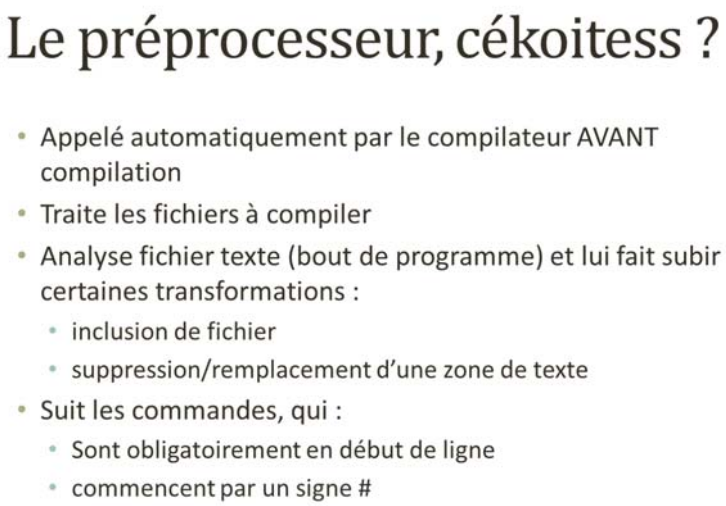
Chapitre 2. Fonctionnalités de base en C++

Il y a une vie avant le programme exécutable : préprocesseur et espaces de nommage.

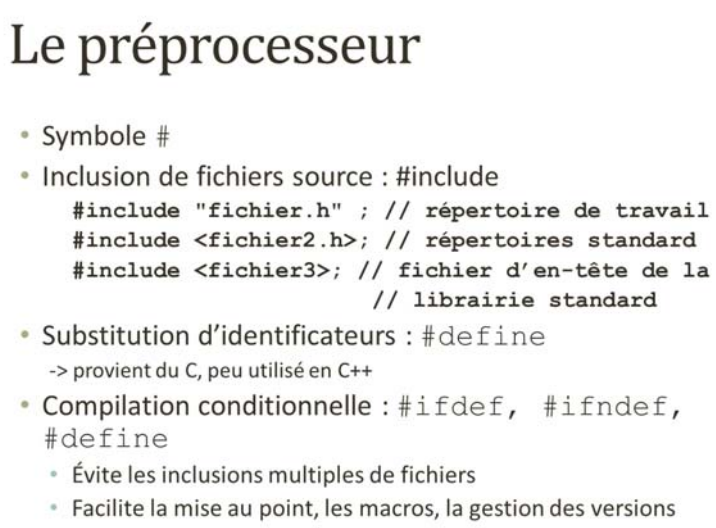
Avant d'obtenir un programme exécutable, il faut le compiler et pour pouvoir le faire, il faut mettre en forme les fichiers de code de façon à ce qu'ils soient conformes au format attendu.
Par ailleurs, dans le code lui-même, il faut définir l'espace-mémoire dans lequel les variables et les procédures et méthodes seront définies, afin d'éviter les conflits entre programmes.

Préprocesseur



SLIDE 3

Le *préprocesseur* est un programme qui analyse un fichier texte et lui fait subir certaines transformations.
Ces transformations peuvent être l'*inclusion d'un fichier*, la *suppression* d'une zone de texte ou le *remplacement* d'une zone de texte.
Le préprocesseur effectue ces opérations en suivant des ordres qu'il lit dans le fichier en cours d'analyse. Il est appelé automatiquement par le compilateur, avant la compilation, pour traiter les fichiers à compiler.
Les commandes destinées se trouvent obligatoirement en début de ligne et démarrent par un signe #.



Slide 4

Toutes les commandes du préprocesseur commencent :

- en début de ligne
- par un signe dièse (#)

On peut :

- Inclure des fichiers
- Définir des constantes de compilation et/ou faire du remplacement de texte (cf. aliases)
- Faire de la compilation conditionnelle

La définition des identificateurs et des constantes de compilation est très utilisée pour effectuer ce que l'on appelle la *compilation conditionnelle*.
La compilation conditionnelle consiste à remplacer certaines portions de code source par d'autres, en fonction de la présence ou de la valeur de constantes de compilation.
On utilise des directives de compilation conditionnelle, dont la plus courante est probablement *#ifdef* :

#ifdef identificateur
⋮
#endif

Dans l'exemple précédent, le texte compris entre le *#ifdef* (c'est-à-dire « if defined ») et le *#endif* est laissé tel quel si l'identificateur *identificateur* est connu du préprocesseur. Sinon, il est supprimé. L'identificateur peut être déclaré en utilisant simplement la commande *#define*.

Il existe d'autres directives de compilation conditionnelle :
#ifndef (if not defined ...) #elif (sinon, si ...) #if (si ...)
La directive *#if* attend en paramètre une expression constante. La séquence d'instructions qui la suit est incluss dans le fichier si et seulement si cette expression est non nulle (vraie).

Par exemple :
#if (__cplusplus==199711L)
⋮
#endif
permet d'inclure un morceau de code C++ strictement conforme à la norme décrite dans le projet de norme du 2 décembre 1996.

Une autre application courante des directives de précompilation est la protection des fichiers d'en-tête contre les inclusions multiples :
#ifndef DejaLa
#define DejaLa
Texte à n'inclure qu'une seule fois au plus.
#endif

Cela permet d'éviter que le texte soit inclus plusieurs fois, à la suite de plusieurs appels de *#include* (utilisé notamment pour l'inclusion de fichiers d'en-tête, pour éviter les inclusions multiples).
En effet, au premier appel, *DejaLa* n'est pas connu du préprocesseur. Il est donc déclaré et le texte est inclus. Lors de tout autre appel ultérieur, *DejaLa* existe, et le texte n'est pas inclus.
Ce genre d'écriture se rencontre dans les fichiers d'en-tête, pour lesquels en général on ne veut pas qu'une inclusion multiple ait lieu.

Espaces de nommage

Slide 5

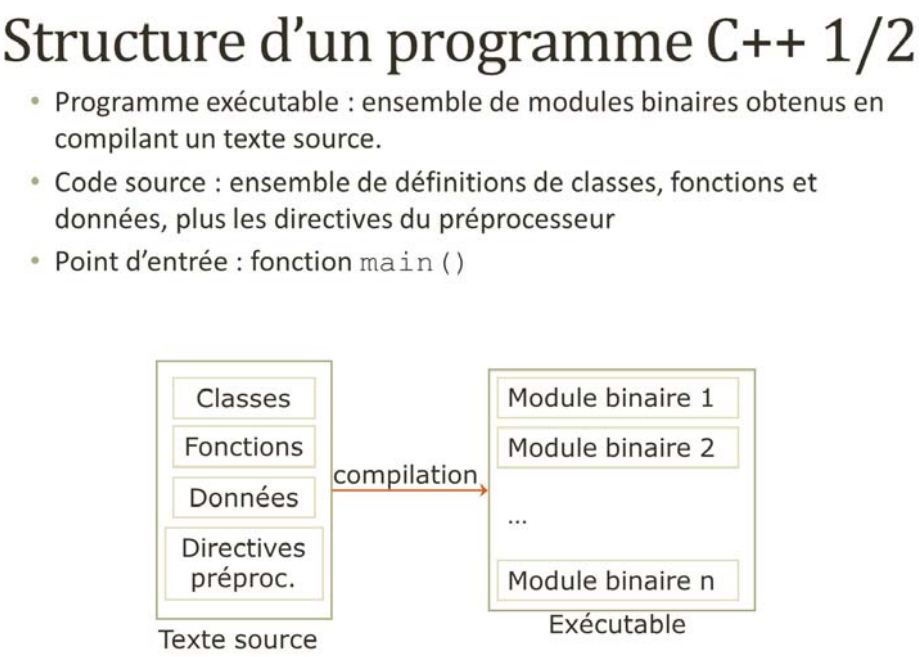
Les *espaces de nommage* sont des zones de déclaration qui permettent de délimiter la recherche des noms des identificateurs par le compilateur.
Leur but est essentiellement de regrouper les identificateurs logiquement et d'éviter les conflits de noms entre plusieurs parties d'un même projet.

Par exemple, si deux programmeurs définissent différemment une même structure dans deux fichiers différents, un conflit entre ces deux structures aura lieu au mieux à l'édition de liens, et au pire lors de l'utilisation commune des sources de ces deux programmeurs.

Ce type de conflit provient du fait que le C++ ne fournit qu'un seul espace de nommage de portée globale, dans lequel il ne doit y avoir aucun conflit de nom. Grâce aux espaces de nommage non globaux, ce type de problème peut être plus facilement évité, parce qu'on peut éviter de définir les objets globaux dans la portée globale.

Attention : la commande using namespace est à réserver aux portions de code dans lesquelles on est sûr qu'il n'y aura pas d'ambiguïté, pour alléger le code. Il faut en limiter la portée autant que possible pour éviter les conflits.

Structure globale d'un programme exécutable en C++



Slide 6

Un programme exécutable est un ensemble de modules binaires obtenus en compilant un texte source (ensemble de définitions de classes, fonctions et données, plus des directives pour le préprocesseur).
Le point d'entrée du programme est constitué par la fonction « main() », le programme principal, qui appelle les sous programmes.

A la compilation, on passe d'un texte lisible par un être humain à un code binaire compréhensible et exécutable par la machine.
Le compilateur et l'architecture sur laquelle le code a été compilé sont importants, et l'on peut rencontrer des problèmes de compatibilité.

Quelques règles de syntaxe de base pour une bonne compilation...

Structure d'un programme C++ 2/2

- Point d'entrée : fonction `main()`
- Caractéristiques lexicales :
  - Présentation du source libre
  - Mots-clés réservés
  - Majuscules ≠ minuscules
  - Commentaires marqués par // et/ou /* ... */
- Inclusion de bibliothèques : plus d'extensions .h (librairies C renommées : `<stdlib.h>` ⇒ `<cstdlib>`)

Slide 7

Exemples de mini programmes

Un exemple 1/2

```
// Premier programme C++
/* Commentaire
   sur plusieurs lignes */
#include <iostream> // préprocesseur
int main( ) {
     int a = 42; // sensible à la casse
     int A = 24;
     std::cout << "a : " << a << std::endl;
     std::cout << "A : " << A << std::endl;
     return 0;
}
```

Slide 8

Un exemple 2/2

```
#include <iostream> // Exemple 1
int main( ) {
     std::cout << "Test" << std::endl;
     return 0;
}

#include <iostream> // Exemple 2
using namespace std;
int main( ) {
     cout << "Test" << endl;
     return 0;
}
```

Slide 9

Notez : la forme des commentaires, des directives préprocesseur, de l'appel à l'espace de nommage, la mise en couleur des mots-clés réservés et la casse des variables…

Bonne pratique : éviter les noms ambigus et choisir des noms de variables explicites pour faciliter la compréhension (du type de la variable et de son rôle dans le programme)

NB : l'inclusion *via* la commande #include a vocation à disparaître, à terme, car elle génère trop d'erreurs...

Syntaxe et types de base en C++

Les variables

- Nom
  - unique (espace de nom)
  - commence par une lettre
  - sensible à casse
- Type
  - fondamental, construit
- Valeur
  - constante ou "variable" (évolutive)
  - initialisation

Slide 11

Déclaration de variables

- Déclaration avant utilisation
- Positionnée n'importe où dans le programme
- Déclaration ≠ définition

```
int i; // déclaration et définition de l'entier i
extern int j; // déclaration de l'entier j
int k=12;
  // déclaration, définition et initialisation de
  // l'entier k
```

- Possibilité de déclarer un *alias* (typedef) :

```
typedef unsigned int uint;
   // uint est un alias pour unsigned int
uint i; // ⇔ unsigned int i
```

Slide 12

Déclaration de variables (bis)

- Attention ! Déclaration ≠ Définition ≠ Initialisation !
  - Déclaration : donne le type d'une variable au compilateur
  - Définition : crée une entité du type de la variable
  - Initialisation : donne une valeur à cette entité
- Portée d'une variable : limitée au groupe {} dans lequel elle est déclarée
- On peut définir des *alias* (mot-clé typedef)

Slide 13

La déclaration revient à préciser au compilateur le type d'un identifiant, tandis que la définition revient à créer une entité du type en question.
Attention, la déclaration, la définition et l'initialisation sont 3 choses différentes !

- Déclaration : donne le type d'une variable au compilateur
- Définition : crée une entité du type de la variable
- Initialisation : donne une valeur à cette entité

La portée des variables, en C++ est limitée au groupe d'accolades {} dans lequel elles sont déclarées.

Le mot clé typedef permet de déclarer des *alias*, *i.e.* des « surnoms » pour une « phrase de programmation » un peu longue.

Variables externes

Fichier "ext.cpp"

```
int c = 10; // déclaration, définition, initialisation de c
```

Fichier "main.cpp"

```
#include "ext.cpp"
int main () {
extern int c; // déclaration de c, c vaut 10 (ext)
c++; // c vaut 11
c = 42; // c vaut 42
//…
}
```

Attention : parcimonie est votre meilleure amie !

Slide 14

Les types fondamentaux en C++

Types fondamentaux en C++ 1/4)

- Caractères : `char` (256 valeurs sur 8 bits)
  - 1 octet, 8 bits, 256 valeurs
    - signed : -128 / +127 (type par défaut)
    - unsigned : 0 / 255
  - Utilisation de wchar_t pour les jeux de caractères étendus
  - Opérations arithmétiques et logiques possibles
  - NB : chaînes de caractères introduites dans la librairie standard, classe `string`

```
char c='A'; // 65 en mémoire
char e = c + 32; //autorisé
⇒e vaut 97 en mémoire et 'a' à l'affichage
```

Slide 15

Le type CHAR correspond aux caractères ASCII.
On peut aussi utiliser les « signed char » ou les « unsigned char »…

NB : Char n'est pas une chaîne de caractères, mais un seul caractère… les chaînes de caractères sont définies par la classe string de la librairie standard (on en parlera plus loin).
Il arrive qu'on utilise le type CHAR pour de très petits nombres… attention aux débordements (modulo) !

Types fondamentaux en C++ (2/4)

- Flottants
  - `float` : simple précision, 4 octets
  - `double` : double précision, 8 octets (par défaut)
  - `long double` : précision étendue, 10 octets (C++)
- Exemple de notation :

```
float pi = 3.14159f;
long pi = 3.14159l;
double pi = 3.14159;
double pi = 314159e-5;
```

NB : le l et le f sont facultatifs, mais évitent les conversions implicites.

Slide 16

Flottants : par défaut, une constante flottante est de type double si l'on ne précise pas son type.
Nb : dans l'exemple, le f et le l ne sont pas obligatoires mais évitent une conversion implicite

Types fondamentaux en C++ (3/4)

- Entiers

  type de base : `int` (2 ou 4 octets)
  - `signed` (par défaut) -2/+2 milliards
  - `unsigned`
  - `short` (par défaut), 2 octets ou `long`, 4 octets
  - Plusieurs formes pour une constante entière :
    - Décimale (int i = 63)
    - Octale (0 suivi du nombre en base 8) (int i = **0**77)
    - Hexadécimale (0x suivi du nombre en base 16) (int i = **0x**3f)
- Booléens (C++, pas C)
  - `bool` : deux valeurs, `true (1)` ou `false (0)`

  Ex: `bool b=true; if (b) {} ; if (b==true) {}`

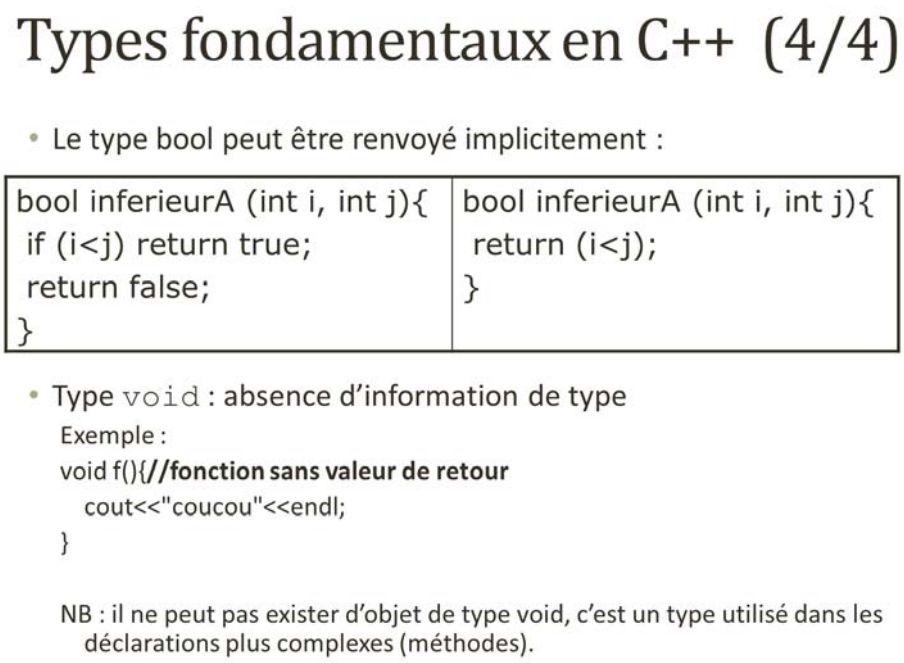
Slide 17

ENTIERS : int (2 octets, ou 4 selon architecture)

BOOLEENS : quand on convertit un booléen en entier, on obtient 0 ou 1. Par contre, quand on convertit un entier en booléen, tout entier différent de 0 vaut « true » (même un entier négatif)

Quand on déclare : bool b;
On peut ensuite faire des tests du genre « if b==true » ou simplement « if (b) »

bool : deux valeurs, true (1) ou false (0)
Conversion int/char : true (<> 0), false (=0)
Ex:
bool b=true;
if (b) {} ;
if (b==true) {}



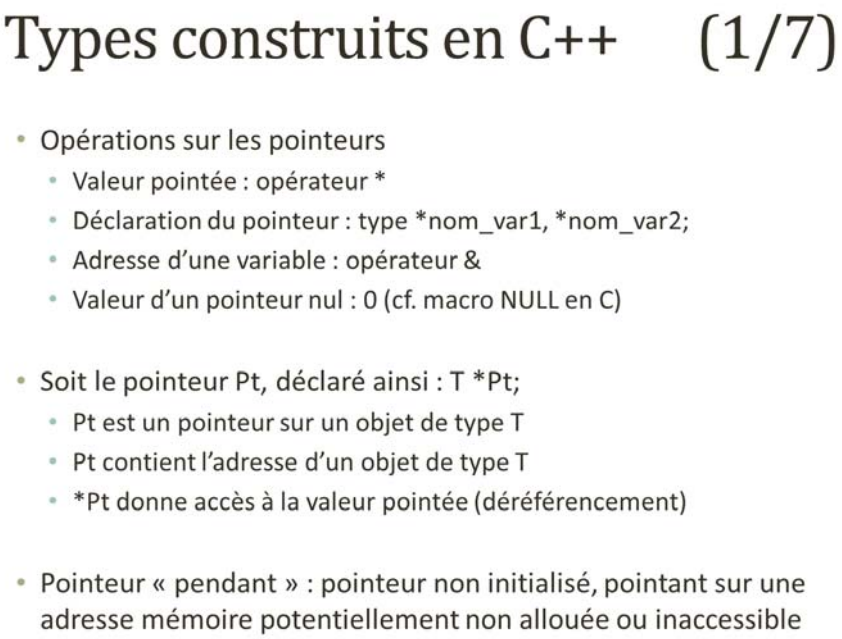
Slide 18

Le type booléen peut être retourné de façon implicite, en renvoyant l'évaluation d'une expression.

Type VOID/absence de type : Attention, il ne peut pas exister d'objets de type void, mais il est utilisé dans des déclarations plus complexes (méthodes)

Types construits en C++

Les pointeurs



Slide 19

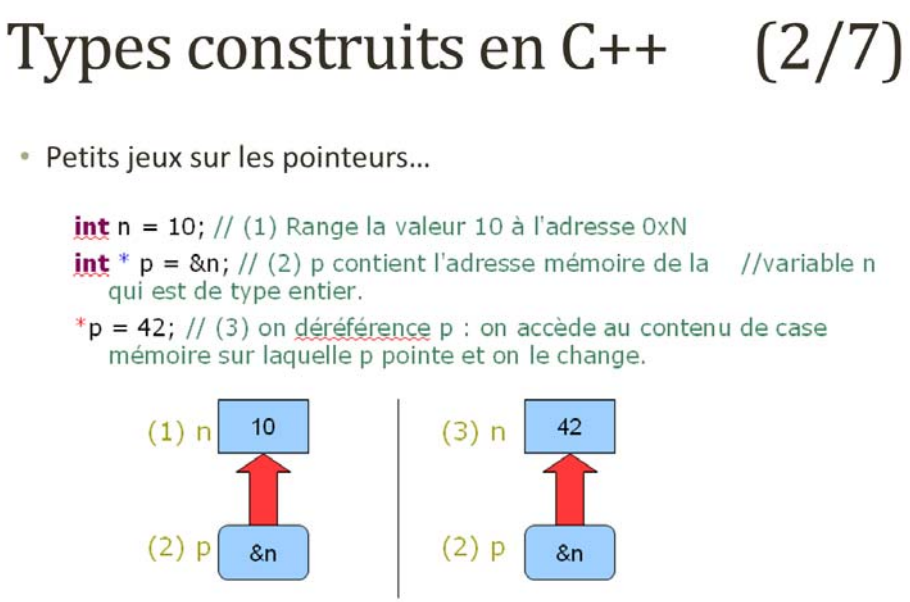
Un pointeur P déclaré comme suit T *Pt; est un pointeur sur un objet de type T : il contient l'adresse d'un objet de type T.

L'opérateur * permet de « déréférencer » un pointeur, i.e. d'accéder à la valeur pointée…

Exemple :
int i;
int *Pi = &i; // Pi contient l'adresse de i
int k=*Pi; // revient à faire int k=i;

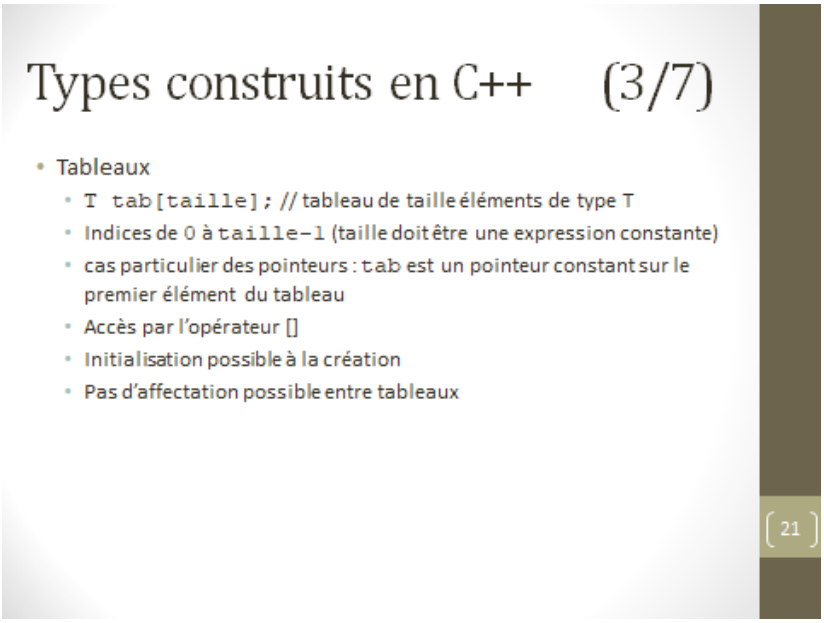
Notion de "Pointeur pendant"
Il s'agit d'un pointeur qui n'a pas été initialisé : il pointe donc sur une adresse mémoire qui peut être non-allouée ou inaccessible

int *p ; // p pointe sur une adresse mémoire qui n'a pas encore été allouée. Cette adresse devra contenir un entier. On peut utiliser p dès maintenant en le faisant pointer vers un entier existant. Sinon, il s'agit d'un pointeur pendant.
*p = 20 ; // DANGER, ça peut compiler et même fonctionner mais c'est périlleux (opération similaire, par exemple, à : int i; cout << i << endl;)
p ne contient pas encore l'adresse d'une variable de type entier, on ne peut donc pas lui attribuer de valeur.



Slide 20

Un type de pointeur un peu spécial : les tableaux !



Slide 21

Les tableaux sont un cas particuliers des pointeurs, en C++.
Le nom d'un tableau est en fait un pointeur constant sur le 1er élément du tableau et les éléments sont stockés de façon contiguë en mémoire…
l'opérateur [] permet d'accéder aux éléments du tableau, mais il est possible d'y accéder sans, en décalant l'adresse de départ d'autant d'éléments que nécessaire (en termes d'espace mémoire).

Exemples :
char* nom = "vdf";
nom est un tableau de taille 3 : | v | d | f |

Exemple d'utilisation des tableaux
int tab[3] = {0,1}; // tab[0], tab[1] et tab[2] sont déclarés ; tab[0] et tab[1] sont initialisés
tab[2] = 2;

tab est un pointeur sur un objet de type int.
Un tableau est un pointeur sur le premier élément : tab[1] <=> *(tab+1) // (adresse de tab[0] +1 emplacement mémoire)

Structures et unions, héritées du C

Types construits en C++ (4/7)

- `Struct` et `union`
  - Structure : agrégat de différents types (ancêtre de la classe en C++)

Ex : struct Personne{
char *nom;
int age;
};
Personne pers; // une instance de type Personne
Personne *ppers; // pointeur sur Personne

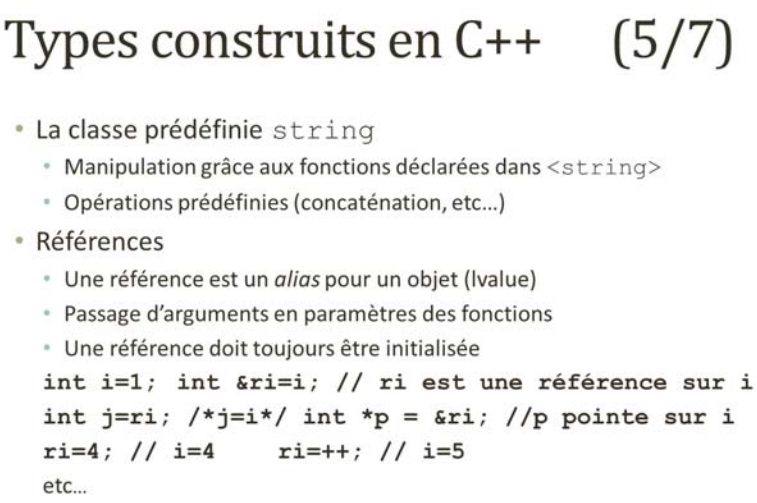
pers.age=6; // champ age de pers
ppers->age=6; // champ age de ppers

- NB : la déclaration est différence de celle du C
- `union` : même fonctionnement que `struct` mais tous les champs partagent le même espace mémoire

Slide 22

La déclaration d'une instance d'une structure est différente de celle du langage C.
Unions et structures sont équivalentes à l'usage, mais pour une union, chaque champ partage le même espace mémoire (celui du plus grand élément déclaré)

Nouveautés apportées par le C++



Slide 23

La classe prédéfinie string : l'alternative aux tableaux de caractères, une classe complète, avec les méthodes permettant de la manipuler de façon efficace !

REFERENCES : leur utilisation principale est le passage d'arguments en paramètres de fonctions

```
int i=1; int &ri=i; //ri est une référence sur i
int j=ri; //j=i
int* p=&ri; //p pointe sur i
ri=4; //i=4
ri++; //i=5
ri = i ;  ri = j ; // ok si ri n'est pas const
extern int& k; //ok
int& m; //PAS ok
int& m = 3; //PAS ok
```

Référence ou pointeur ?

Types construits en C++ (6/7)

- Faut-il utiliser une référence ou un pointeur ?

	Référence	Pointeur
Nom	Le nom de la référence représente directement l'objet	Le nom du pointeur représente l'adresse de l'objet
Adresse	L'adresse d'une référence est l'adresse de l'objet	L'adresse du pointeur est l'adresse de la variable contenant l'adresse de l'objet
Accès	Pas de notation spécifique pour accéder à l'objet	Notation spécifique pour déréférencer le pointeur
Opérations	Une opération sur une référence manipule directement l'objet	Une opération sur un pointeur manipule l'adresse de l'objet
Constantes	Une référence constante rend l'objet constant	Un pointeur peut être un pointeur sur un objet constant, un pointeur constant sur un objet ou un pointeur constant sur un objet constant
Fonctions	Pas de manipulation d'adresse de fonction par une référence	manipulation d'adresse de fonction (passage de traitement en paramètre, tableau de pointeurs de fonction, …)

Slide 24

Un POINTEUR est un concept de bas niveau permettant une manipulation précise de l'adresse d'un objet (arithmétique des pointeurs, pointeurs de fonctions), tandis qu'une REFERENCE est une abstraction de haut niveau fournissant une interface plus simple mais plus limitée pour manipuler l'adresse d'un objet.

Remarque : il est possible que les références finissent par disparaître dans les versions ultérieures de C++.

Types construits en C++ (7/7)

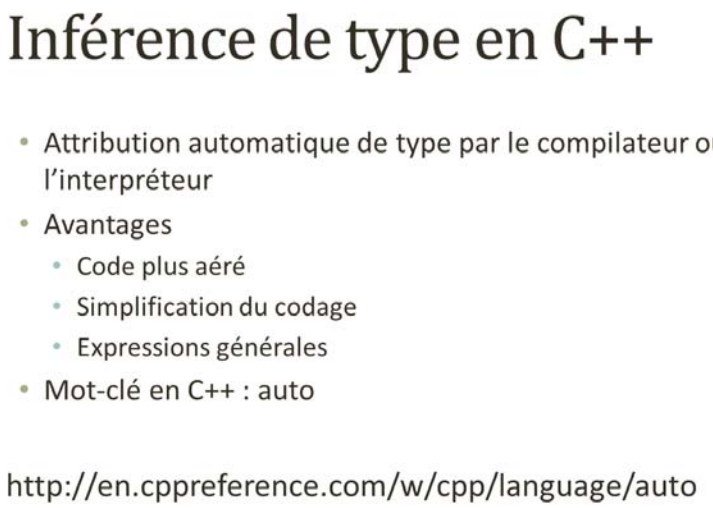
- Constantes
  - Mot-clé `const`
  - Ajoutable à toute déclaration
- Constantes et pointeurs…

```
const char *p1= "azerty"  ;
   // pointeur sur un caractère constant
char c; // caractère
char *const p2=&c;
   // pointeur constant sur caractère
const char * const p3= "azerty";
 // pointeur constant sur caractère constant
```

Slide 25

Les constantes ont plusieurs usages. On peut les utiliser pour déclarer un tableau, par exemple… mais on peut aussi déclarer des pointeurs constants, et des pointeurs sur constantes. On utilise ici le mot-clé *const*.

Inférence de type en C++



Slide 26

Le mécanisme d'inférence de type est présent dans certains langages de programmation fonctionnelle.
C'est un mécanisme qui permet à un compilateur ou un interpréteur de rechercher automatiquement les types associés à des expressions, sans qu'ils soient indiqués explicitement dans le code source.
Il s'agit pour le compilateur ou l'interpréteur de trouver le type le plus général que puisse prendre l'expression.

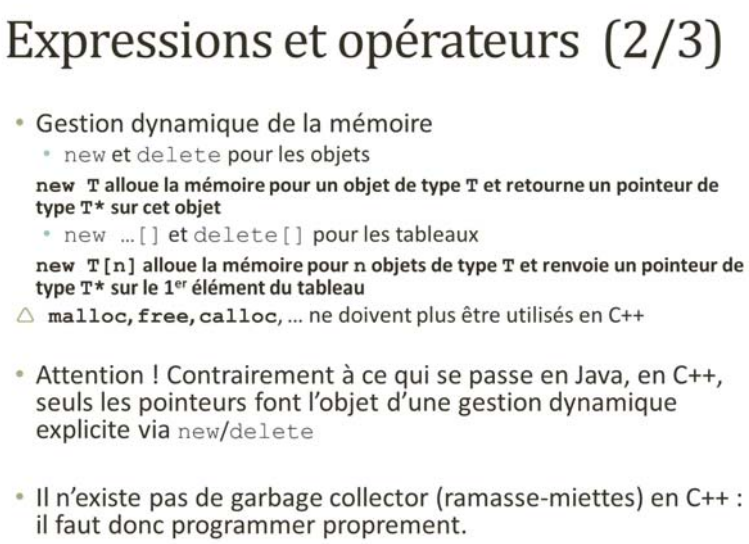
Les avantages de l'inférence de types sont multiples : le code source est plus aéré, le développeur n'a pas à se soucier de retenir les noms de types, l'interpréteur fournit un moyen au développeur de vérifier (en partie) le code qu'il a écrit et enfin, le programme est peu modifié en cas de changement de structure de données. Les expressions restent les plus générales possibles.

L'inférence de types va de pair avec le polymorphisme, puisque le compilateur génère des types abstraits au besoin.

Enfin, le compilateur, lui, dispose de toutes les informations de type, il y a un typage fort et statique, ce qui lui permet de produire du code plus sûr et plus efficace.

Expressions et opérateurs en C++

Slide 27



Slide 28

New et delete sont de nouveaux opérateurs pour les pointeurs, par rapport au C.
Attention à avoir une utilisation cohérente de new et delete : ne pas utiliser delete[] pour un pointeur alloué avec new (donc pas un tableau) ni inversement delete pour un pointeur alloué avec new[] (ne libèrera pas proprement la mémoire).

Ne pas réutiliser un pointeur qui a été désalloué : il faut de nouveau lui attribuer de la mémoire avec new.

Delete : on désalloue et l'emplacement réservé peut désormais être utilisé par le programme.
NB : on évite d'utiliser delete sur des types fondamentaux (char, int, float, double, etc.)

Allocation de mémoire : new type
int *p_t;
p_t = **new int**;// alloue l'espace pour un entier et met l'adresse de cet espace dans p_t
*p_t = 20 ; // OK. Il faudra penser à désallouer l'espace mémoire en fin de programme

Désallocation de mémoire : delete nomVar
int *p_t = **new int**(20); //pt pointe vers une zone mémoire qu'on vient d'allouer et dans laquelle on a placé la valeur 20
delete p_t; // on désalloue, on libère la zone mémoire pointée par p_t
=> Attention : aucun effet sur p_t !! le pointeur p_t contient toujours l'adresse de l'emplacement mémoire qui contenait 20, mais cet emplacement ne contient plus rien. On peut placer une nouvelle adresse dans p_t.

Danger : on ne peut utiliser delete que si l'on a auparavant utilisé new
int x = 1;
int* p_x = &x; // ici, le new n'est pas nécessaire, x a déjà été déclaré et a un espace mémoire permettant de stocker l'adresse de x
delete p; // DANGER : p n'a pas été alloué par new, on ne doit pas utiliser delete.

ATTENTION A LA DIFFERENCE D'USAGE NEW/DELETE ENTRE C++ ET JAVA !

Expressions en C++ (3/3)

- C++ vs Java
  - new est utilisé pour l'instanciation d'objet d'une classe en Java, ici il sert aux pointeurs
  - Gestion dynamique de la mémoire en C++
  - Le Garbage collector (ramasse-miettes) gère la mémoire de Java (durée de vie des variables)… ça n'existe pas en C++ (sûrement dans la prochaine norme)
- Allocation statique (à la compilation) :

```
int tableau[10] ; // mémoire fixée pour
toute l'exécution du programme.
```

- Allocation dynamique (à l'exécution) :

```
int* tableau = new int[10] ;
```

slide 29

Allocation dynamique vs allocation statique
L'allocation dynamique se fait pendant l'exécution du programme, ce qui signifie que l'espace alloué dynamiquement ne se trouve pas déjà dans le fichier exécutable du programme lorsque le

système d'exploitation charge le programme en mémoire pour l'exécuter; la demande d'allocation d'espace au système d'exploitation est faite durant l'exécution du programme.

Intermède…

- Allocation dynamique de la mémoire : l'allocation se fait au cours de l'exécution du programme (elle est demandée au système pendant l'exécution) au lieu d'être déjà allouée.
- Allocation statique de la mémoire : la mémoire est allouée à la compilation, fixée pour toute la durée de l'exécution.

Les deux ont des avantages, et des inconvénients… en C++, lorsqu'on fait de l'allocation dynamique, on doit penser à désallouer la mémoire avant la fin du programme.

Slide 30

Les conversions de types (autre forme de gestion dynamique)

Conversions de types

- Syntaxe : `type (expr)`
- `const_cast` : seul le qualificateur const change
- `static_cast` : conversion entre des types reliés (pointeur en pointeur, flottant en entier,…)
- `dynamic_cast` : mécanisme de type dynamique
- `reinterpret_cast` : libre de toute conversion (à utiliser avec prudence)
- Remarque : la syntaxe C fonctionne encore pour les conversions
- Attention : le compilateur ne fait pas de vérification pour les conversions

Slide 31

Attention à la syntaxe, qui est l'inverse de celle du C, pour les conversions (mais la syntaxe C fonctionne, en principe).
Il y a plusieurs opérateurs de conversion, ce qui est plus clair qu'en C et permet de palier le fait que le compilateur ne fait aucune vérification pour les conversions.

Forme des instructions en C++

Les instructions en C++ (1/3)

- Instructions conditionnelles
  - `if (cond) instr`
  - `if (cond) instr else instr`
  - `switch (cond) {`
    `case const1 : instr;`
    `case const2 : instr;`
    `break; // facultatif`
    `default : instr; // facultatif`
    `}`
  - Remarque : il est possible de déclarer une variable dans une condition (limite sa portée)

slide 32

L'instruction break permet de quitter le switch sans passer par l'évaluation des autres cas.

Remarque sur la portée des variables dans les instructions conditionnelles

```
if (toto==f(3)) {
        double d=3;
        //…
}
 //d n'est pas accessible
```

Les instructions en C++ (2/3)

- Instructions de boucles
  - `while (cond) instr`
  - `do instr while (expr)`
  - `for (expr1;cond;expr2) instr`
    - `expr1` : initialisation
    - `cond` : condition d'arrêt
    - `expr2` : exécutée à chaque fin de boucle
    - 3 parties facultatives, mais 2 ; obligatoires
- Remarque : il est possible de déclarer une variable dans la partie initialisation de la boucle for (limite sa portée)

Slide 32

Les instructions en C++ (3/3)

- Instructions `break` et `continue` permettent de sortir d'une boucle
  - **`break` : sortie immédiate d'une boucle ou d'un case**
  - **`continue` : passe directement à l'itération suivante de la boucle (incrémentation ou test)**
- Instruction `goto` et étiquettes
  - Une utilisation possible de `goto` : pour sortir de plusieurs boucles imbriquées
  - À éviter en pratique (programmes confus)

slide 33

Même remarque que ci-dessus pour la portée des variables dans les boucles...

Portée des variables

Notion de portée 1/2

- Portée d'une variable : partie du programme où le nom de la variable est accessible
- Variable globale cachée/confidentielle

```
static int a;
void main() {
...
}
```

=> a est inaccessible à l'extérieur du fichier source où elle a été définie

Slide 35

Notion de portée 2/2

- L'opérateur ::
  - :: est un opérateur unaire si portée globale (différenciation variable globale/variable locale)
  - :: est un opérateur binaire si portée de classe (différenciation des variables issues de différentes classes ou de différents espaces de nommage)

Slide 36

l'opérateur :: permet de résoudre les problèmes d'ambiguïté de nom liés à la portée des variables, qu'il s'agisse de portée globale (dans et hors d'une boucle, par exemple) ou bien de portée de classe ou encore de méthodes ou variables issues d'espaces de nommage différents.

Exemples d'utilisation de ::

```
#include <iostream>
using namespace std;
int i; // standard : initialisation à 0 (0?0 si double)
int main() {
    int i = 12;
    ::i = i + ::i; // (A)
    cout << i << " " << ::i << endl; // (B)
    return 0;
}
```

=> dans la ligne A, on différencie les deux variables de même nom grâce à l'opérateur ::
=> en ligne B, l'affichage donne 12 12 (l'opérateur :: désigne la variable la plus externe)
Initialisation statique à zéro des variables globales par le compilateur, emplacement mémoire réservé et connu de l'éditeur de lien
i désigne le i interne à la fonction et ::i désigne la variable globale.

```
void f() {
    int n; // n est accessible partout dans le bloc f
    while () {
        int p; // p n'est connu que du bloc while
        int n; // n masque la variable n déclarée plus haut
                // Que donne ::n ? On ne sait pas : ce code ne montre pas si une variable globale a été définie ou non.
        }
}
```

Les fonctions en C++

Les fonctions (1/4)

- La déclaration précise
  - le nom de la fonction
  - le type de retour de la fonction (possibilité d'inférence de type)
  - le nombre et le type des arguments

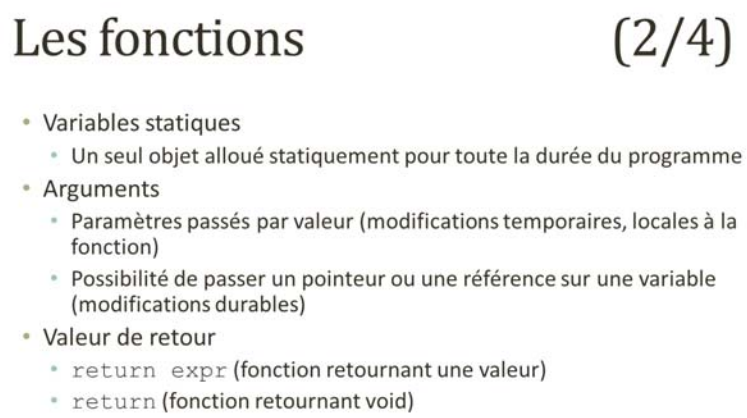
```
void f();
  //fonction sans valeur de retour et sans argument
char *g(const char*, int); //deux arguments
char *strcpy(char *dest, const char *source);
    //deux arguments nommés
```

- La définition reprend la déclaration et y ajoute le corps de la fonction

NB : pour pouvoir être appelée, une fonction doit avoir été définie au préalable.

Slide 37

Pour pouvoir être appelée, une fonction doit avoir été définie au préalable. Les arguments ne sont pas obligatoirement nommés dans la déclaration de la fonction, mais doivent l'être dans sa définition.
Le prototype d'une fonction donne son en-tête avec le type des paramètre. Ne donner aux utilisateurs et utilisatrices d'un code que le prototype des fonctions permet l'encapsulation : la définition de la fonction reste une « boite noire » et toute modification du corps de la fonction sera transparente.



Slide 38

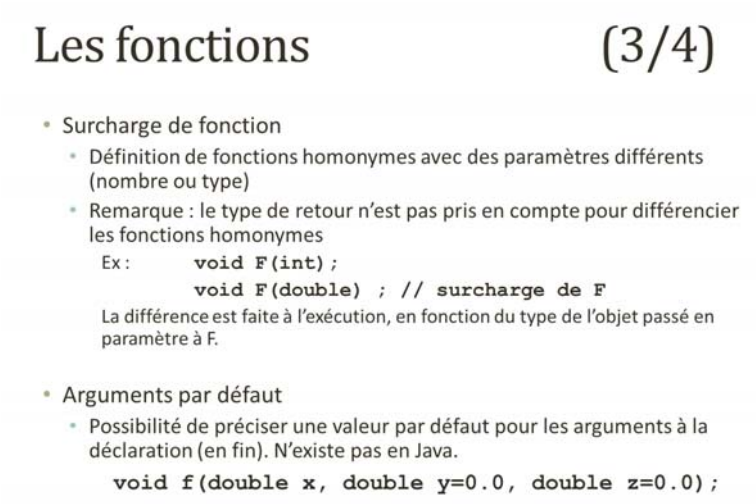
Pour une modification durable (non locale à la fonction) des arguments d'une fonction, il faut passer leur adresse à la fonction *via* un pointeur ou une référence.

NB : toute fonction déclarée comme retournant autre chose que void doit obligatoirement renvoyer une valeur.

Cas particulier des fonctions récursives :

```
long fact(int n) {
    if (n > 1) {
       return (fact(n-1)*n);
    } else {
      return (1);
    }
}
```

=> une variable locale est réservée et empilée pour la valeur de retour, aucune libération jusqu'à la première instruction return qui lance le dépilement en permettant un retour entier (passage dans la condition d'arrêt).

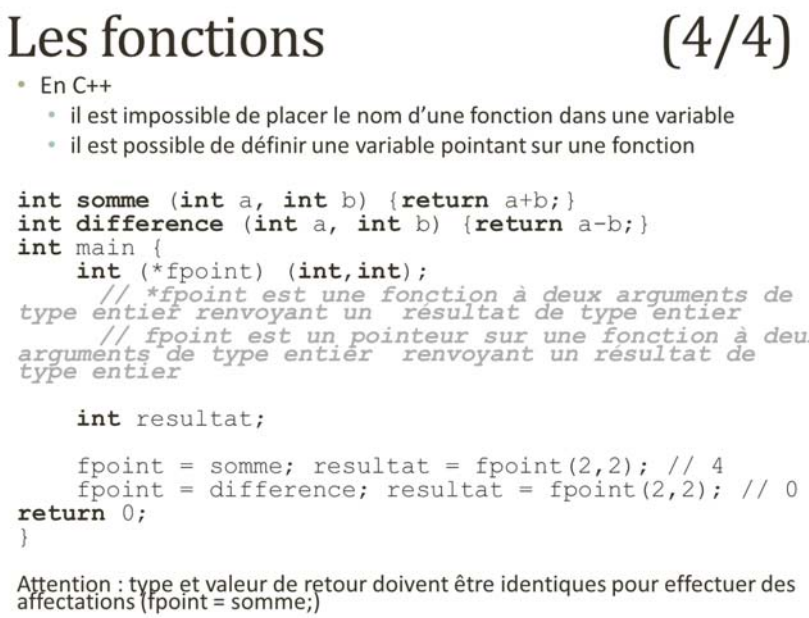


Slide 39

Exemple de surcharge :
void F(int);
void F(double);
La différence est faite à l'exécution, en fonction du type de l'objet ou de la variable passée en paramètre à la fonction F.

On peut également spécifier des valeurs par défaut pour les variables dans l'en-tête d'une fonction. Ces dernières doivent se trouver en fin de liste, le compilateur utilisant alors les valeurs passées en paramètre pour initialiser les variables dépourvues valeur par défaut, dans l'ordre de leur apparition, puis pour remplacer les valeurs par défaut, le cas échéant.

Fonctions en tant qu'arguments



Les fonctions (4/4)

- En C++
  - il est impossible de placer le nom d'une fonction dans une variable
  - il est possible de définir une variable pointant sur une fonction

```
int somme (int a, int b) {return a+b;}
int difference (int a, int b) {return a-b;}
int main {
    int (*fpoint) (int,int);
      // *fpoint est une fonction à deux arguments de
type entier renvoyant un  résultat de type entier
      // fpoint est un pointeur sur une fonction à deux
arguments de type entier  renvoyant un résultat de
type entier

    int resultat;

    fpoint = somme; resultat = fpoint(2,2); // 4
    fpoint = difference; resultat = fpoint(2,2); // 0
return 0;
}
```

Attention : type et valeur de retour doivent être identiques pour effectuer des affectations (fpoint = somme;)

Slide 40

int somme (**int** a, **int** b) {**return** a+b;}
int difference (**int** a, **int** b) {**return** a-b;}
int main {
  int (*fpoint) (**int**,**int**);
    // *fpoint est une fonction à deux arguments de type entier renvoyant un résultat de type entier
    // fpoint est un pointeur sur une fontion à deux arguments de type entier renvoyant un résultat de type entier
  int resultat;

  fpoint = somme; resultat = fpoint(2,2); // 4
  fpoint = difference; resultat = fpoint(2,2); // 0
return 0;
}
Attention : type et valeur de retour doivent être identiques pour effectuer des affectations (fpoint = somme;)

POINTEURS DE FONCTIONS EN ARGUMENTS

```
double integ ( double(*f)(double), double (d), int i ) {
    double resultat = (*f) (d) * i;
}

double f1 (double d) { ...}
int main() {
    double integ ( double(*f)(double), double, int) ;
    double f1(double);
    double resultat1 = integ (f1, 1.0, 10);
```

(inspiré de *Apprendre le C++ - Claude Delannoy*)

(*f)(double) est de type double
(*f) est une fonction prenant un double en argument et renvoyant un double
f est un pointeur sur une fonction prenant un double en argument et renvoyant un double
Utilisation : (*f) (d)

*fpoint est une fonction à deux arguments de type entier renvoyant un résultat de type entier

fpoint est un pointeur sur une fontion à deux arguments de type entier renvoyant un résultat de type entier

Type et valeur de retour doivent être identiques

Pointeurs de fonctions comme arguments

```
double integ ( double(*f)(double), double (d),
int i ) {
     double resultat = (*f) (d) * i;
}

double f1 (double d) { ...}
int main() {
     double integ ( double(*f)(double), double,
int) ;
     double f1(double);
     double resultat1 = integ (f1, 1.0, 10);
}
```

(inspiré de *Apprendre le C++ - Claude Delannoy*)

(*f)(double) est de type double
(*f) est une fonction prenant un double en argument et renvoyant un double
f est un pointeur sur une fonction prenant un double en argument et renvoyant un double
Utilisation : (*f) (d)

Slide 41