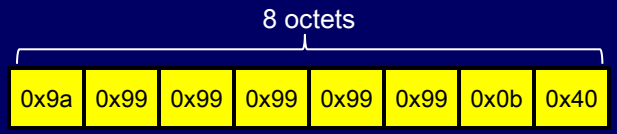
- Union : utilisation
  - Ex

```
union int_double x;
// 8 octets alloués (peut stocker un entier OU un double)

x.i=13;
// Attention à ne pas accéder à x.d


x.d=3.45;
// Attention à ne pas accéder à x.i
```

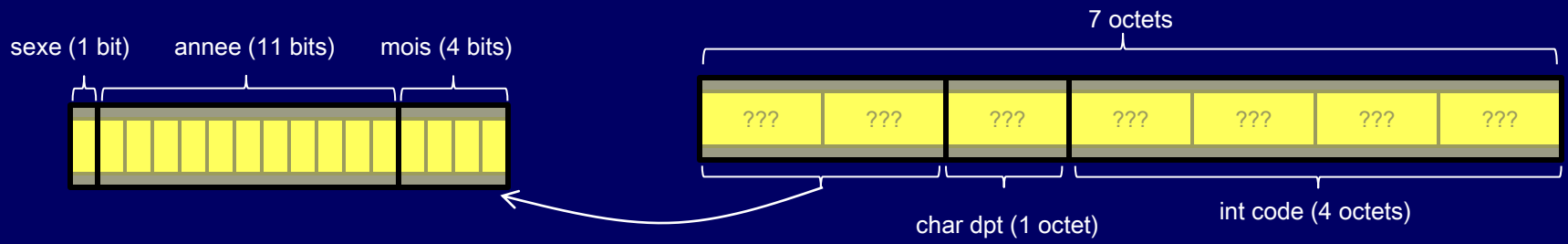




Structures

- Les champ de bits
  - Permet de stocker des éléments avec un nombre de bit spécifié
  - But : économiser de l'espace mémoire
  - Construit à partir d'une structure et des types `int` et `unsigned int`
    - Ex :

```
struct secu_id
{
    int sexe : 1;       // Stocké dans un seul bit
    int annee : 11;     // Stocké dans 11 bits
    int mois : 4;       // Stocké dans 4 bits
    char dpt;
    int code;
};
```



  - La taille mémoire utilisée est obligatoirement multiple de l'emplacement d'un `int`
    - (ou plus en fonction de l'alignement)