

Programmation Impérative

HLIN202

P. Giorgi M. Montassier

Université de Montpellier

Volume horaire

Volume horaire :

- 7 Cours
- 11 TDs
- 10 TP

Le TD et le TP se font sur 3h en salle informatisée (sauf dernière séance et cas exceptionnel).

Les TDs/TPs commencent la semaine prochaine.

Évaluation du cours

- 1 note de contrôle continu (assiduité, participation, contrôle)
- 1 examen terminal

La note finale sera :

$$\max(Exam, 0.6 \times Exam + 0.4 \times CC)$$

Objectifs

- L'objectif n'est pas de faire de vous des spécialistes d'un langage impératif en particulier
- Objectifs :
 - Apprendre les fondements de la programmation impérative
 - Apprendre à les mettre en oeuvre en C/C++
- Vous devez être capables de vous adapter à un autre langage impératif : la syntaxe change, mais les principes restent les mêmes.

Plan

- 1 Introduction
- 2 Bases du langage C++
- 3 Structures de contrôle
- 4 Fonctions
- 5 Pointeurs
- 6 Tableaux
- 7 Structures de données (Facultatif)

Plan

1 Introduction

- Langage de programmation
- Programmation impérative
- Les langages C/C++
- Les erreurs

Langage de programmation

Definition

Un langage de programmation est un système de notation permettant de décrire de manière simple une suite d'actions ordonnée qu'un ordinateur pourra exécuter.

Différents niveaux de langage :

- **langage machine** :
codage binaire propre à chaque processeur
- **langage assembleur** :
codage alphanumérique du langage machine
↪ traduit en langage machine par un assembleur
- **langage haut niveau** :
codage alphanumérique proche du langage naturel
↪ traduit en langage assembleur par un compilateur

Langage assembleur

- Langage procédurale proche de l'architecture de la machine.
- Syntaxe différente d'un processeur à l'autre.

```

        .section      __TEXT,__text,regular,pure_instructions
        .globl  _main
        .align  4,0x90

_main:                                     ## @main
        .cfi_startproc
## BB#0:
        push     rbp
Ltmp2:
        .cfi_def_cfa_offset 16
Ltmp3:
        .cfi_offset rbp, -16
        mov      rbp, rsp
Ltmp4:
        .cfi_def_cfa_register rbp
        mov      eax, 0
        mov      dword ptr [rbp - 4], 0
        mov      dword ptr [rbp - 8], 10
        pop      rbp
        ret
        .cfi_endproc

.subsections_via_symbols
  
```


Langage assembleur

- Il est propre à chaque famille de processeurs.
- Il est loin de la notion d'algorithme.
- Le code d'un même programme change d'un processeur à l'autre.

↪ langage peu adapté à l'écriture simple de programmes.

Remarque

On dit que du code en langage assembleur n'est pas **portable** car il n'est pas garanti de fonctionner sur n'importe quel ordinateur.

Langage haut niveau

- Syntaxe indépendante des processeurs.
- Système d'écriture plus proche de la langue naturelle
- Il s'appuie sur des concepts facilitant la programmation.
exemple : notion d'algorithmme, notion d'objet, etc.

Le même code que précédemment mais écrit en C/C++.

```
1 int main()  
2 {  
3     int a;  
4     a=10;  
5     return 0;  
6 }
```

Langage haut niveau

Ce langage n'est pas exécutable directement par un ordinateur.

Deux solutions possibles :

- le traduire complètement en langage machine :
 ↪ notion de **langage compilé**
- l'exécuter par un programme qui le traduit à la volée :
 ↪ notion de **langage interprété**

Remarque

Certains langages interprétés utilisent une compilation vers un langage intermédiaire pour améliorer les performances, ex. JAVA.

Langage haut niveau

Il existe plusieurs types de langages haut niveau, chacun proposant des notions (*paradigmes*) de programmation différentes.

- langages fonctionnels : CAML
- langages impératifs : C, Pascal, Fortran
- langages à objets : JAVA
- langages logiques : Prolog
- langage multi-paradigme : C++, python, Scala

Remarque

Un paradigme peut être mieux adapté qu'un autre pour certaines tâches. Deux programmes peuvent produire les mêmes résultats tout en étant écrit avec des langages différents.

Plan

1 Introduction

- Langage de programmation
- **Programmation impérative**
- Les langages C/C++
- Les erreurs

Définition

La programmation impérative procédurale correspond à la notion de changement d'état d'un programme à partir :

- d'instructions simples,
- d'appel d'algorithmes (i.e. procédure/fonction).

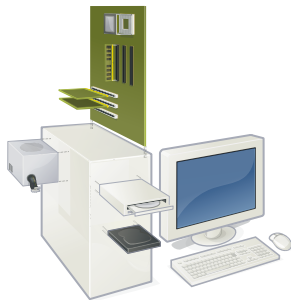
Changement d'état d'un programme

Cela correspond à la modification d'une donnée utilisée par le programme (**une variable**).

Remarque

Ce modèle de programmation est au plus proche des architectures de nos ordinateurs actuels.

Modélisation d'un ordinateur actuel

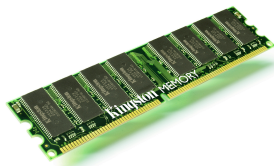


- de la mémoire
- des unités de calcul sur des nombres (entiers/réels)
- des unités de contrôle

Un langage impératif s'appuie essentiellement sur 3 notions :

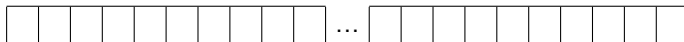
mémoire, calcul, contrôle

Mémoire d'un ordinateur



- Mémoire divisée en case
↪ une case est un bit (soit 0, soit 1)
- Case identifiée par un numéro (adresse)

On peut donc voir la mémoire d'un ordinateur comme un tableau contenant des 0 et des 1.



Contenu de la mémoire

Toutes les données dans un programme sont traduits en 0 et en 1

- des plus simples (booléens, entiers)
- aux plus complexes (une carte routière, un film)

On parle de **codage binaire des données**.

Exemple

La représentation en mémoire de 63576473 est

00000011110010100001100110011001

celle de Π peut-être

1100100100001111101101010100010001000010110100011000

Contenu de la mémoire

Les données étant de diverse nature (**taille et contenu du codage**), on utilisera la notion de **type** pour les différencier.

On aura :

- des types de base fournis par le langage, e.g. float, int.
- un mécanisme d'extension des types, e.g. codage d'une image.

Écrire un programmation impératif

Le programmeur décrit de manière ordonnée chacune des instructions composant son programme.

Instructions possibles :

- de calcul : $3+5$
- de contrôle : **si ... alors ... sinon, pour, tant que**
- de mémoire :
 - nommer une zone mémoire : **variable**
 - accéder directement à une case mémoire : **pointeur**
 - modifier une zone mémoire : **affectation**

Plan

1 Introduction

- Langage de programmation
- Programmation impérative
- Les langages C/C++
- Les erreurs

Noyau de langage impératif procédural

- faiblement typé (**conversion implicite des types**)
- langage compilé
- arithmétique sur les adresses mémoires et sur les bits

Remarque

- permet de rester très proche de l'architecture des ordinateurs,
- utile pour avoir les meilleures performances possibles

Le noyau impératif de C++ pouvant être considéré comme une équivalence du langage C, nous ne parlerons que C++ dans la suite de ce cours.

Attention

- les concepts généraux présentés en C++ restent valables en C,
- la mise en oeuvre et la syntaxe restent parfois différentes.

Concevoir un programme en C++

Trois étapes ordonnées :

- 1 écrire dans un fichier texte le code du programme en C++
- 2 compiler le programme
- 3 exécuter le programme

Écrire un programme C++

Il suffit d'écrire avec un éditeur de texte simple (sans formatage) quelques lignes de code.

exemple.cpp

```
1 #include <iostream>
2
3 int main(){
4     int a,b,c;
5     std::cin>>a>>b;
6     c=a+b;
7     std::cout<<a<<"+"<<b<<"="<<c<<std::endl;
8
9     return 0;
10 }
```

Compilation d'un programme C++

Objectif

Traduire un programme décrit en C++ dans un programme décrit en langage machine.

Attention

Les programmes écrits en C++ ne sont pas exécutables, seules leurs traductions en langage machine le sont.

L'outil permettant la traduction d'un texte écrit en C++ en langage machine s'appelle un compilateur.

Compilation d'un programme C++

Il existe plusieurs compilateurs C++ en fonction du système d'exploitation et de l'architecture.

Dans un environnement GNU/Linux,

- la suite de compilation standard est appelé GCC (the GNU Compiler Collection).
- le compilateur C++ de GCC est : **g++**

Un exemple de compilation

Pour compiler l'exemple ([exemple.cpp](#)), on tape dans un terminal :

```
g++ -Wall exemple.cpp -o Exemple
```

ce qui permet de créer un programme exécutable nommé Exemple.

Syntaxe générale

```
g++ -Wall source.cpp -o execu
```

- `source.cpp` est le fichier du programme en C++.
- `execu` est le programme traduit exécutable sur la machine.
- `-o ...` indique que `...` est le nom du programme.

Un exemple de compilation

Pour compiler l'exemple ([exemple.cpp](#)), on tape dans un terminal :

```
g++ -Wall exemple.cpp -o Exemple
```

ce qui permet de créer un programme exécutable nommé Exemple.

Syntaxe générale

```
g++ -Wall source.cpp -o execu
```

- `source.cpp` est le fichier du programme en C++.
- `execu` est le programme traduit exécutable sur la machine.
- `-o ...` indique que `...` est le nom du programme.

Notion de programme en C++

Un programme est constitué de plusieurs fonctions/procédures qui peuvent s'appeler les unes les autres.

La première fonction appelée par un programme est :

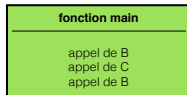
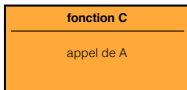
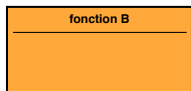
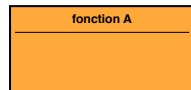
le point d'entrée du programme

Point d'entrée d'un programme C++

Il correspond à une fonction appelée **main** qui est spécifique :

```
1 int main(){  
2     // début du programme  
3     ...  
4     // fin du programme  
5     return 0;  
6 }
```

Principe d'un programme C++



Écriture du programme :

- 1 On définit un ensemble de fonctions **A,B,C**.
- 2 On définit un programme avec **main**

Quand on lance le programme :

- il exécute **main**
 - qui exécute **B**
 - qui exécute **C**
 - qui lui même exécute **A**
 - qui exécute encore **B**

Attention

la fonction main n'est définie qu'une seule fois par programme.

Plan

1 Introduction

- Langage de programmation
- Programmation impérative
- Les langages C/C++
- Les erreurs

Les erreurs

Trois types d'erreurs

- Erreur de syntaxe
- Erreur de conception
- Erreur d'exécution

Erreurs de syntaxe

Les langages de programmation sont définis à partir de règles strictes. La violation d'une seule règle implique **une erreur de syntaxe**.

Remarque

Les langages de programmation ne peuvent être ambigus. Chaque mot clé du langage a un sens et il faut les utiliser correctement.

Correction des erreurs de syntaxe

Le compilateur affiche les éventuelles erreurs de syntaxe en décrivant le type d'erreur et la ligne du programme incriminée.

Erreurs de syntaxe

syntax-error.cpp

```
1 int main()  
2 {  
3     int a  
4     return 0;  
5 }
```

Compilation

```
g++ -Wall syntax-error.cpp -o syntax-error
```

```
syntax-error.cpp: In function 'int main()':  
syntax-error.cpp:3:3: error: expected initializer before 'return'
```

il y a une erreur **ligne 3** avant le mot clé **return** ⇔ il manque un **;**

Erreurs de conception

Vous vous êtes trompé dans la conception du programme.

Exemple

- 1 faire chauffer une poêle avec de l'huile.
- 2 battre les oeufs.
- 3 verser les oeufs battus dans la poêle.
- 4 casser les oeufs.

bien que syntaxiquement correct ce programme ne fera pas une bonne omelette.

Erreurs de conception

Attention

Un ordinateur exécute strictement les ordres qu'on lui donne!!!

Attention

Le compilateur ne peut pas détecter les erreurs de conception!!!

Erreurs d'exécution

Souvent liées à des erreurs d'écriture (bug) ou à des erreurs de conception.

- non détectées à la compilation
- dépendantes de l'état du programme et de son environnement d'exécution

Remarque

Un programme peut fonctionner correctement très longtemps et produire soudainement une erreur.

un cas classique : **une division par 0**

Erreurs d'exécution

execution-error.cpp

```
1 #include <iostream>
2 int main()
3 {
4     int a,b,c;
5     std::cin>>a>>b;
6     c=a/b;
7
8     return 0;
9 }
```

La compilation ne détecte aucune erreur, mais si on exécute le programme en saisissant $b = 0$, on obtient le message suivant :

Floating point exception

Plan

- 1 Introduction
- 2 Bases du langage C++**
- 3 Structures de contrôle
- 4 Fonctions
- 5 Pointeurs
- 6 Tableaux
- 7 Structures de données (Facultatif)

Rappel : programme minimal

Un programme C++ se compose d'une fonction **main**, dont la version la plus simple est :

```
1 int main(){  
2     déclaration de variables  
3     instructions du programme  
4 return 0;  
5 }
```

Rappel : programme minimal

Comme dans le langage algorithmique :

- les variables ont un type (ex : entier, flottant)
- les instructions sont :
 - des instructions simples (ex : affectation)
 - des expressions (ex : $a + b \times c$)
 - des structures de contrôle (ex : tant que)

Les composants de base d'un programme C++

Un programme C++ est constitué des composants élémentaires suivants :

- les types de données,
- les constantes,
- les variables,
- les opérateurs,
- les structures de contrôle,
- les appels de fonctions

Plan

2 Bases du langage C++

■ Les type de données

■ Les constantes

■ Les variables

■ Opérateurs

■ Les conversions de type

■ Les entrées-sorties

■ Instructions

Les types de données

- **des nombres** : entiers, approximations flottantes des réels
- **des lettres** : caractères
- **des booléens** : vrai, faux

type	type en C++	valeurs
entiers	int	$[-2^{31}, 2^{31} - 1]$
réels (simple précision)	float	≈ 7 chiffres significatifs
réels (double précision)	double	≈ 15 chiffres significatifs
caractère	char	
booléen	bool	{ true, false }

Le codage binaire des entiers

Les entiers sont exprimés sous format binaire

$$103 = 64 + 32 + 4 + 2 + 1 = 2^6 + 2^5 + 2^2 + 2^1 + 2^0$$

1	1	0	0	1	1	1
---	---	---	---	---	---	---

Le nombre de bit (case) fixe l'ensemble des entiers représentables.

Sur 7 bits on représente les entiers positifs dans $[0, 127]$

- le plus grand : $127 = 2^7 - 1$

1	1	1	1	1	1	1
---	---	---	---	---	---	---

- le plus petit : 0

0	0	0	0	0	0	0
---	---	---	---	---	---	---

Le codage binaire des entiers

Les entiers négatifs sont également représentés par un codage binaire : **le codage en complément à deux**

Complément à deux

$$-103 = -128 + 25 = -128 + 16 + 8 + 1 = -2^7 + 2^4 + 2^3 + 2^0$$

1	0	0	1	1	0	0	1
---	---	---	---	---	---	---	---

 sur 8 bits

$$-103 = -256 + 128 + 16 + 8 + 1 = -2^8 + 2^7 + 2^4 + 2^3 + 2^0$$

1	1	0	0	1	1	0	0	1
---	---	---	---	---	---	---	---	---

 sur 9 bits

On dit que le bit de poids fort (le plus à gauche) est le bit de signe.

↪ **il a un poids négatif si non nul.**

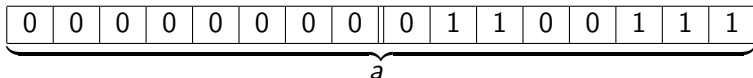
Les types de données entiers

Plusieurs variantes (**complément à deux**) qui utilisent plus ou moins d'octets et qui ont différentes plages de valeurs : *norme C++11*.

type entier	type en C++	nbr octets	valeurs
court	short int	≥ 2	$[-2^{15}, 2^{15} - 1]$
standard	int	≥ 2	$[-2^{16}, 2^{16} - 1]$
long	long int	≥ 4	$[-2^{31}, 2^{31} - 1]$
très long	long long int	≥ 8	$[-2^{63}, 2^{63} - 1]$

Exemple

```
short int a = 103;
```



Les types de données entiers

On peut étendre la valeur maximum des entiers en utilisant des entiers positifs (**versions non signées = pas de complément à deux**)

type en C++	nbr octets	valeurs
unsigned short int	≥ 2	$[0, 2^{16} - 1]$
unsigned int	≥ 2	$[0, 2^{16} - 1]$
unsigned long int	≥ 4	$[0, 2^{32} - 1]$
unsigned long long int	≥ 8	$[0, 2^{64} - 1]$

Remarque

Sur une machine Unix 64-bits, on admet communément que
taille int = 4 octets; taille long int = 8 octets;

Les types de données réels

Les réels ne sont pas représentables en codage binaire. On représente un réel x par une approximation rationnelle particulière :

$$x \approx (-1)^s \times \frac{m}{2^e}$$

exemple : $0.75 = (-1)^0 \times \frac{3}{2^2}$

Le codage de x correspond au codage binaire de s, m et e .

s		m				e	
0	0	0	1	1	0	1	0

Les types de données réels

Les réels ne sont pas représentables en codage binaire. On représente un réel x par une approximation rationnelle particulière :

$$x \approx (-1)^s \times \frac{m}{2^e}$$

exemple : $0.75 = (-1)^0 \times \frac{3}{2^2}$

Le codage de x correspond au codage binaire de s, m et e .

s		m				e		
0	0	0	1	1	0	1	0	

Norme IEEE754 :

- float (32 bits) : 1 bit pour s , 23 bits pour m , 8 bits pour e
- double (64 bits) : 1 bit pour s , 52 bits pour m , 11 bits pour e

Le type de données caractère

Le mot clé `char` désigne un caractère codé sur un octet basé sur une extension du code ASCII (les caractères accentués en plus) :

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040	<code>&#32;</code>	Space	64	40	100	<code>&#64;</code>	@	96	60	140	<code>&#96;</code>	`
1	1	001	SOH (start of heading)	33	21	041	<code>&#33;</code>	!	65	41	101	<code>&#65;</code>	A	97	61	141	<code>&#97;</code>	a
2	2	002	STX (start of text)	34	22	042	<code>&#34;</code>	"	66	42	102	<code>&#66;</code>	B	98	62	142	<code>&#98;</code>	b
3	3	003	ETX (end of text)	35	23	043	<code>&#35;</code>	#	67	43	103	<code>&#67;</code>	C	99	63	143	<code>&#99;</code>	c
4	4	004	EOT (end of transmission)	36	24	044	<code>&#36;</code>	\$	68	44	104	<code>&#68;</code>	D	100	64	144	<code>&#100;</code>	d
5	5	005	ENQ (enquiry)	37	25	045	<code>&#37;</code>	%	69	45	105	<code>&#69;</code>	E	101	65	145	<code>&#101;</code>	e
6	6	006	ACK (acknowledge)	38	26	046	<code>&#38;</code>	&	70	46	106	<code>&#70;</code>	F	102	66	146	<code>&#102;</code>	f
7	7	007	BEL (bell)	39	27	047	<code>&#39;</code>	'	71	47	107	<code>&#71;</code>	G	103	67	147	<code>&#103;</code>	g
8	8	010	BS (backspace)	40	28	050	<code>&#40;</code>	(72	48	110	<code>&#72;</code>	H	104	68	150	<code>&#104;</code>	h
9	9	011	TAB (horizontal tab)	41	29	051	<code>&#41;</code>)	73	49	111	<code>&#73;</code>	I	105	69	151	<code>&#105;</code>	i
10	A	012	LF (NL line feed, new line)	42	2A	052	<code>&#42;</code>	*	74	4A	112	<code>&#74;</code>	J	106	6A	152	<code>&#106;</code>	j
11	B	013	VT (vertical tab)	43	2B	053	<code>&#43;</code>	+	75	4B	113	<code>&#75;</code>	K	107	6B	153	<code>&#107;</code>	k
12	C	014	FF (NP form feed, new page)	44	2C	054	<code>&#44;</code>	,	76	4C	114	<code>&#76;</code>	L	108	6C	154	<code>&#108;</code>	l
13	D	015	CR (carriage return)	45	2D	055	<code>&#45;</code>	-	77	4D	115	<code>&#77;</code>	M	109	6D	155	<code>&#109;</code>	m
14	E	016	SO (shift out)	46	2E	056	<code>&#46;</code>	.	78	4E	116	<code>&#78;</code>	N	110	6E	156	<code>&#110;</code>	n
15	F	017	SI (shift in)	47	2F	057	<code>&#47;</code>	/	79	4F	117	<code>&#79;</code>	O	111	6F	157	<code>&#111;</code>	o
16	10	020	DLE (data link escape)	48	30	060	<code>&#48;</code>	0	80	50	120	<code>&#80;</code>	P	112	70	160	<code>&#112;</code>	p
17	11	021	DC1 (device control 1)	49	31	061	<code>&#49;</code>	1	81	51	121	<code>&#81;</code>	Q	113	71	161	<code>&#113;</code>	q
18	12	022	DC2 (device control 2)	50	32	062	<code>&#50;</code>	2	82	52	122	<code>&#82;</code>	R	114	72	162	<code>&#114;</code>	r
19	13	023	DC3 (device control 3)	51	33	063	<code>&#51;</code>	3	83	53	123	<code>&#83;</code>	S	115	73	163	<code>&#115;</code>	s
20	14	024	DC4 (device control 4)	52	34	064	<code>&#52;</code>	4	84	54	124	<code>&#84;</code>	T	116	74	164	<code>&#116;</code>	t
21	15	025	NAK (negative acknowledge)	53	35	065	<code>&#53;</code>	5	85	55	125	<code>&#85;</code>	U	117	75	165	<code>&#117;</code>	u
22	16	026	SYN (synchronous idle)	54	36	066	<code>&#54;</code>	6	86	56	126	<code>&#86;</code>	V	118	76	166	<code>&#118;</code>	v
23	17	027	ETB (end of trans. block)	55	37	067	<code>&#55;</code>	7	87	57	127	<code>&#87;</code>	W	119	77	167	<code>&#119;</code>	w
24	18	030	CAN (cancel)	56	38	070	<code>&#56;</code>	8	88	58	130	<code>&#88;</code>	X	120	78	170	<code>&#120;</code>	x
25	19	031	EM (end of medium)	57	39	071	<code>&#57;</code>	9	89	59	131	<code>&#89;</code>	Y	121	79	171	<code>&#121;</code>	y
26	1A	032	SUB (substitute)	58	3A	072	<code>&#58;</code>	:	90	5A	132	<code>&#90;</code>	Z	122	7A	172	<code>&#122;</code>	z
27	1B	033	ESC (escape)	59	3B	073	<code>&#59;</code>	:	91	5B	133	<code>&#91;</code>	[123	7B	173	<code>&#123;</code>	{
28	1C	034	FS (file separator)	60	3C	074	<code>&#60;</code>	<	92	5C	134	<code>&#92;</code>	\	124	7C	174	<code>&#124;</code>	
29	1D	035	GS (group separator)	61	3D	075	<code>&#61;</code>	=	93	5D	135	<code>&#93;</code>	}	125	7D	175	<code>&#125;</code>	}
30	1E	036	RS (record separator)	62	3E	076	<code>&#62;</code>	>	94	5E	136	<code>&#94;</code>	^	126	7E	176	<code>&#126;</code>	~
31	1F	037	US (unit separator)	63	3F	077	<code>&#63;</code>	?	95	5F	137	<code>&#95;</code>	_	127	7F	177	<code>&#127;</code>	DEL

Plan

2 Bases du langage C++

- Les type de données
- **Les constantes**
- Les variables
- Opérateurs
- Les conversions de type
- Les entrées-sorties
- Instructions

Les constantes

Chaque type de données possède des constantes pouvant être utilisées dans un programme pour :

- affecter une valeur à une variable,
- effectuer un calcul,
- former une expression booléenne.

Exemple

```
int a=10;  
1+2+3+4+5;  
1<2;
```

Les constantes entières

Elles sont définies en accord avec le type natif des entiers (**int**) et appartiennent donc à l'intervalle $[-2^{31}, 2^{31} - 1]$ (en général).

```
1 int a;  
2 a= 2147483647;    (affectation valide    -> a=231 -1)  
3 a= 2147483648;    (constante erronée -> a= -231)  
4 long int b;  
5 b= 2147483648L;    (affectation valide -> b= 231)
```

Remarque

Le compilateur provoquera une alerte si la constante est supérieure à la capacité du type affecté. **short int a=2147483647;**

Les constantes entières

On peut étendre leur plage en les suffixant par

- U (unsigned)
- L (long)
- LL (long long)

ex : $2^{64} - 1 \Rightarrow 18446744073709551615ULL$

Plusieurs écriture possibles :

- décimal (en base 10) - ex : 1234
- octal (en base 8) - ex : 0477
- hexadécimal (en base 16) - ex : 0xA123F

Les constantes réelles (flottantes)

Elles sont définies en accord avec le type double. On peut forcer le codage en suffixant par F (float) et L (double).

Codage scientifique

- représentation par mantisse et exposant
- l'exposant est introduit par la lettre e
- ex : 2.34e4 représente 2.34×10^4

Les constantes caractères

Elles sont formées par l'expression utilisant les guillemets simples :
'...' ou les ... sont remplacés par :

- un caractère alphanumérique :

```
abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ  
1234567890;,.?!@*()+-[]:;<> ETC.
```

- un caractère spécial :

`\n` → retour à la ligne

`\t` → tabulation

`\b` → backspace

...

Plan

2 Bases du langage C++

- Les type de données
- Les constantes
- **Les variables**
- Opérateurs
- Les conversions de type
- Les entrées-sorties
- Instructions

Manipulation de la mémoire

Rappel :

- la mémoire est binaire (un grand tableau de 0 et de 1)
- un langage impératif manipule directement la mémoire

Comment savoir où se trouvent les données en mémoire ?

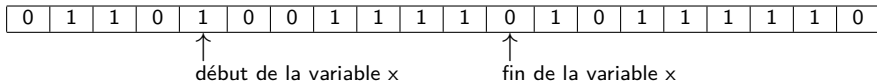
0	1	1	0	1	0	0	1	1	1	1	0	1	0	1	1	1	1	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Manipulation de la mémoire

Rappel :

- la mémoire est binaire (un grand tableau de 0 et de 1)
- un langage impératif manipule directement la mémoire

Comment savoir où se trouvent les données en mémoire ? **les variables**

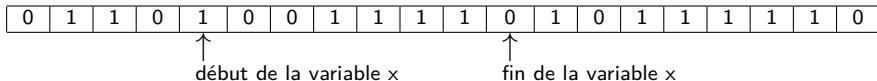


Manipulation de la mémoire

Rappel :

- la mémoire est binaire (un grand tableau de 0 et de 1)
- un langage impératif manipule directement la mémoire

Comment savoir où se trouvent les données en mémoire ? **les variables**



On identifie une variable par :

- une adresse de début dans l'espace mémoire
- une taille indiquant l'espace mémoire occupé par la variable

Les variables en C++

Definition

On appelle variable une zone mémoire de l'ordinateur à laquelle on a donné un nom, ainsi qu'un type de données.

- le nom de la variable, appelé *identificateur*¹, permet de manipuler facilement les données stockées en mémoire.
- le type de données permet au compilateur de réserver l'espace mémoire suffisant pour stocker les valeurs.

(1) on utilise un identificateur plutôt que l'adresse mémoire mais l'on peut facilement récupérer l'adresse d'une variable à partir de son identificateur.

Les variables en C++

Exemple

```
int a;
```

a est une variable de type entier :

- le compilateur lui réservera 4 octets en mémoire pour stocker ses valeurs
- on utilisera le nom « a » pour travailler avec l'espace mémoire attribué à la variable a.

Les variables en C++

Exemple

```
double f;
```

f est une variable de type réel :

- le compilateur lui réservera 8 octets en mémoire pour stocker ses valeurs
- on utilisera le nom « f » pour travailler avec l'espace mémoire attribué à la variable f.

Plan

2 Bases du langage C++

- Les type de données
- Les constantes
- Les variables
- **Opérateurs**
- Les conversions de type
- Les entrées-sorties
- Instructions

Les opérateurs arithmétiques

les opérateurs classiques

- addition : $+$
- soustraction : $-$
- division : $/$
- multiplication : $*$
- opposé : $-$
- modulo : $\%$

Attention : les opérateurs utilisent des opérandes du même type et renvoie un résultat du même type que les opérandes

- `int + int = int;`
- `float * float = float;`
- `int + float = ???`

Priorité des opérateurs arithmétiques

Les règles de priorités mathématiques sont conservées :

- $a + b \times c$ sera interprété $a + (b \times c)$
- $(a + b) \times c$ sera interprété correctement car les parenthèses fixent l'ordre de priorité

nb : les opérations entre parenthèses sont prioritaires et seront toujours évaluées en premier

Opérateurs de comparaisons

les opérateurs classiques sur les types numériques :

égalité	==
différent	!=
supérieur	>
inférieur	<
supérieur ou égal	>=
inférieur ou égal 1	<=

Attention : le langage C++ dispose d'un type booléen en standard, le résultat d'une expression booléenne est un booléen :

- false si l'expression est fausse (ex : $3 < 2$)
- true si l'expression est vraie (ex : $3 > 2$)

Opérateurs logiques

les opérateurs logiques en C :

- et : `&&`
- ou : `||`
- non : `!`

les opérandes sont de type booléen.

Exemple

`(2.5 > 3.5) && (1 < 3)` est égale à `false`
`true || (1 >= 3)` est égale à `true`

Opérateur d'affectation

Cette opérateur permet d'affecter la valeur d'une expression à une variable. **L'affectation se fait avec =**

- `var = exp`
 - `exp` : expression du même type que `var`
 - `var` : nom d'une variable déclarée
 - la variable `var` prend la valeur de l'expression `exp`
 - ex : `a=2+3`; `a` prend la valeur de l'expression `2+3` donc 5
- comme en algorithmique, l'opérande de gauche ne peut être qu'une variable : l'affectation **`a+b=3`**; n'est pas correcte syntaxiquement.

Autres opérateurs

- incrémentation/décrémentation d'une variable entière a :
 - $a++$ incrémente la valeur de a par 1
 - $a--$ décrémente la valeur de a par 1
- affectations élargies : $+=$, $-=$, $*=$, $/=$
 - $a+=3$; correspond à l'expression $a=a+3$;
- taille mémoire des variables : `sizeof`
 - `sizeof(a)` renvoie la taille en octets de la variable a .
- et beaucoup d'autres ...

Opérateur d'adresse mémoire

Chaque variable est stockée en mémoire à une adresse précise.
L'opérateur d'adresse `&` permet de récupérer l'adresse associée à une variable

- si `a` est une variable définie :
 `&a` renvoie la valeur de l'adresse de `a`

Attention

- l'adresse des variables n'est pas choisie par le programmeur :
 `&a = ...` est interdit !!!
- l'adresse des variables peut être stockée dans une variable :
 `b = &a` si `b` a le bon type

Priorités des opérateurs : récapitulatif

Opération associative (\star) :

■ à droite : $a \star b \star c \Rightarrow a \star (b \star c)$

■ à gauche : $a \star b \star c \Rightarrow (a \star b) \star c$

Catégorie	Opérateurs	Associativité
Unaire	$+$, $-$, $++$, $--$, $!$, $*$, $\&$, <code>sizeof</code> , <code>cast</code>	Droite
Binaire	$*$, $/$, $\%$	Gauche
Binaire	$+$, $-$	Gauche
Comparaison	$<$, $<=$, $>$, $>=$	Gauche
Comparaison	$==$, $!=$	Gauche
Logique	$\&\&$	Gauche
Logique	$ $	Gauche
Affectation	$=$, $+=$, $-$, $*=$, $/$, $\% =$	Droite

Le tableau est classé par ordre de priorité décroissante.

Priorités des opérateurs : exemples

Expression	Expression parenthésée	Valeur
$2+3*7$	$2+(3*7)$	23
$15*3\%7/2$	$((15*3)\%7)/2$	1
$x=y=2$	$x=(y=2)$	x :2, y :2
$1<4==7<5!=9<4$	$((1<4)==(7<5))!=(9<4)$	false (ou 0)

Plan

2 Bases du langage C++

- Les type de données
- Les constantes
- Les variables
- Opérateurs
- **Les conversions de type**
- Les entrées-sorties
- Instructions

Conversions de types

Attention

Le langage C++ est faiblement typé.

⇒ écriture d'expressions avec des types différents

par exemple : `1+2.5>true`

Résolution d'expression de types mixtes

Le compilateur convertit implicitement les données pour satisfaire un type unique dans les expressions. L'ordre des conversions est le suivant :

```
char -> int -> float -> double
      bool -> int
      signed -> unsigned
```

Conversions de types

- lors de l'évaluation d'une expression :
ex : $3.5+1$ sera calculé comme $3.5+1.0$
↪ pas de perte d'information
- lors d'une affectation :
`int a = 3.5;` sera effectué comme `int a = 3;`
↪ perte d'information possible

Conversions de types

Le cas des booléens

Les booléens sont vus comme des nombres :

- la valeurs numérique 0 \rightarrow **false**
- toute valeur numérique \neq 0 \rightarrow **true**

Exemple

- `1 && true \Rightarrow true`
- `1.5 && true \Rightarrow true`
- `0 || false \Rightarrow false`
- `int a=true; \Rightarrow a=1`

Conversions de types explicite

On peut forcer le changement de type en effectuant un **cast**.

Conversion de type par cast

L'expression : **(type1) exp;**

permet de convertir l'évaluation de **exp** dans le type **type1**.

Exemple

```
1 int a=3,b=4;  
2 double c= a/b;           // c=0.0  
3 double d= (double)a/((double) b) // d=0.75
```


Plan

2 Bases du langage C++

- Les type de données
- Les constantes
- Les variables
- Opérateurs
- Les conversions de type
- **Les entrées-sorties**
- Instructions

Les entrées-sorties

Comme pour tout langage de programmation il est souhaitable de pouvoir interagir avec le programme :

- saisir des valeurs au clavier
- afficher des valeurs à l'écran

En C++, les fonctionnalités d'entrée-sortie standards sont définies dans le fichier `iostream`.

```
1  #include <iostream>
2  int main() {
3      ...
4      return 0;
5  }
```

Instruction d'affichage

Definition

```
std::cout << exp;
```

- `exp` est une expression quelconque
- `cout` est le nom de la sortie standard (l'écran par défaut)
- l'opérateur d'écriture `<<` indique ici d'envoyer la valeur de l'expression `exp` sur le flux de sortie standard `cout`

```
1  #include <iostream>
2  int main() {
3      std::cout << 1;
4      return 0;
5  }
```

Instruction d'affichage

Definition

```
std::cout << exp;
```

exp peut être :

- expression arithmétique
- expression booléenne
- une constante ou une variable de type standard
- `std::endl` → instruction de retour à la ligne

On peut enchaîner les affichages :

```
std::cout << exp1 << exp2 << exp3 << ... << expn;
```

Instruction d'affichage : exemple

```
1  #include <iostream>
2  int main() {
3      int a = 18;
4      float b = 2.3;
5      std::cout << "l'entier a = " << a << std::endl;
6      std::cout << "le flottant b = " << b << std::endl;
7      return 0;
8  }
```

ce programme affichera à l'écran :

l'entier a = 18

le flottant b = 2.3

Instruction de saisie clavier

Definition

```
std::cin >> var;
```

- var est un identificateur de variable valide
- std::cin est le nom de l'entrée standard (le clavier)
- l'opérateur de lecture >> indique ici d'affecter la valeur du flux d'entrée standard dans la variable var

```
1 #include <iostream>
2
3 int main() {
4     int a;
5     std::cin >> a;
6     return 0;
7 }
```

Instruction de saisie clavier

Definition

```
std::cin >> var;
```

Attention

- `var` est un identificateur de variable valide

Une saisie clavier est l'action d'écriture d'une donnée en mémoire. Pour savoir où écrire en mémoire, on le spécifie par l'identificateur d'une variable (*dualité mémoire/variable*)

var doit être une variable existante dans la mémoire!!!

On peut enchaîner les saisies clavier :

```
std::cin >> var1 >> var2 >> ... >> varn;
```

Instruction de saisie clavier : exemple

```
1  #include <iostream>
2
3  int main() {
4      int a;
5      float b;
6      cout << "Entrez un entier puis un flottant: ";
7      std::cin >> a >> b;
8      return 0;
9  }
```

ce programme affichera à l'écran :

Entrez un entier puis un flottant

et attendra la saisie au clavier d'un entier et d'un réel qu'il affectera respectivement à la variable a et b.

Instruction de saisie clavier : séparateurs

Si plusieurs valeurs sont saisies au clavier les affectations se font dans l'ordre des saisies.

Pour séparer 2 valeurs à saisir, on peut utiliser :

- une tabulation
- un retour à la ligne
- un espace

Instruction d'affichage : gabarit

Un gabarit d'affichage

permet de spécifier un nombre minimum de caractère à afficher.

↪ des espaces sont ajoutés implicitement si nécessaire

En C++

On utilise des manipulateurs pour formater l'affichage avec un gabarit. Les manipulateurs disponibles dans `<iomanip>` sont :

- `std::left` → aligne l'affichage à gauche du gabarit
- `std::right` → aligne l'affichage à droite du gabarit
- `std::setw(i)` → `i` est le nombre de caractères du gabarit
↪ **actif uniquement sur l'affichage suivant.**

Exemple : affichage avec gabarit

```

1 #include <iostream>
2 #include <iomanip>
3 int main(){
4     int a = 2;
5     std::cout<<"déf   : a= "<<a<<std::endl;
6     std::cout<<"gab  1: a= "<<std::setw(1)<<a<<std::endl;
7     std::cout<<"gab  2: a= "<<std::setw(2)<<a<<std::endl;
8     std::cout<<"gab  3: a= "<<std::setw(3)<<a<<std::endl;
9     std::cout<<"gab  4: a= "<<std::setw(3)<<std::right<<a
10        <<std::endl;
11     return 0;
12 }
```

Ce code affiche à l'écran :

déf : a = 2

gab 1 : a = 2

gab 2 : a = 2

gab 3 : a = 2

gab 4 : a = 2

Affichage : format des entiers

On peut contrôler le format d'affichage des entiers par un manipulateur.

Les manipulateurs disponibles sont :

- `std::hex` → affichage en hexadécimal
- `std::dec` → affichage en décimal
- `std::octal` → affichage en octal

Remarque

Par défaut c'est le manipulateur `std::dec` (décimal) qui est actif. Un manipulateur de format reste actif tant qu'on en active pas un autre.

Exemple : affichage avec format des entiers

```
1 #include <iostream>
2 int main(){
3     int a = 12; int b=8;
4     std::cout<< "a = "<<a<< " (défaut)"<<std::endl;
5     std::cout<<std::oct
6         << "a = "<<a<< " (octal)"<<std::endl;
7     std::cout<< "b = "<<b<<std::endl;
8     std::cout<<std::hex
9         << "a = "<<a<< " (hexadécimal)"<<std::endl;
10    return 0;
11 }
```

Ce code affiche à l'écran :

a = 12 (défaut)

a = 14 (octal)

b = 10

a = c (hexadécimal)

Affichage des booléens

On peut contrôler le format d'affichage des booléens par un manipulateur : **affichage true, false ou 1, 0.**

Les manipulateurs disponibles sont :

- `std::boolalpha` → affichage true ou false
- `std::noboolalpha` → affichage 1 ou 0

Remarque

Par défaut c'est `std::noboolalpha` qui est actif.
Ce manipulateur reste actif tant qu'on en active pas un autre.

Exemple : affichage des booléens

```
1 #include <iostream>
2
3 int main(){
4     bool a = true; bool b=false;
5     std::cout<< "a = "<<a<<" (default)"<<std::endl;
6     std::cout<<std::boolalpha
7         <<"a = "<<a<<" (boolalpha)"<<std::endl;
8     std::cout<<std::noboolalpha
9         << "a = "<<a<<" (noboolalpha)"<<std::endl;
10    std::cout<< "b = "<<b<<std::endl;
11    return 0;
12 }
```

Ce code affiche à l'écran :

a = 1 (default)

a = true (boolalpha)

a = 1 (noboolalpha)

b = 0

Affichage : format des réels

On peut contrôler le format d'affichage des réels par des manipulateurs.

Manipulateurs de format

- `std::fixed` → notation décimale
- `std::scientific` → notation scientifique

Manipulateur de la précision

- `std::setprecision(i)` où `i` est le nombre de chiffres significatifs choisis.
- besoin de `#include <iomanip>`

`std::setprecision` n'agit que sur l'affichage et non sur la valeur stockée en mémoire.

Exemple : format des réels

```
1 #include <iostream>
2
3 int main(){
4     float x = 89.786543;
5
6     std::cout<<"x= "<<x<<" (default)"<<std::endl;
7     std::cout<<std::fixed
8         <<"x= "<<x<<" (fixed)"<<std::endl;
9     std::cout<<std::scientific
10        <<"x= "<<x<<" (scientific)"<<std::endl;
11     return 0;
12 }
```

Ce code affiche à l'écran :

x= 89.7865 (default)

x= 89.786545 (fixed)

x= 8.978654e+01 (scientific)

Exemple : précisions d'affichage des réels

```
1 #include <iostream>
2 #include <iomanip>
3 int main(){
4     float x = 89.786543;
5
6     std::cout<<"x= "<<x<<" (default)"<<std::endl;
7     std::cout<<std::setprecision(2)
8         <<"x= "<<x<<" (precision 2)"<<std::endl;
9     std::cout<<std::setprecision(12)
10        <<"x= "<<x<<" (precision 12)"<<std::endl;
11     return 0;
12 }
```

Ce code affiche à l'écran :

x= 89.7865 (default)

x= 90 (precision 2)

x= 89.7865447998 (precision 12)

Plan

2 Bases du langage C++

- Les type de données
- Les constantes
- Les variables
- Opérateurs
- Les conversions de type
- Les entrées-sorties
- Instructions

Instructions

Un programme impératif est constitué d'une succession d'instructions exécutées les unes après les autres.

Definition

Une instruction correspond à une étape atomique dans le programme.

↔ toute instruction s'exécute complètement.

En C++, une instruction est terminée par un **point-virgule**.

Exemple

```
int a=3,c;  
c=a+10;
```

Bloc d'instructions

Definition

Un bloc d'instruction est constitué par un ensemble d'instructions délimitées par des accolades

```
{ instr1; instr2; ...; instrn; }
```

Les blocs d'instruction peuvent être :

- imbriqués :

```
{ instr1; { instr2; instr3; } }
```
- disjoints :

```
{ instr1; instr2; } { instr3; instr4; }
```

Remarque

La fonction `main` en C++ correspond au bloc d'instructions définissant le programme.

Plan

- 1 Introduction
- 2 Bases du langage C++
- 3 Structures de contrôle**
- 4 Fonctions
- 5 Pointeurs
- 6 Tableaux
- 7 Structures de données (Facultatif)

Intérêt des structures de contrôle

Contrôler l'exécution des instructions du programme :

↪ **notion d'algorithmes**

- exécution conditionnelle (si alors sinon)
- exécuter plusieurs fois les mêmes instructions (boucle pour)
- exécuter des instructions tant que (boucle tant que)

Plan

3 Structures de contrôle

■ Conditionnelles

- La boucle 'Pour'
- La boucle 'Tant que'
- La boucle 'do while'
- Erreurs classiques

if ... else

Definition

```
if (exp)
    instr1
else
    instr2
```

- `exp` est une expression booléenne
- `instr1` et `instr2` sont :
 - une instruction
 - un bloc d'instructions

Exemple: if ... else

Exemple

Déterminer si un entier est pair ou impair

```

1 #include <iostream>
2
3 int main(){
4     int a;
5     a= 17;
6     if (a%2 == 0)
7         std::cout << a << " est pair "<< std::endl;
8     else
9         std::cout << a << " est impair "<< std::endl;
10    return 0;
11 }
```

Exemple: if ... else

Exemple

Déterminer le plus grand et le plus petit entre deux entiers

```
1 #include <iostream>
2 using namespace std;
3
4 int main(){
5     int a=17,b=13,min , max;
6     if (a > b){
7         max=a;
8         min=b;
9     }
10    else {
11        max=b;
12        min=a;
13    }
14    std::cout << "le min est "<<min
15              <<" le max est " << max<<endl;
16    return 0;
17 }
```

Conditionnelle à choix multiple : switch

Lorsqu'on a besoin de faire un choix parmi plus de 2 possibilités, on peut

- imbriquer des conditionnelles `if .. else`

```
1 unsigned int a;  
2 std::cin >> a;  
3 if (a==1)  
4     std::cout << "a=1" << std::endl;  
5 else if (a==2)  
6     std::cout << "a=2" << std::endl;  
7 else if (a==3)  
8     std::cout << "a=3" << std::endl;  
9 else  
10    std::cout << "a>3" << std::endl;
```

- faire une conditionnelle à choix multiple

Conditionnelle à choix multiple : switch

Choix multiple switch

```
switch (exp) {  
    case cst1:  
        instr1; break;  
    ...  
    case cstn:  
        instrn; break;  
    default:  
        instrdefault;  
}
```

- **exp** est une expression à valeur entière
- **cst1, ..., cstn** sont des constantes entières
- **instr1, ..., instrn, instrdefault** sont des suites d'instructions

Les instructions sont exécutées en fonction de la valeur **exp**

Si la valeur de **exp** est une des constantes **cst1,...,cstn** on exécute la suite d'instructions correspondante sinon on exécute **instrdefault**.

Exemple : switch

```
1 unsigned int a;  
2 std::cin>>a;  
3 switch (a){  
4     case 1:  
5         std::cout<<"a=1"<<std::endl; break;  
6     case 2:  
7         std::cout<<"a=2"<<std::endl; break;  
8     case 3:  
9         std::cout<<"a=3"<<std::endl; break;  
10    default:  
11        std::cout<<"a>3"<<std::endl;  
12 }
```

L'affichage sera

- **a=1** si la valeur de a est 1
- **a=2** si la valeur de a est 2
- **a=3** si la valeur de a est 3
- **a>3** si la valeur de a est différent de 1,2 ou 3

Plan

3 Structures de contrôle

- Conditionnelles

- La boucle 'Pour'

- La boucle 'Tant que'

- La boucle 'do while'

- Erreurs classiques

Répétition d'instructions : la boucle pour

Intérêt : répéter un nombre de fois donné une même suite d'instructions.

Exemple

calculer la somme des entiers entre 1 et 10

```
s := 0;  
pour i de 1 à 10 faire  
    s := s + i;  
fin pour;
```

En C++, les boucles **pour** sont effectuées avec l'instruction **for**

L'instruction for

Definition

```
for (exp1; exp2; exp3)  
    instr
```

- exp1 est une expression quelconque évaluée une seule fois au début de la boucle
- exp2 est une expression booléenne qui permet d'arrêter la boucle
- exp3 est une expression quelconque évaluée à chaque tour de boucle (en dernier).
- instr est une instruction ou un bloc d'instructions

L'instruction for

Definition

```
for (exp1; exp2; exp3)  
    instr
```

- **exp1** est une expression quelconque évaluée une seule fois au début de la boucle

On se sert de **exp1** pour initialiser la variable de boucle.

Exemple

exp1 est remplacée par `int i=1` ou `i=1` si `i` est déjà déclaré.

L'instruction for

Definition

```
for (exp1; exp2; exp3)  
    instr
```

- **exp2** est une expression booléenne

Si **exp2** est :

- vrai : la boucle for continue et on exécute instr
- faux : on sort de la boucle for sans exécuter instr

Exemple

exp2 est remplacée par `i<11`

L'instruction for

Definition

```
for (exp1; exp2; exp3)  
    instr
```

- **exp3** est une expression quelconque évaluée à chaque tour de boucle (en dernier).

On sert de **exp3** pour modifier la variable de boucle (incrémentation ou décrémentation) :

Exemple

exp3 est remplacée par `i=i+1` ou `i++`

L'instruction for : schéma d'exécution

Definition

```
for (exp1; exp2; exp3)  
    instr
```

- 1 exp1
...
- 2 exp2 → on continue car exp2=true
- 3 instr
- 4 exp3
...
- 5 exp2 → sort de la boucle car exp2=false

L'instruction for : exemple 1

Exemple

calculer la somme des entiers entre 1 et 10

```
1 #include <iostream>
2 using namespace std;
3
4 int main(){
5
6     int i,s;
7     s=0;
8     for (i=1;i<11;i++) // i++ est équivalent à i=i+1
9         s=s+i;
10
11     std::cout <<"la somme est "<<s<<std::endl;
12     return 0;
13 }
```

ce programme affiche : la somme est 55

L'instruction for : exemple 2

Exemple

afficher les entiers entre 1 et 10 qui sont multiples de 2 ou de 3

```
1 #include <iostream>
2
3 int main(){
4     int i;
5     for (i=1;i<11;i++){
6         if (i%2==0 || i%3==0)
7             std::cout << i<< " ";
8     }
9     std::cout << std::endl;
10    return 0;
11 }
```

ce programme affiche : 2 3 4 6 8 9 10

L'instruction for : ce qu'il ne faut pas faire

Attention aux boucles qui ne se terminent jamais!!!
les boucles infinies ...

```
1  int i , s ;  
2  s=0;  
3  for ( i=1 ; i<11 ; s=s+i )  
4      ...
```

→ la variable de boucle n'est pas incrémentée

```
1  int i , s ;  
2  for ( i=1 ; i!=10 ; i+=2 )  
3      ...
```

→ la condition d'arrêt de la boucle n'est jamais atteinte

Plan

3 Structures de contrôle

- Conditionnelles
- La boucle 'Pour'
- **La boucle 'Tant que'**
- La boucle 'do while'
- Erreurs classiques

Répétition d'instructions : la boucle tant que

Intérêt : répéter une instruction tant qu'une condition est vérifiée

Exemple

calculer la somme des entiers entre 1 et 10

```
s := 0;  
i := 1;  
tant que i < 11 faire  
    s := s + i;  
    i := i + 1;  
fin tant que;
```

En C++, les boucles **tant que** se font avec l'instruction `while`

L'instruction while

Definition

```
while (exp)  
  instr
```

- `exp` est une expression booléenne permettant de contrôler la boucle
- `instr` est une instruction ou un bloc d'instructions

L'instruction while

Definition

```
while (exp)  
  instr
```

- `exp` est une expression booléenne permettant de contrôler la boucle

Si `exp` est :

- vrai : la boucle `while` continue et on exécute `instr`
- faux : on sort de la boucle `while` sans exécuter `instr`

ex : `exp` est remplacée par `i<11`

L'instruction while : schéma d'exécution

Definition

```
while (exp)  
    instr;
```

1 exp → on rentre dans la boucle car exp=true

2 instr

...

3 exp → on continue car exp=true

4 instr

...

5 exp → on sort de la boucle car exp=false

...

L'instruction while : exemple 1

Exemple

calculer la somme des entiers entre 1 et 10

```
1 #include <iostream>
2
3 int main(){
4     int i,s;
5     s=0;
6     i=1;
7     while (i<11){
8         s=s+i;
9         i++; // on peut aussi écrire i=i+1
10    }
11    std::cout << "la somme est " << s << std::endl;
12    return 0;
13 }
```

ce programme affiche : la somme est 55

L'instruction while : exemple 2

Exemple

afficher les entiers entre 1 et 10 qui sont multiples de 2 ou de 3

```
1 #include <iostream>
2
3 int main(){
4     int i;
5     i=1;
6     while (i<11){
7         if (i%2==0 || i%3==0)
8             std::cout << i<< " ";
9         i++;
10    }
11    std::cout << std::endl;
12    return 0;
13 }
```

ce programme affiche : 2 3 4 6 8 9 10

L'instruction while : exemple 3

Exemple

calcul de la plus petite puissance de 2 supérieure à un entier

```
1 #include <iostream>
2
3 int main(){
4     int a,p;
5     a=27;
6     p=1;
7     while (p<a){
8         p=2*p;
9     }
10    std::cout << p << " est supérieur à " << a << std::endl;
11    return 0;
12 }
```

ce programme affiche : 32 est supérieur à 27

Plan

3 Structures de contrôle

- Conditionnelles
- La boucle 'Pour'
- La boucle 'Tant que'
- La boucle 'do while'
- Erreurs classiques

L'instruction `do while`

Parfois, il est souhaitable d'exécuter le corps de boucle avant la condition de boucle (`instr` avant `exp`).

Dans ce cas, on peut utiliser l'instruction `do {} while;`

Definition

```
do {  
    instr;  
} while (exp);
```

- `instr` et `exp` sont identiques à ceux utilisés dans la boucle `while` classique

RAPPEL : schéma d'exécution du while

Definition

```
while (exp)  
  instr
```

1 **exp** → sort de la boucle si **exp**=false

2 **instr**

...

3 **exp** → sort de la boucle si **exp**=false

4 **instr**

...

5 **exp** → sort de la boucle si **exp**=false

6 **instr**

...

L'instruction `do{} while` : schéma d'exécution

Definition

```
do {  
    instr;  
}  
while (exp);
```

1 instr

...

2 $\text{exp} \rightarrow \text{sort de la boucle si } \text{exp} = \text{false}$

3 instr

...

4 $\text{exp} \rightarrow \text{sort de la boucle si } \text{exp} = \text{false}$

5 instr

...

L'instruction `do{} while` : exemple

calculer le plus grand entier tel que son carré est inférieur à un entier `x`.

```
1 #include <iostream>
2
3 int main(){
4     int i,s,x;
5     x=1000;
6     i=0;
7     do {
8         i++;
9         s=i*i;
10    } while (s <= x);
11    i=i-1;
12    std::cout << "l'entier est "<<i<<std::endl;
13    return 0;
14 }
```

ce programme affiche : l'entier est 31

Plan

3 Structures de contrôle

- Conditionnelles
- La boucle 'Pour'
- La boucle 'Tant que'
- La boucle 'do while'
- Erreurs classiques

Structures de contrôle : erreur classique

L'erreur classique avec les structures de contrôle est l'oubli d'accolade pour définir un bloc d'instructions :

```
1  ...
2  if (a > b){
3      max=a;
4      min=b;
5  }
6  else
7      max=b;
8      min=a; // ATTENTION: n'appartient pas au else
9
10 std::cout << "le min est " << min
11      << " le max est " << max << std::endl;
12 ...
```

Structures de contrôle : erreur classique

L'erreur classique avec les structures de contrôle est l'oubli d'accolade pour définir un bloc d'instructions :

```
1  ...
2  if (a > b)
3      max=a;
4      min=b;
5  else {
6      max=b;
7      min=a; // grâce aux accolades appartient au else
8  }
9  std::cout << "le min est " << min
10             << " le max est " << max << std::endl;
11  ...
```

L'oubli des accolades sur le bloc du `if` génère une erreur de compilation car le `else` se retrouve isolé.

Structures de contrôle : erreur classique

L'erreur classique avec les structures de contrôle est l'oubli d'accolade pour définir un bloc d'instruction :

```
1  ...  
2  int i,s;  
3  s=0;  
4  i=1;  
5  while (i<11)  
6      s=s+i;  
7      i++; // n'appartient pas à la boucle  
8  
9  std::cout << "la somme est " << s << std::endl;  
10 ...
```

Structures de contrôle : erreur classique

L'erreur classique avec les structures de contrôle est l'oubli d'accolade pour définir un bloc d'instruction :

```
1  ...
2  int i,s;
3  s=0;
4  i=1;
5  while (i<11){
6      s=s+i;
7      i++; // grâce aux accolades appartient à la boucle
8  }
9  std::cout << "la somme est " << s << std::endl;
10 ...
```

Récapitulatif

- les variables ont un type (ex : int, float, bool)
- on peut calculer grâce aux opérateurs (+,*,%,...)
- on modifie un programme par des affectations (ex : a=6)
- faiblement typé → conversions de type implicite
- on affiche les valeurs à l'écran avec `std::cout`«
- on saisit les valeurs au clavier avec `std::cin`»
- on peut écrire des algorithmes avec les structures de contrôle classiques : if else, for, while, switch

Plan

- 1 Introduction
- 2 Bases du langage C++
- 3 Structures de contrôle
- 4 Fonctions**
- 5 Pointeurs
- 6 Tableaux
- 7 Structures de données (Facultatif)

Plan

4 Fonctions

■ Introduction

- Définition
- Appel d'une fonction
- Portée des variables
- Définir des constantes
- Où placer les fonctions
- Récursivité

A quoi sert une fonction

Les fonctions servent à partitionner les gros traitements en traitements plus petits.

Elle permettent de définir des briques de calcul qui sont :

- identifiables facilement,
- conçues pour faire un traitement précis,
- réutilisables.

Remarque

Les programmes sont plus lisibles et donc plus faciles à maintenir.

Fonction : une brique de calcul paramétrée

On veut écrire un programme qui affiche la décomposition en facteurs premiers d'un nombre entier.

```
1  int main(){
2      int x,y,i,k;
3      std::cout<<"donnez une entier "<<std::endl;
4      std::cin>>x;
5      y=x;i=2;
6      while (x>=i){
7          if ( i est premier && (x%i==0)){
8              //calcul plus grande puissance k de i divisant x
9              std::cout<<i<<" puissance "<<k<<
10                 <<" divise "<<y<<std::endl;
11              // on calcule d= i^k
12              x= x/d;
13          }
14          i++;
15      }
16      return 0;
17 }
```

Fonction : une brique de calcul paramétrée

Deux solutions :

- ✗ on complète le code directement par les instructions de calcul correspondantes.
- ✓ on complète le code par l'appel à trois briques de calcul paramétrées.

```
1 bool premier(unsigned int n){  
2     if (n==0|| n==1) return false;  
3     else if (n==2)    return true;  
4         else {  
5             for (unsigned int i=2; i*i<=n; i++)  
6                 if (n%i==0) return false;  
7         }  
8     return true;  
9 }
```


Fonction : une brique de calcul paramétrée

```
1 int puissanceInf(int i, int x){
2     int j =1;int p=i;
3     while(p<=x && (x%p==0)){
4         j++;
5         p*=i;
6     }
7     return j-1;
8 }
```

```
1 int puissance(int x, int n){
2     int j;int p = 1;
3     for (j=1;j<=n;j++)
4         p*=x;
5     return p;
6 }
```

Fonction : une brique de calcul paramétrée

Le programme s'écrit donc :

```
1  int main(){
2      int x,y,i,k;
3      std::cout<<"donnez une entier "<<std::endl;
4      std::cin>>x;
5      y=x; i=2;
6      while (x>=i){
7          if ( premier(i) && (x%i==0)){
8              k=puissanceInf(i,x);
9              std::cout<<i<<" puissance "<<k<<
10                 <<" divise "<<y<<std::endl;
11              x= x/ puissance(i,k);
12          }
13          i++;
14      }
15      return 0;
16 }
```

Qu'est-ce qu'une fonction

Definition

Une fonction est une description formelle d'un calcul en fonction de ses arguments

Exemple

Si on vous donne deux variables entières a et b décrivez les instructions du langage nécessaires pour calculer a^b .

→ cela rejoint la notion d'algorithme

Qu'est-ce qu'une fonction

Comme en algorithmique, une fonction comportera :

- **un nom** (l'identifiant de manière non-ambigu),
- **des paramètres d'entrées**,
- **des paramètres de sortie**,
- **des variables locales**,
- **une description dans le langage** de ce que fait la fonction.

Plan

4 Fonctions

- Introduction
- **Définition**
- Appel d'une fonction
- Portée des variables
- Définir des constantes
- Où placer les fonctions
- Récursivité

Définition de fonctions

Definition

```
type_retour nom(liste_param_formel){  
    corps  
}
```

- `nom` correspond au nom donné à la fonction
- `type_retour` est le type du résultat de la fonction
- `liste_param_formel` est la liste des variables d'entrée de la fonction
- `corps` correspondant aux instructions effectuées par la fonction en fonction des paramètres formels.

Fonctions : nom

Definition

Le nom d'une fonction correspond à un identifiant unique permettant d'effectuer l'appel sans aucune ambiguïté

Attention

Une fonction ne peut être assimilée à une variable et vice-versa. On les différencie par la présence de parenthèse après l'identifiant d'une fonction.

```
1 int a();  
2 a=3; // ERREUR 'a' n'est pas une variable  
3 int b;  
4 b(); // ERREUR 'b' n'est pas une fonction
```

Fonctions : nom

Attention

Une variable et une fonction ne peuvent pas partager le même nom au sein d'un même bloc.

```
1 {  
2   int a(); // déclaration fonction  
3   int a;  // déclaration variable  
4 }
```

Le compilateur générera l'erreur suivante :

*erreur : int a redeclared as different kind of symbol erreur :
previous declaration of int a()*

Fonctions : type de retour

Les fonctions ne possèdent qu'un seul paramètre de sortie.

Ce paramètre :

- n'est pas identifié par une variable particulière
- **est spécifié uniquement par son type**

```
1 int f();    // f renvoie un résultat entier  
2 double g(); // g renvoie un résultat flottant
```

Si la fonction ne renvoie rien, on utilise le type **void**.

ex : **void** h();

Fonctions : paramètres formels

La liste des paramètres formels d'une fonction est :

- **vide** si la fonction n'a aucun paramètre
- de la forme : $type_1 \ p_1, \dots, type_n \ p_n$

La déclaration des variables formelles ($type_i \ p_i$) est de la forme :

- **type** **p** pour une variable **p** simple
- ... (à voir plus loin dans le cours)

où **type** est un type de base et **p** est le nom de la variable formelle manipulable dans la fonction.

Fonctions : paramètres formels

Les paramètres formels d'une fonction permettent :

- de définir le code de la fonction à partir de données inconnues.
- de transmettre (**lors de l'appel**) les données réelles sur lesquelles doit agir la fonction.

Ils ne sont connus que dans le bloc associé à la fonction.

```
1 int f(int a, int b){...}  
2 void g(double a){...}  
3 a=12; // erreur a n'est pas déclaré
```

Fonctions : corps

Definition

Le corps d'une fonction est le bloc défini juste après la déclaration de l'interface de la fonction :

```
type_retour nom(liste_param_formel)
    { corps }
```

Le corps d'une fonction est composé :

- de la déclaration des variables locales
- de la description du code qui sera exécuté par la fonction

```
1 int plus(int x,int y){
2     int z;
3     z=x+y;
4     return z;
5 }
```

Fonction : le mot clé return

Definition

Dans une fonction, le mot clé **return** *exp* permet :

- d'arrêter l'exécution du corps
- de renvoyer la valeur de l'expression *exp* au bloc appelant

Attention

La valeur de *exp* doit être du même type que le type de retour de la fonction (*sinon conversion implicite*)

```
1 double inverse(int x){  
2     return 1/x;  
3 }
```

Fonction : le mot clé return

Le mot clé **return** peut apparaître plusieurs fois dans une même fonction. C'est le premier **return** rencontré qui stoppera le corps de la fonction (ex : conditionnelle).

```
1 int impair(int x){  
2     if (x%2 == 0)  
3         return 0;  
4     else  
5         return 1;  
6 }
```

Fonction : le mot clé return

Le mot clé **return** n'est pas obligatoire lorsque le retour de la fonction est **void**. La fonction s'arrête dès qu'elle atteint la fin du bloc.

```
1 void bonjour(){  
2     std::cout<<" Bonjour "<<std::endl;  
3 }
```

Fonction : le mot clé return

On peut toutefois forcer l'arrêt en appelant **return** sans expression.

```
1 void bonjour(int heure){  
2     if (heure > 8 && heure < 20){  
3         std::cout<<"Bonjour"<<std::endl;  
4         return;  
5     }  
6     std::cout<<"Bonsoir"<<std::endl;  
7 }
```


Fonctions : pas d'amalgame

ATTENTION

La déclaration d'une fonction est une description formelle d'un calcul, elle n'exécute rien toute seule : il faut *appeler* la fonction pour que les instructions du corps soient exécutées.

```
1 void bonjour(){  
2     std::cout<<" Bonjour"<<std::endl;  
3 }
```

Ce code ne fait que déclarer une fonction, il n'affichera rien à l'écran.

↔ c'est l'instruction **bonjour()** ; dans le programme qui exécutera ce code.

Plan

4 Fonctions

- Introduction
- Définition
- **Appel d'une fonction**
- Portée des variables
- Définir des constantes
- Où placer les fonctions
- Récursivité

Appel de fonction

Definition

```
var = nom(liste des paramètres effectifs);
```

ou

```
nom(liste des paramètres effectifs);
```

- la liste des paramètres effectifs correspond à l'ensemble des variables et des constantes que l'on souhaite donner comme argument à la fonction. ex : `max(2,3)`.

Attention : le passage des arguments se fait par copie.

la valeur des paramètres effectifs est copiée dans les variables formelles correspondantes.

Appel de fonction : exemple

```
1 #include <iostream>
2
3 int max(int a, int b){
4     if (a>b) return a;
5     else    return b;
6 }
7 int main(){
8     int x,y;
9     x=3;
10    y=5;
11    std::cout<<"le max est : "<<max(x,y)<<std::endl;
12    return 0;
13 }
```

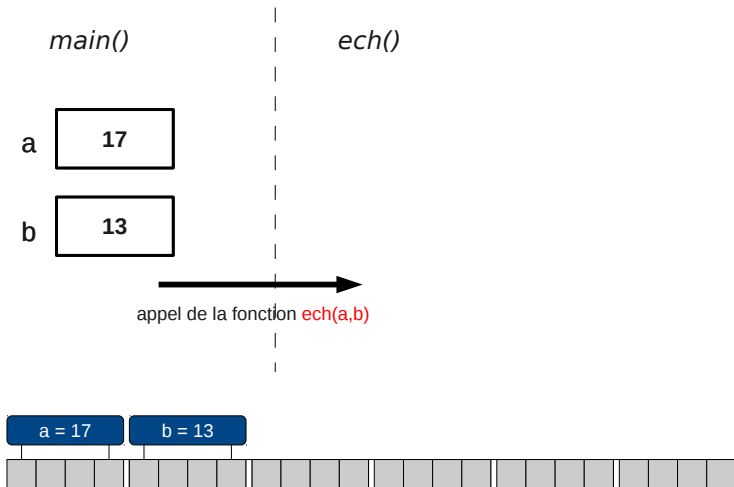
Fonction : passage de paramètres par valeur

Les paramètres d'une fonction sont **toujours** initialisés par une **copie des valeurs** des paramètres réels.

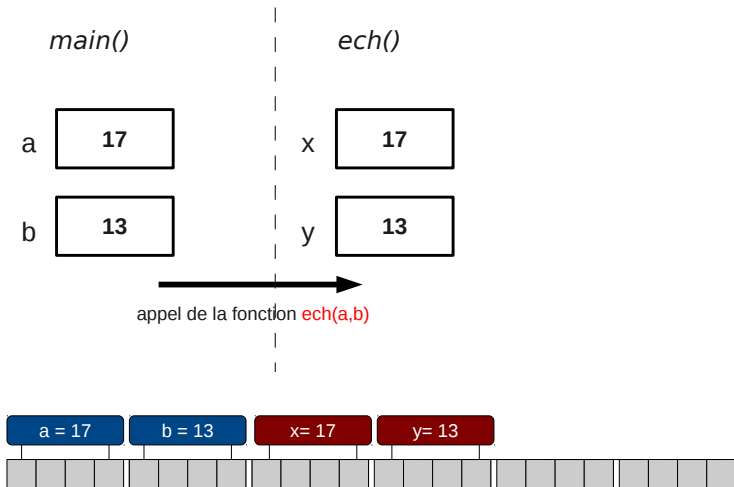
Modifier la valeur des paramètres formels dans le corps de la fonction **ne change en aucun cas la valeur des paramètres réels**.

```
1 void ech(int x, int y){
2     int r;
3     r=x; x=y; y=r;
4 }
5 int main(){
6     int a,b;
7     a=17;b=13;
8     std::cout<<"a = "<<a<<" b = "<<b<<std::endl;
9     ech(a,b); // ne change pas la valeur de a et de b
10    std::cout<<"a = "<<a<<" b = "<<b<<std::endl;
11 }
```

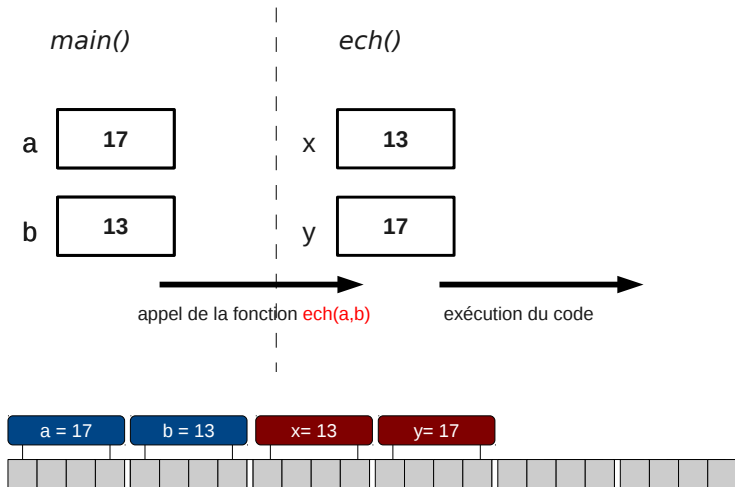
Fonction : passage de paramètres par valeur



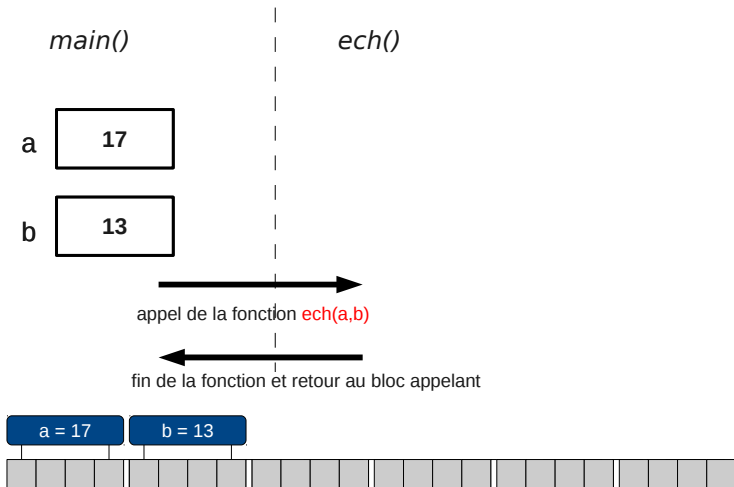
Fonction : passage de paramètres par valeur



Fonction : passage de paramètres par valeur



Fonction : passage de paramètres par valeur



- Introduction
- Définition
- Appel d'une fonction
- **Portée des variables**
- Définir des constantes
- Où placer les fonctions
- Récursivité

Où utiliser une variable ?

Question

Quand on déclare une variable, où est-elle connue ?
où peut-on l'utiliser ?

Réponse

Dans tout le bloc où elle a été déclarée et à partir de sa déclaration.

Portée d'une variable

Definition

On appelle **Portée** d'une variable, la portion du code où cette variable est connue et utilisable.

Exemple 1

Quelles sont les portées de `a` et de `x` ?

```
1 int main()  
2 {  
3     int a;  
4     float x;  
5     ...  
6     ...  
7     return 0;  
8 }
```

Exemple 1

Quelles sont les portées de a et de x ?

```

1  int main()
2  {                                <— début du bloc
3      int a;                        <— début de la portée de a
4      float x;                     <— début de la portée de x
5      ...
6      ...                          <— utilisation possible de a et x
7      return 0;
8  }                                <— fin de bloc et de portée de a et x
    
```

Exemple 2

Est-ce que l'utilisation de a et de x est conforme à leur portée ?

```
1 int main()  
2 {  
3     int a=3;  
4     x=2./a;  
5     float x;  
6     std::cout<<"a = "<<a<<" x = "<<x<<std::endl;  
7     return 0;  
8 }
```

Exemple 2

Est-ce que l'utilisation de a et de x est conforme à leur portée ?

```
1 int main()  
2 {  
3     int a=3;           <— début portée de a  
4     x=2./a;           <— on peut utiliser a mais pas x  
5     float x;          <— début portée de x  
6     std::cout<<"a = "<<a<<" x = "<<x<<std::endl;  
7     return 0;  
8 }
```


Exemple 2

```
1 #include <iostream>
2 int main()
3 {
4     int a=3;
5     x=2./a;
6     float x;
7     std::cout<<"a = "<<a<<" x = "<<x<<std::endl;
8     return 0;
9 }
```

Le compilateur vous affiche alors le message suivant :

exemple2.c :4 error : x was not declared in this scope)

Portée d'une variable

Definition

On appelle **Portée** d'une variable, la portion du code où cette variable est connue et utilisable.

Remarque

Dans le cas de blocs imbriqués, une variable est utilisable dans tout le bloc où elle a été déclarée, à partir de sa déclaration et également dans les blocs internes au bloc de la déclaration.

Exemple

```
1 int main(){  
2     int i;  
3     for (i=0;i<10;i++){  
4         int S;  
5         S=2*i*i-1;  
6         std::cout<<"S="<<S<<std::endl;  
7     }  
8     return 0;  
9 }
```

Exemple

```

1 int main(){
2     int i;                <— début portée de i
3     for (i=0;i<10;i++){
4         int S;            <— début portée de S
5         S=2*i*i-1;
6         std::cout<<"S="<<S<<std::endl;
7     }                    <— fin portée S
8     return 0;
9 }                        <— fin portée i
    
```

Exemple

```
1 int main(){
2     int i;
3     for (i=0;i <10;i++){
4         int S;
5         S=2*i*i-1;
6         std::cout<<"S="<<S<<std::endl;
7     }
8     std::cout<<"S="<<S<<std::endl;    ← impossible
9     return 0;
10 }
```

Question

Que se passe-t-il si on a plusieurs variables de même nom ?

Réponse

- ❌ Impossible si les variables sont définies dans le même bloc
- ✅ Possible si les variables sont définies dans des blocs différents
↳ même si ils sont imbriqués

Exemple

Qu'affiche le code suivant ?

```
1 int main(){  
2     int a=3;  
3     {  
4         int a=5;  
5         std::cout<<"a="<<a<<std::endl;  
6     }  
7     return 0;  
8 }
```

Exemple

Qu'affiche le code suivant ?

```
1 int main(){  
2     int a=3;  
3     {  
4         int a=5;  
5         std::cout<<"a="<<a<<std::endl;  
6     }  
7     return 0;  
8 }
```

Le programme affiche :

a=5

Plusieurs variables ?

Question

Que se passe-t-il si on a plusieurs variables de même nom ?

Réponse

La variable utilisée est celle du bloc englobant le plus proche (au sens de l'inclusion).

Plusieurs variables ?

Question

Et dans le cas de fonctions ?

```
1 float cube(float x){  
2     float c=x*x*x;  
3     return c;  
4 }  
5 int main(){  
6     float c;  
7     c=cube(2);  
8     std::cout<<"le cube de 2 est "<<c<<std::endl;  
9     return 0;  
10 }
```

Plusieurs variables ?

Question

Et dans le cas de fonctions ?

Il n'y a pas de problème : la portée d'une variable étant le bloc où elle a été déclarée, la variable n'existe qu'à l'intérieur de la fonction.

Réponse

Tout ce qui a été dit précédemment s'applique pour les fonctions.

Plusieurs variables : continuons

Question

Comment fait-on si on veut tout de même utiliser la même variable dans plusieurs fonctions ?"

Réponse

Rien de plus facile, on passe la variable en paramètre !!!

Plusieurs variables : continuons

Remarque

Il peut être parfois embêtant de rajouter une variable à toutes les fonctions, si celles-ci doivent partager une variable commune.

Une solution :

Il est possible de définir des variables communes à tout le monde :
les variables globales.

Variables globales

Definition

Une variable est **globale** si elle est définie en dehors de tout bloc (et donc de toute fonction).

Question

La portée d'une variable étant son bloc, quelle est la portée d'une variable globale ?

Variables globales

Une variable globale est connue dans toute la partie du fichier qui suit sa déclaration.

```
1 void uneautrefonction(){
2     // ne peut pas utiliser a
3 }
4
5 int a;
6
7 void mafonction(){
8     // peut utiliser la variable a
9 }
10
11 int main(){
12     // peut utiliser la variable a
13     return 0;
14 }
```

Variables globales : c'est dangereux !

Qu'affiche le code suivant :

```
1 int i;  
2  
3 void coucou(){  
4     for (i=0;i<3;i++)  
5         std::cout<<"coucou "<<i<<" fois"<<std::endl;  
6 }  
7  
8 int main(){  
9     for (i=0;i<3;i++)  
10        coucou();  
11    return 0;  
12 }
```


Variables globales : c'est dangereux !

On a 3 appels à la fonction `coucou()`. Chaque exécution de cette fonction devrait afficher 3 messages. On s'attend donc à 9 messages. Au lieu de cela, nous obtenons :

`coucou 0 fois`

`coucou 1 fois`

`coucou 2 fois`

Pourquoi c'est dangereux !

Voilà l'exécution du programme :

main : $i=0$

main : appel de `coucou()`

coucou : $i=0$

coucou : affiche "coucou 0 fois"

coucou : $i=1$

coucou : affiche "coucou 1 fois"

coucou : $i=2$

coucou : affiche "coucou 2 fois"

coucou : $i=3$

coucou : $i < 3$? faux, on sort de la fonction

main : retour dans le main, i est toujours égal à 3

main : $i < 3$? faux on quitte le programme

Pourquoi c'est dangereux !

Conclusion :

- C'est une très mauvaise idée de modifier une variable globale.
- Mais alors quoi ? les variables globales restent constantes ?
- Pire que ça ! On oublie les variables globales et on définit des constantes.

Plan

4 Fonctions

- Introduction
- Définition
- Appel d'une fonction
- Portée des variables
- Définir des constantes
- Où placer les fonctions
- Récursivité

Définir une constante

Pour définir une constante, il est possible de définir une **macro** en tête du programme.

Syntaxe

```
#define NOM valeur
```

Exemple

```
#define MAX 1000  
#define PI 3.1415
```

Remarque : par convention, les macros sont notées en majuscules.

Exemple

```
1 #include <iostream>
2 #include <iomanip>
3
4 #define MAX 10
5 void produit(int a){
6     int i;
7     for (i=1;i<=MAX;i++)
8         std::cout<<std::setw(4)<<a*i;
9 }
10 int main(){
11     int i;
12     for (i=1;i<=MAX; i++){
13         produit(i);
14         std::cout<<std::endl;
15     }
16     return 0;
17 }
```

Exemple

On obtient alors :

1	2	3	4	5	6	7	8	9	10
2	4	6	8	10	12	14	16	18	20
3	6	9	12	15	18	21	24	27	30
4	8	12	16	20	24	28	32	36	40
5	10	15	20	25	30	35	40	45	50
6	12	18	24	30	36	42	48	54	60
7	14	21	28	35	42	49	56	63	70
8	16	24	32	40	48	56	64	72	80
9	18	27	36	45	54	63	72	81	90
10	20	30	40	50	60	70	80	90	100

Définir une constante

Pour définir une constante, il est aussi possible de définir des variables globales constantes.

Syntaxe

```
const type nom=valeur;
```

Exemple

```
const int max=10;  
const double pi=3.1415;
```


Exemple

```
1 #include <iostream>
2 const int max=10;
3 void produit(int a){
4     int i;
5     for (i=1;i<=max;i++)
6         std::cout<<std::setw(4)<<a*i;
7 }
8 int main(){
9     int i;
10    for (i=1;i<=max; i++){
11        produit(i);
12        std::cout<<std::endl;
13    }
14    return 0;
15 }
```

Exemple

On obtient alors :

1	2	3	4	5	6	7	8	9	10
2	4	6	8	10	12	14	16	18	20
3	6	9	12	15	18	21	24	27	30
4	8	12	16	20	24	28	32	36	40
5	10	15	20	25	30	35	40	45	50
6	12	18	24	30	36	42	48	54	60
7	14	21	28	35	42	49	56	63	70
8	16	24	32	40	48	56	64	72	80
9	18	27	36	45	54	63	72	81	90
10	20	30	40	50	60	70	80	90	100

Peut-on modifier une constante ?

```
1 const int max=10;  
2 int main(){  
3     max=11;  
4     return 0;  
5 }
```

Le compilateur affiche l'erreur suivante :

exemple.c :3 : error : assignment of read-only variable 'max'

Différence entre MACRO et variables constantes

Variable constante

C'est une vraie variable, avec un espace mémoire réservé. Le compilateur interdit juste de la modifier.

Macro

Ce n'est pas une variable. Il n'y a pas d'espace mémoire allouée pour. Au moment de la compilation, le compilateur commence par remplacer toutes les occurrences des macros par leurs valeurs, puis compile le nouveau fichier obtenu.

Plan

4 Fonctions

- Introduction
- Définition
- Appel d'une fonction
- Portée des variables
- Définir des constantes
- Où placer les fonctions
- Récursivité

Portée des fonctions

Question

Quand on déclare une fonction, à quel endroit peut-on l'utiliser ?

Réponse

Partout après sa déclaration !

Autrement dit, si une fonction `f1` appelle une fonction `f2`, la déclaration de `f2` doit se situer avant la fonction `f1`.

Exemple

```
1 void f1(){
2     std::cout<<"je suis f1"<<std::endl;
3 }
4 void f2(){
5     std::cout<<"je suis f2 et je peux me servir de f1"
6         <<std::endl;
7     f1();
8 }
9 int main(){
10     f1(); // je peux me servir de f1
11     f2(); // je peux me servir de f2
12 }
```

Il faut donc déclarer les fonctions par rapport à l'ordre dans lequel elles seront utilisées.

Problème

On souhaite définir ces deux fonctions :

$$pair(n) = \begin{cases} vrai & \text{si } n = 0 \\ impair(n-1) & \text{sinon} \end{cases}$$

et

$$impair(n) = \begin{cases} faux & \text{si } n = 0 \\ pair(n-1) & \text{sinon} \end{cases}$$

Quelle est la fonction qui se place avant l'autre ?

Problème

On souhaite définir ces deux fonctions :

$$pair(n) = \begin{cases} vrai & \text{si } n = 0 \\ impair(n - 1) & \text{sinon} \end{cases}$$

et

$$impair(n) = \begin{cases} faux & \text{si } n = 0 \\ pair(n - 1) & \text{sinon} \end{cases}$$

Quelle est la fonction qui se place avant l'autre ?

Aucune, elles sont inter-dépendantes!!!

Comment faire ?

Solution

On peut séparer la **déclaration** de la **définition** d'une fonction.

Dans le corps de la définition d'une fonction, je peux utiliser n'importe quelle fonction qui aura été déclarée précédemment (mais pas forcément définie).

Remarque

Rappelez-vous, on avait dit qu'il était possible d'utiliser une fonction n'importe où après sa *déclaration*...

Comment faire ?

La **déclaration** de la fonction consiste à donner la **signature** de la fonction, c'est-à-dire son type de retour et le type de chacun de ses paramètres.

Syntaxe

```
type nom(type,type,...,type);
```

Exemple

```
bool pair(int);  
bool impair(int);
```

Exemple

```
1 bool pair(int);  
2 bool impair(int);  
3  
4 bool pair(int n){  
5     if (n==0) return true;  
6     else      return impair(n-1);  
7 }  
8  
9 bool impair(int n){  
10    if (n==0) return false;  
11    else      return pair(n-1);  
12 }
```

Structure d'un fichier

```
1  /* Début déclarations des fonctions */
2  bool pair(int);
3  ...
4  ...
5  /* Fin déclarations des fonctions */
6
7  /* Début définitions des fonctions */
8  bool pair(int n){
9      if (n==0) return true;
10     else return impair(n-1);
11 }
12 ...
13 ...
14 /* Fin définitions des fonctions */
15
16 int main(){
17     ....
18     return 0;
19 }
```

Programmation modulaire

Remarque

Il est possible de scinder son programme (toutes les fonctions qui le composent) en plusieurs fichiers. On parle alors de **programmation modulaire**.

Dans les faits, il s'agit de mettre dans un fichier les signatures des fonctions et dans un autre les définitions. Tout programme qui voudra utiliser une des ces fonctions devra seulement inclure le fichier des signatures.

Fichier d'en-tête

Definition

Un fichier d'en-tête (ex : `fonction.h`) contient des déclarations de fonctions.

Exemple

```
1  /* Début déclarations des fonctions */
2  bool pair(int);
3  bool impair(int);
4  ...
5  /* Fin déclarations des fonctions */
```

Fichier de définition

Definition

Un fichier de définition (ex : `fonction.cpp`) contient les définitions des fonctions déclarées dans `fonction.h`.

Remarque

Les deux noms de fichiers doivent être les mêmes (`toto.h` et `toto.cpp`).

Mise en oeuvre

Dans le fichier de définition, il faut inclure le fichier de déclarations.

Syntaxe

```
#include "fonction.h"
```

```
1 #include "fonction.h"
2 /* Début définitions des fonctions */
3 bool pair(int n){
4     if (n==0) return true;
5     else return impair(n-1);
6 }
7
8 bool impair(int n){
9     if (n==0) return false;
10    else return pair(n-1);
11 }
12 ...
13 /* Fin définitions des fonctions */
```

Problème d'inclusion multiple

Si le fichier d'en-tête est inclus plusieurs fois, cela va provoquer une erreur (**redéclaration de fonction**). Pour éviter cela, il existe des directives pour n'inclure dans un projet qu'une seule fois un ensemble de déclarations.

```
1 #ifndef FONCTIONS_H
2 #define FONCTIONS_H
3
4 /* Début déclarations des fonctions */
5 bool pair(int);
6 bool impair(int);
7 ...
8 /* Fin déclarations des fonctions */
9 #endif
```

Plan

4 Fonctions

- Introduction
- Définition
- Appel d'une fonction
- Portée des variables
- Définir des constantes
- Où placer les fonctions
- Récursivité

Fonction récursive

Definition

Une fonction est **récursive** si elle s'appelle elle-même.

Mise en oeuvre implicite

Rendu possible car une fonction est utilisable dès qu'on a spécifié sa signature, ce qui est le cas dans le blocs de définition d'une fonction.

Étude d'un cas simple

Considérons le problème suivant :

- On veut calculer la somme des carrés des entiers compris dans un intervalle (entre m et n).
- Par exemple,
 $\text{SommeCarres}(5, 10) = 5^2 + 6^2 + 7^2 + 8^2 + 9^2 + 10^2$.

Première solution

```
1 #include <iostream>
2 int SommeCarres(int m, int n){
3     int i ,som=0;
4     for (i=m; i<=n; i++)
5         som=som+i*i;
6     return som;
7 }
8 int main(){
9     int m=5, n=10;
10    int sc=SommeCarres(5,10);
11    std::cout<<"Carres entre 5 et 10 :"<< sc<<std::endl;
12    return 0;
13 }
```

Vision récursive du problème

- S'il y a plus d'un nombre dans $[m..n]$, on ajoute le carré de m à la somme des carrés de $[m+1..n]$
- S'il n'y a qu'un nombre ($m=n$), le résultat est le carré de m

Mathématiquement,

$$SommeCarres(m, n) = \begin{cases} m * m + SommeCarres(m + 1, n) & \text{si } m \neq n \\ m * m & \text{sinon} \end{cases}$$

Solution récursive

```
1 #include <iostream>
2 int SommeCarres(int m, int n){
3     if (m==n)
4         return m*m;
5     else
6         return (m*m+SommeCarres(m+1,n));
7 }
8 int main(){
9     int sc=SommeCarres(5,9);
10    std::cout<<"somme des carres entre 5 et 9 : "
11        << sc<<std::endl;
12    return 0;
13 }
```


Trace des appels

$$\begin{aligned} \text{SommeCarres}(5, 9) &= 25 + \text{SommeCarres}(6, 9) \\ &= 25 + (36 + \text{SommeCarres}(7, 9)) \\ &= 25 + (36 + (49 + \text{SommeCarres}(8, 9))) \\ &= 25 + (36 + (49 + (64 + \text{SommeCarres}(9, 9)))) \\ &= 25 + (36 + (49 + (64 + 81))) \\ &= 25 + (36 + (49 + 145)) \\ &= 25 + (36 + 194) \\ &= 255 \end{aligned}$$

Principe de construction

- Des instructions résolvant les cas particuliers,
- Des instructions décomposant le problème en sous-problèmes,
- Des appels récursifs résolvant les sous-problèmes,
- Des instructions pour résoudre le problème à partir des solutions des sous-problèmes.

Un autre problème

La suite de Fibonacci :

$$\begin{cases} F(0) = 1 \\ F(1) = 1 \\ F(n+2) = F(n+1) + F(n) \end{cases}$$

Très facile à mettre en oeuvre !

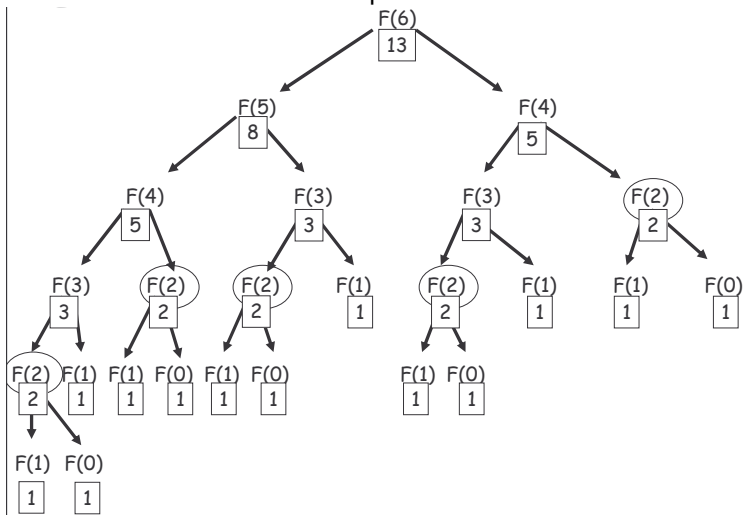
Solution récursive

La version récursive est très proche de l'écriture mathématique :

```
1 int Fibo(int n){  
2     if (n==0)|| (n==1)  
3         return 1;  
4     else  
5         return Fibo(n-1)+Fibo(n-2);  
6 }
```

Exécution

Certains termes sont calculés plusieurs fois :



Solution itérative

La solution itérative est moins intuitive :

```
1 int Fibo(int n){  
2     int a,b,F;  
3  
4     F=a=b=1;  
5  
6     while (n>1){  
7         F=a+b;  
8         a=b;  
9         b=F;  
10        n--;  
11    }  
12    return F;  
13 }
```

F	a	b	n
1	1	1	6

Solution itérative

La solution itérative est moins intuitive :

```

1  int Fibo(int n){
2      int a,b,F;
3
4      F=a=b=1;
5
6      while (n>1){
7          F=a+b;
8          a=b;
9          b=F;
10         n--;
11     }
12     return F;
13 }
```

F	a	b	n
1	1	1	6
2	1	2	5

Solution itérative

La solution itérative est moins intuitive :

```

1  int Fibo(int n){
2      int a,b,F;
3
4      F=a=b=1;
5
6      while (n>1){
7          F=a+b;
8          a=b;
9          b=F;
10         n--;
11     }
12     return F;
13 }
```

F	a	b	n
1	1	1	6
2	1	2	5
3	2	3	4

Solution itérative

La solution itérative est moins intuitive :

```
1 int Fibo(int n){  
2     int a,b,F;  
3  
4     F=a=b=1;  
5  
6     while (n>1){  
7         F=a+b;  
8         a=b;  
9         b=F;  
10        n--;  
11    }  
12    return F;  
13 }
```

F	a	b	n
1	1	1	6
2	1	2	5
3	2	3	4
5	3	5	3

Solution itérative

La solution itérative est moins intuitive :

```
1 int Fibo(int n){  
2     int a,b,F;  
3  
4     F=a=b=1;  
5  
6     while (n>1){  
7         F=a+b;  
8         a=b;  
9         b=F;  
10        n--;  
11    }  
12    return F;  
13 }
```

F	a	b	n
1	1	1	6
2	1	2	5
3	2	3	4
5	3	5	3
8	5	8	2

Solution itérative

La solution itérative est moins intuitive :

```

1  int Fibo(int n){
2      int a,b,F;
3
4      F=a=b=1;
5
6      while (n>1){
7          F=a+b;
8          a=b;
9          b=F;
10         n--;
11     }
12     return F;
13 }
```

F	a	b	n
1	1	1	6
2	1	2	5
3	2	3	4
5	3	5	3
8	5	8	2
13	8	13	1

Récursivité !!!

Attention

La récursivité n'est pas toujours efficace, même si elle est plus facile à exprimer.

Plan

- 1 Introduction
- 2 Bases du langage C++
- 3 Structures de contrôle
- 4 Fonctions
- 5 Pointeurs**
- 6 Tableaux
- 7 Structures de données (Facultatif)

Plan

5 Pointeurs

■ Introduction

■ Les pointeurs en C++

■ Fonction avec paramètre de type pointeur

Variables et fonctions : *une entente peu cordiale*

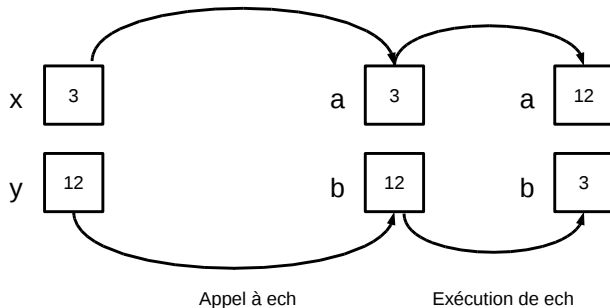
Exemple

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

l'appel à la fonction `echange` sur des variables `x` et `y` n'effectue pas l'effet attendu (échanger les valeurs).

→ paramètres toujours passés par copie dans les fonctions!!!

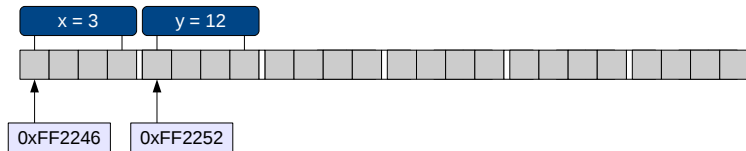
Passage des paramètres par copie

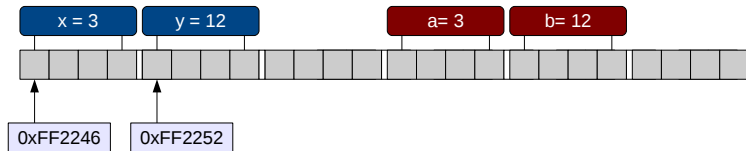
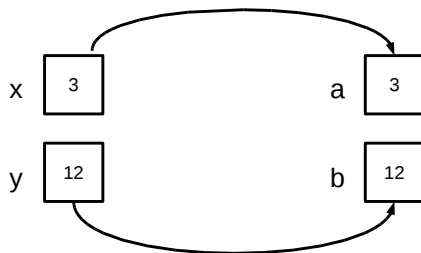


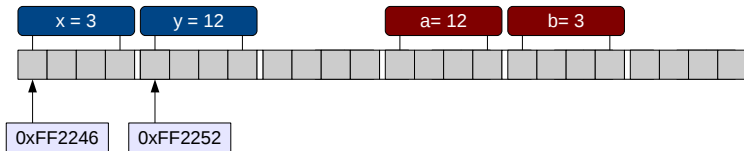
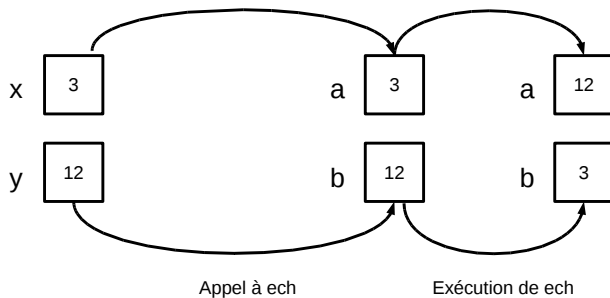
```
int x,y;
x=3;y=12;
echange(x,y);
```


x 3

y 12

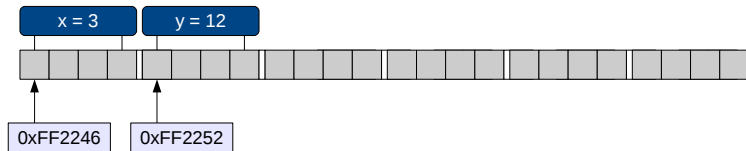
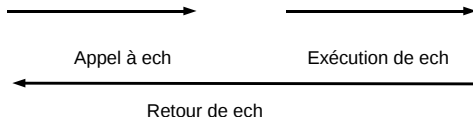






x 3

y 12



Limitations des variables : telles qu'on les connaît

Les variables :

- ne peuvent pas être modifiées par les fonctions
- sont limitées au bloc dans lequel elles ont été définies

Remarque

Ces limitations proviennent de la manipulation des variables par leur identificateur.

Solution : manipuler les variables par leur adresse (*pointeur*)

↪ *dire où se trouve la donnée plutôt que de donner sa valeur !*

Plan

5 Pointeurs

- Introduction

- **Les pointeurs en C++**

- Fonction avec paramètre de type pointeur

Les pointeurs

Definition

Un pointeur est une variable qui contient l'adresse mémoire d'une donnée (une autre variable).

Une variable dite *pointeur* est définie par

- un identificateur : le nom du pointeur
- un type de donnée : pour la donnée pointée

Remarque

Les pointeurs permettent de manipuler des données par leur adresse plutôt que par leur identificateur.

Les variables de type pointeur

Definition

```
type *var;
```

- var est l'identificateur (le nom) du pointeur
- type est le type de donnée de la donnée pointée
- var doit contenir une adresse mémoire valide (pas une valeur)

Exemple

```
int *ptr;
```

définit la variable ptr comme une pointeur sur un entier.

→ ptr **devra donc contenir l'adresse d'une donnée de type int.**

Les variables de type pointeur : Exemple

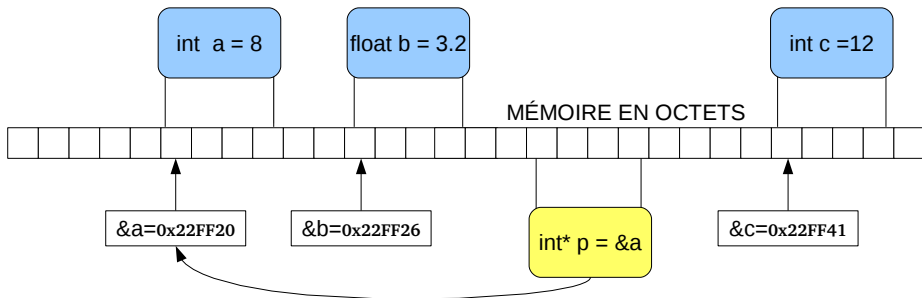
Déclaration de pointeurs :

```
1 int *Aptr;  
2 double *Bptr;  
3 int *Cptr;
```

Affectation de pointeurs :

```
1 int a; double b;  
2  
3 Aptr= &a;    // OK  
4 Bptr= &b;    // OK  
5 Aptr= &b;    // ERREUR  
6 Bptr= &a;    // ERREUR  
7  
8 Aptr= Bptr;  // ERREUR  
9 Aptr= Cptr;  // OK
```

Les variables de type pointeur : Vue mémoire



- `a, b, c` sont des variables normales (elles stockent des valeurs).
- `p` est une variable *dite* pointeur (elle stocke une adresse).
- `p` est de type `int*` et doit stocker l'adresse d'un entier (ici `a`).

Les variables de type pointeur : Affectation

On peut **affecter une variable pointeur** avec :

- l'adresse d'une variable existante *compatible*,
- avec la valeur d'une autre pointeur *compatible*,
- avec le pointeur vide (NULL ou nullptr en C++11),

Exemple

```
int a, *p, *q  
p=&a;  
q=NULL;  
q=p;
```

NULL est défini dans `<iostream>` alors que nullptr nécessite uniquement l'option de compilation `-std=c++11`

Les variables de type pointeur : Accès aux données

Pour accéder à la zone mémoire pointée par un pointeur, il faut utiliser l'opérateur prefixe de déréférencement `*`.

Definition

Soit `var` une variable de type pointeur, le déréférencement `*var` permet d'accéder à la donnée qui est stockée à l'adresse mémoire contenue dans `var` \hookrightarrow on parle de donnée pointée.

```
1 int a;  
2 int *ptr;  
3 ptr=&a; // affecte ptr avec l'adresse de a  
4 *ptr=3; // affecte la zone mémoire d'adresse ptr  
5         // avec la valeur 3 (ici a=3)
```

Manipulation des données avec les pointeurs

Lorsqu'on déréférence une variable pointeur, on peut :

- utiliser la valeur stockée dans la donnée pointée
- modifier la valeur stockée dans la donnée pointée

Exemple

```
int a, *p;  
a=10;p=&a;  
a=*p+1;  
*p=13;
```

Les pointeurs : Exemple 1

```

1 #include <iostream>
2 int main(){
3     int A, *Aptr;
4     A=150;
5     Aptr=&A;
6     std::cout<<" val = "<<A
7         <<" addr= "<<&A<<std::endl;
8     std::cout<<" val = "<<Aptr
9         <<" addr= "<<&Aptr
10        <<" valp= "<<*Aptr<<std::endl;
11    A=99;
12    std::cout<<" val = "<<Aptr
13        <<" addr= "<<&Aptr
14        <<" valp= "<<*Aptr<<std::endl;
15    *Aptr=45;
16    std::cout<<" val = "<<Aptr
17        <<" addr= "<<&Aptr
18        <<" valp= "<<*Aptr<<std::endl;
19    return 0;
20 }
```

Les pointeurs : Exemple 2

Attention

Les pointeurs sont dangereux et cause des erreurs dites de **segmentation** dans l'exécution des programmes.

```
1 int main(){  
2  
3     int A, *Aptr;  
4     A=100;  
5     *Aptr=17; // cette ligne compile correctement  
6               // mais cause une erreur a l'execution  
7     return 0;  
8 }
```

Plan

5 Pointeurs

- Introduction

- Les pointeurs en C++

- Fonction avec paramètre de type pointeur


Fonctions avec pointeurs

Definition

```
type_retour mafonction(type_param * ptr){...}
```

Dans la fonction, c'est une copie du pointeur qui est manipulée et non pas le pointeur.

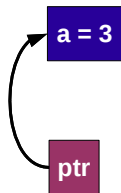
A l'intérieur de la fonction :

- on peut récupérer la donnée pointée : `*ptr`
- on peut modifier la valeur de la donnée pointée : `*ptr=...`
- on ne peut pas modifier le pointeur :  `ptr=...`

Fonctions avec pointeurs : Exemple 1

```
1 #include <iostream>
2
3 void plusUn(int *p){
4     *p= *p+1;
5 }
6
7 int main(){
8     int a; int *ptr;
9     a=3; ptr=&a;
10
11     plusUn(ptr);
12     std::cout<<"a="<<a<<std::endl;
13     plusUn(&a);
14     std::cout<<"a="<<a<<std::endl;
15
16     return 0;
17 }
```

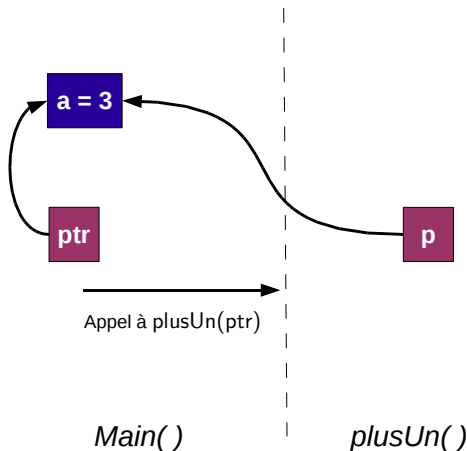
Fonctions avec pointeurs : Exemple 1



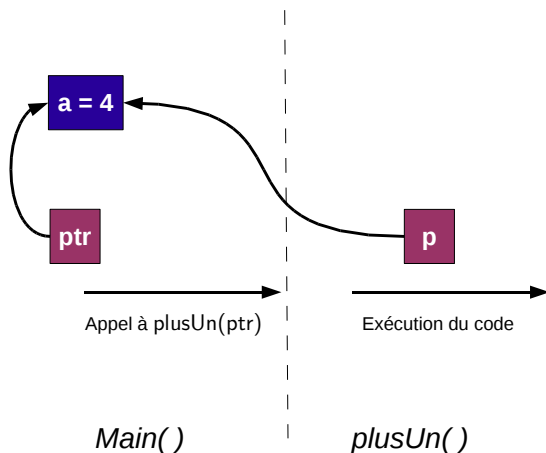
Main()

plusUn()

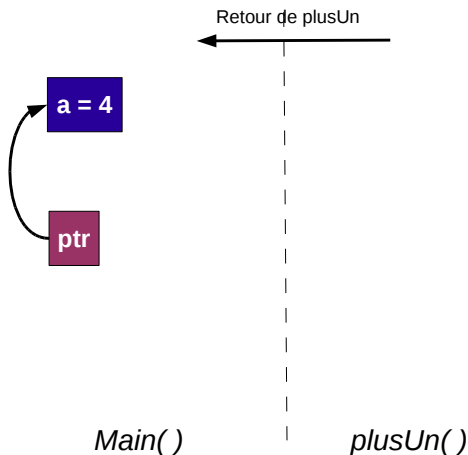
Fonctions avec pointeurs : Exemple 1



Fonctions avec pointeurs : Exemple 1



Fonctions avec pointeurs : Exemple 1



Fonctions avec pointeurs : Exemple 2

```
1 #include <iostream>
2
3 void ech(int *a, int *b){
4     int r;
5     r=*a;
6     *a=*b;
7     *b=r;
8 }
9 int main(){
10     int x,y;
11     x=2;y=3;
12     ech(&x,&y);
13     std::cout<<"x="<<x<<std::endl;
14     std::cout<<"y="<<y<<std::endl;
15     return 0;
16 }
```

Fonction avec pointeur : paramètres résultats

En utilisant les pointeurs, on peut donc interagir dans les deux sens avec une fonction :

- donner des valeurs en entrée dans une fonction
- récupérer **plusieurs** valeurs en sortie d'une fonction

Exemple

```
void minmax(int a, int b, int *min, int *max){  
    if (a>b)  
        {*min=b; *max=a;}  
    else  
        {*min=a; *max=b;}  
}
```


Fonction avec pointeur : Exemple 3

```
1 #include <iostream>
2
3 void minmax(int a, int b, int *min, int *max){
4     if (a>b)
5         { *min=b; *max=a;}
6     else
7         { *min=a; *max=b;}
8 }
9
10 int main(){
11     int a,b,min,max;
12     a=10; b=15;
13     minmax(a,b,&min,&max);
14     std::cout<<"min="<<min<<std::endl
15             <<"max="<<max<<std::endl;
16     return 0;
17 }
```

Fonctions avec pointeurs

L'utilisation classique des pointeurs dans une fonction ne permet pas de modifier la valeur des pointeurs.

Exemple

```
void echPtr(int *p1, int *p2){  
    int *p;  
    p=p1;  
    p1=p2;  
    p2=p;  
}
```

Bien que syntaxiquement correcte, cette fonction n'échangera pas la valeur de p1 et de p2.

Passage de pointeur par copie

Le code suivant n'effectue aucun échange entre a et b car le **passage des paramètres** se fait toujours **par copie**!!!

```
1 void echPtr(int *p1, int *p2){  
2     int *p;  
3     p=p1;  p1=p2;  p2=p;  
4 }  
5 int main(){  
6     int *a,*b;  
7     int c,d;  
8     c=1;  d=3;  
9     a=&c; b=&d;  
10    echPtr(a,b); // pas d'echange entre a et b  
11    return 0;  
12 }
```

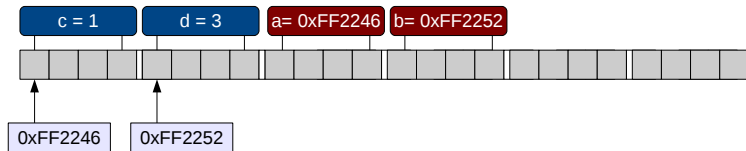
main()

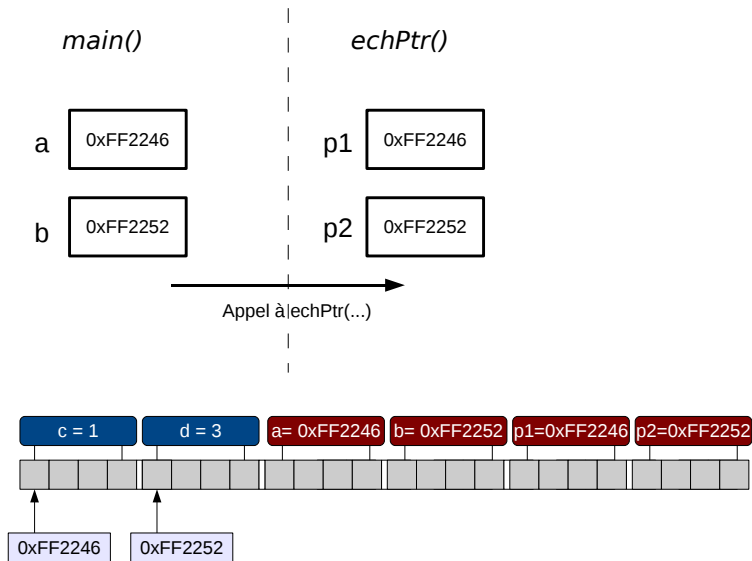
a

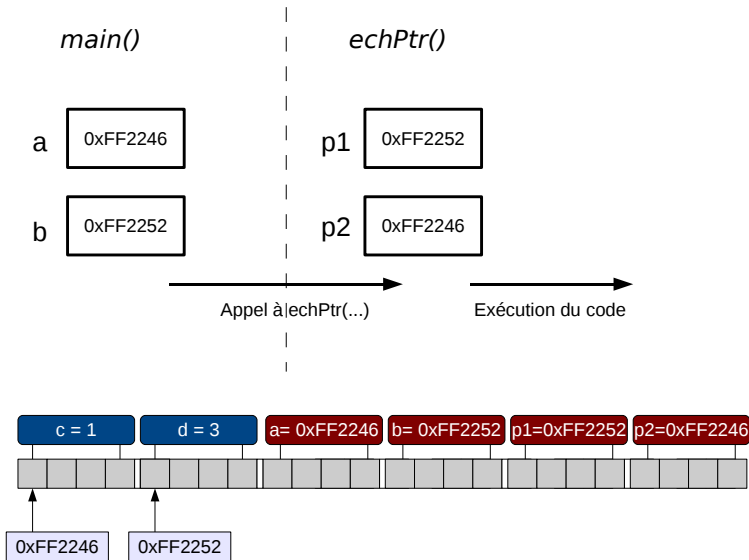
0xFF2246

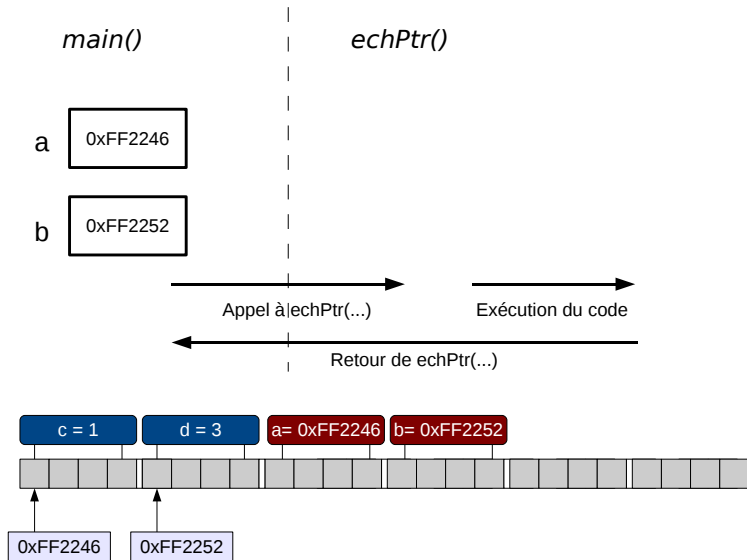
b

0xFF2252

echPtr()








Fonctions avec pointeurs modifiables

Un pointeur étant une variable comme une autre, on peut également utiliser un pointeur pour le manipuler par son adresse.

Definition

```
type **ptr_ptr;
```

La variable `ptr_ptr` est donc une variable qui stocke l'adresse d'un pointeur sur un donnée de type `type`.

↪ *on parle de pointeur de pointeur.*

Exemple

```
int *p;  
int **pp;  
pp=&p;
```


Fonctions avec pointeurs modifiables

Definition

```
type_retour mafonction(type_param **ptr){...}
```

A l'intérieur de la fonction :

- on peut récupérer le pointeur : `*ptr`
- on peut modifier le pointeur : `*ptr=...`
- et bien évidemment récupérer/modifier la donnée pointée par le pointeur : `__(*ptr)`

Fonctions avec pointeurs modifiables

Il faut donc réécrire la fonction `echPtr` comme suit :

Exemple

```
void echPtr(int **p1, int **p2){  
    int *p;  
    p=*p1;  
    *p1=*p2;  
    *p2=p;  
}
```

Fonctions avec pointeurs modifiables : Exemple 1

```
1 void echPtr(int **p1, int **p2){  
2     int *p;  
3     p=*p1;  *p1=*p2; *p2=p;  
4 }  
5 int main(){  
6     int *a,*b;  
7     int c,d;  
8     c=1; d=3;  
9     a=&c; b=&d;  
10    echPtr(&a,&b);  
11    return 0;  
12 }
```

Pointeur de pointeur de ... de pointeur

De manière plus générale, l'imbrication de plusieurs niveaux de pointeurs n'est pas limitée.

Definition

`int $\underbrace{*\dots*}_{n \text{ fois}}$ a ;`

définit la variable `a` comme `n` imbrications de pointeur sur un `int`.

On peut donc stocker dans `a` l'adresse d'une variable étant définie comme imbrication de `n-1` niveaux de pointeur sur un `int`.

Plan

- 1 Introduction
- 2 Bases du langage C++
- 3 Structures de contrôle
- 4 Fonctions
- 5 Pointeurs
- 6 Tableaux**
- 7 Structures de données (Facultatif)

Plan

6 Tableaux

■ Tableaux statiques

■ Fonction et tableaux statiques

■ Tableaux et pointeurs

■ Tableaux dynamiques

■ Fonction et tableaux dynamiques

Manipulation des données

Une seule donnée à la fois (**données scalaires**) :

- une variable : `int a;`
- un pointeur : `int *p;`

Comment faire pour regrouper des données et les manipuler de manière uniforme ?

Manipulation des données

Une seule donnée à la fois (**données scalaires**) :

- une variable : `int a;`
- un pointeur : `int *p;`

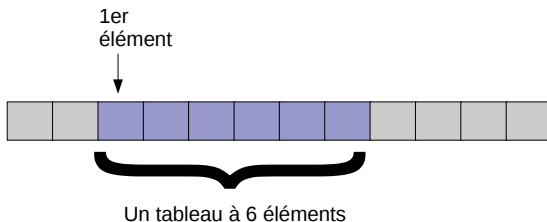
Comment faire pour regrouper des données et les manipuler de manière uniforme ?

utiliser une structure de tableau

La structure de donnée tableau

Definition

Un tableau est un ensemble de données de même type qui sont stockées de manière contiguë en mémoire



- un tableau est identifiable par son 1er élément et sa taille
- les éléments sont accessibles en parcourant la mémoire à partir du 1er élément

Les tableaux en C++

Definition

Un tableau permet de définir avec une seule variable un ensemble de variables de même type de donnée.

Un tableau en C++ est caractérisé par :

- son identificateur (son nom)
- sa taille (le nombre de données)

Les tableaux en C++

Déclaration

```
type T[n];
```

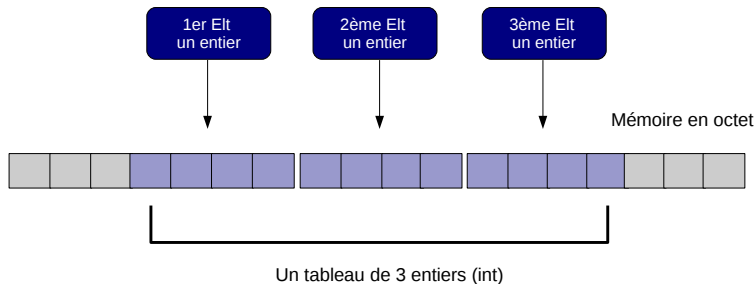
- **T** est l'identificateur du tableau
- **type** est le type de donnée des éléments du tableau
- **n** est une constante entière représentant la taille du tableau

Exemple

```
int T[3];
```

définit la variable T comme un tableau de trois entiers.

Les tableaux en C++ : vue mémoire



Un tableau de n éléments occupe $n \times \text{sizeof}(\text{type})$ octets en mémoire où type est le type de donnée des éléments du tableau.
Ici, un tableau de 3 int occupe 12 octets en mémoire.

Les tableaux en C++

Déclaration

```
type T[n];
```

Cette déclaration de tableau

- crée une variable **T** de type tableau sur **type**
- alloue un espace contigu en mémoire de taille suffisant pour stocker **n** données
- **n'initialise pas** les éléments du tableau

Attention

Il est préférable que la taille **n** du tableau soit une constante connue à la compilation.

Les tableaux en C++ : accès aux éléments

L'accès aux éléments d'un tableau se fait par un calcul d'adresse mémoire à partir du 1er élément : l'opérateur `[]` facilite le calcul.

Definition

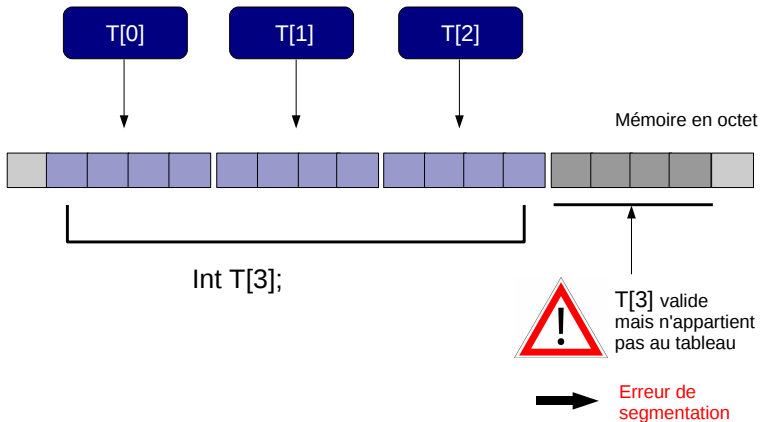
Soit la déclaration : `type T[n];`

L'instruction `T[i]` accède à la $i+1$ -ème case du tableau `T`

Attention :

- les indices du tableau `T` commence à 0 et se termine à $n-1$
- aucune vérification numérique est faite pour la valeur de i
- `T[i]` est un identificateur valide (une variable)
 - ↪ accès en lecture/écriture

Les tableaux en C++ : vue mémoire



Les tableaux en C++ : initialisation des éléments

Deux possibilités :

- lors de la déclaration du tableau
- par initialisation successive des éléments du tableau

Les tableaux en C++ : initialisation des éléments

Déclaration d'un tableau avec initialisation des éléments :

Déclaration

```
type T[n] = { $v_0, v_1, \dots, v_{n-1}$ };
```

Cette déclaration permet de déclarer un tableau T de n variables et d'initialiser les éléments du tableau tels que $T[i] = v_i$.

Exemple

```
int T[3]={4,17,3};
```

Les tableaux en C++ : initialisation des éléments

Déclaration d'un tableau avec initialisation des éléments :

Déclaration

```
type T[n] = { $v_0, v_1, \dots, v_{n-1}$ };
```

Cette déclaration permet de déclarer un tableau T de n variables et d'initialiser les éléments du tableau tels que $T[i] = v_i$.

Exemple

```
int T[3]={4,17,3};
```

Remarque : *la taille n ou toutes les valeurs ne sont pas obligatoires.*

Les tableaux en C++ : initialisation des éléments

Déclaration d'un tableau avec initialisation des éléments :

Plusieurs possibilités :

- `int T[]={1,2,3,4,5};`
↪ tableau de 5 éléments initialisé avec les valeurs données
- `int T[5]={12,13};`
↪ tableau de 5 éléments initialisé avec les valeurs données,
le reste est initialisé à 0

Les tableaux en C++ : initialisation des éléments

On peut initialiser les éléments d'un tableau dans une boucle par affectation successive :

Déclaration

```
int T[10], i;  
for(i=0;i<10;i++)  
    T[i]=...; ← affectation avec la valeur souhaitée
```

Les tableaux en C++ : initialisation des éléments

On peut initialiser les éléments d'un tableau dans une boucle par affectation successive :

Déclaration

```
int T[10], i;  
for(i=0;i<10;i++)  
    T[i]=...; ← affectation avec la valeur souhaitée
```

Remarque : on peut remplacer l'affectation par une saisie
`std::cin>>T[i];`

Les tableaux en C++ : parcours des données

Pour parcourir les données d'un tableau, il suffit d'accéder aux éléments de manière itérative.

Exemple

```
int T[10],i;  
...  
for (i=0;i<10;i++){  
    instructions avec T[i]  
}
```

Attention avec T[i]

il faut toujours garantir que $0 \leq i < \text{taille du tableau}$
le compilateur ne le fait pas pour vous!!!

Les tableaux en C++ : exemple 1

Affichage d'un tableau

```
1 #include <iostream>
2
3 int main(){
4     int T[5]={4,3,1,8,6};
5     int i;
6     for (i=0;i<5;i++){
7         std::cout<<T[i]<<" ";
8     }
9     std::cout<<std::endl;
10    return 0;
11 }
```

Les tableaux en C++ : exemple 2

Calcul du plus petit élément d'un tableau

```
1 #include <iostream>
2 int main(){
3     int T[5]={4,3,1,8,6};
4     int i ,min;
5
6     min=T[0];
7     for ( i=1;i <5;i++){
8         if (T[i]<min)
9             min=T[i];
10    }
11    std::cout<<"le min est: "<<min<<std::endl;
12    return 0;
13 }
```


Les tableaux en C++ : exemple 3

Renverser l'ordre des éléments d'un tableau

```
1 #include <iostream>
2 #define TAILLE 5
3
4 int main(){
5     int T[TAILLE]={4,3,1,8,6};
6     int i,tmp,milieu=TAILLE/2;
7
8     for (i=0;i<milieu;i++){
9         tmp=T[i];
10        T[i]=T[TAILLE-1-i];
11        T[TAILLE-1-i]=tmp;
12    }
13
14    for (i=0;i<TAILLE;i++){
15        std::cout<<T[i]<<" ";
16    }
17    std::cout<<std::endl;
18    return 0;
19 }
```

Copie de tableau

Considérons le programme suivant :

```
1 int main()  
2 {  
3     int t1[5]={1,2,3,4,5};  
4     int t2[5];  
5     t2=t1;  
6     return 0;  
7 }
```

A la compilation, nous avons le message suivant :

error : invalid array assignment

Copie de tableau

Pour recopier un tableau dans un autre, il est obligatoire de passer par des affectations successives :

```
1 int main()  
2 {  
3     int t1[5]={1,2,3,4,5};  
4     int t2[5];  
5     int i;  
6     for (i=0; i<5; i++)  
7         t2[i]=t1[i];  
8     return 0;  
9 }
```

Les tableaux multidimensionnels en C++

Il est possible en C++ de déclarer des tableaux de tableaux. On les appelle des tableaux multidimensionnels.

Exemple

```
int T[3][2];
```

T est un tableau à 3 éléments où chaque élément est un tableau de 2 entiers.

On peut voir T comme une matrice : $T = \begin{bmatrix} * & * \\ * & * \\ * & * \end{bmatrix}$

Les tableaux multidimensionnels en C++

L'enchaînement des opérateurs `[]` permet de récupérer les éléments d'un tableau multidimensionnels

Déclaration

Soit la déclaration `int T[m][n];`

`T[i][j]` donne accès au `j-ème` élément du `i-ème` tableau de `T`

Dans une vue matricielle, cela donne :

$$\text{int } T[3][2] = \begin{bmatrix} T[0][0] & T[0][1] \\ T[1][0] & T[1][1] \\ T[2][0] & T[2][1] \end{bmatrix}$$

Les tableaux multidimensionnels en C++

Comme pour les tableaux unidimensionnels, il est possible d'initialiser les éléments du tableau lors de la déclaration.

Déclaration

```
int T[3][2] = {{1,2},{3,4},{5,6}};
```

Ce qui donne matriciellement :

$$\text{int } T = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$$

Remarque

La notion de tableau multidimensionnel s'étend à plus de dimensions.

Les tableaux multidimensionnels : Exemple

Calcul du déterminant de matrices 2×2

```
1 #include <iostream>
2
3 int main(){
4     int A[2][2]={ {4,3} , {5,1} };
5     int det;
6
7     det = A[0][0]*A[1][1] - A[0][1]*A[1][0];
8
9     std::cout<<"le déterminant est: "<<det<<std::endl;
10
11     return 0;
12 }
```

Plan

6 Tableaux

- Tableaux statiques
- **Fonction et tableaux statiques**
- Tableaux et pointeurs
- Tableaux dynamiques
- Fonction et tableaux dynamiques

Tableau en paramètre d'entrée

Definition

```
type_retour mafonction(type ident[taille])
```

- `type_retour` peut-être `void`, `int`, `float`,...
- `type` est le type commun à tous les éléments du tableau,
- `ident` est le nom du tableau,
- `taille` est une constante entière précisant la taille du tableau.

Tableau en paramètre d'entrée

Definition

```
type_retour mafonction(type ident[taille])
```

Exemple

- `void AfficheTab(int tab[10])`
- `double Somme(double A[20])`
- `double Moyenne(int t[15])`

Tableau en paramètre d'entrée

Afin d'écrire des fonctions pour des tableaux de taille quelconques, il faut

- enlever la taille dans le type du tableau : `type ident[]`
- ajouter un paramètre à la fonction : `int taille`

Definition

```
type_retour mafonction(type ident[], int taille)
```

Attention : Il faut spécifier la taille du tableau car on ne peut pas la récupérer à partir de `ident`.

Tableau en paramètre d'entrée

Exemple

- `void AfficheTab(int tab[], int taille)`
- `double Somme(double A[], int taille)`
- `double Moyenne(int t[], int taille)`

Remarque : Le paramètre `taille` doit être inférieur ou égal à la taille réelle du tableau :

- si égal : tout est normal,
- si inférieur : seul les `taille` premiers éléments sont traités,
- si supérieur : une erreur de segmentation se produira.

Tableau en paramètre d'entrée

Accès aux éléments du tableau

On utilise naturellement l'opérateur `[]`.

L'accès en lecture du $(i+1)$ -ème élément d'un tableau `tab` passé en paramètre s'écrit `tab[i]`.

```
1 void AfficheTab(int tab[], int taille)
2 {
3     for (int i=0;i<taille;i++){
4         std::cout<<tab[i]<<" ";
5     }
6     std::cout<<std::endl;
7 }
```

Exemple : indice du plus petit élément

On cherche le minimum d'un tableau de 10 entiers, mais on renvoie sa position et non sa valeur.

```
1 int MinTab(int t[10])  
2 {  
3     int i, m=0;  
4     for (i=1; i<10; i++)  
5         if (t[i]<t[m])  
6             m=i;  
7     return m;  
8 }
```

Exemple : indice du plus petit élément

On cherche maintenant le plus petit élément d'un tableau de taille quelconque.

```
1 int MinTab(int t[], int taille)
2 {
3     int i, m=0;
4     for (i=1;i<taille;i++)
5         if (t[i]<t[m])
6             m=i;
7     return m;
8 }
```

Tableau en retour de fonction

Attention

Les fonctions ne peuvent retourner qu'une seule donnée (entiers, réels, pointeurs...), il est donc impossible de retourner comme résultat un tableau statique.

Nous verrons dans la suite que par les pointeurs nous pourrons renvoyer des tableaux.

Accès en écriture dans un paramètre tableau

Les paramètres tableaux sont modifiables

L'accès en écriture d'un tableau passé en paramètre d'une fonction se fait avec l'opérateur **[]**.

↔ cela marche comme l'accès en lecture.

```
1 void MaZ(int t[10])  
2 {  
3     int i;  
4     for (i=0; i<10; i++)  
5         t[i]=0;  
6 }
```

Que se passe-t-il ?

```
1 #include <iostream>
2 int main()
3 {
4     int t[10]={3,5,7,2,-5,-7,4,-2,2,1};
5     AfficheTab(t);
6     MaZ(t);
7     AfficheTab(t);
8 }
```

Qu'est-ce qui se passe-t-il ?

```
1 int main()  
2 {  
3     int t[10]={3,5,7,2,-5,-7,4,-2,2,1};  
4     AfficheTab(t);  
5     MaZ(t);  
6     AfficheTab(t);  
7 }
```

Le tableau t a été modifié. Pourquoi ?

Plan

6 Tableaux

- Tableaux statiques
- Fonction et tableaux statiques
- **Tableaux et pointeurs**
- Tableaux dynamiques
- Fonction et tableaux dynamiques

Tableau et pointeurs

Comment peut-on modifier un paramètre d'une fonction ?

En utilisant un **pointeur** sur la variable qu'on veut modifier.

Fonctions et tableaux

Tout se passe donc comme si le paramètre de la fonction n'était pas le tableau, mais un pointeur sur le tableau.

Tableau et pointeurs

Les tableaux et la mémoire :

Quand on déclare un tableau de 10 entiers :

```
int tab[10];
```

on alloue une zone contigue en mémoire de 10 entiers qui sont accessibles via la variable `tab` par les instructions suivantes :

```
tab[0]; tab[1]; ..., tab[9];
```

Mais que contient la variable `tab` ?

Tableau et pointeurs

Les tableaux et la mémoire :

Quand on déclare un tableau de 10 entiers :

```
int tab[10];
```

on alloue une zone contigue en mémoire de 10 entiers qui sont accessibles via la variable `tab` par les instructions suivantes :

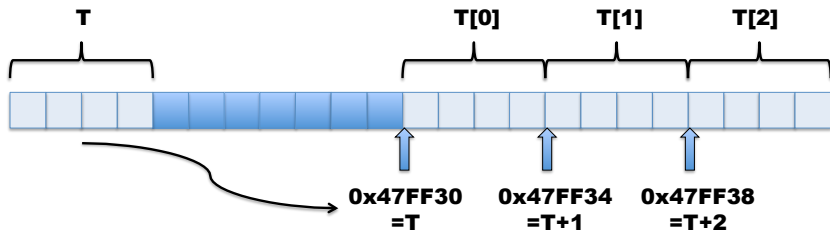
```
tab[0]; tab[1]; ..., tab[9];
```

Mais que contient la variable `tab` ?

Les tableaux sont des pointeurs :

La variable `tab` est en fait un **pointeur sur la première case du tableau**.

Structure d'un tableau



- T contient l'adresse du premier élément du tableau ($\&T[0]$)
- les adresses des autres éléments du tableau se déduisent

Arithmétique de pointeurs

Caractéristique des tableaux en mémoire

- chaque case occupe la même taille,
- les cases sont contiguës.

À partir de l'adresse du premier élément on en déduit les suivantes en y ajoutant le bon nombre d'octets.

Soit la déclaration `int tab[10];` on a :

`adresse(tab[i]) = adresse(tab[0]) + i * sizeof(int)`

Remarque : La manipulation de tableau n'existe pas, le compilateur se charge de tout convertir en pointeurs.

Arithmétique de pointeurs

On peut manipuler explicitement les tableaux comme des pointeurs.

Si `tab` est l'adresse du premier élément, alors l'adresse du i -ème élément est

$$\text{tab} + i$$

et on peut accéder à un élément en faisant :

$$*(\text{tab} + i)$$

Remarque

Quand le compilateur rencontre `tab[i]`, il le réécrit en `*(tab+i)`

Arithmétique de pointeurs

```
1 void AfficheTab(int tab[10])
2 {
3     int i;
4     for (i=0; i<10; i++)
5         std::cout<<*(tab+i)<<" "
6     std::cout<<std::endl;
7 }
```

Ou encore, en n'utilisant que des pointeurs :

```
1 void AfficheTab(int tab[10])
2 {
3     int *p;
4     for (p=tab; p<tab+10; p++)
5         std::cout<<*p<<" "
6     std::cout<<std::endl;
7 }
```

Exemple complet : Tri d'un tableau généré aléatoirement

Une fonction qui modifie un tableau en le remplissant de valeurs aléatoires comprises entre 0 et 20.

```
1 void InitTab(int tab[10])  
2 {  
3     int i;  
4     for (i=0; i<10; i++)  
5         tab[i]=rand() % 21;  
6 }
```

Exemple complet

Une fonction qui trie un tableau de taille quelconque.

```
1 int Tri(int t[], int taille)
2 {
3     int i,j,m;
4     for (i=0;i<taille-1;i++)
5     {
6         m=i;
7         for (j=i;j<taille;j++)
8             if (t[j]<t[m])
9                 m=j;
10        echange(t+i,t+m);
11    }
12 }
```

La fonction echange étant celle du cours précédent.

Exemple complet

L'utilisation de tableau aléatoire permet de tester facilement son algorithme sur de nombreux cas différents.

```
1 #include <iostream>
2 #include <ctime>
3 #include <cstdlib>
4
5 int main()
6 {
7     int t[10];
8     srand(time(NULL));
9     InitTab(t);
10    AfficheTab(t);
11    Tri(t,5);
12    AfficheTab(t);
13    Tri(t,10);
14    AfficheTab(t);
15    return 0;
16 }
```

Exemple complet

A la première exécution, on obtient (par exemple) :

1	7	15	8	12	12	5	6	17	2	1
2	7	8	12	12	15	5	6	17	2	1
3	1	2	5	6	7	8	12	12	15	17

A la seconde, on a :

1	7	5	4	20	14	11	17	1	7	7
2	4	5	7	14	20	11	17	1	7	7
3	1	4	5	7	7	7	11	14	17	20

Plan

6 Tableaux

- Tableaux statiques
- Fonction et tableaux statiques
- Tableaux et pointeurs
- **Tableaux dynamiques**
- Fonction et tableaux dynamiques

Modèle mémoire

Données statiques :

- On connaît **à l'avance** (lors de l'écriture du programme) la taille des données,
- La mémoire peut donc être **réservée dès le début du programme**.

Données dynamiques :

- On connaît la taille des données **lors de l'exécution** du programme
- La réservation de la mémoire doit donc se faire **dynamiquement au cours de l'exécution**.

Remarques

Sur les tableaux statiques :

- ils sont alloués dans un espace mémoire particulier (la pile)
- cette pile est très limitée
- on ne peut pas allouer beaucoup de tableaux ou de grands tableaux

Sur les tableaux dynamiques

- ils sont alloués dans une autre zone mémoire (le tas)
- cet espace mémoire est très vaste
- on peut y allouer beaucoup de grands tableaux

Outils de gestion de mémoire en C++

En C++, la gestion de la mémoire liée aux données

- statiques est gérée automatiquement par le compilateur.
- dynamiques est déléguée à l'utilisateur.

Gestion de la mémoire dynamique :

- **réservation** de l'espace mémoire en fonction d'une taille et d'un type
- **libération** de l'espace mémoire
- **manipulation** de la mémoire uniquement par pointeur

Réservation de la mémoire

On parle plus généralement **d'allocation mémoire**.

Allocation mémoire en C++

On utilise l'instruction

```
type *p = new type[n]
```

- **new** est l'opérateur de réservation de mémoire
- **type** correspond au type de données qu'on va stocker
- **[n]** indique que l'on veut stocker **n** données
- **p** correspond au pointeur permettant l'accès à la mémoire

Réservation de la mémoire

Allocation mémoire en C++

Plus généralement, l'instruction `new type[n]` réclame au système de réserver un espace mémoire permettant de stocker `n` données de type `type`.

- si l'allocation est possible, l'évaluation de cette instruction retournera l'adresse de la mémoire allouée,
- sinon, le programme s'arrêtera en levant une exception.

Exemple

`new int[3]` demande au système de réserver une espace mémoire permettant de stocker 3 `int`.

Réservation de la mémoire

Attention

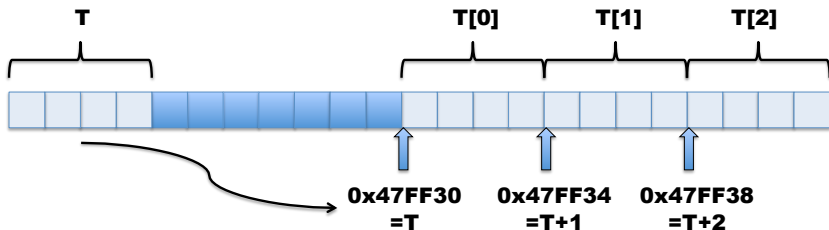
Pour accéder à la mémoire réservée, il est obligatoire qu'une variable pointeur contienne son adresse, ce sera le seul accès possible.

```
int *T = new int[3];
```

Représentation en mémoire

- L'opérateur `new` renvoie l'adresse de la 1ère case de la mémoire dynamique qui a été allouée
- Le pointeur `T` permet d'accéder aux cases de la mémoire (comme un tableau).

```
int *T = new int[3];
```



Utilisation

Une fois qu'une zone mémoire est allouée dynamiquement, on peut la parcourir comme un tableau en utilisant l'opérateur `[]` ou avec l'arithmétique des pointeurs.

```
1 #include <iostream>
2 int main()
3 {
4     float * T;
5     T = new float [40];
6     for (int i=0; i<40; i++)
7         { T[i]=1./(1+i); }
8
9     for (int i=0; i<40; i++)
10        { std::cout<<T[i]<<" "; }
11 return 0;
12 }
```


Exemple

On veut qu'un utilisateur rentre des valeurs numériques dans un tableau de taille N.

```
1 #include <iostream>
2 int main()
3 {
4     int *adr,N;
5     std::cout<<"Entrez le nombre de valeurs :";
6     std::cin>>N;
7     adr= new int[N];
8     for (int i=0;i <N; i++){
9         std::cout<<"Entrez la valeur "<<i<<" : ";
10        std::cin>>adr[i];
11    }
12    return 0;
13 }
```

Libération de la mémoire

Nous sommes capables de réserver de la mémoire. Comme celle-ci est limitée et partagée, il faut donc la libérer.

Libération de mémoire en C++

```
delete[] p;
```

- `delete[]` indique que l'on veut libérer de la mémoire.
- `p` est obligatoirement un pointeur contenant l'adresse de la mémoire à libérer.

Libération de la mémoire

Nous sommes capables de réserver de la mémoire. Comme celle-ci est limitée et partagée, il faut donc la libérer.

Libération de mémoire en C++

```
delete[] p;
```

- `delete[]` indique que l'on veut libérer de la mémoire.
- `p` est obligatoirement un pointeur contenant l'adresse de la mémoire à libérer.

Attention

On ne peut libérer que de la mémoire qui a été allouée dynamiquement par `new`.

Libération de la mémoire

Libération de mémoire en C++

L'évaluation de l'instruction `delete[] p;` libère la mémoire commençant à l'adresse mémoire stockée dans le pointeur `p`.

Remarque

Le pointeur `p` reste utilisable :

```
1  int *p;  
2  p= new int [5];  
3  delete [] p;  
4  p= new int [10];
```

Bonne utilisation de la mémoire dynamique

```
1 #include <iostream>
2 int main()
3 {
4     int * tab;
5     tab = new int [10]; // allocation dynamique
6     ...
7     delete[] tab; // plus besoin du contenu de tab
8     ... // suite du programme sans le contenu de tab
9     return 0;
10 }
```

Initialisation des tableaux dynamiques

On ne peut pas le faire en même temps que l'allocation à l'inverse des tableaux statiques. On le fait donc en deux étapes :

- 1 allocation du tableau
- 2 affectation d'une valeur à chaque case

L'initialisation se fait après le **new** par des affectations successives.

```
1  int * tab, N;  
2  std::cin>>N;  
3  tab = new int [N]; // allocation  
4  
5  for (int i=0;i<N;i++) // initialisations  
6      tab[i]=i;
```

Exemple : extraire les éléments pairs d'un tableau

On veut mettre dans un tableau tous les éléments pairs d'un autre tableau.

```
1  int main(){  
2      int T[40];  
3      ...  
4      int c=0,  
5      for (int i=0;i<40;i++){  
6          if (T[i]%2==0) { c++; }  
7      }  
8      int *tab = new int [c];  
9      int j=0;  
10     for (int i=0;i<40;i++){  
11         if (T[i]%2==0){  
12             tab[j]=T[i];  
13             j++;  
14         }  
15     }  
16     delete [] tab;  
17 }
```

Comment copier des tableaux dynamiques

```
1 #include <iostream>
2 int main()
3 {
4     int i, *t1, *t2;
5     t1=new int [20];
6     t2=new int [20];
7     for (i=0;i<20;i++)
8         { t1[i]=i; }
9     t2=t1;
10    for (i=0;i<20;i++)
11        { std::cout<<t2[i]<<" ";}
12    return 0;
13 }
```

A l'exécution, c'est bien les valeurs de 0 à 19 qui s'affichent mais pourtant le tableau n'a pas été copié.

Copie de tableau

Copie virtuelle

C'est quand deux tableaux partagent la même zone mémoire

↪ $t2=t1$ comme dans notre cas.

Attention : la modification d'un élément de $t1$ implique que celui dans $t2$ l'est aussi.

Copie profonde

On souhaite que les tableaux aient leur propre zone mémoire pour éviter les effets de bords.

↪ il faut allouer une nouvelle zone mémoire et copier les données.

Copie de tableau

La bonne solution pour recopier t1 dans t2 est donc une copie en profondeur.

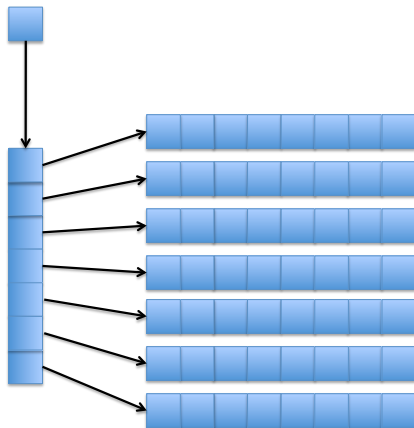
```
1 #include <iostream>
2 int main()
3 {
4     int i, *t1, *t2;
5     t1= new int [20];
6     for (i=0;i<20;i++)
7         t1[i]=i;
8     t2= new int [20];
9     for (i=0;i<20;i++)
10        t2[i]=t1[i];
11
12     delete [] t1;
13     delete [] t2;
14 }
```

Tableaux dynamiques bi-dimensionnels

- On veut utiliser un tableau de taille $N \times M$ entiers
- N et M ne seront connus qu'à l'exécution.
- Comment faire ?

Tableau dynamique 7×8

- Un pointeur sur un tableau de 7 cases
- Chaque case contient un pointeur
- Chacun de ses pointeurs indiquent une zone de 8 cases



Mise en oeuvre

```
1  int main{
2      int **tableau;
3      int i,j;
4
5      tableau= new int*[N];
6      for(i=0;i<N;i++)
7          { tableau[i]=new int[M]; }
8
9      for(i=0;i<N;i++)
10         for(j=0;j<M;j++)
11             { tableau[i][j]=0; }
12     return 0;
13 }
```

Libération de la mémoire

Pour libérer la mémoire, il faut libérer chacune des zones précédemment allouées :

```
1 for ( i=0; i<N; i++)  
2     delete [] tableau [ i ];  
3  
4 delete [] tableau ;
```

Plan

6 Tableaux

- Tableaux statiques
- Fonction et tableaux statiques
- Tableaux et pointeurs
- Tableaux dynamiques
- Fonction et tableaux dynamiques

Tableau en paramètre d'entrée

Pour passer un tableau dynamique en paramètre, on doit passer en argument :

- le pointeur sur la zone mémoire correspondante
- la taille du tableau

Tableau en paramètre d'entrée

Definition

```
type_retour mafonction(type *ident, int taille)
```

- `type_retour` peut-être `void`, `int`, `float`,...
- `type` est le type commun à tous les éléments du tableau,
- `ident` est le nom du tableau,
- `taille` est un paramètre précisant la taille du tableau.

Tout exactement comme pour les tableaux statiques!!!

Tableau en paramètre d'entrée

L'accès en **lecture/écriture** du i-ème élément d'un tableau dynamique `tab` passé en paramètre se fait par l'opérateur `[]`.

```
1 void AfficheTab(int *tab, int n){  
2     for (int i=0;i<n;i++)  
3         std::cout<<tab[i]<<std::endl;  
4 }  
5  
6 void MaZ(int *tab, int n){  
7     for (int i=0;i <n; i++)  
8         tab[i]=0;  
9 }
```

Tableau en paramètre d'entrée

Remarque

On peut également utiliser l'arithmétique des pointeurs.

```
1 void AfficheTab(int *tab, int n){  
2     for (int i=0;i<n;i++)  
3         std::cout<< *(tab+i)<<std::endl;  
4 }  
5  
6 void MaZ(int *tab, int n){  
7     for (int i=0;i <n; i++)  
8         *(tab+i)=0;  
9 }
```

Tableau en paramètre de sortie

Remarque

Les tableaux étant essentiellement une abstraction de la mémoire, il n'existe pas de type tableau que l'on pourrait renvoyer.

⇨ par contre, il est possible de renvoyer l'adresse de la première case du tableau via un pointeur!!!

Fonction permettant de renvoyer un tableau

```
type* mafonction(...) {  
    type *tab=new int[10];  
    ...  
    return tab;  
}
```

Tableau en paramètre de sortie

Fonction permettant de renvoyer un tableau

```
type* mafonction(...) {  
    type *tab=new int[10];  
    ...  
    return tab;  
}
```

Attention

On ne renvoie que l'adresse de la 1ère case du tableau, on ne renvoie pas sa taille.

↪ on doit s'assurer de connaître la taille du tableau, ou de la récupérer par un paramètre.

Tableau en paramètre de sortie

```
1 int* creeTableau(int n)
2 {
3     int i;
4     int *t=new int[n];
5     for (i=0;i<n; i++)
6         { t[i]=0; }
7     return t;
8 }
9 int main()
10 {
11     int taille=20;
12     int * tab = creeTableau(taille);
13     ...
14     delete[] tab; // ATTENTION, on libère la mémoire
15     return 0;
16 }
```

Exemple : extraire les nombres premiers d'un tableau

Exercice

- On suppose avoir la fonction `premier?(int)` qui renvoie *true* si *n* est premier et *false* sinon.
- On souhaite écrire une fonction qui, à partir d'un tableau *T*, va renvoyer un tableau ne contenant que les éléments de *T* qui sont des nombres premiers.

Exemple : extraire les nombres premiers d'un tableau

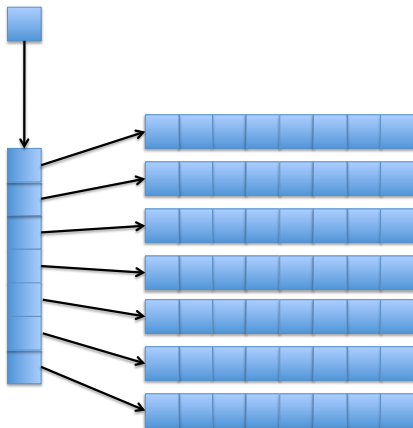
- T : un tableau de dimension n
- nbprime : l'adresse d'un int pour la taille du tableau en sortie

```
1  int* tabPremier(int *T, int n, int *nbprime){
2      *nbprime=0;
3      int *tabp=new int [n];
4      for (int i=0; i<n; i++){
5          if (premier?(t[i])){
6              tabp[*nbprime]=t[i];
7              (*nbprime)++;
8          }
9      }
10     int *prime=new int [*nbprime];
11     for (int i=0; i<*nbprime; i++)
12         prime[i]=tabp[i];
13
14     delete [] tabp;
15     return prime;
16 }
```


Cas de la dimension 2

On suppose avoir choisi comme représentation celle-ci :

- Un pointeur sur un tableau de N cases
- Chaque case contient un pointeur
- Chacun de ses pointeurs indiquent une zone de M cases



Cas de la dimension 2

Pour passer un tableau bi-dimensionnel en paramètre d'une fonction, on utilise **un pointeur de pointeur** et **deux dimensions**.

fonction avec tableau bi-dimensionnel

```
type_retour mafonction(type **T, int N, int M)
```

- `type_retour` peut-être `void`, `int`, `float` *,...
- `type` est le type commun à tous les éléments du tableau,
- `T` est le nom du tableau dynamique,
- `N` est un entier représentant la première dimension.
- `M` est un entier représentant la seconde dimension.

Exemple : pluviométrie

Exercice

Nous allons nous intéresser à l'écriture de différentes fonctions sur des relevés de précipitations.

Ceux-ci seront stockés dans un tableau 2D (mois, jours) où les dimensions pourront ne pas être identiques.

Pour commencer, nous allons commencer par créer un tableau `mois` qui contiendra le nombre de jours du mois correspondant :

```
int mois[12]={31,28,31,30,31,30,31,31,30,31,30,31};
```

Exemple : pluviométrie

Question :

Comment créer un tableau 2D nommé `pluie` tel que la ligne i contienne `mois[i]` cases ?

Exemple : pluviométrie

Question :

Comment créer un tableau 2D nommé *pluie* tel que la ligne *i* contienne *mois[i]* cases ?

```
1 int **pluie;  
2 int mois[12]={31,28,31,30,31,30,31,31,30,31,30,31};  
3 pluie=new int*[12];  
4 for(int i=0;i<12;i++)  
5     pluie[i]=new int[mois[i]];
```

Exemple : pluviométrie

Question :

Calculer le total de précipitations par mois et renvoyer le résultat dans un tableau

Exemple : pluviométrie

Question :

Calculer le total de précipitations par mois et renvoyer le résultat dans un tableau

```
1 int* pluieMois(int **pluie , int mois[12])
2 {
3     int *t=new int [12];
4     for (int i=0;i<12;i++){
5         t[i]=0;
6         for (int j=0;j<mois[i]; j++){
7             t[i]+=pluie[i][j];
8         }
9     }
10    return t;
11 }
```

Exemple : pluviométrie

Question

Afficher le jour de l'année où il pleut le plus

Exemple : pluviométrie

Question

Afficher le jour de l'année où il pleut le plus

```
1 void pluieMax(int **pluie , int mois[12]){  
2     int mi=0,mj=0;  
3     for (int i=0;i<12;i++){  
4         for (int j=0;j<mois[i]; j++){  
5             if (pluie[i][j]>pluie[mi][mj]){  
6                 mi=i;  
7                 mj=j;  
8             }  
9         }  
10    std::cout<<"il pleut le plus le : "  
11        <<mj+1<<"/"<<mi+1<<std::endl;  
12 }
```

Exemple : pluviométrie

Question

Calculer le cumulé de la pluviométrie sur l'année

Exemple : pluviométrie

Question

Calculer le cumulé de la pluviométrie sur l'année

```
1 int pluieAnnuelle(int **pluie , int mois[12])  
2 {  
3     int s=0;  
4     int *t=pluieMois(pluie , mois);  
5     for (int i=0;i<12;i++)  
6         {  
7             s=s+t[i];  
8         }  
9     return s;  
10 }
```

Plan

- 1 Introduction
- 2 Bases du langage C++
- 3 Structures de contrôle
- 4 Fonctions
- 5 Pointeurs
- 6 Tableaux
- 7 Structures de données (Facultatif)**

A quoi sert une structure de données

Les types scalaires :

- type de base : entier (`int`, `double`,...)
- type pointeur : `type *`

Les types composés :

- données homogènes (les tableaux) : `type[]`

A quoi sert une structure de données

Les types scalaires :

- type de base : entier (`int`, `double`,...)
- type pointeur : `type *`

Les types composés :

- données homogènes (les tableaux) : `type[]`
- données hétérogènes : les structures

Les structures de données

- étendent les types du langage pour une problématique donnée :
nombre complexe, matrices, compte bancaire, voiture, ...
- regroupent des données dans une seule variable :
(hétérogène) un compte bancaire = n° banque, n° compte, solde
(homogène) un nbr complexe = partie imaginaire, partie réelle

Les structures de données

- étendent les types du langage pour une problématique donnée :
nombre complexe, matrices, compte bancaire, voiture, ...
- regroupent des données dans une seule variable :
(hétérogène) un compte bancaire = n° banque, n° compte, solde
(homogène) un nbr complexe = partie imaginaire, partie réelle

Remarque

Cela facilite la manipulation des données et la structuration des programmes

Spécifications

Une structure de données est composée d'un nombre fixé de **champs** :

- nommés (banque, compte, solde)
- typés (int pour banque, compte et float pour solde)

Une **variable structurée** peut être manipulée :

- champ par champ (lecture, mise à jour)
- globalement (initialisation, copie, paramètre fonction)

Déclaration d'une structure en C++

Syntaxe

```
struct NomStruct{  
    type1 champ1;  
    type2 champ2;  
    ...  
    typeN champN;  
};
```

Cela déclare un nouveau type de données

- de nom `NomStruct`
- composé des champs `champ1`, ..., `champN`
- ayant respectivement pour type `type1`, ..., `typeN`

Déclaration d'une structure en C++

Syntaxe

```
struct NomStruct{  
    type1 champ1;  
    type2 champ2;  
    ...  
    typeN champN;  
};
```

Attention

- **NomStruct** est un identificateur pas encore utilisé (ex. \neq **int**)
- **type1, ..., typeN** sont des types connus dont on connaît la taille mémoire (ex. \neq **NomStruct**)

Déclaration d'une structure en C++ : Exemple

```
1 struct compteB {  
2     int    banque;  
3     int    compte;  
4     float  solde  
5 };
```

Attention

Le ; est obligatoire après la déclaration de la structure!!!

Déclaration d'une variable structurée

Syntaxe

```
NomStruct var;
```

`var` est une variable de type `NomStruct`.

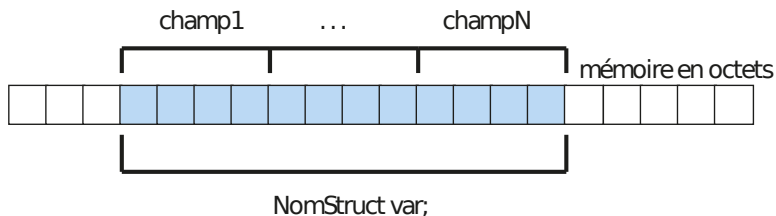
Exemple

```
compteB CB;
```

Une structure en mémoire ???

Syntaxe

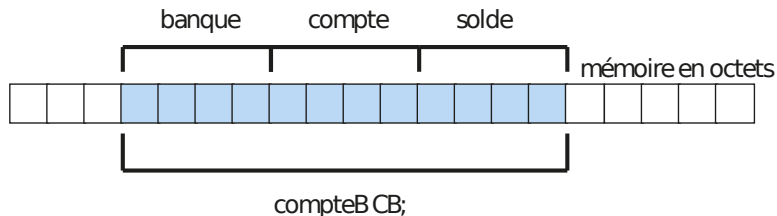
```
NomStruct var;
```



Une structure en mémoire ???

Exemple

```
compteB CB;
```

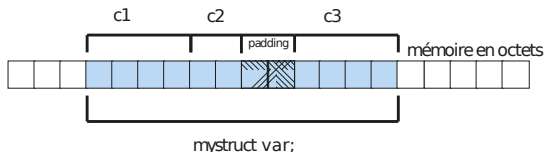


Une structure en mémoire ???

Attention

Les champs des structures sont alignés suivant l'alignement des types de base en mémoire (32 ou 64 bits).

```
struct mystruct {
    int c1;
    short int c2;
    float c3;
};
```



- on peut obtenir l'occupation mémoire en utilisation `sizeof`
exemple : `sizeof(mystruct)` ou `sizeof(var)`
- les règles de padding sont complexes (au delà de ce cours)

Accès aux champs d'une donnée structurée

On utilise la notation pointée :

Syntaxe

```
var.champ
```

Cela permet de récupérer la partie `champ` de la variable `var`.

Accès aux champs d'une donnée structurée

On utilise la notation pointée :

Syntaxe

```
var.champ
```

Cela permet de récupérer la partie `champ` de la variable `var`.

Remarque

Cela définit un identificateur sur la donnée concernée

- possédant une adresse : `&(var.champ)`
- accessible en lecture/écriture : `var.champ=...`

Accès aux champs d'une donnée structurée

```
1 #include <iostream>
2
3 struct compteB{
4     int banque;
5     int compte;
6     float solde;
7 };
8
9 int main(){
10     compteB c;
11     c.solde = 900;
12     std::cout<< " solde : "<<c.solde<<"\n";
13     return 0;
14 }
```

Initialisation d'une structure

Initialisation lors de la déclaration (*comme les tableaux statiques*)

Syntaxe

```
NomStruct var= {exp1,...,expN};
```

équivalent à :

- `var.champ1=exp1;`
- ...
- `var.champN=expN;`

Initialisation d'une structure

Initialisation lors de la déclaration (*comme les tableaux statiques*)

Syntaxe

```
NomStruct var= {exp1,...,expN};
```

Remarque

- l'ordre des expressions est identique à celui des champs
- les champs manquants sont initialisés à 0 ou NULL
- on peut imbriquer les initialisations {..} (e.g. struct, tableau)

Initialisation d'une structure : Exemple

```
1 #include <iostream>
2
3 struct compteB{
4     int banque;
5     int compte;
6     float solde;
7 };
8
9 int main(){
10     compteB c = {123, 456, 900};
11     std::cout<<"le compte : "<<c.compte;
12     std::cout<<" de la banque : "<< c.banque;
13     std::cout<<" a pour solde : "<<c.solde<<"\n";
14     return 0;
15 }
```

Copie des structures

- A l'inverse des tableaux statiques la copie de structure fonctionne via l'opérateur `=`

Remarque

la copie se fait champ par champ

Exemple

```
compteB cb1, cb2;  
cb1=cb2;
```

est équivalent à

```
cb1.banque=cb2.banque;  
cb1.compte=cb2.compte;  
cb1.solde=cb2.solde
```

Copie des structures

Remarque

L'initialisation par recopie fonctionne également

Exemple

```
compteB cb1;  
...  
compteB cb2=cb1;
```


Variable structurée : généralités

- Bien que comprenant plusieurs données, une variable structurée est considérée comme une seule donnée au sens des variables.
 - copie/initialisation possible
 - paramètre/retour de fonctions

Variable structurée : généralités

- Bien que comprenant plusieurs données, une variable structurée est considérée comme une seule donnée au sens des variables.
 - copie/initialisation possible
 - paramètre/retour de fonctions
- A l'inverse des variables tableaux qui correspondent à une adresse mémoire référençant les données.

Plan

- 7 Structures de données (Facultatif)**
 - Structures et fonctions
 - Structures et tableaux
 - Structure et pointeur
 - Structure et allocation dynamique

Passage des structures comme paramètre

Comme tous les paramètres de fonction, les variables structurées sont passées par **copie/valeur**.

Syntaxe

```
type_r mafonction(nomStruct var,...) {...}
```

- **var** est un paramètre de type **nomStruct**
- lors de l'appel, une copie du paramètre structuré sera utilisée.

Passage des structures comme paramètre

Exemple :

```

1 #include <iostream>
2 struct compteB{
3     int banque;
4     int compte;
5     float solde;
6 };
7
8 void interet(compteB cb, float taux){
9     cb.solde *= (1.0+taux);
10 }
11 int main(){
12     compteB c = {123, 456, 900};
13     std::cout<<" a pour solde : "<<c.solde<<"\n";
14     interet(c,0.025); //NE MODIFIE PAS c
15     std::cout<<" a pour solde : "<<c.solde<<"\n";
16     return 0;
17 }
    
```

Retour d'une structure

Comme toutes les variables scalaires, une structure peut être retournée par une fonction.

Syntaxe

```
nomStruct mafonction(...) {  
    ...  
    return exp;  
}
```

- `exp` doit être une variable ou une constante de type `nomStruct`
- c'est une copie de `exp` qui est renvoyée

Retour d'une structure : exemple

```
1 compteB creationCompte(int banque){  
2     compteB c = {banque, rand()%1000, 0.0};  
3     return c;  
4 }
```

- init. : `compteB c = creationCompte(1234);`
- copie : `cb = creationCompte(4321);`

Attention

Dans ce cas d'utilisation, le nombre de copies de la structure est de deux → (coût mémoire potentiellement prohibitif)

Passage de structures par adresse

On utilise un pointeur sur la structure

Syntaxe

```
type_r mafonction(nomStruct *var,...) {...}
```

- `var` contient l'adresse mémoire d'une variable de type `nomStruct`
- aucune copie mémoire lors de l'appel de la fonction

Passage de structures par adresse : Exemple

```
1 void interet(compteB *cb, float taux){  
2     (*cb).solde*=(1.0+taux);  
3 }  
4 int main(){  
5     compteB CB;  
6     interet(&CB, 0.025); //MODIFIE BIEN CB  
7     return 0;  
8 }
```

Attention

La priorité des opérateurs est très importante :

`*var.champ` correspond à **`*(var.champ)`** et non à **`(*var).champ`**

Simplification d'accès aux champs via pointeur

Comme on a souvent besoin de la construction `(*var).champ` le langage C++ propose un opérateur spécial : `->`

Syntaxe

```
nomStruct *var= ...;  
var->champ=...;
```

- strictement équivalent à `(*var).champ=...`;
- les champs sont accessibles en lecture et en écriture (bien sûr)

Exemple

L'affichage d'une structure nécessite souvent une fonction particulière

```
1 void affiche(compteB *CB){  
2     std::cout<<"banque : "<<c->banque  
3         <<" compte : "<<c->compte  
4         <<" solde : "<<c->solde<<"\n";  
5 }  
6 int main(){  
7     compteB cb={1234,0482671234,1270.50};  
8     affiche(&cb); //passage par adresse -> evite la copie  
9     return 0;  
10 }
```

Plan

7 Structures de données (Facultatif)

- Structures et fonctions
- **Structures et tableaux**
- Structure et pointeur
- Structure et allocation dynamique

Champ de type tableau dans une structure

Les champs d'une structure peuvent être des tableaux statiques

Syntaxe

```
struct nomStruct{  
    type1 champ1[10];  
    type2 champ2;  
};
```

- `champ1` contient un tableau de 10 `type1`

Attention

La structure **ne contient pas que l'adresse** du 1er élément du tableau mais bien **l'ensemble des éléments du tableau**.

Exemple

```
1 struct Etudiant {  
2     char nom[32];  
3     int  numero;  
4     int  naissance [3];  
5 };
```

- tous les éléments du tableau sont intégrés dans la structure
taille de la structure Etudiant : 48 octets = 32 char + 4 int
- ils sont accessibles par l'accès au champ puis au tableau

Exemple : `Etudiant etud; etud.naissance[2]=1984;`

Exemple

On peut imbriquer les initialisations lors de la construction d'une donnée structurée :

```
1 struct Etudiant {  
2     char nom[32];  
3     int  numero;  
4     int  naissance[3];  
5 };  
6 ...  
7 Etudiant e={"Beri",2010003111,{12,4,1987}};  
8 ...
```

Tableau dans les structures

Remarque

L'affectation, l'initialisation, le passage en argument et le retour de fonction d'une structure **copie récursivement chacun des champs ...**

Tous les tableaux statiques dans les structures seront donc **copié élément par élément** lors

- de l'affectation de la structure
- du passage en argument d'une fonction
- du retour de la structure par une fonction

Tableau dans les structures : Exemple

```
1 void anniversaire(Etudiant e){
2     std::cout<<e.nom<<" est né le "
3         << e.naissance[0]<<"/"
4         << e.naissance[1]<<"/"
5         << e.naissance[2]<<"\n";
6 }
7
8 int main(){
9     Etudiant e1,e2={"Beri",2010003111,{12,4,1987}};
10    e1=e2; // copie des tableaux
11    anniversaire(e1); // copie des tableaux
12    return 0;
13 }
```

Champs de type structure dans une structure

Les champs d'une structure peuvent être des structures

Syntaxe

```
struct nomStruct{  
    otherStruct champ1;  
    type2 champ2;  
};
```

- `champ1` est un type structuré de type `otherStruct`
- le comportement est similaire aux tableaux statiques (copie)

Champs de type structure dans une structure

Les champs d'une structure peuvent être des structures

Syntaxe

```
struct nomStruct{  
    otherStruct champ1;  
    type2 champ2;  
};
```

- `champ1` est un type structuré de type `otherStruct`
- le comportement est similaire aux tableaux statiques (copie)

Attention

La taille de la structure imbriquée doit être connue ...

→ **les structures récursives sont impossibles**

Exemple

```
1 struct Date {  
2     int    jj ,mm, aaaa ;  
3 };  
4 struct Etudiant {  
5     char  nom[32];  
6     int   numero;  
7     Date  naissance ;  
8 };
```

- tous les champs des structures sont intégrés
taille de la structure Etudiant : 48 octets = 32 char + 4 int
- ils sont accessibles par les appels imbriqués des champs

Exemple : Etudiant e; e.naissance.aaaa=1984;

Exemple

On peut imbriquer les initialisations lors de la construction d'une structure imbriquée :

```
1 struct Date {  
2     int    jj ,mm, aaaa ;  
3 };  
4  
5 struct Etudiant {  
6     char  nom[32];  
7     int   numero;  
8     Date  naissance ;  
9 };  
10 ...  
11 Etudiant e={" Beri " ,2010003111 ,{12,4,1987}};
```

Structure dans les structures

Remarque

L'affectation, l'initialisation, le passage en argument et le retour de fonction d'une structure **copie récursivement chacun des champs ...**

Comme avec les tableaux statiques, les structures imbriquées seront donc **copiées champ par champ** lors

- de l'affectation de la structure
- du passage en argument d'une fonction
- du retour de la structure par une fonction

Structure dans les structures : Exemple

```
1 void anniversaire(Etudiant e){
2     std::cout<<e.nom<<" est né le "
3         << e.naissance.jj<<"/"
4         << e.naissance.mm<<"/"
5         << e.naissance.aa<<"\n";
6 }
7
8 int main(){
9     Etudiant e1,e2={"Beri",2010003111,{12,4,1987}};
10    e1=e2; // copie tableau et structure
11    anniversaire(e2); // copie tableau et structure
12    return 0;
13 }
```

Tableau statique de structures

Les structures étant un type de données, on peut les mettre dans un tableau

Syntaxe

```
nomStruct var[N];
```

- **var** est un tableau contenant **N** données structurées
- **N** doit être une constante connue

Attention

var reste un tableau et contient l'adresse mémoire où se trouvent les **N** données structurées.

Exemple

On peut imbriquer les initialisations lors de la construction d'un tableau statique de structure :

```
1 struct Date {  
2     int    jj ,mm, aaaa ;  
3 };  
4 ...  
5 Date D[3]={ {2,1,1923} , {23,12,2004} , {13,01,2011} };
```

- on ne peut pas copier le tableau par l'opérateur d'affectation :

`Date T[3]=D;` (INTERDIT)

Exemple

On peut imbriquer les initialisations lors de la construction d'un tableau statique de structure :

```
1 struct Date {  
2     int    jj ,mm, aaaa ;  
3 };  
4 ...  
5 Date D[3]={ {2,1,1923} , {23,12,2004} , {13,01,2011} };
```

- on ne peut pas copier le tableau par l'opérateur d'affectation :
`Date T[3]=D;` (INTERDIT)
- lors de passage du tableau en paramètre d'une fonction, les données structurées ne sont pas copiées

Tableaux de structures

```
1 struct Date {  
2     int    jj ,mm, aaaa ;  
3 };  
4 ...  
5 Date D[3]={ {2,1,1923} , {23,12,2004} , {13,01,2011} };
```

- comment accéder au mois de la 2ème date ?
- D[2] est-il passé par copie d'adresse ou par copie de données ?
- D est-il passé par copie d'adresse ou par copie de données ?

Tableaux de structures

```
1 struct Date {  
2     int    jj ,mm, aaaa ;  
3 };  
4 ...  
5 Date D[3]={ {2,1,1923} , {23,12,2004} , {13,01,2011} };
```

- comment accéder au mois de la 2ème date ? `D[1].mm`
- `D[2]` est-il passé par copie d'adresse ou **par copie de données** ?
- `D` est-il passé **par copie d'adresse** ou par copie de données ?

Plan

7 Structures de données (Facultatif)

- Structures et fonctions
- Structures et tableaux
- **Structure et pointeur**
- Structure et allocation dynamique

Champs pointeur dans une structure

Les champs pointeurs dans une structure permettent de stocker l'adresse de n'importe quelle donnée (scalaire, tableau, structure)

Syntaxe

```
struct nomStruct {  
    type1 *champ1;  
    type2 champ2;  
};
```

- `champ1` contient l'adresse d'une donnée de type `type1`
- `type1` est n'importe quel type connu (`nomStruct` compris)

Exemple

```
1 struct Etudiant {  
2     char *nom;  
3     Date anniversaire;  
4     Etudiant *binome;  
5 };  
6 ...  
7 Etudiant e1={"berio",{1.3.1987}}  
8 Etudiant e2={"lefort",{12,1,1988}};  
9 e1.binome=&e2;  
10 e2.binome=&e1;
```

Plan

- 7** Structures de données (Facultatif)
 - Structures et fonctions
 - Structures et tableaux
 - Structure et pointeur
 - Structure et allocation dynamique

Allocation dynamique de structure

Comme pour les types de base (int, double, ...) il est possible de faire de l'allocation dynamique de données structurées.

Remarque

Il suffit d'utiliser l'allocation avec `new` et la libération avec `delete`.

Allocation dynamique de structure

Comme pour les types de base (int, double, ...) il est possible de faire de l'allocation dynamique de données structurées.

Remarque

Il suffit d'utiliser l'allocation avec `new` et la libération avec `delete`.

Exemple

```
struct Date {int jj,mm,aaaa;};  
Date *T= new Date[3];  
...  
delete [] T;
```

Un exemple plus complexe

En cuisine, un plat a un nom. Il est composé d'un nombre quelconque d'ingrédients. On fixera le nombre maximum d'ingrédients à 100. Chaque ingrédient a un nom et un prix. On définit les structures plat et ingrédient.

```
1 struct ingredient {  
2     char nom [32];  
3     float prix;  
4 };  
5 struct plat {  
6     char nom [32];  
7     int nb; // le nombre d'ingrédients connus du plat  
8     ingredient liste [100]; // le tableau des ingrédients  
9  
10 };
```

Un exemple plus complexe

On définit les fonctions suivantes pour les ingrédients :
creerIngredient pour permettre à l'utilisateur de saisir les caractéristiques d'un ingrédient
afficheIngredient pour afficher à l'écran les caractéristiques d'un ingrédient

```
1 void creerIngredient(ingredient * ing){
2     std::cout<<"donnez le nom de l'ingredient \n";
3     std::cin>> (*ing).nom;
4     std::cout<<"donnez le prix de l'ingredient \n";
5     std::cin>>(*ing).prix;
6 }
7
8 void afficheIngredient(ingredient * ing){
9     std::cout<<"nom de l'ingredient : ";
10    std::cout<< ing->nom<<"\n";
11    std::cout<<"prix de l'ingredient : ";
12    std::cout<<ing->prix<<"\n";
13 }
```

Un exemple plus complexe

On définit les fonctions suivantes pour les plats : `creerPlat` pour permettre à l'utilisateur de saisir les caractéristiques d'un plat
`affichePlat` pour afficher à l'écran les caractéristiques d'un plat

```
1 void creerPlat( plat * p){  
2     std::cout<<"donnez le nom du plat \n";  
3     std::cin>> (*p).nom;  
4     (*p).nb = 0;  
5 }  
6 void affichePlat(plat *p){  
7     std::cout<<"nom du plat : ";  
8     std::cout<< p->nom<<"\n";  
9     int n =p->nb;int i;  
10    std::cout<<"nb d'ingrédients : "<<n<<"\n";  
11    for(i=0;i<n;i++)  
12        afficheIngredient(&(p->liste)[i]);  
13 }
```

Un exemple plus complexe

ajouteIngredient pour ajouter un ingrédient à un plat

```
1 void ajouteIngredient(ingredient * ing, plat * p){  
2     if (p-> nb < 100){  
3         (p->nb)++;  
4         // on ajoute ing  
5         (p->liste)[p->nb -1]=*ing;  
6     }  
7 }
```

Un exemple plus complexe

et enfin le programme :

```
1 int main(){  
2     ingredient i1={ "farine", 34}, i2= {"oeufs", 5};  
3     plat p;  
4     creerPlat(&p);  
5     ajouteIngredient(&i1, &p);  
6     ajouteIngredient(&i2, &p);  
7     affichePlat(&p);  
8     return 0;  
9 }
```