Cours d'Algorithmique et de Langage C 2005 - v 3.0

Bob CORDEAU
cordeau@onera.fr

Mesures Physiques
IUT d'Orsay

15 mai 2006

Avant-propos

Ce cours en libre accès repose sur trois partis pris :

- le choix d'un langage algorithmique minimal, francisé ;
- le choix du langage C, mais en utilisant le compilateur C++, pour bénéficier d'améliorations reconnues en génie logiciel ;
- enfin le choix d'un logiciel libre : Dev-C++ (environnement de développement intégré téléchargeable sur le site `http ://www.bloodshed.net`).

Cette deuxième version est destinée à évoluer en fonction des remarques émises.

Avant-propos

Ce cours en libre accès repose sur trois partis pris :

- le choix d'un langage algorithmique minimal, francisé ;
- le choix du langage C, mais en utilisant le compilateur C++, pour bénéficier d'améliorations reconnues en génie logiciel ;
- enfin le choix d'un logiciel libre : Dev-C++ (environnement de développement intégré téléchargeable sur le site `http ://www.bloodshed.net`).

Cette deuxième version est destinée à évoluer en fonction des remarques émises.

Avant-propos

Ce cours en libre accès repose sur trois partis pris :

- le choix d'un langage algorithmique minimal, francisé ;
- le choix du langage C, mais en utilisant le compilateur C++, pour bénéficier d'améliorations reconnues en génie logiciel ;
- enfin le choix d'un logiciel libre : Dev-C++ (environnement de développement intégré téléchargeable sur le site `http ://www.bloodshed.net`).

Cette deuxième version est destinée à évoluer en fonction des remarques émises.

Remerciements

Les corrections des versions précédentes doivent beaucoup à Nicolas Férey, Georges Vincents, Fanny Jambon et Michelle Cordeau, à qui je sais gré de leurs conseils ou lectures attentives.

Sommaire de la Partie 1

1. Introduction à l'informatique
   - Environnement matériel
   - Environnement logiciel
   - Langages
   - Production des programmes
   - Historique
   - Méthodologie

Sommaire de la Partie 2

2 Les bases de la programmation
- Algorithme
- C
- Programme
- Importance des notations
- Types de base
- Variable et affectation
- Entrées/Sorties
- Instruction
- Condition
- Trace d'un algorithme

Sommaire de la Partie 3

3 Les instructions en algorithmique et en C
- Opérateurs arithmétiques
- Opérateurs relationnels
- Opérateurs booléens
- Opérateurs binaires
- Priorités des opérateurs

Sommaire de la Partie 4

4. Les structures en algorithmique et en C
- Sélection
- Boucle `Faire .. TantQue`
- Boucle `TantQue .. FinTantQue`
- Boucle `Pour .. FinPour`
- Ruptures de boucle

Sommaire de la Partie 5

5 Les fonctions
- Caractéristiques
- Notation
- Le passage des arguments
- Récursivité

Sommaire de la Partie 6

6 Les pointeurs, tableaux et procédures
- Procédures
- Pointeurs
- Passage des arguments par adresse
- Relation avec les tableaux

Sommaire de la Partie 7

7. Quelques aspects avancés du langage C
- Entrées/Sorties sur fichiers
- Chaînes de caractères
- Structures
- Allocation programmée
- Arguments du `main()`

Première partie I

Introduction générale

L'ordinateur

Définition

Automate déterministe à composants électroniques.

L'ordinateur comprend entre autres :

- un microprocesseur avec une UC (Unité de Contrôle), une UAL (Unité Arithmétique et Logique), une horloge, une mémoire cache rapide ;
- de la mémoire vive (RAM), contenant les instructions et les données. La RAM est formée de cellules binaires organisées en mot de 8 bits (octet) ;
- des périphériques : entrées/sorties, mémoires mortes (disque dur, CD-ROM. . .), réseau. . .

L'ordinateur

Définition

Automate déterministe à composants électroniques.

L'ordinateur comprend entre autres :

- un microprocesseur avec une UC (Unité de Contrôle), une UAL (Unité Arithmétique et Logique), une horloge, une mémoire cache rapide ;
- de la mémoire vive (RAM), contenant les instructions et les données. La RAM est formée de cellules binaires organisées en mot de 8 bits (octet) ;
- des périphériques : entrées/sorties, mémoires mortes (disque dur, CD-ROM. . .), réseau. . .

L'ordinateur

Définition

Automate déterministe à composants électroniques.

L'ordinateur comprend entre autres :

- un microprocesseur avec une UC (Unité de Contrôle), une UAL (Unité Arithmétique et Logique), une horloge, une mémoire cache rapide ;
- de la mémoire vive (RAM), contenant les instructions et les données. La RAM est formée de cellules binaires organisées en mot de 8 bits (octet) ;
- des périphériques : entrées/sorties, mémoires mortes (disque dur, CD-ROM...), réseau...

L'ordinateur

Définition

Automate déterministe à composants électroniques.

L'ordinateur comprend entre autres :

- un microprocesseur avec une UC (Unité de Contrôle), une UAL (Unité Arithmétique et Logique), une horloge, une mémoire cache rapide ;
- de la mémoire vive (RAM), contenant les instructions et les données. La RAM est formée de cellules binaires organisées en mot de 8 bits (octet) ;
- des périphériques : entrées/sorties, mémoires mortes (disque dur, CD-ROM. . .), réseau. . .

Deux sortes de programmes

- Le système d'exploitation : ensemble des programmes qui gèrent les ressources matérielles et logicielles ; il propose une aide au dialogue entre l'utilisateur et l'ordinateur : l'interface textuelle (interprète de commande) ou graphique (gestionnaire de fenêtres). Il est souvent multitâche et parfois multiutilisateur ;
- les programmes applicatifs dédiés à des tâches particulières. Ils sont formés d'une série de commandes contenues dans un programme *source* écrit dans un langage « compris » par l'ordinateur.

Deux sortes de programmes

- Le système d'exploitation : ensemble des programmes qui gèrent les ressources matérielles et logicielles ; il propose une aide au dialogue entre l'utilisateur et l'ordinateur : l'interface textuelle (interprète de commande) ou graphique (gestionnaire de fenêtres). Il est souvent multitâche et parfois multiutilisateur ;
- les programmes applicatifs dédiés à des tâches particulières. Ils sont formés d'une série de commandes contenues dans un programme *source* écrit dans un langage « compris » par l'ordinateur.

Des langages de différents niveaux

- Chaque processeur possède un langage propre, directement exécutable : le langage machine. Il est formé de 0 et de 1 et n'est pas portable, mais c'est le seul que l'ordinateur « comprend » ;
- le langage d'assemblage est un codage alphanumérique du langage machine. Il est plus lisible que la langage machine, mais n'est toujours pas portable. On le traduit en langage machine par un *assembleur* ;
- les langages de haut niveau. Souvent normalisés, ils permettent le portage d'une machine à l'autre. Ils sont traduits en langage machine par un *compilateur* ou un *interpréteur*.

Des langages de différents niveaux

- Chaque processeur possède un langage propre, directement exécutable : le langage machine. Il est formé de 0 et de 1 et n'est pas portable, mais c'est le seul que l'ordinateur « comprend » ;
- le langage d'assemblage est un codage alphanumérique du langage machine. Il est plus lisible que la langage machine, mais n'est toujours pas portable. On le traduit en langage machine par un *assembleur* ;
- les langages de haut niveau. Souvent normalisés, ils permettent le portage d'une machine à l'autre. Ils sont traduits en langage machine par un *compilateur* ou un *interpréteur*.

Des langages de différents niveaux

- Chaque processeur possède un langage propre, directement exécutable : le langage machine. Il est formé de 0 et de 1 et n'est pas portable, mais c'est le seul que l'ordinateur « comprend » ;
- le langage d'assemblage est un codage alphanumérique du langage machine. Il est plus lisible que la langage machine, mais n'est toujours pas portable. On le traduit en langage machine par un *assembleur* ;
- les langages de haut niveau. Souvent normalisés, ils permettent le portage d'une machine à l'autre. Ils sont traduits en langage machine par un *compilateur* ou un *interpréteur*.

Deux techniques de production des programmes

- La compilation est la traduction du source en langage objet. Elle comprend au moins quatre phases (trois phases d'analyse — lexicale, syntaxique et sémantique — et une de production de code objet). Pour générer le langage machine il faut encore une phase particulière : l'édition de liens. Cette technique est contraignante mais offre une grande vitesse d'exécution ;
- dans la technique de l'interprétation chaque ligne du source analysé est traduite au fur et à mesure en instructions directement exécutées. Aucun programme objet n'est généré. C'est très souple mais l'interpréteur doit être utilisé à chaque exécution...
- Ces deux techniques peuvent coexister pour un même langage.

Deux techniques de production des programmes

- La compilation est la traduction du source en langage objet. Elle comprend au moins quatre phases (trois phases d'analyse — lexicale, syntaxique et sémantique — et une de production de code objet). Pour générer le langage machine il faut encore une phase particulière : l'édition de liens. Cette technique est contraignante mais offre une grande vitesse d'exécution ;
- dans la technique de l'interprétation chaque ligne du source analysé est traduite au fur et à mesure en instructions directement exécutées. Aucun programme objet n'est généré. C'est très souple mais l'interpréteur doit être utilisé à chaque exécution...
- Ces deux techniques peuvent coexister pour un même langage.

Deux techniques de production des programmes

- La compilation est la traduction du source en langage objet. Elle comprend au moins quatre phases (trois phases d'analyse — lexicale, syntaxique et sémantique — et une de production de code objet). Pour générer le langage machine il faut encore une phase particulière : l'édition de liens. Cette technique est contraignante mais offre une grande vitesse d'exécution ;
- dans la technique de l'interprétation chaque ligne du source analysé est traduite au fur et à mesure en instructions directement exécutées. Aucun programme objet n'est généré. C'est très souple mais l'interpréteur doit être utilisé à chaque exécution...
- Ces deux techniques peuvent coexister pour un même langage.

Bref historique des langages

- Années 50 (approches expérimentales) : FORTRAN, LISP, COBOL...
- Années 60 (langages universels) : ALGOL, PL/1, PASCAL...
- Années 70 (génie logiciel) : C, MODULA-2, ADA...
- Années 80–90 (programmation objet) : C++, Labview, Eiffel, Matlab...
- Années 90–2000 (langages interprétés objet) : Java, Perl, tcl/Tk, Ruby, Python...

Parmi des centaines de langages créés, une minorité est vraiment utilisée.

Bref historique des langages

- Années 50 (approches expérimentales) : FORTRAN, LISP, COBOL...
- Années 60 (langages universels) : ALGOL, PL/1, PASCAL...
- Années 70 (génie logiciel) : C, MODULA-2, ADA...
- Années 80–90 (programmation objet) : C++, Labview, Eiffel, Matlab...
- Années 90–2000 (langages interprétés objet) : Java, Perl, tcl/Tk, Ruby, Python...

Parmi des centaines de langages créés, une minorité est vraiment utilisée.

Bref historique des langages

- Années 50 (approches expérimentales) : FORTRAN, LISP, COBOL. . .
- Années 60 (langages universels) : ALGOL, PL/1, PASCAL. . .
- Années 70 (génie logiciel) : C, MODULA-2, ADA. . .
- Années 80–90 (programmation objet) : C++, Labview, Eiffel, Matlab. . .
- Années 90–2000 (langages interprétés objet) : Java, Perl, tcl/Tk, Ruby, Python. . .

Parmi des centaines de langages créés, une minorité est vraiment utilisée.

Bref historique des langages

- Années 50 (approches expérimentales) : FORTRAN, LISP, COBOL...
- Années 60 (langages universels) : ALGOL, PL/1, PASCAL...
- Années 70 (génie logiciel) : C, MODULA-2, ADA...
- Années 80–90 (programmation objet) : C++, Labview, Eiffel, Matlab...
- Années 90–2000 (langages interprétés objet) : Java, Perl, tcl/Tk, Ruby, Python...

Parmi des centaines de langages créés, une minorité est vraiment utilisée.

Bref historique des langages

- Années 50 (approches expérimentales) : FORTRAN, LISP, COBOL...
- Années 60 (langages universels) : ALGOL, PL/1, PASCAL...
- Années 70 (génie logiciel) : C, MODULA-2, ADA...
- Années 80–90 (programmation objet) : C++, Labview, Eiffel, Matlab...
- Années 90–2000 (langages interprétés objet) : Java, Perl, tcl/Tk, Ruby, Python...

Parmi des centaines de langages créés, une minorité est vraiment utilisée.

La construction des programmes : le génie logiciel

Plusieurs modèles sont envisageables, par exemple :

- la méthodologie procédurale. On emploie l'analyse descendante et remontante qui procède par raffinements successifs : on s'efforce de décomposer un problème complexe en sous-programmes plus simples. Cette méthode structure d'abord les actions ;
- la méthodologie objet définit d'abord des composants (les *objets*) qui contiennent des *attributs* (données) et des *méthodes* (actions). On communique entre objets par l'envoi de *messages* qui donnent accès à un attribut ou qui lancent une méthode.

Tous les modèles reposent sur la notion d'algorithme.

La construction des programmes : le génie logiciel

Plusieurs modèles sont envisageables, par exemple :

- la méthodologie procédurale. On emploie l'analyse descendante et remontante qui procède par raffinements successifs : on s'efforce de décomposer un problème complexe en sous-programmes plus simples. Cette méthode structure d'abord les actions ;
- la méthodologie objet définit d'abord des composants (les *objets*) qui contiennent des *attributs* (données) et des *méthodes* (actions). On communique entre objets par l'envoi de *messages* qui donnent accès à un attribut ou qui lancent une méthode.

Tous les modèles reposent sur la notion d'algorithme.

Exemple d'algorithme : volume d'un cône droit

DébutProgramme
  DébutDéclaration
    $pi \leftarrow 3.141593$: flottant
    $rayon, hauteur, volume$: flottant
  FinDéclaration

  Afficher("Rayon du cône : ")
  Saisir($rayon$**)**
  Afficher("Hauteur du cône : ")
  Saisir($hauteur$**)**

  $volume \leftarrow (pi * rayon * rayon * hauteur)/3$
  Afficher("Volume du cône = ", $volume$**)**
FinProgramme

Deuxième partie II

Les bases de la programmation

Qu'est-ce qu'un algorithme ?

Définition

Moyen d'atteindre un but en répétant un nombre fini de fois un nombre fini d'instructions.

Donc, un algorithme se termine en un temps fini.

Qu'est-ce qu'un algorithme ?

Définition

Moyen d'atteindre un but en répétant un nombre fini de fois un nombre fini d'instructions.

Donc, un algorithme se termine en un temps fini.

Exemple : calcul manuel de la racine carrée de 37815

On sépare le nombre en groupes de deux chiffres en partant de la droite : $\overset{\frown}{03}\ \overset{\frown}{78}\ \overset{\frown}{15}$.
Pour le premier groupe $\overset{\frown}{03}$:

- on cherche sa plus grande racine carrée : 1
- on retire $1^2 = 1$ du groupe. Reste : 2
- on abaisse le groupe suivant : $2\overset{\frown}{78}$

Pour tous les autres groupes :

1. on double la racine courante : $1 \times 2 = 2$
2. on cherche n tel que $2n \times n \leqslant 278$. Soit $n = 9$
3. on retire $29 \times 9 = 261$ du reste. Soit 17
4. on abaisse le groupe suivant : $17\overset{\frown}{15}$

Puis on itère les étapes 1 à 4 jusqu'à la précision désirée.

Le langage algorithmique

L'exemple précédent illustre la difficulté de décrire complètement et sans ambiguïté un processus non trivial. D'où le besoin de *formaliser* un langage de description des algorithmes.

Le langage algorithmique choisi doit être simple, clair et précis.

Nous utiliserons un langage francisé formé d'une vingtaine de mots réservés.

Le langage algorithmique

L'exemple précédent illustre la difficulté de décrire complètement et sans ambiguïté un processus non trivial. D'où le besoin de *formaliser* un langage de description des algorithmes.

Le langage algorithmique choisi doit être simple, clair et précis.

Nous utiliserons un langage francisé formé d'une vingtaine de mots réservés.

Le langage algorithmique

L'exemple précédent illustre la difficulté de décrire complètement et sans ambiguïté un processus non trivial. D'où le besoin de *formaliser* un langage de description des algorithmes.

Le langage algorithmique choisi doit être simple, clair et précis.

Nous utiliserons un langage francisé formé d'une vingtaine de mots réservés.

Mots réservés du langage algorithmique

`Afficher`	`Dans`	`DébutDéclarations`
`DébutFonction`	`DébutProcédure`	`DébutProgramme`
`Faire`	`FinDéclarations`	`FinFonction`
`FinPour`	`FinProcédure`	`FinProgramme`
`FinSi`	`FinTantQue`	`ParPasDe`
`Pour`	`Retourner`	`Saisir`
`Si`	`Sinon`	`TantQue`

On y ajoutera des fonctions utiles, par exemple : `Alea(n)`, qui fournit un entier au hasard entre 0 et $n - 1$, plus toutes les fonctions mathématiques usuelles : `sin(x)`, `arccos(x)`, `tanh(x)`, `log(x)`...

Mots réservés du langage algorithmique

`Afficher`	`Dans`	`DébutDéclarations`
`DébutFonction`	`DébutProcédure`	`DébutProgramme`
`Faire`	`FinDéclarations`	`FinFonction`
`FinPour`	`FinProcédure`	`FinProgramme`
`FinSi`	`FinTantQue`	`ParPasDe`
`Pour`	`Retourner`	`Saisir`
`Si`	`Sinon`	`TantQue`

On y ajoutera des fonctions utiles, par exemple : `Alea(n)`, qui fournit un entier au hasard entre 0 et $n - 1$, plus toutes les fonctions mathématiques usuelles : `sin(x)`, `arccos(x)`, `tanh(x)`, `log(x)`...

Mots réservés du langage C (norme ANSI C99)

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

Programme et algorithme

Définition

Un programme est la traduction d'un algorithme en un langage compilable ou interprétable par un ordinateur.

Il est souvent écrit en plusieurs parties, dont une qui *pilote* les autres, le programme principal.

Programme et algorithme

Définition

Un programme est la traduction d'un algorithme en un langage compilable ou interprétable par un ordinateur.

Il est souvent écrit en plusieurs parties, dont une qui *pilote* les autres, le programme principal.

La présentation des programmes

Un programme est destiné à l'être humain. Pour en faciliter la lecture, il doit être correctement indenté et judicieusement commenté (cf. annexe « programme type »).

On nomme indentation les retraits en débuts de ligne permettant d'extraire rapidement la structure (le *squelette*) d'un programme, la logique d'une boucle ou d'un test.

La signification de parties non triviales doit être signalée par un commentaire :

- en algorithmique, il est souligné ou commence par `//` *ceci est un commentaire* `// en voici un autre`
- le C possède deux notations :
  - commentaire ligne `// ceci est un commentaire`
  - commentaire bloc, il commence par `/*` et se termine par `*/`

La présentation des programmes

Un programme est destiné à l'être humain. Pour en faciliter la lecture, il doit être correctement indenté et judicieusement commenté (cf. annexe « programme type »).

On nomme indentation les retraits en débuts de ligne permettant d'extraire rapidement la structure (le *squelette*) d'un programme, la logique d'une boucle ou d'un test.

La signification de parties non triviales doit être signalée par un commentaire :

- en algorithmique, il est souligné ou commence par // : *ceci est un commentaire* `// en voici un autre`
- le C possède deux notations :
  - commentaire ligne : `// ceci est un commentaire`
  - commentaire bloc, il commence par `/*` et se termine par `*/`

La présentation des programmes

Un programme est destiné à l'être humain. Pour en faciliter la lecture, il doit être correctement indenté et judicieusement commenté (cf. annexe « programme type »).

On nomme indentation les retraits en débuts de ligne permettant d'extraire rapidement la structure (le *squelette*) d'un programme, la logique d'une boucle ou d'un test.

La signification de parties non triviales doit être signalée par un commentaire :

- en algorithmique, il est souligné ou commence par `//` :
  ceci est un commentaire   `// en voici un autre`
- le C possède deux notations :
  - commentaire ligne : `// ceci est un commentaire`
  - commentaire bloc, il commence par `/*` et se termine par `*/`

La présentation des programmes

Un programme est destiné à l'être humain. Pour en faciliter la lecture, il doit être correctement indenté et judicieusement commenté (cf. annexe « programme type »).

On nomme indentation les retraits en débuts de ligne permettant d'extraire rapidement la structure (le *squelette*) d'un programme, la logique d'une boucle ou d'un test.

La signification de parties non triviales doit être signalée par un commentaire :

- en algorithmique, il est souligné ou commence par // :
  ceci est un commentaire `// en voici un autre`
- le C possède deux notations :
  - commentaire ligne : `// ceci est un commentaire`
  - commentaire bloc, il commence par /* et se termine par */

La présentation des programmes

Un programme est destiné à l'être humain. Pour en faciliter la lecture, il doit être correctement indenté et judicieusement commenté (cf. annexe « programme type »).

On nomme indentation les retraits en débuts de ligne permettant d'extraire rapidement la structure (le *squelette*) d'un programme, la logique d'une boucle ou d'un test.

La signification de parties non triviales doit être signalée par un commentaire :

- en algorithmique, il est souligné ou commence par // :
  ceci est un commentaire `// en voici un autre`
- le C possède deux notations :
  - commentaire ligne : `// ceci est un commentaire`
  - commentaire bloc, il commence par `/*` et se termine par `*/`

Le type booléen

- Ensemble de définition : $\mathcal{B} = \{FAUX, VRAI\}$
- Opérations booléennes :

Non booléen

a	$NON(a)$
$VRAI$	$FAUX$
$FAUX$	$VRAI$

OU et ET booléens

a	b	a OU b	a ET b
$FAUX$	$FAUX$	$FAUX$	$FAUX$
$FAUX$	$VRAI$	$VRAI$	$FAUX$
$VRAI$	$FAUX$	$VRAI$	$FAUX$
$VRAI$	$VRAI$	$VRAI$	$VRAI$

- Déclaration algorithmique : $a, b : booléen$
- Déclaration C : `bool a, b;`

Le type booléen

- Ensemble de définition : $\mathcal{B} = \{FAUX, VRAI\}$
- Opérations booléennes :

Non booléen

a	$NON(a)$
$VRAI$	$FAUX$
$FAUX$	$VRAI$

OU et ET booléens

a	b	$a\ OU\ b$	$a\ ET\ b$
$FAUX$	$FAUX$	$FAUX$	$FAUX$
$FAUX$	$VRAI$	$VRAI$	$FAUX$
$VRAI$	$FAUX$	$VRAI$	$FAUX$
$VRAI$	$VRAI$	$VRAI$	$VRAI$

- Déclaration algorithmique : a, b : *booléen*
- Déclaration C : `bool a, b;`

Le type booléen

- Ensemble de définition : $\mathcal{B} = \{FAUX, VRAI\}$
- Opérations booléennes :

Non booléen

a	$NON(a)$
$VRAI$	$FAUX$
$FAUX$	$VRAI$

OU et ET booléens

a	b	$a\ OU\ b$	$a\ ET\ b$
$FAUX$	$FAUX$	$FAUX$	$FAUX$
$FAUX$	$VRAI$	$VRAI$	$FAUX$
$VRAI$	$FAUX$	$VRAI$	$FAUX$
$VRAI$	$VRAI$	$VRAI$	$VRAI$

- Déclaration algorithmique : $a, b : booléen$
- Déclaration C : `bool a, b;`

Le type booléen

- Ensemble de définition : $\mathcal{B} = \{FAUX, VRAI\}$
- Opérations booléennes :

Non booléen

a	$NON(a)$
$VRAI$	$FAUX$
$FAUX$	$VRAI$

OU et ET booléens

a	b	a OU b	a ET b
$FAUX$	$FAUX$	$FAUX$	$FAUX$
$FAUX$	$VRAI$	$VRAI$	$FAUX$
$VRAI$	$FAUX$	$VRAI$	$FAUX$
$VRAI$	$VRAI$	$VRAI$	$VRAI$

- Déclaration algorithmique : a, b : *booléen*
- Déclaration C : `bool a, b;`

Le type entier

- Ensemble de définition : $\mathbb{Z}$
- Opérations sur les entiers :
  - $n_1 + n_2$ $n_1 - n_2$ $n_1 * n_2$
  - division entière : n_1 / n_2 (35 / 8 donne 4)
  - reste de la division entière : $n_1 \% n_2$ (35 % 8 donne 3)
- Notation algorithmique : $n1, n2$: *entier*
- Notation C : `int n1, n2;`

Le type entier

- Ensemble de définition : $\mathbb{Z}$
- Opérations sur les entiers :
  - $n_1 + n_2 \quad n_1 - n_2 \quad n_1 * n_2$
  - division entière : n_1 / n_2 (35 / 8 donne 4)
  - reste de la division entière : $n_1 \% n_2$ (35 % 8 donne 3)
- Notation algorithmique : $n1, n2$: *entier*
- Notation C : `int n1, n2;`

Le type entier

- Ensemble de définition : $\mathbb{Z}$
- Opérations sur les entiers :
  - $n_1 + n_2 \quad n_1 - n_2 \quad n_1 * n_2$
  - division entière : n_1 / n_2 (35 / 8 donne 4)
  - reste de la division entière : $n_1 \% n_2$ (35 % 8 donne 3)
- Notation algorithmique : $n1, n2$: *entier*
- Notation C : `int n1, n2 ;`

Le type entier

- Ensemble de définition : $\mathbb{Z}$
- Opérations sur les entiers :
  - $n_1 + n_2 \quad n_1 - n_2 \quad n_1 * n_2$
  - division entière : n_1 / n_2 (35 / 8 donne 4)
  - reste de la division entière : $n_1 \% n_2$ (35 % 8 donne 3)
- Notation algorithmique : $n1, n2$: *entier*
- Notation C : `int n1, n2;`

Le type flottant

- Ensemble de définition : $\mathbb{F}$
- Opérations sur les flottants :
  - $x_1 + x_2$ $x_1 - x_2$ $x_1 * x_2$ x_1 / x_2
  - toutes les opérations mathématiques usuelles : ln, cos, arctan ...
- Notation algorithmique : $x1, x2$: *flottant*
- Notation C : `float x1, x2;`

IUT ORSAY

Le type flottant

- Ensemble de définition : $\mathbb{F}$
- Opérations sur les flottants :
  - $x_1 + x_2 \quad x_1 - x_2 \quad x_1 * x_2 \quad x_1 / x_2$
  - toutes les opérations mathématiques usuelles : ln, cos, arctan ...
- Notation algorithmique : $x1, x2$: *flottant*
- Notation C : `float x1, x2 ;`

Le type flottant

- Ensemble de définition : $\mathbb{F}$
- Opérations sur les flottants :
  - $x_1 + x_2 \quad x_1 - x_2 \quad x_1 * x_2 \quad x_1/x_2$
  - toutes les opérations mathématiques usuelles : $\ln, \cos, \arctan \ldots$
- Notation algorithmique : $x1, x2$: *flottant*
- Notation C : `float x1, x2;`

Le type flottant

- Ensemble de définition : $\mathbb{F}$
- Opérations sur les flottants :
  - $x_1 + x_2 \quad x_1 - x_2 \quad x_1 * x_2 \quad x_1 / x_2$
  - toutes les opérations mathématiques usuelles : $\ln, \cos, \arctan \ldots$
- Notation algorithmique : $x1, x2$: *flottant*
- Notation C : `float x1, x2;`

Le type caractère

- Ensemble de définition : la table `ASCII`
- Opérations sur les caractères :
  - $SUCC(c)$: caractère suivant dans la table `ASCII` ;
  - $PRED(c)$: caractère précédent dans la table `ASCII`.
- Notation algorithmique : $c1, c2$: *caractère*
- Notation C : `char c1, c2 ;`

Le type caractère

- Ensemble de définition : la table `ASCII`
- Opérations sur les caractères :
  - $SUCC(c)$: caractère suivant dans la table `ASCII` ;
  - $PRED(c)$: caractère précédent dans la table `ASCII`.
- Notation algorithmique : $c1, c2$: *caractère*
- Notation C : `char c1, c2;`

Le type caractère

- Ensemble de définition : la table `ASCII`
- Opérations sur les caractères :
  - $SUCC(c)$: caractère suivant dans la table `ASCII` ;
  - $PRED(c)$: caractère précédent dans la table `ASCII`.
- Notation algorithmique : $c1, c2$: *caractère*
- Notation C : `char c1, c2;`

Le type caractère

- Ensemble de définition : la table `ASCII`
- Opérations sur les caractères :
  - $SUCC(c)$: caractère suivant dans la table `ASCII` ;
  - $PRED(c)$: caractère précédent dans la table `ASCII`.
- Notation algorithmique : $c1, c2$: *caractère*
- Notation C : `char c1, c2 ;`

iUT ORSAY

Taille et amplitude du type caractère

N°	char	N°	char	N°	char	N°	char
32		56	8	80	P	104	h
33	!	57	9	81	Q	105	i
34	"	58	:	82	R	106	j
35	#	59	;	83	S	107	k
36	$	60	<	84	T	108	l
37	%	61	=	85	U	109	m
38	&	62	>	86	V	110	n
39	'	63	?	87	W	111	o
40	(	64	@	88	X	112	p
41	)	65	A	89	Y	113	q
42	*	66	B	90	Z	114	r
43	+	67	C	91	[	115	s
44	,	68	D	92	\	116	t
45	-	69	E	93	]	117	u
46	.	70	F	94	^	118	v
47	/	71	G	95	_	119	w
48	0	72	H	96	`	120	x
49	1	73	I	97	a	121	y
50	2	74	J	98	b	122	z
51	3	75	K	99	c	123	{
52	4	76	L	100	d	124	¦
53	5	77	M	101	e	125	}
54	6	78	N	102	f	126	~
55	7	79	O	103	g	127	⌂

La table ASCII code tous les caractères usuels (états-uniens) sur 7 bits (128 positions). Sur un octet (8 bits), en C, on dispose :

- du type char défini sur $[-128..127]$;
- du type unsigned char défini sur $[0..255]$.

Taille et amplitude du type caractère

N°	char	N°	char	N°	char	N°	char
32		56	8	80	P	104	h
33	!	57	9	81	Q	105	i
34	"	58	:	82	R	106	j
35	#	59	;	83	S	107	k
36	$	60	<	84	T	108	l
37	%	61	=	85	U	109	m
38	&	62	>	86	V	110	n
39	'	63	?	87	W	111	o
40	(	64	@	88	X	112	p
41	)	65	A	89	Y	113	q
42	*	66	B	90	Z	114	r
43	+	67	C	91	[	115	s
44	,	68	D	92	\	116	t
45	-	69	E	93	]	117	u
46	.	70	F	94	^	118	v
47	/	71	G	95	_	119	w
48	0	72	H	96	`	120	x
49	1	73	I	97	a	121	y
50	2	74	J	98	b	122	z
51	3	75	K	99	c	123	{
52	4	76	L	100	d	124	¦
53	5	77	M	101	e	125	}
54	6	78	N	102	f	126	~
55	7	79	O	103	g	127	⌂

La table `ASCII` code tous les caractères usuels (états-uniens) sur 7 bits (128 positions). Sur un octet (8 bits), en C, on dispose :

- du type `char` défini sur $[-128..127]$;
- du type `unsigned char` défini sur $[0..255]$.

Taille et amplitude du type entier

Le type `int` en C peut être qualifié par un modificateur.

Avec Dev-C++ sous Windows :

- `short int` (2 octets) : [−32768..32767]
- `unsigned short int` (2 octets) : [0..65535]
- `int` (4 octets) : [−2147483648..2147483647]
- `unsigned int` (4 octets) : [0..4294967295]
- `long int` (4 octets) : [−2147483648..2147483647]
- `unsigned long int` (4 octets) : [0..4294927295]

Taille et amplitude du type flottant

Il existe deux types flottant en C : `float` et `double`.

Avec Dev-C++ sous Windows :

- `float` (4 octets) : $\pm[1.17549435 \times 10^{-38}..3.40282347 \times 10^{38}]$
- `double` (8 octets) : $\pm[2.2250738585072014 \times 10^{-308}..1.7976931348623158 \times 10^{308}]$

Les variables

Variable

C'est un nom donné à une valeur.

Un nom de variable représente une et une seule valeur qui peut évoluer au cours du temps (la valeur antérieure est perdue).

Déclaration

On déclare une variable en lui donnant un nom et un type.

- Déclaration algorithmique : *tension, puissance : flottant*
- Déclaration C : `float tension, puissance ;`

Les variables

Variable

C'est un nom donné à une valeur.

Un nom de variable représente une et une seule valeur qui peut évoluer au cours du temps (la valeur antérieure est perdue).

Déclaration

On déclare une variable en lui donnant un nom et un type.

- Déclaration algorithmique : *tension, puissance : flottant*
- Déclaration C : `float tension, puissance ;`

Les variables

Variable

C'est un nom donné à une valeur.

Un nom de variable représente une et une seule valeur qui peut évoluer au cours du temps (la valeur antérieure est perdue).

Déclaration

On déclare une variable en lui donnant un nom et un type.

- Déclaration algorithmique : *tension*, *puissance* : *flottant*
- Déclaration C : `float tension, puissance ;`

Les variables

Variable

C'est un nom donné à une valeur.

Un nom de variable représente une et une seule valeur qui peut évoluer au cours du temps (la valeur antérieure est perdue).

Déclaration

On déclare une variable en lui donnant un nom et un type.

- Déclaration algorithmique : *tension*, *puissance* : *flottant*
- Déclaration C : `float tension, puissance ;`

L'affectation

Définition

C'est l'opération qui permet de donner une valeur à une variable.

Dans une affectation, le membre de gauche reçoit le membre de droite (ce qui nécessite deux temps d'horloge différents).

- Notations algorithmiques :
  $index \leftarrow 3$
  $i \leftarrow i + 1$
- Notations C :
  ```
  index = 3 ; // Attention !
  i = i + 1 ; // ce n'est pas l'égalité mathématique
  ```

L'affectation

Définition

C'est l'opération qui permet de donner une valeur à une variable.

Dans une affectation, le membre de gauche reçoit le membre de droite (ce qui nécessite deux temps d'horloge différents).

- Notations algorithmiques :
  $index \leftarrow 3$
  $i \leftarrow i + 1$
- Notations C :
  ```
  index = 3 ; // Attention !
  i = i + 1 ; // ce n'est pas l'égalité mathématique
  ```

L'affectation

Définition

C'est l'opération qui permet de donner une valeur à une variable.

Dans une affectation, le membre de gauche reçoit le membre de droite (ce qui nécessite deux temps d'horloge différents).

- Notations algorithmiques :
  $index \leftarrow 3$
  $i \leftarrow i + 1$
- Notations C :
  ```
  index = 3 ; // Attention !
  i = i + 1 ; // ce n'est pas l'égalité mathématique
  ```

Les constantes

Définition

C'est une « variable » qui ne change pas de valeur au cours du programme.

Une constante doit toujours recevoir une valeur dès sa déclaration.

- Notation algorithmique : *phi* $\leftarrow$ 1.618 : *flottant*
- Notation C : `const float phi = 1.618;`

Les constantes

Définition

C'est une « variable » qui ne change pas de valeur au cours du programme.

Une constante doit toujours recevoir une valeur dès sa déclaration.

- Notation algorithmique : $phi \leftarrow 1.618$: *flottant*
- Notation C : `const float phi = 1.618 ;`

Les constantes

Définition

C'est une « variable » qui ne change pas de valeur au cours du programme.

Une constante doit toujours recevoir une valeur dès sa déclaration.

- Notation algorithmique : $phi \leftarrow 1.618$: *flottant*
- Notation C : `const float phi = 1.618 ;`

Les entrées/sorties

Ces deux opérations de base permettent de dialoguer : soit on saisit une valeur (*i-e* l'ordinateur attend une valeur pour une variable), soit on l'affiche à l'écran.

- Notations algorithmiques :
  AFFICHER("Entrez une couleur primaire")
  SAISIR(couleurPrim)
- Notations C :
  ```
  cout << "Entrez une couleur primaire" ;
  cin >> couleurPrim ;
  ```

Les entrées/sorties

Ces deux opérations de base permettent de dialoguer : soit on saisit une valeur (*i-e* l'ordinateur attend une valeur pour une variable), soit on l'affiche à l'écran.

- Notations algorithmiques :
  AFFICHER("Entrez une couleur primaire")
  SAISIR(couleurPrim)
- Notations C :
  ```
  cout << "Entrez une couleur primaire" ;
  cin >> couleurPrim ;
  ```

Les entrées/sorties

Ces deux opérations de base permettent de dialoguer : soit on saisit une valeur (*i-e* l'ordinateur attend une valeur pour une variable), soit on l'affiche à l'écran.

- Notations algorithmiques :
  AFFICHER("Entrez une couleur primaire")
  SAISIR(couleurPrim)
- Notations C :

```
cout << "Entrez une couleur primaire" ;
cin >> couleurPrim ;
```

Un programme est constitué d'instructions...

- Une instruction est une expression du langage qui peut être soit simple soit composée (on parle alors d'un *bloc d'instructions*).
- En algorithmique :
  - on écrit une instruction simple par ligne ;
  - un bloc d'instruction, dans une structure, est délimité par un début et une fin.
- en C :
  - une instruction simple est suivie par un terminateur (;) le point virgule ;
  - un bloc d'instructions est un ensemble d'instructions simples, encadré par une paire d'accolades : { `bloc_instructions` }.

Un programme est constitué d'instructions...

- Une instruction est une expression du langage qui peut être soit simple soit composée (on parle alors d'un *bloc d'instructions*).
- En algorithmique :
  - on écrit une instruction simple par ligne ;
  - un bloc d'instruction, dans une structure, est délimité par un début et une fin.
- en C :
  - une instruction simple est suivie par un terminateur (;) le point virgule ;
  - un bloc d'instructions est un ensemble d'instructions simples, encadré par une paire d'accolades : { `bloc_instructions` }.

Un programme est constitué d'instructions...

- Une instruction est une expression du langage qui peut être soit simple soit composée (on parle alors d'un *bloc d'instructions*).
- En algorithmique :
    - on écrit une instruction simple par ligne ;
    - un bloc d'instruction, dans une structure, est délimité par un début et une fin.
- en C :
    - une instruction simple est suivie par un terminateur (;) le point virgule ;
    - un bloc d'instructions est un ensemble d'instructions simples, encadré par une paire d'accolades : { `bloc_instructions` }.

...et de choix logiques

Définition

Une condition est une expression écrite entre parenthèses à valeur booléenne.

Une condition permet de prendre une décision élémentaire suivant son résultat : *VRAI* ou *FAUX*.

Dans un langage, les conditions sont mises en œuvre par des structures de contrôle.

Cette technique permet de suivre pas à pas l'exécution d'un algorithme.
On numérote les instructions de l'algorithme et, dans un tableau, on *suit* l'évolution des variables intéressantes :

Algorithme inconnu

DébutDéclaration
a, b : entier
FinDéclaration
1 $a \leftarrow 4$
2 $b \leftarrow 11$
3 $a \leftarrow b - a$
4 $b \leftarrow b - a$
5 $a \leftarrow a + b$

Trace de l'algorithme

N^o	a	b
1	4	
2		11
3	7	
4		4
5	11	

L'algorithme inconnu permute les valeurs de a et b.

Cette technique permet de suivre pas à pas l'exécution d'un algorithme.
On numérote les instructions de l'algorithme et, dans un tableau, on *suit* l'évolution des variables intéressantes :

Algorithme inconnu

DébutDéclaration
a, b : entier
FinDéclaration
1 $a \leftarrow 4$
2 $b \leftarrow 11$
3 $a \leftarrow b - a$
4 $b \leftarrow b - a$
5 $a \leftarrow a + b$

Trace de l'algorithme

N^o	a	b
1	4	
2		11
3	7	
4		4
5	11	

L'algorithme inconnu permute les valeurs de a et b.

Cette technique permet de suivre pas à pas l'exécution d'un algorithme.
On numérote les instructions de l'algorithme et, dans un tableau, on *suit* l'évolution des variables intéressantes :

Algorithme inconnu

DébutDéclaration
a, b : entier
FinDéclaration
1 $a \leftarrow 4$
2 $b \leftarrow 11$
3 $a \leftarrow b - a$
4 $b \leftarrow b - a$
5 $a \leftarrow a + b$

Trace de l'algorithme

N^o	a	b
1	4	
2		11
3	7	
4		4
5	11	

L'algorithme inconnu permute les valeurs de a et b.

Troisième partie III

Les instructions en algorithmique et en langage C

Les opérateurs arithmétiques

- addition + $a + b$
- soustraction − $a - b$
- opposé − $-a$
- produit * $a * b$
- division / a/b (attention à la division entière)
- modulo % $a\%b$ (pour les types entiers)

De plus, en C :

- incrémentation ++ pré : ++a ou post : a++
- décrémentation -- pré : --a ou post : a--

Les opérateurs relationnels

opérateur	algorithmique	C
égal	$==$	`==`
différent	$\neq$	`!=`
inférieur	$<$	`<`
supérieur	$>$	`>`
inférieur ou égal	$\leqslant$	`<=`
supérieur ou égal	$\geqslant$	`>=`

Les opérateurs booléens

opérateur	algorithmique	C
négation	$NON(a)$	!a
ou	$a\ OU\ b$	a || b
et	$a\ ET\ b$	a && b

Les opérateurs C binaires

Ces opérateurs C, très proches de la machine, servent à manipuler des mots bit à bit, par exemple pour faire des *masques* :

- et binaire & `a & b`
- ou binaire | `a | b`
- ou exclusif binaire ^ `a ^ b`
- décalage gauche << `a << b`
- décalage droit >> `a >> b`
- complément à 1 ~ `~a` (négation bit à bit)

Priorités des opérateurs arithmétiques

Il est sage d'utiliser la **règle pratique** suivante :

Niveau de priorité	Opérateurs	arithmétiques	
1	*	/	%
2	+	-	

Entourez tout le reste de parenthèses !

Quatrième partie IV

Les structures de contrôle

Notations de la sélection simple

Sélection simple : notation algorithmique

Si (*condition*)
    bloc_instructions
FinSi

Sélection simple : notation C

```
if(condition)
  bloc_instructions
```

Notations de la sélection simple

Sélection simple : notation algorithmique

Si (*condition*)
  bloc_instructions
FinSi

Sélection simple : notation C

```
if(condition)
  bloc_instructions
```

Notations de la sélection complète

Sélection complète : notation algorithmique

Si (*condition*)
    bloc_instructions
Sinon
    bloc_instructions
FinSi

Sélection complète : notation C

```
if(condition)
  bloc_instructions
else
  bloc_instructions
```

Notations de la sélection complète

Sélection complète : notation algorithmique

Si (*condition*)
    bloc_instructions
Sinon
    bloc_instructions
FinSi

Sélection complète : notation C

```
if(condition)
  bloc_instructions
else
  bloc_instructions
```

Sélection complète : exemple algorithmique

DébutDéclaration

$a \leftarrow 1.0, b \leftarrow -1.0, c \leftarrow -1.0$: flottant

$delta \leftarrow (b * b) - 4 * (a * c), r$: flottant

FinDéclaration

Si $(delta < 0)$

Afficher("Pas de solution réelle")

Sinon

Si $(delta == 0)$

Afficher("une solution double : ", $-b/(2 * a)$**)**

Sinon

$r \leftarrow Racine(delta)$

Afficher("deux solutions réelles : ", $(-b - r)/(2 * a)$**,
" et ",** $(-b + r)/(2 * a)$**)**

FinSi

FinSi

Sélection complète : exemple C

```
#include <iostream>
using namespace std; // evite d'écrire "std::cout"
#include <math.h>
int main(void)
{
    double a = 1.0, b = -1.0, c = -1.0, delta = (b*b)-4.0*(a*c), r;

  if(delta < 0)
    cout << "Pas de solutions reelles";
  else
    if(delta == 0)
      cout << "Une solution double : " << (-b/(2.0*a));
    else
    {
      r = sqrt(delta);
      cout << "Deux solutions reelles : " << ((-b-r)/(2.0*a))
           << " et " << ((-b+r)/(2.0*a)) << endl << endl;
    }
  system("PAUSE");
  return 0;
}
```

Notations de la boucle Faire .. TantQue

Boucle Faire .. TantQue : notation algorithmique

Faire
    bloc_instructions
TantQue(*condition*)

Boucle Faire .. TantQue : notation C

```
do
  bloc_instructions
while(condition);
```

Notations de la boucle Faire .. TantQue

Boucle Faire .. TantQue : notation algorithmique

Faire
bloc_instructions
TantQue(*condition*)

Boucle Faire .. TantQue : notation C

```
do
  bloc_instructions
while(condition);
```

Faire .. TantQue : exemples algorithmiques

Faire *exemple 1 de saisie filtrée*
  Afficher("Entrez un entier < 4 **: ")**
  Saisir(n**)**
TantQue$(n \geqslant 4)$

Faire *exemple 2 de saisie filtrée*
  Afficher("Entrez un entier $[4..9]$ **: ")**
  Saisir(n**)**
TantQue$((n < 4) OU (n > 9))$

Faire .. TantQue : exemples C

```
do  // Exemple 1 de saisie filtree
{
  cout << "Entrez un entier < 4 : ";
  cin >> n;
} while(n >= 4);


do  // Exemple 2 de saisie filtree
{
  cout << "Entrez un entier [4 .. 9] : ";
  cin >> n;
} while((n < 4) || (n > 9));
```

Notation de la boucle TantQue .. FinTantQue

Boucle TantQue .. FinTantQue

TantQue (*condition*)
  bloc_instructions
FinTantQue

Boucle Faire .. TantQue : notation C

```
while(condition)
  bloc_instructions
```

Notation de la boucle TantQue .. FinTantQue

Boucle TantQue .. FinTantQue

TantQue (*condition*)
bloc_instructions
FinTantQue

Boucle Faire .. TantQue : notation C

```
while(condition)
  bloc_instructions
```

TantQue .. FinTantQue : exemple algorithmique

DébutProgramme
  DébutDéclaration
    $n \leftarrow 32$: entier   *entier à tester*
    $cpt \leftarrow 0$: entier   *compteur de divisions par 2 de n*
  FinDéclaration

  TantQue $(n\%2 == 0)$
    $n \leftarrow n/2$
    $cpt \leftarrow cpt + 1$
  FinTantQue

  Afficher("32 est ", cpt**, " fois divisible par 2.")**
FinProgramme

TantQue .. FinTantQue : exemple C

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

int main(void)
{
    int n = 32, cpt = 0;

  while((n % 2) == 0)
  {
    n = n / 2;
    cpt = cpt + 1;
  }

  cout << "32 est " << cpt << " fois divisible par 2." << endl;

  cout << endl;
  system("PAUSE");
  return 0;
}
```

Notation de la boucle Pour .. FinPour

Boucle Pour .. FinPour : notation algorithmique

Pour *cpt* **Dans** [*a*..*b*] **ParPasDe** *n*
    bloc_instructions
FinPour

Boucle Pour .. FinPour : notation C

```
for(expression_1; expression_2; expression_3)
  bloc_instructions
```

Où expression_1 initialise la variable de contrôle, expression_2 donne la condition de rebouclage et expression_3 incrémente la variable de contrôle en fin de bloc_instructions.

Notation de la boucle Pour .. FinPour

Boucle Pour .. FinPour : notation algorithmique

Pour *cpt* **Dans** [*a*..*b*] **ParPasDe** *n*
    bloc_instructions
FinPour

Boucle Pour .. FinPour : notation C

```
for(expression_1; expression_2; expression_3)
  bloc_instructions
```

Où `expression_1` initialise la variable de contrôle, `expression_2` donne la condition de rebouclage et `expression_3` incrémente la variable de contrôle en fin de `bloc_instructions`.

Pour .. FinPour : exemple algorithmique

DébutProgramme <u>*Calcul de factorielle 5*</u>

DébutDéclaration

$n \leftarrow 5, fact \leftarrow 1, cpt$: entier

FinDéclaration

Pour *cpt* **Dans** $[1..n]$ **ParPasDe** 1

fact ← fact*cpt

FinPour

Afficher("5 ! = ", *fact***)**

FinProgramme

Pour .. FinPour : exemple C

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

int main(void)
{
    long n = 5, fact = 1;

  for(long cpt = 1; cpt <= n; cpt = cpt + 1)
  {
    fact = fact * cpt;
  }
  cout << n << "! = " << fact << endl << endl;

  system("PAUSE");
  return 0;
}
```

Les instructions C de ruptures de boucle

Le langage C offre deux instructions :

- `break` provoque la fin prématurée de la boucle `while`, `do` ou `for` qui contient *directement* le `break` ;
- `continue` provoque le rebouclage immédiat, comme si on venait d'exécuter la dernière instruction du corps de la boucle.

Les instructions C de ruptures de boucle

Le langage C offre deux instructions :

- `break` provoque la fin prématurée de la boucle `while`, `do` ou `for` qui contient *directement* le `break` ;
- `continue` provoque le rebouclage immédiat, comme si on venait d'exécuter la dernière instruction du corps de la boucle.

iUT ORSAY

Les instructions C de ruptures de boucle

Le langage C offre deux instructions :

- `break` provoque la fin prématurée de la boucle `while`, `do` ou `for` qui contient *directement* le `break` ;
- `continue` provoque le rebouclage immédiat, comme si on venait d'exécuter la dernière instruction du corps de la boucle.

break : exemple C

```
for(int i = 0; i < n; i = i + 1)
{
  for(int j = 0; j < m; j = j + 1)
  {
    if(a[j] == x)
    {
      break; // sort du second for
    }
  }
  if(b[i] == x)
  {
    break; // sort du premier for
  }
}
```

continue : exemple C

```
while(x < 2)
{
  x = f(x);
  if(g(x) > x)
  {
    continue; // boucle directement sur while
  }
  x = h(x);
}
```

Cinquième partie V

Les fonctions en algorithmique et en C

Caractéristiques des fonctions

- Une fonction ne modifie pas la valeurs de ses arguments en entrée ;
- elle se termine par une instruction de retour qui rend un résultat et un seul ;
- on l'utilise toujours dans une expression (affectation, affichage. . .).

Caractéristiques des fonctions

- Une fonction ne modifie pas la valeurs de ses arguments en entrée ;
- elle se termine par une instruction de retour qui rend un résultat et un seul ;
- on l'utilise toujours dans une expression (affectation, affichage. . .).

Caractéristiques des fonctions

- Une fonction ne modifie pas la valeurs de ses arguments en entrée ;
- elle se termine par une instruction de retour qui rend un résultat et un seul ;
- on l'utilise toujours dans une expression (affectation, affichage. . .).

Notation algorithmique

L'écriture d'une fonction requiert un en-tête et un corps :

- l'en-tête comprend : `DébutFonction`, le nom de la fonction, une liste d'arguments entre parenthèses, puis le type de la fonction. Les arguments sont séparés par une virgule et sont notés → *nom_arg* : *type_arg* ;
- le corps se termine par `FinFonction` précédé d'une instruction de retour d'une expression de même type que la fonction.

Exemple :

DébutFonction DOUBLER(→ x : flottant) : flottant
  Retourner $(2 * x)$
FinFonction

Notation algorithmique

L'écriture d'une fonction requiert un en-tête et un corps :

- l'en-tête comprend : `DébutFonction`, le nom de la fonction, une liste d'arguments entre parenthèses, puis le type de la fonction. Les arguments sont séparés par une virgule et sont notés $\longrightarrow$ *nom_arg* : *type_arg* ;
- le corps se termine par `FinFonction` précédé d'une instruction de retour d'une expression de même type que la fonction.

Exemple :

DébutFonction DOUBLER($\rightarrow$ x : flottant) : flottant
  Retourner $(2 * x)$
FinFonction

Notation algorithmique

L'écriture d'une fonction requiert un en-tête et un corps :

- l'en-tête comprend : `DébutFonction`, le nom de la fonction, une liste d'arguments entre parenthèses, puis le type de la fonction. Les arguments sont séparés par une virgule et sont notés $\rightarrow$ *nom_arg* : *type_arg* ;
- le corps se termine par `FinFonction` précédé d'une instruction de retour d'une expression de même type que la fonction.

Exemple :

DébutFonction DOUBLER($\rightarrow$ x : flottant) : flottant
  Retourner $(2 * x)$
FinFonction

Notation C 1/2

L'écriture d'une fonction requiert un en-tête et un corps :

- l'en-tête comprend : le type de retour, le nom de la fonction suivit d'une liste d'arguments entre parenthèses. Les arguments sont séparés par une virgule et sont notés :

  `type_arg nom_arg, type_arg nom_arg, ...`

- le corps est un bloc d'instructions précédé d'une instruction de retour d'une expression de même type que la fonction.

Exemple :

```
float Doubler(float x)
{
  return 2.0*x;
}
```

Notation C 1/2

L'écriture d'une fonction requiert un en-tête et un corps :

- l'en-tête comprend : le type de retour, le nom de la fonction suivit d'une liste d'arguments entre parenthèses. Les arguments sont séparés par une virgule et sont notés :

  `type_arg nom_arg, type_arg nom_arg, ...`

- le corps est un bloc d'instructions précédé d'une instruction de retour d'une expression de même type que la fonction.

Exemple :

```
float Doubler(float x)
{
  return 2.0*x;
}
```

Notation C 1/2

L'écriture d'une fonction requiert un en-tête et un corps :

- l'en-tête comprend : le type de retour, le nom de la fonction suivit d'une liste d'arguments entre parenthèses. Les arguments sont séparés par une virgule et sont notés :

  `type_arg nom_arg, type_arg nom_arg, ...`
- le corps est un bloc d'instructions précédé d'une instruction de retour d'une expression de même type que la fonction.

Exemple :

```
float Doubler(float x)
{
  return 2.0*x;
}
```

Notation C 2/2

Une fonction possède trois aspects :

- le prototype : c'est la déclaration nécessaire avant tout ;
- l'appel : c'est l'utilisation d'une fonction à l'intérieur d'une autre fonction (par exemple le programme principal) ;
- la définition : c'est l'écriture proprement dite de la fonction, en-tête et corps.

Notation C 2/2

Une fonction possède trois aspects :

- le prototype : c'est la déclaration nécessaire avant tout ;
- l'appel : c'est l'utilisation d'une fonction à l'intérieur d'une autre fonction (par exemple le programme principal) ;
- la définition : c'est l'écriture proprement dite de la fonction, en-tête et corps.

Notation C 2/2

Une fonction possède trois aspects :

- le prototype : c'est la déclaration nécessaire avant tout ;
- l'appel : c'est l'utilisation d'une fonction à l'intérieur d'une autre fonction (par exemple le programme principal) ;
- la définition : c'est l'écriture proprement dite de la fonction, en-tête et corps.

Notation C 2/2

Une fonction possède trois aspects :

- le prototype : c'est la déclaration nécessaire avant tout ;
- l'appel : c'est l'utilisation d'une fonction à l'intérieur d'une autre fonction (par exemple le programme principal) ;
- la définition : c'est l'écriture proprement dite de la fonction, en-tête et corps.

Le passage des arguments en C 1/2

Entre la fonction appelante et la définition appelée se pose le problème de la *communication* des arguments. On distingue trois modes :

- le passage par valeur pour les arguments en entrée : une copie est créée, affectée de la valeur. L'appel ne modifie pas la valeur du paramètre ;
- le passage par référence pour les arguments en entrée/sortie : le compilateur travaille sur un alias du paramètre. La valeur du paramètre est changée par la fonction ;
- le passage par adresse, implicitement utilisé par les tableaux, sera étudié dans le chapitre dédié aux pointeurs.

Le passage des arguments en C 1/2

Entre la fonction appelante et la définition appelée se pose le problème de la *communication* des arguments. On distingue trois modes :

- le passage par valeur pour les arguments en entrée : une copie est créée, affectée de la valeur. L'appel ne modifie pas la valeur du paramètre ;
- le passage par référence pour les arguments en entrée/sortie : le compilateur travaille sur un alias du paramètre. La valeur du paramètre est changée par la fonction ;
- le passage par adresse, implicitement utilisé par les tableaux, sera étudié dans le chapitre dédié aux pointeurs.

Le passage des arguments en C 1/2

Entre la fonction appelante et la définition appelée se pose le problème de la *communication* des arguments. On distingue trois modes :

- le passage par valeur pour les arguments en entrée : une copie est créée, affectée de la valeur. L'appel ne modifie pas la valeur du paramètre ;
- le passage par référence pour les arguments en entrée/sortie : le compilateur travaille sur un alias du paramètre. La valeur du paramètre est changée par la fonction ;
- le passage par adresse, implicitement utilisé par les tableaux, sera étudié dans le chapitre dédié aux pointeurs.

Le passage des arguments en C 1/2

Entre la fonction appelante et la définition appelée se pose le problème de la *communication* des arguments. On distingue trois modes :

- le passage par valeur pour les arguments en entrée : une copie est créée, affectée de la valeur. L'appel ne modifie pas la valeur du paramètre ;
- le passage par référence pour les arguments en entrée/sortie : le compilateur travaille sur un alias du paramètre. La valeur du paramètre est changée par la fonction ;
- le passage par adresse, implicitement utilisé par les tableaux, sera étudié dans le chapitre dédié aux pointeurs.

Le passage des arguments en C 2/2

Exemple :

```
int F_ParValeur(int x)
{
    x = ...  // x ne change pas dans l'appelant
}

int F_ParReference(int &y)
{
    y = ...  // y change dans l'appelant
}

int F_ParAdresse(int t[])
{
    t[1] = ...  // t[1] change dans l'appelant
}
```

Notion de récursivité 1/2

Définition

Une fonction récursive peut s'appeler elle-même.

C'est le cas des fonctions mathématiques définies par récurrence :

$$n! \begin{cases} 1 & \text{si} \quad n = 0 \\ n\,(n-1)! & \text{si} \quad n \geqslant 1 \end{cases}$$

Bien que cette méthode soit souvent plus difficile à comprendre et à coder que la méthode classique dite *itérative*, elle est dans certains cas l'application la plus directe de sa définition mathématique.

Notion de récursivité 1/2

Définition

Une fonction récursive peut s'appeler elle-même.

C'est le cas des fonctions mathématiques définies par récurrence :

$$n! \begin{cases} 1 & \text{si} \quad n = 0 \\ n\,(n-1)! & \text{si} \quad n \geqslant 1 \end{cases}$$

Bien que cette méthode soit souvent plus difficile à comprendre et à coder que la méthode classique dite *itérative*, elle est dans certains cas l'application la plus directe de sa définition mathématique.

Notion de récursivité 1/2

Définition

Une fonction récursive peut s'appeler elle-même.

C'est le cas des fonctions mathématiques définies par récurrence :

$$n! \begin{cases} 1 & \text{si} \quad n = 0 \\ n\,(n-1)! & \text{si} \quad n \geqslant 1 \end{cases}$$

Bien que cette méthode soit souvent plus difficile à comprendre et à coder que la méthode classique dite *itérative*, elle est dans certains cas l'application la plus directe de sa définition mathématique.

Notion de récursivité : calcul de $n!$ 2/2

DébutFonction FACTORIELLE($\rightarrow n$: entier) : entier
  Si $(n == 0)$
    Retourner 1
  Sinon
    Retourner $(n * Factorielle(n-1))$
  FinSi
FinFonction

```
long Factorielle(long n)
{
  if(n == 0)
    return 1;
  else
    return n*Factorielle(n-1);
}
```

Sixième partie VI

Pointeurs, tableaux et procédures

Les procédures 1/2

- Généralisant la notion de fonction, les procédures permettent de modifier la valeur de plusieurs de leurs arguments ;
- Notation algorithmique proche de celle des fonctions, sauf :
  - `DébutProcédure` ... `FinProcédure`
  - pas de type de retour (donc pas d'instruction `Retourner`)
  - les arguments peuvent encore être en entrée ($\longrightarrow$) non modifiés, mais peuvent aussi être modifiés en étant soit en sortie ($\longleftarrow$) soit en entrée/sortie ($\longleftrightarrow$)
- Leur appel constitue à lui seul une instruction.

Les procédures 1/2

- Généralisant la notion de fonction, les procédures permettent de modifier la valeur de plusieurs de leurs arguments ;
- Notation algorithmique proche de celle des fonctions, sauf :
  - `DébutProcédure` ... `FinProcédure`
  - pas de type de retour (donc pas d'instruction `Retourner`)
  - les arguments peuvent encore être en entrée ($\longrightarrow$) non modifiés, mais peuvent aussi être modifiés en étant soit en sortie ($\longleftarrow$) soit en entrée/sortie ($\longleftrightarrow$)
- Leur appel constitue à lui seul une instruction.

Les procédures 1/2

- Généralisant la notion de fonction, les procédures permettent de modifier la valeur de plusieurs de leurs arguments ;
- Notation algorithmique proche de celle des fonctions, sauf :
  - `DébutProcédure` ...`FinProcédure`
  - pas de type de retour (donc pas d'instruction `Retourner`)
  - les arguments peuvent encore être en entrée ($\longrightarrow$) non modifiés, mais peuvent aussi être modifiés en étant soit en sortie ($\longleftarrow$) soit en entrée/sortie ($\longleftrightarrow$)
- Leur appel constitue à lui seul une instruction.

Les procédures : échange de deux entiers 2/2

DébutProcedure ECHANGER(↔a : entier, ↔b : entier)
    DébutDéclaration
        temp ← a : entier   *variable temporaire*
    FinDéclaration
    a ← *b*
    b ← *temp*
FinProcedure

```
void Echanger(int &a, int &b) // passage par reference
{
   int temp = a; // variable temporaire

  a = b;
  b = temp;
}
```

Les procédures : échange de deux entiers 2/2

DébutProcedure ECHANGER(↔a : entier, ↔b : entier)
  DébutDéclaration
    temp ← a : entier <u>*variable temporaire*</u>
  FinDéclaration
  $a \leftarrow b$
  $b \leftarrow temp$
FinProcedure

```
void Echanger(int &a, int &b) // passage par reference
{
    int temp = a; // variable temporaire

  a = b;
  b = temp;
}
```

Définition des pointeurs

Définition

C'est une variable qui contient l'adresse d'une autre variable.

- Un objet dont la valeur est une adresse d'octet est donc un pointeur.
- Un pointeur est typé (pointeur d'`int`, de `double`, de `char`. . .).
- Il existe un pointeur particulier, `NULL`, qui est défini dans `<stddef.h>`

Définition des pointeurs

Définition

C'est une variable qui contient l'adresse d'une autre variable.

- Un objet dont la valeur est une adresse d'octet est donc un pointeur.
- Un pointeur est typé (pointeur d'`int`, de `double`, de `char`...).
- Il existe un pointeur particulier, `NULL`, qui est défini dans `<stddef.h>`

Définition des pointeurs

Définition

C'est une variable qui contient l'adresse d'une autre variable.

- Un objet dont la valeur est une adresse d'octet est donc un pointeur.
- Un pointeur est typé (pointeur d'`int`, de `double`, de `char`...).
- Il existe un pointeur particulier, `NULL`, qui est défini dans `<stddef.h>`

Définition des pointeurs

Définition

C'est une variable qui contient l'adresse d'une autre variable.

- Un objet dont la valeur est une adresse d'octet est donc un pointeur.
- Un pointeur est typé (pointeur d'`int`, de `double`, de `char`...).
- Il existe un pointeur particulier, `NULL`, qui est défini dans `<stddef.h>`

Notations et déclaration

Le C introduit deux opérateurs sur les pointeurs :

- l'opérateur & qui désigne l'*adresse* d'une variable ;
- l'opérateur * qui donne la *valeur* de l'objet pointé.

Déclaration et manipulation :

```
int *p ;      // p : pointeur d'int (c'est une adresse)
int x = 3 ;      // x est un int qui vaut 3
p = &x ;      // p pointe vers l'adresse de x
cout << *p ;      // affiche 3
```

Notations et déclaration

Le C introduit deux opérateurs sur les pointeurs :

- l'opérateur & qui désigne l'*adresse* d'une variable ;
- l'opérateur * qui donne la *valeur* de l'objet pointé.

Déclaration et manipulation :

```
int *p ;      // p : pointeur d'int (c'est une adresse)
int x = 3 ;      // x est un int qui vaut 3
p = &x ;      // p pointe vers l'adresse de x
cout << *p ;     // affiche 3
```

Notations et déclaration

Le C introduit deux opérateurs sur les pointeurs :

- l'opérateur & qui désigne l'*adresse* d'une variable ;
- l'opérateur * qui donne la *valeur* de l'objet pointé.

Déclaration et manipulation :

```
int *p ;      // p : pointeur d'int (c'est une adresse)
int x = 3 ;      // x est un int qui vaut 3
p = &x ;      // p pointe vers l'adresse de x
cout << *p ;     // affiche 3
```

Procédures et passage des arguments par adresse

Le mécanisme est le même que pour le passage par référence (la procédure change la valeur du paramètre), mais les notations sont différentes :

```
void RAZ(int *a); // prototype de la procedure

int main(void)
{
    int i = 15;

  RAZ(&i); // apres l'appel, i vaut 0
  return 0;
}

void RAZ(int *a) // definition, passage par adresse
{
  *a = 0;
}
```

Procédures et passage des arguments par adresse

Le mécanisme est le même que pour le passage par référence (la procédure change la valeur du paramètre), mais les notations sont différentes :

```
void RAZ(int *a); // prototype de la procedure

int main(void)
{
    int i = 15;

  RAZ(&i); // apres l'appel, i vaut 0
  return 0;
}

void RAZ(int *a) // definition, passage par adresse
{
  *a = 0;
}
```

Relations entre pointeurs et tableaux

Les pointeurs et les tableaux sont très liés en C :

```
int tab[N], *ptab;
```

Écrire `ptab = &tab[0];` signifie que `ptab` pointe sur l'adresse de la première case de `tab`. Ce qui s'écrit plus simplement :

```
ptab = tab;
```

Passage des tableaux en arguments :

- appel : `f(tab);  // on passe bien une adresse`
- prototype (et définition sans le ;), deux formes :
  - `int f(int tab[]);  // aspect tableau`
  - `int f(int *tab);  // aspect pointeur`

L'utilisation des pointeurs à la place des tableaux implique une gestion de la mémoire (*cf* « Allocation programmée » en annexe).

Relations entre pointeurs et tableaux

Les pointeurs et les tableaux sont très liés en C :

```
int tab[N], *ptab;
```

Écrire `ptab = &tab[0];` signifie que `ptab` pointe sur l'adresse de la première case de `tab`. Ce qui s'écrit plus simplement :

```
ptab = tab;
```

Passage des tableaux en arguments :

- appel : `f(tab);   // on passe bien une adresse`
- prototype (et définition sans le ;), deux formes :
  - `int f(int tab[]);   // aspect tableau`
  - `int f(int *tab);   // aspect pointeur`

L'utilisation des pointeurs à la place des tableaux implique une gestion de la mémoire (*cf* « Allocation programmée » en annexe).

Relations entre pointeurs et tableaux

Les pointeurs et les tableaux sont très liés en C :

```
int tab[N], *ptab;
```

Écrire `ptab = &tab[0];` signifie que `ptab` pointe sur l'adresse de la première case de `tab`. Ce qui s'écrit plus simplement :

```
ptab = tab;
```

Passage des tableaux en arguments :

- appel : `f(tab);   // on passe bien une adresse`
- prototype (et définition sans le ;), deux formes :
  - `int f(int tab[]);   // aspect tableau`
  - `int f(int *tab);   // aspect pointeur`

L'utilisation des pointeurs à la place des tableaux implique une gestion de la mémoire (*cf* « Allocation programmée » en annexe).

Relations entre pointeurs et tableaux

Les pointeurs et les tableaux sont très liés en C :

```
int tab[N], *ptab;
```

Écrire `ptab = &tab[0];` signifie que `ptab` pointe sur l'adresse de la première case de `tab`. Ce qui s'écrit plus simplement :

```
ptab = tab;
```

Passage des tableaux en arguments :

- appel : `f(tab);  // on passe bien une adresse`
- prototype (et définition sans le ;), deux formes :
  - `int f(int tab[]);  // aspect tableau`
  - `int f(int *tab);  // aspect pointeur`

L'utilisation des pointeurs à la place des tableaux implique une gestion de la mémoire (*cf* « Allocation programmée » en annexe).

Septième partie VII

Aspects avancés du langage C

Les entrées/sorties sur fichiers `ASCII` 1/2

En C, un fichier `ASCII` est géré par un tampon d'entrée-sortie appelé un flux de type `FILE*`. En déclarant : `FILE *fic ;` la bibliothèque standard (en-tête <stdio.h>) permet :

- l'ouverture du flux : `fic = fopen(nomFichier, mode) ;` où mode est "w" (écriture) ou "r" (lecture) ;
- l'écriture dans le flux : `fprintf(fic, "format", vars) ;`
- la saisie depuis le flux : `fscanf(fic, "format", vars) ;`
- la fermeture du flux : `fclose(fic) ;`

(Voir les *formats* en annexe.)

Les entrées/sorties sur fichiers ASCII 1/2

En C, un fichier `ASCII` est géré par un tampon d'entrée-sortie appelé un flux de type `FILE*`. En déclarant : `FILE *fic ;` la bibliothèque standard (en-tête <stdio.h>) permet :

- l'ouverture du flux : `fic = fopen(nomFichier, mode) ;` où mode est "w" (écriture) ou "r" (lecture) ;
- l'écriture dans le flux : `fprintf(fic, "format", vars) ;`
- la saisie depuis le flux : `fscanf(fic, "format", vars) ;`
- la fermeture du flux : `fclose(fic) ;`

(Voir les *formats* en annexe.)

Les entrées/sorties sur fichiers ASCII 1/2

En C, un fichier ASCII est géré par un tampon d'entrée-sortie appelé un flux de type `FILE*`. En déclarant : `FILE *fic ;` la bibliothèque standard (en-tête <stdio.h>) permet :

- l'ouverture du flux : `fic = fopen(nomFichier, mode) ;` où mode est "w" (écriture) ou "r" (lecture) ;
- l'écriture dans le flux : `fprintf(fic, "format", vars) ;`
- la saisie depuis le flux : `fscanf(fic, "format", vars) ;`
- la fermeture du flux : `fclose(fic) ;`

(Voir les *formats* en annexe.)

Les entrées/sorties sur fichiers ASCII 1/2

En C, un fichier ASCII est géré par un tampon d'entrée-sortie appelé un flux de type `FILE*`. En déclarant : `FILE *fic ;` la bibliothèque standard (en-tête <stdio.h>) permet :

- l'ouverture du flux : `fic = fopen(nomFichier, mode) ;` où mode est "w" (écriture) ou "r" (lecture) ;
- l'écriture dans le flux : `fprintf(fic, "format", vars) ;`
- la saisie depuis le flux : `fscanf(fic, "format", vars) ;`
- la fermeture du flux : `fclose(fic) ;`

(Voir les *formats* en annexe.)

Les entrées/sorties sur fichiers ASCII 1/2

En C, un fichier ASCII est géré par un tampon d'entrée-sortie appelé un flux de type FILE*. En déclarant : `FILE *fic ;` la bibliothèque standard (en-tête <stdio.h>) permet :

- l'ouverture du flux : `fic = fopen(nomFichier, mode) ;` où mode est "w" (écriture) ou "r" (lecture) ;
- l'écriture dans le flux : `fprintf(fic, "format", vars) ;`
- la saisie depuis le flux : `fscanf(fic, "format", vars) ;`
- la fermeture du flux : `fclose(fic) ;`

(Voir les *formats* en annexe.)

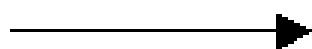
Les entrées/sorties sur fichiers ASCII 2/2

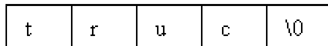
```
#include <stdio.h>  // pour la gestion des fichiers
#include <stdlib.h>

// déclarations
    const int N = 100;
    FILE    *fic;                   // déclaraion d'un flux
    double   x[N], y[N];            // tableaux a lire
    char     nomFichier[]= "..."; // nom du fichier

// utilisation
  fic = fopen(nomFichier, "r");   // ouverture en lecture
  if(fic == NULL)  // gestion d'erreur
  {
    fprintf(stderr, "ouverture impossible, on sort...");
    exit(1);
  }
  for(int i = 0; i < N; i = i + 1)
    fscanf(fic, "%lf %lf", &x[i], &y[i]); // lecture
  fclose(fic);     // fermeture du flux
```

Les chaînes de caractères

En C, les chaînes de caractères sont des tableaux de `char` :

```
char str[] = "truc";
```





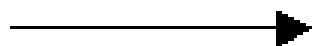
Dans leur représentation interne, ils doivent être terminés par le caractère nul (valeur *sentinelle*) pour en détecter la fin. On parle de chaîne bien formée.

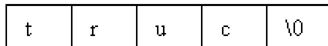
On les déclare soit comme un tableau de caractères (qui est donc *modifiable*), soit comme un pointeur de caractères (chaîne littérale *constante*).

Les chaînes de caractères

En C, les chaînes de caractères sont des tableaux de `char` :

```
char str[] = "truc";
```





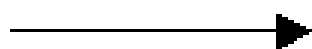
Dans leur représentation interne, ils doivent être terminés par le caractère nul (valeur *sentinelle*) pour en détecter la fin. On parle de chaîne bien formée.

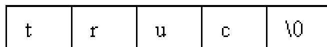
On les déclare soit comme un tableau de caractères (qui est donc *modifiable*), soit comme un pointeur de caractères (chaîne littérale *constante*).

Les chaînes de caractères

En C, les chaînes de caractères sont des tableaux de `char` :

```
char str[] = "truc";
```





Dans leur représentation interne, ils doivent être terminés par le caractère nul (valeur *sentinelle*) pour en détecter la fin. On parle de chaîne bien formée.

On les déclare soit comme un tableau de caractères (qui est donc *modifiable*), soit comme un pointeur de caractères (chaîne littérale *constante*).

Traitements sur les chaînes

Les fonctions de traitements de chaînes sont définies dans l'en-tête <string.h>. Les principales sont :

- longueur : fonction `strlen` ;
- comparaison : fonction `strcmp` ;
- recopie : fonction `strcpy` ;
- concaténation : fonction `strcat`.

Traitements sur les chaînes

Les fonctions de traitements de chaînes sont définies dans l'en-tête <string.h>. Les principales sont :

- longueur : fonction `strlen` ;
- comparaison : fonction `strcmp` ;
- recopie : fonction `strcpy` ;
- concaténation : fonction `strcat`.

Traitements sur les chaînes

Les fonctions de traitements de chaînes sont définies dans l'en-tête <string.h>. Les principales sont :

- longueur : fonction `strlen` ;
- comparaison : fonction `strcmp` ;
- recopie : fonction `strcpy` ;
- concaténation : fonction `strcat`.

Traitements sur les chaînes

Les fonctions de traitements de chaînes sont définies dans l'en-tête <string.h>. Les principales sont :

- longueur : fonction `strlen` ;
- comparaison : fonction `strcmp` ;
- recopie : fonction `strcpy` ;
- concaténation : fonction `strcat`.

Traitements sur les chaînes

Les fonctions de traitements de chaînes sont définies dans l'en-tête <string.h>. Les principales sont :

- longueur : fonction `strlen` ;
- comparaison : fonction `strcmp` ;
- recopie : fonction `strcpy` ;
- concaténation : fonction `strcat`.

Exemple de traitements sur les chaînes

```
// fonction strcpy(), version tableaux de caracteres
void strcpy(char s[], char t[])
{
    int i = 0;

  while((s[i] = t[i]) != '\0')
  {
    i = i + 1;
  }
}
```

```
// fonction strcpy(), version pointeurs de caracteres
void strcpy(char *s, char *t)
{
  while(*s++ = *t++)
    ;
}
```

Exemple de traitements sur les chaînes

```
// fonction strcpy(), version tableaux de caracteres
void strcpy(char s[], char t[])
{
    int i = 0;

  while((s[i] = t[i]) != '\0')
  {
    i = i + 1;
  }
}
```

```
// fonction strcpy(), version pointeurs de caracteres
void strcpy(char *s, char *t)
{
  while(*s++ = *t++)
    ;
}
```

Les structures 1/2

Les structures font parties des types étiquetés (`enum`, `union`, `struct`), les seuls que l'on peut définir en tant que *nouveaux types*.

Une structure permet de traiter un ensemble d'informations hétérogènes comme un tout. Exemple, l'adresse sur une lettre :

```
struct Adresse
{
  char nom[20];
  prenom[20];
  unsigned short numeroRue;
  char rue[20];
  unsigned int codePostal;
  char ville[20];
};
```

IUT ORSAY

Les structures 1/2

Les structures font parties des types étiquetés (`enum`, `union`, `struct`), les seuls que l'on peut définir en tant que *nouveaux types*.

Une structure permet de traiter un ensemble d'informations hétérogènes comme un tout. Exemple, l'adresse sur une lettre :

```
struct Adresse
{
  char nom[20];
  prenom[20];
  unsigned short numeroRue;
  char rue[20];
  unsigned int codePostal;
  char ville[20];
};
```

Les structures 1/2

Les structures font parties des types étiquetés (`enum`, `union`, `struct`), les seuls que l'on peut définir en tant que *nouveaux types*.

Une structure permet de traiter un ensemble d'informations hétérogènes comme un tout. Exemple, l'adresse sur une lettre :

```
struct Adresse
{
  char nom[20];
  prenom[20];
  unsigned short numeroRue;
  char rue[20];
  unsigned int codePostal;
  char ville[20];
};
```

Les structures 2/2

Exemple de déclaration et d'utilisation de structure :

```
// declaration
struct Adresse monAdresse;

// affectation
strcpy(monAdresse.nom, "Perinet");
strcpy(monAdresse.prenom, "Francois");
monAdresse.numeroRue = 99;
strcpy(monAdresse.rue, "avenue des Champs-Elysees");
monAdresse.codePostal = 75008;
strcpy(monAdresse.ville, "Paris");
```

Les structures 2/2

Exemple de déclaration et d'utilisation de structure :

```
// declaration
struct Adresse monAdresse;

// affectation
strcpy(monAdresse.nom, "Perinet");
strcpy(monAdresse.prenom, "Francois");
monAdresse.numeroRue = 99;
strcpy(monAdresse.rue, "avenue des Champs-Elysees");
monAdresse.codePostal = 75008;
strcpy(monAdresse.ville, "Paris");
```

L'allocation programmée 1/2

Lorsqu'on utilise un tableau, on doit le déclarer en précisant son type et sa taille. La taille est parfois inconnue *a priori*, on doit la gérer à l'exécution. On parle d'allocation programmée ou de tableaux dynamiques.

La bibliothèque standard C offre une fonction d'allocation, `malloc`, qui attend une taille d'octets à allouer et retourne un pointeur de `void (void *)`, et une fonction de désallocation, `free`, qui reçoit l'adresse d'une zone mémoire allouée par `malloc` (un pointeur) et la rend au système d'exploitation.

L'allocation programmée 1/2

Lorsqu'on utilise un tableau, on doit le déclarer en précisant son type et sa taille. La taille est parfois inconnue *a priori*, on doit la gérer à l'exécution. On parle d'allocation programmée ou de tableaux dynamiques.

La bibliothèque standard C offre une fonction d'allocation, `malloc`, qui attend une taille d'octets à allouer et retourne un pointeur de `void` (`void *`), et une fonction de désallocation, `free`, qui reçoit l'adresse d'une zone mémoire allouée par `malloc` (un pointeur) et la rend au système d'exploitation.

L'allocation programmée 2/2

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
    char *str;

  // allocation programmee
  if ((str = (char *) malloc(10)) == NULL) {
    printf("pas assez de memoire !\n");
    exit(1); // sortie en erreur
  }
  strcpy(str, "Bonjour");
  printf("Message : %s\n", str);

  free(str); // libere la memoire

  return 0;
}
```

Les arguments de la ligne de commande

On peut passer deux arguments à la fonction `main()` :

- `int argc` nombre d'arguments de la ligne de commande ;
- `char *argv[]` contient les arguments, un par chaîne de caractères.

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

int main(int argc, char *argv[])
{
  cout << "argc = " << argc << endl;
  for(int i = 0; i <= argc; i = i + 1)
  {
    cout << "argv[" << i << "] = " << argv[i] << endl;
  }
  return 0;
}
```

Les arguments de la ligne de commande

On peut passer deux arguments à la fonction `main()` :

- `int argc` nombre d'arguments de la ligne de commande ;
- `char *argv[]` contient les arguments, un par chaîne de caractères.

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

int main(int argc, char *argv[])
{
  cout << "argc = " << argc << endl;
  for(int i = 0; i <= argc; i = i + 1)
  {
    cout << "argv[" << i << "] = " << argv[i] << endl;
  }
  return 0;
}
```

Les arguments de la ligne de commande

On peut passer deux arguments à la fonction `main()` :

- `int argc` nombre d'arguments de la ligne de commande ;
- `char *argv[]` contient les arguments, un par chaîne de caractères.

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

int main(int argc, char *argv[])
{
  cout << "argc = " << argc << endl;
  for(int i = 0; i <= argc; i = i + 1)
  {
    cout << "argv[" << i << "] = " << argv[i] << endl;
  }
  return 0;
}
```

Pour aller plus loin. . .

Brian W. Kerningham et Denis Ritchie
Le langage C
Masson.

Philippe Drix
Langage C, norme ANSI. Vers une approche orientée objet
Masson.

S.P. Harbisson et G.L. Steele Jr.
Langage C, manuel de référence
Masson.

Peter Prinz et Ulla Kirch–Prinz
C précis et concis
O'Reilly.

Pour aller plus loin. . .

Brian W. Kerningham et Denis Ritchie
Le langage C
Masson.

Philippe Drix
Langage C, norme ANSI. Vers une approche orientée objet
Masson.

S.P. Harbisson et G.L. Steele Jr.
Langage C, manuel de référence
Masson.

Peter Prinz et Ulla Kirch–Prinz
C précis et concis
O'Reilly.

Pour aller plus loin...

Brian W. KERNINGHAM et Denis RITCHIE
Le langage C
Masson.

Philippe DRIX
Langage C, norme ANSI. Vers une approche orientée objet
Masson.

S.P. HARBISSON et G.L. STEELE JR.
Langage C, manuel de référence
Masson.

Peter PRINZ et Ulla KIRCH–PRINZ
C précis et concis
O'Reilly.

Pour aller plus loin...

Brian W. Kerninghan et Denis Ritchie
Le langage C
Masson.

Philippe Drix
Langage C, norme ANSI. Vers une approche orientée objet
Masson.

S.P. Harbisson et G.L. Steele Jr.
Langage C, manuel de référence
Masson.

Peter Prinz et Ulla Kirch–Prinz
C précis et concis
O'Reilly.

Annexes

- Les formats d'entrée-sortie
- Un « programme type » en C

Les principaux formats des familles printf et scanf

- %d : int, %hd : short int, %ld : long int
- %u : unsigned int
- %o : conversion octale
- %x : conversion héxadécimale
- %c : char
- %s : char[] ou char*
- %f : float
- %lf : double

Exemple type d'un programme C 1/4

Première partie identification du programme :

```
// TP747e2.c
// Dennis Ritchie, 29/02/2013
```

Exemple type d'un programme C 1/4

Première partie identification du programme :

```
// TP747e2.c
// Dennis Ritchie, 29/02/2013
```

Exemple type d'un programme C 2/4

Deuxième partie directives, nouveaux types, variables et prototypes globaux :

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

const int N = 25;
typedef int TAB[N];
```

Exemple type d'un programme C 2/4

Deuxième partie directives, nouveaux types, variables et prototypes globaux :

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

const int N = 25;
typedef int TAB[N];
```

Exemple type d'un programme C 3/4

Troisième partie programme principal :

```
int main(void)
{
    int iMin, tmp;
    TAB t;
    int IndiceDuMin(TAB, // tableau d'entiers
                    int); // taille du tableau
  srand(time(NULL));
  for(int i = 0; i < N; i++)
  {
    t[i] = rand() % N;
  }
  iMin = IndiceDuMin(t, N); // appel de fonction
  cout << t[0] = " << t[0] << "\t t[iMin] = " << t[iMin];
  tmp = t[0]; t[0] = t[iMin]; t[iMin] = tmp;
  cout << "\nt[0] = " << t[0] << "\t t[iMin] = " << t[iMin];
  system("PAUSE");
  return 0;
}
```

Exemple type d'un programme C 3/4

Troisième partie programme principal :

```
int main(void)
{
    int iMin, tmp;
    TAB t;
    int IndiceDuMin(TAB, // tableau d'entiers
                    int); // taille du tableau
  srand(time(NULL));
  for(int i = 0; i < N; i++)
  {
    t[i] = rand() % N;
  }
  iMin = IndiceDuMin(t, N); // appel de fonction
  cout << t[0] = " << t[0] << "\t t[iMin] = " << t[iMin];
  tmp = t[0]; t[0] = t[iMin]; t[iMin] = tmp;
  cout << "\nt[0] = " << t[0] << "\t t[iMin] = " << t[iMin];
  system("PAUSE");
  return 0;
}
```

Exemple type d'un programme C 4/4

Quatrième partie définitions des fonctions :

```
int IndiceDuMin(TAB t, int n)
{
    int u = t[0], iMin = 0;

  for(int i = 1; i < n; i = i+1)
  {
    if(t[i] < u)
    {
      u = t[i];
      iMin = i;
    }
  }
  return iMin;
}
```

Exemple type d'un programme C 4/4

Quatrième partie définitions des fonctions :

```
int IndiceDuMin(TAB t, int n)
{
    int u = t[0], iMin = 0;

  for(int i = 1; i < n; i = i+1)
  {
    if(t[i] < u)
    {
      u = t[i];
      iMin = i;
    }
  }
  return iMin;
}
```