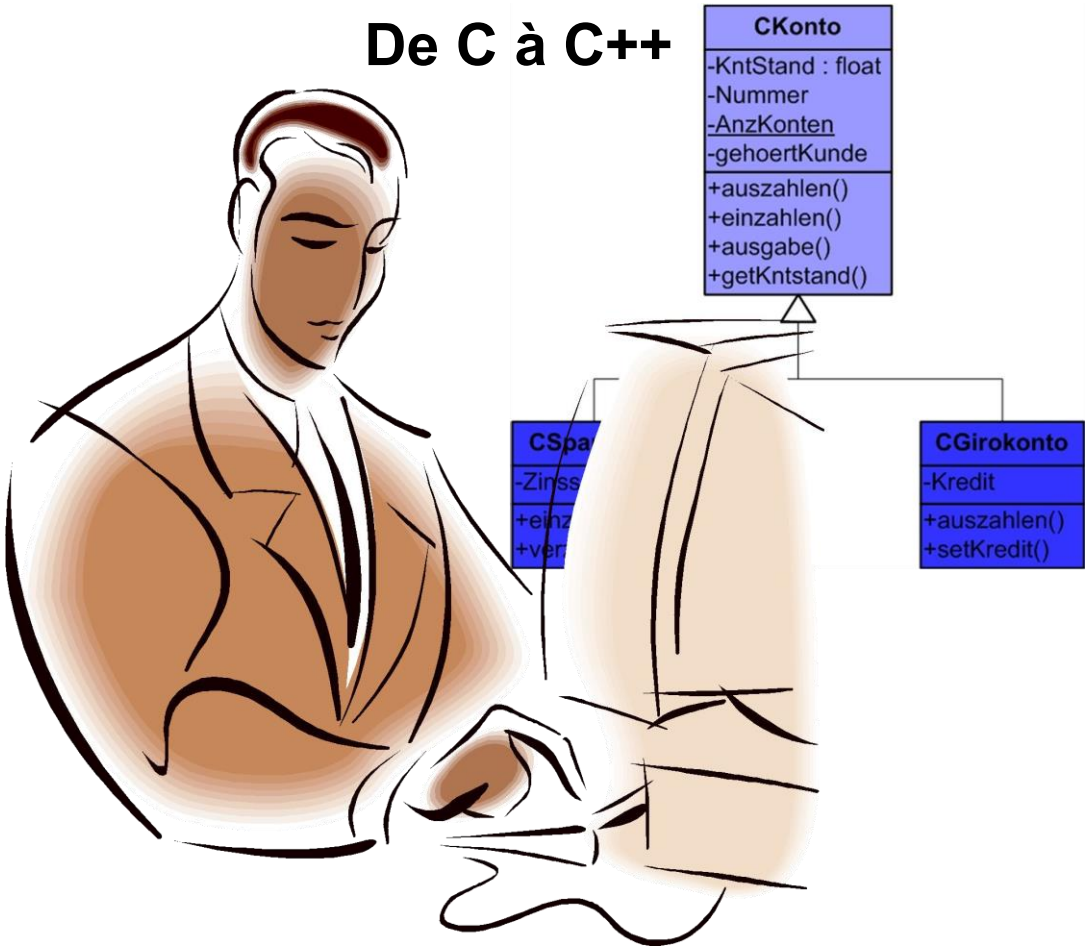
Programmation orientée objet



Objectifs d'apprentissage I

- Vous connaissez les objectifs de conception du langage de programmation C++.
- Vous savez pourquoi les espaces de noms sont nécessaires et comment les utiliser.
- Vous pouvez utiliser les courants d'entrée/sortie par défaut **`cin`** et **`cout`**
  pour une entrée/sortie flexible des données.
- Vous connaissez quelques extensions simples du C++ comme les types de données `bool` et `string`, la définition de `variables enum` et `struct` ainsi que la déclaration locale de variables.
- Vous pouvez utiliser les opérateurs **`new`** et **`delete`** pour gérer la mémoire de manière dynamique.

Objectifs d'apprentissage II

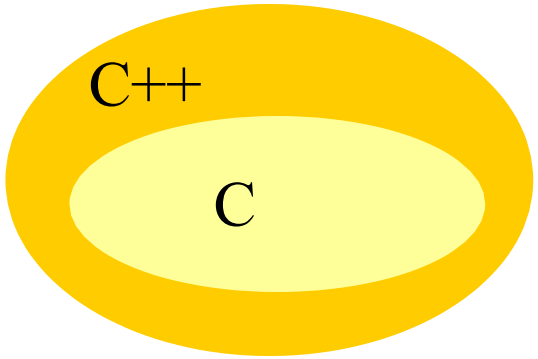
- Ils savent ce que sont **les fonctions en ligne** et comment et quand les utiliser.
- Vous savez comment utiliser les paramètres de fonction avec **des valeurs par défaut.**
  de l'entreprise.
- Ils savent que les fonctions peuvent être **surchargées** et comment elles peuvent l'être.
- Vous savez ce que l'on entend par **variable de référence** et comment celle-ci peut être utilisée, notamment dans le contexte des appels de fonction.
- Vous savez comment utiliser les fonctions C dans les programmes C++ et inversement.

Agenda

1. Introduction
2. Espaces de noms
3. Flux d'entrée/sortie de données
4. Extensions C++ simples
5. Extensions pour les fonctions
    1. Fonctions en ligne
    2. Valeurs par défaut des paramètres de fonction
    3. Surcharge de fonctions
    4. Fonctions C dans les programmes C++
    5. Références

Comparaison entre C et C++ I

- C++ est un véritable **super-ensemble** de C
- C est un langage "proche de la machine
- C++ est un langage "proche du problème



- **Points forts de C**
  - fournit un code **efficace**
  - permet une programmation **proche du système**
  - **Les algorithmes** peuvent être **formulés de manière concise** et élégante
  - Fonctionne partout et sur n'importe quel matériel

Comparaison de C et C++ II

- **Faiblesses du C** - *largement supprimées dans le C++*
  - **Types de données** essentiels **manquants** (Bool, String)
  - Modularisation uniquement au niveau du fichier ; **pas de** niveau d'accès
    Schéma d'autorisation pour l'utilisabilité des fonctions du module
  - **Pas de types de données définis par l'utilisateur dans le sens où** des opérateurs ou des opérations pourraient être mis à disposition pour ceux-ci, de sorte qu'ils puissent être utilisés de manière similaire aux types intégrés.
  - **Pas de** possibilité d'**initialisation automatique** des variables avec des valeurs définies par l'utilisateur (risque d'oubli)
  - **Pas de couplage** entre les **structures** et leurs **fonctions de manipulation**
    - Les éléments de structure peuvent être manipulés sans passer par les fonctions d'accès

Comparaison de C et C++ III

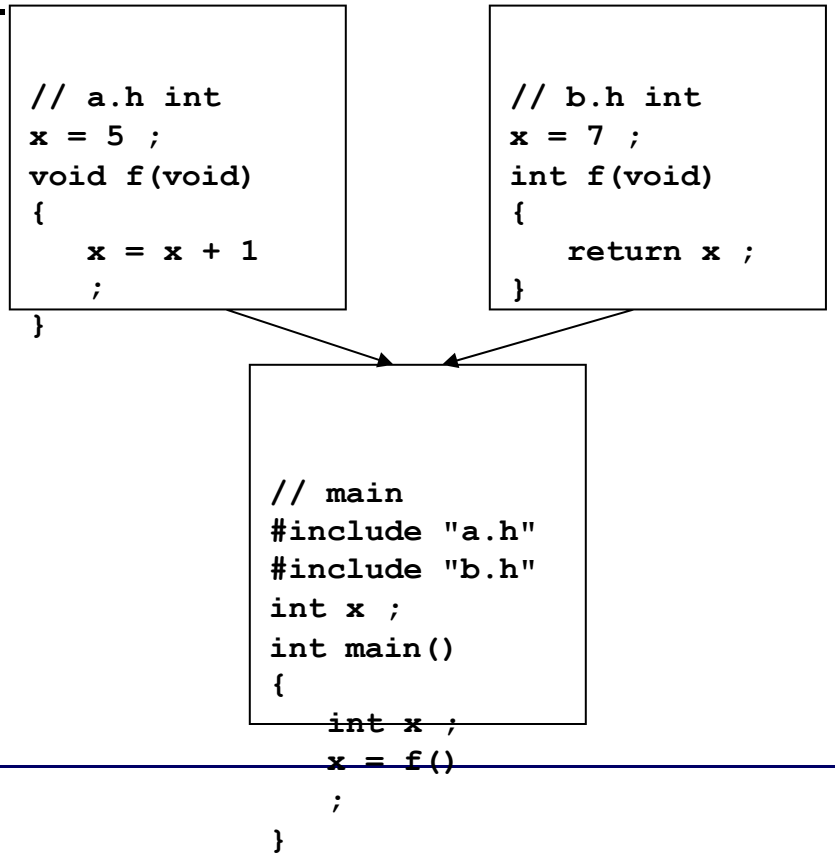
- **Faiblesses de C** (suite)
  - **Les fonctions similaires** ont besoin de **noms différents**
    - `float add_float(float, float) ;`
      `double add_double(double, double))`
  - Absence de vérification du type lors de l'utilisation de macros paramétrées
  - Le traitement ordonné des **situations d'erreur** n'est **pas pris en charge** par le langage de programmation

Agenda

1. Introduction
2. Espaces de noms
3. Flux d'entrée/sortie de données
4. Extensions C++ simples
5. Extensions pour les fonctions
    1. Fonctions en ligne
    2. Valeurs par défaut des paramètres de fonction
    3. Surcharge de fonctions
    4. Fonctions C dans les programmes C++
    5. Références

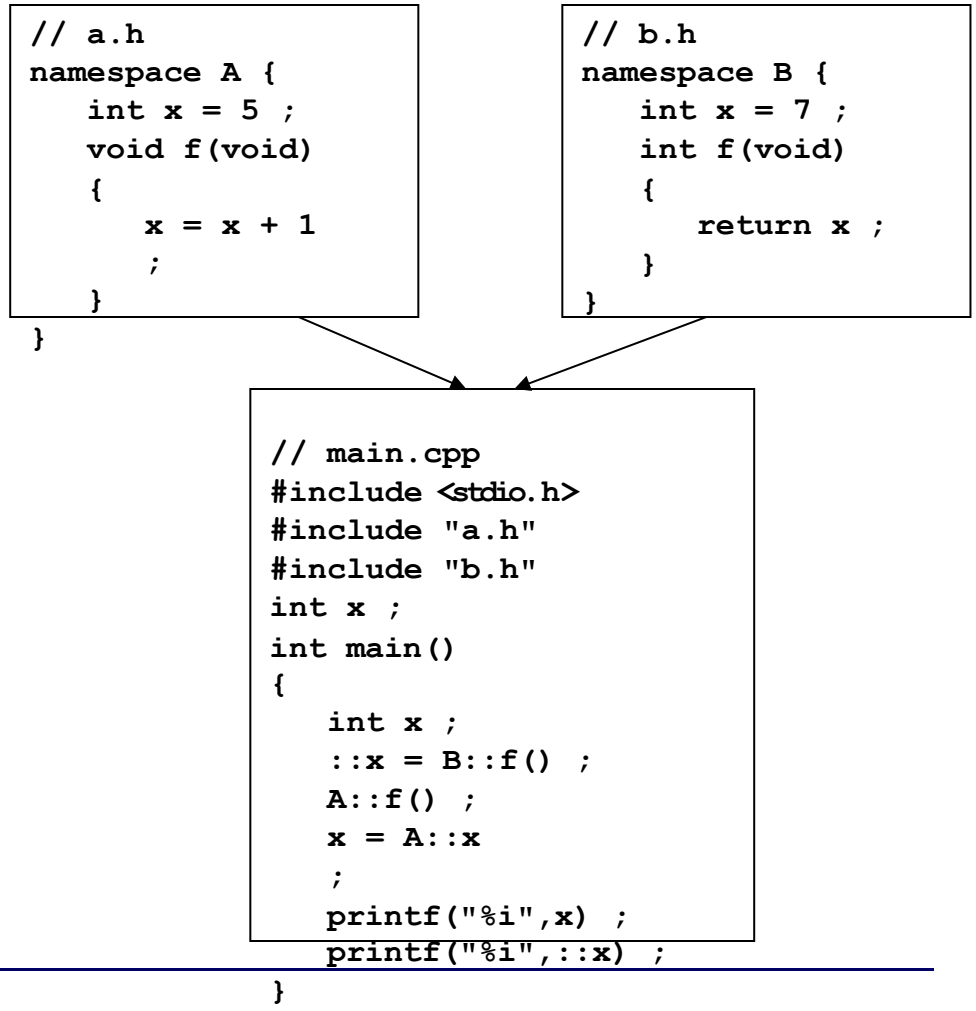
Espaces de noms I

- **Problème**
  - Dans les grands projets, il arrive souvent que des conflits de noms surviennent en raison de l'utilisation des mêmes identificateurs dans différents modules (fichiers ou bibliothèques).



Espaces de noms II

- **Solution**
  - Définition de différents **espaces de noms** (namespace) et utilisation de l'**opérateur de validité : :** pour résoudre les conflits de noms.



Espaces de noms III

- Pour tous les identificateurs au sein d'un fichier/d'une bibliothèque, on définit en quelque sorte un nom de famille qui permet ensuite de s'adresser aux objets de manière "pleinement qualifiante".
- L'**`instruction namespace`** donne à tous les identificateurs entre accolades le nom qui suit `namespace` comme **élément du nom.**
- Lors de l'**accès** aux variables, l'espace nom de la variable est précédé de l'**opérateur de validité** (**`A::x`**).
- **Si** l'**opérateur de validité est absent**, on accède aux variables locales à l'intérieur d'un bloc ou, s'il n'y a pas de variable locale avec le nom, à la variable globale.

Espaces de noms IV

- **S'il** existe des variables **locales** et **globales** de même nom, il est possible d'accéder à la variable globale à l'intérieur des blocs dans lesquels la variable (locale) de même nom a été définie, en la faisant précéder uniquement de l'opérateur de validité `: :` (sans indication d'espace de noms). Pour l'espace de noms de cette variable "normale", le compilateur utilise le nom **`unique`**.

Espaces de noms V

- **Clause d'utilisation**
  - Au lieu de qualifier chaque variable en indiquant l'espace de noms, la `clause using` permet d'**étendre** l'espace de noms standard `unique` à d'autres **espaces de noms**.
  - Les variables de ces espaces de noms peuvent alors être **utilisées directement. tant qu'ils sont clairs.**
  - Exemple important
    - **`using namespace std ;`** pour l'utilisation des bibliothèques standard C++.
    - Les fichiers d'en-tête de ces bibliothèques contiennent l'instruction **`namespace std{...}.`**
    - Ces fichiers d'en-tête n'ont pas l'extension **`.h`** (par exemple : <iostream>).

```
// a.h
namespace A {
   int x = 5 ;
   void f(void)
   {
      x = x + 1 ;
   }
}
```

```
// b.h
namespace B {
   int x = 7 ;
   int f(void)
   {
      return x ;
   }
}
```

```
// main.cpp

#include <stdio.h>
#include "a.h"
#include "b.h"
int x ;
using namespace B ;
int main()
{
   int x ;
   ::x = f()
   ;
   A::f() ;
   x = A::x
   ;
   printf("%i",x) ;
   printf("%i",::x) ;
}
```

Conflit : pas d'erreur de compilation, `B::x` est masqué

Agenda

1. Introduction
2. Espaces de noms
3. Flux d'entrée/sortie de données
4. Extensions C++ simples
5. Extensions pour les fonctions
    1. Fonctions en ligne
    2. Valeurs par défaut des paramètres de fonction
    3. Surcharge de fonctions
    4. Fonctions C dans les programmes C++
    5. Références

Flux pour l'entrée et la sortie de données

- Un **flux** est un **flux de données** d'une **source** vers un **Objectif** .
- En C++, les flux de données suivants, entre autres, sont mis à disposition via `#include <iostream>` (dont l'implémentation se fait dans différentes *classes* - nous y reviendrons plus tard)

Stream	Beschreibung	Bemerkung
cout	Standardausgabestrom	in der Regel der Bildschirm
cin	Standardeingabestrom	in der Regel die Tastatur
cerr	Standardfehlerausgabestrom	in der Regel der Bildschirm

- **Notez**
  - Le mécanisme de flux ne peut pas encore être entièrement compris ici (pour cela, il faut notamment comprendre la notion de classe)
- **`cin`** et **`cout`**, associés aux opérateurs d'entrée et de sortie correspondants, permettent de **se passer** de **`printf`**`()` et de **`scanf`**`()`

Sortie de données avec *cout*

- **`cout`** (console **output** ; `cout << "Hello World\N" ;`)
  - avec **`cout << "un, deux, trois" ;`** la **chaîne de caractères** **`"un, deux, trois"`** est envoyée au flux de sortie standard via l'**opérateur de sortie** <<.
  - **La source** du flux de sortie par défaut est ici le **programme**, son **La cible** est l'**écran**
  - dans une instruction d'édition, plusieurs blocs de données à éditer peuvent être concaténés à l'aide de **l'opérateur d'édition <<.**

```
cout << "La somme de "<< 7<<" et de " << 5 << " est "<< 7+5 ;
```

  - **Notez**
    - Contrairement à l'utilisation de `printf()`, il n'y a pas de code de formatage à indiquer ici
    - Le type des arguments est défini et la forme de la sortie est déterminée en fonction de cela

Cout et formatage de la sortie I

- **Les manipulateurs** listés ci-dessous sont annoncés par **`#include <iomanip>`** (après `#include <iostream>`) pour un formatage adapté

Manipulator	Beschreibung
`dec, hex, oct`	die auf den Manipulator folgende Zahl wird dezimal (dec) bzw. hexadezimal (hex) bzw. octal (oct) ausgegeben `cout << "hexadezimal: " << hex << 255 << endl;`
`setw(n)`	für genau die nächste Ausgabe wird eine Ausgabebreite von n Positionen festgelegt; Ausgabe erfolgt rechtsbündig `cout << setw(8) << 4711 << endl;`

`setfill(z)`	nimmt das auszugebende Datenelement weniger Positionen als über setw angegeben ein, so wird das Zeichen z für die restlichen Positionen als Füllzeichen eingesetzt
	`cout << setfill('.');`
	`cout << setw(8) << "test\n";`

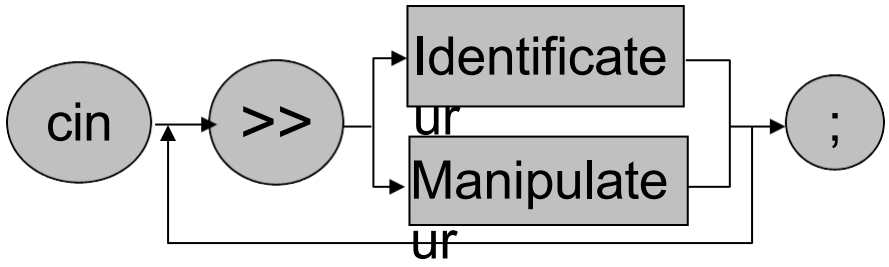
`setprecision(n)`	Festlegung der Anzahl n der Nachkommastellen aller folgenden auszugebenden Fließkommazahlen; letzte Stelle wird gerundet `cout << setprecision(3) << 4711.12345 << endl;`
`endl`	Erzeugung eines Zeilenumbruchs (Alternative zu '\n')
`flush`	Ausgabepuffer werden geleert, d.h. ggfs. zwischengespeicherte Ausgaben werden jetzt in jedem Fall auf das Ausgabeziel geschrieben

Cout et formatage de la sortie II

Manipulateur	Description
setiosflags(ios::*drapeau*)	permet de définir des paramètres de formatage avec les valeurs possibles suivantes pour *flag* (signification entre parenthèses) : left (sortie alignée à gauche) right (sortie alignée à droite) internal (signe à gauche, valeur à droite) dec(sortie décimale) oct (sortie octale) hex (sortie hexadécimale) showbase (ajouter '0x' aux nombres hexadécimaux et 'O' aux nombres octaux showpoint (insérer des zéros après la virgule en fonction de la précision uppercase (la sortie est en majuscules) showpos (les nombres positifs sont précédés de '+' scientific (notation exponentielle) fixed (écriture décimale) **cout << setiosflags(ios::left) ;**
resetiosflags(ios::drapeau)	Réinitialisation de l'indicateur spécifié

Saisie de données avec *cin* I

- **`cin`** (**entrée** console)
  - avec **`cin >> x ;`** la **saisie au clavier** est "poussée" ou déposée dans la variable `x` par l'**opérateur de saisie `>>`**, ***conformément au type*** (!)
  - **La source** du flux d'entrée standard est ici le **clavier** ; son **L'objectif** est le **programme**
  - **Syntaxe**



  - **Exemple**

```
int num ;
char c ;
cin >> hex >> num >> c ;
// Saisir d'abord un nombre puis une lettre séparée par un espace blanc
```

Saisie de données avec *cin* II

- L'opérateur de saisie >> **ignore les caractères d'espace blanc** (espace, tabulation, retour) lors de la lecture ; la saisie se termine généralement par un retour.
- Les caractères sont **convertis** dans le type souhaité (type de variable) lors de la saisie jusqu'à ce qu'un caractère qui ne peut pas être converti apparaisse :

```
int i ;
cin >> i
;
```

  Saisie de `34MaintenantPas d`'affectation à `i` de la valeur `34`
- Lors de la saisie de **nombres à virgule flottante**, le **point décimal**
  ou autorise l'utilisation de la **notation exponentielle**

```
float x, y ;
cin >> x >> y
;
```

  Saisie de `47.11 3e2` affecte `x` à `47.11` et `y` à `300`

Agenda

1. Introduction
2. Espaces de noms
3. Flux d'entrée/sortie de données
4. Extensions C++ simples
5. Extensions pour les fonctions
    1. Fonctions en ligne
    2. Valeurs par défaut des paramètres de fonction
    3. Surcharge de fonctions
    4. Fonctions C dans les programmes C++
    5. Références

Type de données *bool*

- En C++, le type de données **`bool`**, qui peut contenir les valeurs de vérité **`true`** et **`false, est un`** **autre type de données élémentaire.**
- Exemple d'utilisation

```
bool erg = true ;
int a, b ;
cout << "Deux entiers, s'il vous
plaît... " ; cin << a << b ;
erg = a > b ;
if( erg == true)....
```

- **`erg = true ;`** correspond à l'affectation **`erg = 1 ;`** `true` et `false` représentent donc les valeurs numériques 1 et 0
- **`cout << erg ;`** entraînerait la sortie d'un 1 ou d'un 0
  - mais **`cout << setiosflags(ios::boolalpha) << erg ;`** affiche `true`.

Type de données *string*

- Le type de données **`string`** n'est **pas un type de données élémentaire** en C++.
  mais une **classe (**déclaration de classe via `#include <string>` inclure)
- La manipulation du `type de données string` ne diffère pas de celle d'un type de données élémentaire.
- Les variables ou objets de type chaîne de caractères prennent en compte **les chaînes de caractères.**
  sur
- Exemple d'utilisation

```
chaîne de caractères s0, s1 =
"Bon" ; chaîne de caractères
s2 = " Jour
string s3 = s1 + s2 ;          //s3 représente "Bonjour"s3 =
s3 + " Mme Müller" ;           //s3 : "Bonjour Mme Müller"
cout << s3 << "Entrée : " ;
cin >> s0 ;
```

```
if( s0 == s2)....
```

Noms de type pour les enums et les structs

- En C++, il est possible de définir **des variables** pour les types d'énumération ou de structure **sans spécifier** les mots-clés **`enum`** ou **`struct`.**
  - ```
    enum NoteT {bon, passable, suffisant} ;
    NoteT n ;
    ```
  - ```
    struct PointT{
       int x, y ;
    } ;
    PointT p ;
    ```
- Il n'est donc plus nécessaire d'utiliser l'**`opérateur typedef`** pour attribuer un nom de type de données abrégé.

Constantes

- Au lieu de définir les constantes avec `#define`, le modificateur **`const`** peut être placé avant la définition de la variable en C++.
- Forme générale :
  `const <typ> <Notifiant> = <valeur> ;`
- exemple :
  `const int MaxEtudiants = 100 ;`
- Avantage : le compilateur vérifie les domaines de type et de validité
- **Pointeur constant vs. pointeur sur constante**

```
const int *p ;          // Pointeur sur la constante
:
                        // *p n'est pas modifiable
int *const p ;          // pointeur constant sur
                        // une variable de type
                        entier :
                        // p lui-même n'est pas
```

```
// modifiable
```

Localisation des définitions de variables

- **Les définitions de variables** peuvent
  - à **n'importe quel endroit** du programme où se trouve également une **instruction**
    peut se tenir,
  - dans **les initialisations d'instructions for** et
  - en **conditions**

  peuvent être effectuées.
- Exemples :
  - ```
    int rest(int a, int b)
    {
        if ( !(a % b)) return 0
        ; int r = a % b ;
        retour r ;
    }
    ```
  - `for (int i=1 ; i<102 ; i++) {...}`
  - `if (int r = rest(47,6)) {...} ;`

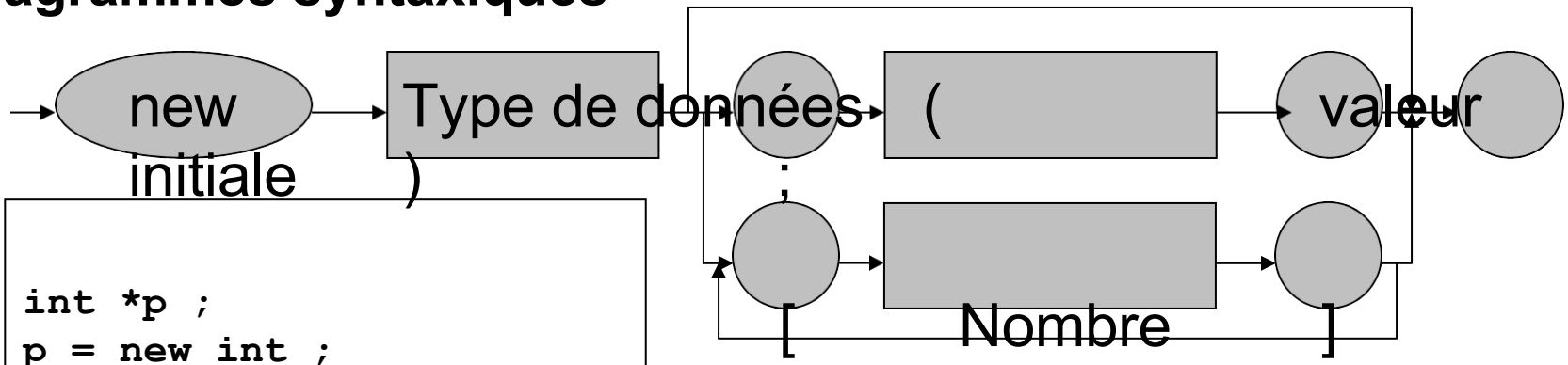

    ```

Structures de données dynamiques avec *new* et *delete*

- **La gestion dynamique de la mémoire** en C se faisait à l'aide des **fonctions** **`malloc`**`()` et **`free`**`()`**.**
  - ```
    int *z = (int *) malloc (10 * sizeof(int)) ;
    ....
    free(z) ;
    ```
- en C++, on peut utiliser plus confortablement les **opérateurs** **`new`** et
  être gérés **`de manière plus souple`**
- Les opérations de casting et de sizeof sont notamment supprimées
  - ```
    int *z = new int[10]
                    ;
    ...
    delete[] z ;
    ```

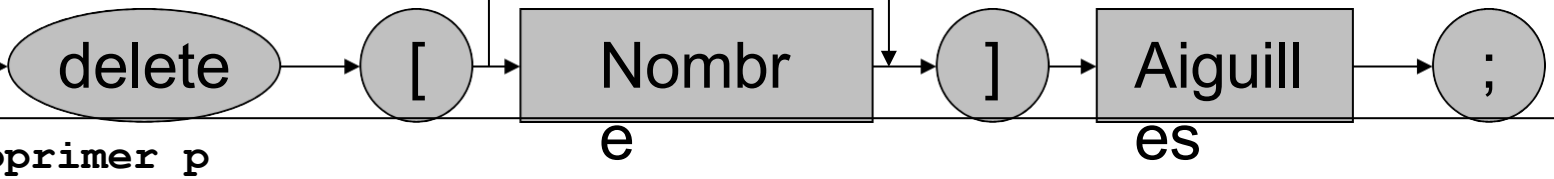
new et *delete* : syntaxe / autres

- **Diagrammes syntaxiques**




```
int *p ;
p = new int ;
int *z = new int(7) ;
char *za = new char[10]
;
```

- **`new`** renvoie le **pointeur zéro** si la mémoire est mise à disposition **a échoué**




```
supprimer p
;                  // utiliser [] si [] a été utilisé dans new
delete[] za        // sinon, suppression incomplète
;
```

Agenda

1. Introduction
2. Espaces de noms
3. Flux d'entrée/sortie de données
4. Extensions C++ simples
5. Extensions pour les fonctions
    1. Fonctions en ligne
    2. Valeurs par défaut des paramètres de fonction
    3. Surcharge de fonctions
    4. Fonctions C dans les programmes C++
    5. Références

Fonctions en ligne I

- "Fonctions "normales
  - **La fonction** est **traduite une fois par** le compilateur
  - Chaque **appel de fonction** correspond alors à un **saut** vers le **code de fonction** correspondant. L'adresse de programme actuelle (adresse de retour) et les variables locales de la fonction sont poussées dans la pile et, au retour de la fonction, le processus inverse reprend.
    Les appels de fonction sont **coûteux** du point de vue de l'exécution
- Le mot-clé `inline` placé devant la définition de la fonction indique au compilateur d'**insérer** le **code complet de la fonction** à chaque **point d'appel** de la fonction.
- Si une fonction en ligne `f` est appelée `n` fois, le code complet de la fonction `f` se trouve à `n` endroits dans le programme
  Le code des programmes devient **plus efficace**, mais aussi **plus long**

Fonctions en ligne II

- **Syntaxe**
  - La définition de fonction traditionnelle est accompagnée du mot-clé précédé de **`inline`**
  - Exemple :
    - ```
      en ligne int carré(int x)
      {
        return x*x ;
      }
      ```
- **Notez**
  - **`inline`** n'est **qu'**une **suggestion** au compilateur de traiter la fonction comme inline. En fin de compte, c'est le compilateur lui-même qui vérifie et décide si le remplacement a effectivement lieu.
  - Inversement, le compilateur ne transforme pas une fonction en fonction en ligne si cela n'a pas été proposé.
      ```

Fonctions en ligne III

- Une **fonction en ligne** est particulièrement adaptée lorsque
  - la priorité est donnée à une **efficacité élevée en termes de temps de fonctionnement**
  - la fonction est **appelée très fréquemment**
  - la fonction ne comporte qu'**une** ou **deux lignes**

  ... c'est-à-dire que le **travail administratif** à effectuer par l'appel de la fonction (données locales sur la pile, branchement, retour, réinitialisation du pointeur de pile,...) est **élevé** par rapport au travail réel dans la fonction.
- **Avantages** par rapport aux **macros paramétrées**
  - **Test de type** lors du passage des paramètres et du retour des résultats
  - Aucun risque de négligence dans l'agrafage

Valeurs par défaut des paramètres de fonction I

- Lors de la **définition d'une fonction**, il est possible d'attribuer une **valeur par défaut** à certains paramètres de transfert.
- Il est alors possible de renoncer à l'indication de ces paramètres lors de l'appel de la fonction :
  - **si** les **paramètres manquent** lors de l'appel, les **valeurs par défaut sont prises en compte**
  - si les paramètres sont **indiqués**, ils sont **utilisés comme d'habitude**.

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

void output(int nombre, int modus = 0)
{
  if(modus == 0)
    cout << dec << nombre << endl ;
  else
    cout << hex << nombre << endl ;
}


int main()
{
   sortie(255) ;
   sortie(255,1) ;
}
```

édition :

```
255
ff
```

Valeurs par défaut des paramètres de fonction II

- **Syntaxe** et **directives**
  - Dans la définition de fonction par ailleurs traditionnelle, la valeur (constante !) est attribuée au paramètre qui doit recevoir une **valeur par défaut** via l'**opérateur d'affectation**
  - Un paramètre **avec une** valeur par défaut ne peut pas être suivi **d'un** paramètre **sans valeur par défaut.**
    Suivre la valeur par défaut
    ```
    int test(int a, int b = 3, int c)...
    ```
  - Lors de l'**appel de la fonction**, les valeurs des paramètres par défaut peuvent être
    être omis **de droite à gauche**
    - Exemple
      ```
      int f(int a=0, int b=0, int c=0)...
      ```
      Appels possibles : `f() ; f(a) ; f(a,b) ; f(a,b,c) ;`

Surcharge de fonctions I

- **Les fonctions C++** sont identifiées et donc différenciées non seulement par leur **nom**, mais aussi par le **nombre** et **le type** de leurs **paramètres**.
- Ainsi, en C++, **différentes fonctions** peuvent avoir le **même nom** si elles diffèrent par le nombre ou le type de leurs paramètres - la lisibilité du texte source peut ainsi être nettement améliorée.
  - `int somme(int a, int b)` et `int summe(int a, int b, int c)` désignent différentes fonctions, chacune nommée **`summe`**.
- **Note : Le type de retour n'**est **pas pertinent ici**, c'est-à-dire qu'il ne contribue *pas* à la distinction des fonctions.
  - `double quotient(int numérateur, int dénominateur)` et `int quotient(int numérateur, int dénominateur)` ne sont pas utilisables ensemble, car ils ne peuvent pas être distingués lors de l'appel.

Surcharge de fonctions II

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

int valeur moyenne(int a, int b, int c)
{
     return ((a + b + c)/3.0 + 0.5) ;
}

double moyenne(double a, double b, double c)
{
     return (a + b + c)/3 ;
}

int main()
{
    cout << valeur moyenne(1,2,2) <<
    endl ; cout << valeur
    moyenne(1.0,2.0,2.0) ;
}
```

édition :

```
2
1.666667
```

Mélange de fichiers sources C et C++ I

- Les compilateurs C++ génèrent des informations dans les fichiers d'objets pour la traduction des fonctions, informations que les compilateurs C ne saisissent **pas** ;
- Ex : `int getMax(int, int) ;`
  - est traduite par le **compilateur C** en `_getMax, sans tenir` compte des types de paramètres de transfert - en C, les fonctions étaient clairement définies par leur nom.
  - est traduit par le **compilateur C++** en `_getMax_int_int` en tenant compte de l'information de type
- Pour utiliser la fonction **`getMax()`** d'une bibliothèque C ou d'un fichier objet C dans un programme C++, il faut attirer l'attention du compilateur C++ sur la **mise à disposition de tiers** comme suit
  - `"C" int getMax(int, int) ;`

Mélange de fichiers sources C et C++ II

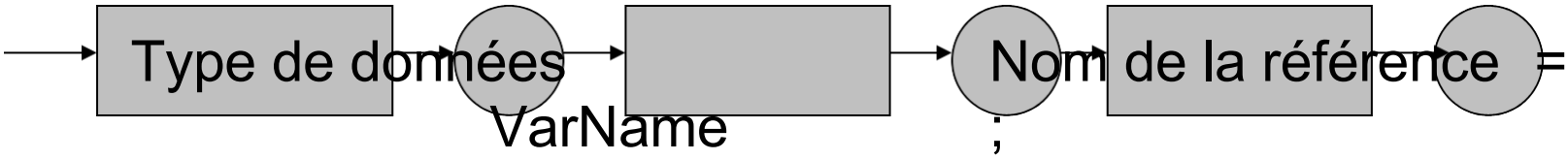
- Si vous souhaitez utiliser **plusieurs** fonctions "**mises à disposition par des tiers**", vous pouvez l'indiquer à l'aide d'un bloc
- Exemple
  - ```
    externe "C" {
          char strcpy(char *, const char*) ;
          char strlen(const char *) ;
        }
    ```
- Pour créer une **fonction C++ dans** un **programme C**, par ex. `int f(int)`, à **utiliser**
  - Commencez par écrire votre fonction C++ **`f`**
  - Dans le même fichier source C++, insérez ensuite le code suivant :
    ```
    externe "C" {
                  int f(int) ;
                }
    ```
  - Ensuite, vous pouvez utiliser **`f`** dans un programme C
    ```

Références I

- En C++, **les variables ne peuvent** plus seulement être transmises aux fonctions sous forme de copie (call-by-value), mais aussi sous forme de **référence** (**call-by-reference**).
- Une **référence** est un **alias** pour une variable, c'est-à-dire qu'une variable - s'il y a une référence sur elle - peut être adressée sous deux noms
  - ```
    int i = 3 ;
    int &ri = i
    ;
    cout << ri ;   //Sortie 3
    ri = ri + 2 ;
    cout << i ;    // sortie 5
    ```
- **Note** : L'adresse d'une référence est **égale à l**'adresse de la variable qu'elle référence, c'est-à-dire que `&ri` **est égale à** `&i`.
  - ```
    cout << &ri << "    " << &i
    // Sortie : 0x1ac80ff4    0x1ac80ff4
    ```

Références II

- **Syntaxe** de la définition de référence



  - Une variable de référence doit être **initialisée lors de sa définition**
  - Au cours du programme, il n'est **plus** possible d'**attribuer à** une variable de référence une **autre variable** dont elle doit être la référence.
- Une référence n'utilise pas d'espace mémoire supplémentaire ; une **référence** n'est **qu'**un **alias de nom** pour quelque chose qui a déjà un nom et un espace mémoire.

Références en tant que paramètres de fonction

- Exemple d'utilisation : **call-by-reference**

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

void inc(int &ri)
{
   ri++ ;
}
void main()
{
  int i = 3 ;
  cout << "Avant :   " << i << endl
                       ;
  inc(i)
  ;
  cout << "Plus      " << i ;
          tard :
}
```

édition :

```
Avant :   3
Après : 4
```

- **`ri`** est initialisé par l'appel de fonction et est donc un autre nom pour la variable **`i`**
- Dans les fonctions, il est ainsi possible de modifier durablement la taille de l'environnement d'appel sans utiliser de pointeurs (!!).

Références comme paramètres de retour I

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

int a[5] = {1,1,1,1} ;

int &select(int i)
{
  if( i >= 0 && i < 5)
     return a[i] ;
  else
     return a[0] ;
}

void main()
{
   int i ;
   cout << "Quel composant doit être incrémenté ? " ; cin >>
   i ;
   select(i)++ ;
   for (i = 0 ; i < 5 ; i++)
      cout << a[i] << endl ;
}
```

*Si vous saisissez 3, vous **obtenez** le résultat suivant :*

```
1
1
1
2
1
```

Références comme paramètres de retour II

- **Notez** que dans l'exemple ci-dessus, **`select ne renvoie`** pas la valeur de la variable **`a[i]`**, mais une référence à la variable **`a[i]`** elle-même.
- **Les appels de fonction** peuvent donc - s'ils représentent une référence - apparaître **à gauche des affectations.**
- **Ne jamais renvoyer** une **référence** à une **variable locale** d'une fonction (!)

```
#include <iostream.h>
int a[5] = {1,1,1,1,1}
;
int &select(int i)
{
  if( i >= 0 && i < 5)
     return a[i] ;
  else
     return a[0] ;
}
void main()
{
  int i ;
  cout <<
    "Définir quel composant ? " ;
  cin >> i ;
  select(i) = 5 ;
  for (i = 0 ; i < 5 ; i++)
    cout << a[i] << endl ;
}
```

Entrée : 3
Sortie : 1
1
1
5
1

Les références et la spécification const

- **Call-by-reference en association avec const** `int efficace(const VarType &var) ;` permet
  - **transfert efficace des paramètres**, même pour les variables ou les objets très gourmands en mémoire, car seule une référence à l'objet complet est créée, et non une copie de celui-ci
  - la protection de l'objet contre toute modification de sa fonction
- **Références constantes comme valeurs de retour**
  `const int &protection(int v) ;`
  protègent l'objet pour lequel une référence est renvoyée contre l'accès en écriture
  - `protection(v)++ ;` **`// ne compile pas !`**

Résumé I

- **Le C++** est un **véritable super-ensemble** du langage **C** en ce qui concerne l'étendue du langage.
- Les principales **faiblesses du langage C** résident dans l'ensemble des **types de données élémentaires** disponibles, dans la **structure plate du programme C**, dans le **découplage des types de données** (définis par l'utilisateur) et **des fonctions de manipulation** correspondantes, dans l'**absence d**'un **système d'autorisation** pour l'utilisation de fonctions par d'autres fonctions, dans l'**absence de gestion des exceptions**.
- Des faiblesses du C découlent les principaux objectifs de conception du C++.

Résumé II

- L'utilisation d'**espaces de noms** permet de **résoudre les conflits de noms** lorsque des **identificateurs identiques sont** utilisés dans différents modules.
- Les flux de fichiers **cin** et **cout** offrent des possibilités d'entrée/sortie **flexibles** et **faciles** à utiliser.
- Le C++ met à disposition un **type de données bool** ainsi que des opérations flexibles sur les chaînes de caractères avec la **classe string**.
- Avec **new** et **delete**, les structures de données dynamiques peuvent être gérées de manière pratique et alternative à l'utilisation de malloc() et free().
- **Les fonctions en ligne** sont un moyen de programmation **sûr** et **efficace** dans des situations d'application spéciales.

Résumé III

- En C++, les fonctions peuvent être **définies** avec **des paramètres par défaut.**
  pourvu que
- En C++, les fonctions ne sont pas seulement identifiées par leur nom, mais aussi par le nombre et le type de leurs paramètres. Cela permet de **surcharger les fonctions**.
- **Les références** sont des **alias** pour des variables ou des objets déjà existants.
- Les principaux domaines d'utilisation des références sont le **call-by-reference** dans les fonctions (**transfert de paramètres efficace** pour les grands objets) et le **retour d'une référence d'objet**.
- Les bibliothèques C et C++ peuvent être identifiées par la commande ***externe "C".***

Utiliser les instructions **ensemble** dans un projet.