UNIVERSITÉ DES SCIENCES ET DE LA TECHNOLOGIE D'ORAN « USTOMB »
FACULTÉ DE GÉNIE ÉLECTRIQUE - DÉPARTEMENT D'ELN

MASTER 1: ESE / I / RT / IB
MODULE: POO / LP

Introduction à la Programmation Orientée Objet :: Cours de C++

La responsable du module: Dr. Zouagui-Meddeber L.
lmeduniv@gmail.com

Plan du module

- **Chapitre 1: Introduction à la POO**
- **Chapitre 2 : Langage C++**
  - **Bases du langage (1)**

    Types de base, variables, entrées/sorties , opérateurs, expressions et conversions
  - **Bases du langage (2)**

    Les structures de contrôle, Fonctions
  - **Bases du langage (3)**

    Tableaux, Pointeurs et arithmétique des pointeurs, Allocation dynamique, Énumérations et Structures
- **Chapitre 3: Classes et objets**

Déclaration de classe, Variables et méthodes d'instance, Définition des méthodes, Droits d'accès et encapsulation, Séparations prototypes et définitions, Constructeur et destructeur, Les méthodes constantes, Association des classes entre elles, Classes et pointeurs.

Plan du module

Chapitre 4: Héritage et polymorphisme

Héritage, Règles d'héritage, Chaînage des constructeurs, Classes de base, Préprocesseur et directives de compilation, Polymorphisme, Règles à suivre, Méthodes et classes abstraites, Interfaces, Traitements uniformes, Tableaux dynamiques, Chaînage des méthodes, Implémentation des méthodes virtuelles, Classes imbriquées.

- **Chapitre 5: Les conteneurs, itérateurs et foncteurs**

Les séquences et leurs adaptateurs, Les tables associatives, Choix du bon conteneur, Itérateurs : des pointeurs boostés, La pleine puissance des *list* et *map,* Foncteur : la version objet des fonctions, Fusion des deux concepts.

- **Chapitre 6: Notions avancées**

La gestion des exceptions, Les exceptions standard, Les assertions, Les fonctions templates, La spécialisation, Les classes templates.

Pourquoi ce cours ?

Place importante de la modélisation en electronique:

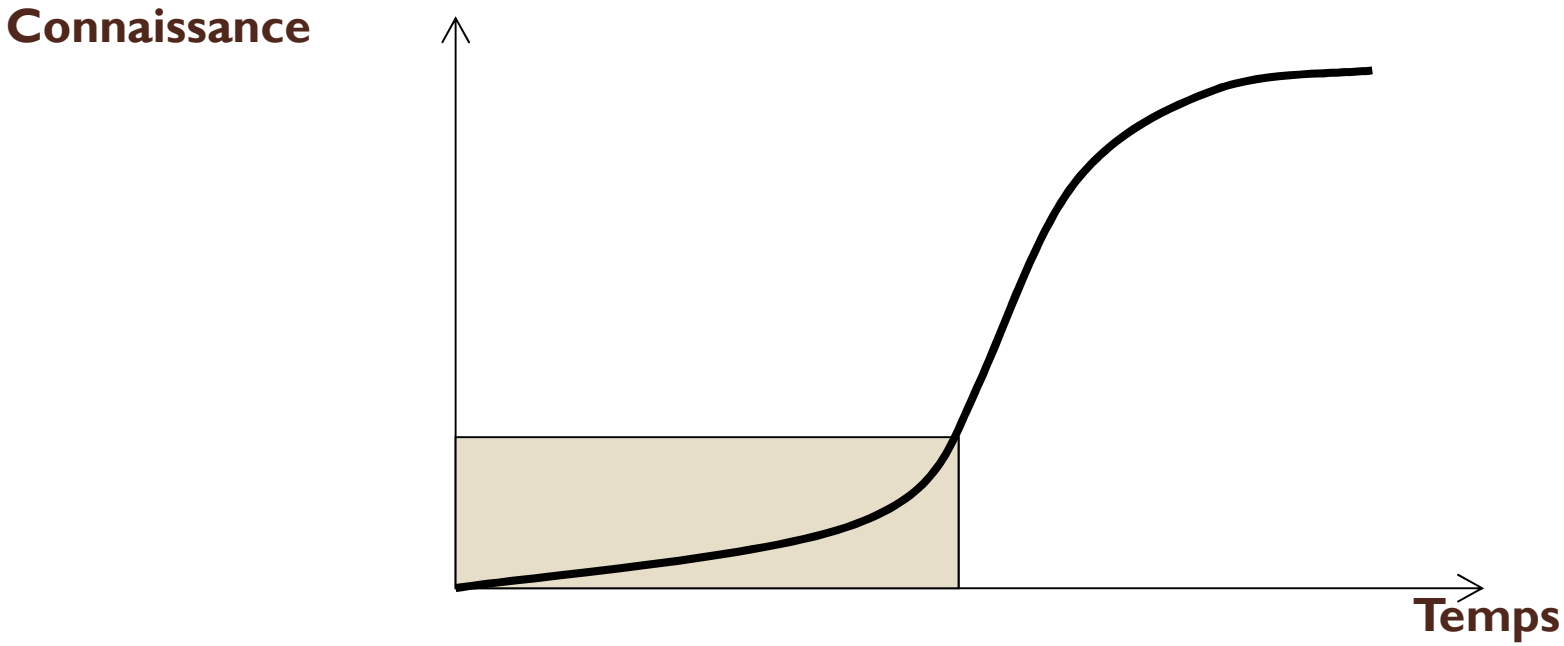
- validation d'une théorie
- prédiction et calculs
- évaluation des risques

Les phénomènes étudiés sont complexes

La programmation orientée-objets peut faire gagner beaucoup de temps

Objectif du cours

- Comprendre les concepts élémentaires de programmation orientée objet (POO)
- Etre capable de lire et comprendre du code C++
- Analyser un problème et le décomposer pour mieux le traiter
- Mettre en oeuvre les concepts de POO en langage C++



Qualité d'un Programme ?

- **Validité** — Le programme fonctionne
- **Extensibilité** — Je peux le faire évoluer
- **Réutilisabilité** — Je peux utiliser des composantes du programme pour d'autres applications

Comment s'y prendre ?

Séparer le programme en modules réutilisables

Unité fonctionelle autonome,
possédant une interface `(header.h)`
Indépendante de son implantation `(impl.cpp)`

La conception par objets

Cette méthode privilégie les données, et non les fonctions

- ✓ Trouver les objets (physiques)
- ✓ Décrire et classer les objets
  - Opérations
  - Liens
  - Caractéristiques communes
- ✓ Implanter ces objets

Un langage orienté objet

Permet:

La définition d'objets, avec partie publique et privée

La lisibilité (distinction interface / implémentation)

La modularité (compilation séparée)

La réutilisation (mise en facteur des fonctionalités)

L'extension des objets (héritage)

La définition d'objets abstraits

Historique

Créé par B. Stroustrup (Bell Labs.) à partir de 1979 (“C with classes”).

Devient public en 1985 sous le nom de C++.

La version normalisée (ANSI) paraît en 1998.

C++ = C +

- Vérifications de type + stricte
- Surcharge de fonctions
- Opérateurs
- Références
- Gestion mémoire + facile
- Entrées/sorties + facile
- Classes et héritage
- Programmation générique
- …

Un langage compilé…

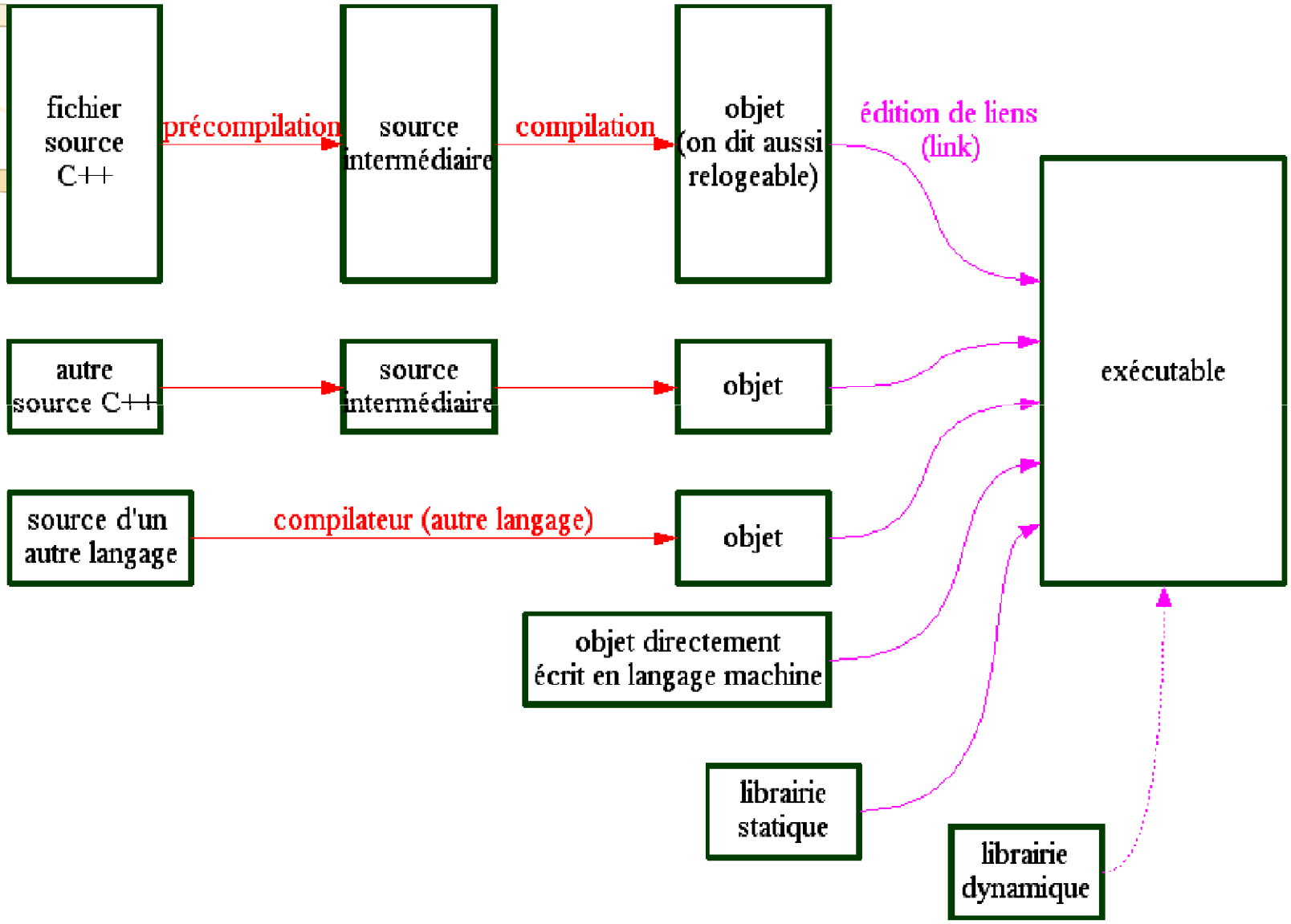
Langage = protocole de communication entre le programmeur et le processeur.

Le processus de compilation

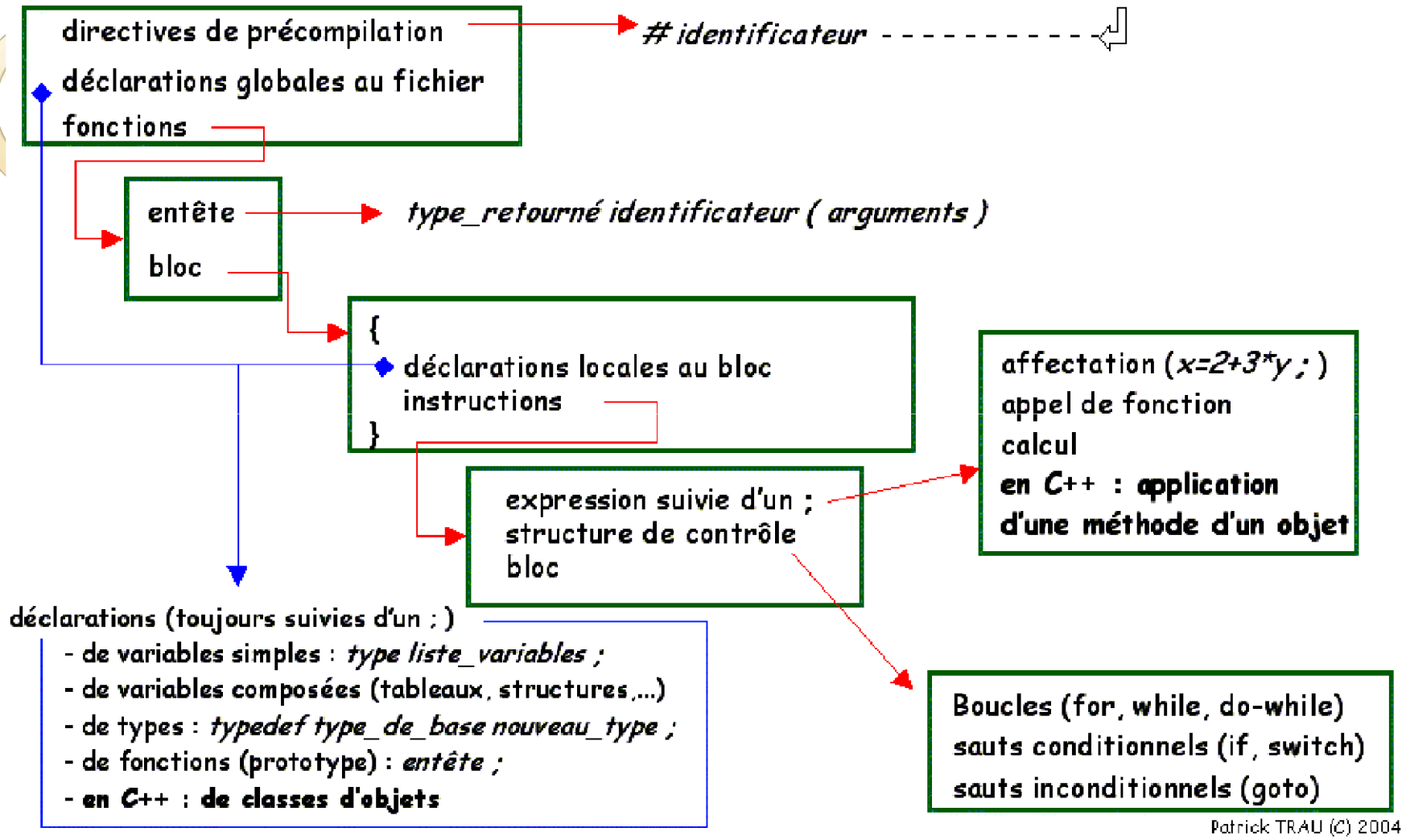
Le C++ est un langage compilé, c'est à dire qu'il faut :

- entrer un texte dans l'ordinateur (à l'aide d'un programme appelé EDITEUR),
- le traduire en langage machine (c'est à dire en codes binaires compréhensibles par l'ordinateur) : c'est la compilation ,
- l'exécuter.

Le processus de compilation



Structure d'un fichier source, définitions



Structure d'un fichier source, définitions

Prototypes de fonctions

header.h

```
// declaration in a header file.
void function( int param1, int param2 = 10 );
```

impl.cpp

```
// implementation file
void function( int param1, int param2 )
{
        // print 'param1' and 'param2' to
        // the standard output.
}
```

client.cpp

```
// main function
int main()
 {
        function( 5, 3 );
        function( 5 );
        return 0;
}
```

Structure d'un fichier source, définitions

```
/* premier exemple de programme C++ */
 #include <iostream>
using namespace std;
 const  float  TVA=19.6;
void main(void)
{
   float HT, TTC;  //on déclare deux variables
   cout<<"veuillez entrer le prix HT : ";
   cin>>HT;
  TTC=HT*(1+(TVA/100));
   cout<<"prix TTC : "<<TTC<<endl;
}
```

Entrées/sorties: **cout et cin**

Entrées/sorties fournies à travers la librairie ***iostream***

- **cout << expr1 << … << exprn**
  - Instruction affichant *expr1 puis expr2, etc.*
  - **cout :** « flot de sortie » associé à la sortie standard (*stdout)*
  - **<< :** opérateur binaire associatif à gauche, de première opérande cout et de 2ème l'expression à afficher, et de résultat le flot de sortie
  - **<< : opérateur surchargé** (ou sur-défini)⇒ utilisé aussi bien pour les chaînes de caractères, que les entiers, les réels etc.
- **cin >> var1 >> … >> varn**
  - Instruction affectant aux variables *var1, var2, etc. les valeurs lues*
  - **cin : « flot d'entrée » associée à l'entrée standard (*stdin)***
  - **>> :** opérateur similaire à **<<**

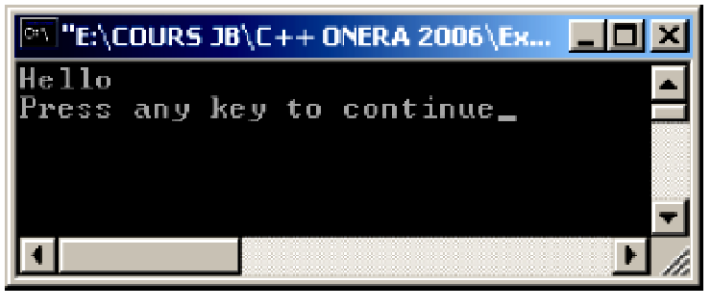
Entrées/sorties: **cout** et **cin**

Possibilité de modifier la façon dont les éléments sont lus ou écrits dans le flot :

dec: lecture/écriture d'un entier en décimal
oct: lecture/écriture d'un entier en octal
hex: lecture/écriture d'un entier en hexadécimal
endl: insère un saut de ligne et vide les tampons
setw(int n): affichage de *n caractères*
setprecision(int n): affichage de la valeur avec *n chiffres* avec éventuellement un arrondi de la valeur
setfill(char): définit le caractère de remplissage flush vide les tampons après écriture

```
#include <iostream>
#include <iomanip>      // attention a bien inclure cette librairie
int main()
{  int i=1234;
   float p=12.3456;
   std::cout << "|" << setw(8) << setfill('*')
          << hex << i << "|" <<std::endl << "|"
          << setw(6) << setprecision(4)
          << p << "|" << std::endl;
return0;
}
```

```
|*****4d2|
|*12.35|
```




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

int main()
{
   cout << "Hello " << endl;
   return 0;
}
```

```
/* Une fonction
   aimable en C */

#include <stdio.h>

void main(void)
{
  printf("Hello\n");
}
```

```
// Une fonction
// aimable en C++

#include <iostream>

void main()
{
  std::cout << "Hello" << std::endl;
}
```

Commentaires C : `/* sur plusieurs lignes */`

Commentaires C++ : `// sur une ligne`

Tout ce qui était possible en C, l'est aussi en C++ :

```
/* les commentaires
   a la mode C */

#include <iostream>
#include <cstdio>                    // les librairies C

int main()
{
  std::cout << "Hello" << std::endl;
  printf ("Hello bis\n");            // les instructions C
  return 0;
}
```

En général, on considère que le programme principal « `main` » doit retourner « **0** » s'il s'est correctement exécuté.

Tout ce qui était possible en C, l'est aussi en C++ :

```
/* les commentaires
   a la mode C */

#include <iostream>
#include <cstdio>

int main()
{
  std::cout << "Hello" << std::endl;
  printf ("Hello bis\n");          // les instructions C
  return 0;
}
```

```
#include <iostream>
#include <cstdio>
using namespace std;
int main()
{
   cout << "Hello " << endl;
    printf("Hello bis\n" )
   return 0;
}
```

En général, on considère que le programme principal « `main` » doit retourner « **0** » s'il s'est correctement exécuté.

Espaces de noms

- Utilisation d'**espaces de noms** (*namespace*) lors de l'utilisation de nombreuses bibliothèques pour éviter les conflits de noms
- **Espace de noms** : association d'un nom à un ensemble de variable, types ou fonctions

  Ex. Si la fonction *MaFonction()* est définie dans l'espace de noms *MonEspace*, l'appel de la fonction se fait par *MonEspace::MaFonction()*

- Pour être parfaitement correct :

```
std::cin
std::cout
std::endl
```

`::` *opérateur de résolution de portée*

- **Pour éviter l'appel explicite à un espace de noms** : `using`

```
using std::cout ; // pour une fonction spécifique
using namespace std; // pour toutes les fonctions
```

Entrées/sorties: C/C++

C

```
#include <stdio.h>
int value = 10;
printf( "value = %d\n", value );
printf( "New value = ??\n" );
scanf( "%d", &value );
```

C++

```
#include <iostream>
using namespace std;
int value = 10;
cout << "Value = " << value << endl;
cout << "New value = ?? " << endl;
cin >> value;
```

Les opérateurs '<<' et '>>' sont surchargeables.

C : mots-clés

Le langage C dispose de 33 mot-clés :

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

Auxquels on ajoute :

- des opérateurs unaires : `-` `~` `!` `*` `&` `sizeof` `+` `++` `--`
- des opérateurs binaires : `.` `->` `*` `/` `%` `+` `-` `<<` `>>` `<` `>` `<=` `>=` `==` `!=` `&` `|` `^` `&&` `||`
- des « ponctuations » : `[ ]` `( )` `{ }` `,` `:` `=` `;` `...` `#`

C++ : nouveaux mots-clés

Le C++ introduit 41 nouveaux mot-clés qui s'ajoutent aux 33 du langage C :

`bool`	`catch`	`class`	`const_cast`
`delete`	`dynamic_cast`	`explicit`	`export`
`false`	`friend`	`inline`	`mutable`
`namespace`	`new`	`operator`	`private`
`protected`	`public`	`static_cast`	`reinterpret_cast`
`template`	`this`	`throw`	`true`
`try`	`typeid`	`typename`	`using`
`virtual`	`wchar_t`	`and(&&)`	`and_eq(&=)`
`bitand(&)`	`bitor(|)`	`compl(~)`	`not(!)`
`not_eq(!=)`	`or(||)`	`or_eq(|=)`	`xor(^)`
`xor_eq(^=)`			

C++ : définition des variables

La définition d'une variable se compose au minimum d'un nom (symbole) du type de données et du nom (symbole) de l'identifiant :

```
type identifiant;
```

On peut optionnellement donner une valeur à la variable :

```
type identifiant [= valeur];
```

On peut définir d'un coup plusieurs variables du même type en les séparant par une virgule :

```
type ident1 [= val1], ident2 [= val2], … identN [= valN];
```

On peut les déclarer pratiquement n'importe où (en se limitant ainsi aux stricts endroits nécessaires), ce qui est une grande différence avec le C (cf. planche suivante).

C++ : définition des variables

1) en C++, pas nécessairement regroupées en début de bloc
2) leur portée s'arrête à la fin du bloc (accolade fermante `}`)
3) on peut initialiser avec des valeurs obtenues par calcul
4) on peut déclarer dans les parenthèses des `if`, `for`, `while`, `do`

En C

```c
int main(void)
{
  int i, n, m;
  float x;
  scanf("%d", &n);
  m = n;

  for (i=0; i<m; i++)
  {
    /* ... */
  }
  return 0;
}
```

En C++

```c
int main()
{
  int n;
  std::cin >> n;
  const int m = n;

  for (int i=0; i<m; i++)
  {
    float x;
    // ...
  }
  return 0;
}
```

C++ : définition des variables

En C

```c
int main(void)
{
  int i, n, m;
  float x;
  scanf("%d", &n);
  m = n;

  for (i=0; i<m; i++)
  {
    /* ... */
  }
  return 0;
}
```

En C++

```c
int main()
{
  int n;
  std::cin >> n;
  const int m = n;

  for (int i=0; i<m; i++)
  {
    float x;
    // ...
  }
  return 0;
}
```

Précisions :

- dans le code C, les variables sont valables dans tout le code du « `main` »
- dans le code C++ :
  - la variable « `n` » et la constante sont valables dans tout le « `main` »
  - la variable « `i` » est seulement valable dans le « `for` »
  - la variable « `x` » est seulement valable dans le bloc interne au « `for` »

C++ : variables « globales »

Il est possible de définir/déclarer des variables dans un fichier « cpp », c'est-à-dire dans ce que l'on appelle **une unité de compilation**, en dehors de toute fonction, classe ou méthode :

```
int globale;
static int locale;
```

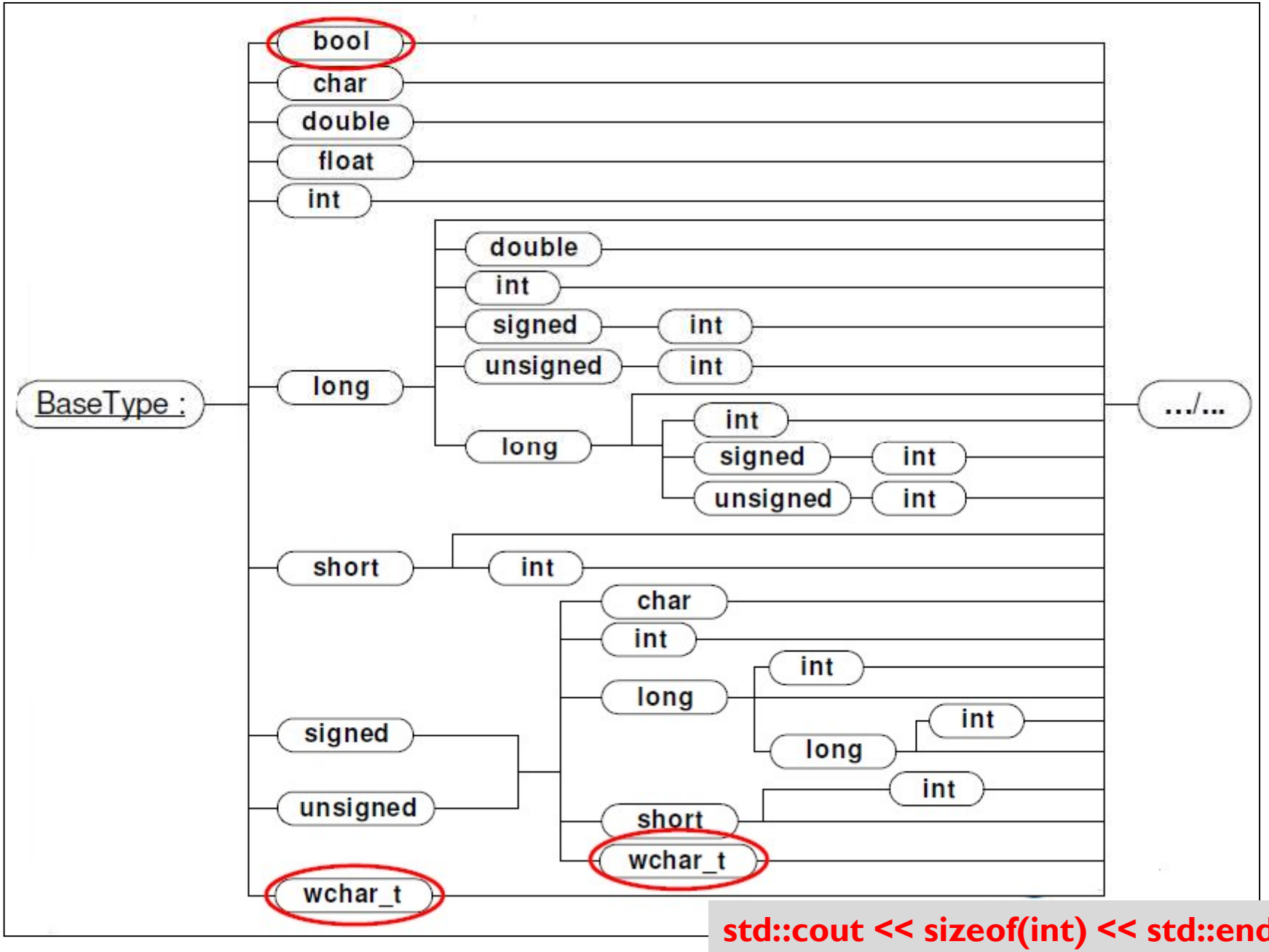
fichier1.cpp

```
int globale;           // Erreur de link
extern int globale;    // OK
int locale;            // OK, mais pas locale …
```

fichier2.cpp

- « **static** » : la variable est locale au fichier
- « **extern** » : la variable est définie ailleurs, il s'agit d'une déclaration

C++ : types de base




```
std::cout << sizeof(int) << std::endl;
```

C : types de base

Types de base : **char, int, float, double**

\+ des modificateurs optionnels : **long, short, unsigned, signed**

```
char c, ch;
c  = 'a';
ch = '\n';

int i, j;
i = 10;

long int ii;
long jj;
ii = 12345;
jj = 888888L;

int m = 1000;
long n = 999999L;
```

```
short int mm;
short nn;
short pp = 100;

unsigned int i1;
unsigned long j1;
unsigned short k1;
i1 = 345U;
j1 = 888888UL;

signed int i2;
signed long j2;
signed short k2;
```

```
float x1, x2;
float x3 = 0.1F;

double y;
long double z;

x1 = 0.45;
x2 =-3.5e12F;
y  = 0.45;
z  = 0.99e-2L;
```

void : pour spécifier qu'il n'y a pas de type

C++ : types de base

- Héritage des mécanismes de bases du C (pointeurs inclus)

Attention : typage fort en C++!!

- Déclaration et initialisation de variables :

```
bool this_is_true = true; // variable boléenne
int i = 0; // entier
long j = 123456789; // entier long
float f = 3.1; // réel
// réel à double précision
double pi = 3.14159265358979323846264643;
char c='a'; // caractère
```

- « Initialisation à la mode objet » :

```
int i(0) ;
long  j(123456789);
…
```

C++ : types de base

Précisions :

En ce qui concerne le type « **wchar_t »,** il s'agit d'une représentation des caractères sur 16 bits ce qui permet d'utiliser le codage Unicode des caractères à la place du codage ASCII (8 bits). Ce codage étendu est nécessaire pour représenter des alphabets non latins et envisager ce que l'on appelle *l'internationalisation des programmes*.

Note importante :

Le C et le C++ n'ont pas de type de base pour représenter les chaînes de caractères (comme en JAVA le type « string »). Les chaînes de caractères sont représentées par des tableaux de caractères. Il existe cependant dans la librairie standard de C++ une classe **std::string**, mais il ne s'agit pas d'un type de base, faisant partie du noyau du langage.

C++ : types de base (les caractères)

Les littéraux caractères se notent avec des guillemets simples (« quote » en anglais). Eventuellement précédés de « **L** » pour les caractères longs :

```
char    c1 = 'a';
wchar_t c2 = L'b';
```

Les caractères inaccessibles au clavier peuvent être fournis grâce à leur code en base 16 (précédé de **\x**) ou en base 8 (simplement précédé de ****) :

```
char  c1 = 'a';
char  c2 = L'a';
char  c3 = '\141';
char  c4 = '\x61';
```

toutes représentations du caractère `'a'`

Il existe un certain nombre de séquences d'échappement reconnues par le langage C/C++ :

```
\'   \"   \?   \\
\a   \b   \f   \n   \r   \t   \v
```

C++ : types de base (les entiers)

Les littéraux entiers sont des nombres, sans exposant, ni virgule.

- en base décimale (10), les nombres entiers ne peuvent commencer par « **0** »
- en base octale (8), ils doivent commencer par « **0** »
- en base hexadécimale (16), ils doivent commencer par « **0x** » ou « **0X** »
- les nombres peuvent être précédés d'un opérateur unaire (« **–** » ou « **+** »)

```
unsigned int a1 = 67;
unsigned int a2 = 0103;
unsigned int a3 = 0x43;
```

toutes représentations du nombre `67'

Rappel : les suffixes « **U** » et « **L** », en minuscules ou majuscules, seuls ou ensembles peuvent être employés pour préciser si l'entier est non signé ou s'il est long.

C++ : types de base (les réels)

Les littéraux flottants sont des nombres réels avec un exposant et/ou une virgule. Les nombres peuvent être précédés d'un signe, opérateur unaire (« – » ou « + ») :

```
float x1, x2;
float x3 = 0.1F;

double y;
long double z;

x1 =  0.45;
x2 = -3.5e12F;
y  =  0.45;
z  =  0.99e-2L;
```

Rappel : les suffixes « **F** » et « **L** », en minuscules ou majuscules, permettent de préciser respectivement que la constante est simple précision (« `float` ») ou quadruple precision (« `long double` »). S'il n'y a aucun suffixe, il s'agit d'une constante double précision (« `double` »).

C++ : types de base (les chaines de caractères)

En C et C++, les littéraux chaînes de caractères sont définis comme des tableaux de n caractères du type « **`const char`** ».

En C et C++, les chaînes de caractères se terminent par le caractère zéro (**`\0`**) qui compte dans le nombre de caractères. C'est ce que l'on appelle les chaines de caractères « *null terminated* » par opposition aux chaînes de la classe **`string`** de la librairie standard C++ que nous verrons plus tard.

```
const char s1[5] = "abcd";    // OK
const char s2[5] = "abcde";   // error: array bounds overflow
s1[0] = 'R';                  // error: l-value specifies
                              // const object

char s3[5] = "abcd";          // OK
s3[0] = 'R';                  // OK
```

Type référence (&) et déréférencement automatique

- **Possibilité de définir une variable de type *référence***

```
int i = 5;
int & j = i; // j reçoit i
   // i et j désignent le même emplacement mémoire
```

Impossible de définir une référence sans l'initialiser

- **Déréférencement automatique** :

  Application automatique de l'opérateur d'indirection * à chaque utilisation de la référence

```
int i = 5;
int & j = i;    // j reçoit i
int k = j;      // k reçoit 5
j += 2;  // i reçoit i+2 (=7)
j = k;   // i reçoit k (=5)
```

Une fois initialisée, une référence ne peut plus être modifiée – elle correspond au même emplacement mémoire

C++ : types de base (le type Pointeur)

<Type> *<NomPointeur>

déclare un pointeur <NomPointeur> qui peut recevoir des adresses de variables du type <Type>.

Une déclaration comme :	int *PNUM ;
peut être interprétée comme suit :	" *PNUM est du type int"
ou	"PNUM est un pointeur sur int"
ou	"PNUM peut contenir l'adresse d'une variable du type int"

C++ : types de base (le type Pointeur)

Exemple :

Le programme complet effectuant les transformations de l'exemple ci-dessus peut se présenter comme suit (2 propositions):

```
main( )
{
     /*déclarations*/
   Short A=10 ;
   Short B=50 ;
   Short *P ;
   /*traitement*/
   P= &A ;
   B= *P ;
   *P= 20 ;
  Return 0 ;
}
```

```
main( )
{
    /*déclarations*/
   short A, B, *P ;
   /*traitement*/
   A=10 ;
   B=50 ;
   P=&A ;
   B=*P ;
   *P=20 ;
   return 0 ;
}
```

Résumons :

```
          int A ;
          int*P ;
          P=&A ;
A         désigne le contenu de A.
&A        désigne l'adresse de A.
P         désigne l'adresse de A.
*P        désigne le contenu A.
```

Affectation d'une valeur réelle à une variable entière

Quand on affecte une valeur réelle (`float` ou `double`) à une variable de type `int`, la partie fractionnaire est perdue.

Exemple:

```
float x = 1.5;
int n = 3 * x; // !! provoque un warning
```

→ *n vaut 4* (commentaire)

MAIS le compilateur g++ génère un *warning* :

`warning: converting to 'int' from 'float'`

pour signaler qu'il y a potentiellement un problème, puisqu'on perd la partie fractionnaire.

Pour préciser au compilateur qu'il sait ce qu'il fait, le programmeur **DOIT** avoir une conversion de type `float` vers `int`:

```
n = int(3 * x);
```

Conversions de type (*cast*)

• Le ***cast* implicite** est la conversion effectuée automatiquement par le compilateur quand nécessaire.

Le compilateur signale les *casts* implicites qui peuvent poser problème par un *warning* (notamment quand on affecte une valeur flottante à un entier):

```
float x = 1.5;
int n = 3 * x;  // !! provoque un warning
```

L'inverse:

```
int n = 5;
float x = 3 * n;
```

ne provoque pas de *warning* avec g++, car il n'y a pas de perte de précision contrairement à avant.

• Un ***cast* explicite** est une conversion ajoutée par le programmeur, notamment pour éviter le *warning* précédent, ou pour effectuer une division flottante entre entiers:

```
float x = 1.5;
int n = int(3 * x);  // pas de warning

float x = float(1) / 2; // x contient 0.5
```

C/C++ : Conversions de types

C++ possède une nouvelle syntaxe pour le *transtypage* explicite, bien que l'opérateur *cast* du langage C soit toujours valable, et même indispensable dans certains cas :

En C

```
int i=1;
float x=3.14;

i = (int) x;
x = (float) i;
```

En C++

```
int i=1;
float x=3.14;

i = int(x);                 // Nouvelle syntaxe
x = float(i);

i = (int) x;                // Toujours valable
x = (float) i;

ptr1 = char*(ptr2);         // INCORRECT
ptr1 = (char*)ptr2;         // L'ancien opérateur
                            // est ici obligatoire
```

Exercice

Soit les instructions suivantes:

```
int n, p, q;
float x, y;

n = 15;
p = 10;
x = 2.5;
```

Que contiennent les variables `q` ou `y` après les instructions suivantes:

```
y = x + n % p;
q = x + n / p;
y = (x + n) / p;
y = (n + 1) / p;
y = (n + 1.) / p;
q = (n + 1.) / p;
y = (float(n) + 1) / p;
```

Où faut-il ajouter des conversions de type pour éviter les *warnings*?

Exercice

Soit les instructions suivantes:

```
int n, p, q;
float x, y;

n = 15;
p = 10;
x = 2.5;
```

Que contiennent les variables `q` ou `y` après les instructions suivantes:

`y = x + n % p;`	y=7.5
`q = x + n / p;`	q=3
`y = (x + n) / p;`	y=1.75
`y = (n + 1) / p;`	y=1
`y = (n + 1.) / p;`	y=1.6
`q = (n + 1.) / p;`	q=1
`y = (float(n) + 1) / p;`	y=1.6

Où faut-il ajouter des conversions de type pour éviter les *warnings*?

C++ : Expressions constantes

Une expression constante est une expression qui peut être évaluée à la compilation :

```
const int MAX = 100;  // expression non constante en C
                      // expression constante en C++
```

Les constantes C++ se rapprochent des symboles définis avec **#define** (que l'on évitera d'utiliser désormais pour définir des constantes) :

En C

```
#define M 1000
const int N = 100;
```

En C++

```
const int M = 1000;
const int N = 100;
```

C/C++ : opérateurs

Opérateur	Signification	Type
: :	*Scope resolution* Résolution de portée	binaire
: :	*Global* Portée globale	unaire
[ ]	*Array subscript* Manipulation de tableaux	binaire
( )	*Function call* Appel de fonction	variable
.	*Member selection (object)* Sélection de membre	binaire
–>	*Member selection (pointer)* Sélection de membre	binaire

Précisions :

- L'opérateur **::** permet de lever les ambiguïtés entre des variables de même nom
- Le choix de l'opérateur de sélection dépend de l'entité à laquelle il s'applique
- Il existe des opérateurs d'incrémentation et de décrémentation préfixés

C/C++ : opérateurs

Opérateur	Signification	Type
`new`	*Allocate object* Allocation dynamique	variable
`delete`	*Deallocate object* Libération mémoire	unaire ou binaire
`delete[ ]`	*Deallocate object* Libération mémoire (tableaux)	unaire ou binaire
`++`	*Prefix increment* Incrément (préfixe)	unaire
`--`	*Prefix decrement* Décrément (préfixe)	unaire
`*`	*Dereference* Valeur du pointeur	unaire
`&`	*Address-of* Adresse de	unaire
`-`	*Arithmetic negation (unary)* Négation unaire	unaire

Précisions :

- Les opérateurs **new** et **delete** ont une grande importance en C++
- Les opérateurs * et **&** sont inverses l'un de l'autre
- L'opérateur de négation unaire est différent de l'opérateur de négation binaire

C/C++ : opérateurs

Opérateur	Signification	Type
`!`	*Logical NOT* Négation logique	unaire
`~`	*Bitwise complement* Complément à un	unaire
`sizeof`	*Size of object or type* Taille d'une variable ou d'un type	unaire
`typeid( )`	*Type name* Nom d'un type (chaîne de caractères)	unaire
`(type)`	*Type cast (conversion)* Conversion de type (transtypage)	binaire
`.*`	*Apply pointer to class member (objects)* Sélection de membre et déréférencement	binaire
`->*`	*Dereference pointer to class member* Sélection de membre et déréférencement	binaire
`*`	*Multiplication* Multiplication	binaire

Précisions :

- L'opérateur **sizeof** retourne une valeur du type **size_t**
- L'opérateur **typeid** retourne une valeur du type **const type_info&**
- Ne pas confondre la multiplication * et l'opérateur d'indirection *

C/C++ : opérateurs logiques et mathématiques

Opérateurs « multiplicatifs » :

Opérateur	Signification	Type	Associativité
*	*Multiplication* Multiplication	binaire	gauche à droite
/	*Division* Division	binaire	gauche à droite
%	*Remainder (modulus)* Modulo	binaire	gauche à droite

Précisions :

- la division est la division entière si les deux arguments sont entiers
- le modulo ne travaille que sur des arguments entiers

C/C++ : opérateurs logiques et mathématiques

Opérateurs « additifs » :

Opérateur	Signification	Type	Associativité
+	*Arithmetic addition (unary)* « addition » unaire	unaire	non
–	*Arithmetic negation (unary)* Négation unaire	unaire	non
+	*Addition* Addition	binaire	gauche à droite
–	*Subtraction* Soustraction	binaire	gauche à droite

Précisions :

Il est clair que « l'addition » unaire ne fait rien pour les types simples (nombres entiers et réels), mais grâce à la *surcharge des opérateurs*, on peut donner un sens à cet opérateur sur des types (= classes) complexes.

C/C++ : opérateurs logiques et mathématiques

Opérateurs de « décalage » :

Opérateur	Signification	Type	Associativité
<<	*Left shift* Décalage à gauche	binaire	gauche à droite
>>	*Right shift* Décalage à droite	binaire	gauche à droite

Précisions :

Ces opérateurs sont assez peu employés dans leur contexte « C » (décalages de bits), mais ils prennent une importance considérable en C++ dans la manipulation des flux d'entrées-sorties (y compris lecture de fichiers et formatages de toutes sortes).

C/C++ : opérateurs logiques et mathématiques

Opérateurs de « décalage » :

Opérateur	Signification	Type	Associativité
<<	*Left shift* Décalage à gauche	binaire	gauche à droite
>>	*Right shift* Décalage à droite	binaire	gauche à droite

Précisions :

Ces opérateurs sont assez peu employés dans leur contexte « C » (décalages de bits), mais ils prennent une importance considérable en C++ dans la manipulation des flux d'entrées-sorties (y compris lecture de fichiers et formatages de toutes sortes).

C/C++ : opérateurs logiques et mathématiques

Exemple :

```
int b = 100;  // 1100100 en code binaire

std::cout << "     b = " << b << std::endl;
std::cout << "(b<<1) = " << (b<<1) << std::endl; // 200 = 11001000
std::cout << "(b<<2) = " << (b<<2) << std::endl; // 400 = 110010000

std::cout << "(b>>1) = " << (b>>1) << std::endl; // 50 = 110010
std::cout << "(b>>2) = " << (b>>2) << std::endl; // 25 =  11001
std::cout << "(b>>3) = " << (b>>3) << std::endl; // 12 =   1100
```

C/C++ : opérateurs logiques et mathématiques

Opérateurs « relationnels » :

Opérateur	Signification	Type	Associativité
<	*Less than* Inférieur à	binaire	gauche à droite
>	*Greater than* Supérieur à	binaire	gauche à droite
<=	*Less than or equal to* Inférieur ou égal à	binaire	gauche à droite
>=	*Greater than or equal to* Supérieur ou égal à	binaire	gauche à droite

Précisions :

En C++, ces opérateurs retournent un type **`bool`** (valeurs **`true`** ou **`false`**), contrairement **au C** où ils retournent un type **`int`** (valeurs **`0`** ou **`1`**).

C/C++ : opérateurs logiques et mathématiques

Opérateurs « d'équalité » :

Opérateur	Signification	Type	Associativité
==	*Equality* Egalité	binaire	gauche à droite
!=	*Inequality* Inégalité	binaire	gauche à droite

Précisions :

En C++, ces opérateurs retournent un type **`bool`** (valeurs **`true`** ou **`false`**), contrairement **au C** où ils retournent un type **`int`** (valeurs `0` ou `1`).

Erreur classique : ne pas confondre l'opérateur d'égalité (==) avec l'opérateur d'affectation (=), cf exemple.

C/C++ : opérateurs logiques et mathématiques

Opérateurs « sur les bits » :

Opérateur	Signification	Type	Associativité
&	*Bitwise AND* ET binaire (bits)	binaire	gauche à droite
^	*Bitwise exclusive OR* OU exclusif binaire (bits)	binaire	gauche à droite
|	*Bitwise OR* OU binaire (bits)	binaire	gauche à droite

Précisions :

Ces opérateurs travaillent sur les bits, comme les décalages (<< et >>).

Ils sont assez peu employés (surtout dans un contexte scientifique). Attention à ne pas les confondre avec les opérateurs logiques.

C/C++ : opérateurs logiques et mathématiques

Opérateurs « logiques » :

Opérateur	Signification	Type	Associativité
`&&`	*Logical AND* ET logique (bool)	binaire	gauche à droite
`||`	*Logical OR* OU logique (bool)	binaire	gauche à droite
e1`?`*e2*`:`*e3*	*Conditional* Condition if-else	ternaire	droite à gauche

Précisions :

Ces opérateurs travaillent sur le type **`bool`** (si ce n'est pas le cas, il y aura éventuellement une conversion implicite) et retournent un **`bool`**.

Le dernier opérateur signifie : « si *e1* est vraie alors faire *e2*, sinon faire *e3* »

C/C++ : opérateurs logiques et mathématiques

Opérateurs « d'affectation » :

Opérateur	Signification	Type	Associativité
=	*Assignment* Affectation	binaire	droite à gauche
*=	Multiplication et affectation	binaire	droite à gauche
/=	Division et affectation	binaire	droite à gauche
%=	Modulo et affectation	binaire	droite à gauche
+=	Addition et affectation	binaire	droite à gauche
-=	Soustraction et affectation	binaire	droite à gauche
<<=	Décalage à gauche et affectation	binaire	droite à gauche
>>=	Décalage à droite et affectation	binaire	droite à gauche
&=	ET binaire et affectation	binaire	droite à gauche
|=	OU binaire et affectation	binaire	droite à gauche
^=	OU exclusif binaire et affectation	binaire	droite à gauche

C/C++ : les priorités des opérateurs

Le tableau ci-dessous illustre la priorité des opérateurs. Elle diminue du haut vers le bas.

Catégorie d'opérateurs	Opérateurs	Sens
Parenthèses	( )	
Multiplication, division, modulo	* / %	G=>D
Addition, soustraction	+ -	G=>D
opérateurs relationnels	< <= > >=	G=>D
opérateurs de comparaison	== !=	G=>D
et logique	&&	G=>D
ou logique	||	G=>D
opérateurs d'affectation	= += -= *= /= %= &= ^= |= <<= >>=	D=>G
Opérateur virgule	,	G=>D

Exemple :

```
X = 3 + 16 / 4 * 2 % 7 ;        X aura la valeur 4
Y = (3 + 16) / 4 * 2 % 7 ;      Y aura la valeur 1
M = 7 > 5 >= 1==3 ;             M aura la valeur 0
N = (7 > 5) >= (1==3) ;         N aura la valeur 1
```

C/C++ : Opérations mathématiques de base

```
int i = 100 + 50;
int j = 100 - 50;
int n = 100 * 2;
int m = 100 / 2; // division entière
int k= 100 % 2; // modulo - reste de la division entière

i = i+1;
i = i-1;

j++; // équivalent à j = j+1;
j--; // équivalent à j = j-1;

n += m; // équivalent à n = n+m;
m -= 5; // équivalent à m = m-5;
j /= i; // équivalent à j = j/i;
j *= i+1; // équivalent à j = j*(i+1);

int a, b=3, c, d=3;
a=++b; // équivalent à b++; puis a=b; => a=b=4
c=d++; // équivalent à c=d; puis d++; => c=3 et d=4
```

Exercice: Evaluations d'expressions

Soient les déclarations suivantes:

int i1, i2 = 3; float f1, f2 = 3.0;

Donnez la valeur des variables modifiées par chacune des expressions suivantes:

1) *i1 = i2 + 1, i1 += i2;*
2) f1 = i2 / 2 + f2;
3) i1 = i2 / 2 + f2 + .5;
4) i1 = i2 / 2 + f2 + .6;
5) i1 = (int) f2 % i2;
6) i1 = i2 >= i2 + 1;
7) i1 = (i2 = 3) + 1;
8) i1 = (i2 == 3) + 1;
9) i1 = i2 < 2 || f2 > 2;
10) i1 = i2 && f2;
11) i1 = !!!i2;
12) i1 = 5 << 3;
13) i1 = -1 >> 1;
14) i1 = -5 >> 1;
15) i1 = ((i2 + i2) * 2) >> 2;
16) i1 = i2 | 1;
17) i1 = i2 & 1;
18) i1 = i2 ^ 1;
19) i1 = ~0;
20) f1 = f2++;
21) i1 = i2 == 0 ? 0 : i2 < 0 ? -1 : 1;
22) i1 = ++i2 == 4 ? i2++ : i2--;
23) i2 += ++i2;

C/C++ : Opérateurs de comparaison

```
int i,j;
…
if(i==j) // évalué à vrai (true ou !=0) si i égal j
{
  … // instructions exécutées si la condition est vraie
}

if(i!=j) // évalué à vrai (true ou !=0) si i est différent de j

if(i>j) // ou (i<j) ou (i<=j) ou (i>=j)


if(i) // toujours évalué à faux si i==0 et vrai si i!=0
if(false) // toujours évalué à faux
if(true) // toujours évalué à vrai
```

C/C++ : les structures de contrôles:
« Traitement alternatif »

La structure conditionnelle « `if` » :

```
if (test)                    // forme 1
     operation;
```

```
if (test)                    // forme 2
     operation1;
else
     operation2;
```

Précisions :

- les parenthèses autour de `test` sont nécessaires
- `test` est en principe un **`bool`**, mais toute valeur non nulle est considérée comme vraie

C/C++ : les structures de contrôles

La forme la plus simple est :

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

Chaque instruction peut être un bloc d'instructions qu'il faut mettre entre accolades :

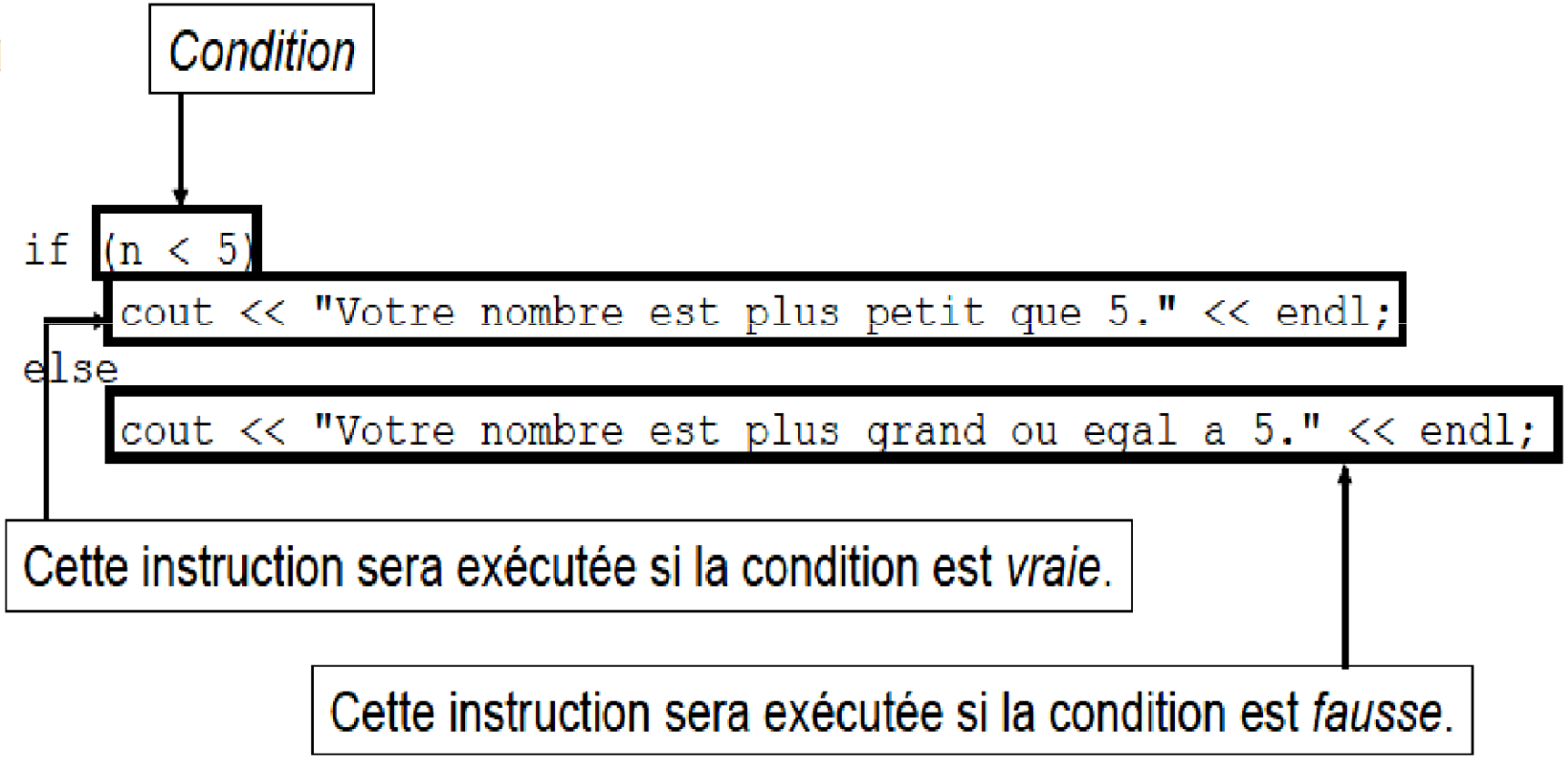
```
if (expression)
        {
                Instruction1 ;
                Instruction2 ;
                … ;
        }
```

La forme la plus générale est celle-ci :

```
if (expression1)
        instruction1 ;
else  if (expression2)
        instruction2 ;
…
else if (expressionN)
        instructionN ;
else
        instructionM ;
```

C/C++ : les structures de contrôles

«La structure conditionnelle IF..else»



C/C++ : les structures de contrôles

«La structure conditionnelle alternative IF..else..»

```
x = 10;
y = x > 9 ? 100 : 200; // équivalent à
                       // if(x>9) y=100;
                       // else y=200;

int main()
{
   float a;

   cout << "Entrer un réel :";
   cin >> a;

   if(a > 0) cout << a << " est positif\n";
   else
    if(a == 0) cout << a << " est nul\n";
    else cout << a << " est négatif\n";    return 0;
}

// Mettre des {} pour les blocs d'instructions des if/else pour
// éviter les ambiguïtés et lorsqu'il y a plusieurs instructions
```

C/C++ : les structures de contrôles

«La structure conditionnelle alternative IF..else..»

```
cout << "Entrez le premier nombre:" << endl;
cin >> n;
cout << "Entrez le deuxieme nombre:" << endl;
cin >> p;

if ((n < p) && (2 * n >= p))
  cout << "Test 1" << endl;

if ((n < p) || (2 * n > p))
  cout << "Test 2" << endl;

if ( ((n < p) && (2 * n > p)) || (n > p) )
  cout << "Test 3" << endl;

if (!(n > p))
{
  if ((2 * n <= p) && (3 * n > p))
    cout << "A" << endl;

  if (p % 2 == 0)
    cout << "B" << endl;
}
else
  if (3 * p > n)
    cout << "C" << endl;
```

Qu'affiche ce programme quand l'utilisateur entre 10 et 5 ? 1 et 4 ?

C/C++ : les structures de contrôles

«La structure conditionnelle répétitive For»

La boucle « `for` » :

```
for (initialisation; test; iteration)
    operation;
```

- `initialisation` est une instruction (ou un bloc d'instructions) exécutée avant le premier parcours
- `test` est une expression qui, si elle est vraie, détermine la fin de la boucle
- `iteration` est une instruction (ou un bloc d'instructions) qui est effectuée à chaque boucle

Forme « habituelle » en C++ :

```
for (int i=0; i < n; i++)
{
  // operations
}
```

« Boucle infinie » :

```
for (;;)
{
  // operations
}
```

C/C++ : les structures de contrôles

«La structure conditionnelle répétitive For»

Déclaration et initialisation:
n'est exécutée qu'une seule fois, avant d'entrer dans la boucle

Condition:
testée avant l'exécution de chaque tour de boucle. Si elle est fausse, on sort de la boucle.

```
for(int i = 0; i < 5; i++
{
    cout << "i = " << i << endl;
    cout << "Bonjour" << endl;
}
```

Corps de la boucle:
Bloc d'instructions qui sera exécuté à chaque tour de boucle.

Incrémentation:
exécutée à la fin de chaque tour de boucle. Elle permet de changer la valeur du compteur de boucle (ici, la variable `i`).

Rappel: `i++;` ajoute 1 à la variable `i`. Cette instruction fait la même chose que `i = i + 1;`

C/C++ : les structures de contrôles

«La structure conditionnelle répétitive For»

```
for(initialisation; condition; incrémentation)
   instruction; // entre {} si plusieurs instructions
```

Exemples :

```
for(int i=1; i <= 10; i++)
  cout << i << " " << i*i << "\n"; // Affiche les entiers de
                                   // 1 à 10  et leur carré

int main()
{
  int i,j;
  for(i=1, j=20; i < j; i++, j-=5)
   {
     cout << i << " " << j << "\n";
   }
}
```

Résultat affiché :

```
1 20
2 15
3 10
4 5
```

C/C++ : les structures de contrôles

«La structure conditionnelle répétitive While»

Le « `while` » :

```
while (test)
    operation;
```

Mêmes précisions que pour le « if » :

- les parenthèses autour de `test` sont nécessaires
- `test` est en principe un **`bool`**, mais toute valeur non nulle est considérée comme vraie
- il faut mettre des accolades s'il y a plusieurs instructions.

C/C++ : les structures de contrôles

«La structure conditionnelle répétitive While»

```
int main ()
{
    int n=8;
    while (n>0)
    { cout << n << " ";
       --n;
    }
     return 0;
 }
```

Instructions exécutées tant que *n* est supérieur à zéro

Résultat affiché :

```
8 7 6 5 4 3 2 1
```

C/C++ : les structures de contrôles

«La structure conditionnelle répétitive Do ..While»

Le « do-while » :

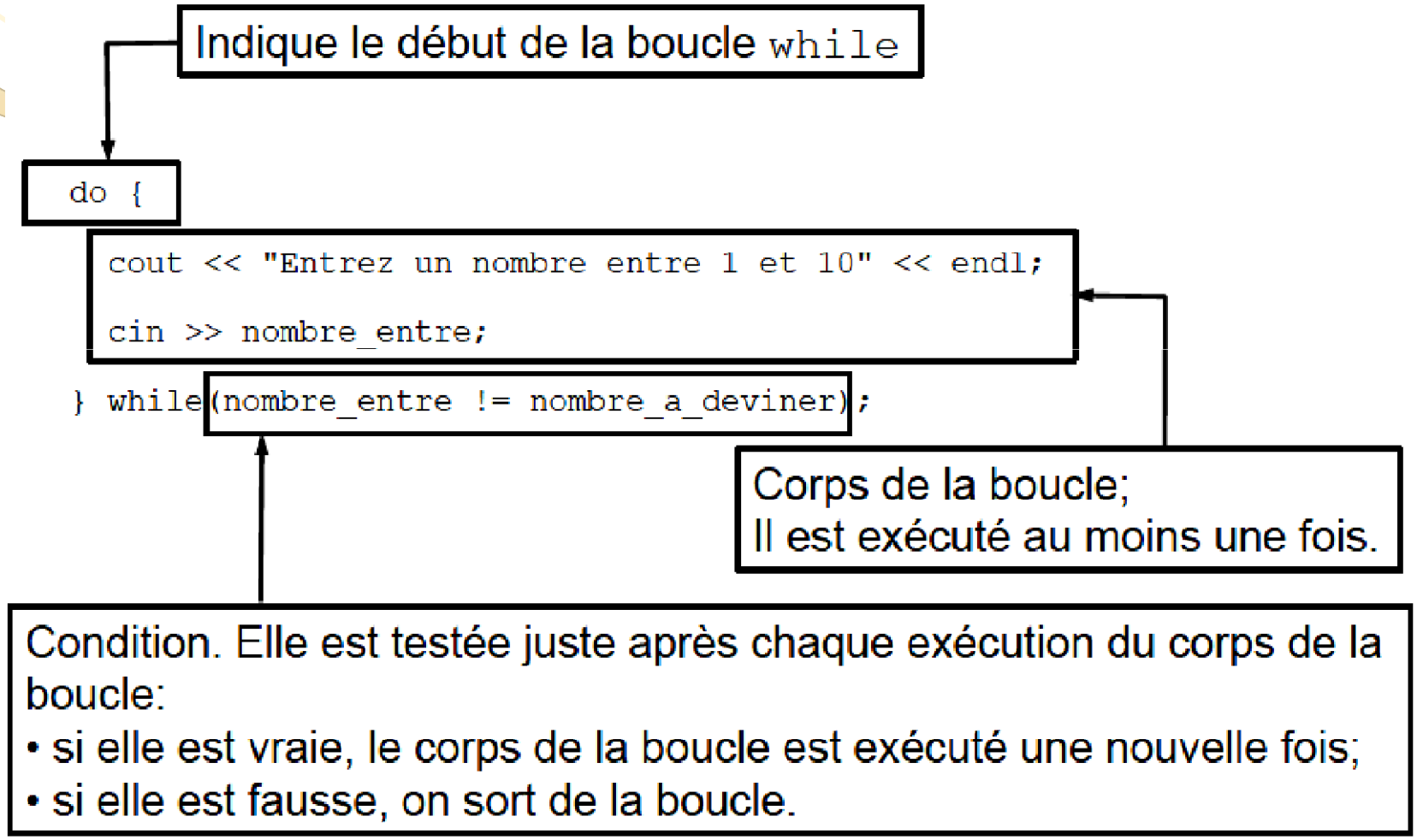
```
do
    operation;
while (test);
```

Mêmes précisions que pour le « `if` » et que pour le « `while` ».

Le « `do-while` » est semblable au « `while` », sauf sur un point : les instructions qui sont mentionnées dans « `operation` » sont exécutées au moins une fois.

C/C++ : les structures de contrôles

«La structure conditionnelle répétitive Do ..While»



C/C++ : les structures de contrôles

«La structure conditionnelle répétitive Do ..While»

```
int main ()
{
   int n=0;
   do
   { cout << n << " ";
      --n;
   }
   while (n>0);
   return 0;
}
```

Instructions exécutées une fois puis une tant que *n* est supérieur à zéro

Résultat affiché :

0

C/C++ : les structures de contrôles

«La structure conditionnelle répétitive Do ..While»

Exercices:

1. Ecrire un programme C++ qui demande à l'utilisateur d'entrer des notes. L'utilisateur indiquera qu'il veut arrêter en entrant la valeur -1. Le programme affichera alors la moyenne des notes.

2. Modifier le programme pour limiter le nombre de notes à 10. L'utilisateur peut toujours entrer -1 s'il souhaite arrêter.

3. Modifier le programme pour qu'il affiche la note la plus grande.

4. Modifier le programme pour qu'il affiche le nombre de notes supérieures à 4.

Qu'affiche le programme suivant ?

```
bool a = true;
if (a)
  cout << "+" << endl;
else
  cout << "-" << endl;

bool b = (1 == 2);
if (b)
  cout << "A" << endl;
else
  cout << "B" << endl;

if (a || b)
  cout << "C" << endl;
if (a && b)
  cout << "D" << endl;

bool c = a || b;
if (c)
  cout << "E" << endl;

if (!c)
  cout << "F" << endl;
```

Qu'affichent les programmes suivants ?

D:

```
bool d = false;
int n = 25;
for(int i = 2; i < 13; i++)
  if (n % i == 0)
    d = true;
if (d)
  cout << "A" << endl;
else
  cout << "B" << endl;
```

E:

Même programmes mais avec
n = 13, 16, 23.

F:

```
bool d = false;
int n = 25, i = 2;
do
{
  if (n % i == 0)
    d = true;
  i++;
} while(!d && i < 13);
if (d)
  cout << "A" << endl;
else
  cout << "B" << endl;
```

C/C++ : les structures de contrôles

«La structure conditionnelle alternative multiple Switch»

Le branchement conditionnel « `switch / case` » :

```
switch (valeur)
{
   case cas1:
     operations1;
     break;              // Facultatif
   …
   case casN:
     operationsN;
     break;              // Facultatif

   default:              // Facultatif
     operationsDefaut;
}
```

Utile pour éviter les « `if – else` » successifs.

C/C++ : les structures de contrôles

«La structure conditionnelle alternative multiple Switch»

Syntaxe de l'instruction `switch`

```
switch(expression)
{
  case constante_1:
    suite d'instructions1
    break;
  case constante_2:
    suite d'instructions2
    break;
  case constante_3:
    suite d'instructions3
    break;
  default:
    suite d'instructions default
}
```

Obligatoirement une expression entière (par exemple de type `int` ou `short`)

Obligatoirement des constantes entières

Attention à ne pas oublier le mot-clé `break` !!

La *suite d'instructions default* est exécutée si aucune des *constante_1, constante_2*... ne correspond à l'*expression*.
Pas obligatoire.

C/C++ : les structures de contrôles

«La structure conditionnelle alternative multiple Switch»

Exercice:

On veut écrire un programme qui demande à l'utilisateur d'entrer le rayon d'un cercle, puis qui lui demande s'il souhaite en obtenir le diamètre, le périmètre ou l'aire:

- Si l'utilisateur entre 1, le diamètre sera affiché.
- Si l'utilisateur entre 2, le périmètre sera affiché.
- Si l'utilisateur entre 3, l'aire sera affichée.

C/C++ : les structures de contrôles

Les ruptures de séquence :

```
continue;
```

Saute à la fin du bloc d'instructions pour un `for`, `do` ou `while`

```
break;
```

Permet de sortir du `for`, `do` ou `while` et de continuer après

```
return;
```

```
return valeur;
```

Quitte la fonction en cours

C/C++ : les structures de contrôles

Exemple :

```
void main()
        {
                int i;
                for (i = 0; i < 5; i++)
                        {
                                if (i == 3)
                                continue;
                                cout<<"i = " << i <<endl;
                        }
                cout<<« Valeur de i a la sortie de la boucle = " << i <<endl;
        }
```

Ce programme imprime :

```
i = 0
i = 1
i = 2
i = 4
```

La valeur de i à la sortie de la boucle = 5.

C/C++ : les structures de contrôles

Par exemple :

```
# include <stdio.h>
void main( )
{
        int i;
        for (i = 0; i < 10; i++)
                {
                        cout<<"i = " << i <<endl;
                        if (i == 3)
                        break;
                }
        cout<<« Valeur de i a la sortie de la boucle = " << i <<endl;
}
```

Ce programme imprime à l'écran :

```
i = 0
i = 1
i = 2
i = 3

La valeur de i à la sortie de la boucle = 3
```