# Le langage C++

| REVISION HISTORY | | | |
|---|---|---|---|
| NUMBER | DATE | DESCRIPTION | NAME |
| 0.4 | novembre 2007 | | ET |

# Contents

1 Rappels sur le langage C 1
1.1 Chaîne de compilation . . . 1
1.2 Types de données . . . 1
1.3 Fonctions . . . 2
1.4 Instructions de contrôle . . . 3
1.5 La récursivité . . . 4
1.6 Structures . . . 4
1.7 Des alias avec `typedef` . . . 4
1.8 Adresses et pointeurs . . . 5
1.9 Gestion de la mémoire . . . 7

2 Débuter en C++ 8
2.1 Entrée/sorties . . . 8
2.2 Commentaires . . . 9
2.3 Espaces de noms . . . 9
2.4 Arguments par défaut . . . 11
2.5 Passage par référence . . . 11
2.6 Les opérateurs `new` et `delete` . . . 12

3 Classes et objets 12
3.1 Des structures étendues . . . 12
3.2 Les classes . . . 15
3.3 Création et destruction . . . 16
3.4 Les opérateurs `new` et `delete` . . . 18
3.5 Membres statiques . . . 18

4 L'héritage 19
4.1 Classes parentes . . . 19

5 Les patrons 20
5.1 Patrons de fonctions . . . 20
5.2 Patrons de classes . . . 22

6 La bibliothèque standard 23
6.1 Introduction . . . 23
6.2 Les flux d'entrée/sortie . . . 23

Ce document sert de support aux TPs de C++ effectués en troisième année d'électronique à l'École polytechnique universitaire (ÉPU) de l'Université de Nice Sophia-Antipolis. Ce cours ainsi que les travaux pratiques qui l'accompagnent sont disponibles sur http://emilien.tlapale.com/cours-cpp.

Creative Commons License
Ce document est mis à disposition selon les termes de la licence Creative Commons Paternité-Pas d'Utilisation Commerciale-Partage des Conditions Initiales à l'Identique 2.0 France.

# 1 Rappels sur le langage C

## 1.1 Chaîne de compilation

Différentes étapes sont nécessaires pour passer d'un code source en C écrit par le programmeur à un programme exécutable sur l'ordinateur.

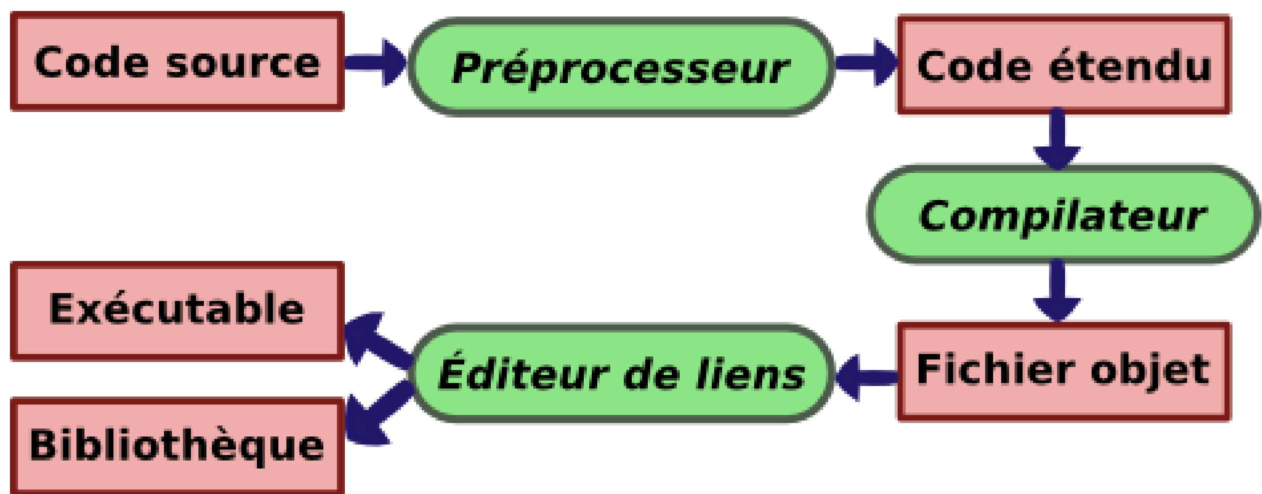



Figure 1: Chaîne de compilation

- Le *préprocesseur* effectue des actions sur les éléments lexicaux du code source permettant la prise en compte des inclusions de fichier, de la compilation conditionelle, de l'expansion des macro, ...
- Le *compilateur* proprement dit transforme le code source lisible par un humain en instructions binaires interprétées par le processeur de l'ordinateur cible.
- L'*éditeur de liens* va gérer les appels entre plusieurs fichiers sources compilés indépendamment ou avec des bibliothèques externes.

Dans le cas de programmes utilisant des bibliothèques externes (les `.so` sous GNU/Linux ou les `.dll` sous Windows) une dernière étape de chargement des bibliothèques semblable à l'édition des liens s'effectue lors de l'exécution du programme.

Les compilateurs et les interfaces graphiques de développement (IDE) cachent généralement ces étapes à l'usager mais les connaître s'avère souvent utile.

## 1.2 Types de données

Les programmes C travaillent sur des données possédant chacunes un type particulier : entiers, flottants, adresses, structures, ... Les valeurs entières sont les `char`, les `short`, les `int` et les `long`. Celles-ci sont *signées* : elles peuvent prendre des valeurs négatives. Les `char` occupent par exemple un octet et peuvent prendre des valeurs de `-128` à `127`. Si on n'utilise pas les valeurs négatives on peut utiliser des types signés qui vont déplacer les limites d'un type donné. Par exemple un `unsigned char` pourra prendre une valeur entre `0` et `255`. La déclaration d'une variable d'un type donné, c'est à dire la réservation d'un emplacement mémoire d'une taille suffisante pour stocker les données, se fait comme indiqué dans le code suivant :

```
int a;
unsigned char b;
unsigned long ma_variable;
```

Les emplacements mémoires réservés sont alors accessibles par les noms spécifiés : `a`, `b` ou `ma_variable`.

Table 1: Exemples de types en C

| Type | Description |
|---|---|
| `long` | Entier signé au moins aussi grand qu'un `int`. |
| `int` | Entier signé au moins aussi grand qu'un `short`. |
| `short` | Entier signé au moins aussi grand qu'un `char`. |
| `char` | Entier signé de 8 bits. |
| `unsigned int` | Pareil qu'un `int` mais non signé. |
| `float` | Flottant simple précision. |
| `double` | Flottant double précision. |
| `char*` | Pointeur sur un `char`. |
| `struct bla_t` | Type nommé `bla_t` définit à partir d'autres types. |

Deux types flottants, `float` pour la simple précision et `double` pour la précision double, permettent de manipuler des nombres à virgule fixe.

Pour affecter une valeur à une variable on utilise l'opérateur `=`. Ce qui va d'ailleurs nous pousser à utiliser l'opérateur `==` pour tester des égalités. Les opérateurs arithmétiques classiques tels que `*` pour le produit ou `/` pour la division sont accessibles et des parenthèses permettent de spécifier des priorités.

```
int a;
double b;

a = 123 + 2;
b = 3.14159 / (a + 3);
```

### 1.3 Fonctions

L'unité logique de calcul d'un programme en C est la *fonction*. Chaque fonction effectue un certain calcul sur des données, les *arguments* ou *paramètres*, et renvoie éventuellement un résultat. Les données des programmes, par exemple les arguments d'une fonction, possèdent toutes un *type*.

Le code suivant présente un simple programme C compilable avec deux fonctions : la fonction `carre()` qui prend un entier de type `int` et retourne son carré ; la fonction `main()` obligatoire pour un programme exécutable qui correspond au *point d'entrée* du programme, i.e. la première fonction appelée.

```
#include <stdio.h>

int
carre(int a)
{
  return a * a;
}

int
main(int argc, char** argv)
{
  int a;
  int b;
```

```
  a = 3;
  b = carre(a);

  printf("Le carré de %d est %d\n", a, b);
  return 0;
}
```

La fonction `main` prend en entrée les arguments passés au programme et retourne un entier de valeur zéro si l'exécution s'est déroulée normalement, un code d'erreur sinon.

Une fonction retourne une valeur à la fonction appellante en utilisant l'instruction `return`. Toutes les instructions après un `return` sont ignorées.

## 1.4 Instructions de contrôle

Pour contrôler le flux d'exécution d'un programme diverses instructions existent. Nous commençons par présenter l'exécution conditionnelle avec `if` qui exécute une partie du code si une condition est vérifiée. On peut utiliser l'instruction `else` pour effectuer une autre opération si la condition du `if` n'est pas vérifiée.

```
#include <stdio.h>

void
affiche_parite(int a)
{
  // Teste le reste de la division entière par 2
  if (a % 2 == 0)
    {
      printf("%d est pair\n", a);
    }
  else
    {
      printf("%d est impair\n", a);
    }
}

int
main()
{
  affiche_parite(3);
  affiche_parite(200);
  return 0;
}
```

Tout comme pour les fonctions les instructions à exécuter conditionellement sont délimités par des accolades : `{` et `}`. Un ensemble d'instructions entre accolades est appelé un *bloc d'instructions*. Ils peuvent être imbriqués les uns dans les autres si cela est nécessaire. Notez que dans notre cas nous n'avons qu'une seule instruction dans chaque bloc. Le cours de `affiche_parite()` peut être réécrit de la façon suivante :

```
if (a % 2 == 0)
  printf("%d est pair\n", a);
else
  printf("%d est impair\n", a);
```

L'instruction de boucle `while` va exécuter une instruction ou un bloc d'instruction tant qu'une condition est vérifiée.

```
unsigned int
factorielle(unsigned int n)
{
  unsigned int ans = 1;
```

```
  while (n > 1)
    {
      ans = ans * n;
      n = n - 1;
    }
  return ans;
}
```

### 1.5 La récursivité

Peu utilisée dans les débuts de l'informatique car la mémoire était très limitée et les compilateur ne l'optimisaient pas correctement, la récursivité est une manière élégante de résoudre les problèmes. Tout code récursif peut être cependant transformé en non récursif en utilisant des boucles. Un exemple très simple est le calcul de la factorielle. On sait que `0! = 1` et `n! = n × (n - 1)!`.

```
unsigned int
factorielle(unsigned int n)
{
  if (n == 0)
    return 1;

  /* Le 'else' est inutilise ici puisque
     le 'return' précédent retourne sans exécuter la suite */

  return n * factorielle(n - 1);
}
```

### 1.6 Structures

Les structures permettent de créer des types personalisés à partir de types prédéfinis. Elles sont en fait des agreggats d'autre types de base ou d'autre structures. Chaque nouveau type de structure possède un nom ainsi que des champs qui en sont les éléments et qui sont eux-même nommés.

```
struct Complexe
{
  double reel;
  double imaginaire;
};
```

Le code précédent définit un nouveau type pour représenter les nombres complexes qui sont donc constitué d'une partie réelle et d'une partie imaginaire. Pour accéder à l'un des champs d'une structure on utilise l'opérateur point `.`.

```
struct Complexe
addition_complexe(struct Complexe c1, struct Complexe c2)
{
  struct Complexe res;

  res.reel = c1.reel + c2.reel;
  res.imaginaire = c1.imaginaire + c2.imaginaire;

  return res;
}
```

### 1.7 Des alias avec `typedef`

Il peut arriver que l'on souhaite donner un nom plus explicite à un type, pour ce faire on utilise l'opérateur `typedef` qui va lui associer un alias.

```
typedef int entier;
typedef int* pointeur_sur_entier;
typedef struct Complexe Complexe;
```

C'est tout de même un opérateur à utiliser avec parcimonie et surtout pas dans les deux premiers cas. Le dernier alias permet de ne pas avoir à indiquer toujours `struct` en C, c'est cependant automatique en C+\+ où l'on peut déclarer `struct Complexe` et utiliser juste `Complexe`.

Les `typedefs` sont par exemple utilisés pour définir des types dont on garantit le nombre de bits, en particulier dans `stdint.h`. Parmis les types qui y sont définit on trouve `uint32_t` qui représente les entiers non signés sur 32 bits, `int8_t` qui représente les entiers signés sur 8 bits, etc. En effet le C ne définit pas avec précision la taille en bits des types et elle change en fonction des architectures.

## 1.8 Adresses et pointeurs

Les ordinateurs actuels possèdent une mémoire vive, appelée la RAM (*Random access memory*) utilisée par les programmes pour stocker le code et les données. Ainsi lorsque l'on écrit `int a;` pour déclarer un entier signé nommé `a` on réserve en fait un emplacement de la mémoire d'une taille suffisante pour y stocker un `int`. La mémoire peut être vue comme une bande de cases dans lesquelles on peut accèder et modifier des nombres. Pour accéder à une case mémoire il suffit de connaître son *adresse*, une sorte d'index.

```
int
main()
{
  int a;
  int* ptr;

  a = 123;
  ptr = &a;

  return 0;
}
```

Le bout de code précédent commence par déclarer deux variables nommées `a` et `ptr` et respectivement de type `int` et `int*`. On suppose qu'une variable de type `int` prend 4 octets et qu'une variable de type `int*` qui est un pointeur sur un `int` prend aussi 4 octets. Ces deux première lignes alloue donc deux blocs de 4 octets chacun. On suppose que la case-mémoire de `a` est à l'adresse `0x420010` et que celle de `ptr` est à l'adresse `0x420014`.

L'instruction `a = 123;` place la valeur `123` dans la case-mémoire réservé pour `a` par le compilateur dans la pile de la fonction (puisque c'est une variable locale). L'expression `&a` va récupérer non pas la valeur contenue dans la case mémoire de `a`, soit `123`, mais l'adresse de cette case-mémoire, soit `0x420010`. Cette adresse, qui est aussi une valeur numérique, est placée dans la case mémoire réservée pour `ptr`.

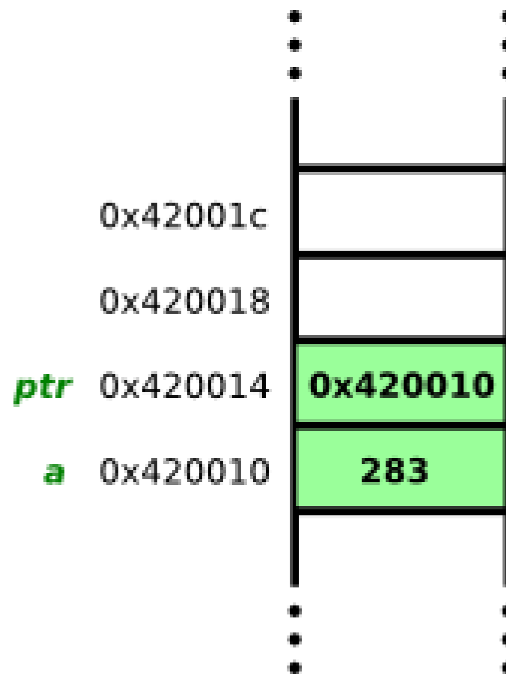



Figure 2: Utilisation de la mémoire dans la pile locale

Maintenant que l'on dispose de l'adresse de `a` dans la case-mémoire de `ptr` on peut modifier de façon indirecte. L'opérateur unaire `*`, à ne pas confondre avec l'opérateur binaire `*` pour la multiplication, permet de lire et de modifier une variable dont on connaître l'adresse. Par exemple en utilisant le code suivant, on mettre à jour la valeur contenue dans la case mémoire à l'adresse `0x420010` par `398`.

```
printf("ptr vaut: 0x%lx\n", (unsigned long) ptr); // Affiche 0x420010
printf("*ptr vaut: %d\n", *ptr); // Affiche 283

*ptr = *ptr + 115;
printf("a vaut maintenant: %d\n", a); // Affiche 398
```

Si l'on avait connu l'adresse `0x420010` définie par le compilateur on aurait même pu directement écrire :

```
*((int*) 0x420010) = *((int*) 0x420010) + 115;
```

La façon standard de gérer des tableaux en C consiste à utiliser des zones contigues de mémoire et à dénoter le tableau comme étant l'adresse du premier élément. Les chaînes de caractères sont un exemple de tels tableaux : les caractères sont stockés dans des cases consécutives avec un caractère `\0` nul qui termine la chaîne et elle est utilisée en donnant l'adresse du premier caractère.

```
const char *str = "Hello world!";

printf("*str = %c\n", *str); // Affiche 'H'
printf("*(str + 4) = %c\n", *(str + 4)); // Affiche 'o'
```

**Note**
Lorsqu'on définit une chaîne de caractère dans le code en l'entourant de guillemets double comme dans l'exemple précédent, le compilateur rajoute automatiquement un caractère nul à la fin.

Une syntaxe alternative à `*(tabl + pos)` consiste à utiliser `tabl[pos]` qui est plus proche des autres langages de programmation.

**Warning**
Les tableaux en C sont de simples adresses mémoire, des pointeurs qui en indiquent le début. Par conséquent ils ne disposent pas d'une information concernant leur taille, pas plus qu'un éventuel débordement de tableau n'est géré. Ainsi un accès à l'élement -123 ou 5998 d'un tableau de 10 éléments ne sera pas signalé de façon explicite à l'exécution et votre code pourrait bien continuer à s'exécuter avec des erreurs bizarres.

```
/**
 * Recherche une valeur dans un tableau d'entiers.
 * La taille du tableau doit évidemment être indiquée en argument.
 * Si aucun élément de cette valeur n'est trouvé, retourne -1.
 */
int
recherche_int (int valeur, int* tableau, int taille)
{
  int i;

  for (i = 0; i < taille; i++)
    if (tableau[i] == valeur)
      return i;

  return -1;
}
```

## 1.9 Gestion de la mémoire

Il existe différents types d'allocation de mémoire en fonction de la déclaration des variables dans le code :

- les variables globales, celles définies à l'extérieur de toute fonction, sont accessible à tout moment et leur emplacement n'est jamais libéré ;
- les variables locales à une fonction ou à un bloc d'instructions ne sont utilisables que dans ce bloc, même si on peut renvoyer leur adresse en sortie de fonction, elle sera inutilisable ;
- les variables du tas, allouées par `malloc()` et co., sont disponible tant qu'elles ne sont pas explicitement libérées par un `free()`.

L'allocation par `malloc()` va donc permettre d'allouer un espace mémoire dont la taille ne sera connu qu'à l'exécution. Par exemple lorsque vous définissiez une chaîne de caractère par `char* str = "Hello World!";` vous réserviez un espace mémoire suffisamment grand pour contenir les 13 caractères dont le caractère `\0` qui termine la chaîne. Si vous voulez rajouter un caractère à la chaîne par la suite, vous ne pouvez pas, car l'espace mémoire situé après peut contenir des données importantes (voire être innacessible).

```
/* Cette fonction concatène deux chaînes de caractères. */
char*
concatenne_chaines(const char* s1, const char* s2)
{
  char* str;
  int i;
  int len_s1, len_s2;

  /* Réserve suffisamment de mémoire + \0 */
  len_s1 = strlen(s1);
  len_s2 = strlen(s2);
  str = (char*) malloc(len_s1 + len_s2 + 1);

  /* Vérifie si l'allocation a réussie */
  if (str == NULL)
```

```
    return NULL;

  /* Recopie les contenu */
  for (i = 0; i < len_s1; i++)
    str[i] = s1[i];
  for (i = 0; i < len_s2; i++)
    str[len_s1 + i] = s2[i];

  /* N'oublie pas le caractère nul final */
  str[len_s1 + len_s2] = '\0';

  return str;
}
```

**Warning**
Veillez à bien gérer tous vos `malloc()` en libérant la mémoire une fois son utilisation terminée. Sans cela vous vous exposez à des fuites de mémoire pouvant consommer l'intégralité de ce qui est disponible pour votre programme, le rendant inutilisable.

```
/* Cette fonction retourne un pointeur sur variable locale, elle ne
   doit en aucun cas être utilisée~! */
int*
f()
{
  int a = 3;
  return &a;
}

/* Cette fonction alloue un tableau de 'taille' entiers. */
int*
cree_tableau(int taille)
{
  int* tab;

  tab = (int*) malloc(taille * sizeof(int));
  if (tab == NULL)
    {
      fprintf(stderr, "Erreur fatale, plus de mémoire disponible!\n");
      exit(1);
    }

  return tab;
}
```

**Note**
Il est possible d'utiliser un *garbage collector* en C (et C+\+) qui va créer des zones de mémoire qui seront automatiquement libérées lorsqu'elles deviendront inacessible. Par exemple en utilisant celui de Boehm. Notez toutefois que les *garbage collectors* ne font pas partis du standard C, ce ne sont que des bibliothèques supplémentaires.

# 2 Débuter en C++

## 2.1 Entrée/sorties

Le C++ arrive avec de nouvelles bibliothèques en particulier celles d'entrées/sorties qui utilisent les nouveaux opérateurs.

```
#include <iostream>

using namespace std;

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

Dans l'exemple ci-dessus on inclut l'en-tête `iostream` qui contient les définitions de base pour les entrées sorties. On indique ensuite d'accéder aux fonctions de l'espace de nommage (ou *namespace*) `std` sans avoir besoin de le spécifier explicitement. La fonction `main`, point d'entrée du programme, va ensuite utiliser l'opérateur << de `cout` pour y envoyer une chaîne de caractère puis une fin de ligne via `end`. Nous reviendrons sur ces différents points par la suite.

Sachez qu'il est possible d'envoyer d'autres données sur la sortie standard via l'opérateur << de `cout`. Pour lire depuis l'entrée standard utilisez l'opérateur >> sur `cin`.

```
#include <iostream>

using namespace std;

int
main()
{
  int a;
  int b;

  cout << "Entrez un nombre: ";
  cin >> a;

  b = a * a;
  cout << "Le carré de " << a << " est " << b << endl;

  return 0;
}
```

## 2.2 Commentaires

En plus des commentaires multi-lignes délimités par `/*` et `*/` du C89, le C+\+ possède également des commentaires monoligne débutant par `//` et continuant jusqu'en fin de ligne comme en C99.

```
int a; /* commentaire */
double b; // encore un commentaire valide

a = 1 + /* 5 // */ 3;
// Ci-dessus a prendra la valeur 4.
```

## 2.3 Espaces de noms

Vous avez déjà pu constater que nous utilisions l'instruction `using namespace std;`. Celle-ci permet d'accéder aux fonction de l'*espace de noms* `std` de la bibliothèque standard sans avoir besoin de le spécifier. De façon explicite on aurait put écrire :

```
#include <iostream>

int
```

```
main()
{
  int a;
  int b;

  std::cout << "Entrez un nombre: ";
  std::cin >> a;

  b = a * a;
  std::cout << "Le carré de " << a << " est " << b << std::endl;

  return 0;
}
```

Notez que dans ce cas chaque utilisation d'une fonction ou d'un objet de l'espace de noms_ `std` doit être préfixé de `std::` où l'opérateur `::` est l'opérateur de résolution de portée.

Ainsi on peut définir plusieurs fonctions de même nom mais dans des espaces de noms différents. Cela est en particulier très utile pour les bibliothèques qui seront évidemment utilisées par d'autres personnes. Là où en C on devait écrire par exemple `malib_mafonction()`, dans une bibliothèque C+\+ on écrira `malib::mafonction()`. En utilisant un `using namespace malib;` on pourra alors supprimer tous les `malib::` et utiliser par exemple `mafonction()` directement. Évidemment si deux espaces de noms possèdent des fonctions de même nom il faudra utiliser l'opérateur de résolution de portée pour éviter des conflits. En C lorsque deux fonctions avaient des fonctions de même nom elles étaient simplement incompatibles.

```
#include <iostream>

namespace ns_a
{
  int
  f(int a, int b)
  {
    return a * b;
  }
}

namespace ns_b
{
  int
  f (int a, int b)
  {
    return a + b;
  }
}

using namespace std;
using namespace ns_b;

int
main()
{
  cout << "f(1,2) = " << f(1, 2) << endl; // Affiche 3
  cout << "ns_a::f(1,2) = " << ns_a::f(1, 2) << endl; // Affiche 2
  cout << "ns_b::f(1,2) = " << ns_b::f(1, 2) << endl; // Affiche 3
  return 0;
}
```

**Important**

Veillez à ne pas utiliser d'instruction `using namespace` dans des en-têtes car vous polluriez l'espace de noms courant de tout code incluant cet en-tête.

## 2.4 Arguments par défaut

Tout comme en C, les fonctions doivent être déclarées, ne serait-ce que par leur prototype, avant d'être utilisée. On peut cependant intégrer des arguments par défaut dans les prototypes de façon à simplifier les appels. Les arguments par défaut sont très utile pour régler certains paramètres qui sont le plus souvent utilisés ou que l'on n'oblige pas à calculer.

```
// Fonction qui fait quelque chose avec une chaîne de caractères
void
ma_fonction(const char* str, size_t len = -1)
{
  size_t str_len = len;

  // Si on ne passe pas la taille de la chaîne
  // alors on la calcule (terminée par \0)
  if (len < 0)
    str_len = strlen (str);

  // ...
}
```

## 2.5 Passage par référence

Les *références* permettent d'utiliser de façon automatique des pointeurs, soit des références vers un autre emplacement mémoire. Lors de la déclaration d'une référence on lui affecte un objet du même type vers lequel elle pointe. La référence agit alors comme un alias de cet objet (et non pas comme une copie).

```
#include <iostream>

using namespace std;

void
style_c(int* a)
{
  *a = 3;
}

void
style_cpp(int& a)
{
  a = 5;
}

void
style_qui_marche_pas(int a)
{
  a = 18;
}

int
main()
{
  int x  = 9;
  int& y = x;

  cout << x << endl; // --> 9
  style_c(&x);
  cout << x << endl; // --> 3
  style_cpp(x);
  cout << x << endl; // --> 5
  style_qui_marche_pas(x);
```

```
  cout << x << endl; // --> 5

  cout << y << endl; // --> 5
}
```

Le passage par référence allège non seulement l'utilisation des références qui s'utilisent comme une variable classique, sans avoir besoin d'utiliser l'indirection en préfixant par `*`, mais aussi l'appel des fonctions qui en utilise puisque le compilateur se charge de tout, plus besoin de passer l'adresse avec `&`.

### 2.6 Les opérateurs `new` et `delete`

Lorsque vous souhaitiez allouer de la mémoire sur le tas en C vous pouviez utiliser les fonctions `malloc()` et `calloc()`. Les données du tas restant accessibles jusqu'à ce qu'elles soient libérées par un `free()`. Le C+\+ rajoute les opérateurs `new` et `delete` qui seront très utile pour la manipulation des objets comme nous le verrons au chapitre suivant.

```
// Alloue un tableau de 3 int
int* a = (int *) malloc (sizeof (int) * 3);

// Alloue un seul int
int b = new int;

// Alloue 3 int
int* c = new int[3];

// Libérations
free (a);
delete b;
delete [] c;
```

Les opérateurs `new[]` et `delete[]` sont utilisés pour allouer et libérer des tableaux.

**Warning**
Une zone mémoire allouée par `malloc()` ne doit en aucun cas être libérée par un `delete`, de même qu'une zone allouée par `new` ne doit pas être libérée avec `free()` et qu'une zone allouée par `new[]` ne doit pas être libérée avec `delete`.

## 3 Classes et objets

### 3.1 Des structures étendues

En C vous pouviez définir vos propres types à l'aide de structures. Pour les manipuler vous deviez créer des fonctions auxquelles vous passiez votre structure en argument.

```
#include <stdio.h>


struct Complex
{
  double real;
  double imaginary;
};

void
complex_add_to(struct Complex* dst, const struct Complex* src)
{
```

```
  dst->real += src->real;
  dst->imaginary += src->imaginary;
}

int
main()
{
  struct Complex a = {1.0, 2.0};
  struct Complex b = {3.0, 4.0};

  complex_add_to(&a, &b);
  printf("a + b = (%f, %f)\n", a.real, a.imaginary);

  return 0;
}
```

What is new in C+\+ is that you can really associate the `complex_add_to()` function to the `Complex` structure. The function is then called a *method* of the `Complex` structure.

```
#include <iostream>

using namespace std;

struct Complex
{
  double real;
  double imaginary;

  void
  add_to(const struct Complex* src)
  {
    this->real += src->real;
    this->imaginary += src->imaginary;
  }
};

int
main()
{
  Complex a = {1.0, 2.0};
  Complex b = {3.0, 4.0};

  a.add_to(&b);
  cout << "a + b = (" << a.real << "," << a.imaginary << ")" << endl;

  return 0;
}
```

Comme vous pouvez le constater la syntaxe est similaire. Cependant la fonction d'addition est désormais attaché à la structure `Complex`, on n'a alors pas besoin de préfixer son nom de `complex_` comme on le faisait en C puisque ce nom de fonction ne sera utilisé que pour les `Complex`. L'appel ne peut être ambigu puisque l'on utilise `structure.methode()` et que `methode` ne sera recherché que parmis les méthodes de la structure `structure`.

Une méthode de structure se voit automatiquement doté d'un objet `this` qui représente la structure sur laquelle ma méthode est invoqué. Dans notre cas `this` sera un pointeur sur l'objet `a` de la fonction `main()` puisqu'on a utilisé `a.add_to()`. Plus besoin de passer explicitement cet objet.

Sachez que l'on sépare le plus souvent le code source en fichiers `.h` pour les en-têtes et en fichiers `.cc` ou `.cpp` pour l'implémentation. On déclare alors simplement les prototypes dans l'en-tête comme en C.

```
#ifndef __COMPLEX_H
#define __COMPLEX_H
```

```
struct Complex
{
  double real;
  double imaginary;

  void add_to(const Complex* src);
};

#endif
```

On utilise alors l'opérateur de résolution de portée dans l'implémentation. Notez que l'on n'est pas obligé d'utiliser le mot clé `struct` pour référencer un type structure à l'utilisation.

```
#include "complex.h"

void
Complex::add_to(const Complex* src)
{
  this->real += src->real;
  this->imaginary += src->imaginary;
}
```

En effet on a l'habitude de séparer les déclaration de fonction ou de méthode de leur implémentation, comme en C. Ceci permettant de fournir à un utilisateur de notre bibliothèque uniquement les en-têtes qui contiennent les déclaration, avec le code déjà compilé pour l'implémentation. Cette programmation modulaire permet aussi de modifier simplement une implémentation, voire de la remplacer totalement par une autre, sans rien changer au reste du code.

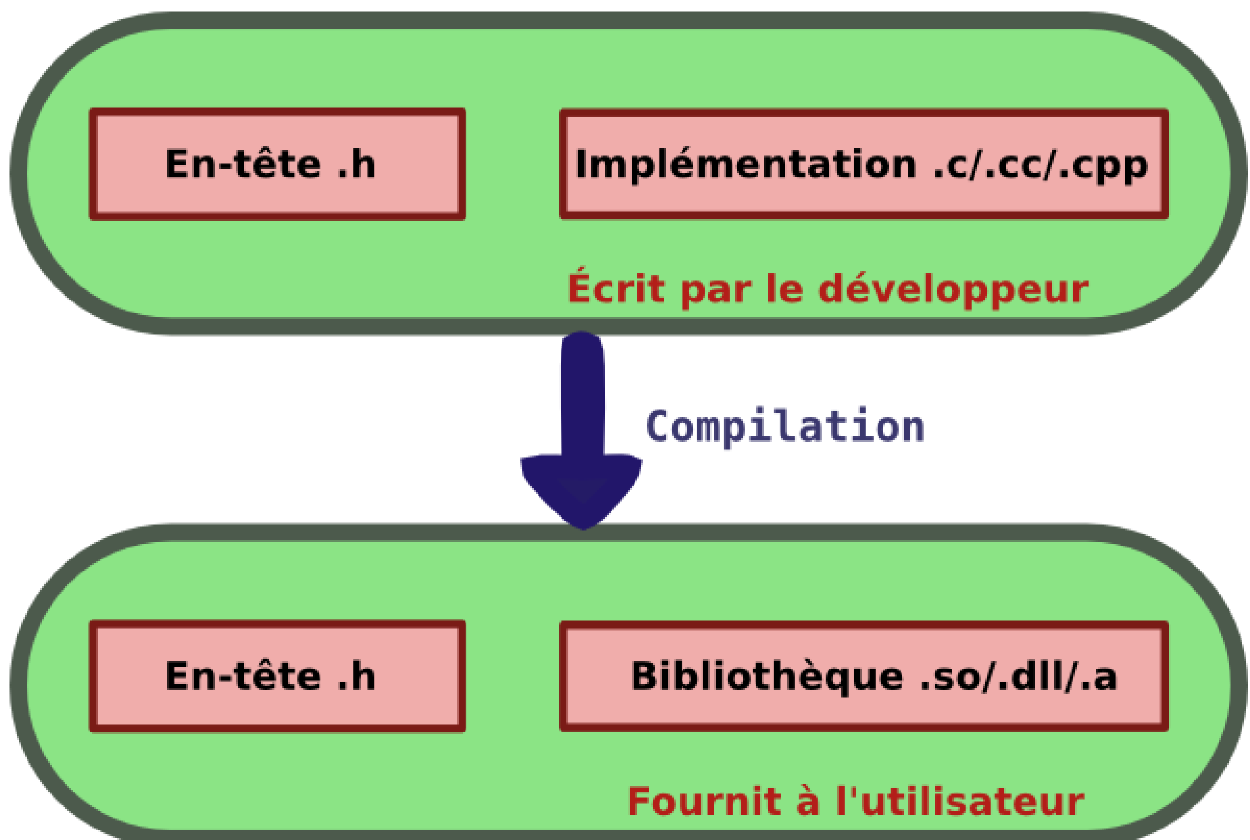



Figure 3: Programmation modulaire

### 3.2 Les classes

Les structures issues du C ne sont qu'un cas particuliers de *classes* que l'on retrouve dans d'autres langages qui permettent la programmation orientée objet tel que le Python, le Java ou le C#. En particulier les structures sont des classes dont tous les champs, `real` et `imaginary` dans notre exemple précédent, sont accessibles.

Une classe définit un type non simplement à partir de ses données mais à partir de méthodes qui peuvent être appelées. On pourrait par exemple définir une classe boîte aux lettres de la façon suivante :

```
class BoiteAuxLettres
{
public:
  /** Rajoute une lettre dans la boîte aux lettres. */
  void posteLettre(Lettre l);

  /** Retourne le nombre de lettres dans la boîte. */
  int nombreLettres();

  /** Place toutes les lettres dans le sac donné et vide la boîte. */
  void recupereLettres(Sac& s);

private:
  /** Nombre de lettres dans la boîte. */
  int m_nombre_lettres;

  /** Les lettres sous forme de tableau C. */
  Lettre* m_lettres;
};
```

La boîte aux lettres est alors principalement définie à partir de ce qu'elle permet de faire et non à partir de ce qu'elle contient réellement. Les mot-clés `public:` et `private:` permettent de définir si les champs ou méthodes qui suivent sont accessibles depuis l'extérieur ou non. Dans ce cas seules les méthodes sont accessibles, les champs `m_nombre_lettres` et `m_lettres` étant uniquement utilisé à l'intérieur de la boîte aux lettres.

```
void
BoiteAuxLettres::posteLettre(Lettre l)
{
  // On augmente la taille du tableau de lettre
  // et on ajoute la nouvelle lettre à la fin
  m_nombre_lettres = m_nombre_lettres + 1;
  m_lettres = (Lettre*) realloc(m_lettres,
                                sizeof(Lettre) * m_nombre_lettres);
  m_lettres[m_nombre_lettres - 1] = l;
}
```

Dans la section précédente nous avons présente les méthodes comme un raccourcis pour l'écriture de fonctions associées à une structure. Le mot-clé `this` permettant d'accéder à l'objet surlequel la méthode est invoquée. Dans l'exemple ci-dessus, nous utilisons directement `m_nombre_lettres` et `m_lettres` sans utiliser `this→m_nombre_lettres`. En effet, au sein d'une méthode, le compilateur va rechercher les symboles d'abord dans les variables locales au bloc, puis dans les arguments de la méthode, puis dans les champs (ou membres) de la classe de la méthode, et enfin dans les objets globaux. On peut donc accéder directement aux champs de la classe.

```
int
BoiteAuxLettres::nombreLettres()
{
  return m_nombre_lettres;
}
```

Si n'importe quelle autre fonction pouvait accéder au champ `m_nombre_lettres` qui contient le nombre de lettre dans la boîte aux lettres, alors il pourrait y avoir des erreurs et des incohérences, comme un nombre négatif comme valeur qui pourrait faire crasher la méthode `posteLettre()`.

```
void
BoiteAuxLettres::recupereLettres(Sac& s)
{
  // Remplit le sac et efface les lettres de la boîte
  s.remplit(m_nombre_lettres, m_lettres);
  free(m_lettres);
  m_lettres = NULL;
  m_nombre_lettres = 0;
}
```

### 3.3 Création et destruction

Comme les structures et les types de base en C, les instances de classe doivent être créées. Il vous suffit évidemment de déclarer un objet comme une structure, par exemple au sein d'une fonction, pour qu'une place en mémoire lui soit réservée.

```
int
main()
{
  BoiteAuxLettres b;

  // Les deux lignes suivantes ne marchent pas
  b.m_nombre_lettres = 0;
  b.m_lettres = NULL;

  // ...
  b.recupereLettres();

  return 0;
}
```

Par contre, à cause de l'accessibilité des champs de l'objet `b` instance de la classe `BoiteAuxLettres`, on ne peut pas écrire `b.m_lettres`. En effet `m_lettres` est une donnée privée, donc seulement accessible à l'intérieur de la classe `BoiteAuxLettres`, soit à l'intérieur de ses méthodes.

Pour pallier aux problèmes d'accessibilité à l'initialisation, et parce qu'il est beaucoup plus logique que la classe `BoiteAuxLettres` soit elle-même responsable de l'initilisation de ses instances, il existe des *constructeurs*. Ces méthodes spéciales sont invoquées lorsqu'un objet est créé et servent entre autres à sont initialisation.

```
class BoiteAuxLettres
{
public:
  BoiteAuxLettres()
  // ...
};

BoiteAuxLettres::BoiteAuxLettres()
{
  m_lettres = NULL;
  m_nombre_lettres = 0;
  std::cout << "Une boîte aux lettres créée" << std::endl;
}
```

Les constructeurs sont des méthodes qui ne retourne rien, donc on ne spécifie même pas `void`, et qui porte le nom de leur classe.

```
#include "boiteauxlettres.h"

int
main()
{
```

```
  BoiteAuxLettres b;  // Provoque l'affichage de «~Une boîte ...~»
  Lettre l1, l2;

  b.posteLettre(l1);
  b.posteLettre(l2);

  cout << b.nombreLettres() << " lettres postée" << endl;

  return 0;
}
```

Notez qu'il est possible de passer des paramètre au constructeur. Par exemple si vous souhaitez donner un nom à vos boîte aux lettres, vous pouvez utiliser `std::string`, la classe standard de C++ dédiée à la manipulation de chaîne de caractères.

```
class BoiteAuxLettres
{
public:
  BoiteAuxLettres(const std::string& nom);
  // ...
protected:
  std::string m_nom;
  // ...
};

BoiteAuxLettres::BoiteAuxLettres(const std::string& nom)
{
  m_nom = nom;
  m_lettres = NULL;
  m_nombre_lettres = 0;
}
```

On a remplacé le constructeur sans argument par un constructeur avec argument. Il n'est alors plus possible d'utiliser le constructeur sans argument et on devra obligatoirement utiliser une déclaration du type `BoiteAuxLettres b("Boite A");` et non `BoiteAuxLettres b;`. Vous pouvez néanmoins laisser les deux constructeurs un avec argument et l'autre sans, ou utiliser un argument par défaut comme vu au chapitre précédent.

```
class BoiteAuxLettres
{
public:
  BoiteAuxLettres(const std::string& nom = "Anonyme");
  // ...
};

// L'argument par défaut se place juste dans le prototype
BoiteAuxLettres::BoiteAuxLettres(const std::string& nom)
{
  // ...
}

int
main()
{
  BoiteAuxLettres b1; // Appelle BoiteAuxLettres("Anonyme");
  BoiteAuxLettres b2("Martine");

  // ...
}
```

Vous pouvez aussi déclarer des métodes qui seront invoquées lorsque un objet sera détruit. Dans notre cas, `m_lettres` est un bloc mémoire alloué sur le tas avec `realloc()`. Il reste donc disponible tant qu'il n'est pas libéré explicitement avec `free()` dont l'oubli peut provoquer des fuites de mémoire. Le *constructeur* dont le nom correspond à celui de la classe préfixé d'un tilde ~ est là pour ça.

```
class BoiteAuxLettres
{
public
  ~BoiteAuxLettres();
  // ...
};

BoiteAuxLettres::~BoiteAuxLettres()
{
  if (m_lettres != NULL)
    free(m_lettres);
}
```

### 3.4 Les opérateurs `new` et `delete`

Au chapitre précédent nous avons vu les opérateurs `new` et `delete` qui permettaient respectivement d'allouer de la mémoire sur le tas et de la libérer comme le faisaient les fonctions `malloc()` et `free()` en C bien que l'on n'ait pas à spécifier la taille. Il est possible de faire de même avec les classes.

```
int
main()
{
  BoiteAuxLettres* b1;
  BoiteAuxLettres* b2;

  b1 = new BoiteAuxLettres();
  b2 = new BoiteAuxLettres("Lucie");

  // ...
  b1->posteLettre(...);

  // ...
  delete b1;
  delete b2;

  return 0;
}
```

### 3.5 Membres statiques

Alors que les champs des classes sont différents pour chaque instance, tout comme pour les structures en C, il est possible de créer des variables partagées par toutes les instances d'une classe. Il s'agit donc de variables appartenant à une classe, on les nomme des membres statiques.

```
class BoiteAuxLettres
{
public:
  static int nombre_boites;
  // ...
};


// Implémentation

int BoiteAuxLettres::nombre_boites = 0;

BoiteAuxLettres::BoiteAuxLettres(const std::string& nom)
```

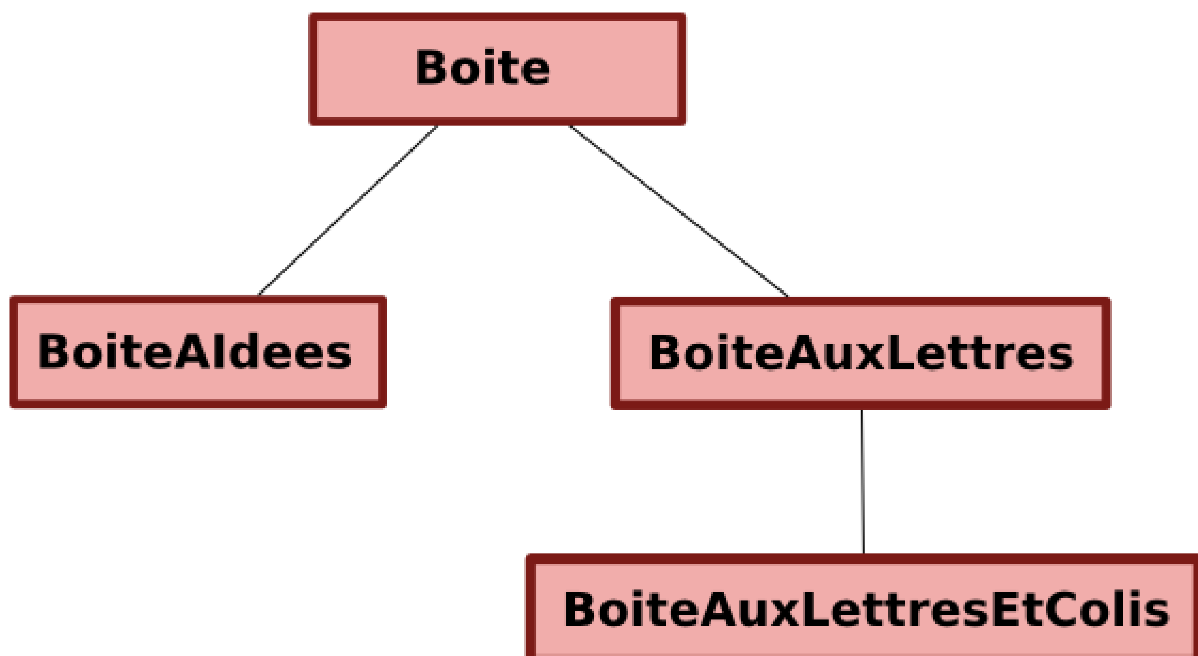
```
{
  nombre_boites++;
  // ...
}

int
main()
{
  BoiteAuxLettres b1, b2, b3;
  cout << "Nombre de boites: "
       << BoiteAuxLettres::nombre_boites << endl;
  // ...
}
```

Les variables statiques sont déclarées comme telles en préfixant leur déclaration du mot-clé `static`. De plus on doit lui affecter un espace mémoire ce qu'on fait dans l'implémentation dans un fichier `.cc` (ou `.cpp`) dans lequel on peut initialiser la variable. Dans l'exemple précédent, c'est parce qu'on a déclaré `nombre_boites` comme `public` que l'on peut y accéder depuis l'extérieur avec `BoiteAuxLettres::nombre_boites`.

## 4 L'héritage

### 4.1 Classes parentes

L'héritage permet de créer des types génériques ou spécialisés formant une hiérarchie. Dans notre cas on peut vouloir rajouter une classe `BoiteAuxLettresEtColis` qui soit exactement comme `BoiteAuxLettres` tout en offrant la possibilité de poster de colis. De même on pourrait vouloir créer une `BoiteAIdee` qui permet de poster des idées sous forme de lettre. Cette dernière classe est donc assez semblable à `BoiteAuxLettres` et les deux peuvent être des spécialisation d'une classe `Boite`.




Ce que l'on écrira en C de la façon suivante:

```
class Boite
{
  // ...
};
```

```
class BoiteAIdees : public Boite
{
  // ...
};

class BoiteAuxLettres : public Boite
{
  // ...
};

class BoiteAuxLettresEtColis : public BoiteAuxLettres
{
  // ...
};
```

On indique donc qu'une class *hérite* d'une autre classe en rajoutant `:public ClasseParente` après son nom. Le mot-clé `public` étant là pour indiquer que les champs `public` de `ClasseParente` le seront aussi dans la classe fille.

Pour que les champs privée d'une classe parente soit accessible dans une classe fille, remplacez l'utilisation de `private:` par `protected:`. Ainsi le champ `m_nombre_lettres` sera accessible dans `BoiteAuxLettresEtColis` mais pas à l'extérieur des classes de la hiérarche.

# 5 Les patrons

## 5.1 Patrons de fonctions

Supposiez que vous vouliez écrire une fonction pour faire la moyenne de plusieurs nombres. Vous pourriez procéder ainsi :

```
int
moyenne(int a, int b)
{
  return (a + b) / 2;
}
```

On se contente ici d'utiliser deux nombre pour en donner la moyenne. Cette section s'applique néanmoins à une fonction `moyenne()` définie de la façon suivante :

```
int
moyenne(int nombre, int* valeurs)
{
  int sum = 0;

  for (int i = 0; i < nombre; i++)
    sum += valeurs[i];
  return sum / nombre;
}
```

Le problème est que cela ne marcherait pas avec des nombres flottants. Vous pouvez espérer vous en sortir en créant une fonction `double moyenne(double, double)` ainsi qu'en opérant à des conversions pour retrouver des nombres entiers si besoin est :

```
double
moyenne(double a, double b)
{
  return (a + b) / 2.;
}

int
main()
```

```
{
  int a = 3, b = 8;
  double c = 1.5, d = 9.0;

  int m1 = (int) moyenne(a, b);
  double m2 = moyenne(c, d);

  // ...
}
```

Une autre stratégie serait d'utiliser l'héritage auquel cas vous pourrez créer une classe *Nombre* qui redéfinirait les opérations nécessaires pour avoir par exemple:

```
Nombre*
moyenne(Nombre* a, Nombre *b)
{
  return *(*a + *b) / 2;
}
```

À condition de gérer les blocs de mémoire alloué et de redéfinir les operateurs `operator+()` et `operator/(int)` pour chaque combinaison de nombre, ce qui peut vite devenir un travail de titan. Une telle solution ne marcherait de plus que pour des instances de la classe `Nombre` et de ses classes enfants.

Mais cela ne fonctionnerait évidemment alors pas avec des nombres complexes. Le C++ vous offre un méchanisme beaucoup plus puissant pour exprimer ce type de généricité, les *templates*.

```
template <typename T>
T
moyenne(T a, T b)
{
  return (a + b) / 2;
}

int
main()
{
  int a = 3, b = 8;
  double c = 1.5, d = 9.0;

  int m1 = moyenne(a, b);
  double m2 = moyenne(c, d);

  // ...
}
```

Il est ainsi directement possible d'utiliser `moyenne()` pour n'importe quel type d'objet pour lequel l'opération \+ est définie. Pour créer vos propre templates, définissez vos fonctions comme telle en rajoutant `template <typename T>` où `T` est un nom que vous choisissez et que vous utiliserez pour référencer le type générique dans la déclaration et le corps de la fonction. C'est le **compilateur** qui est chargé de créer les fonctions `moyenne(int, int)` ainsi que `moyenne(double, double)` sur le modèle donné par le patron.

**Important**
Bien qu'il existe des compilateurs de template, les templates sont généralement *instancié* pour les types particuliers effectivement utilisés à la compilation et doivent donc être placé intégralement dans les en-tête (déclarationis et corps de fonction).

Vous pouvez aussi utiliser plusieurs types dans un template.

```
template <typename T, template U>
U
moyenne(T a, T b)
{
  return ((U) a + (U) b) / 2;
}

int
main()
{
  int a = 2, b = 3;
  double c = moyenne<int, double>(a, b);
  // ...
}
```

Ce dernier exemple fait en sorte que la moyenne de deux entiers puisse être un flottant (`2.5` dans notre cas). On voit aussi que dans ce cas les types correspondant à `T` et `U` ont du être explicitement spécifiés à l'aide de `<int, double>`. L'ambiguïté étant qu'on aurait pu vouloir avoir `moyenne<int, int>(a, b)` soit un type de retour `int` qui serait convertit en `double` pour être mis dans `c`.

### 5.2 Patrons de classes

De même qu'il existe des patrons de fonctions on peut aussi créer des patrons de classes. Ainsi la `BoiteAuLettre` du chapitre précédent aurait pu être remplacée par un template de classe générique, `template <typename T> class Boite`, que l'on aurait instancié avec `Boite<Lettre>` ou `Boite<Idee>`.

À titre d'exemple nous allons définir une très simple classe `Tableau` qui représentera des tableaux de taille fixe d'objets de type générique.

```
template <typename T>
class Tableau
{
public:
  Tableau(int taille)
  {
    m_taille = taille;
    m_elements = new T[taille];
  }

  ~Tableau()
  {
    delete [] m_elements;
  }

  int taille() { return m_taille; }

  T& elem(int pos) { return m_elements[pos]; }

private:
  int m_taille;
  T* m_elements;
};

int
main()
{
  Tableau<int> i(3);
  Tableau<double> d(2);

  i.elem(1) = 2;
```

```
  d.elem(0) = 1.14159 + i.elem(1);

  // ...
}
```

La bibliothèque STL, *Standard template library*, fournit avec toute plateforme de développement C `contient différentes classes patrons dont` std::vector<T>++ qui représente un tableau de taille dynamique.

```
#include <iostream>
#include <vector>

using namespace std;

int
main()
{
  vector<int> v;
  v.push_back(123);
  v.push_back(38);

  int sum = v[0] + v[1];
  cout << v[0] << " + " << v[1] << " = " << sum << endl;

  v.erase(); // Efface tout le tableau
  cout << "v contient " << v.size() << " éléments" << endl; // 0

  return 0;
}
```

# 6 La bibliothèque standard

## 6.1 Introduction

Jusqu'à présent nous n'avons qu'effleuré l'usage de bibliothèques, que ce soit la `libc`, bibliothèque standard du C qui définit entre autres les fonctions de `stdio.h`, ou bien la STL, bibliothèque standard du C+\+ avec les fonctions de flux de `iostream` ou `std::vector`.

Les compilateurs C+\+ viennent ainsi avec la STL, pour *Standard template library*, qui définit en particulier :

- des *flux* de gestion des entrées sorties, que ce soit sur les fichiers ou bien sur les entrées-sorties console ;
- des *conteneurs génériques* qui sont des patrons pouvant donc contenir n'importe quel type sous forme de tableaux extensibles, de listes, de paires clé/valeur, ... ;
- des *itérateurs* pour parcourir les éléments présents dans les conteneurs ;
- des *algorithmes* à utiliser pour chercher, ordonner ou modifier les conteneurs.

Les fonctions et objets de la STL sont définis dans des en-têtes sans extension, on utilise ainsi `#include <iostream>` et non `#include <iostream.h>`. Tout est placé dans l'espace des noms `std`.

## 6.2 Les flux d'entrée/sortie

Nous avons déjà vu l'utilisation basique des flots d'entrée/sortie en C+\+ avec `std::cout` et `std::cin`. En plus de ces deux flots prédéfinis et déjà ouverts lors de l'exécution, le flux `std::cerr` permet d'écrire sur la sortie d'erreur, dédiée comme son nom l'indique aux messages d'erreur. Les opérateurs >> et << permettent respectivement de récupérer des données sur un flux d'entrée ou d'en envoyer sur un flux de sortie.