Programmation Impérative II
Info 121

Florent Hivert

Mél : Florent.Hivert@lri.fr
Adresse universelle : http://www.lri.fr/~hivert

Informations pratiques

- 10 heures de cours ;
- 18 heures de travaux dirigés ;
- 20 heures de travaux pratiques ;
- Évaluation (par ordre décroissant de coefficients) :

  examen final + 1 contrôle +
  projets + notes TP + PL (contrôle en ligne).

Rappels

Notions supposées connues :

- Notion de programme
- Notion de variable, de type, déclaration
- Instruction conditionnelle
  - `if (...) {...} else {...}`
- Instruction itérative :
  - `while (...) {...}`
  - `do {...} while (...);`
  - `for (...; ...; ...) {...}`
- Notion de fonction, appel de fonction.

Généralités sur le traitement de l'information

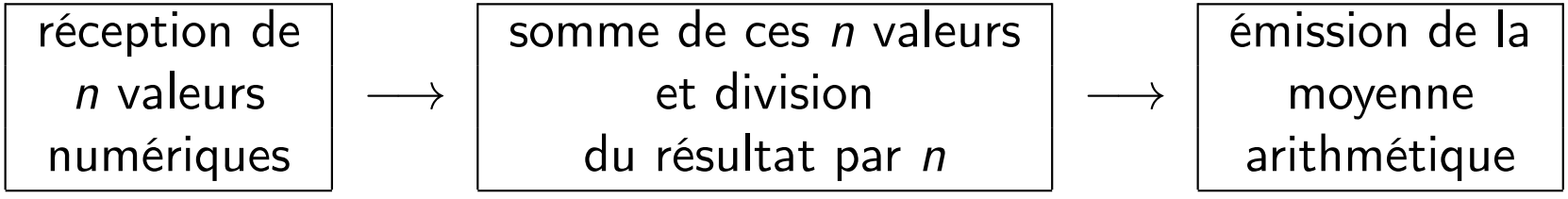
Les ordinateurs sont utilisés pour

- le traitement d'informations ;
- le stockage d'informations.

Généralités sur le traitement de l'information (2)

Le schéma global d'une application informatique est toujours le même :

Exemple :



La notion de programme

Tout traitement demandé à la machine, par l'utilisateur, est effectué par l'exécution séquencée d'opérations appelées **instructions**. Une suite d'instructions est appelée un **programme**.

Retenir

*Un programme est une **suite d'instructions** permettant à une système informatique d'exécuter une tâche donnée*

écrit dans un langage de programmation compréhensible (directement ou indirectement) par un ordinateur.

Programme Impératif

Définition (Rappel)

*un **programme impératif** est une séquence d'**instructions** qui spécifie étape par étape les opérations à effectuer pour obtenir à partir des entrées un résultat (la sortie).*

Paradigmes de programmation : Impératif/Procédural, Objet, Fonctionnel, déclaratif, concurrent.

Exemples de langages de programmations :
`C C++ Java Python Perl Ruby PHP Javascript Rust Go Scala Ocaml Haskell Lisp Erlang Ada Assembleur Fortran Pascal Delphi Julia Prolog`

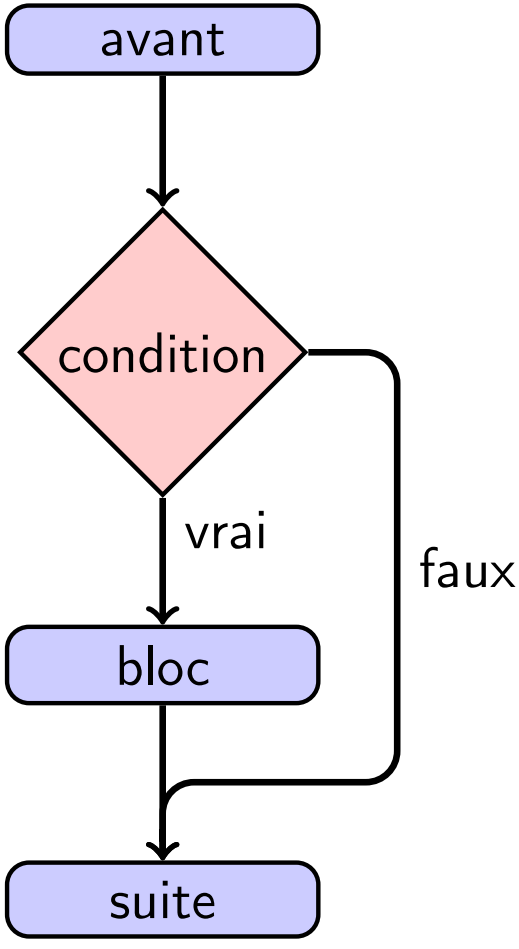
Rappel : instruction conditionnelle

```
instructions avant;

if (condition) {
    bloc d'instructions;
}

instructions suite;
```

Remarque : s'il n'y a qu'une instruction dans un bloc les accolades ne sont pas obligatoires. On choisira ce qui est le plus **lisible**.

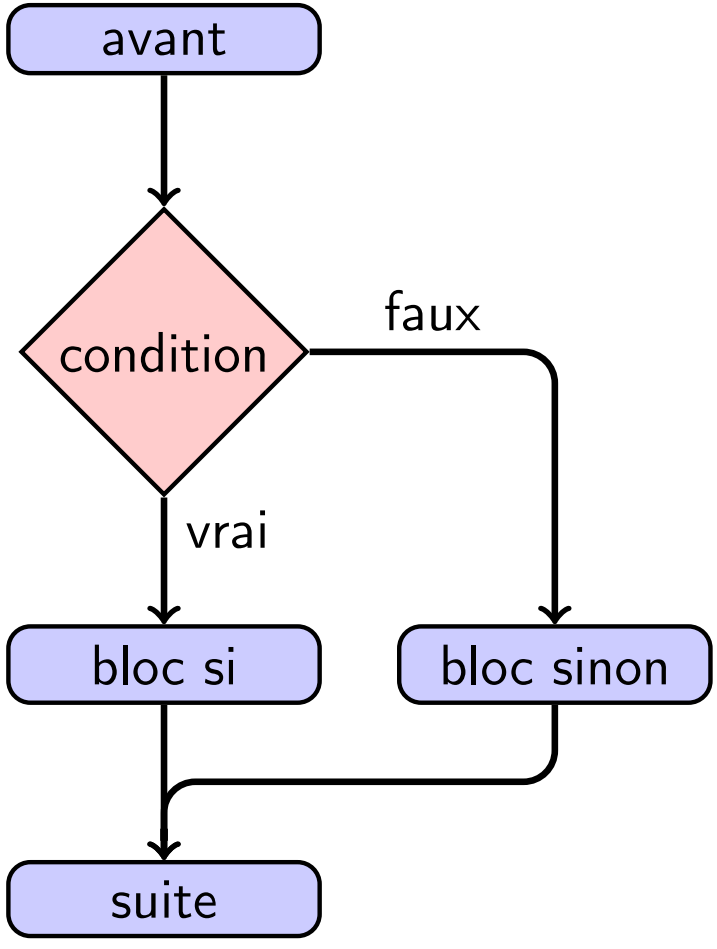


Rappel : instruction conditionnelle alternative

```
avant;

if (condition) {
    bloc si;
} else {
    bloc sinon;
}

suite;
```

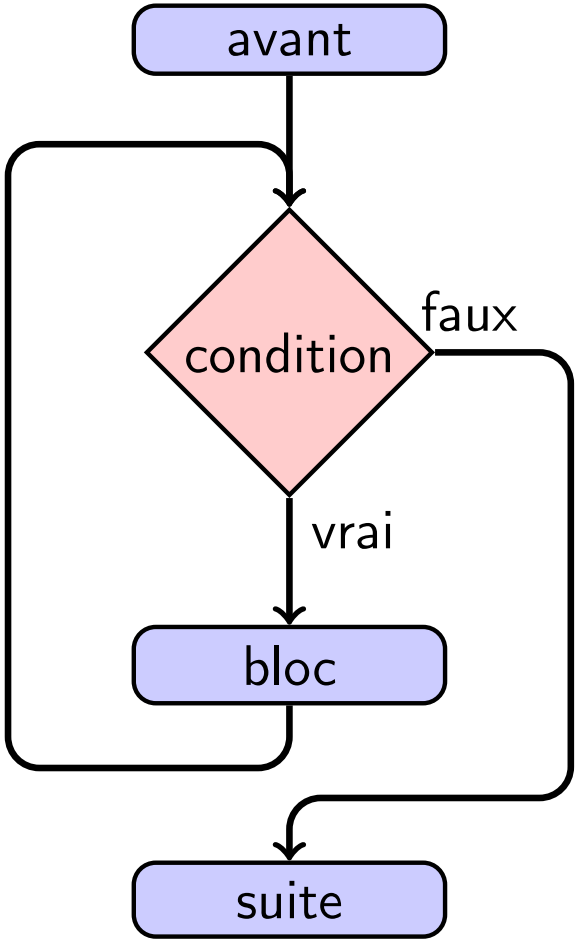


Rappel : instruction itérative (boucle `while`)

```
instructions avant;

while (condition) {
    bloc d'instructions;
}

instructions suite;
```



Remarque : Si la condition est fausse dès le début, le bloc n'est **jamais** exécuté.

Exemple de boucle while

while.cpp

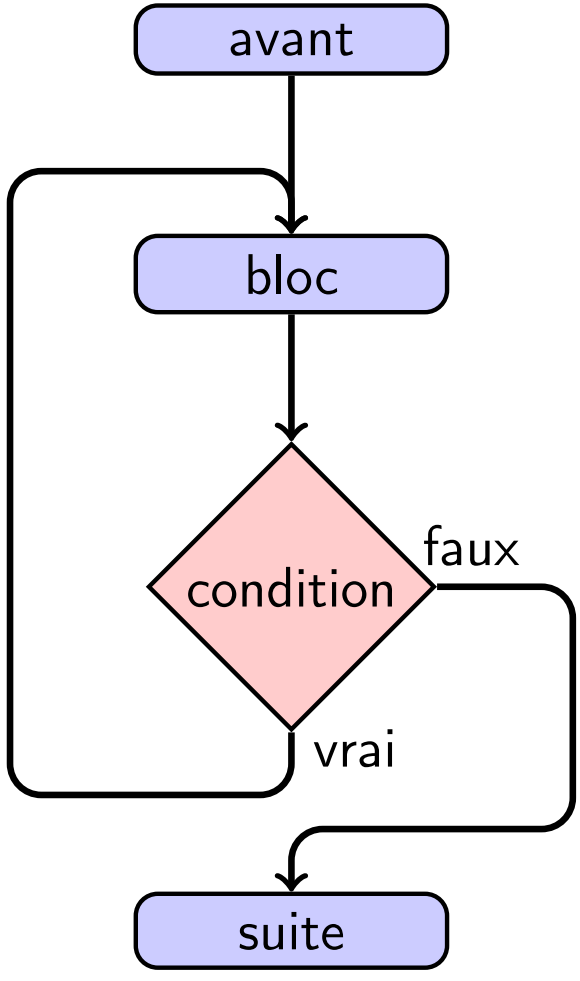
```
1  #include<iostream>
2  using namespace std;
3
4  int main() {
5    int n;
6
7    cout << "Saisir un nombre positif :  ";
8    cin >> n;
9    while (n < 0) {
10       cout << "Ce nombre est négatif !" << endl;
11       cout << "Saisir un nombre positif :  ";
12       cin >> n;
13   }
14   cout << "le nombre saisi est " << n << endl;
15   return EXIT_SUCCESS;
16 }
```

Complément : instruction itérative (boucle do while)

```
instructions avant;

do  {
    bloc d'instructions;
} while (condition);

instructions suite;
```



Remarque : le bloc est **toujours** exécuté au moins une fois.

Exemple de boucle do while

dowhile.cpp

```
1  #include<iostream>
2  using namespace std;
3
4  int main() {
5    int n;
6
7    do {
8      cout << "Saisir un nombre positif :  ";
9      cin >> n;
10     if (n < 0) {
11         cout << "Ce nombre est négatif !" << endl;
12     }
13   } while (n < 0);
14   cout << "le nombre saisi est " << n << endl;
15   return EXIT_SUCCESS;
16 }
```

Rappel : boucle `for`

Syntaxe

```
for (initialisation; condition; incrémentation) {
     instructions;
}
```

C'est un **raccourcis plus lisible** de la boucle `while` :

```
initialisation;
while (condition) {
    instructions;
    incrémentation;
}
```

Rappel : fonction

Syntaxe

Déclaration (mode d'emploi) :

```
type_retour nom(type1 <param1>, type2 <param2>, ...);
```

Définition :

```
type_retour nom(type1 param1, type2 param2, ...) {
  déclaration des variables locales;
  intructions;
  ...
  return valeur_de_retour;
}
```

Appel (utilisation dans une expression) :

```
   ... nom(valeur_param1, valeur_param2, ...) ...
```

Exemple : fonction

Syntaxe

```
int somme(int, int);

int somme(int a, int b) {
  int res;
  res = a + b;
  return res;
}
```

```
int main() {
  int x, y, s;
  cin >> x >> y;
  s = somme (x, y);
  cout << s << endl;
  return EXIT_SUCCESS;
}
```

Intruction `return`

Retenir

Deux rôles :

- ***quitter** la fonction en cours ;*
- ***renvoyer** une valeur au programme appelant.*

- Dans une procédure (valeur de retour `void`), seul le premier rôle est utilisé.
- Il y a toujours une instruction `return` implicite à la fin d'une procédure.

Intruction `return`

Retenir

Deux rôles :

- ***quitter** la fonction en cours ;*
- ***renvoyer** une valeur au programme appelant.*

- Dans une procédure (valeur de retour `void`), seul le premier rôle est utilisé.
- Il y a toujours une instruction `return` implicite à la fin d'une procédure.

Données et instructions

- Un programme est composé d'**instructions** qui travaillent sur des **données**.
- En programmation impérative, les données sont stockées dans des **variables**.
- Dans de nombreux langages de programmation (C/C++/Java...), il faut **déclarer** les variables.

```
#include<iostream>
void main(void) {
  double a,b;                  // Déclarations
  a=1; b=1;
  while ((((a+1)-a)-1)==0) a*=2;
  while ((((a+b)-a)-b)!=0) b++;
  std::cout << "a=" << a << ", b=", b << std::endl;
}
```

Rappel : Organisation générale de la machine

Retenir

Une machine est composée essentiellement de 4 éléments :

- *de la* ***mémoire*** *pour stocker les données*
- *des unités de* ***calcul*** *(logique, arithmétique, . . .)*
- *une unité de* ***commande*** *qui organise le travail*
- *des unités de* ***communications*** *(clavier, écran, réseau, . . .)*

Complément : La mémoire

La mémoire principale d'un ordinateur est composée de

- un très grand nombre (10^{10}–10^{11}) de **condensateurs** pouvant être soit chargés soit déchargés (0 ou 1)
- un gigantesque **réseau de fils et d'aiguillages** (multiplexeur)

Retenir

*On groupe les condensateurs par groupes (typiquement 8, 16, 32 ou 64) appelés **mots**. On repère un mot en mémoire grâce à un nombre appelé **adresse**.*

L'opérateur `&`, retourne l'adresse d'une variable :

addr.cpp

```
1    int a = 5;
2    cout << a << " " << &a << endl;
```

Affiche quelque chose comme `5 0x7ffe94f3a8bc`

La notion de variable

- but : stocker des informations en mémoire centrale durant l'exécution d'un programme
- on veut éviter d'avoir à manipuler directement les adresses : on manipule des **variables**
- le programmeur donne aux variables des noms de son choix (**identificateur**)

Définition (**Notion de Variable**)

*Une **variable** est un espace de stockage où le programme peut mémoriser une donnée.*

Les variables désignent une ou plusieurs cases mémoires contenant une suite de 0 et de 1.

La notion de variable (2)

Retenir

Une variable en cours d'utilisation possède quatre propriétés :

- *un nom ;*
- *une adresse ;*
- *un type ;*
- *une valeur.*

But du cours :

- Avoir une idée générale de comment tout ça marche...
- Savoir organiser son code et ses données.

Objectifs du cours :

- Apprendre le **vocabulaire** précis de la programmation
- Décrire en détail ce que fait un programme (**sémantique**)
- Comprendre comment la **mémoire est gérée** par le programme
- Apprendre à **organiser ses données** (tableaux, structures)
- Apprendre à **organiser ses programmes** (modularité, tests, encaspulation).

Plan du cours

1. Sémantique d'un programme C++

   variables, fonctions, passages de paramêtres

2. Structures de données

   énumération, structures et tableaux

3. Initiation au génie logiciel

   Éléments de programmation modulaire, méthodologie, décomposition du travail, encapsulation, tests