



1

# C++

UE M2 MIAGE IF (Université Paris Dauphine)

[julien.lesca@dauphine.fr](mailto:julien.lesca@dauphine.fr)
















# UE C++

- Transparents de cours inspirés du livre « Apprendre le C++ » de Claude Delannoy
- 8 séances de 3h de TP
- Note finale: 100% projet



# Language C++

- Conçu à partir de 1982 par Bjarne Stroustrup
  - Extension du langage C
  - Ajoute les outils de Programmation Orientée Objet (POO)
- Classé second au classement 2018 de l'IEEE

				Web Mobile Enterprise Embedded	
Language Rank	Types	Spectrum Ranking		Spectrum Ranking	
1. Python	  	100.0		100.0	
2. C++	  	98.4		99.7	
3. C	  	98.2		99.4	
4. Java	  	97.5		97.3	
5. C#	  	89.8		88.7	

# POO

- Programmation structurée
  - Un programme est formé de procédures et de structures de données
- Programmation Orientée objet
  - Fondée sur le concept d'objet
    - Association des données et des procédures agissant sur ces données

# Premier programme



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

main(){
    int i = 0;
    cout << "Hello world ";
    cout << i;
}
```

# Différents éléments

- La ligne « *main()* » se nomme l'en-tête
  - Programme principale délimité par des accolades {}
- L'instruction « *int i = 0;* » est une déclaration
  - La variable *i* est de type *int* (entier)
  - Doit apparaitre avant l'utilisation de la variable
- Les instructions « *cout << "Hello world ";* » et « *cout << i;* »
  - Affichage à l'écran d'une chaîne de caractères et d'une valeur de variable
  - Chaîne de caractères délimitées par des guillemets

# Différents éléments

- La ligne « `#include<iostream>` » est une directive au préprocesseur
  - Prise en compte avant la traduction du programme
  - Doit être écrite à raison d'une par ligne et commencer en début de ligne
    - Préférable de les placer en début de programme
  - Demande d'introduire des instructions du fichier `iostream.h`
    - Déclarations relatives à `cout` et `<<`
- L'instruction « `using namespace std;` » précise l'espace de nom
  - Les espaces de noms permettent de restreindre la portée des symboles
  - Les symboles de `iostream` appartiennent à l'espace de nom `std`



# Règles d'écritures

- Les identificateurs servent à désigner les différentes « choses »
  - Variables, fonctions, ...
  - Formés d'une suite de caractères
    - Chiffres et lettres (incluant '\_')
    - Commence par une lettre
    - Majuscule et minuscules autorisées
- Certains mots-clés sont réservés
  - *int, char, float, ...*





# Commentaires

- Commentaire: texte explicatif qui n'a pas d'incidence sur la compilation
  - Commentaire libre
    - Placé entre les symboles /\* et \*/
  - Commentaire de fin de ligne
    - Placé en fin de ligne après le symbole //

# Premier programme bis

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

main(){
    /* ce programme effectue un affichage à l'écran*/
    int i = 0;
    cout << "Hello world "; //une chaine de caractère est affichée
    cout << i; //la valeur d'une variable est affichée
}
```

# Création d'un programme

- Edition du programme: écrire le texte d'un programme source
  - Appelé le fichier source qui porte l'extension `.cpp`
- Compilation: traduction du programme source en langage machine
  - Préprocesseur: exécute les directives qui le concerne (précédées par #)
    - Inclut les fichiers d'en-tête d'extension `.h`
    - Produit un programme source pur
  - Compilation: traduction en langage machine
    - Laisse certains identificateurs non définis
    - Le résultat est un module objet d'extension `.o`
- Edition des liens: fait le lien entre les identifiants et leurs définitions
  - Va chercher dans la bibliothèque standard les modules objets nécessaires
  - Le résultat est un programme exécutable

# Expression et opérateurs

Types de base

Opérateurs

Priorités

# Types de base

- Nombres entiers
  - *short, int, long*
- Nombres flottants
  - *float, double, long double*
- Caractères
  - *char*
- Booléens
  - *bool*

# Entiers

- Trois tailles différentes d'entiers
  - Dépend de la machine utilisée
    - Tous les *int* n'ont pas la même taille sur toutes les machines
  - Constantes *INT\_MAX* et *INT\_MIN* du fichier d'en tête *climits*
    - Si on ajoute 1 au plus grand nombre entier, on obtient le plus petit nombre entier

# Flottants

- Représentent, de manière approchée, une partie des nombres réels
- Représentation sous la forme  $M \times B^E$  en base  $B$ 
  - $M$  est la mantisse,  $E$  est l'exposant
    - Base unique pour une machine donnée (souvent 2 ou 16)
- La notation des constantes doit comporter un point
  - Exemple: 12.43 ou 4. ou .27
  - Utilise la lettre e (ou E) pour introduire un exposant puissance 10
    - Exemple: 4.25e4 ou 42.5E3 ou 48e13





# Caractères

- Caractères codés sur un octet
  - Code employé dépend de l'environnement de programmation
    - ASCII, UTF-8
  - Les constantes se notent entre apostrophes
    - Exemple: 'a', 'Y', '+'
  - Certains caractères possèdent une représentation utilisant le caractère \
    - \n (saut de ligne), \t (tabulation)
    - \\, \', \", \?

# Booléens

- Formés de deux valeurs notées *true* et *false*
- N'existe pas en C
  - La valeur correspondant à *false* est 0
  - La valeur correspondant à *true* est 1 ou toute autre valeur différente de 0

# Initialisation et constantes

- Initialisation d'une variable lors de sa déclaration
  - `int n = 5;`
- Peut déclarer que la valeur d'une variable ne doit pas changer
  - `const int N = 5;`
  - Les instructions modifiant la valeur d'une constante seront rejetées

# Opérateurs et expressions

- Expression formée à l'aide d'opérateur
  - Une expression possède une valeur
- En C++, la notion d'expression et d'instruction sont étroitement liés
  - La principale instruction est une expression suivie de ;
    - Exemple:  $i = 5$ ;

# Opérateurs arithmétiques

## ➤ Opérateurs binaires

➤ + (addition), - (soustraction), / (division), \* (multiplication)

➤ L'opérateur % (modulo) porte sur des entiers

➤ Renvoie le reste de la division du premier opérande par le second

➤ Le quotient de deux entiers (flottants) fournit un entier (flottant)



➤  $5/2 = 2$  et  $5./2. = 2.5$

## ➤ Opérateur unaire

➤ - (opposé)

# Priorités relatives

- Une expression peut contenir plusieurs opérandes
  - Nécessaire de savoir dans quel ordre ils sont appliqués
- Règle de l'algèbre traditionnel
  - \* et / plus prioritaires que + et -
  - Opérateur unaire plus prioritaire que les opérateurs binaires
  - Priorité de gauche à droite en cas de priorité identiques
  - Les parenthèses permettent d'outrepasser les règles de priorité

# Conversion implicite

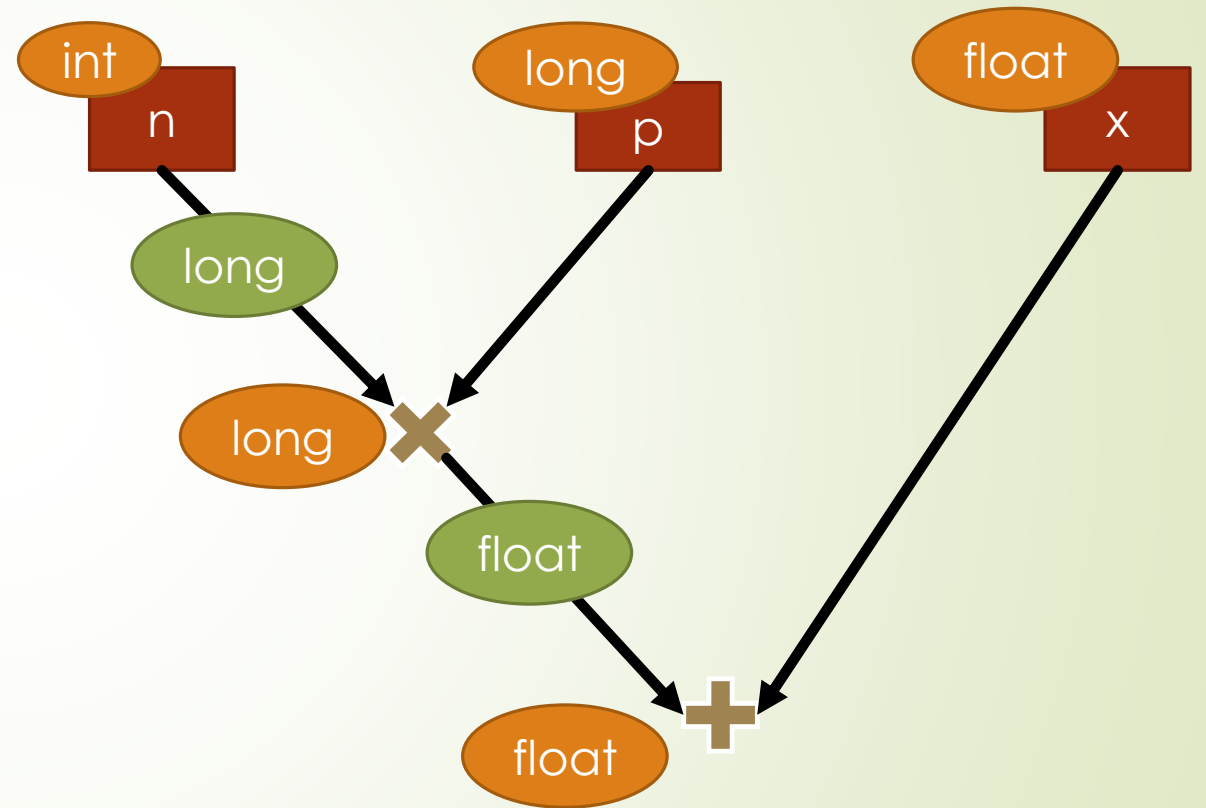
- Les opérandes binaires sont définis pour deux opérandes du même type
- Peut utiliser des expressions mixtes avec des opérandes de types différents
  - Une conversion de type est effectuée avant d'appliquer l'opérateur
- Une conversion d'ajustement de type ne peut se faire que suivant une hiérarchie qui permet de ne pas dénaturer la valeur





# Exemple

- Expression  $n * p + x$ 
  - $n$  int,  $p$  long et  $x$  float

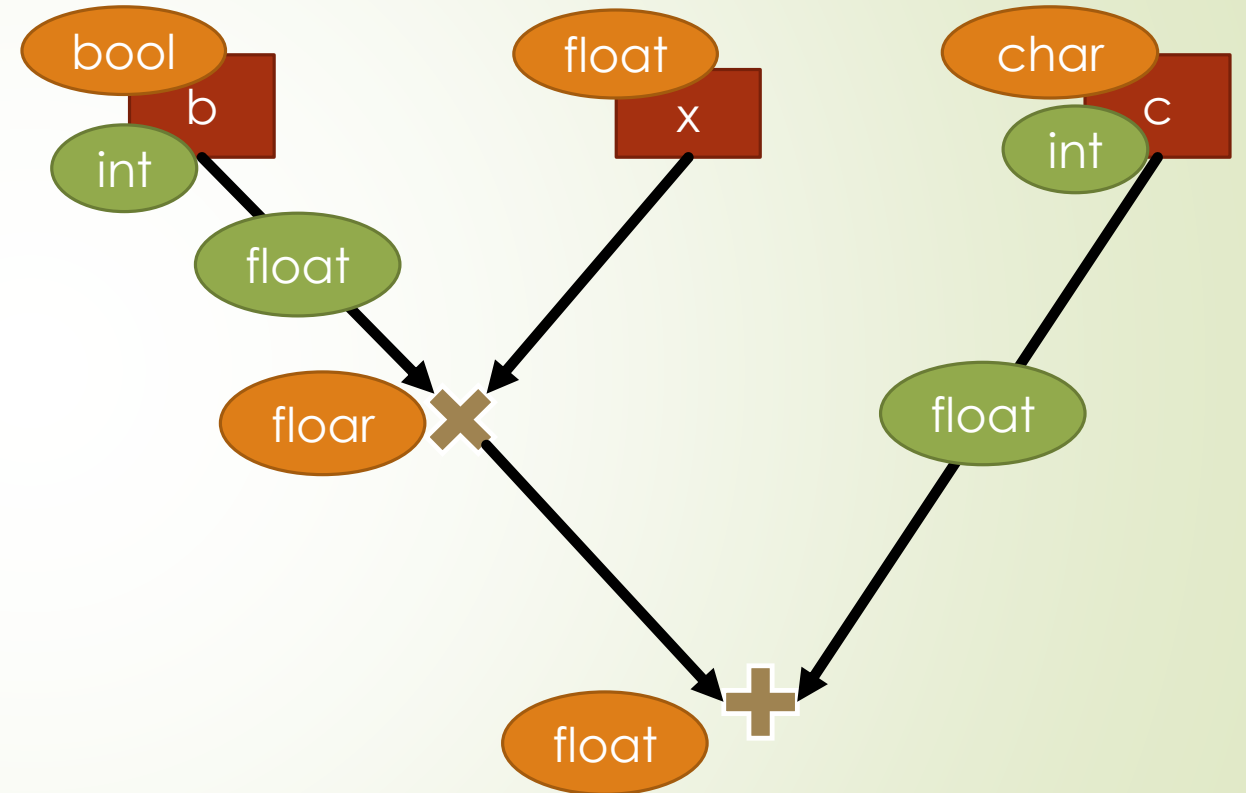


# Promotion numérique

- Les opérateurs arithmétiques ne sont pas définis pour certains types
  - *short*, *char* et *bool*
  - D'abord converti en *int* sans considérer les types des autres opérandes
    - La valeur pour *char* est le code du caractère
    - La valeur pour *bool* est 0 pour *false* et 1 pour *true*

# Exemple

- Expression  $b*x+c$ 
  - $b$  bool,  $x$  float et  $c$  char



# Opérateurs relationnels

- ▶ Permet de comparer des expressions
  - ▶  $<$  (inférieur),  $\leq$  (inférieur ou égal)
  - ▶  $>$  (supérieur),  $\geq$  (supérieur ou égal)
  - ▶  $==$  (égal à),  $!=$  (différent de)
- ▶ Résultat de la comparaison est un *bool*
  - ▶ Exemple:  $x+y > a+2$
- ▶ Opérandes doivent être du même type
  - ▶ Utilise les règles de conversions si ce n'est pas le cas

# Opérateurs logiques

- Opérateurs binaires
  - `&&` (et logique), `||` (ou logique)
  - Le second opérande n'est évalué que si c'est nécessaire
    - `false && (x+1)`
      - `x+1` n'est pas évalué car le premier opérande impose le résultat à *false*
- Opérateur unaire
  - `!` (non logique)
- Acceptent n'importe quel opérande numérique
  - 0 correspond à *false* et toute valeur différente de 0 correspond à *true*

# L'opérateur d'affectation

- Affectation du type *lvalue = expression*
  - Affecte la valeur de l'expression à la lvalue
  - Expression dont la valeur est celle de la lvalue après affectation
  - Priorité de droite à gauche
    - Exemple:  $i = j = 3$
- ⚠ ➤ Une lvalue est une valeur qui peut apparaître à gauche d'une affectation
  - Référence à un emplacement mémoire dont on pourra modifier la valeur
    - Exemple: variable

# Conversion

- Peut fournir à l'opérateur d'affectation des opérandes de types différents
- Le type de la lvalue ne peut pas être converti
- ⚠ ➤ Conversion systématique du type de l'expression dans celui de la lvalue
  - Ne respecte pas nécessairement la hiérarchie des types
  - Peut conduire à une dégradation de l'information



# Opérateurs d'incrémentations

- Opérateur unaire s'appliquant à une lvalue
  - `++i` incrémente la valeur de `i` et prend la valeur de `i` après incrémentation
  - `i++` incrémente la valeur de `i` et prend la valeur de `i` avant incrémentation
- Il existe également un opérateur de décrémentation
  - `--i` décrémente la valeur de `i` et prend la valeur de `i` après décrémentation
  - `i--` décrémente la valeur de `i` et prend la valeur de `i` avant décrémentation

# Opérateur d'affectation élargies

- Peut remplacer  $i = i+k$  par  $i += k$
- L'expressions ***lvalue = lvalue opérande expression*** peut être remplacé par l'expression ***lvalue opérande = expression***
  - $+=, -=, *=, /=, \%=$

# Opérateur de cast

- ▶ Permet de forcer la conversion d'une expression dans un type donné
  - ▶ Exemple: *(double) (n/p)*
  - ▶ Opérateur unaire
- ! ▶ Conversion du résultat de l'expression et non celle des valeurs intermédiaires
  - ▶ Dans l'exemple, la division se fera en fonction du type de n et p, et la conversion en double se fera sur le résultat
- ▶ Il existe autant d'opérateur de cast que de types différents

# Opérateur conditionnel

- Opérateur ternaire
  - ***expression1 ? expression2 : expression 3***
  - Évalue *expression1* qui joue le rôle d'une condition
    - Si *expression1* est différent de 0 alors prend la valeur d'*expression2*
    - Sinon alors prend la valeur d'*expression3*
  - Exemple:  $x > 0 ? x : -x$

# Récapitulatif et priorités

Plus prioritaire

↓

Moins prioritaire

Catégorie	Opérateurs	Priorité
Unaire	+ - ++ -- ! (cast)	←
Arithmétique	* / %	→
Arithmétique	+ -	→
Relationnels	< <= > >=	→
Relationnels	== !=	→
Logique	&&	→
Logique		→
Conditionnel	?:	→
Affectation	= += -= *= /= %=	←



Quizz

# Entrées/sorties de base

Affichage à l'écran

Lecture au clavier

# Affichage à l'écran

- L'opérateur binaire << possède deux opérandes
  - Celui de gauche qui correspond à un flux de sortie
    - Par exemple, `cout` pour l'affichage à l'écran
  - Celui de droite qui est une expression
- Cet opérateur possède une priorité de gauche à droite
  - Le résultat fournit par l'opérateur << est le flux après avoir reçu l'information
    - Exemple: `cout << "Hello world " << i;`
      - Affiche « Hello world » suivi de la valeur de `i`



- Nécessaire d'inclure `<iostream>` et d'utiliser l'espace de nom `std`



# Possibilités d'écriture

- L'opérateur << permet d'afficher une expression de tout type de base
  - *char*: afficher le caractère correspondant
  - Entier: *short*, *int* ou *long*
  - Flottant: *float*, *double* et *long double*
  - *bool*: affiche 0 si faux et 1 si vrai
  - Chaîne de caractères

# Lecture au clavier

- L'opérateur binaire >> possède deux opérandes
  - Celui de gauche qui correspond à un flux d'entrée
    - Par exemple, *cin* qui correspond au clavier
  - Celui de droite qui est une lvalue
- Cet opérateur possède une priorité de gauche à droite
  - Le résultat fourni par l'opérateur >> est le flux après avoir extrait l'information
    - Exemple: *cin >> i >> j;*
      - Lit deux valeurs et met la première dans *i* et la seconde dans *j*



- Nécessaire d'inclure *<iostream>* et d'utiliser l'espace de nom *std*

# Possibilités de lecture

- L'opérateur >> permet d'accéder à des informations de tout type de base
  - Caractère: *char*
  - Entier: *short, double* ou *long*
  - Flottant: *float, double* ou *long double*
  - *bool*: seul les valeur 0 (faux) et 1 (vrai) acceptées


# Tampon et caractères séparateurs

- L'utilisateur fournit une suite de caractères qu'il valide avec « entrée »
  - Rangée provisoirement dans un espace mémoire appelé tampon
  - Le tampon est exploré caractère par caractère par l'opérateur >>
    - Si une partie n'est pas exploitée, elle reste disponible pour les prochaines lectures
    - Si les informations du tampon ne suffisent pas, l'opérateur attendra que l'utilisateur fournisse une nouvelle ligne
- Certains caractères dits séparateurs jouent un rôle particulier
  - ' ' (l'espace), '\n' (fin de ligne), '\t' (tabulation)

# Règle de lecture tampon


- Avance le pointeur jusqu'au premier caractère différent d'un séparateur
  - Prend en compte tous les caractères suivants jusqu'au prochain séparateur
    - Place le pointeur sur ce séparateur
  - S'arrête en cas de caractère invalide
    - Soit on a pu fabriquer une valeur pour la lvalue
      - La lecture suivante continu à partir du caractère invalide
    - Soit on n'a pas pu fabriquer une valeur pour la lvalue
      - La valeur de la lvalue reste inchangée
      - Le flot est bloqué et toute tentative de lecture ultérieur échouera

# Example

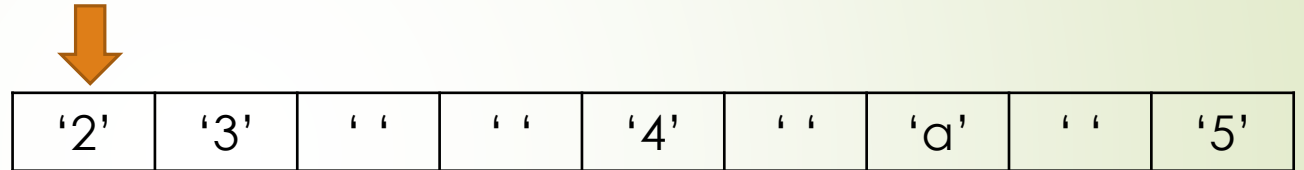


```
int a, b, c;  
cin >> a;  
cin >> b;  
cin >> c;
```

# Example



```
int a, b, c;  
cin >> a;  
cin >> b;  
cin >> c;
```





# Example

*a* = 23



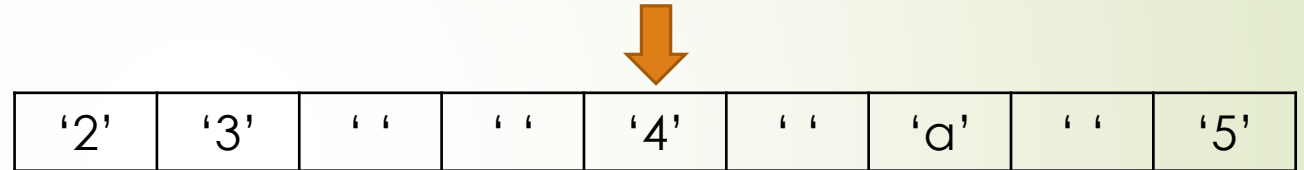
```
int a, b, c;  
cin >> a;  
cin >> b;  
cin >> c;
```



'2'	'3'	' '	' '	'4'	' '	'a'	' '	'5'
-----	-----	-----	-----	-----	-----	-----	-----	-----

# Example

*int* a, b, c;  
*cin* >> a;  
→ *cin* >> b;  
*cin* >> c;

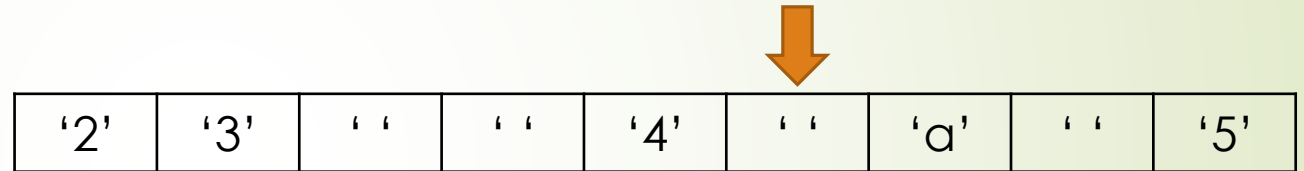


# Example

*b = 4*

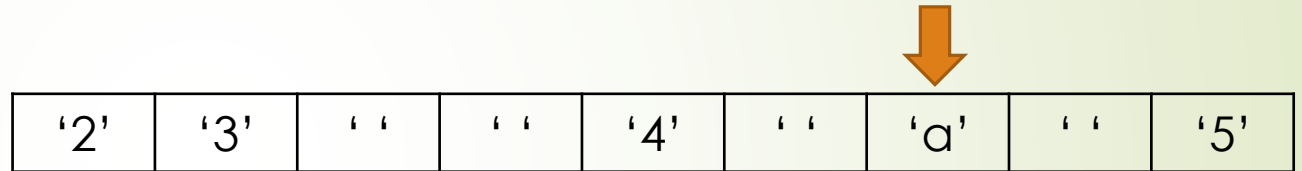


```
int a, b, c;  
cin >> a;  
cin >> b;  
cin >> c;
```



# Example

```
int a, b, c;  
cin >> a;  
cin >> b;  
→ cin >> c;
```



# Example

```
int a, b, c;  
cin >> a;  
cin >> b;  
→ cin >> c;
```

valeur incorrecte  
cin bloqué



'2'	'3'	' '	' '	'4'	' '	'a'	' '	'5'
-----	-----	-----	-----	-----	-----	-----	-----	-----

# Instructions de contrôle

Conditionnelles

Boucles

# Instructions de contrôle

- Les instructions d'un programme sont exécutées séquentiellement
- Les instructions de contrôle permettent d'effectuer
  - Des choix et de se comporter différemment suivant les circonstances
  - Des boucles qui répètent plusieurs fois un ensemble d'instruction



# Blocs d'instructions

- Différents types d'instructions
  - Simples (terminées par un point virgule)
  - Structurées (choix et boucles)
  - Blocs
    - Suite d'instructions placées entre accolades { }
    - Instructions de types quelconques

# Instruction if...else

- Syntaxe de l'instruction

***if (expression)***

***instruction\_1***

***else***

***instruction\_2***

- *expression* est une expression quelconque qui sera convertie en *bool*
  - Si *expression* est évalué à *false* (vaut 0) alors *instruction\_2* est exécutée
  - Sinon (*expression* ne vaut pas 0) *instruction\_1* est exécutée
- *Instruction\_1* et *instruction\_2* sont des instructions quelconques

# Exemple

```
main(){  
    int i;  
    cout << "Entrez une valeur: ";  
    cin >> i;  
    if(i >= 0)  
        cout << "Cette valeur est positive \n ";  
    else{  
        cout << "Cette valeur est négative \n ";  
    }  
}
```

# Imbrication des instructions if

➤ Le *else* est facultatif

➤ Son absence équivaut à une *instruction\_2* vide



➤ Un *else* se rapporte au dernier *if* du bloc dont le *else* n'a pas encore été attribué

```
if (a<=b)
    if (b<=c)
        cout << "ordonné";
    else
        cout << "non ordonné";
```

Ici, le *else* est associé  
au second *if*



➤ Pour lever l'ambiguïté: utilisez des blocs pour *instruction\_1* et *instruction\_2*

# L'instruction switch: exemple

```
int n;  
cout << "Entrez une valeur: ";  
cin >> n;  
switch (n){  
    case 0: cout << "nul\n";  
            break;  
    case 1: cout << "un\n";  
            break;  
    case 2: cout << "deux\n";  
}  

```

# L'instruction switch

- Syntaxe de l'instruction

```
switch(expression){  
    case constante_1: [suite_instructions_1]  
    case constante_2: [suite_instructions_2]  
    ...  
    case constante_n: [suite_instructions_n]  
    [default: suite_instructions_default]  
}
```

- *expression*: expression entière classique



- *constante*: expression constante d'un type entier quelconque (*char* accepté)
- *suite\_instructions*: séquence d'instructions quelconques

L'instruction *default*  
est optionnelle



# L'instruction switch

- La valeur de *expression* est évalué par le *switch*
  - Si cette valeur correspond à *constante\_i*
    - L'exécution commence à partir de *suite\_instructions\_i*
    - ⚠ ➤ Elle continue jusqu'à la fin du bloc ou au premier *break* rencontré
  - Si cette valeur ne correspond à aucune constante
    - L'exécution commence à partir de *default* si il est présent
      - Elle s'arrête à la fin du bloc ou au premier *break* rencontré
    - Si *default* n'est pas présent, on passe à l'instruction qui suit le bloc



## L'instruction switch: exemple 2

```
int n;  
cout << "Entrez une valeur: ";  
cin << n;  
switch (n){  
    case 0: cout << "nul\n";  
    case 1: cout << "un\n";  
            break;  
    case 2: cout << "deux\n";  
}
```

Ici, si  $n$  est égal à 0 alors les instructions `cout << "nul\n";` et `cout << "un\n";` sont exécutées avant de sortir du block (après l'instruction `break`)



# L'instruction do...while

- Syntaxe de l'instruction

***do instruction***

***while (expression);***

- *instruction* est une instruction quelconque (simple, structurée ou bloc)
- *expression* est une expression quelconque
  - Répète *instruction* tant que *expression* est évalué vrai
    - La sortie de boucle ne se fait qu'après l'exécution intégrale de *instruction*
  - *instruction* exécuté au moins une fois (avant la première évaluation de *expression*)

# Exemple

```
int n;  
do{  
    cout << "donnez un nombre strictement positif: ";  
    cin >> n;  
    cout << "\n";  
}while (n <= 0);  
cout << "le nombre choisi est " << n;
```

# L'instruction while

- Syntaxe de l'instruction

***while (expression)***

***instruction***

- Similaire à *do...while*

- Excepté que *expression* est évalué avant l'exécution d'*instruction*

- *instruction* peut ne pas être exécuté

# Exemple

```
int n;  
cout << " donnez un nombre strictement positif: ";  
cin >> n;  
cout << "\n";  
while (n <= 0){  
    cout << " donnez un nombre strictement positif: ";  
    cin >> n;  
    cout << "\n";  
}  
cout << " le nombre choisi est " << n;
```

# L'instruction for

- Syntaxe de l'instruction

***for ([expression\_déclaration\_1]; [expression\_2]; [expression\_3])  
instruction***

- *expression\_déclaration\_1* est une expression ou une déclaration de variable(s)
  - Évalué une seul fois avant d'entrer dans la boucle
  - Les variables déclarées sont locales à la boucle
    - Si plusieurs variables sont déclarées, elles doivent être du même type
- *expression\_2* et *expression\_3* sont des expressions quelconques
  - *expression\_2* est évalué avant l'exécution d'instruction
    - Répète *instruction* tant qu'*expression\_2* est évalué à vrai
  - *expression\_3* est évalué après l'exécution d'*instruction*

# Exemple

```
int n;  
for(n = 0; n <= 0;){  
    cout << " donnez un nombre strictement positif: ";  
    cin >> n;  
    cout << '\n';  
}  
cout << "le nombre choisi est " << n;  
for(int i = 1; i < n; i++){  
    cout << i << " carré égal " << i*i << '\n';  
}
```





# Instructions de branchement

- L'instruction *break* peut être également utilisée dans une boucle
  - Interrompt la boucle pour passer à l'instruction qui suit la boucle
    - En cas de boucles imbriquées, *break* fait sortir de la boucle la plus interne
- L'instruction *continue* permet de passer au tour de boucle suivant
  - Dans le cas du *for*, *expression\_2* et *expression\_3* sont tout de même évaluées
    - En cas de boucles imbriquées, *continue* ne concerne que la boucle la plus interne

# Exemple

```
int n;  
do{  
    cout << " donnez un nombre positif: ";  
    cin >> n;  
    cout << "\n";  
    if(n == 0)  
        break;  
}while (n <= 0);  
for(int i = 1; i < n; i++){  
    if(i%2 == 0)  
        continue;  
    cout << i << " carré égal " << i*i << "\n";  
}
```

Ici, si  $n$  est égal à 0 alors l'instruction `break` fait sortir de la boucle `while`. Lorsque  $i\%2 == 0$  est vrai, l'instruction `continue` fait passer à l'itération suivante de la boucle `for` (sans effectuer l'affichage), mais ne fait pas sortir de la boucle `for`.



**ID 1**  
**Exercices**  
**1-4**

# Fonctions

Déclaration

Arguments

Variables globales/locales

# Les fonctions

- Il est pratique de décomposer un programme en parties indépendantes
  - Fonction: bloc d'instructions utilisable en donnant son nom et des paramètres

```
float square(float valeur){  
    float carre = valeur*valeur;  
    return carre;  
}
```

- La définition d'une fonction comprend un en-tête et un corps
  - En-tête: type de la valeur de retour, nom de la fonction et la liste des arguments
  - Le corps est un ensemble d'instructions délimitées par {}

# Arguments mués et effectifs

- Les arguments de l'en-tête se nomment des arguments mués
  - Permet au sein de la fonction de décrire ce qu'elle doit en faire
- Ceux fournis à l'appel de la fonction se nomment des arguments effectifs

```
float square(float valeur){  
    float carre = valeur*valeur;  
    return carre;  
}  
main(){  
    float y;  
    cin >> y;  
    cout << square(y);  
}
```

Ici, lors de l'appel de la fonction *square*, l'argument effectif est la valeur de *y*, et l'argument muet et la variable (locale) *valeur*



# L'instruction return

- Précise la valeur que la fonction fournira
  - Peut mentionner n'importe quelle expression
    - Si le type est différent du type de retour, une conversion systématique est effectuée
  - Interrompt l'exécution de la fonction en revenant dans la fonction appelante
  - Peut apparaître à plusieurs reprises dans la fonction

```
int val_absolue(int valeur){  
    if(valeur < 0)  
        return -valeur;  
    return valeur;  
}
```

Si *valeur* < 0 est vrai alors  
l'instruction *return -valeur*;  
va interrompre la fonction  
et la dernière instruction  
ne sera pas effectuée



# Void

- Quand une fonction ne renvoie pas de résultat, le type de retour est *void*
  - Peut disposer de l'instruction `return` sans expression pour interrompre l'exécution
- La liste des arguments d'une fonction peut être vide

```
void message_erreur(){  
    cout << "Message erreur\n";  
    return;  
}
```



# Fonction main

- En toute rigueur, la fonction main devrait fournir une valeur
  - Valeur susceptible d'être utilisée par l'environnement de programmation
    - 0 pour indiquer le bon déroulement du programme

```
int main(){  
    float i;  
    cout << "Entrez une valeur positive: ";  
    cin >> i;  
    if(i >= 0)  
        return 0;  
    return 1;  
}
```



# Déclaration



- La déclaration sert à prévenir le compilateur de l'existence d'une fonction
  - Facultative si la définition de la fonction est effectuée avant son utilisation
- La déclaration d'une fonction comporte
  - Son type de retour
  - Son nom
  - Le type de ses arguments

```
main(){  
    int val_absolve(int);  
    int i = val_absolve(-6);  
    ...  
}
```

Ici, la fonction `val_absolve` pourra être utilisée dans la fonction `main`, et ce quelque soit la position de sa définition dans le fichier



# Déclaration

- La déclaration d'une fonction peut être placée
  - À l'intérieur de toute fonction l'utilisant
    - Déclaration locale dont la portée est limitée à la fonction
  - Avant la définition de la première fonction
    - Déclaration globale

```
int val_absolve(int);  
main(){  
    int i = val_absolve(-6);  
    ...  
}
```

Ici, la fonction *val\_absolve* pourra être utilisée dans la fonction *main* et dans toutes les fonctions placées après dans le fichier



# Transmission des arguments pas valeur

➤ La valeur d'un argument effectif est donnée à la fonction



➤ Recopiée dans l'arguments muet correspondant qui est local à la fonction

```
void exchange(int a, int b){
```

```
    int c = a;
```

```
    a = b;
```

```
    b = c;
```

```
}
```

```
main(){
```

```
    int d = 5, e = 6;
```

```
    exchange(d, e);
```

```
    ...
```

```
}
```

Ici, l'échange est effectué sur les variables locales *a* et *b* de la fonction *exchange*, mais pas sur les variables *d* et *e* de la fonction *main*



# Transmission par référence

- La notion de référence correspond à l'adresse d'un argument effectif
  - Revient à considérer l'adresse d'une variable et non sa valeur

```
void exchange(int & a, int & b){  
    int c = a;  
    a = b;  
    b = c;  
}  
main(){  
    int d = 5, e = 6;  
    exchange(d, e);  
    ...  
}
```

Ici, l'échange est effectué sur les variables locales *a* et *b* de la fonction *exchange*, ainsi que sur les variables *d* et *e* de la fonction *main*



# Transmission par référence

- *int & a*: *a* est une information de type *int* transmise par référence
  - *a* est un argument muet qui reçoit l'adresse de l'argument effectif *d*
- L'argument effectif doit être une lvalue
  - Le type doit correspond exactement à celui de l'argument muet
- L'argument effectif peut être constant (utilisation du mot clé *const*)
  - La fonction ne pourra pas modifier cet argument
  - Exemple: *int val\_absolue(const int & valeur)*



La transmission par référence a néanmoins quelques désavantages et ne peut pas être utilisée de manière systématique:

- La valeur donnée à la fonction ne pourra plus être une constante
- La variable donnée en paramètre peut être modifiée sans que l'utilisateur ne s'en rende compte

C'est pour cette raison que le mot clé *constant* peut être utile dans certains cas

# Variable globale

- Plusieurs fonctions peuvent partager des variables communes
  - Ces variables sont appelées variables globales
    - Déclarées en dehors de toute fonction
    - Connues que dans la partie du programme suivant leurs déclarations
    - Elles existent pendant toute l'exécution du programme dans lequel elles apparaissent
- 💡 ➤ Pratique risquée qu'il faut éviter au maximum




# Exemple

```
int d = 5, e = 6;  
void exchange(){  
    int c = d;  
    d = e;  
    e = c;  
}  
main(){  
    exchange();  
    ...  
}
```

Ici, l'échange est effectué sur les variables globales *d* et *e*, qui sont accessibles à toutes les fonctions



# Variables locales

- Une variable définie au sein d'une fonction est dite locale
  - Sa portée est limitée à la fonction où elle est déclarée
  - Sa durée de vie est limitée à l'exécution de la fonction où elle est déclarée
- 
  - Un espace mémoire est alloué à chaque entrée dans la fonction et libéré à sa sortie
  - Les valeurs des variables locales ne sont pas conservées d'un appel à l'autre
- Les arguments muets d'une fonction sont des variables locales



# Variables locales à un bloc

► Une variable peut être déclarée dans un bloc



► Sa portée est limitée à ce bloc

► Emplacement mémoire alloué à l'entrée du bloc et libéré à sa sortie

► Les variables déclarées dans un *for* sont locales à cette instruction

```
int j = 1;
for(int i = 1; i < 5; i++)
    cout << "affichage: 1." << i << "\n ";
while(j < 5){
    int i = j++;
    cout << "affichage: 2." << i << "\n ";
}
```



TD 1  
Exercices  
5-8

Ici, la variable *i* déclarée dans le *for* est locale à la boucle et n'existe plus à la sortie de la boucle. La variable *i* du *while* est une autre variable qui est locale au bloc de la boucle *while*



# Spécification *inline*

- Demande au compilateur de traiter la fonction différemment
  - Exemple: `inline void exchange(int a, int b){...}`
  - Incorpore au sein du programme les instructions correspondantes
  - Le mécanisme habituel de gestion de l'appel n'existera plus
    - Permet une économie de temps
  - Les instructions correspondantes seront générées à chaque appel
    - Consommara une quantité de mémoire croissante avec le nombre d'appel
  - Doit être définie dans le même fichier source que celui où on l'utilise
    - Pour la partager entre différents programmes, il faut la placer dans un fichier d'en-tête

Pour ce type de fonctions, chaque appel de fonction est remplacé par le code correspondant (sorte de copier/coller). Il n'y a donc pas réellement d'appel à la fonction, ce qui accélère l'exécution du code, mais le rallonge également (plus de lignes)



# Tableaux et pointeurs

Tableaux

Pointeurs

Gestion dynamique des tableaux

Tableaux et fonctions

Chaines de caractères

# Tableau

- Ensemble d'éléments de même type désigné par un identificateur unique

***float tab[10];***

```
for(int i = 0; i < 10; i++){  
    cout << "\n entrez la valeur " << i+1 << " :";  
    cin >> tab[i];  
}
```

- La déclaration *float tab[10]* réserve 10 éléments de type *float*

# Indices

- Chaque élément est repéré par son indice
  - La première position porte le numéro 0
- $tab[i]$  désigne l'élément à la position  $i$  du tableau  $tab$ 
  - Un indice peut être une expression arithmétique d'un type quelconque entier
- ⚠ ➤ Les éléments d'un tableau sont des lvalues (peut affecter une valeur)
  - $tab[i] = j;$

# Dimension

- ! ➤ L'affectation globale de tableau n'est pas possible
  - Si *tab\_1* et *tab\_2* sont des tableaux, l'instruction *tab\_1 = tab\_2* n'est pas autorisée
- ! ➤ La dimension d'un tableau ne peut être qu'une constante
- ! ➤ Aucun contrôle de débordement d'indice n'est mis en place
  - Il est très facile de sortir d'un tableau

# Tableau à plusieurs indices

- Il est possible d'utiliser des tableaux à plusieurs indices

```
float tab[10][10];
```

```
for(int i = 0; i < 10; i++){  
    for(int j = 0; j < 10; j++){  
        cout << "\n entrez la valeur " << i+1 << " *" << j+1 << " :";  
        cin >> tab[i][j];  
    }  
}
```

- La déclaration `float tab[10][10]` réserve 10×10 éléments de type `float`



# Indices

- Un élément de ce tableau est repéré par deux indices
  - Chaque élément de ce tableau est une lvalue
- ⚠ ➤ Dans ce cas, *tab[i]* n'est ni un élément du tableau, ni une lvalue
- Aucune limitation sur le nombre d'indices que peut comporter un tableau



# Initialisation de tableau

- Il est possible d'initialiser un tableau lors de sa déclaration
  - `int tab[5] = { 10, 20, 5, 0, 3};`
- Il est possible de ne mentionner que les premières valeurs
  - `int tab[5] = { 10, 20};`
- Si un tableau est initialisé, il est possible d'omettre sa dimension
  - `int tab[] = { 10, 20, 5, 0, 3};`
  - Sa taille sera définie comme le nombre de valeurs énumérées

# Initialisation pour plusieurs indices

- On peut initialiser un tableau à plusieurs indices lors de sa déclaration
  - `int tab[3][2] = { { 0, 1}, {2, 3}, {4, 5} };`
- A chacun des niveaux, des valeurs peuvent être omises
  - `int tab[3][2] = { { 0}, {2, 3} };`



# Pointeurs

- Il est possible de manipuler des adresses par l'intermédiaire de pointeurs

```
int n = 20;
```

```
int *addr;
```

```
addr = &n;
```

```
*addr = 10;
```

- `int *addr` réserve une variable nommée `addr` comme un pointeur sur un `int`
- `addr = &n` affecte à la variable `addr` l'adresse de la variable `n`
  - `&` est un opérateur unaire qui fournit l'adresse de son opérande



# Pointeurs

```
int n = 20;  
int *addr;  
addr = &n;  
*addr = 10;
```

Attention, il ne faut pas confondre l'opérateur binaire `*` de la dernière instruction et le `*` de la seconde instruction.

- dans la seconde instruction, `*` signifie que la variable `addr` est un pointeur,
- dans la dernière instruction, l'opérateur `*` appliqué à l'adresse contenue dans `addr`



- L'instruction `*addr = 10` affecte à la lvalue `*addr` la valeur 10
  - `*` est un opérateur unaire qui fournit le contenu situé à l'adresse qui le suit
    - `*addr` est l'`int` ayant pour adresse celle contenue dans `addr`
  - Équivalent ici à `n=10`



# Pointeurs

```
int n = 20;  
int *addr;  
addr = &n;  
*addr = 10;
```

- *addr* est une variable (de type pointeur) qui contient une adresse (celle de *n* ici).
- *\*addr* correspond à la variable dont l'adresse est contenue dans la variable *addr* (ici il s'agit de la variable *n*).
- *&addr* correspondrait à l'adresse en mémoire de la variable *addr*.



# Simuler transmission par référence

```
void exchange(int *a, int *b){  
    int c = *a;  
    *a = *b;  
    *b = c;  
}  
  
main(){  
    int d = 5, e = 6;  
    exchange(&d, &e);  
    ...  
}
```

Ici, *a* prend pour valeur l'adresse de *d*, et *b* prend pour valeur l'adresse de *e*. Dans le bloc d'instructions de la fonction *exchange*, *\*a* correspond à la variable *d*, et *\*b* correspond à la variable *e*. L'échange est donc bien effectif dans la fonction *main*.





# Simuler transmission par référence

- Les arguments effectifs sont les adresses des variables *d* et *e*
  - La transmission se fait par valeur des expressions *&d* et *&e*
- Les arguments muets de la fonction sont des pointeurs vers des *int*
  - *\*a* et *\*b* correspondent aux variables pointées par *a* et *b*



# Opérations sur les pointeurs

- Si *addr* est un pointeur sur un *int* alors l'expression *addr+1* représente l'adresse de l'*int* suivant en mémoire
  - Il s'agit de l'adresse contenue dans *addr* augmenté de la taille d'un *int*
  - Si *addr* était un *\*double* alors *addr+1* serait augmenté de la taille d'un *double*
- L'expression *addr++* fait pointer *addr* sur l'*int* suivant en mémoire
-  ➤ Attention: valide quelle que soit l'information réellement présente en mémoire
  - Peut pointer sur un emplacement mémoire qui ne correspond pas en réalité à un *int*
-  ➤ Il est possible d'affecter à un pointeur la valeur 0
  - Permet de représenter un pointeur nul, c'est-à-dire ne pointant sur rien



# Tableau et pointeur



➤ Une variable tableau est un pointeur (constant) sur le début du tableau

➤ Si *tab* est un tableau alors *tab* est équivalent à *&tab[0]*

➤ Les expressions *tab[i]* et *\*(tab+i)* sont équivalentes

➤ L'expression *tab++* est invalide

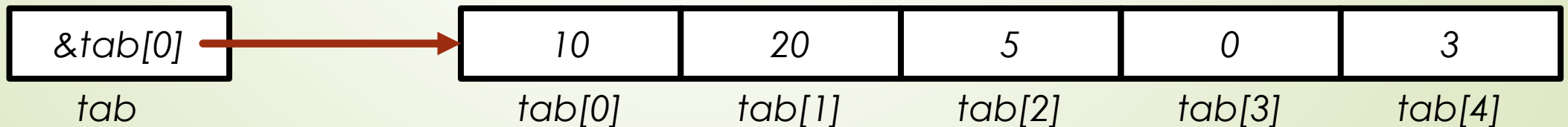


➤ Un tableau n'est pas une lvalue car c'est un pointeur constant



Une variable de type tableau contient l'adresse de sa première case (ici, l'adresse de *tab[0]*)

*tab+i* est l'adresse de la *i<sup>eme</sup>* case en partant de la première case du tableau. *\*(tab+i)* est son contenu



# Gestion dynamique et new



- C++ offre la possibilité d'allocation dynamique de mémoire
  - Les emplacements sont alloués et libérés à la demande du programme
- L'opérateur unaire *new* permet d'allouer un ou plusieurs espaces mémoires
  - L'expression ***new type*** réserve un emplacement mémoire et renvoie son adresse
    - ***type*** correspond à un type quelconque (*char*, *int*, ...)
  - L'expression ***new type[n]*** réserve un tableau de *n* éléments
- En cas d'échec, *new* déclenche une exception de type *bad\_alloc*

```
char *tab;  
tab = new char[100];  
for(int i = 0; i < 100; i++)  
    tab[i] = 0;
```



Ici, 100 cases mémoires de type *char* sont réservés. L'adresse de la première case mémoire est renvoyée et placée dans *tab*. *tab* pointe donc sur la première case d'un tableau alloué dynamiquement. *tab[i]* permet d'accéder à la  $i^{eme}$  case de ce tableau

# Gestion dynamique et delete

- L'opérateur *delete* permet de libérer un espace mémoire alloué par *new*
  - Syntaxe: ***delete* adresse**
    - *adresse* est une expression pointeur sur un emplacement alloué par *new*
    - Exemple: *delete p;*
  - Syntaxe ***delete[]* adresse**
    - *adresse* est une expression pointeur sur un emplacement alloué par *new []*
    - Exemple: *delete [] tab;*
  - Le comportement du programme n'est pas défini lorsque
    - L'adresse correspond à un emplacement déjà libéré
    - L'adresse ne correspond pas à un emplacement alloué par *new*

# Tableau et fonction

- Lorsqu'un tableau est un argument effectif d'une fonction
  - C'est l'adresse du premier élément du tableau qui est transmis
  - Permet d'effectuer toutes les manipulations voulues sur ses éléments
  - La notion de référence à un tableau n'a pas de sens en C++



```
void initialisation(int tab[10]){  
    for(int i = 0; i < 10; i++)  
        tab[i] = 0;  
}  
...  
int tab2[10];  
initialisation(tab2)
```



Ici, l'appel de la fonction *initialisation* va effectivement modifier le contenu de *tab2* car la variable *tab* va pointer sur la même case que *tab2*, qui est la première case du tableau *tab2*

# Tableau, fonction et pointeur

- Il est possible ne pas mentionner la dimension du tableau dans l'en-tête
  - Exemple: `void initialisation(int tab[]){...}`
  - Attention: ce n'est pas le cas pour les tableaux à plusieurs indices
- Il est également possible d'utiliser le formalisme pointeur
  - Exemple: `void initialisation(int *tab){...}`
  - Quel que soit l'en-tête, on peut utiliser le formalisme pointeur ou tableau

```
void initialisation(int *tab){  
    for(int i = 0; i < 10; i++)  
        tab[i] = 0;  
}
```



Comme dans la fonction précédente, la variable *tab* va pointer sur la première case du tableau donné en paramètre de la fonction *initialisation*



# Tableau de taille variable

- ▀ Peut faire une fonction qui travaille avec un tableau de taille quelconque
  - ▀ La taille du tableau doit être donnée en argument

```
void initialisation(int *tab, int taille){  
    for(int i = 0; i < taille; i++)  
        tab[i] = 0;  
}
```



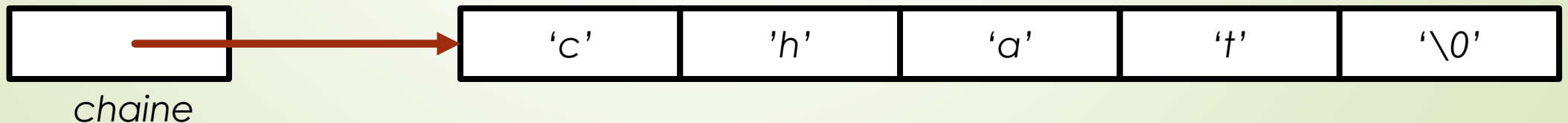
La variable *tab* est un pointeur sur la première case du tableau, c'est-à-dire une adresse. Cette variable à elle seule n'indique pas la taille du tableau. Il faut donc un second paramètre à la fonction correspondant à la taille



# Chaine de caractères

- Convention de représentation héritée du langage C
- Suite de *char* correspondant à chacun de ses caractères
- ! ➤ Terminée par un *char* supplémentaire de code nul ('\0')
- Une chaîne constante sera traduite en pointeur sur le premier *char* réservé
  - Le compilateur crée en mémoire la suite d'octets correspondante

```
char *chaine;  
chaine = "chat";
```





# Lecture de chaine

- Il est possible de lire une chaine de caractères à l'aide d'un tableau

```
char chaine[20];
```

```
cout << "donnez une chaine de caractères: ";
```

```
cin >> chaine;
```

- La lecture de la chaine s'arrête au premier séparateur rencontré
  - La chaine de caractères ne peut comprendre ni d'espace, ni '\n'
- Le tableau doit être suffisamment grand pour contenir la chaine lue
  - Ici, au plus 19 caractères (plus le caractère de fin de chaine '\0')

# Lecture et tableau dynamique

- ▀ Peut également utiliser un tableau dynamique

```
char *chaine = new char[20];  
cout << "donnez une chaine de caractères: ";  
cin >> chaine;
```



# Initialisation de tableau



- Il n'est pas possible de transférer une chaîne constante dans un tableau
  - L'expression `tab = "bonjour"` n'est pas autorisée



- Mais Il est possible d'initialiser un tableau avec une chaîne de caractères
  - L'instruction `char tab[] = "bonjour";` est autorisée
    - Équivalente à `char tab[] = {'b', 'o', 'n', 'j', 'o', 'u', 'r', '\0'};`
    - Ici `"bonjour"` n'est pas une chaîne constante



- Dans le cas d'un tableau dynamique
  - Il n'est pas possible d'initialiser le tableau à sa création

# Fonctions portant sur des chaines

## ➤ ***int strlen(char \*chaine)***

- Donne la longueur de *chaine* dont l'adresse est fournie en argument
  - Nombre de caractères à partir de l'adresse indiquée jusqu'au premier '\0'

## ➤ ***int strcmp(char \*chaine\_1, char \*chaine\_2)***

- Compare les chaines de caractères fournies en paramètre
  - >0: *chaine\_1* arrive après *chaine\_2* dans l'ordre défini par le code des caractères
  - =0: les deux chaines contiennent la même suite de caractères
  - <0: *chaine\_2* arrive après *chaine\_1* dans l'ordre défini par le code des caractères

# Fonctions portant sur des chaines

- **`char* strcat(char *but, char *source)`**
  - Recopie la chaine source à la suite de la chaine *but*
    - Efface la caractère '`\0`' qui était présent dans la chaine *but*
  - Renvoie 0 si l'opération s'est mal déroulée
- **`char* strcpy(char *but, char *source)`**
  - Recopie la chaine source dans l'emplacement d'adresse *but*

