

Principes des langages de programmation: le cas du C

François Yvon

Faculté des Sciences de l'Université de Paris XI
Département d'Informatique

26 octobre 2007

Le(s) thème(s) du jour: Les bases du langage

Généralités

- ▶ 8 leçons (F. Yvon), 14 TPs (T. Tang)
- ▶ Évaluation :
 - ▶ micro-projets (TT)
 - ▶ contrôle (FY)
- ▶ Objectifs :
 - ▶ connaissance du C [utile pour C++, C#...]
 - ▶ compréhension des concepts de la programmation impérative
 - ▶ introduction aux outils et techniques de développement (analyse, compilation séparée, tests...)

Généralités

- ▶ 8 leçons (F. Yvon), 14 TPs (T. Tang)
- ▶ Évaluation :
 - ▶ micro-projets (TT)
 - ▶ contrôle (FY)
- ▶ Objectifs :
 - ▶ connaissance du C [utile pour C++, C#...]
 - ▶ compréhension des concepts de la programmation impérative
 - ▶ introduction aux outils et techniques de développement (analyse, compilation séparée, tests...)

Généralités

- ▶ 8 leçons (F. Yvon), 14 TPs (T. Tang)
- ▶ Évaluation :
 - ▶ micro-projets (TT)
 - ▶ contrôle (FY)
- ▶ Objectifs :
 - ▶ connaissance du C [utile pour C++, C#...]
 - ▶ compréhension des concepts de la programmation impérative
 - ▶ introduction aux outils et techniques de développement (analyse, compilation séparée, tests...)

Généralités

- ▶ 8 leçons (F. Yvon), 14 TPs (T. Tang)
- ▶ Évaluation :
 - ▶ micro-projets (TT)
 - ▶ contrôle (FY)
- ▶ Objectifs :
 - ▶ connaissance du C [utile pour C++, C#...]
 - ▶ compréhension des concepts de la programmation impérative
 - ▶ introduction aux outils et techniques de développement (analyse, compilation séparée, tests...)

Généralités

- ▶ 8 leçons (F. Yvon), 14 TPs (T. Tang)
- ▶ Évaluation :
 - ▶ micro-projets (TT)
 - ▶ contrôle (FY)
- ▶ Objectifs :
 - ▶ connaissance du C [utile pour C++, C#...]
 - ▶ compréhension des concepts de la programmation impérative
 - ▶ introduction aux outils et techniques de développement (analyse, compilation séparée, tests...)

Généralités

- ▶ 8 leçons (F. Yvon), 14 TPs (T. Tang)
- ▶ Évaluation :
 - ▶ micro-projets (TT)
 - ▶ contrôle (FY)
- ▶ Objectifs :
 - ▶ connaissance du C [utile pour C++, C#...]
 - ▶ compréhension des concepts de la programmation impérative
 - ▶ introduction aux outils et techniques de développement (analyse, compilation séparée, tests...)

Généralités

- ▶ 8 leçons (F. Yvon), 14 TPs (T. Tang)
- ▶ Évaluation :
 - ▶ micro-projets (TT)
 - ▶ contrôle (FY)
- ▶ Objectifs :
 - ▶ connaissance du C [utile pour C++, C#...]
 - ▶ compréhension des concepts de la programmation impérative
 - ▶ introduction aux outils et techniques de développement (analyse, compilation séparée, tests...)

Généralités

- ▶ 8 leçons (F. Yvon), 14 TPs (T. Tang)
- ▶ Évaluation :
 - ▶ micro-projets (TT)
 - ▶ contrôle (FY)
- ▶ Objectifs :
 - ▶ connaissance du C [utile pour C++, C#...]
 - ▶ compréhension des concepts de la programmation impérative
 - ▶ introduction aux outils et techniques de développement (analyse, compilation séparée, tests...)

Anatomie d'un programme

```
1      /* Auteur: F. Yvon */
2      /* Imprime 'Hello world' sur la sortie standard */
3
4      #include<stdio.h>
5
6      int main() {
7          printf("%s\n", "Hello, world");
8      }
```

- ▶ lignes 1, 2 : /* F. Yvon */ : lignes de *commentaire*
- ▶ lignes 3, 5 : les lignes vides sont ignorées (comme les espaces et les tabulations)
- ▶ ligne 4 : #include<stdio.h> : utilisation de la bibliothèque I/O
- ▶ ligne 6 : int main() { : int est un *mot-clé* du langage ; main annonce le programme *principal*
- ▶ ligne 7 : printf ("%s\n", "Hello, world"); : commande d'impression via la fonction standard printf
- ▶ ligne 8 : } : ferme l'accolade ligne 5.

Anatomie d'un programme

```
1      /* Auteur: F. Yvon */
2      /* Imprime 'Hello world' sur la sortie standard */
3
4      #include<stdio.h>
5
6      int main() {
7          printf("%s\n", "Hello, world");
8      }
```

- ▶ lignes 1, 2 : `/* F. Yvon */` : lignes de *commentaire*
- ▶ lignes 3, 5 : les lignes vides sont ignorées (comme les espaces et les tabulations)
- ▶ ligne 4 : `#include<stdio.h>` : utilisation de la bibliothèque I/O
- ▶ ligne 6 : `int main()` : `int` est un *mot-clé* du langage ; `main` annonce le programme *principal*
- ▶ ligne 7 : `printf ("%s\n", "Hello, world");` : commande d'impression via la fonction standard `printf`
- ▶ ligne 8 : `}` : ferme l'accolade ligne 5.

Anatomie d'un programme

```
1  /* Auteur: F. Yvon */
2  /* Imprime 'Hello world' sur la sortie standard */
3
4  #include<stdio.h>
5
6  int main() {
7      printf("%s\n", "Hello, world");
8 }
```

- ▶ lignes 1, 2 : */* F. Yvon */* : lignes de *commentaire*
- ▶ lignes 3, 5 : les lignes vides sont ignorées (comme les espaces et les tabulations)
- ▶ ligne 4 : **#include<stdio.h>** : utilisation de la bibliothèque I/O
- ▶ ligne 6 : **int main()** { : int est un *mot-clé* du langage ; **main** annonce le programme *principal*
- ▶ ligne 7 : **printf ("%s\n", "Hello, world");** : commande d'impression via la fonction standard **printf**
- ▶ ligne 8 : } : ferme l'accolade ligne 5.

Anatomie d'un programme

```
1      /* Auteur: F. Yvon */
2      /* Imprime 'Hello world' sur la sortie standard */
3
4      #include<stdio.h>
5
6      int main() {
7          printf("%s\n", "Hello, world");
8      }
```

- ▶ lignes 1, 2 : */* F. Yvon */* : lignes de *commentaire*
- ▶ lignes 3, 5 : les lignes vides sont ignorées (comme les espaces et les tabulations)
- ▶ ligne 4 : **#include**<stdio.h> : utilisation de la bibliothèque I/O
- ▶ ligne 6 : **int** main() { : **int** est un *mot-clé* du langage ; **main** annonce le programme *principal*
- ▶ ligne 7 : printf ("%s\n", "Hello, world"); : commande d'impression via la fonction standard printf
- ▶ ligne 8 : } : ferme l'accolade ligne 5.

Anatomie d'un programme

```
1      /* Auteur: F. Yvon */
2      /* Imprime 'Hello world' sur la sortie standard */
3
4      #include<stdio.h>
5
6      int main() {
7          printf("%s\n", "Hello, world");
8      }
```

- ▶ lignes 1, 2 : */* F. Yvon */* : lignes de *commentaire*
- ▶ lignes 3, 5 : les lignes vides sont ignorées (comme les espaces et les tabulations)
- ▶ ligne 4 : **#include**<stdio.h> : utilisation de la bibliothèque I/O
- ▶ ligne 6 : **int** main() { : **int** est un *mot-clé* du langage ; **main** annonce le programme *principal*
- ▶ ligne 7 : printf ("%s\n", "Hello, world"); : **commande** d'impression via la fonction standard **printf**
- ▶ ligne 8 : } : ferme l'accolade ligne 5.

Anatomie d'un programme

```
1      /* Auteur: F. Yvon */
2      /* Imprime 'Hello world' sur la sortie standard */
3
4      #include<stdio.h>
5
6      int main() {
7          printf("%s\n", "Hello, world");
8      }
```

- ▶ lignes 1, 2 : */* F. Yvon */* : lignes de *commentaire*
- ▶ lignes 3, 5 : les lignes vides sont ignorées (comme les espaces et les tabulations)
- ▶ ligne 4 : **#include**<stdio.h> : utilisation de la bibliothèque I/O
- ▶ ligne 6 : **int** main() { : **int** est un *mot-clé* du langage ; **main** annonce le programme *principal*
- ▶ ligne 7 : printf ("%s\n", "Hello, world"); : commande d'impression via la fonction standard **printf**
- ▶ ligne 8 : } : ferme l'accolade ligne 5.

Anatomie d'un autre programme

```
1 // Auteur: F. Yvon
2 #include<stdio.h>
3 #include<stdlib.h>
4 int main() {
5     int i = 0, j = 0;
6     scanf("%i", &i);
7     j = i*i;
8     printf("i*i = %i\n", j);
9     return(EXIT_SUCCESS);
10 }
```

- ▶ lignes 1 : // F. Yvon : une autre forme de *commentaire*
- ▶ ligne 5 : int i = 0, j = 0; *définition* de deux variables, valant initialement à 0 ;
- ▶ ligne 6 : scanf("%i", &i) : scanf lit la valeur de i
- ▶ ligne 7 : j = i**2; : calcule i^2 , affecte le résultat dans la variable j
- ▶ ligne 9 : **return** est un mot clé du langage ; EXIT_SUCCESS est une *constante* prédéfinie dans stdlib

Anatomie d'un autre programme

```
1 // Auteur: F. Yvon
2 #include<stdio.h>
3 #include<stdlib.h>
4 int main() {
5     int i = 0, j = 0;
6     scanf("%i", &i);
7     j = i*i;
8     printf("i*i = %i\n", j);
9     return(EXIT_SUCCESS);
10 }
```

- ▶ lignes 1 : `// F. Yvon` : une autre forme de *commentaire*
- ▶ ligne 5 : `int i = 0, j = 0;` *définition de deux variables*, valant initialement à 0 ;
- ▶ ligne 6 : `scanf("%i", &i)` : `scanf` lit la valeur de `i`
- ▶ ligne 7 : `j = i**2;` calcule i^2 , affecte le résultat dans la variable `j`
- ▶ ligne 9 : `return` est un mot clé du langage ; `EXIT_SUCCESS` est une *constante* prédéfinie dans `stdlib`

Anatomie d'un autre programme

```
1 // Auteur: F. Yvon
2 #include<stdio.h>
3 #include<stdlib.h>
4 int main() {
5     int i = 0, j = 0;
6     scanf("%i", &i);
7     j = i*i;
8     printf("i*i = %i\n", j);
9     return(EXIT_SUCCESS);
10 }
```

- ▶ lignes 1 : `// F. Yvon` : une autre forme de *commentaire*
- ▶ ligne 5 : `int i = 0, j = 0;` *définition* de deux variables, valant initialement à 0 ;
- ▶ ligne 6 : `scanf("%i", &i)` : `scanf` lit la valeur de `i`
- ▶ ligne 7 : `j = i**2;` calcule i^2 , affecte le résultat dans la variable `j`
- ▶ ligne 9 : `return` est un mot clé du langage ; `EXIT_SUCCESS` est une *constante* prédéfinie dans `stdlib`

Anatomie d'un autre programme

```
1 // Auteur: F. Yvon
2 #include<stdio.h>
3 #include<stdlib.h>
4 int main() {
5     int i = 0, j = 0;
6     scanf("%i", &i);
7     j = i*i;
8     printf("i*i = %i\n", j);
9     return(EXIT_SUCCESS);
10 }
```

- ▶ lignes 1 : `// F. Yvon` : une autre forme de *commentaire*
- ▶ ligne 5 : `int i = 0, j = 0;` *définition* de deux variables, valant initialement à 0 ;
- ▶ ligne 6 : `scanf("%i", &i)` : `scanf` lit la valeur de `i`
- ▶ ligne 7 : `j = i**2;` : calcule i^2 , affecte le résultat dans la variable `j`
- ▶ ligne 9 : `return` est un mot clé du langage ; `EXIT_SUCCESS` est une *constante* prédéfinie dans `stdlib`

Anatomie d'un autre programme

```
1 // Auteur: F. Yvon
2 #include<stdio.h>
3 #include<stdlib.h>
4 int main() {
5     int i = 0, j = 0;
6     scanf("%i", &i);
7     j = i*i;
8     printf("i*i = %i\n", j);
9     return(EXIT_SUCCESS);
10 }
```

- ▶ lignes 1 : `// F. Yvon` : une autre forme de *commentaire*
- ▶ ligne 5 : `int i = 0, j = 0;` *définition* de deux variables, valant initialement à 0 ;
- ▶ ligne 6 : `scanf("%i", &i)` : `scanf` lit la valeur de `i`
- ▶ ligne 7 : `j = i**2;` : calcule i^2 , affecte le résultat dans la variable `j`
- ▶ ligne 9 : `return` est un mot clé du langage ; `EXIT_SUCCESS` est une *constante* prédéfinie dans `stdlib`

Un programme : kesako ?

Réponse « impérative »

Liste d'opérations à effectuer par le processeur en interaction avec les autres composants (mémoire(s), périphériques, réseau).

- ▶ respecte une *syntaxe*
- ▶ possède une *sémantique*
- ▶ est l'objet d'une série de traductions (par ex. compilation)

- ▶ Des lignes de commentaires
- ▶ Des *directives* de pré-compilation
- ▶ Des définitions ou des déclarations de variables ou de fonctions
- ▶ Des *instructions* simples
- ▶ Des structures de contrôle organisant des instructions complexes

Un programme : kesako ?

Réponse « impérative »

Liste d'opérations à effectuer par le processeur en interaction avec les autres composants (mémoire(s), périphériques, réseau).

- ▶ respecte une *syntaxe*
- ▶ possède une *sémantique*
- ▶ est l'objet d'une série de traductions (par ex. compilation)

- ▶ Des lignes de commentaires
- ▶ Des *directives* de pré-compilation
- ▶ Des définitions ou des déclarations de variables ou de fonctions
- ▶ Des *instructions* simples
- ▶ Des structures de contrôle organisant des instructions complexes

Un programme : kesako ?

Réponse « impérative »

Liste d'opérations à effectuer par le processeur en interaction avec les autres composants (mémoire(s), périphériques, réseau).

- ▶ respecte une *syntaxe*
- ▶ possède une *sémantique*
- ▶ est l'objet d'une série de traductions (par ex. compilation)
 - ▶ Des lignes de commentaires
 - ▶ Des *directives* de pré-compilation
 - ▶ Des définitions ou des déclarations de variables ou de fonctions
 - ▶ Des *instructions* simples
 - ▶ Des structures de contrôle organisant des instructions complexes

Un programme : kesako ?

Réponse « impérative »

Liste d'opérations à effectuer par le processeur en interaction avec les autres composants (mémoire(s), périphériques, réseau).

- ▶ respecte une *syntaxe*
- ▶ possède une *sémantique*
- ▶ est l'objet d'une série de traductions (par ex. compilation)
 - ▶ Des lignes de commentaires
 - ▶ Des *directives* de pré-compilation
 - ▶ Des définitions ou des déclarations de variables ou de fonctions
 - ▶ Des *instructions* simples
 - ▶ Des structures de contrôle organisant des instructions complexes

Un programme : kesako ?

Réponse « impérative »

Liste d'opérations à effectuer par le processeur en interaction avec les autres composants (mémoire(s), périphériques, réseau).

- ▶ respecte une *syntaxe*
- ▶ possède une *sémantique*
- ▶ est l'objet d'une série de traductions (par ex. compilation)
 - ▶ Des lignes de commentaires
 - ▶ Des *directives* de pré-compilation
 - ▶ Des définitions ou des déclarations de variables ou de fonctions
 - ▶ Des *instructions* simples
 - ▶ Des structures de contrôle organisant des instructions complexes

Un programme : kesako ?

Réponse « impérative »

Liste d'opérations à effectuer par le processeur en interaction avec les autres composants (mémoire(s), périphériques, réseau).

- ▶ respecte une *syntaxe*
- ▶ possède une *sémantique*
- ▶ est l'objet d'une série de traductions (par ex. compilation)
 - ▶ Des lignes de commentaires
 - ▶ Des *directives* de pré-compilation
 - ▶ Des définitions ou des déclarations de variables ou de fonctions
 - ▶ Des *instructions* simples
 - ▶ Des structures de contrôle organisant des instructions complexes

Un programme : kesako ?

Réponse « impérative »

Liste d'opérations à effectuer par le processeur en interaction avec les autres composants (mémoire(s), périphériques, réseau).

- ▶ respecte une *syntaxe*
- ▶ possède une *sémantique*
- ▶ est l'objet d'une série de traductions (par ex. compilation)
- ▶ Des lignes de commentaires
 - ▶ Des *directives* de pré-compilation
 - ▶ Des définitions ou des déclarations de variables ou de fonctions
 - ▶ Des *instructions* simples
 - ▶ Des structures de contrôle organisant des instructions complexes

Un programme : kesako ?

Réponse « impérative »

Liste d'opérations à effectuer par le processeur en interaction avec les autres composants (mémoire(s), périphériques, réseau).

- ▶ respecte une *syntaxe*
- ▶ possède une *sémantique*
- ▶ est l'objet d'une série de traductions (par ex. compilation)
- ▶ Des lignes de commentaires
- ▶ Des *directives* de pré-compilation
- ▶ Des définitions ou des déclarations de variables ou de fonctions
- ▶ Des *instructions* simples
- ▶ Des structures de contrôle organisant des instructions complexes

Un programme : kesako ?

Réponse « impérative »

Liste d'opérations à effectuer par le processeur en interaction avec les autres composants (mémoire(s), périphériques, réseau).

- ▶ respecte une *syntaxe*
- ▶ possède une *sémantique*
- ▶ est l'objet d'une série de traductions (par ex. compilation)
- ▶ Des lignes de commentaires
- ▶ Des *directives* de pré-compilation
- ▶ Des définitions ou des déclarations de variables ou de fonctions
- ▶ Des *instructions* simples
- ▶ Des structures de contrôle organisant des instructions complexes

Un programme : kesako ?

Réponse « impérative »

Liste d'opérations à effectuer par le processeur en interaction avec les autres composants (mémoire(s), périphériques, réseau).

- ▶ respecte une *syntaxe*
- ▶ possède une *sémantique*
- ▶ est l'objet d'une série de traductions (par ex. compilation)
- ▶ Des lignes de commentaires
- ▶ Des *directives* de pré-compilation
- ▶ Des définitions ou des déclarations de variables ou de fonctions
- ▶ Des *instructions* simples
- ▶ Des structures de contrôle organisant des instructions complexes

Un programme : kesako ?

Réponse « impérative »

Liste d'opérations à effectuer par le processeur en interaction avec les autres composants (mémoire(s), périphériques, réseau).

- ▶ respecte une *syntaxe*
- ▶ possède une *sémantique*
- ▶ est l'objet d'une série de traductions (par ex. compilation)
- ▶ Des lignes de commentaires
- ▶ Des *directives* de pré-compilation
- ▶ Des définitions ou des déclarations de variables ou de fonctions
- ▶ Des *instructions* simples
- ▶ Des structures de contrôle organisant des instructions complexes

Programmation : une question de formes

- ▶ forme des identificateurs (de variables et de fonctions) :
 - ▶ [a – ZA – Z][a – ZA – Z]*
 - ▶ sauf : auto, break, case, char, const, continue... qui sont des *mots-clés* du langage
 - ▶ lisibilité et expressivité : int i ; moins bon que int nelements;
- ▶ forme des constantes ⇒ dépend de leur type :
 - ▶ 'a' pour des caractères
 - ▶ "une chaîne" pour les chaînes de caractères
 - ▶ 12, 0x0F, 0465 pour les entiers
 - ▶ -12.67, 1.2e-6 pour les réels
- ▶ blancs, tabulations, retour à ligne sont ignorés : des atouts pour rendre le code lisible

Programmation : une question de formes

- ▶ forme des identificateurs (de variables et de fonctions) :
 - ▶ [a – ZA – Z_][a – ZA – Z_]*
 - ▶ sauf : auto, break, case, char, const, continue... qui sont des *mots-clés* du langage
 - ▶ lisibilité et expressivité : int i ; moins bon que int nelements;
- ▶ forme des constantes ⇒ dépend de leur type :
 - ▶ 'a' pour des caractères
 - ▶ "une chaîne" pour les chaînes de caractères
 - ▶ 12, 0x0F, 0465 pour les entiers
 - ▶ -12.67, 1.2e-6 pour les réels
- ▶ blancs, tabulations, retour à ligne sont ignorés : des atouts pour rendre le code lisible

Programmation : une question de formes

- ▶ forme des identificateurs (de variables et de fonctions) :
 - ▶ [a – ZA – Z_][a – ZA – Z_]*
 - ▶ sauf : **auto**, **break**, **case**, **char**, **const**, **continue**... qui sont des *mots-clés* du langage
 - ▶ lisibilité et expressivité : `int i`; moins bon que `int nelements`;
- ▶ forme des constantes ⇒ dépend de leur type :
 - ▶ 'a' pour des caractères
 - ▶ "une chaîne" pour les chaînes de caractères
 - ▶ 12, 0x0F, 0465 pour les entiers
 - ▶ -12.67, 1.2e-6 pour les réels
- ▶ blancs, tabulations, retour à ligne sont ignorés : des atouts pour rendre le code lisible

Programmation : une question de formes

- ▶ forme des identificateurs (de variables et de fonctions) :
 - ▶ [a – ZA – Z_][a – ZA – Z_]*
 - ▶ sauf : **auto**, **break**, **case**, **char**, **const**, **continue**... qui sont des *mots-clés* du langage
 - ▶ lisibilité et expressivité : **int** i ; moins bon que **int** nelements;
- ▶ forme des constantes ⇒ dépend de leur type :
 - ▶ 'a' pour des caractères
 - ▶ "une chaîne" pour les chaînes de caractères
 - ▶ 12, 0x0F, 0465 pour les entiers
 - ▶ -12.67, 1.2e-6 pour les réels
- ▶ blancs, tabulations, retour à ligne sont ignorés : des atouts pour rendre le code lisible

Programmation : une question de formes

- ▶ forme des identificateurs (de variables et de fonctions) :
 - ▶ [a – ZA – Z_][a – ZA – Z_]*
 - ▶ sauf : **auto**, **break**, **case**, **char**, **const**, **continue**... qui sont des *mots-clés* du langage
 - ▶ lisibilité et expressivité : **int** i ; moins bon que **int** nelements;
- ▶ forme des constantes ⇒ dépend de leur type :
 - ▶ 'a' pour des caractères
 - ▶ "une chaîne" pour les chaînes de caractères
 - ▶ 12, 0x0F, 0465 pour les entiers
 - ▶ -12.67, 1.2e-6 pour les réels
- ▶ blancs, tabulations, retour à ligne sont ignorés : des atouts pour rendre le code lisible

Programmation : une question de formes

- ▶ forme des identificateurs (de variables et de fonctions) :
 - ▶ [a – ZA – Z_][a – ZA – Z_]*
 - ▶ sauf : **auto**, **break**, **case**, **char**, **const**, **continue**... qui sont des *mots-clés* du langage
 - ▶ lisibilité et expressivité : **int** i ; moins bon que **int** nelements;
- ▶ forme des constantes ⇒ dépend de leur type :
 - ▶ 'a' pour des caractères
 - ▶ "une chaîne" pour les chaînes de caractères
 - ▶ 12, 0x0F, 0465 pour les entiers
 - ▶ -12.67, 1.2e-6 pour les réels
- ▶ blancs, tabulations, retour à ligne sont ignorés : des atouts pour rendre le code lisible

Programmation : une question de formes

- ▶ forme des identificateurs (de variables et de fonctions) :
 - ▶ [a – ZA – Z_][a – ZA – Z_]*
 - ▶ sauf : **auto**, **break**, **case**, **char**, **const**, **continue**... qui sont des *mots-clés* du langage
 - ▶ lisibilité et expressivité : **int** i ; moins bon que **int** nelements;
- ▶ forme des constantes ⇒ dépend de leur type :
 - ▶ 'a' pour des caractères
 - ▶ "une chaîne" pour les chaînes de caractères
 - ▶ 12, 0x0F, 0465 pour les entiers
 - ▶ -12.67, 1.2e-6 pour les réels
- ▶ blancs, tabulations, retour à ligne sont ignorés : des atouts pour rendre le code lisible

Programmation : une question de formes

- ▶ forme des identificateurs (de variables et de fonctions) :
 - ▶ [a – ZA – Z_][a – ZA – Z_]*
 - ▶ sauf : **auto**, **break**, **case**, **char**, **const**, **continue**... qui sont des *mots-clés* du langage
 - ▶ lisibilité et expressivité : **int** i ; moins bon que **int** nelements;
- ▶ forme des constantes ⇒ dépend de leur type :
 - ▶ 'a' pour des caractères
 - ▶ "une chaîne" pour les chaînes de caractères
 - ▶ 12, 0x0F, 0465 pour les entiers
 - ▶ -12.67, 1.2e-6 pour les réels
- ▶ blancs, tabulations, retour à ligne sont ignorés : des atouts pour rendre le code lisible

Programmation : une question de formes

- ▶ forme des identificateurs (de variables et de fonctions) :
 - ▶ [a – ZA – Z_][a – ZA – Z_]*
 - ▶ sauf : **auto**, **break**, **case**, **char**, **const**, **continue**... qui sont des *mots-clés* du langage
 - ▶ lisibilité et expressivité : **int** i ; moins bon que **int** nelements;
- ▶ forme des constantes ⇒ dépend de leur type :
 - ▶ 'a' pour des caractères
 - ▶ "une chaîne" pour les chaînes de caractères
 - ▶ 12, 0x0F, 0465 pour les entiers
 - ▶ -12.67, 1.2e-6 pour les réels
- ▶ blancs, tabulations, retour à ligne sont ignorés : des atouts pour rendre le code lisible

Programmation : une question de formes

- ▶ forme des identificateurs (de variables et de fonctions) :
 - ▶ $[a - ZA - Z_][a - ZA - Z_]^*$
 - ▶ sauf : **auto**, **break**, **case**, **char**, **const**, **continue**... qui sont des *mots-clés* du langage
 - ▶ lisibilité et expressivité : **int i**; moins bon que **int nelements**;
- ▶ forme des constantes \Rightarrow dépend de leur type :
 - ▶ 'a' pour des caractères
 - ▶ "une chaîne" pour les chaînes de caractères
 - ▶ 12, 0x0F, 0465 pour les entiers
 - ▶ -12.67, 1.2e-6 pour les réels
- ▶ blancs, tabulations, retour à ligne sont ignorés : des atouts pour rendre le code lisible

Un programme sans commentaire est comme ???

```
/* Un commentaire  
    peut durer  
    plusieurs lignes  
*/  
/* Attention aux ouvertures  
int i = 0;  
/* et fermetures ... */
```

- ▶ tout ce qui apparaît entre /* et le */ suivant
- ▶ tout ce qui apparaît entre // et la fin de la ligne courante
- ▶ les commentaires sont ignorés par la machine
- ▶ fonction de documentation *administrative* (qui, quand, pourquoi)
- ▶ fonction de documentation *technique*, essentielle pour la maintenance

Un programme sans commentaire est comme ???

```
/* Un commentaire  
    peut durer  
    plusieurs lignes  
*/  
/* Attention aux ouvertures  
int i = 0;  
/* et fermetures ... */
```

- ▶ tout ce qui apparaît entre /* et le */ suivant
- ▶ tout ce qui apparaît entre // et la fin de la ligne courante
- ▶ les commentaires sont ignorés par la machine
- ▶ fonction de documentation *administrative* (qui, quand, pourquoi)
- ▶ fonction de documentation *technique*, essentielle pour la maintenance

Un programme sans commentaire est comme ???

```
/* Un commentaire  
    peut durer  
    plusieurs lignes  
*/  
/* Attention aux ouvertures  
int i = 0;  
/* et fermetures ... */
```

- ▶ tout ce qui apparaît entre /* et le */ suivant
- ▶ tout ce qui apparaît entre // et la fin de la ligne courante
- ▶ **les commentaires sont ignorés par la machine**
- ▶ fonction de documentation *administrative* (qui, quand, pourquoi)
- ▶ fonction de documentation *technique*, essentielle pour la maintenance

Un programme sans commentaire est comme ???

```
/* Un commentaire  
    peut durer  
    plusieurs lignes  
*/  
/* Attention aux ouvertures  
int i = 0;  
/* et fermetures... */
```

- ▶ tout ce qui apparaît entre `/*` et le `*/` suivant
- ▶ tout ce qui apparaît entre `//` et la fin de la ligne courante
- ▶ les commentaires sont ignorés par la machine
- ▶ fonction de documentation *administrative* (qui, quand, pourquoi)
- ▶ fonction de documentation *technique*, essentielle pour la maintenance

Un programme sans commentaire est comme ???

```
/* Un commentaire  
    peut durer  
    plusieurs lignes  
*/  
/* Attention aux ouvertures  
int i = 0;  
/* et fermetures... */
```

- ▶ tout ce qui apparaît entre `/*` et le `*/` suivant
- ▶ tout ce qui apparaît entre `//` et la fin de la ligne courante
- ▶ les commentaires sont ignorés par la machine
- ▶ fonction de documentation *administrative* (qui, quand, pourquoi)
- ▶ fonction de documentation *technique*, essentielle pour la maintenance

Les instructions élémentaires

```
i = i + 2;  
j = sqrt(16) + 3;  
printf("%i\n", i);  
k = i * i, l = i + i;
```

- ▶ var op expr ;
 - ▶ variable est un *identificateur* valide
([a-zA-Z_][a-zA-Z0-9]*)
 - ▶ op est un *opérateur* := (affectation), == (test) ...
 - ▶ expr est une *expression*
- ▶ un appel de procédure ou de fonction
- ▶ une séquence
- ▶ toute instruction possède un *type* et une *valeur*
 - ▶ pour i = i + 2 le type est celui de i, la valeur celle de i (après l)
 - ▶ pour printf () le type est int, la valeur le nombre de caractères imprimés
 - ▶ pour une séquence, le type et la valeur sont celles de la dernière instruction

Les instructions élémentaires

```
i = i + 2;  
j = sqrt(16) + 3;  
printf("%i\n", i);  
k = i * i, l = i + i;
```

- ▶ var op expr ;
 - ▶ variable est un *identificateur valide*
([a-zA-Z_][a-zA-Z0-9])*
 - ▶ op est un *opérateur* := (affectation), == (test) ...
 - ▶ expr est une *expression*
- ▶ un appel de procédure ou de fonction
- ▶ une séquence
- ▶ toute instruction possède un *type* et une *valeur*
 - ▶ pour `i = i + 2` le type est celui de `i`, la valeur celle de `i` (après l=)
 - ▶ pour `printf()` le type est `int`, la valeur le nombre de caractères imprimés
 - ▶ pour une séquence, le type et la valeur sont celles de la dernière instruction

Les instructions élémentaires

```
i = i + 2;  
j = sqrt(16) + 3;  
printf("%i\n", i);  
k = i * i, l = i + i;
```

- ▶ var op expr ;
 - ▶ variable est un *identificateur* valide
([a-zA-Z_][a-zA-Z0-9]*)
 - ▶ op est un *opérateur* := (affectation), == (test) ...
 - ▶ expr est une *expression*
- ▶ un appel de procédure ou de fonction
- ▶ une séquence
- ▶ toute instruction possède un *type* et une *valeur*
 - ▶ pour i = i + 2 le type est celui de i, la valeur celle de i (après l)
 - ▶ pour printf () le type est int, la valeur le nombre de caractères imprimés
 - ▶ pour une séquence, le type et la valeur sont celles de la dernière instruction

Les instructions élémentaires

```
i = i + 2;  
j = sqrt(16) + 3;  
printf("%i\n", i);  
k = i * i, l = i + i;
```

- ▶ var op expr ;
 - ▶ variable est un *identificateur* valide
([a-zA-Z_][a-zA-Z0-9]*)
 - ▶ op est un *opérateur* := (affectation), == (test) ...
 - ▶ expr est une *expression*
- ▶ un appel de procédure ou de fonction
- ▶ une séquence
- ▶ toute instruction possède un *type* et une *valeur*
 - ▶ pour i = i + 2 le type est celui de i, la valeur celle de i (après l)
 - ▶ pour printf () le type est int, la valeur le nombre de caractères imprimés
 - ▶ pour une séquence, le type et la valeur sont celles de la dernière instruction

Les instructions élémentaires

```
i = i + 2;  
j = sqrt(16) + 3;  
printf("%i\n", i);  
k = i * i, l = i + i;
```

- ▶ var op expr ;
 - ▶ variable est un *identificateur* valide
([a-zA-Z_][a-zA-Z0-9]*)
 - ▶ op est un *opérateur* := (affectation), == (test) ...
 - ▶ expr est une *expression*
- ▶ un appel de procédure ou de fonction
- ▶ une séquence
- ▶ toute instruction possède un *type* et une *valeur*
 - ▶ pour i = i + 2 le type est celui de i, la valeur celle de i (après l)
 - ▶ pour printf () le type est int, la valeur le nombre de caractères imprimés
 - ▶ pour une séquence, le type et la valeur sont celles de la dernière instruction

Les instructions élémentaires

```
i = i + 2;  
j = sqrt(16) + 3;  
printf("%i\n", i);  
k = i * i, l = i + i;
```

- ▶ var op expr ;
 - ▶ variable est un *identificateur* valide
([a-zA-Z_][a-zA-Z0-9]*)
 - ▶ op est un *opérateur* := (affectation), == (test) ...
 - ▶ expr est une *expression*
- ▶ un appel de procédure ou de fonction
- ▶ une séquence
- ▶ toute instruction possède un *type* et une *valeur*
 - ▶ pour i = i + 2 le type est celui de i, la valeur celle de i (après l)
 - ▶ pour printf () le type est int, la valeur le nombre de caractères imprimés
 - ▶ pour une séquence, le type et la valeur sont celles de la dernière instruction

Les instructions élémentaires

```
i = i + 2;  
j = sqrt(16) + 3;  
printf("%i\n", i);  
k = i * i, l = i + i;
```

- ▶ var op expr ;
 - ▶ variable est un *identificateur* valide
([a-zA-Z_][a-zA-Z0-9]*)
 - ▶ op est un *opérateur* := (affectation), == (test) ...
 - ▶ expr est une *expression*
- ▶ un appel de procédure ou de fonction
- ▶ une séquence
- ▶ toute instruction possède un *type* et une *valeur*
 - ▶ pour i = i + 2 le type est celui de i, la valeur celle de i (après !)
 - ▶ pour printf () le type est int, la valeur le nombre de caractères imprimés
 - ▶ pour une séquence, le type et la valeur sont celles de la dernière instruction

Les instructions élémentaires

```
i = i + 2;  
j = sqrt(16) + 3;  
printf("%i\n", i);  
k = i * i, l = i + i;
```

- ▶ var op expr ;
 - ▶ variable est un *identificateur* valide
([a-zA-Z_][a-zA-Z0-9]*)
 - ▶ op est un *opérateur* := (affectation), == (test) ...
 - ▶ expr est une *expression*
- ▶ un appel de procédure ou de fonction
- ▶ une séquence
- ▶ toute instruction possède un *type* et une *valeur*
 - ▶ pour i = i + 2 le type est celui de i, la valeur celle de i (après !)
 - ▶ pour printf () le type est int, la valeur le nombre de caractères imprimés
 - ▶ pour une séquence, le type et la valeur sont celles de la dernière instruction

Les instructions élémentaires

```
i = i + 2;  
j = sqrt(16) + 3;  
printf("%i\n", i);  
k = i * i, l = i + i;
```

- ▶ var op expr ;
 - ▶ variable est un *identificateur* valide
([a-zA-Z_][a-zA-Z0-9]*)
 - ▶ op est un *opérateur* := (affectation), == (test) ...
 - ▶ expr est une *expression*
- ▶ un appel de procédure ou de fonction
- ▶ une séquence
- ▶ toute instruction possède un *type* et une *valeur*
 - ▶ pour i = i + 2 le type est celui de i, la valeur celle de i (après !)
 - ▶ pour printf () le type est int, la valeur le nombre de caractères imprimés
 - ▶ pour une séquence, le type et la valeur sont celles de la dernière instruction

Les instructions élémentaires

```
i = i + 2;  
j = sqrt(16) + 3;  
printf("%i\n", i);  
k = i * i, l = i + i;
```

- ▶ var op expr ;
 - ▶ variable est un *identificateur* valide
([a-zA-Z_][a-zA-Z0-9]*)
 - ▶ op est un *opérateur* := (affectation), == (test) ...
 - ▶ expr est une *expression*
- ▶ un appel de procédure ou de fonction
- ▶ une séquence
- ▶ toute instruction possède un *type* et une *valeur*
 - ▶ pour i = i + 2 le type est celui de i, la valeur celle de i (après !)
 - ▶ pour printf () le type est int, la valeur le nombre de caractères imprimés
 - ▶ pour une séquence, le type et la valeur sont celles de la dernière instruction

Les instructions complexes

- ▶ Les opérations conditionnelles

Selon la saison, ajouter des poireaux ou des courgettes

⇒ les branchements conditionnels if, else, switch

- ▶ Des instructions répétitives

Tant que les œufs ne sont pas blancs

Battre les œufs

⇒ les structures itératives while, do, for

- ▶ Cloisonner / Structurer les instructions :

Faire une pâte feuilletée (voir p. 96)

Préparer la Béchamel (voir p. 33)

...

⇒ les procédures et les fonctions

Compilateurs et interpréteurs

- ▶ Langages compilés : 2 phases de traitement

1. traduction du programme source en un programme objet en langage machine
2. exécution du programme objet

- ▶ Interprétation : 1 phase de traitement

1. exécution du programme source, par décodage progressif des instructions

Compilateurs et interpréteurs

- ▶ Langages compilés : 2 phases de traitement
 1. traduction du programme source en un programme objet en langage machine
 2. exécution du programme objet
- ▶ Interprétation : 1 phase de traitement
 1. exécution du programme source, par décodage progressif des instructions

Compilateurs et interpréteurs

- ▶ Langages compilés : 2 phases de traitement
 1. traduction du programme source en un programme objet en langage machine
 2. exécution du programme objet
- ▶ Interprétation : 1 phase de traitement
 1. exécution du programme source, par décodage progressif des instructions

Compilateurs et interpréteurs

- ▶ Langages compilés : 2 phases de traitement
 1. traduction du programme source en un programme objet en langage machine
 2. exécution du programme objet
- ▶ Interprétation : 1 phase de traitement
 1. exécution du programme source, par décodage progressif des instructions

Compilateurs et interpréteurs

- ▶ Langages compilés : 2 phases de traitement
 1. traduction du programme source en un programme objet en langage machine
 2. exécution du programme objet
- ▶ Interprétation : 1 phase de traitement
 1. exécution du programme source, par décodage progressif des instructions

Coder, tout un programme

1. Éditer le fichier **texte** mon-prog.c (vi, emacs...)

2. Construire le fichier **exécutable** :

cc mon-prog.c -o mon-prog ou encore mieux :

cc -Wall -pedantic -ansi mon-prog.c -o mon-prog

3. La compilation réussit ?

3.1 NON \Rightarrow goto 1

```
cc      erreur.c -o erreur
erreur.c: In function 'main':
erreur.c:5: error: 'xint' undeclared (first use in this function)
erreur.c:5: error: (Each undeclared identifier is reported only once
erreur.c:5: error: for each function it appears in.)
erreur.c:5: error: parse error before 'i'
make: *** [exemples/erreur] Error 1
```

3.2 OUI \Rightarrow tester le programme ./mon-prog

L'exécution est correcte ?

3.2.1 OUI \Rightarrow repos des braves

3.2.2 NON \Rightarrow goto 1

Des sources d'informations

- ▶ Les *man pages* (sous UNIX) : man 3 **printf**
- ▶ Des sites (une petite sélection) :
 - ▶ http://www.iri.fr/~aze/page_c/aide_c/
 - ▶ http://www.linux-kheops.com/doc/ansi-c/Introduction_ANSI_C.htm
 - ▶ http://fr.wikibooks.org/wiki/Programmation_C
 - ▶ C facile www.enst.fr/~charon
- ▶ Des livres :
 - ▶ Kernighan & Ritchie : le langage C (la « bible »)
 - ▶ Varrette & N. Bernard : Programmation avancée en C, Hermès, Paris.
 - ▶ C. Delannoy : le langage C, la référence ; Programmer en langage C.

Des sources d'informations

- ▶ Les *man pages* (sous UNIX) : man 3 **printf**
- ▶ Des sites (une petite sélection) :
 - ▶ http://www.lri.fr/~aze/page_c/aide_c/
 - ▶ http://www.linux-kheops.com/doc/ansi-c/Introduction_ANSI_C.htm
 - ▶ http://fr.wikibooks.org/wiki/Programmation_C
 - ▶ C facile www.enst.fr/~charon
- ▶ Des livres :
 - ▶ Kernighan & Ritchie : le langage C (la « bible »)
 - ▶ Varrette & N. Bernard : Programmation avancée en C, Hermès, Paris.
 - ▶ C. Delannoy : le langage C, la référence ; Programmer en langage C.

Des sources d'informations

- ▶ Les *man pages* (sous UNIX) : man 3 **printf**
- ▶ Des sites (une petite sélection) :
 - ▶ http://www.lri.fr/~aze/page_c/aide_c/
 - ▶ http://www.linux-kheops.com/doc/ansi-c/Introduction_ANSI_C.htm
 - ▶ http://fr.wikibooks.org/wiki/Programmation_C
 - ▶ C facile www.enst.fr/~charon
- ▶ Des livres :
 - ▶ Kernighan & Ritchie : le langage C (la « bible »)
 - ▶ Varrette & N. Bernard : Programmation avancée en C, Hermès, Paris.
 - ▶ C. Delannoy : le langage C, la référence ; Programmer en langage C.

Des sources d'informations

- ▶ Les *man pages* (sous UNIX) : man 3 **printf**
- ▶ Des sites (une petite sélection) :
 - ▶ http://www.lri.fr/~aze/page_c/aide_c/
 - ▶ http://www.linux-kheops.com/doc/ansi-c/Introduction_ANSI_C.htm
 - ▶ http://fr.wikibooks.org/wiki/Programmation_C
 - ▶ C facile www.enst.fr/~charon
- ▶ Des livres :
 - ▶ Kernighan & Ritchie : le langage C (la « bible »)
 - ▶ Varrette & N. Bernard : Programmation avancée en C, Hermès, Paris.
 - ▶ C. Delannoy : le langage C, la référence ; Programmer en langage C.

Des sources d'informations

- ▶ Les *man pages* (sous UNIX) : man 3 **printf**
- ▶ Des sites (une petite sélection) :
 - ▶ http://www.lri.fr/~aze/page_c/aide_c/
 - ▶ http://www.linux-kheops.com/doc/ansi-c/Introduction_ANSI_C.htm
 - ▶ http://fr.wikibooks.org/wiki/Programmation_C
 - ▶ C facile www.enst.fr/~charon
- ▶ Des livres :
 - ▶ Kernighan & Ritchie : le langage C (la « bible »)
 - ▶ Varrette & N. Bernard : Programmation avancée en C, Hermès, Paris.
 - ▶ C. Delannoy : le langage C, la référence ; Programmer en langage C.

Des sources d'informations

- ▶ Les *man pages* (sous UNIX) : man 3 **printf**
- ▶ Des sites (une petite sélection) :
 - ▶ http://www.lri.fr/~aze/page_c/aide_c/
 - ▶ http://www.linux-kheops.com/doc/ansi-c/Introduction_ANSI_C.htm
 - ▶ http://fr.wikibooks.org/wiki/Programmation_C
 - ▶ **C facile** www.enst.fr/~charon
- ▶ Des livres :
 - ▶ Kernighan & Ritchie : le langage C (la « bible »)
 - ▶ Varrette & N. Bernard : Programmation avancée en C, Hermès, Paris.
 - ▶ C. Delannoy : le langage C, la référence ; Programmer en langage C.

Des sources d'informations

- ▶ Les *man pages* (sous UNIX) : man 3 **printf**
- ▶ Des sites (une petite sélection) :
 - ▶ http://www.lri.fr/~aze/page_c/aide_c/
 - ▶ http://www.linux-kheops.com/doc/ansi-c/Introduction_ANSI_C.htm
 - ▶ http://fr.wikibooks.org/wiki/Programmation_C
 - ▶ **C facile** www.enst.fr/~charon
- ▶ Des livres :
 - ▶ Kernighan & Richie : le langage C (la « bible »)
 - ▶ Varrette & N. Bernard : Programmation avancée en C, Hermès, Paris.
 - ▶ C. Delannoy : le langage C, la référence ; Programmer en langage C.

Des sources d'informations

- ▶ Les *man pages* (sous UNIX) : man 3 **printf**
- ▶ Des sites (une petite sélection) :
 - ▶ http://www.lri.fr/~aze/page_c/aide_c/
 - ▶ http://www.linux-kheops.com/doc/ansi-c/Introduction_ANSI_C.htm
 - ▶ http://fr.wikibooks.org/wiki/Programmation_C
 - ▶ C facile www.enst.fr/~charon
- ▶ Des livres :
 - ▶ Kernighan & Richie : le langage C (la « bible »)
 - ▶ Varrette & N. Bernard : Programmation avancée en C, Hermès, Paris.
 - ▶ C. Delannoy : le langage C, la référence ; Programmer en langage C.

Des sources d'informations

- ▶ Les *man pages* (sous UNIX) : man 3 **printf**
- ▶ Des sites (une petite sélection) :
 - ▶ http://www.lri.fr/~aze/page_c/aide_c/
 - ▶ http://www.linux-kheops.com/doc/ansi-c/Introduction_ANSI_C.htm
 - ▶ http://fr.wikibooks.org/wiki/Programmation_C
 - ▶ C facile www.enst.fr/~charon
- ▶ Des livres :
 - ▶ Kernighan & Richie : le langage C (la « bible »)
 - ▶ Varrette & N. Bernard : Programmation avancée en C, Hermès, Paris.
 - ▶ C. Delannoy : le langage C, la référence ; Programmer en langage C.

Des sources d'informations

- ▶ Les *man pages* (sous UNIX) : man 3 **printf**
- ▶ Des sites (une petite sélection) :
 - ▶ http://www.lri.fr/~aze/page_c/aide_c/
 - ▶ http://www.linux-kheops.com/doc/ansi-c/Introduction_ANSI_C.htm
 - ▶ http://fr.wikibooks.org/wiki/Programmation_C
 - ▶ C facile www.enst.fr/~charon
- ▶ Des livres :
 - ▶ Kernighan & Richie : le langage C (la « bible »)
 - ▶ Varrette & N. Bernard : Programmation avancée en C, Hermès, Paris.
 - ▶ C. Delannoy : le langage C, la référence ; Programmer en langage C.

Les carottes et les navets (fable)



+



= ???

- ▶ Les entités manipulées dans les programmes diffèrent :
- ▶ Des entités physiques
- ▶ Des entités abstraites
- ▶ Chaque entité utilisée dans un programme possède un *type*.
- ▶ Les types des entités sont explicitement *déclarés* ou implicitement inférés par le compilateur

Les carottes et les navets (fable)



+



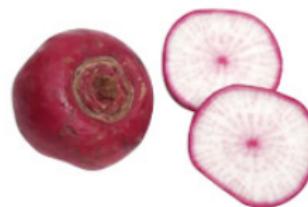
= ???

- ▶ Les entités manipulées dans les programmes diffèrent :
 - ▶ par leur représentation
 - ▶ par les opérations associées
- ▶ Chaque entité utilisée dans un programme possède un *type*.
- ▶ Les types des entités sont explicitement *déclarés* ou implicitement inférés par le compilateur

Les carottes et les navets (fable)



+



= ???

- ▶ Les entités manipulées dans les programmes diffèrent :
 - ▶ par leur représentation
 - ▶ par les opérations associées
- ▶ Chaque entité utilisée dans un programme possède un *type*.
- ▶ Les types des entités sont explicitement *déclarés* ou implicitement inférés par le compilateur

Les carottes et les navets (fable)



+



= ???

- ▶ Les entités manipulées dans les programmes diffèrent :
 - ▶ par leur représentation
 - ▶ par les opérations associées
- ▶ Chaque entité utilisée dans un programme possède un *type*.
- ▶ Les types des entités sont explicitement *déclarés* ou implicitement inférés par le compilateur

Les carottes et les navets (fable)



+



= ???

- ▶ Les entités manipulées dans les programmes diffèrent :
 - ▶ par leur représentation
 - ▶ par les opérations associées
- ▶ Chaque entité utilisée dans un programme possède un *type*.
- ▶ Les types des entités sont explicitement *déclarés* ou implicitement inférés par le compilateur

Les carottes et les navets (fable)



+



= ???

- ▶ Les entités manipulées dans les programmes diffèrent :
 - ▶ par leur représentation
 - ▶ par les opérations associées
- ▶ Chaque entité utilisée dans un programme possède un *type*.
- ▶ Les types des entités sont explicitement *déclarés* ou implicitement inférés par le compilateur

Déclarer les variables : une obligation

```
int      i = 1;  
char    c = 'c', d;  
float   f = 3, pi = 3.14, c=3e+8;  
double rac2 = sqrt(2);
```

- ▶ utiliser une variable non-déclarée cause une erreur de compilation :

error : 'i' undeclared (first use in this function)

- ▶ les déclarations/définitions sont en tête de bloc (!! sauf en C99 !)
- ▶ une bonne habitude : initialiser explicitement toute variable
- ▶ C admet des types *primitifs* pour les entiers et les réels
- ▶ un type = des opérations, des mécanismes de construction, des procédures et des formats d'entrée / sortie
- ▶ le *transtypage* permet de convertir un type en un autre
- ▶ il est possible de construire de nouveaux types

Déclarer les variables : une obligation

```
int      i = 1;  
char    c = 'c', d;  
float   f = 3, pi = 3.14, c=3e+8;  
double rac2 = sqrt(2);
```

- ▶ utiliser une variable non-déclarée cause une erreur de compilation :

error : 'i' undeclared (first use in this function)

- ▶ les déclarations/définitions sont en tête de bloc (!! sauf en C99 !)
- ▶ une bonne habitude : initialiser explicitement toute variable
- ▶ C admet des types *primitifs* pour les entiers et les réels
- ▶ un type = des opérations, des mécanismes de construction, des procédures et des formats d'entrée / sortie
- ▶ le *transtypage* permet de convertir un type en un autre
- ▶ il est possible de construire de nouveaux types

Déclarer les variables : une obligation

```
int      i = 1;  
char    c = 'c', d;  
float   f = 3, pi = 3.14, c=3e+8;  
double rac2 = sqrt(2);
```

- ▶ utiliser une variable non-déclarée cause une erreur de compilation :

error : 'i' undeclared (first use in this function)

- ▶ les déclarations/définitions sont en tête de bloc (*!! sauf en C99 !!*)
- ▶ une bonne habitude : initialiser explicitement toute variable
- ▶ C admet des types *primitifs* pour les entiers et les réels
- ▶ un type = des opérations, des mécanismes de construction, des procédures et des formats d'entrée / sortie
- ▶ le *transtypage* permet de convertir un type en un autre
- ▶ il est possible de construire de nouveaux types

Déclarer les variables : une obligation

```
int      i = 1;  
char    c = 'c', d;  
float   f = 3, pi = 3.14, c=3e+8;  
double rac2 = sqrt(2);
```

- ▶ utiliser une variable non-déclarée cause une erreur de compilation :

error : 'i' undeclared (first use in this function)

- ▶ les déclarations/définitions sont en tête de bloc (!! sauf en C99 !!)
- ▶ une bonne habitude : initialiser explicitement toute variable
- ▶ C admet des types *primitifs* pour les entiers et les réels
- ▶ un type = des opérations, des mécanismes de construction, des procédures et des formats d'entrée / sortie
- ▶ le *transtypage* permet de convertir un type en un autre
- ▶ il est possible de construire de nouveaux types

Déclarer les variables : une obligation

```
int      i = 1;  
char    c = 'c', d;  
float   f = 3, pi = 3.14, c=3e+8;  
double rac2 = sqrt(2);
```

- ▶ utiliser une variable non-déclarée cause une erreur de compilation :

error : 'i' undeclared (first use in this function)

- ▶ les déclarations/définitions sont en tête de bloc (!! sauf en C99 !!)
- ▶ une bonne habitude : initialiser explicitement toute variable
- ▶ **C admet des types *primitifs* pour les entiers et les réels**
- ▶ un type = des opérations, des mécanismes de construction, des procédures et des formats d'entrée / sortie
- ▶ le *transtypage* permet de convertir un type en un autre
- ▶ il est possible de construire de nouveaux types

Déclarer les variables : une obligation

```
int      i = 1;  
char    c = 'c', d;  
float   f = 3, pi = 3.14, c=3e+8;  
double rac2 = sqrt(2);
```

- ▶ utiliser une variable non-déclarée cause une erreur de compilation :

error : 'i' undeclared (first use in this function)

- ▶ les déclarations/définitions sont en tête de bloc (*!! sauf en C99 !!*)
- ▶ une bonne habitude : initialiser explicitement toute variable
- ▶ C admet des types *primitifs* pour les entiers et les réels
- ▶ un type = des opérations, des mécanismes de construction, des procédures et des formats d'entrée / sortie
- ▶ le *transtypage* permet de convertir un type en un autre
- ▶ il est possible de construire de nouveaux types

Déclarer les variables : une obligation

```
int      i = 1;  
char    c = 'c', d;  
float   f = 3, pi = 3.14, c=3e+8;  
double rac2 = sqrt(2);
```

- ▶ utiliser une variable non-déclarée cause une erreur de compilation :

error : 'i' undeclared (first use in this function)

- ▶ les déclarations/définitions sont en tête de bloc (!! sauf en C99 !!)
- ▶ une bonne habitude : initialiser explicitement toute variable
- ▶ C admet des types *primitifs* pour les entiers et les réels
- ▶ un type = des opérations, des mécanismes de construction, des procédures et des formats d'entrée / sortie
- ▶ le *transtypage* permet de convertir un type en un autre
- ▶ il est possible de construire de nouveaux types

Déclarer les variables : une obligation

```
int      i = 1;  
char    c = 'c', d;  
float   f = 3, pi = 3.14, c=3e+8;  
double rac2 = sqrt(2);
```

- ▶ utiliser une variable non-déclarée cause une erreur de compilation :

error : 'i' undeclared (first use in this function)

- ▶ les déclarations/définitions sont en tête de bloc (!! sauf en C99 !!)
- ▶ une bonne habitude : initialiser explicitement toute variable
- ▶ C admet des types *primitifs* pour les entiers et les réels
- ▶ un type = des opérations, des mécanismes de construction, des procédures et des formats d'entrée / sortie
- ▶ le *transtypage* permet de convertir un type en un autre
- ▶ **il est possible de construire de nouveaux types**

Les types d'entiers « historiques »

<i>signés</i>	taille	domaine	format
int	??	??	%i,%d
char	8 bits	[−128; 127]	%c
short int , short	16 bits	[32768; −32768]	%hi
long int , long	32 bits	[−2 ³¹ ; 2 ³¹ − 1]	%li
long long int	64 bits	[−2 ⁶³ ; 2 ⁶³ − 1]	%lli

- ▶ la taille des **int** dépend des machines/architectures ; le plus souvent 32 bits. Pour être certain : sizeof()
- ▶ les opérations arithmétiques habituelles : +, −, *,
- ▶ la division entière : /, le modulo : % : $p = q * (p/q) + p \% q$
- ▶ attention : **char** c=127; c = c + 1; donne c= −128
- ▶ plus généralement $i + j = (i + j) \text{mod } n$, avec $n = 2^{\text{sizeof(type)}}$

Les entiers de \mathbb{N}

<i>non signés</i>	taille	domaine	format
unsigned int , uint,			%u
unsigned char , uchar	8 bits	[0; 255]	%c
unsigned short int , ushort	16 bits	[0; 65535]	%hu
unsigned long int , ulong	32 bits	[0; $2^{32} - 1$]	%lu
unsigned long long int	64 bits	[0; $2^{64} - 1$]	%llu

- mêmes opérations que pour les entiers *signés*
- attention aux soustractions...

Les entiers de \mathbb{N}

<i>non signés</i>	taille	domaine	format
unsigned int , uint,			%u
unsigned char , uchar	8 bits	[0; 255]	%c
unsigned short int , ushort	16 bits	[0; 65535]	%hu
unsigned long int , ulong	32 bits	[0; $2^{32} - 1$]	%lu
unsigned long long int	64 bits	[0; $2^{64} - 1$]	%llu

- mêmes opérations que pour les entiers *signés*
- attention aux soustractions...

Dualité des caractères

- ▶ les caractères sont des entiers $\in [-128; 127]$
- ▶ la valeur d'un **char** dans $[0, 127]$ représente un *code ASCII*

Decimal	Octal	Hexa	Binaire	Valeur
000	000	000	00000000	NUL (Null char.)
009	011	009	00001001	HT (Horizontal Tab)
013	015	00D	00001101	CR (Carriage Return)
032	040	020	00100000	SP (Space)
033	041	021	00100001	! (exclamation mark)
048	060	030	00110000	0
049	061	031	00110001	1
065	101	041	01000001	A
090	132	05A	01011010	Z
097	141	061	01100001	a
098	142	062	01100010	b
126	176	07E	01111110	(tilde)
127	177	07F	01111111	DEL (delete)

voir <http://www.asciitable.com/>

- ▶ `char c = 56; printf ("char=%c int=%i")` imprime `char=8, int=56`
- ▶ les valeurs au-delà de 127 (“ASCII étendu” dépendent de la machine)
- ▶ nouvelles normes pour les caractères : UTF8 (2 bits), Unicode (4 bits)

Types entiers : opérations bit à bit

- ▶ constantes hexadécimales (base 16) : `char c = 0x0F;`
- ▶ opérations “logiques” : `&` (et), `|` (ou), `^`(xou), `~`(non) :

1	0	1	0		1	0	1	0		1	0	1	0	
&	0	1	1	0		0	1	1	0	^	0	1	1	0
	0	0	1	0		1	1	1	0		1	1	0	0

exemple : `i & 0x04` teste la divisibilité par 4

- ▶ décalage droite `>>` et gauche `<<`

1	0	1	0	<code><< 2 =</code>	1	0	0	0
0	1	1	0	<code>>> 2 =</code>	0	0	0	1

- ▶ décalage gauche de n bits = multiplication par 2^n (par analogie à la base 10!).

Types entiers : opérations booléennes

```
int i = 6, j = 7, k;  
  
k = (i && j);           /* k = 1 */  
k = (i || j);           /* k = 1 */  
k = (! i);               /* k = 0 */  
k = (! i && (j=j-1));   /* k = 0, j = 7 */  
k = ((j=j-1) && !i);    /* k = 0, j = 6 */
```

- ▶ pas de type booléen en C standard, contrairement à Java...
- ▶ mais C99 introduit `_Bool`
- ▶ 0 est **faux**; $\neq 0$ est **vrai**
- ▶ **faux** est 0; **vrai** est 1
- ▶ les opérateurs logiques ne sont pas commutatifs `!!`: évaluation de gauche à droite

Types entiers : opérations booléennes

```
int i = 6, j = 7, k;  
  
k = (i && j);           /* k = 1 */  
k = (i || j);           /* k = 1 */  
k = (! i);               /* k = 0 */  
k = (! i && (j=j-1));   /* k = 0, j = 7 */  
k = ((j=j-1) && !i);    /* k = 0, j = 6 */
```

- ▶ pas de type booléen en C standard, contrairement à Java...
- ▶ mais C99 introduit `_Bool`
- ▶ 0 est **faux**; $\neq 0$ est **vrai**
- ▶ **faux** est 0; **vrai** est 1
- ▶ les opérateurs logiques ne sont pas commutatifs `!!`: évaluation de gauche à droite

Types entiers : opérations booléennes

```
int i = 6, j = 7, k;  
  
k = (i && j);           /* k = 1 */  
k = (i || j);           /* k = 1 */  
k = (! i);               /* k = 0 */  
k = (! i && (j=j-1));   /* k = 0, j = 7 */  
k = ((j=j-1) && !i);    /* k = 0, j = 6 */
```

- ▶ pas de type booléen en C standard, contrairement à Java...
- ▶ mais C99 introduit _Bool
- ▶ 0 est **faux**; \neq 0 est **vrai**
- ▶ faux est 0 ; vrai est 1
- ▶ les opérateurs logiques ne sont pas commutatifs !! : évaluation de gauche à droite

Types entiers : opérations booléennes

```
int i = 6, j = 7, k;  
  
k = (i && j);           /* k = 1 */  
k = (i || j);           /* k = 1 */  
k = (! i);               /* k = 0 */  
k = (! i && (j=j-1));   /* k = 0, j = 7 */  
k = ((j=j-1) && !i);    /* k = 0, j = 6 */
```

- ▶ pas de type booléen en C standard, contrairement à Java...
- ▶ mais C99 introduit `_Bool`
- ▶ 0 est **faux**; $\neq 0$ est **vrai**
- ▶ **faux** est 0 ; **vrai** est 1
- ▶ les opérateurs logiques ne sont pas commutatifs `!!` : évaluation de gauche à droite

Types entiers : opérations booléennes

```
int i = 6, j = 7, k;  
  
k = (i && j);           /* k = 1 */  
k = (i || j);           /* k = 1 */  
k = (! i);               /* k = 0 */  
k = (! i && (j=j-1));   /* k = 0, j = 7 */  
k = ((j=j-1) && !i);    /* k = 0, j = 6 */
```

- ▶ pas de type booléen en C standard, contrairement à Java...
- ▶ mais C99 introduit `_Bool`
- ▶ 0 est **faux**; $\neq 0$ est **vrai**
- ▶ **faux** est 0 ; **vrai** est 1
- ▶ les opérateurs logiques ne sont pas commutatifs !! : évaluation de gauche à droite

Opérateurs d'incrément : des affectations masquées

```
short int i=0, j=10;  
// calcul i+1 PUIS change la valeur de i  
i = i + 1;  
i += 1;  
i++;  
// change d'un coup i et j  
j = i++;  
j = ++i;
```

- ▶ $j *= 10$ signifie $j = j * 10$, idem pour les opérations entières
 $/=,%=,<<=,>>=,&=,^=,|=$
- ▶ $j=++i$ signifie $j = (i = i+1)$;
- ▶ $j=i++$ signifie $j = i; i = i+1$;
- ▶ $j=i--$ est différent de $j = --i$;
- ▶ instruction peu lisible, risque d'erreurs \Rightarrow à éviter

Opérateurs d'incrément : des affectations masquées

```
short int i=0, j=10;  
// calcul i+1 PUIS change la valeur de i  
i = i + 1;  
i += 1;  
i++;  
// change d'un coup i et j  
j = i++;  
j = ++i;
```

- ▶ $j *= 10$ signifie $j = j * 10$, idem pour les opérations entières
 $/=, \% =, < < =, > > =, \& =, ^ =, | =$
- ▶ $j=++i$ signifie $j = (i = i+1)$;
- ▶ $j=i++$ signifie $j = i; i = i+1$;
- ▶ $j=i--$ est différent de $j = --i$;
- ▶ instruction peu lisible, risque d'erreurs \Rightarrow à éviter

Opérateurs d'incrément : des affectations masquées

```
short int i=0, j=10;  
// calcul i+1 PUIS change la valeur de i  
i = i + 1;  
i += 1;  
i++;  
// change d'un coup i et j  
j = i++;  
j = ++i;
```

- ▶ **j *= 10** signifie $j = j * 10$, idem pour les opérations entières
 $/=, \%=<=>=&^|=$
- ▶ **j=++i** signifie $j = (i = i+1)$;
- ▶ **j=i++** signifie $j = i$; $i = i+1$;
- ▶ $j=i--$ est différent de $j = --i$;
- ▶ instruction peu lisible, risque d'erreurs \Rightarrow à éviter

Opérateurs d'incrément : des affectations masquées

```
short int i=0, j=10;  
// calcul i+1 PUIS change la valeur de i  
i = i + 1;  
i += 1;  
i++;  
// change d'un coup i et j  
j = i++;  
j = ++i;
```

- ▶ $j *= 10$ signifie $j = j * 10$, idem pour les opérations entières
 $/=, \%=<=>=&^|=$
- ▶ $j=++i$ signifie $j = (i = i+1)$;
- ▶ $j=i++$ signifie $j = i$; $i = i+1$;
- ▶ $j=i--$ est différent de $j = --i$;
- ▶ instruction peu lisible, risque d'erreurs \Rightarrow à éviter

Opérateurs d'incrément : des affectations masquées

```
short int i=0, j=10;  
// calcul i+1 PUIS change la valeur de i  
i = i + 1;  
i += 1;  
i++;  
// change d'un coup i et j  
j = i++;  
j = ++i;
```

- ▶ $j *= 10$ signifie $j = j * 10$, idem pour les opérations entières
 $/=, \%=<=>=&^|=$
- ▶ $j=++i$ signifie $j = (i = i+1)$;
- ▶ $j=i++$ signifie $j = i$; $i = i+1$;
- ▶ $j=i--$ est différent de $j = --i$;
- ▶ instruction peu lisible, risque d'erreurs ⇒ à éviter

Les entiers modernes (C99)

- ▶ des certitudes sur la taille : int8_t , uint32_t...
- ▶ des garanties sur la taille : int_least64_t ...
- ▶ des garanties sur la vitesse : uint_fast32_t ...
- ▶ des formats associés pour lire et écrire PRI[dux](LEAST,FAST)?

Les opérateurs : priorité à ?

priorité	Opérateur	Associativité
16	() [] -> . ++ --	G
15	! ~ ++ -- - + * sizeof	D
14	conversion	D
13	* / %	G
12	+ -	G
11	<< >>	G
10	< <= > >=	G
9	== !=	G
8	&	G
7	^	G
6		G
5	&&	G
4		G
3	?:	D
2	= += -= *= /= %= >>= <<= &= ^= =	D
1	,	G

Respirations (d'après A. Feuer)

```
int x, y, z;  
x = - 3 + 4 * 5 - 6;  
x = 3 + 4 % 5 - 6;  
z = - 3 * 4 % - 6 / 5;  
x = (7 + 6) % 5 / 2 + 1;  
  
x *= 3 + 2;  
x *= y = z = 4;  
x = y == z;  
x == (y = z);
```

```
// x = (-3)+(4*5)-6 = 11;  
// x = 3+(4%5)-6 = 1;  
// x = (((-3)*4)%(-6))/5 = 0  
// x = (((7+6)%5)/2)+1 = 2;  
  
// x *= (3 + 2) = 10;  
// x = 40; y = 4; z = 4;  
// x = 1; y = 4, z = 4;  
// x = 1; y = 4; z = 4;
```

Respirations (d'après A. Feuer)

```
int x, y, z;  
x = - 3 + 4 * 5 - 6;  
x = 3 + 4 % 5 - 6;  
z = - 3 * 4 % - 6 / 5;  
x = (7 + 6) % 5 / 2 + 1;  
  
x *= 3 + 2;  
x *= y = z = 4;  
x = y == z;  
x == (y = z);
```

```
// x = (-3)+(4*5)-6 = 11;  
// x = 3+(4%5)-6 = 1;  
// x = (((-3)*4)%(-6))/5 = 0  
// x = (((7+6)%5)/2)+1 = 2;  
  
// x *= (3 + 2) = 10;  
// x = 40; y = 4; z = 4;  
// x = 1; y = 4, z = 4;  
// x = 1; y = 4; z = 4;
```

Respirations

```
int x, y, z;  
x = 2; y = 1; z = 0;  
x = x && y || z;  
z = x || ! y && z;  
  
x = y = 1;  
z = x++ -1;  
z += -x++ + ++y;  
z = x / ++x;
```

```
// x = 2; y = 1; z = 0;  
// x = 1; y = 1; z = 0;  
// x = 2; y = 1; z = 1;  
  
// x = 1; y = 1; z = 1;  
// x = 2; y = 1; z = 0;  
// x = 3; y = 2; z = 0;  
// x = 4; y = 2; z = ?;
```

Respirations

```
int x, y, z;  
x = 2; y = 1; z = 0;  
x = x && y || z;  
z = x || ! y && z;  
  
x = y = 1;  
z = x++ -1;  
z += -x++ + ++y;  
z = x / ++x;
```

```
// x = 2; y = 1; z = 0;  
// x = 1; y = 1; z = 0;  
// x = 2; y = 1; z = 1;  
  
// x = 1; y = 1; z = 1;  
// x = 2; y = 1; z = 0;  
// x = 3; y = 2; z = 0;  
// x = 4; y = 2; z = ?;
```

Les types réels : représenter l'infini

- ▶ le type **float** représente les réels sur 32 bits
- ▶ le type **double** représente les réels sur 64 bits

	maximum	minimum
> 0	1.797693134862231E+308	4.940656458412465E-324
< 0	-4.940656458412465E-324	-1.797693134862231E+308

- ▶ 64 bits = 1 bit de signe, 52 pour la partie significative k , 11 pour l'exposant i

$$r = +/ - (1 + k) * 2^{i-1023}$$

- ▶ E/S en notation partie entière et décimale : $+/- \text{int}.\text{dec}$, format "%n.pf"
- ▶ E/S en notation scientifique : $+/- \text{int}.\text{dec}E + / - \text{exp}$, format "%n.pe"
- ▶ l'arithmétique (+, *, -, /, **), à la précision près...
- ▶ la précision est $\approx 10e^{-15}$ pour les doubles

Transtypage implicite et explicite

- ▶ tout **char** est *mathématiquement* un **short** ... ; tout **int** est *mathématiquement* un **float** ;
ce n'est plus vrai informatiquement !
- ▶ conversion implicites :
 - ▶ opérations
 - ▶ affectations
 - ▶ appels et retours de fonctions
- ▶ Les principales règles de transtypage :
 - ▶ « promotion » des **int**
 - ▶ **double** > **float** > tous les types entiers
 - ▶ **long** > **int** ; **unsigned** > **int**
 - ▶ **unsigned** + **long** ⇒ **unsigned long**
- ▶ l'opérateur de coercion (*cast*) rend explicite le transtypage :
(float)expr force l'interprétation de expr comme un float ;
- ▶ les transtypages implicites sont signalés durant la compilation

Respirations (C. Delannoy)

```
char c = 1;  
short p = 10;  
unsigned int i = 100;  
long l = 1000;  
float x = 1.25;  
double z = 5.5;
```

Quels sont les types et valeurs de ?

p + 3

c + 1

3 * p + 5 * c

l * i + c

l + p + c

2 * x + c

(char)p + c

(float)z + (p+1) / 2

Réponses :

(int)13

(int)2

(int)35

(unsigned long)100~001

(long)1011

(float)3.5

(int)10

(float)10.5

Respirations (C. Delannoy)

```
char c = 1;
short p = 10;
unsigned int i = 100;
long l = 1000;
float x = 1.25;
double z = 5.5;
```

Quels sont les types et valeurs de ?

p + 3

c + 1

3 * p + 5 * c

l * i + c

l + p + c

2 * x + c

(**char**)p + c

(**float**)z + (p+1) / 2

Réponses :

(**int**)13

(**int**)2

(**int**)35

(**unsigned long**)100~001

(**long**)1011

(**float**)3.5

(**int**)10

(**float**)10.5

Respiration

1. Qu'imprime ?

```
unsigned int i = 0;
if (i < -1) printf("J'y perds mon latin !\n")
else printf("Quel est le probl\`eme ?\n")
```

2. Qu'imprime ?

```
uchar c1 = 150, c2 = 150, c3;
c3 = c1 + c2;
printf("c1 + c2 = %i, c3 = %i\n", c1 + c2, c3);
```

Des variables catégorielles : les énumérations

```
/* la liste des mois */  
enum month {jan , feb , mar, apr , may, jun} m1, m2;  
  
/* la liste des jours ouvr'és */  
enum day {mon=1, tue=2, wed=3, thu=4, fri=5} d1;  
  
/* une nouvelle variable , qui vaut wednesday (3) */  
enum day d2=wed;
```

- ▶ définit des sous-ensembles d'entiers (de 0 à n) pour représenter des variables catégorielles (= sans arithmétique) : les jours de la semaine...
- ▶ auxquels sont associés des identificateurs symboliques
- ▶ en C, une énumération est un type unsigned int ;
- ▶ pas de contrôle syntaxique ni sémantique : $d2 = d2 * d2;$ est ok

Des variables catégorielles : les énumérations

```
/* la liste des mois */
enum month {jan , feb , mar, apr , may, jun} m1, m2;

/* la liste des jours ouvr'és */
enum day {mon=1, tue=2, wed=3, thu=4, fri=5} d1;

/* une nouvelle variable , qui vaut wednesday (3) */
enum day d2=wed;
```

- ▶ définit des sous-ensembles d'entiers (de 0 à n) pour représenter des variables catégorielles (= sans arithmétique) : les jours de la semaine...
- ▶ auxquels sont associés des identificateurs symboliques
- ▶ en C, une énumération est un type unsigned int ;
- ▶ pas de contrôle syntaxique ni sémantique : $d2 = d2 * d2;$ est ok

Des variables catégorielles : les énumérations

```
/* la liste des mois */
enum month {jan , feb , mar, apr , may, jun} m1, m2;

/* la liste des jours ouvr'és */
enum day {mon=1, tue=2, wed=3, thu=4, fri=5} d1;

/* une nouvelle variable , qui vaut wednesday (3) */
enum day d2=wed;
```

- ▶ définit des sous-ensembles d'entiers (de 0 à n) pour représenter des variables catégorielles (= sans arithmétique) : les jours de la semaine...
- ▶ auxquels sont associés des identificateurs symboliques
- ▶ en C, une énumération est un type **unsigned int** ;
- ▶ pas de contrôle syntaxique ni sémantique : $d2 = d2 * d2;$ est ok

Des variables catégorielles : les énumérations

```
/* la liste des mois */  
enum month {jan , feb , mar, apr , may, jun} m1, m2;  
  
/* la liste des jours ouvr'és */  
enum day {mon=1, tue=2, wed=3, thu=4, fri=5} d1;  
  
/* une nouvelle variable , qui vaut wednesday (3) */  
enum day d2=wed;
```

- ▶ définit des sous-ensembles d'entiers (de 0 à n) pour représenter des variables catégorielles (= sans arithmétique) : les jours de la semaine...
- ▶ auxquels sont associés des identificateurs symboliques
- ▶ en C, une énumération est un type unsigned int ;
- ▶ pas de contrôle syntaxique ni sémantique : $d2 = d2 * d2;$ est ok

Fabrication d'entités nouvelles : les structures

```
struct point {  
    double coordx;  
    double coordy;  
} p1 = {0,0}, p2;  
struct rectangle {  
    struct point coinbg, coinhd;  
    double surface;  
} r;  
  
p2.coordx = 1; p2.coordy = 1;  
r.coinbg = p1;  
r.coinbg = p2;  
r.surface = (r.coinhd.coordx - r.coinbg.coordx) *  
            (r.coinhd.coordy - r.coinbg.coordy);
```

- ▶ une structure est une entité complexe fondée de types connus ;
- ▶ les *membres* sont les composants de la structure
- ▶ p1.coordx accède à la valeur du membre coordx pour p1

Fabrication d'entités nouvelles : les structures

```
struct point {  
    double coordx;  
    double coordy;  
} p1 = {0,0}, p2;  
struct rectangle {  
    struct point coinbg, coinhd;  
    double surface;  
} r;  
  
p2.coordx = 1; p2.coordy = 1;  
r.coinbg = p1;  
r.coinbg = p2;  
r.surface = (r.coinhd.coordx - r.coinbg.coordx) *  
            (r.coinhd.coordy - r.coinbg.coordy);
```

- ▶ une structure est une entité complexe fondée de types connus ;
- ▶ *les membres* sont les composants de la structure
- ▶ p1.coordx accède à la valeur du membre coordx pour p1

Fabrication d'entités nouvelles : les structures

```
struct point {  
    double coordx;  
    double coordy;  
} p1 = {0,0}, p2;  
struct rectangle {  
    struct point coinbg, coinhd;  
    double surface;  
} r;  
  
p2.coordx = 1; p2.coordy = 1;  
r.coinbg = p1;  
r.coinbg = p2;  
r.surface = (r.coinhd.coordx - r.coinbg.coordx) *  
            (r.coinhd.coordy - r.coinbg.coordy);
```

- ▶ une structure est une entité complexe fondée de types connus ;
- ▶ *les membres* sont les composants de la structure
- ▶ **p1.coordx accède à la valeur du membre coordx pour p1**

Structures élémentaires... (suite)

```
struct date_t {  
    unsigned short day;  
    unsigned short month;  
    unsigned int year;  
}  
struct car {  
    struct date_t launchdate;  
    int colour;  
    double maxspeed;  
} austin;
```

- ▶ les structures peuvent contenir des sous-structures
- ▶ austin.launchdate.year est l'année de lancement
- ▶ une fois définie une structure est réutilisable (au sein du même bloc)
- ▶ if (p1 == p2); est une erreur !!
- ▶ ⇒ comparaison champ par champ

Structures élémentaires... (suite)

```
struct date_t {  
    unsigned short day;  
    unsigned short month;  
    unsigned int year;  
}  
struct car {  
    struct date_t launchdate;  
    int colour;  
    double maxspeed;  
} austin;
```

- ▶ les structures peuvent contenir des sous-structures
- ▶ **austin.launchdate.year est l'année de lancement**
- ▶ une fois définie une structure est réutilisable (au sein du même bloc)
- ▶ if (p1 == p2); est une erreur !!
- ▶ ⇒ comparaison champ par champ

Structures élémentaires... (suite)

```
struct date_t {  
    unsigned short day;  
    unsigned short month;  
    unsigned int year;  
}  
struct car {  
    struct date_t launchdate;  
    int colour;  
    double maxspeed;  
} austin;
```

- ▶ les structures peuvent contenir des sous-structures
- ▶ austin.launchdate.year est l'année de lancement
- ▶ une fois définie une structure est réutilisable (au sein du même bloc)
- ▶ if (p1 == p2); est une erreur !!
- ▶ ⇒ comparaison champ par champ

Structures élémentaires... (suite)

```
struct date_t {  
    unsigned short day;  
    unsigned short month;  
    unsigned int year;  
}  
struct car {  
    struct date_t launchdate;  
    int colour;  
    double maxspeed;  
} austin;
```

- ▶ les structures peuvent contenir des sous-structures
- ▶ austin.launchdate.year est l'année de lancement
- ▶ une fois définie une structure est réutilisable (au sein du même bloc)
- ▶ **if (p1 == p2);** est une erreur !!
- ▶ ⇒ comparaison champ par champ

Structures élémentaires... (suite)

```
struct date_t {  
    unsigned short day;  
    unsigned short month;  
    unsigned int year;  
}  
struct car {  
    struct date_t launchdate;  
    int colour;  
    double maxspeed;  
} austin;
```

- ▶ les structures peuvent contenir des sous-structures
- ▶ austin.launchdate.year est l'année de lancement
- ▶ une fois définie une structure est réutilisable (au sein du même bloc)
- ▶ if (p1 == p2); est une erreur !!
- ▶ ⇒ **comparaison champ par champ**

Des entités polymorphes : les unions

```
union couteau_suisse {
    unsigned long i;
    float f;
} cs;

cs.i = 1;
printf("1 lu comme un float vaut %f\n", cs.f);
```

- ▶ une union permet de varier *l'interprétation* d'une zone de mémoire ;
- ▶ cs.i interprète cs comme un unsigned int
- ▶ cs.f interprète cs comme un float
- ▶ toujours connaître le type de la valeur courante de l'union, sinon...
- ▶ les objets n'ont pas nécessairement la même taille : double f est aussi ok ;

Des entités polymorphes : les unions

```
union couteau_suisse {
    unsigned long i;
    float f;
} cs;

cs.i = 1;
printf("1 lu comme un float vaut %f\n", cs.f);
```

- ▶ une union permet de varier *l'interprétation* d'une zone de mémoire ;
- ▶ cs.i interprète cs comme un unsigned int
- ▶ cs.f interprète cs comme un float
- ▶ toujours connaître le type de la valeur courante de l'union, sinon...
- ▶ les objets n'ont pas nécessairement la même taille : double f est aussi ok ;

Des entités polymorphes : les unions

```
union couteau_suisse {
    unsigned long i;
    float f;
} cs;

cs.i = 1;
printf("1 lu comme un float vaut %f\n", cs.f);
```

- ▶ une union permet de varier *l'interprétation* d'une zone de mémoire ;
- ▶ cs.i interprète cs comme un unsigned int
- ▶ cs.f interprète cs comme un float
- ▶ toujours connaître le type de la valeur courante de l'union, sinon...
- ▶ les objets n'ont pas nécessairement la même taille : double f est aussi ok ;

Des entités polymorphes : les unions

```
union couteau_suisse {  
    unsigned long i;  
    float f;  
} cs;  
  
cs.i = 1;  
printf("1 lu comme un float vaut %f\n", cs.f);
```

- ▶ une union permet de varier *l'interprétation* d'une zone de mémoire ;
- ▶ cs.i interprète cs comme un unsigned int
- ▶ cs.f interprète cs comme un float
- ▶ toujours connaître le type de la valeur courante de l'union, sinon...
- ▶ les objets n'ont pas nécessairement la même taille : double f est aussi ok ;

Des entités polymorphes : les unions

```
union couteau_suisse {  
    unsigned long i;  
    float f;  
} cs;  
  
cs.i = 1;  
printf("1 lu comme un float vaut %f\n", cs.f);
```

- ▶ une union permet de varier *l'interprétation* d'une zone de mémoire ;
- ▶ cs.i interprète cs comme un unsigned int
- ▶ cs.f interprète cs comme un float
- ▶ toujours connaître le type de la valeur courante de l'union, sinon...
- ▶ les objets n'ont pas nécessairement la même taille : double f est aussi ok ;

Un type écolo : le champ (vecteur) de bits

```
struct trilettre {  
    unsigned int l1 : 5;  
    unsigned int l2 : 5;  
    unsigned int l3 : 5;  
    unsigned :1;  
};
```

0	0	1	1	0	1	0	1	0	0	1	1	1	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

- ▶ possibilité d'identifier chaque bit ou groupe de bit trilettre .l1 = 1;
- ▶ chaque valeur est codée « au plus économique »
- ▶ utilisation : programmation système et réseau

Des types personnalisés avec `typedef`

```
typedef int index;
typedef struct {
    double real;
    double imag;
} complex;

complex c = {0, 1};
complex Points [10];
```

- ▶ **typedef** Type NewType permet de définir le nouveau nom de type NewType
- ▶ **typedef** ne crée pas de variable (sauf si on insiste :
`typedef struct foo { ... };`)
- ▶ un nom de type défini par **typedef** s'emploie comme un type prédéfini
- ▶ intérêt : portabilité, lisibilité des programmes.
- ▶ NB : le type complex fait partie de la norme C99...

Des types personnalisés avec `typedef`

```
typedef int index;
typedef struct {
    double real;
    double imag;
} complex;

complex c = {0, 1};
complex Points [10];
```

- ▶ **`typedef`** Type NewType permet de définir le nouveau nom de type NewType
- ▶ **`typedef`** ne crée pas de variable (sauf si on insiste :
`typedef struct foo { ... };`)
- ▶ un nom de type défini par `typedef` s'emploie comme un type prédéfini
- ▶ intérêt : portabilité, lisibilité des programmes.
- ▶ NB : le type complex fait partie de la norme C99...

Des types personnalisés avec `typedef`

```
typedef int index;
typedef struct {
    double real;
    double imag;
} complex;

complex c = {0, 1};
complex Points [10];
```

- ▶ **`typedef`** Type NewType permet de définir le nouveau nom de type NewType
- ▶ **`typedef`** ne crée pas de variable (sauf si on insiste :
`typedef struct` foo { ... });
- ▶ un nom de type défini par **`typedef`** s'emploie comme un type prédefini
- ▶ intérêt : portabilité, lisibilité des programmes.
- ▶ NB : le type complex fait partie de la norme C99...

Des types personnalisés avec **typedef**

```
typedef int index;
typedef struct {
    double real;
    double imag;
} complex;

complex c = {0, 1};
complex Points [10];
```

- ▶ **typedef** Type NewType permet de définir le nouveau nom de type NewType
- ▶ **typedef** ne crée pas de variable (sauf si on insiste :
typedef struct foo { ... });
- ▶ un nom de type défini par **typedef** s'emploie comme un type prédéfini
- ▶ intérêt : portabilité, lisibilité des programmes.
- ▶ NB : le type complex fait partie de la norme C99...

Des types personnalisés avec **typedef**

```
typedef int index;
typedef struct {
    double real;
    double imag;
} complex;

complex c = {0, 1};
complex Points [10];
```

- ▶ **typedef** Type NewType permet de définir le nouveau nom de type NewType
- ▶ **typedef** ne crée pas de variable (sauf si on insiste :
typedef struct foo { ... });
- ▶ un nom de type défini par **typedef** s'emploie comme un type prédéfini
- ▶ intérêt : portabilité, lisibilité des programmes.
- ▶ **NB : le type complex fait partie de la norme C99...**

Le(s) thème(s) du jour: Structures de contrôle

Branchements



La condition informatique

```
int i = 0, j = 1, k = 2;  
char c = 'a', C = 'A';  
i = (j == k);  
i = (j < k);  
i = (j >= k);  
i = (j != k);  
i = (c >= A);
```

- ▶ tester l'égalité : ==
- ▶ un test ne réalise aucune affectation
- ▶ l'ordre dans les entiers et les réels >, <, <=, >=
- ▶ un opérateur pour les négations a != b; !(a <= b)

La condition informatique

```
int i = 0, j = 1, k = 2;
char c = 'a', C = 'A';
i = (j == k);
i = (j < k);
i = (j >= k);
i = (j != k);
i = (c >= A);
```

► tester l'égalité : ==

- ▶ un test ne réalise aucune affectation
- ▶ l'ordre dans les entiers et les réels >, <, <=, >=
- ▶ un opérateur pour les négations a != b; !(a <= b)

La condition informatique

```
int i = 0, j = 1, k = 2;
char c = 'a', C = 'A';
i = (j == k);
i = (j < k);
i = (j >= k);
i = (j != k);
i = (c >= A);
```

- ▶ tester l'égalité : `==`
- ▶ un test ne réalise aucune affectation
- ▶ l'ordre dans les entiers et les réels `>`, `<`, `<=`, `>=`
- ▶ un opérateur pour les négations `a != b`; `!(a <= b)`

La condition informatique

```
int i = 0, j = 1, k = 2;  
char c = 'a', C = 'A';  
i = (j == k);  
i = (j < k);  
i = (j >= k);  
i = (j != k);  
i = (c >= A);
```

- ▶ tester l'égalité : ==
- ▶ un test ne réalise aucune affectation
- ▶ l'ordre dans les entiers et les réels >, <, <=, >=
- ▶ un opérateur pour les négations a != b; !(a <= b)

La condition informatique

```
int i = 0, j = 1, k = 2;
char c = 'a', C = 'A';
i = (j == k);
i = (j < k);
i = (j >= k);
i = (j != k);
i = (c >= A);
```

- ▶ tester l'égalité : `==`
- ▶ un test ne réalise aucune affectation
- ▶ l'ordre dans les entiers et les réels `>`, `<`, `<=`, `>=`
- ▶ un opérateur pour les négations `a != b`; `!(a <= b)`

Le branchement simple : if-else

```
if (b) p;  
else q;
```

```
if (b) p;
```

```
if (b) {  
    p;  
    q  
}
```

- ▶ la condition b est une expression dont l'évaluation est un entier
- ▶ l'instruction (ou le bloc) p; est exécuté(e) si l'évaluation de b est $\neq 0$; sinon l'instruction (ou le bloc) q est exécuté ;
- ▶ l'absence de clause else équivaut à else continue;
- ▶ de nombreux langages écrivent if (b) then p;
- ▶ on trouve également if ... fi (p. ex en shell).

Le branchement simple : if-else

```
if (b) p;  
else q;
```

```
if (b) p;
```

```
if (b) {  
    p;  
    q  
}
```

- ▶ la condition b est une expression dont l'évaluation est un entier
- ▶ l'instruction (ou le bloc) p; est exécuté(e) si l'évaluation de b est $\neq 0$; sinon l'instruction (ou le bloc) q est exécuté ;
- ▶ l'absence de clause else équivaut à else continue;
- ▶ de nombreux langages écrivent if (b) then p;
- ▶ on trouve également if ... fi (p. ex en shell).

Le branchement simple : if-else

```
if (b) p;  
else q;
```

```
if (b) p;
```

```
if (b) {  
    p;  
    q  
}
```

- ▶ la condition b est une expression dont l'évaluation est un entier
- ▶ l'instruction (ou le bloc) p; est exécuté(e) si l'évaluation de b est $\neq 0$; sinon l'instruction (ou le bloc) q est exécuté ;
- ▶ l'absence de clause else équivaut à else continue;
- ▶ de nombreux langages écrivent if (b) then p;
- ▶ on trouve également if ... fi (p. ex en shell).

Le branchement simple : if-else

```
if (b) p;  
else q;
```

```
if (b) p;
```

```
if (b) {  
    p;  
    q  
}
```

- ▶ la condition b est une expression dont l'évaluation est un entier
- ▶ l'instruction (ou le bloc) p; est exécuté(e) si l'évaluation de b est $\neq 0$; sinon l'instruction (ou le bloc) q est exécuté ;
- ▶ l'absence de clause else équivaut à else continue;
- ▶ de nombreux langages écrivent if (b) then p;
- ▶ on trouve également if ... fi (p. ex en shell).

Le branchement simple : if-else

```
if (b) p;  
else q;
```

```
if (b) p;
```

```
if (b) {  
    p;  
    q  
}
```

- ▶ la condition b est une expression dont l'évaluation est un entier
- ▶ l'instruction (ou le bloc) p; est exécuté(e) si l'évaluation de b est $\neq 0$; sinon l'instruction (ou le bloc) q est exécuté ;
- ▶ l'absence de clause else équivaut à else continue;
- ▶ de nombreux langages écrivent if (b) then p;
- ▶ on trouve également if ... fi (p. ex en shell).

Le branchement multiple : **if-else-if...**

```
if (b1) p1;  
else if (b2) p2;  
...  
else p;
```

▶ play ifelseif

- ▶ if (b1) if (b2) p2; else p3; est *ambigu*; lire (et écrire !!) :
if (b1) {if (b2) p2; else p3;}
- ▶ de nombreux langages (p. ex sh) introduisent **elif** pour else if.

Le branchement multiple : **if-else-if...**

```
if (b1) p1;  
else if (b2) p2;  
...  
else p;
```

▶ play ifelseif

- ▶ if (b1) if (b2) p2; else p3; est *ambigu*; lire (et écrire !!) :
if (b1) {if (b2) p2; else p3;}
- ▶ de nombreux langages (p. ex sh) introduisent **elif** pour else if.

Le branchement sélectif : **switch**

```
switch (var) {  
    case (c1): p1; break;  
    case (c2): p2; break;  
    default: p3;  
}  
q;
```

- ▶ **switch** équivaut à une série de **else if** ;
- ▶ var est une variable entière (y compris **char**) ;
- ▶ c1, c2 sont des *constantes*
- ▶ **break** interrompt l'exécution, qui reprendra après le bloc courant (avec q)
- ▶ le même sans **break**; : après p1, p2 est également exécuté

Le branchement sélectif : **switch**

```
switch (var) {  
    case (c1): p1; break;  
    case (c2): p2; break;  
    default: p3;  
}  
q;
```

- ▶ **switch** équivaut à une série de **else if** ;
- ▶ var est une variable entière (y compris **char**) ;
- ▶ c1, c2 sont des *constantes*
- ▶ **break** interrompt l'exécution, qui reprendra après le bloc courant (avec q)
- ▶ le même sans **break**; : après p1, p2 est également exécuté

Le branchement sélectif : **switch**

```
switch (var) {  
    case (c1): p1; break;  
    case (c2): p2; break;  
    default: p3;  
}  
q;
```

- ▶ **switch** équivaut à une série de **else if** ;
- ▶ var est une variable entière (y compris **char**) ;
- ▶ c1, c2 sont des *constantes*
- ▶ **break** interrompt l'exécution, qui reprendra après le bloc courant (avec q)
- ▶ le même sans **break**; : après p1, p2 est également exécuté

Le branchement sélectif : **switch**

```
switch (var) {  
    case (c1): p1; break;  
    case (c2): p2; break;  
    default: p3;  
}  
q;
```

- ▶ **switch** équivaut à une série de **else if** ;
- ▶ var est une variable entière (y compris **char**) ;
- ▶ c1, c2 sont des *constantes*
- ▶ **break** interrompt l'exécution, qui reprendra après le bloc courant (avec q)
- ▶ le même sans **break**; : après p1, p2 est également exécuté

Le branchement sélectif : **switch**

```
switch (var) {  
    case (c1): p1; break;  
    case (c2): p2; break;  
    default: p3;  
}  
q;
```

- ▶ **switch** équivaut à une série de **else if** ;
- ▶ var est une variable entière (y compris **char**) ;
- ▶ c1, c2 sont des *constantes*
- ▶ **break** interrompt l'exécution, qui reprendra après le bloc courant (avec q)
- ▶ le même sans **break**; : après p1, p2 est également exécuté

Une *expression* conditionnelle

```
// Le maximum  
max = (x > y ? x : y);  
// signifie  
if (x > y) max = x;  
else max = y;
```

- ▶ opérateur ternaire (`test ? expr1 : expr2`) ; vaut `expr1` si `test` est « vrai » ; `expr2` sinon
- ▶ possède un type (celui de `expr1` et `expr2`) et une valeur (celle de `expr1` ou `expr2`)

Une *expression* conditionnelle

```
// Le maximum  
max = (x > y ? x : y);  
// signifie  
if (x > y) max = x;  
else max = y;
```

- ▶ opérateur ternaire (`test ? expr1 : expr2`) ; vaut `expr1` si `test` est « vrai » ; `expr2` sinon
- ▶ possède un type (celui de `expr1` et `expr2`) et une valeur (celle de `expr1` ou `expr2`)

Respirations

```

int x, y = 1, z;

if (y != 0) x = 5;

if (y == 0) x = 3;
else x = 1;

if (y<0) if (y > 0) x = 3;
else x = 5;

if (z=y<0) x = 3;
else if (y == 0) x = 5;
else x = 7;

if (z=(y==0) ) x = 5; x = 3;

if (x = y = z) ; x = 3;

```

```

// x = ? ; y = 1; z = ?;

// x = 5 ; y = 1; z = ?;

// Faux (y = 1)
// x = 1;

// Faux (y > 0)
// x = 1;

// Faux (y > 0, donc z = 0)
// Faux (y = 1)
// x = 7;

// x = 3;

// x = 3;

```

Respirations

```
int x, y = 1, z;  
  
if (y != 0) x = 5;  
  
if (y == 0) x = 3;  
else x = 1;  
  
if (y<0) if (y > 0) x = 3;  
else x = 5;  
  
if (z=y<0) x = 3;  
else if (y == 0) x = 5;  
else x = 7;  
  
if (z=(y==0) ) x = 5; x = 3;  
  
if (x = y = z) ; x = 3;
```

```
// x = ? ; y = 1; z = ?;  
  
// x = 5 ; y = 1; z = ?;  
  
// Faux (y = 1)  
// x = 1;  
  
// Faux (y > 0)  
// x = 1;  
  
// Faux (y > 0, donc z = 0)  
// Faux (y = 1)  
// x = 7;  
  
// x = 3;  
  
// x = 3;
```

Respirations (J. Feuer)

```
int main()
{
    int x = 1, y = 1, z = 1;

    x += y += z;
    printf ("%d", (x < y ? y : x));

    printf ("%d", (x < y ? x++ : y++));

    printf ("%d", (z += x < y ? x++ : y++));

    x = 3; y = z = 4;
    printf ("%d", (z > y > x ? 1 : 0));
    printf ("%d", (z > y && y > x));
}
```

```
//  
//  
//  
// y = 2, x = 3  
// → 3  
  
// → 2,  
// y = 3, x = 3  
// → 4  
// z = 4 (= 4 !)  
//  
// → 0  
// → 0  
//
```

Respirations (J. Feuer)

```
int main()
{
    int x = 1, y = 1, z = 1;

    x += y += z;
    printf ("%d", (x < y ? y : x));

    printf ("%d", (x < y ? x++ : y++));

    printf ("%d", (z += x < y ? x++ : y++));

    x = 3; y = z = 4;
    printf ("%d", (z > y > x ? 1 : 0));
    printf ("%d", (z > y && y > x));
}
```

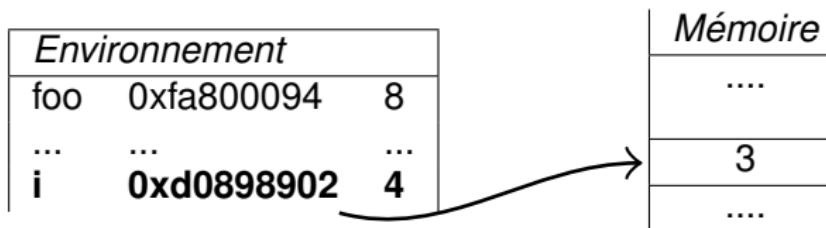
```
//  
//  
//  
  
// y = 2, x = 3  
// --> 3  
  
// --> 2,  
// y = 3, x = 3  
// --> 4  
// z = 4 (= 4 !)  
//  
// --> 0  
// --> 0  
//
```

Définir une variable est comme utiliser une consigne



Définir une variable est comme utiliser une consigne

- ▶ int i = 3; a trois effets :
 - ▶ déclare le nom `i` comme un identificateur entier
 - ▶ associe une zone de la mémoire de la taille d'un int au nom `i` : cette assignation est mémorisée dans l'*environnement*;
 - ▶ inscrit dans la zone de la mémoire assignée à `i` la valeur 3



- ▶ les trois actions peuvent être découplées
- ▶ pour retrouver ses affaires : &var contient l'adresse où est stockée la valeur de var
- ▶ pour combien de temps, le casier ?

La portée d'une définition

```
int i = 10;  
do {  
    int j = 10;  
    ...  
} while (test);  
// i existe toujours, j n'existe plus,
```

- ▶ par défaut une variable n'existe qu'au sein du bloc / de la fonction où elle est définie : c'est une variable *locale* ;
- ▶ à la fin du bloc, la variable disparaît et sa valeur est perdue
- ▶ pour « sauvegarder » les valeurs d'une variable : static
- ▶ une variable définie hors de toute fonction est visible dans tout le fichier : c'est une variable *globale* ;

La portée d'une définition

```
int i = 10;  
do {  
    int j = 10;  
    ...  
} while (test);  
// i existe toujours, j n'existe plus,
```

- ▶ par défaut une variable n'existe qu'au sein du bloc / de la fonction où elle est définie : c'est une variable *locale* ;
- ▶ à la fin du bloc, la variable disparaît et sa valeur est perdue
- ▶ pour « sauvegarder » les valeurs d'une variable : static
- ▶ une variable définie hors de toute fonction est visible dans tout le fichier : c'est une variable *globale* ;

La portée d'une définition

```
int i = 10;  
do {  
    int j = 10;  
    ...  
} while (test);  
// i existe toujours, j n'existe plus,
```

- ▶ par défaut une variable n'existe qu'au sein du bloc / de la fonction où elle est définie : c'est une variable *locale* ;
- ▶ à la fin du bloc, la variable disparaît et sa valeur est perdue
- ▶ pour « sauvegarder » les valeurs d'une variable : **static**
- ▶ une variable définie hors de toute fonction est visible dans tout le fichier : c'est une variable *globale* ;

La portée d'une définition

```
int i = 10;  
do {  
    int j = 10;  
    ...  
} while (test);  
// i existe toujours, j n'existe plus,
```

- ▶ par défaut une variable n'existe qu'au sein du bloc / de la fonction où elle est définie : c'est une variable *locale* ;
- ▶ à la fin du bloc, la variable disparaît et sa valeur est perdue
- ▶ pour « sauvegarder » les valeurs d'une variable : **static**
- ▶ une variable définie hors de toute fonction est visible dans tout le fichier : c'est une variable *globale* ;

Une variable peut en cacher une autre

```
int i = 3, j = 4;  
if (i > 0) {  
    int i = 2;  
    printf("i=%i\nj=%i\n", i, j);  
}  
printf("i=%i\n", i);
```

- ▶ qu'imprime le premier printf ? le second ?
- ▶ les variables sont recherchées dans l'environnement en commençant par la plus récente
- ▶ rajouter : printf ("j=%i\n", j); à la fin du programme provoque une erreur : j n'est plus connue.

Une variable peut en cacher une autre

```
int i = 3, j = 4;  
if (i > 0) {  
    int i = 2;  
    printf("i=%i\nj=%i\n", i, j);  
}  
printf("i=%i\n", i);
```

- ▶ qu'imprime le premier printf ? le second ?
- ▶ les variables sont recherchées dans l'environnement en commençant par la plus récente
- ▶ rajouter : printf ("j=%i\n", j); à la fin du programme provoque une erreur : j n'est plus connue.

Une variable peut en cacher une autre

```
int i = 3, j = 4;  
if (i > 0) {  
    int i = 2;  
    printf("i=%i\nj=%i\n", i, j);  
}  
printf("i=%i\n", i);
```

- ▶ qu'imprime le premier printf ? le second ?
- ▶ les variables sont recherchées dans l'environnement en commençant par la plus récente
- ▶ rajouter : printf ("j=%i\n", j); à la fin du programme provoque une erreur : j n'est plus connue.

Classes de stockage : des variables sur mesure

▶ play static

- ▶ il existe plusieurs mémoires
- ▶ **static int** possède deux sens :
 - ▶ dans un bloc, sauvegarde la valeur, *initialisation implicite à 0*
 - ▶ hors d'un bloc, évite « l'exportation »
- ▶ **auto int i** n'est pas **static** : *pas d'initialisation implicite*
- ▶ **register int i** accélère l'exécution des opérations (désuet)

Respirations

```
int i = 0;
int main() {
    auto int i = 1;
    printf("%d\n", i);
{
    int i = 2;
    printf("%d\n", i);
{
    i += 1;
    printf("%d\n", i);
}
    printf("%d\n", i);
}
printf("%d\n", i);
```

```
// i.1 = 0;
// i.2 = 1;
// → 1

// i.3 = 2;
// → 2

// i.3 = 3
// → 3

// → 3
// → 1
//
```

Respirations

```
int i = 0;
int main() {
    auto int i = 1;
    printf("%d\n", i);
{
    int i = 2;
    printf("%d\n", i);
{
    i += 1;
    printf("%d\n", i);
}
    printf("%d\n", i);
}
printf("%d\n", i);
```

```
// i.1 = 0;
// i.2 = 1;
// --> 1

// i.3 = 2;
// --> 2

// i.3 = 3
// --> 3

// --> 3

// --> 1
//
```

extern : une pure déclaration

▶ play extern

Le réserver à l'interaction entre fichiers

extern : une pure déclaration

▶ play extern

Le réserver à l'interaction entre fichiers

Qualifier des variables

```
const i = 1;    // une constante
const j;        // paradoxal, non ?
i = 5;          // interdit
```

- ▶ const int i interdit i = ... (cf. final en Java)
- ▶ le contraire de const : *mutable*
- ▶ une variable const doit être initialisée
- ▶ il reste pourtant possible de modifier une constante...
- ▶ volatile int i signale que i est modifiable par d'autres processus ; garantit un accès fiable mais non optimisé à l'adresse
- ▶ autre qualificatif (pour les pointeurs) restrict

Qualifier des variables

```
const i = 1;    // une constante
const j;        // paradoxal, non ?
i = 5;          // interdit
```

- ▶ const int i interdit i = ... (cf. final en Java)
- ▶ le contraire de const : *mutable*
- ▶ une variable const doit être initialisée
- ▶ il reste pourtant possible de modifier une constante...
- ▶ volatile int i signale que i est modifiable par d'autres processus ; garantit un accès fiable mais non optimisé à l'adresse
- ▶ autre qualificatif (pour les pointeurs) restrict

Qualifier des variables

```
const i = 1;    // une constante
const j;        // paradoxal, non ?
i = 5;          // interdit
```

- ▶ const int i interdit i = ... (cf. final en Java)
- ▶ le contraire de const : *mutable*
- ▶ une variable const doit être initialisée
- ▶ il reste pourtant possible de modifier une constante...
- ▶ volatile int i signale que i est modifiable par d'autres processus ; garantit un accès fiable mais non optimisé à l'adresse
- ▶ autre qualificatif (pour les pointeurs) restrict

Qualifier des variables

```
const i = 1;    // une constante
const j;        // paradoxal, non ?
i = 5;          // interdit
```

- ▶ const int i interdit i = ... (cf. final en Java)
- ▶ le contraire de const : *mutable*
- ▶ une variable const doit être initialisée
- ▶ il reste pourtant possible de modifier une constante...
- ▶ volatile int i signale que i est modifiable par d'autres processus ; garantit un accès fiable mais non optimisé à l'adresse
- ▶ autre qualificatif (pour les pointeurs) restrict

Qualifier des variables

```
const i = 1;    // une constante
const j;        // paradoxal, non ?
i = 5;          // interdit
```

- ▶ const int i interdit i = ... (cf. final en Java)
- ▶ le contraire de const : *mutable*
- ▶ une variable const doit être initialisée
- ▶ il reste pourtant possible de modifier une constante...
- ▶ volatile int i signale que i est modifiable par d'autres processus ; garantit un accès fiable mais non optimisé à l'adresse
- ▶ autre qualificatif (pour les pointeurs) restrict

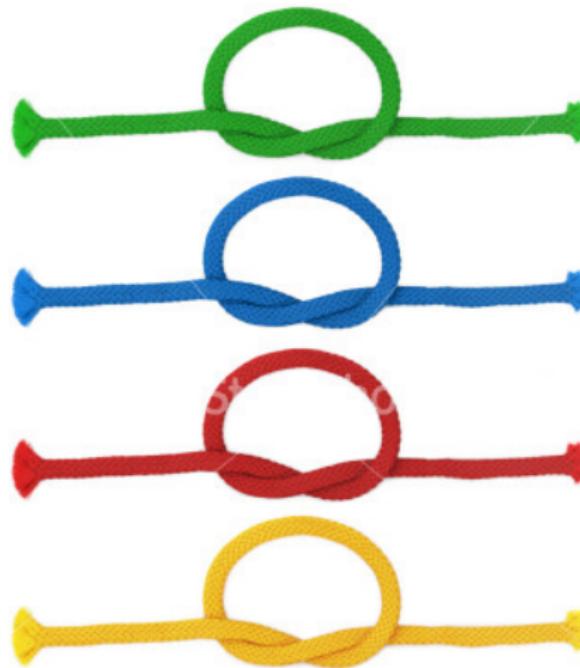
Qualifier des variables

```
const i = 1;    // une constante
const j;        // paradoxal, non ?
i = 5;          // interdit
```

- ▶ const int i interdit i = ... (cf. final en Java)
- ▶ le contraire de const : *mutable*
- ▶ une variable const doit être initialisée
- ▶ il reste pourtant possible de modifier une constante...
- ▶ volatile int i signale que i est modifiable par d'autres processus ; garantit un accès fiable mais non optimisé à l'adresse
- ▶ autre qualificatif (pour les pointeurs) restrict

Le(s) thème(s) du jour: Boucles et tableaux

La boucle, figure de l'itération



La boucle while

```
while (b)
```

```
  p;
```

```
while (b) {
```

```
  p;
```

```
  q;
```

```
}
```

- ▶ la condition b est évaluée de manière répétitive ;
- ▶ chaque fois que le résultat est non-nul, p est exécutée

La boucle while

```
while (b)
```

```
  p;
```

```
while (b) {
```

```
  p;
```

```
  q;
```

```
}
```

- ▶ la condition b est évaluée de manière répétitive ;
- ▶ chaque fois que le résultat est non-nul, p est exécutée

La boucle **while** (suite)

- ▶ littéralement :

```
if (b) {  
    p;  
    if (b) {  
        p;  
        if (b) {  
            ...  
        }  
    }  
}
```

- ▶ une boucle **while** équivaut à une série **arbitrairement longue** de tests imbriqués
- ▶ ⇒ attention aux *boucles sans fin while* (true) p;
- ▶ La valeur de b doit changer durant l'exécution du corps de la boucle

La boucle for

```
for ( i ; b ; e )  
    p ;
```

```
for ( d ; b ; e ) {  
    p ;  
    q ;  
}
```

- ▶ l'instruction *i* est exécuté *une fois* avant la boucle ;
- ▶ la condition *b* est testée *avant chaque itération* ;
- ▶ l'instruction *e* est exécutée *après chaque itération* ;
- ▶ littéralement :

```
i ;  
while ( b ) {  
    p ;  
    e ;  
}
```

La boucle for

```
for ( i ;b;e)
```

```
    p;
```

```
for ( d;b;e) {
```

```
    p;
```

```
    q;
```

```
}
```

- ▶ l'instruction i est exécuté *une fois* avant la boucle ;
- ▶ la condition b est testée *avant chaque* itération ;
- ▶ l'instruction e est exécutée *après chaque* itération ;
- ▶ littéralement :

```
i;
```

```
while (b) {
```

```
    p;
```

```
    e;
```

```
}
```

La boucle for

```
for ( i ;b;e )  
    p ;
```

```
for ( d;b;e ) {  
    p ;  
    q ;  
}
```

- ▶ l'instruction *i* est exécuté *une fois* avant la boucle ;
- ▶ la condition *b* est testée *avant chaque* itération ;
- ▶ l'instruction *e* est exécutée *après chaque itération* ;
- ▶ littéralement :

```
i ;  
while ( b ) {  
    p ;  
    e ;  
}
```

La boucle for

```
for ( i ;b;e )  
    p ;
```

```
for ( d;b;e ) {  
    p ;  
    q ;  
}
```

- ▶ l'instruction i est exécuté *une fois* avant la boucle ;
- ▶ la condition b est testée *avant chaque* itération ;
- ▶ l'instruction e est exécutée *après chaque itération* ;
- ▶ littéralement :

```
i ;  
while ( b ) {  
    p ;  
    e ;  
}
```

La boucle **for** par l'exemple

```
for (i=0; i<100 ; i = i + 1)
    if (i % 2 != 0)
        printf("%i est un nombre pair\n", i);

/* i++; <=> i = i + 1; */
int res = 1;
for (int i=1; i<10 ; i++)
    res = res*i;
```

Factorielle, avec un prompt

La boucle do

```
do {  
    p;  
}  
while (b)
```

- ▶ la condition b est évaluée *après* chaque exécution de p ;
- ▶ p est toujours exécutée *au moins une fois* ;
- ▶ littéralement :

```
p;  
while (b)  
    p;
```

La boucle do

```
do {  
    p;  
}  
while (b)
```

- ▶ la condition b est évaluée *après* chaque exécution de p ;
- ▶ p est toujours exécutée *au moins une fois* ;
- ▶ littéralement :

```
p;  
while (b)  
    p;
```

La boucle do

```
do {  
    p;  
}  
while (b)
```

- ▶ la condition b est évaluée *après* chaque exécution de p ;
- ▶ p est toujours exécutée *au moins une fois* ;
- ▶ littéralement :

```
p;  
while (b)  
    p;
```

Se prendre les pieds dans la boucle

- ▶ oublier d'initialiser ou d'incrémenter *i*

```
while ( i < 10 ) {  
    printf("hello world !\n");  
    i++;  
}
```

- ▶ se tromper dans le test

```
i = 0;  
while ( i > 10 ) { /* '>' ou lieu de '<' */  
    printf("hello world !\n");  
    i++;  
}
```

- ▶ confondre , et ;

```
for ( i=0, i < 10, i++ )  
    printf("hello world !\n");
```

Se prendre les pieds dans la boucle

- ▶ oublier d'initialiser ou d'incrémenter *i*

```
while ( i < 10 ) {  
    printf("hello world !\n");  
    i++;  
}
```

- ▶ se tromper dans le test

```
i=0;  
while ( i > 10 ) { /* ' > ' ou lieu de '<' */  
    printf("hello world !\n");  
    i++;  
}
```

- ▶ confondre , et ;

```
for ( i=0, i < 10, i++ )  
    printf("hello world !\n");
```

Se prendre les pieds dans la boucle

- ▶ oublier d'initialiser ou d'incrémenter *i*

```
while ( i < 10 ) {  
    printf("hello world !\n");  
    i++;  
}
```

- ▶ se tromper dans le test

```
i = 0;  
while ( i > 10 ) { /* ' > ' ou lieu de '<' */  
    printf("hello world !\n");  
    i++;  
}
```

- ▶ confondre , et ;

```
for ( i=0, i<10, i++ )  
    printf("hello world !\n");
```

Respirations

```

int x = 0, y = 0, z = 0;
while (y < 10) ++y; x += y;

x = y = 0;
while (y < 10) x += ++y;

y = 1;
while (y < 10) {
    x = y++; z = ++y;
}

for (y = 1; y < 10; y++)
    x = y;

for (y=1; (x=y) < 10; y++);
for (x=0, y=1000;
      y > 1; x++, y /= 10) ;

```

```

// y = 1 ... 10; x = 10

// x = y = 0
// x = 1 + 2 + ... 10 = 55

// y = 1
// y = 1, 3, 5, 7, 9, 11
// x = 1, 3 ... 9
// z = 3, 5 ... 11

// y = 1, 2 ... 10
// x = 9

// x = 1 ... 10

// y = 1000, 100, 10, 1
// x = 0, 1, 2, 3

```

Respirations

```

int x = 0, y = 0, z = 0;
while (y < 10) ++y; x += y;

x = y = 0;
while (y < 10) x += ++y;

y = 1;
while (y < 10) {
    x = y++; z = ++y;
}

for (y = 1; y < 10; y++)
    x = y;

for (y=1; (x=y) < 10; y++);
for (x=0, y=1000;
        y > 1; x++, y /= 10) ;

```

```

// y = 1 ... 10; x = 10
// x = y = 0
// x = 1 + 2 + ... 10 = 55

// y = 1
// y = 1, 3, 5, 7, 9, 11
// x = 1, 3 ... 9
// z = 3, 5 ... 11

// y = 1, 2 ... 10
// x = 9

// x = 1 ... 10
// y = 1000, 100, 10, 1
// x = 0, 1, 2, 3

```

Récréations

1. Écrivez un programme qui lit n entiers et affiche leur somme, leur produit et leur moyenne :
 - ▶ en utilisant **while**
 - ▶ en utilisant **do – while**,
 - ▶ en utilisant **for**
2. Écrire un programme qui dénombre et imprime toutes les manières d'obtenir exactement un euro avec des pièces de 2, 5, 10 centimes ;
3. Afficher un triangle isocèle avec le caractère '**', la taille est saisie sur l'entrée standard ;
4. Calculer le nombre d'or comme rapport de deux termes de la suite de Fibonacci ($u_n = u_{n-1} + u_{n-2}$) ; l'indice du dernier terme est lu sur la sortie courante ;
5. Écrire un programme testant la primalité du nombre saisi sur l'entrée courante (par le crible d'Erathostène)

Vecteurs et tableaux

TABLEAU DE LA NOMENCLATURE CHIMIQUE,
PROPOSÉE PAR MM. DE MORVEAU, LAVOISIER, BERTHOLET ET DE FOURCROY, le Mai 1787.

Page 100.

L. SUBSTANCES NON DÉCOMPOSÉES <small>NOMS COURTS ET DE BRUTS</small>	I. MISES A L'ÉTAT DE GAZ PAR LE CALORIQUE	III. COMBINÉES AVEC L'OXYGÈNE	IV. OXIGÈNES GAZEUSES	V. OXIGÈNES AVEC BASES	VI. COMBINÉES A L'ÉTAT D'ACIDE.
NOM ANCIEN <small>OU BRUT</small>	NOM ANCIEN <small>OU BRUT</small>	NOM ANCIEN <small>OU BRUT</small>	NOM ANCIEN <small>OU BRUT</small>	NOM ANCIEN <small>OU BRUT</small>	NOM ANCIEN <small>OU BRUT</small>
Géloque.	Chloro. Javelle, ou mat reine de la chaleur.	Gaz oxygène. Non. Il pa sse dans l'eau, et se d pose sur la partie en bas.			
Diglyc.	Ross de l'air sual.	Non. On distingue, au sual de l'air, deux sortes, l'une et la moins en état, qui sont des acids suals.			
Hydrog.	Ross de gaz inflammable.	Gaz hydrogène.	Ross.		
Jug. ou Butad sain que.	Ross de l'air inflammable ou de la moitié sual hydrog.	Gaz sualique.	Ross de gaz sualique. Gaz sualique, ou sual hydrog.	Ross sualique. Ross acide sualique.	Ross sualique.
Carbon. ou Radial sur bois.	Carbone pur.		Acide carbonique.	Acide de propane. Et ferre, Bic.	Carbonate de fer.
Sulfur. ou Radial sulf rite.			Acide sulfurique.		Sulfure de fer.
Phosphor. ou Radial phosphor.			Acide phosphorique.		Phosphore de fer.
Radic. mangan.			Acide manganique.		Manganate de fer.
Radic. borac.			Acide boracique.		Boracate de fer.
Radic. silice.			Acide silicie.		Silicate de fer.
Radic. selen.			Acide selenique.		Selenite de fer.
Radic. ferrique.			Acide ferrique.		Fer ferrique.
Radic. magnés.			Acide magnésique.		Magnétite de fer.
Radic. alcali.			Acide alcalin.		Alcaline de fer.
Radic. sapon.			Acide saponique.		Sapone de fer.
Radic. gélip.			Acide gélip.		Gélipate de fer.
Radic. sulfur.			Acide sulfurique.		Sulfure de fer.
Radic. iodique.			Acide iodique.		Iodure de fer.
Radic. selenite.			Acide selenitique.		Selenite de fer.
Radic. selenite.			Acide selenite.		Selenite de fer.
Radic. pyro-sulfur.			Acide pyrosulfurique.		Sulfure de fer.
Radic. pyro-selenite.			Acide pyroselenitique.		Selenite de fer.
Radic. iodite.			Acide ioditique.		Iodure de fer.
Radic. gélipite.			Acide gélipique.		Gélipate de fer.
Radic. chloro.			Acide chlorique.		Chlorate de fer.
Radic. bromo.			Acide bromique.		Bromate de fer.
Radic. iodato.			Acide iodato.		Iodate de fer.
Radic. chloro-iodato.			Acide chlоро-iodatique.		Chlоро-iodate de fer.
Radic. pyro-sulfate.			Acide pyrosulfatique.		Sulfate de fer.
Radic. pyro-selenite.			Acide pyroselenitique.		Selenite de fer.
Radic. pyro-iodite.			Acide pyro-ioditique.		Iodite de fer.
Radic. gélipate.			Acide gélipatique.		Gélipate de fer.
Radic. chloro-			Acide chloropatique.		Chloropatate de fer.
Radic. bromo-			Acide bromopatique.		Bromo-patate de fer.
Radic. iodato-			Acide iodopatique.		Iodo-patate de fer.
Radic. chlоро-iodato-			Acide chlоро-iodopatique.		Chlоро-iodopatate de fer.
Radic. pyro-sulfate.			Acide pyrosulfatopatique.		Sulfate de fer.
Radic. chlоро-iodato-			Acide chlоро-iodopatopatique.		Chlоро-iodopatate de fer.
Radic. chlоро-iodato-			Acide chlоро-iodopatopatopatique.		Chlоро-iodopatate de fer.
Radic. selenite.			Acide selenite-patique.		Selenite-patate de fer.
Radic. selenite.			Acide selenite-patopatique.		Selenite-patopatate de fer.
Radic. iodite.			Acide iodite-patique.		Iodite-patate de fer.
Radic. iodite.			Acide iodite-patopatique.		Iodite-patopatate de fer.
Radic. bromate.			Acide bromate-patique.		Bromo-patate de fer.
Radic. bromate.			Acide bromate-patopatique.		Bromo-patopatate de fer.
Radic. iodopatate.			Acide iodopatate-patique.		Iodo-patopatate de fer.
Radic. iodopatate.			Acide iodopatate-patopatique.		Iodo-patopatate de fer.
Radic. selenite-patate.			Acide selenite-patopatopatique.		Selenite-patopatopatate de fer.
Radic. selenite-patate.			Acide selenite-patopatopatopatique.		Selenite-patopatopatopatate de fer.
Radic. iodite-patate.			Acide iodite-patopatopatopatique.		Iodite-patopatopatopatate de fer.
Radic. iodite-patate.			Acide iodite-patopatopatopatopatique.		Iodite-patopatopatopatopatate de fer.
Radic. iodite-patopatate.			Acide iodite-patopatopatopatopatique.		Iodite-patopatopatopatopatate de fer.
Radic. iodite-patopatopatate.			Acide iodite-patopatopatopatopatopatique.		Iodite-patopatopatopatopatopatate de fer.

OXIDES AVEC DIVERSES BASES. (*)

Dioxide chloro-, ou chlor ate de fer.	Chlorate de fer.	Acidose assav. ou d' assav.	Acidose assav. ou d' assav.	Dioxide assav. ou d' assav.
Dioxide chloro- et chl oro-chloro- de fer.	Chlorate et chlorochlorate de fer.	Acidose assav. ou d' assav.	Acidose assav. ou d' assav.	Dioxide assav. ou d' assav.
Dioxide bromo-, ou brom ate de fer.	Bromo-bromate de fer.	Acidose assav. ou d' assav.	Acidose assav. ou d' assav.	Dioxide assav. ou d' assav.
Dioxide iodate de fer.	Iodate de fer.	Acidose assav. ou d' assav.	Acidose assav. ou d' assav.	Dioxide assav. ou d' assav.
Dioxide iodate-iodate de fer.	Iodate-iodate de fer.	Acidose assav. ou d' assav.	Acidose assav. ou d' assav.	Dioxide assav. ou d' assav.
Dioxide iodate-iodate-iodate de fer.	Iodate-iodate-iodate de fer.	Acidose assav. ou d' assav.	Acidose assav. ou d' assav.	Dioxide assav. ou d' assav.



Tableaux de taille fixe

```
int tab[5] = {1, 2, 3, 4, 5};
char s[6];
float f[20];
...
for (int i=0; i < 6; i++) s[i] = 'a';
printf("%c\n", s[5]);
```

- ▶ déclarer **int** tab[n] = ; associe le nom tab à une zone de *n* cellules *contigües* de taille *sizof* (**int**)
- ▶ les indices vont de 0 à *n* – 1 ; tab[n] n'existe pas

i =	0	1	2	3	4
tab[i] =	1	2	3	4	5

- ▶ la lecture ou l'affectation de tab[m] pour *m* > *n* n'est pas détectée lors de la compilation : cause de nombreuses erreurs
- ▶ les tableaux de scalaires représentent des vecteurs ou des séquences

Tableaux de taille fixe

```
int tab[5] = {1, 2, 3, 4, 5};
char s[6];
float f[20];
...
for (int i=0; i < 6; i++) s[i] = 'a';
printf("%c\n", s[5]);
```

- déclarer **int** tab[n] = ; associe le nom tab à une zone de n cellules *contigües* de taille `sizof(int)`
- les indices vont de 0 à $n - 1$; tab[n] n'existe pas

i =	0	1	2	3	4
tab[i] =	1	2	3	4	5

- la lecture ou l'affectation de tab[m] pour $m > n$ n'est pas détectée lors de la compilation : cause de nombreuses erreurs
- les tableaux de scalaires représentent des vecteurs ou des séquences

Tableaux de taille fixe

```
int tab[5] = {1, 2, 3, 4, 5};
char s[6];
float f[20];
...
for (int i=0; i < 6; i++) s[i] = 'a';
printf("%c\n", s[5]);
```

- ▶ déclarer int tab[n] = ; associe le nom tab à une zone de n cellules *contigües* de taille sizeof(int)
- ▶ les indices vont de 0 à $n - 1$; tab[n] n'existe pas

i =	0	1	2	3	4
tab[i] =	1	2	3	4	5

- ▶ la lecture ou l'affectation de tab[m] pour $m > n$ n'est pas détectée lors de la compilation : cause de nombreuses erreurs
- ▶ les tableaux de scalaires représentent des vecteurs ou des séquences

Tableaux de taille fixe

```
int tab[5] = {1, 2, 3, 4, 5};
char s[6];
float f[20];
...
for (int i=0; i < 6; i++) s[i] = 'a';
printf("%c\n", s[5]);
```

- ▶ déclarer **int** tab[n] = ; associe le nom tab à une zone de n cellules *contigües* de taille `sizof(int)`
- ▶ les indices vont de 0 à $n - 1$; tab[n] n'existe pas

i =	0	1	2	3	4
tab[i] =	1	2	3	4	5

- ▶ la lecture ou l'affectation de tab[m] pour $m > n$ n'est pas détectée lors de la compilation : cause de nombreuses erreurs
- ▶ les tableaux de scalaires représentent des vecteurs ou des séquences

Tableaux à double entrée : matrices

```
int m[5][5] = {{0, 0, 0, 0, 0},  
                {1, 1, 1, 1, 1},  
                {2, 2, 2, 2, 2},  
                {3, 3, 3, 3, 3},  
                {4, 4, 4, 4, 4}};  
  
for (i=0; i < 5; i++) {  
    for (j=0; j < 5; j++)  
        printf("%i ", m[i][j]);  
    printf("\n");  
}
```

- ▶ **float** m[5][5] construit un tableau de tableaux
- ▶ l'initialisation se fait ligne par ligne
- ▶ cette construction se généralise : matrices à 3, 4... dimensions

Les tableaux dynamiques : une amélioration récente

```
#include<stdlib.h> // pour random()
#include<stdint.h> // pour INT32_MAX
...
int n;
scanf("%d", &n);
if (n > 0) {
    float tab[n];
    for (int i=1; i< n; i++)
        tab[i] = (float)rand() / float(INT32_MAX);
    // traite le tableau
}
```

- ▶ construction interdite avant C99
- ▶ n ne sera connu qu'à la compilation
- ▶ tab[n] est traité comme une variable locale
- ▶ tab[10] = {[2] = 5, [4]=78} est aussi nouveau

Les tableaux dynamiques : une amélioration récente

```
#include<stdlib.h> // pour random()
#include<stdint.h> // pour INT32_MAX
...
int n;
scanf("%d", &n);
if (n > 0) {
    float tab[n];
    for (int i=1; i< n; i++)
        tab[i] = (float)rand() / float(INT32_MAX);
    // traite le tableau
}
```

- ▶ construction interdite avant C99
- ▶ `n` ne sera connu qu'à la compilation
- ▶ `tab[n]` est traité comme une variable locale
- ▶ `tab[10] = {[2] = 5, [4]=78}` est aussi nouveau

Les tableaux dynamiques : une amélioration récente

```
#include<stdlib.h> // pour random()
#include<stdint.h> // pour INT32_MAX
...
int n;
scanf("%d", &n);
if (n > 0) {
    float tab[n];
    for (int i=1; i< n; i++)
        tab[i] = (float)rand() / float(INT32_MAX);
    // traite le tableau
}
```

- ▶ construction interdite avant C99
- ▶ n ne sera connu qu'à la compilation
- ▶ tab[n] est traité comme une variable locale
- ▶ tab[10] = {[2] = 5, [4]=78} est aussi nouveau

Les tableaux dynamiques : une amélioration récente

```
#include<stdlib.h> // pour random()
#include<stdint.h> // pour INT32_MAX
...
int n;
scanf("%d", &n);
if (n > 0) {
    float tab[n];
    for (int i=1; i< n; i++)
        tab[i] = (float)rand() / float(INT32_MAX);
    // traite le tableau
}
```

- ▶ construction interdite avant C99
- ▶ n ne sera connu qu'à la compilation
- ▶ tab[n] est traité comme une variable locale
- ▶ tab[10] = {[2] = 5, [4]=78} est aussi nouveau

Récréations

1. Construire un tableau triangulaire dont les éléments sont les coefficients du binôme
2. On loge une matrice 2×2 dans un vecteur de longueur 4 : prendre deux tels vecteurs A et B et calculer $A \times B$ dans une nouvelle matrice.
3. Transposez une matrice $n \times n$ « sur place »
4. Prendre un tableau d'entiers et :
 - annuler les k derniers éléments
 - supprimer les k premiers éléments (et décaler les autres)
 - supprime l éléments à partir du k^{e} (avec décalage)
5. Représenter un polynôme $P = a_n x^n + a_{n-1} x^{n-1} + \dots + a_0$ du n^{e} degré dans un tableau de taille $n+1$, puis lire x et calculer $P(x)$:
 - naïvement
 - par la méthode de Horner → construire H , $H[n] = a_n$,
 $H[i] = a_i + H[i+1]x$, $P(x) = H[0] = a_0 + H[1]x$

Récréations

1. Construire un tableau triangulaire dont les éléments sont les coefficients du binôme
2. On loge une matrice 2×2 dans un vecteur de longueur 4 : prendre deux tels vecteurs A et B et calculer $A \times B$ dans une nouvelle matrice.
3. Transposez une matrice $n \times n$ « sur place »
4. Prendre un tableau d'entiers et :
 - annuler les k derniers éléments
 - supprimer les k premiers éléments (et décaler les autres)
 - supprime l éléments à partir du k^{e} (avec décalage)
5. Représenter un polynôme $P = a_nx^n + a_{n-1}x^{n-1} + \dots + a_0$ du n^{e} degré dans un tableau de taille $n+1$, puis lire x et calculer $P(x)$:
 - naïvement
 - par la méthode de Horner → construire H , $H[n] = a_n$,
 $H[i] = a_i + H[i+1]x$, $P(x) = H[0] = a_0 + H[1]x$

Récréations

1. Construire un tableau triangulaire dont les éléments sont les coefficients du binôme
2. On loge une matrice 2×2 dans un vecteur de longueur 4 : prendre deux tels vecteurs A et B et calculer $A \times B$ dans une nouvelle matrice.
3. Transposez une matrice $n \times n$ « sur place »
4. Prendre un tableau d'entiers et :
 - annuler les k derniers éléments
 - supprimer les k premiers éléments (et décaler les autres)
 - supprime l éléments à partir du k^{e} (avec décalage)
5. Représenter un polynôme $P = a_n x^n + a_{n-1} x^{n-1} + \dots + a_0$ du n^{e} degré dans un tableau de taille $n+1$, puis lire x et calculer $P(x)$:
 - naïvement
 - par la méthode de Horner → construire H , $H[n] = a_n$,
 $H[i] = a_i + H[i+1]x$, $P(x) = H[0] = a_0 + H[1]x$

Récréations

1. Construire un tableau triangulaire dont les éléments sont les coefficients du binôme
2. On loge une matrice 2×2 dans un vecteur de longueur 4 : prendre deux tels vecteurs A et B et calculer $A \times B$ dans une nouvelle matrice.
3. Transposez une matrice $n \times n$ « sur place »
4. Prendre un tableau d'entiers et :
 - ▶ annuler les k derniers éléments
 - ▶ supprimer les k premiers éléments (et décaler les autres)
 - ▶ supprime l éléments à partir du k^e (avec décalage)
5. Représenter un polynôme $P = a_nx^n + a_{n-1}x^{n-1} + \dots + a_0$ du n^e degré dans un tableau de taille $n+1$, puis lire x et calculer $P(x)$:
 - ▶ naïvement
 - ▶ par la méthode de Horner → construire H , $H[n] = a_n$,
 $H[i] = a_i + H[i+1]x$, $P(x) = H[0] = a_0 + H[1]x$

Récréations

1. Construire un tableau triangulaire dont les éléments sont les coefficients du binôme
2. On loge une matrice 2×2 dans un vecteur de longueur 4 : prendre deux tels vecteurs A et B et calculer $A \times B$ dans une nouvelle matrice.
3. Transposez une matrice $n \times n$ « sur place »
4. Prendre un tableau d'entiers et :
 - ▶ annuler les k derniers éléments
 - ▶ supprimer les k premiers éléments (et décaler les autres)
 - ▶ supprime l éléments à partir du k^e (avec décalage)
5. Représenter un polynôme $P = a_nx^n + a_{n-1}x^{n-1} + \dots + a_0$ du n^e degré dans un tableau de taille $n+1$, puis lire x et calculer $P(x)$:
 - ▶ naïvement
 - ▶ par la méthode de Horner → construire H , $H[n] = a_n$,
 $H[i] = a_i + H[i+1]x$, $P(x) = H[0] = a_0 + H[1]x$

Récréations

1. Construire un tableau triangulaire dont les éléments sont les coefficients du binôme
2. On loge une matrice 2×2 dans un vecteur de longueur 4 : prendre deux tels vecteurs A et B et calculer $A \times B$ dans une nouvelle matrice.
3. Transposez une matrice $n \times n$ « sur place »
4. Prendre un tableau d'entiers et :
 - ▶ annuler les k derniers éléments
 - ▶ supprimer les k premiers éléments (et décaler les autres)
 - ▶ supprime l éléments à partir du k^e (avec décalage)
5. Représenter un polynôme $P = a_nx^n + a_{n-1}x^{n-1} + \dots + a_0$ du n^e degré dans un tableau de taille $n+1$, puis lire x et calculer $P(x)$:
 - ▶ naïvement
 - ▶ par la méthode de Horner → construire H , $H[n] = a_n$,
 $H[i] = a_i + H[i+1]x$, $P(x) = H[0] = a_0 + H[1]x$

Récréations

1. Construire un tableau triangulaire dont les éléments sont les coefficients du binôme
2. On loge une matrice 2×2 dans un vecteur de longueur 4 : prendre deux tels vecteurs A et B et calculer $A \times B$ dans une nouvelle matrice.
3. Transposez une matrice $n \times n$ « sur place »
4. Prendre un tableau d'entiers et :
 - ▶ annuler les k derniers éléments
 - ▶ supprimer les k premiers éléments (et décaler les autres)
 - ▶ supprime l éléments à partir du k^e (avec décalage)
5. Représenter un polynôme $P = a_nx^n + a_{n-1}x^{n-1} + \dots + a_0$ du n^e degré dans un tableau de taille $n+1$, puis lire x et calculer $P(x)$:
 - ▶ naïvement
 - ▶ par la méthode de Horner → construire H , $H[n] = a_n$,
 $H[i] = a_i + H[i+1]x$, $P(x) = H[0] = a_0 + H[1]x$

Récréations

1. Construire un tableau triangulaire dont les éléments sont les coefficients du binôme
2. On loge une matrice 2×2 dans un vecteur de longueur 4 : prendre deux tels vecteurs A et B et calculer $A \times B$ dans une nouvelle matrice.
3. Transposez une matrice $n \times n$ « sur place »
4. Prendre un tableau d'entiers et :
 - ▶ annuler les k derniers éléments
 - ▶ supprimer les k premiers éléments (et décaler les autres)
 - ▶ supprime l éléments à partir du k^e (avec décalage)
5. Représenter un polynôme $P = a_nx^n + a_{n-1}x^{n-1} + \dots + a_0$ du n^e degré dans un tableau de taille $n+1$, puis lire x et calculer $P(x)$:
 - ▶ naïvement
 - ▶ par la méthode de Horner → construire H , $H[n] = a_n$, $H[i] = a_i + H[i+1]x$, $P(x) = H[0] = a_0 + H[1]x$

Récréations

1. Construire un tableau triangulaire dont les éléments sont les coefficients du binôme
2. On loge une matrice 2×2 dans un vecteur de longueur 4 : prendre deux tels vecteurs A et B et calculer $A \times B$ dans une nouvelle matrice.
3. Transposez une matrice $n \times n$ « sur place »
4. Prendre un tableau d'entiers et :
 - ▶ annuler les k derniers éléments
 - ▶ supprimer les k premiers éléments (et décaler les autres)
 - ▶ supprime l éléments à partir du k^e (avec décalage)
5. Représenter un polynôme $P = a_nx^n + a_{n-1}x^{n-1} + \dots + a_0$ du n^e degré dans un tableau de taille $n+1$, puis lire x et calculer $P(x)$:
 - ▶ naïvement
 - ▶ par la méthode de Horner → construire H , $H[n] = a_n$, $H[i] = a_i + H[i+1]x$, $P(x) = H[0] = a_0 + H[1]x$

Récréations

1. Construire un tableau triangulaire dont les éléments sont les coefficients du binôme
2. On loge une matrice 2×2 dans un vecteur de longueur 4 : prendre deux tels vecteurs A et B et calculer $A \times B$ dans une nouvelle matrice.
3. Transposez une matrice $n \times n$ « sur place »
4. Prendre un tableau d'entiers et :
 - ▶ annuler les k derniers éléments
 - ▶ supprimer les k premiers éléments (et décaler les autres)
 - ▶ supprime l éléments à partir du k^e (avec décalage)
5. Représenter un polynôme $P = a_nx^n + a_{n-1}x^{n-1} + \dots + a_0$ du n^e degré dans un tableau de taille $n+1$, puis lire x et calculer $P(x)$:
 - ▶ naïvement
 - ▶ par la méthode de Horner → construire H , $H[n] = a_n$, $H[i] = a_i + H[i+1]x$, $P(x) = H[0] = a_0 + H[1]x$

Les chaînes de caractères : des tableaux particuliers

```
char nom [10] = "Orsay";  
  
for (i=0; i < 10; i++)  
    printf("%c/%i_ ", nom[i], nom[i]);  
    printf("\n");
```

- ▶ une chaîne est un tableau *interprété jusqu'au premier (char)0* ;

0	1	2	3	4	5	6	7	8	9
'O'	'r'	's'	'a'	'y'	0	??	??	??	??

- ▶ la longueur d'une chaîne \neq le nombre de cellules du tableau
- ▶ une chaîne de n caractères demande $n + 1$ cellules

Les chaînes de caractères : des tableaux particuliers

```
char nom [10] = "Orsay";  
  
for (i=0; i < 10; i++)  
    printf("%c/%i_ ", nom[i], nom[i]);  
    printf("\n");
```

- ▶ une chaîne est un tableau *interprété jusqu'au premier (char)0* ;

0	1	2	3	4	5	6	7	8	9
'O'	'r'	's'	'a'	'y'	0	??	??	??	??

- ▶ la longueur d'une chaîne \neq le nombre de cellules du tableau
- ▶ une chaîne de n caractères demande $n + 1$ cellules

Les chaînes de caractères : des tableaux particuliers

```
char nom [10] = "Orsay";  
  
for (i=0; i < 10; i++)  
    printf("%c/%i_ ", nom[i], nom[i]);  
    printf("\n");
```

- ▶ une chaîne est un tableau *interprété jusqu'au premier (char)0* ;

0	1	2	3	4	5	6	7	8	9
'O'	'r'	's'	'a'	'y'	0	??	??	??	??

- ▶ la longueur d'une chaîne \neq le nombre de cellules du tableau
- ▶ une chaîne de n caractères demande $n + 1$ cellules

Travailler la chaîne...

```
int length = 0;  
char nom1 [10] = "Orsay";  
char nom2 [10];  
  
length = strlen(nom2);  
printf("La longueur de '%s' est %i\n", nom2, length);  
strcpy(nom1, nom2);
```

- ▶ les constantes sont entre " ; format de lecture/écriture : %s
- ▶ lecture formatée : char nom [10]; scanf("%s", nom); [sans & !!]
- ▶ les opérations élémentaires dans #include<strings.h>
 - ▶ copie : strcpy
 - ▶ concaténation : strcat
 - ▶ comparaison :strcmp(), strncmp()
 - ▶ recherche de caractères (strchr) et de facteurs (strstr)
- ▶ de nombreux langages (C++, Java,...) définissent un type string et des opérations associées (y.c le matching)

Travailler la chaîne...

```
int length = 0;  
char nom1 [10] = "Orsay";  
char nom2 [10];  
  
length = strlen(nom2);  
printf("La longueur de '%s' est %i\n", nom2, length);  
strcpy(nom1, nom2);
```

- ▶ les constantes sont entre " ; format de lecture/écriture : %s
- ▶ **lecture formatée : `char nom [10]; scanf("%s", nom);` [sans & !!]**
- ▶ les opérations élémentaires dans `#include<strings.h>`
 - ▶ copie : `strcpy`
 - ▶ concaténation : `strcat`
 - ▶ comparaison : `strcmp()`, `strncmp()`
 - ▶ recherche de caractères (`strchr`) et de facteurs (`strstr`)
- ▶ de nombreux langages (C++, Java,...) définissent un type string et des opérations associées (y.c le matching)

Travailler la chaîne...

```
int length = 0;  
char nom1 [10] = "Orsay";  
char nom2 [10];  
  
length = strlen(nom2);  
printf("La longueur de '%s' est %i\n", nom2, length);  
strcpy(nom1, nom2);
```

- ▶ les constantes sont entre " ; format de lecture/écriture : %s
- ▶ lecture formatée : **char** nom [10]; scanf("%s", nom); [sans & !!]
- ▶ les opérations élémentaires dans **#include<strings.h>**
 - ▶ copie : strcpy
 - ▶ concaténation : strcat
 - ▶ comparaison :strcmp(), strncmp()
 - ▶ recherche de caractères (strchr) et de facteurs (strstr)
- ▶ de nombreux langages (C++, Java,...) définissent un type string et des opérations associées (y.c le matching)

Travailler la chaîne...

```
int length = 0;  
char nom1 [10] = "Orsay";  
char nom2 [10];  
  
length = strlen(nom2);  
printf("La longueur de '%s' est %i\n", nom2, length);  
strcpy(nom1, nom2);
```

- ▶ les constantes sont entre " ; format de lecture/écriture : %s
- ▶ lecture formatée : **char** nom [10]; scanf("%s", nom); [sans & !!]
- ▶ les opérations élémentaires dans **#include<strings.h>**
 - ▶ copie : **strcpy**
 - ▶ concaténation : **strcat**
 - ▶ comparaison :**strcmp()**, **strncmp()**
 - ▶ recherche de caractères (**strchr**) et de facteurs (**strstr**)
- ▶ de nombreux langages (C++, Java,...) définissent un type string et des opérations associées (y.c le matching)

Travailler la chaîne...

```
int length = 0;  
char nom1 [10] = "Orsay";  
char nom2 [10];  
  
length = strlen(nom2);  
printf("La longueur de '%s' est %i\n", nom2, length);  
strcpy(nom1, nom2);
```

- ▶ les constantes sont entre " ; format de lecture/écriture : %s
- ▶ lecture formatée : **char** nom [10]; scanf("%s", nom); [sans & !!]
- ▶ les opérations élémentaires dans **#include<strings.h>**
 - ▶ copie : strcpy
 - ▶ concaténation : strcat
 - ▶ comparaison :strcmp(), strncmp()
 - ▶ recherche de caractères (strchr) et de facteurs (strstr)
- ▶ de nombreux langages (C++, Java,...) définissent un type string et des opérations associées (y.c le matching)

Travailler la chaîne...

```
int length = 0;  
char nom1 [10] = "Orsay";  
char nom2 [10];  
  
length = strlen(nom2);  
printf("La longueur de '%s' est %i\n", nom2, length);  
strcpy(nom1, nom2);
```

- ▶ les constantes sont entre " ; format de lecture/écriture : %s
- ▶ lecture formatée : **char** nom [10]; scanf("%s", nom); [sans & !!]
- ▶ les opérations élémentaires dans **#include<strings.h>**
 - ▶ copie : strcpy
 - ▶ concaténation : strcat
 - ▶ **comparaison :strcmp(), strncmp()**
 - ▶ recherche de caractères (strchr) et de facteurs (strstr)
- ▶ de nombreux langages (C++, Java,...) définissent un type string et des opérations associées (y.c le matching)

Travailler la chaîne...

```
int length = 0;  
char nom1 [10] = "Orsay";  
char nom2 [10];  
  
length = strlen(nom2);  
printf("La longueur de '%s' est %i\n", nom2, length);  
strcpy(nom1, nom2);
```

- ▶ les constantes sont entre " ; format de lecture/écriture : %s
- ▶ lecture formatée : **char** nom [10]; scanf("%s", nom); [sans & !!]
- ▶ les opérations élémentaires dans **#include<strings.h>**
 - ▶ copie : strcpy
 - ▶ concaténation : strcat
 - ▶ comparaison :strcmp(), strncmp()
 - ▶ recherche de caractères (strchr) et de facteurs (strstr)
- ▶ de nombreux langages (C++, Java,...) définissent un type string et des opérations associées (y.c le matching)

Travailler la chaîne...

```
int length = 0;  
char nom1 [10] = "Orsay";  
char nom2 [10];  
  
length = strlen(nom2);  
printf("La longueur de '%s' est %i\n", nom2, length);  
strcpy(nom1, nom2);
```

- ▶ les constantes sont entre " ; format de lecture/écriture : %s
- ▶ lecture formatée : **char** nom [10]; scanf("%s", nom); [sans & !!]
- ▶ les opérations élémentaires dans **#include<strings.h>**
 - ▶ copie : strcpy
 - ▶ concaténation : strcat
 - ▶ comparaison :strcmp(), strncmp()
 - ▶ recherche de caractères (strchr) et de facteurs (strstr)
- ▶ de nombreux langages (C++, Java,...) définissent un type string et des opérations associées (y.c le matching)

Respirations

```
int main() {  
    char adr [] = "bonjour";  
    for (int i=0 ; i<3 ; i++) putchar(adr[i]);  
    printf ("\n");  
    i = 0;  
    while (adr[i]) putchar(adr[i++]);  
}
```

```
int main() {  
    char line[100];  
    while(gets(line)) {  
        while (int j=0; line[j] && line[j] != '\t'; j++);  
        if line([j] == '\t') {  
            line[j] = 0;  
            printf("Avant TAB=%s\n", line);  
        }  
    }  
}
```

Respirations

```
int main() {  
    char adr [] = "bonjour";  
    for (int i=0 ; i<3 ; i++) putchar(adr[i]);  
    printf ("\n");  
    i = 0;  
    while (adr[i]) putchar(adr[i++]);  
}
```

```
int main() {  
    char line[100];  
    while(gets(line)) {  
        while (int j=0; line[j] && line[j] != '\t'; j++);  
        if line([j] == '\t') {  
            line[j] = 0;  
            printf("Avant TAB=%s\n", line);  
        }  
    }  
}
```

Récréations

- ▶ Écrire un programme qui teste si un mot est un palindrome
- ▶ Écrire un programme qui conjugue les verbes réguliers du premier groupe
- ▶ Écrire un programme qui convertit une chaîne représentant un nombre en écriture décimale en entier
- ▶ Idem pour un nombre en chiffres romains

Récréations

- ▶ Écrire un programme qui teste si un mot est un palindrome
- ▶ Écrire un programme qui conjugue les verbes réguliers du premier groupe
- ▶ Écrire un programme qui convertit une chaîne représentant un nombre en écriture décimale en entier
- ▶ Idem pour un nombre en chiffres romains

Récréations

- ▶ Écrire un programme qui teste si un mot est un palindrome
- ▶ Écrire un programme qui conjugue les verbes réguliers du premier groupe
- ▶ Écrire un programme qui convertit une chaîne représentant un nombre en écriture décimale en entier
- ▶ Idem pour un nombre en chiffres romains

Récréations

- ▶ Écrire un programme qui teste si un mot est un palindrome
- ▶ Écrire un programme qui conjugue les verbes réguliers du premier groupe
- ▶ Écrire un programme qui convertit une chaîne représentant un nombre en écriture décimale en entier
- ▶ Idem pour un nombre en chiffres romains

Compléments sur les chaînes

- ▶ Des types et fonctions pour les caractères « étendus » :
 - ▶ wchar_t (16 ou 32 bits) : wide chars !
 - ▶ des fonctions de conversions vers et depuis les char (dans <stdlib.h>)
- ▶ Des bibliothèques pour les expressions régulières :
 - ▶ Gnu Regexp (dans la libC) :
 - <http://www.gnu.org/software/libc/>
 - ▶ www.pcre.org

Compléments sur les chaînes

- ▶ Des types et fonctions pour les caractères « étendus » :
 - ▶ `wchar_t` (16 ou 32 bits) : wide chars !
 - ▶ des fonctions de conversions vers et depuis les `char` (dans `<stdlib.h>`)
- ▶ Des bibliothèques pour les expressions régulières :
 - ▶ Gnu Regexp (dans la libC) :
<http://www.gnu.org/software/libc/>
 - ▶ www.pcre.org

Compléments sur les chaînes

- ▶ Des types et fonctions pour les caractères « étendus » :
 - ▶ wchar_t (16 ou 32 bits) : wide chars !
 - ▶ des fonctions de conversions vers et depuis les char (dans <stdlib.h>)
- ▶ Des bibliothèques pour les expressions régulières :
 - ▶ Gnu Regexp (dans la libC) :
<http://www.gnu.org/software/libc/>
 - ▶ www.pcre.org

Compléments sur les chaînes

- ▶ Des types et fonctions pour les caractères « étendus » :
 - ▶ wchar_t (16 ou 32 bits) : wide chars !
 - ▶ des fonctions de conversions vers et depuis les char (dans <stdlib.h>)
- ▶ Des bibliothèques pour les expressions régulières :
 - ▶ Gnu Regexp (dans la libC) :
<http://www.gnu.org/software/libc/>
 - ▶ www.pcre.org

Compléments sur les chaînes

- ▶ Des types et fonctions pour les caractères « étendus » :
 - ▶ wchar_t (16 ou 32 bits) : wide chars !
 - ▶ des fonctions de conversions vers et depuis les char (dans <stdlib.h>)
- ▶ Des bibliothèques pour les expressions régulières :
 - ▶ Gnu Regexp (dans la libC) :
<http://www.gnu.org/software/libc/>
 - ▶ www.pcre.org

Compléments sur les chaînes

- ▶ Des types et fonctions pour les caractères « étendus » :
 - ▶ wchar_t (16 ou 32 bits) : wide chars !
 - ▶ des fonctions de conversions vers et depuis les char (dans <stdlib.h>)
- ▶ Des bibliothèques pour les expressions régulières :
 - ▶ Gnu Regexp (dans la libC) :
<http://www.gnu.org/software/libc/>
 - ▶ www.pcre.org

Stop ou encore ? **break** et **continue**

```
const char Orsay[6] = "Orsay";
// recherche linéaire dans un tableau trié
found = 0;
for (i=1; i < nstring; i++) {
    if (Dictionnaire[i][0] != 'o') continue;
    if (strcmp(Tab[i], Orsay == 0) {
        found = 1;
        break;
    }
}
```

- ▶ **continue** force le passage à l'itération suivante de la boucle ;
- ▶ ce qui suit **continue** est ignoré ;
- ▶ `strcmp(const char *, const char *)` (dans `<string.h>`) compare deux chaînes de caractères ; renvoie 0 (faux) quand les chaînes sont semblables !!
- ▶ **break** force à terminer la boucle courante ;
- ▶ l'exécution reprend juste après la fin de la boucle.

goto 10 : un héritage embarrassant

```
1 // if / else comme en fortran
2 if (! test) goto testfailed;
3 // ce qui suit quand test est vrai
4 ...
5 goto resume;
6 testfailed:
7 // ce qui suit quand test est faux
8 ...
9 resume:
10 // le calcul reprend
```

- ▶ **goto** label; repositionne à partir de *l'étiquette* label:
- ▶ fabrication d'une boucle :

```
loop:
// corps de la boucle
...
if (test) goto loop;
```

- ▶ source majeure d'illisibilité : à proscrire.

Respirations

```
char input[] = "SSWLEC1\1\11W\1WALLMP1";  
  
int i; char c;  
for(i=2; (c=input[i]) != '\0'; i++) {  
    switch(c) {  
        case 'a': putchar('i'); continue;  
        case '1': break;  
        case 1:  
            while( (c=[input[i++]] != '\1'  
                  && c != '\0'));  
        case 9: putchar('s');  
        case 'E':  
        case 'L': continue;  
        default: putchar(c); continue;  
    }  
    putchar('_');  
}  
putchar('\n');
```

```
// i=2, S  
// i=3, S  
// i=4, W  
// i=5,  
// i=6,  
// i=7, C  
// i=8, _  
// i=9-12 S  
// i=13 W  
// i=14 A  
// i=15  
// i=16  
// i=17 M  
// i=18 P  
// i=19 _  
// \n  
//
```

Respirations

```

char input[] = "SSWLEC1\1\11W\1WALLMP1";

int i; char c;
for(i=2; (c=input[i]) != '\0'; i++) {
    switch(c) {
        case 'a': putchar('i'); continue;
        case '1': break;
        case 1:
            while( (c=[input[i++]] != '1'
                      && c != '\0'));
        case 9: putchar('s');
        case 'E':
        case 'L': continue;
        default: putchar(c); continue;
    }
    putchar('_');
}
putchar('\n');

```

```

// i=2, S
// i=3, S
// i=4, W
// i=5,
// i=6,
// i=7, C
// i=8, -
// i=9-12 S
// i=13 W
// i=14 A
// i=15
// i=16
// i=17 M
// i=18 P
// i=19 -
// \n
//
//

```

Respirations (A. Feuer)

```
while(A) {  
    if (B) continue;  
    C;  
}
```

```
while(A) {  
    if (! B)  
    C;  
}
```

```
if (A)  
    if (B)  
        if (C) D;  
        else ;  
    else ;  
    else if (B)  
        if (C) E;  
        else F;
```

```
if (B)  
    if (A && C) D;  
    else if (!A && C) E;  
    else if (!A && !C) F;  
    else ;  
}
```

Respirations (A. Feuer)

```
while(A) {  
    if (B) continue;  
    C;  
}
```

```
while(A) {  
    if (! B)  
    C;  
}
```

```
if (A)  
    if (B)  
        if (C) D;  
        else ;  
    else ;  
    else if (B)  
        if (C) E;  
        else F;
```

```
if (B)  
    if (A && C) D;  
    else if (!A && C) E;  
    else if (!A && !C) F;  
    else ;  
}
```

Respirations (A. Feuer)

```
while(A) {  
    if (B) continue;  
    C;  
}
```

```
while(A) {  
    if (! B)  
    C;  
}
```

```
if (A)  
    if (B)  
        if (C) D;  
        else ;  
    else ;  
    else if (B)  
        if (C) E;  
        else F;
```

```
if (B)  
    if (A && C) D;  
    else if (!A && C) E;  
    else if (!A && !C) F;  
    else ;  
}
```

Respirations (A. Feuer)

```
while(A) {  
    if (B) continue;  
    C;  
}
```

```
while(A) {  
    if (! B)  
    C;  
}
```

```
if (A)  
    if (B)  
        if (C) D;  
        else ;  
    else ;  
    else if (B)  
        if (C) E;  
        else F;
```

```
if (B)  
    if (A && C) D;  
    else if (!A && C) E;  
    else if (!A && !C) F;  
    else ;  
}
```

Respirations

```
do {  
    if (!A) continue;  
    else B;  
    C;  
} while (A);
```

```
while(A) {  
    B;  
    C;  
}
```

Respirations

```
do {  
    if (!A) continue;  
    else B;  
    C;  
} while (A);
```

```
while(A) {  
    B;  
    C;  
}
```

Le(s) thème(s) du jour: Fonctionnement des fonctions

Nouvel éloge de la paresse

```
// imprime dans un cadre [V1.0]
printf("+-+-+\n");
printf("|     Solutions     |\n");
printf("+-+-+\n");
// Les solutions
// ...
printf("+-+-+\n");
```

```
// imprime dans un cadre [V2.0]
void printline(void) { // <-- Definit la procedure
    printf("+-+-+\n");
}
printline();           // <-- Appelle la procedure
printf("|     Solutions     |\n");
printline();
// ...
```

- ▶ `printline ()` n'est codé qu'une fois

- ▶ structuration du programme : améliore *organisation et lisibilité*

Nouvel éloge de la paresse

```
// imprime dans un cadre [V1.0]
printf("-----+\n");
printf("|      Solutions      | \n");
printf("-----+\n");
// Les solutions
// ...
printf("-----+\n");
```

```
// imprime dans un cadre [V2.0]
void printline(void) { // <-- Definit la procedure
    printf("-----+\n");
}
printline();           // <-- Appelle la procedure
printf("|      Solutions      | \n");
printline();
// ...
```

- ▶ `printline ()` n'est codé qu'une fois
- ▶ *structuration du programme : améliore organisation et lisibilité*

Définir une procédure

```
printline1(void) { // une ligne spécifique
    printf("-----+\n");
}
printline2(int length) { // une ligne générale
    int i;
    printf("+");
    // hypothèse (length > 1)
    for (i=1; i < length-1; i++) { printf("-"); }
    printf("+\n");
}
```

- ▶ *length est le paramètre formel de la procédure*
- ▶ paramétriser ⇒ procédures génériques et réutilisables
- ▶ void printline1 (**void**) { ... } équivaut à printline1 () {...}
- ▶ la définition est *extérieure* aux autres fonctions (\neq Pascal...)
- ▶ les variables définies dans la procédure ont une portée *locale*
- ▶ un nom de procédure réservé : main() !!

Définir une procédure

```
printline1(void) { // une ligne spécifique
    printf("-----+\n");
}
printline2(int length) { // une ligne générale
    int i;
    printf("+");
    // hypothèse (length > 1)
    for (i=1; i < length-1; i++) { printf("-"); }
    printf("+\n");
}
```

- ▶ length est le *paramètre formel* de la procédure
- ▶ paramétriser ⇒ procédures génériques et réutilisables
- ▶ void printline1 (void) { ... } équivaut à printline1 () {...}
- ▶ la définition est *extérieure* aux autres fonctions (\neq Pascal...)
- ▶ les variables définies dans la procédure ont une portée *locale*
- ▶ un nom de procédure réservé : main() !!

Définir une procédure

```
printline1(void) { // une ligne spécifique
    printf("-----+\n");
}
printline2(int length) { // une ligne générale
    int i;
    printf("+");
    // hypothèse (length > 1)
    for (i=1; i < length-1; i++) { printf("-"); }
    printf("+\n");
}
```

- ▶ length est le *paramètre formel* de la procédure
- ▶ paramétriser ⇒ procédures génériques et réutilisables
- ▶ **void** printline1(**void**) { ... } équivaut à printline1 () {...}
- ▶ la définition est *extérieure* aux autres fonctions (\neq Pascal...)
- ▶ les variables définies dans la procédure ont une portée *locale*
- ▶ un nom de procédure réservé : main() !!

Définir une procédure

```
printline1(void) { // une ligne spécifique
    printf("-----+\n");
}
printline2(int length) { // une ligne générale
    int i;
    printf("+");
    // hypothèse (length > 1)
    for (i=1; i < length-1; i++) { printf("-"); }
    printf("+\n");
}
```

- ▶ length est le *paramètre formel* de la procédure
- ▶ paramétriser ⇒ procédures génériques et réutilisables
- ▶ **void** printline1 (**void**) { ... } équivaut à printline1 () {...}
- ▶ la définition est *extérieure* aux autres fonctions (\neq Pascal...)
- ▶ les variables définies dans la procédure ont une portée *locale*
- ▶ un nom de procédure réservé : main() !!

Définir une procédure

```
printline1(void) { // une ligne spécifique
    printf("-----+\n");
}
printline2(int length) { // une ligne générale
    int i;
    printf("+");
    // hypothèse (length > 1)
    for (i=1; i < length-1; i++) { printf("-"); }
    printf("+\n");
}
```

- ▶ length est le *paramètre formel* de la procédure
- ▶ paramétriser ⇒ procédures génériques et réutilisables
- ▶ **void** printline1 (**void**) { ... } équivaut à printline1 () {...}
- ▶ la définition est *extérieure* aux autres fonctions (\neq Pascal...)
- ▶ les variables définies dans la procédure ont une portée *locale*
- ▶ un nom de procédure réservé : main() !!

Définir une procédure

```
printline1(void) { // une ligne spécifique
    printf("-----+\n");
}
printline2(int length) { // une ligne générale
    int i;
    printf("+");
    // hypothèse (length > 1)
    for (i=1; i < length-1; i++) { printf("-"); }
    printf("+\n");
}
```

- ▶ length est le *paramètre formel* de la procédure
- ▶ paramétriser ⇒ procédures génériques et réutilisables
- ▶ **void** printline1 (**void**) { ... } équivaut à printline1 () {...}
- ▶ la définition est *extérieure* aux autres fonctions (\neq Pascal...)
- ▶ les variables définies dans la procédure ont une portée *locale*
- ▶ un nom de procédure réservé : main() !!

Utiliser une procédure

```
int main() {  
    int length = 10;  
    void printline2(int); // ← Prototype  
  
    printline2(10);  
    printline2(length);  
}
```

- ▶ utiliser une procédure non-déclarée :

- ▶ peut produire des avertissements à la compilation

foo.c:6: warning : implicit declaration of function 'myfunc'

- ▶ peut causer des *erreurs* à l'exécution (pas de contrôle des paramètres)
 - ▶ **void** printline2 (**int**); déclare le prototype (signature)
 - ▶ le prototype doit s'accorder avec la définition (nombre et type des paramètres)
 - ▶ paramètres réels et formels doivent s'accorder en type

Utiliser une procédure

```
int main() {
    int length = 10;
    void printline2(int); // ← Prototype

    printline2(10);
    printline2(length);
}
```

- ▶ utiliser une procédure non-déclarée :
 - ▶ peut produire des avertissements à la compilation

foo.c :6 : warning : implicit declaration of function 'myfunc'

- ▶ peut causer des *erreurs* à l'exécution (pas de contrôle des paramètres)
- ▶ **void** printline2 (**int**); déclare le prototype (signature)
- ▶ le prototype doit s'accorder avec la définition (nombre et type des paramètres)
- ▶ paramètres réels et formels doivent s'accorder en type

Utiliser une procédure

```
int main() {
    int length = 10;
    void printline2(int); // ← Prototype

    printline2(10);
    printline2(length);
}
```

- ▶ utiliser une procédure non-déclarée :
 - ▶ peut produire des avertissements à la compilation

foo.c :6 : warning : implicit declaration of function 'myfunc'

- ▶ peut causer des *erreurs* à l'exécution (pas de contrôle des paramètres)
- ▶ **void** printline2 (**int**); déclare le prototype (signature)
- ▶ le prototype doit s'accorder avec la définition (nombre et type des paramètres)
- ▶ paramètres réels et formels doivent s'accorder en type

Utiliser une procédure

```
int main() {  
    int length = 10;  
    void printline2(int); // ← Prototype  
  
    printline2(10);  
    printline2(length);  
}
```

- ▶ utiliser une procédure non-déclarée :
 - ▶ peut produire des avertissements à la compilation

foo.c :6 : warning : implicit declaration of function 'myfunc'

- ▶ peut causer des *erreurs* à l'exécution (pas de contrôle des paramètres)
- ▶ **void** printline2 (**int**); déclare le prototype (signature)
- ▶ le prototype doit s'accorder avec la définition (nombre et type des paramètres)
- ▶ paramètres réels et formels doivent s'accorder en type

Utiliser une procédure

```
int main() {  
    int length = 10;  
    void printline2(int); // ← Prototype  
  
    printline2(10);  
    printline2(length);  
}
```

- ▶ utiliser une procédure non-déclarée :
 - ▶ peut produire des avertissements à la compilation

foo.c :6 : warning : implicit declaration of function 'myfunc'

- ▶ peut causer des *erreurs* à l'exécution (pas de contrôle des paramètres)
- ▶ **void printline2 (int); déclare le prototype (signature)**
- ▶ le prototype doit s'accorder avec la définition (nombre et type des paramètres)
- ▶ paramètres réels et formels doivent s'accorder en type

Utiliser une procédure

```
int main() {  
    int length = 10;  
    void printline2(int); // ← Prototype  
  
    printline2(10);  
    printline2(length);  
}
```

- ▶ utiliser une procédure non-déclarée :
 - ▶ peut produire des avertissements à la compilation

foo.c :6 : warning : implicit declaration of function 'myfunc'

- ▶ peut causer des *erreurs* à l'exécution (pas de contrôle des paramètres)
- ▶ **void** printline2 (**int**); déclare le prototype (signature)
- ▶ **le prototype doit s'accorder avec la définition (nombre et type des paramètres)**
- ▶ paramètres réels et formels doivent s'accorder en type

Utiliser une procédure

```
int main() {  
    int length = 10;  
    void printline2(int); // ← Prototype  
  
    printline2(10);  
    printline2(length);  
}
```

- ▶ utiliser une procédure non-déclarée :
 - ▶ peut produire des avertissements à la compilation

foo.c :6 : warning : implicit declaration of function 'myfunc'

- ▶ peut causer des *erreurs* à l'exécution (pas de contrôle des paramètres)
- ▶ **void** printline2 (**int**); déclare le prototype (signature)
- ▶ le prototype doit s'accorder avec la définition (nombre et type des paramètres)
- ▶ **paramètres réels et formels doivent s'accorder en type**

Passer des paramètres

```
void powerprint(int i, int n) {
    int res = 1;
    printf ("%d^%d\n", i, n);
    for(;n>0;n--) res *= i;
    printf ("=%d\n", res);
}
void main() {
    int i=3, n=5;
    powerprint(i,n);
    printf ("%d\n", n);
}
```

- ▶ l'appel `powerprint(i,n)` crée des *copies* des paramètres réels
- ▶ qui sont des variables locales de la procédure
- ▶ les paramètres réels ne sont pas modifiés
- ▶ les paramètres sont passés *par valeur*

Passer des paramètres

```
void powerprint(int i, int n) {
    int res = 1;
    printf ("%d^%d\n", i, n);
    for(;n>0;n--) res *= i;
    printf ("=%d\n", res);
}
void main() {
    int i=3, n=5;
    powerprint(i,n);
    printf ("%d\n", n);
}
```

- ▶ l'appel powerprint(i,n) crée des *copies* des paramètres réels
- ▶ qui sont des variables locales de la procédure
- ▶ les paramètres réels ne sont pas modifiés
- ▶ les paramètres sont passés *par valeur*

Passer des paramètres

```
void powerprint(int i, int n) {
    int res = 1;
    printf ("%d^%d\n", i, n);
    for(;n>0;n--) res *= i;
    printf ("=%d\n", res);
}
void main() {
    int i=3, n=5;
    powerprint(i,n);
    printf ("%d\n", n);
}
```

- ▶ l'appel powerprint(i,n) crée des *copies* des paramètres réels
- ▶ qui sont des variables locales de la procédure
- ▶ les paramètres réels ne sont pas modifiés
- ▶ les paramètres sont passés *par valeur*

Passer des paramètres

```
void powerprint(int i, int n) {
    int res = 1;
    printf ("%d^%d\n", i, n);
    for(;n>0;n--) res *= i;
    printf ("=%d\n", res);
}
void main() {
    int i=3, n=5;
    powerprint(i,n);
    printf ("%d\n", n);
}
```

- ▶ l'appel powerprint(i,n) crée des *copies* des paramètres réels
- ▶ qui sont des variables locales de la procédure
- ▶ les paramètres réels ne sont pas modifiés
- ▶ les paramètres sont passés *par valeur*

Retour à l'envoyeur : les fonctions

```
int square(int x) {  
    if (x > 0)  
        return (x*x);  
    return 0;  
}  
int main() {  
    printf ("%d\n", square(10));  
}
```

► return Expr; :

- ▶ termine sans délai l'exécution de la fonction
- ▶ renvoie la valeur de Expr à l'appelant
- ▶ le prototype d'une fonction indique le type de la valeur retournée
- ▶ return ne renvoie qu'une seule valeur (pas de tableau)

Retour à l'envoyeur : les fonctions

```
int square(int x) {  
    if (x > 0)  
        return (x*x);  
    return 0;  
}  
int main() {  
    printf ("%d\n", square(10));  
}
```

- ▶ **return Expr;**
 - ▶ termine sans délai l'exécution de la fonction
 - ▶ renvoie la valeur de Expr à l'appelant
- ▶ le prototype d'une fonction indique le type de la valeur retournée
- ▶ return ne renvoie qu'une seule valeur (pas de tableau)

Retour à l'envoyeur : les fonctions

```
int square(int x) {  
    if (x > 0)  
        return (x*x);  
    return 0;  
}  
int main() {  
    printf ("%d\n", square(10));  
}
```

- ▶ **return Expr;**
 - ▶ termine sans délai l'exécution de la fonction
 - ▶ renvoie la valeur de Expr à l'appelant
- ▶ le prototype d'une fonction indique le type de la valeur retournée
- ▶ return ne renvoie qu'une seule valeur (pas de tableau)

Retour à l'envoyeur : les fonctions

```
int square(int x) {  
    if (x > 0)  
        return (x*x);  
    return 0;  
}  
int main() {  
    printf ("%d\n", square(10));  
}
```

- ▶ **return** Expr; :
 - ▶ termine sans délai l'exécution de la fonction
 - ▶ renvoie la valeur de Expr à l'appelant
- ▶ le prototype d'une fonction indique le type de la valeur retournée
- ▶ return ne renvoie qu'une seule valeur (pas de tableau)

Retour à l'envoyeur : les fonctions

```
int square(int x) {  
    if (x > 0)  
        return (x*x);  
    return 0;  
}  
int main() {  
    printf ("%d\n", square(10));  
}
```

- ▶ **return Expr;**
 - ▶ termine sans délai l'exécution de la fonction
 - ▶ renvoie la valeur de `Expr` à l'appelant
- ▶ le prototype d'une fonction indique le type de la valeur retournée
- ▶ **return ne renvoie qu'une seule valeur (pas de tableau)**

Respiration (C. Delannoy)

```
int n=10, q=2 ;  
  
int main() {  
    int fct (int);  
    void f (void);  
    int n=0, p=5;  
    n = fct(p);  
    printf ("main: n = %d, p = %d, q = %d\n", n, p, q);  
    f();  
} // manque return  
int fct (int p) {  
    int q = 2 * p + n ;  
    printf ("fct: n = %d, p = %d, q = %d\n", n, p, q);  
    return q;  
}  
void f (void) {  
    int p = q * n;  
    printf ("f: n = %d, p = %d, q = %d\n", n, p, q);  
}
```

Paramètres formels et paramètres réels

```
int i=1;
main () {
    auto int i , j ;
    i = reset();
    for(j=1; j<= 3; j++) {
        printf("%d %d", i, j );
        printf("%d", next(i));
        printf("%d", last(i));
        printf("%d", new(i+j));
    }
}
```

```
int reset(void)  {
    return i ;
}
int last(int j)  {
    static int i=10;
    return (j=i--);
}
```

```
int next(int j) {
    return (j=i++);
}
int new(int i) {
    auto int j=10;
    return (i= j += i );
}
```

Respiration (C. Delannoy)

```
void fcompte (void) {
    static int i;
    i++;
    printf ("i= %d\n", i);
}

int main() {
    void fcompte (void);
    int i;
    for (i=0 ; i<3 ; i++) fcompte ();
}

    return 0;
}
```

Récréations

1. Écrire une fonction qui prend un entier et retourne l'entier obtenu en lisant les chiffres de droite à gauche (hint : utiliser `sprintf(chaine, format, ...)` qui imprime dans une chaîne) ;
2. Écrire une fonction qui calcule et retourne le nombre de 1 dans la représentation binaire d'un entier long
3. Écrire une fonction qui prend un entier et l'imprime en binaire
4. Conversion radian vers degré et réciproquement
5. Calcul du pgcd de deux entiers par l'algorithme d'Euclide, qui utilise la relation : $\text{pgcd}(a, b) = \text{pgcd}(b, r)$, avec r le reste de la division entière de a par b
 - initialiser avec a le plus grand, et b le plus petit
 - stopper quand $r = 0$, le pgcd est alors b

Récréations

1. Écrire une fonction qui prend un entier et retourne l'entier obtenu en lisant les chiffres de droite à gauche (hint : utiliser `sprintf` (`chaine, format, ...`) qui imprime dans une chaîne) ;
2. Écrire une fonction qui calcule et retourne le nombre de 1 dans la représentation binaire d'un entier long
3. Écrire une fonction qui prend un entier et l'imprime en binaire
4. Conversion radian vers degré et réciproquement
5. Calcul du pgcd de deux entiers par l'algorithme d'Euclide, qui utilise la relation : $\text{pgcd}(a, b) = \text{pgcd}(b, r)$, avec r le reste de la division entière de a par b
 - initialiser avec a le plus grand, et b le plus petit
 - stopper quand $r = 0$, le pgcd est alors b

Récréations

1. Écrire une fonction qui prend un entier et retourne l'entier obtenu en lisant les chiffres de droite à gauche (hint : utiliser `sprintf` (`chaine, format, ...`) qui imprime dans une chaîne) ;
2. Écrire une fonction qui calcule et retourne le nombre de 1 dans la représentation binaire d'un entier long
3. Écrire une fonction qui prend un entier et l'imprime en binaire
4. Conversion radian vers degré et réciproquement
5. Calcul du pgcd de deux entiers par l'algorithme d'Euclide, qui utilise la relation : $\text{pgcd}(a, b) = \text{pgcd}(b, r)$, avec r le reste de la division entière de a par b
 - initialiser avec a le plus grand, et b le plus petit
 - stopper quand $r = 0$, le pgcd est alors b

Récréations

1. Écrire une fonction qui prend un entier et retourne l'entier obtenu en lisant les chiffres de droite à gauche (hint : utiliser `sprintf` (`chaine, format, ...`) qui imprime dans une chaîne) ;
2. Écrire une fonction qui calcule et retourne le nombre de 1 dans la représentation binaire d'un entier long
3. Écrire une fonction qui prend un entier et l'imprime en binaire
4. **Conversion radian vers degré et réciproquement**
5. Calcul du pgcd de deux entiers par l'algorithme d'Euclide, qui utilise la relation : $\text{pgcd}(a, b) = \text{pgcd}(b, r)$, avec r le reste de la division entière de a par b
 - Initialiser avec a le plus grand, et b le plus petit
 - stopper quand $r = 0$, le pgcd est alors b

Récréations

1. Écrire une fonction qui prend un entier et retourne l'entier obtenu en lisant les chiffres de droite à gauche (hint : utiliser `sprintf` (`chaine, format, ...`) qui imprime dans une chaîne) ;
2. Écrire une fonction qui calcule et retourne le nombre de 1 dans la représentation binaire d'un entier long
3. Écrire une fonction qui prend un entier et l'imprime en binaire
4. Conversion radian vers degré et réciproquement
5. Calcul du pgcd de deux entiers par l'algorithme d'Euclide, qui utilise la relation : $\text{pgcd}(a, b) = \text{pgcd}(b, r)$, avec r le reste de la division entière de a par b
 - ▶ initialiser avec a le plus grand, et b le plus petit
 - ▶ stopper quand $r = 0$, le pgcd est alors b

Récréations

1. Écrire une fonction qui prend un entier et retourne l'entier obtenu en lisant les chiffres de droite à gauche (hint : utiliser `sprintf` (`chaine, format, ...`) qui imprime dans une chaîne) ;
2. Écrire une fonction qui calcule et retourne le nombre de 1 dans la représentation binaire d'un entier long
3. Écrire une fonction qui prend un entier et l'imprime en binaire
4. Conversion radian vers degré et réciproquement
5. Calcul du pgcd de deux entiers par l'algorithme d'Euclide, qui utilise la relation : $\text{pgcd}(a, b) = \text{pgcd}(b, r)$, avec r le reste de la division entière de a par b
 - ▶ initialiser avec a le plus grand, et b le plus petit
 - ▶ stopper quand $r = 0$, le pgcd est alors b

Récréations

1. Écrire une fonction qui prend un entier et retourne l'entier obtenu en lisant les chiffres de droite à gauche (hint : utiliser `sprintf` (`chaine, format, ...`) qui imprime dans une chaîne) ;
2. Écrire une fonction qui calcule et retourne le nombre de 1 dans la représentation binaire d'un entier long
3. Écrire une fonction qui prend un entier et l'imprime en binaire
4. Conversion radian vers degré et réciproquement
5. Calcul du pgcd de deux entiers par l'algorithme d'Euclide, qui utilise la relation : $\text{pgcd}(a, b) = \text{pgcd}(b, r)$, avec r le reste de la division entière de a par b
 - ▶ initialiser avec a le plus grand, et b le plus petit
 - ▶ stopper quand $r = 0$, le pgcd est alors b

Sous le capot, *la pile*

► L'appel d'une fonction :

- ▶ sauve les valeurs courantes des variables locales de l'appelant
- ▶ crée *un nouvel environnement* contenant :
 - ▶ les paramètres formels
 - ▶ les variables locales
 - ▶ les variables globales
- ▶ Au retour de la fonction :
 - ▶ l'environnement courant est supprimé
 - ▶ l'environnement redévient celui de l'appelant
 - ▶ l'exécution reprend après l'appel de la fonction
- ▶ Nouvelle fonctionnalité : cloisonnement des environnements
- ▶ Les valeurs des variables et les adresses des points de retour sont stockées dans une *pile*

Sous le capot, *la pile*

► L'appel d'une fonction :

- ▶ sauve les valeurs courantes des variables locales de l'appelant
- ▶ crée *un nouvel environnement* contenant :
 - ▶ les paramètres formels
 - ▶ les variables locales
 - ▶ les variables globales
- ▶ Au retour de la fonction :
 - ▶ l'environnement courant est supprimé
 - ▶ l'environnement redévient celui de l'appelant
 - ▶ l'exécution reprend après l'appel de la fonction
- ▶ Nouvelle fonctionnalité : cloisonnement des environnements
- ▶ Les valeurs des variables et les adresses des points de retour sont stockées dans une *pile*

Sous le capot, *la pile*

- ▶ L'appel d'une fonction :

- ▶ sauve les valeurs courantes des variables locales de l'appelant
- ▶ crée *un nouvel environnement* contenant :
 - ▶ les paramètres formels
 - ▶ les variables locales
 - ▶ les variables globales
- ▶ Au retour de la fonction :
 - ▶ l'environnement courant est supprimé
 - ▶ l'environnement redévient celui de l'appelant
 - ▶ l'exécution reprend après l'appel de la fonction
- ▶ Nouvelle fonctionnalité : cloisonnement des environnements
- ▶ Les valeurs des variables et les adresses des points de retour sont stockées dans une *pile*

Sous le capot, *la pile*

► L'appel d'une fonction :

- ▶ sauve les valeurs courantes des variables locales de l'appelant
- ▶ crée *un nouvel environnement* contenant :
 - ▶ les paramètres formels
 - ▶ les variables locales
 - ▶ les variables globales
- ▶ Au retour de la fonction :
 - ▶ l'environnement courant est supprimé
 - ▶ l'environnement redévient celui de l'appelant
 - ▶ l'exécution reprend après l'appel de la fonction
- ▶ Nouvelle fonctionnalité : cloisonnement des environnements
- ▶ Les valeurs des variables et les adresses des points de retour sont stockées dans une *pile*

Sous le capot, *la pile*

► L'appel d'une fonction :

- ▶ sauve les valeurs courantes des variables locales de l'appelant
- ▶ crée *un nouvel environnement* contenant :
 - ▶ les paramètres formels
 - ▶ **les variables locales**
 - ▶ les variables globales
- ▶ Au retour de la fonction :
 - ▶ l'environnement courant est supprimé
 - ▶ l'environnement redévient celui de l'appelant
 - ▶ l'exécution reprend après l'appel de la fonction
- ▶ Nouvelle fonctionnalité : cloisonnement des environnements
- ▶ Les valeurs des variables et les adresses des points de retour sont stockées dans une *pile*

Sous le capot, *la pile*

- ▶ L'appel d'une fonction :

- ▶ sauve les valeurs courantes des variables locales de l'appelant
- ▶ crée *un nouvel environnement* contenant :
 - ▶ les paramètres formels
 - ▶ les variables locales
 - ▶ les variables globales
- ▶ Au retour de la fonction :
 - ▶ l'environnement courant est supprimé
 - ▶ l'environnement redévient celui de l'appelant
 - ▶ l'exécution reprend après l'appel de la fonction
- ▶ Nouvelle fonctionnalité : cloisonnement des environnements
- ▶ Les valeurs des variables et les adresses des points de retour sont stockées dans une *pile*

Sous le capot, *la pile*

- ▶ L'appel d'une fonction :

- ▶ sauve les valeurs courantes des variables locales de l'appelant
- ▶ crée *un nouvel environnement* contenant :
 - ▶ les paramètres formels
 - ▶ les variables locales
 - ▶ les variables globales
- ▶ Au retour de la fonction :
 - ▶ l'environnement courant est supprimé
 - ▶ l'environnement redevient celui de l'appelant
 - ▶ l'exécution reprend après l'appel de la fonction
- ▶ Nouvelle fonctionnalité : cloisonnement des environnements
- ▶ Les valeurs des variables et les adresses des points de retour sont stockées dans une *pile*

Sous le capot, *la pile*

- ▶ L'appel d'une fonction :

- ▶ sauve les valeurs courantes des variables locales de l'appelant
- ▶ crée *un nouvel environnement* contenant :
 - ▶ les paramètres formels
 - ▶ les variables locales
 - ▶ les variables globales
- ▶ Au retour de la fonction :
 - ▶ ***l'environnement courant est supprimé***
 - ▶ l'environnement redevient celui de l'appelant
 - ▶ l'exécution reprend après l'appel de la fonction
- ▶ Nouvelle fonctionnalité : cloisonnement des environnements
- ▶ Les valeurs des variables et les adresses des points de retour sont stockées dans une *pile*

Sous le capot, *la pile*

- ▶ L'appel d'une fonction :

- ▶ sauve les valeurs courantes des variables locales de l'appelant
- ▶ crée *un nouvel environnement* contenant :
 - ▶ les paramètres formels
 - ▶ les variables locales
 - ▶ les variables globales
- ▶ Au retour de la fonction :
 - ▶ l'environnement courant est supprimé
 - ▶ ***l'environnement redevient celui de l'appelant***
 - ▶ l'exécution reprend après l'appel de la fonction
- ▶ Nouvelle fonctionnalité : cloisonnement des environnements
- ▶ Les valeurs des variables et les adresses des points de retour sont stockées dans une *pile*

Sous le capot, *la pile*

- ▶ L'appel d'une fonction :

- ▶ sauve les valeurs courantes des variables locales de l'appelant
- ▶ crée *un nouvel environnement* contenant :
 - ▶ les paramètres formels
 - ▶ les variables locales
 - ▶ les variables globales
- ▶ Au retour de la fonction :
 - ▶ l'environnement courant est supprimé
 - ▶ l'environnement redevient celui de l'appelant
 - ▶ **l'exécution reprend après l'appel de la fonction**
- ▶ Nouvelle fonctionnalité : cloisonnement des environnements
- ▶ Les valeurs des variables et les adresses des points de retour sont stockées dans une *pile*

Sous le capot, *la pile*

- ▶ L'appel d'une fonction :

- ▶ sauve les valeurs courantes des variables locales de l'appelant
- ▶ crée *un nouvel environnement* contenant :
 - ▶ les paramètres formels
 - ▶ les variables locales
 - ▶ les variables globales
- ▶ Au retour de la fonction :
 - ▶ l'environnement courant est supprimé
 - ▶ l'environnement redevient celui de l'appelant
 - ▶ l'exécution reprend après l'appel de la fonction
- ▶ **Nouvelle fonctionnalité : cloisonnement des environnements**
- ▶ Les valeurs des variables et les adresses des points de retour sont stockées dans une *pile*

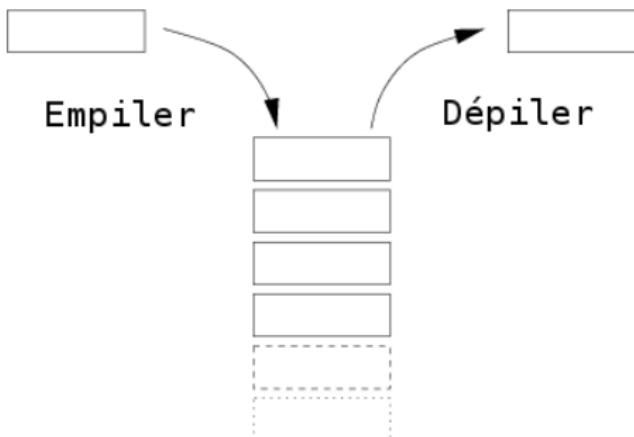
Sous le capot, *la pile*

- ▶ L'appel d'une fonction :

- ▶ sauve les valeurs courantes des variables locales de l'appelant
- ▶ crée *un nouvel environnement* contenant :
 - ▶ les paramètres formels
 - ▶ les variables locales
 - ▶ les variables globales
- ▶ Au retour de la fonction :
 - ▶ l'environnement courant est supprimé
 - ▶ l'environnement redevient celui de l'appelant
 - ▶ l'exécution reprend après l'appel de la fonction
- ▶ Nouvelle fonctionnalité : cloisonnement des environnements
- ▶ Les valeurs des variables et les adresses des points de retour sont stockées dans une *pile*

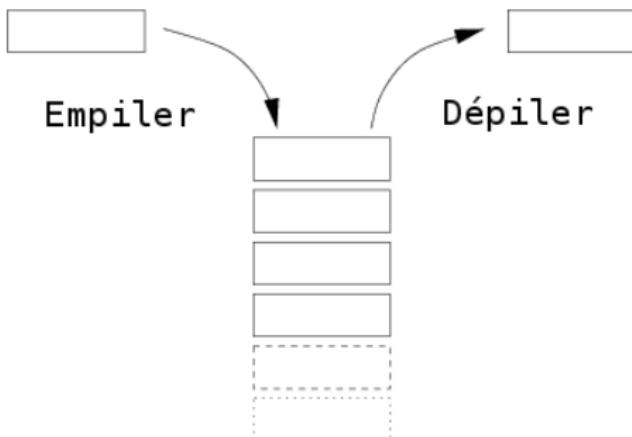
Empiler, dépiler, c'est toujours...

- ▶ Une pile est une structure de données acceptant deux opérations :
 - ▶ empiler
 - ▶ dépiler
- ▶ Une pile n'est accessible que d'un côté et organise les opérations suivant le principe Last In First Out



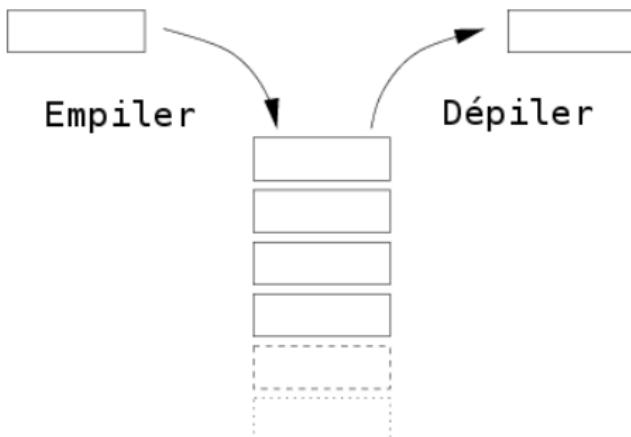
Empiler, dépiler, c'est toujours...

- ▶ Une pile est une structure de données acceptant deux opérations :
 - ▶ empiler
 - ▶ dépiler
- ▶ Une pile n'est accessible que d'un côté et organise les opérations suivant le principe Last In First Out



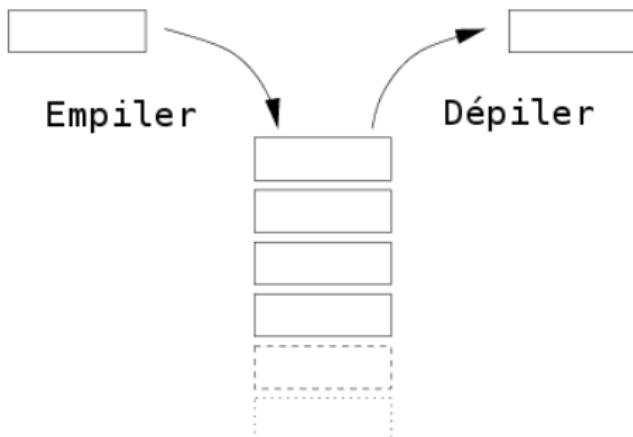
Empiler, dépiler, c'est toujours...

- ▶ Une pile est une structure de données acceptant deux opérations :
 - ▶ empiler
 - ▶ dépiler
- ▶ Une pile n'est accessible que d'un côté et organise les opérations suivant le principe Last In First Out



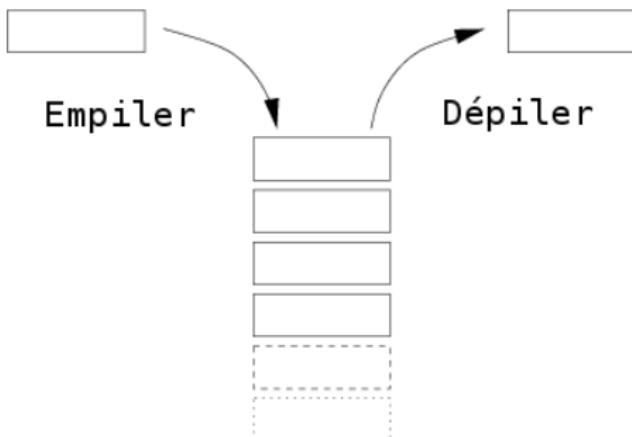
Empiler, dépiler, c'est toujours...

- ▶ Une pile est une structure de données acceptant deux opérations :
 - ▶ empiler
 - ▶ dépiler
- ▶ Une pile n'est accessible que d'un côté et organise les opérations suivant le principe Last In First Out



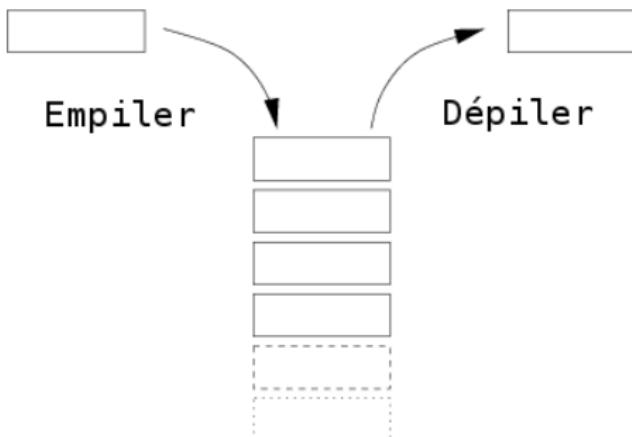
Empiler, dépiler, c'est toujours...

- ▶ Une pile est une structure de données acceptant deux opérations :
 - ▶ empiler
 - ▶ dépiler
- ▶ Une pile n'est accessible que d'un côté et organise les opérations suivant le principe Last In First Out



Empiler, dépiler, c'est toujours...

- ▶ Une pile est une structure de données acceptant deux opérations :
 - ▶ empiler
 - ▶ dépiler
- ▶ Une pile n'est accessible que d'un côté et organise les opérations suivant le principe Last In First Out



Quand le nombre d'arguments varie

- ▶ Il existe des fonctions mathématiques d'arité variable (`max`, `min...`)
- ▶ Il existe des fonctions du C dont le nombre de paramètres varie (`printf`, `scanf...`)
- ▶ Deux problèmes : contrôle du nombre d'arguments, versatilité de leurs types.
- ▶ Déclaration de `min` :

```
int min(int first, ...);
```

... annonce un nombre variable d'arguments

Quand le nombre d'arguments varie

- ▶ Il existe des fonctions mathématiques d'arité variable (`max`, `min...`)
- ▶ Il existe des fonctions du C dont le nombre de paramètres varie (`printf`, `scanf...`)
- ▶ Deux problèmes : contrôle du nombre d'arguments, versatilité de leurs types.
- ▶ Déclaration de `min` :

```
int min(int first, ...);
```

... annonce un nombre variable d'arguments

Quand le nombre d'arguments varie

- ▶ Il existe des fonctions mathématiques d'arité variable (`max`, `min...`)
- ▶ Il existe des fonctions du C dont le nombre de paramètres varie (`printf`, `scanf...`)
- ▶ Deux problèmes : contrôle du nombre d'arguments, versatilité de leurs types.
- ▶ Déclaration de `min` :

```
int min(int first, ...);
```

... annonce un nombre variable d'arguments

Quand le nombre d'arguments varie

- ▶ Il existe des fonctions mathématiques d'arité variable (`max`, `min...`)
- ▶ Il existe des fonctions du C dont le nombre de paramètres varie (`printf`, `scanf...`)
- ▶ Deux problèmes : contrôle du nombre d'arguments, versatilité de leurs types.
- ▶ Déclaration de `min` :

```
int min(int first, ...);
```

... annonce un nombre variable d'arguments

Définir une fonction à nombre variables d'arguments

```
#include <stdarg.h>
int min(int first, ...) {
    va_list ap;           // liste des arguments
    va_start(ap, first); // initialise ap
    int res = first, next = first;
    while (next >= 0) {
        if (next < res) res = next;
        next = va_arg(ap, int); // le nom et le type
    }
    va_end(ap); // termine l'utilisation de ap
    return(res)
}
```

- ▶ `va_list`, `va_start`, `va_arg`, `va_end` sont des *macros*
- ▶ Ces macros rendent *transparentes* l'utilisation d'un nombre variable d'arguments

Définir une fonction à nombre variables d'arguments

```
#include <stdarg.h>
int min(int first, ...) {
    va_list ap;           // liste des arguments
    va_start(ap, first); // initialise ap
    int res = first, next = first;
    while (next >= 0) {
        if (next < res) res = next;
        next = va_arg(ap, int); // le nom et le type
    }
    va_end(ap); // termine l'utilisation de ap
    return(res)
}
```

- ▶ `va_list`, `va_start`, `va_arg`, `va_end` sont des *macros*
- ▶ Ces macros rendent *transparentes* l'utilisation d'un nombre variable d'arguments

Un serpent mordant sa queue : les fonctions récursives

```
int fact(int n) {
    if (n == 0) return (1);
    return (n*fact(n-1));
}
int main() {
    int i = 10;

    printf("fact (%d)=%d\n", i, fact(i));
}
```

[▶ play fact](#)

- ▶ pas besoin de prototype
- ▶ une procédure/fonction récursive correcte doit terminer : condition d'arrêt
- ▶ les fonctions récursives - la programmation sans variable ?
- ▶ une perte d'efficacité ?

La récursion, forme primaire de l'itération

```
main() {  
    int lecture (void); /* fonction de lecture */  
    int n; /* entier à lire */  
  
    printf ("entrez un entier : ");  
    n = lecture ();  
}  
int lecture (void) {  
    int compte, p;  
  
    compte = scanf ("%d", &p);  
    if (!compte) {  
        printf ("** saisie incorrecte - recommencez : ");  
        p = lecture ();  
    }  
    return (p);  
}
```

Comparez avec le même programme avec **while()**.

La récursion, forme primaire de l'itération

```
main() {  
    int lecture (void); /* fonction de lecture */  
    int n; /* entier à lire */  
  
    printf ("entrez un entier : ");  
    n = lecture ();  
}  
int lecture (void) {  
    int compte, p;  
  
    compte = scanf ("%d", &p);  
    if (!compte) {  
        printf ("** saisie incorrecte - recommencez : ");  
        p = lecture ();  
    }  
    return (p);  
}
```

Comparez avec le même programme avec while().

Penser récursif, programmer sans variable

▶ play pi

Récréations

1. Version récursive de l'algorithme d'Euclide
2. Calcul récursif de x^n (par $x^n = x * x^{n-1}$)
3. Calcul récursif du nombre d'or comme rapport de deux nombres successifs de la suite de Fibonacci
4. Calcul récursif des coefficients du binôme C_n^p , par
$$C_n^p = C_{n-1}^p + C_{n-1}^{p_1}, \quad 0 < p < n$$

Récréations

1. Version récursive de l'algorithme d'Euclide
2. Calcul récursif de x^n (par $x^n = x * x^{n-1}$)
3. Calcul récursif du nombre d'or comme rapport de deux nombres successifs de la suite de Fibonacci
4. Calcul récursif des coefficients du binôme C_n^p , par
$$C_n^p = C_{n-1}^p + C_{n-1}^{p_1}, \quad 0 < p < n$$

Récréations

1. Version récursive de l'algorithme d'Euclide
2. Calcul récursif de x^n (par $x^n = x * x^{n-1}$)
3. Calcul récursif du nombre d'or comme rapport de deux nombres successifs de la suite de Fibonacci
4. Calcul récursif des coefficients du binôme C_n^p , par
$$C_n^p = C_{n-1}^p + C_{n-1}^{p_1}, \quad 0 < p < n$$

Récréations

1. Version récursive de l'algorithme d'Euclide
2. Calcul récursif de x^n (par $x^n = x * x^{n-1}$)
3. Calcul récursif du nombre d'or comme rapport de deux nombres successifs de la suite de Fibonacci
4. Calcul récursif des coefficients du binôme C_n^p , par
$$C_n^p = C_{n-1}^p + C_{n-1}^{p_1}, \quad 0 < p < n$$

Le(s) thème(s) du jour: Point sur les pointeurs

Les pointeurs, puissance et nuisance du C



Un voleur connaissant les pointeurs

- ▶ cache les bijoux dans la consigne dans le casier 29878
- ▶ écrit 29878 sur un papier, et le cache dans le casier 89543
- ▶ et recommence encore une fois... s'il est très prudent
- ▶ dans un casier, il y a un numéro de casier
- ▶ dans une variable, il y a l'adresse d'une variable...

Un voleur connaissant les pointeurs

- ▶ cache les bijoux dans la consigne dans le casier 29878
- ▶ écrit 29878 sur un papier, et le cache dans le casier 89543
- ▶ et recommence encore une fois... s'il est très prudent
 - ▶ dans un casier, il y a un numéro de casier
 - ▶ dans une variable, il y a l'adresse d'une variable...

Un voleur connaissant les pointeurs

- ▶ cache les bijoux dans la consigne dans le casier 29878
- ▶ écrit 29878 sur un papier, et le cache dans le casier 89543
- ▶ et recommence encore une fois... s'il est très prudent
 - ▶ dans un casier, il y a un numéro de casier
 - ▶ dans une variable, il y a l'adresse d'une variable...

Un voleur connaissant les pointeurs

- ▶ cache les bijoux dans la consigne dans le casier 29878
- ▶ écrit 29878 sur un papier, et le cache dans le casier 89543
- ▶ et recommence encore une fois... s'il est très prudent
 - ▶ dans un casier, il y a un numéro de casier
 - ▶ dans une variable, il y a l'adresse d'une variable...

Un voleur connaissant les pointeurs

- ▶ cache les bijoux dans la consigne dans le casier 29878
- ▶ écrit 29878 sur un papier, et le cache dans le casier 89543
- ▶ et recommence encore une fois... s'il est très prudent
- ▶ dans un casier, il y a un numéro de casier
- ▶ dans une variable, il y a l'adresse d'une variable...

Un voleur connaissant les pointeurs

- ▶ cache les bijoux dans la consigne dans le casier 29878
- ▶ écrit 29878 sur un papier, et le cache dans le casier 89543
- ▶ et recommence encore une fois... s'il est très prudent
- ▶ dans un casier, il y a un numéro de casier
- ▶ dans une variable, il y a l'adresse d'une variable...

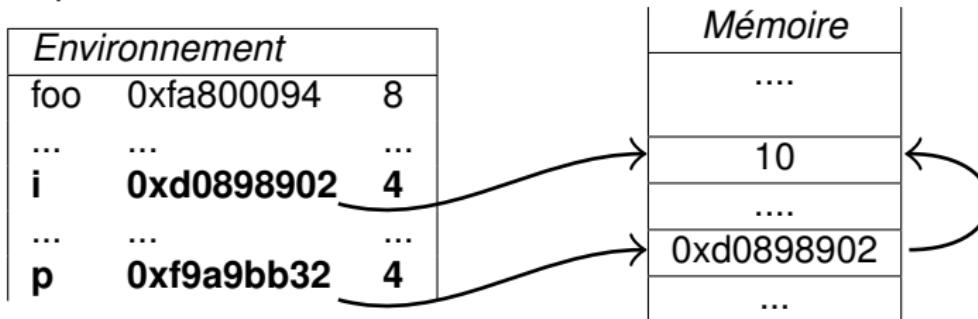
Les pointeurs : de nouveaux types de variables

```

int i = 10, j ;
int *p = &i ;
j = 2 * (*p);
(*p) = j + 1;
printf("l'adresse %p contient la valeur %i\n", p, j);

```

- ▶ Un pointeur est une variable dont la valeur est une adresse



- ▶ \forall type T, $T^* \text{ var}$; définit var de type *pointeur sur T*;
- ▶ lire : $*\text{var}$ est de type T ; la *valeur* de var est une adresse
- ▶ **int** *p1 = &i, *p2; p2 = p1; $*\text{p2} = ???$

Les pointeurs : déréférencement

```
int i = 10, j;
int *pi = &i;
struct {x: double; y:double} point = {1, 1};
point * pp;
j = 2 * (*pi);
(*pi) = j + 1;
if ((*pp).x == pp->y) {...}
```

- ▶ l'opérateur `*` est l'opérateur de *déréférencement*
- ▶ `*p` est un `int` dont la valeur est lue à l'adresse `p`
- ▶ par récurrence : `int **p`, définit une variable dont la valeur est l'adresse d'une variable de type `int *`...
- ▶ si `var` est une structure contenant champ, et `pvar` un pointeur sur `var` : `*pvar.champ` est la valeur à l'adresse (?) `pvar.champ` ⇒ utiliser : `(*pvar).champ` ou `pvar->champ`.

Les pointeurs : déréférencement

```
int i = 10, j;
int *pi = &i;
struct {x: double; y:double} point = {1, 1};
point * pp;
j = 2 * (*pi);
(*pi) = j + 1;
if ((*pp).x == pp->y) {...}
```

- ▶ l'opérateur `*` est l'opérateur de *déréférencement*
- ▶ `*p` est un int dont la valeur est lue à l'adresse `p`
- ▶ par récurrence : `int **p`, définit une variable dont la valeur est l'adresse d'une variable de type `int *`...
- ▶ si `var` est une structure contenant champ, et `pvar` un pointeur sur `var` : `*pvar.champ` est la valeur à l'adresse (?) `pvar.champ` ⇒ utiliser : `(*pvar).champ` ou `pvar->champ`.

Les pointeurs : déréférencement

```
int i = 10, j;
int *pi = &i;
struct {x: double; y:double} point = {1, 1};
point * pp;
j = 2 * (*pi);
(*pi) = j + 1;
if ((*pp).x == pp->y) { ... }
```

- ▶ l'opérateur `*` est l'opérateur de *déréférencement*
- ▶ `*p` est un `int` dont la valeur est lue à l'adresse `p`
- ▶ par récurrence : `int **p`, définit une variable dont la valeur est l'adresse d'une variable de type `int *`...
- ▶ si `var` est une structure contenant champ, et `pvar` un pointeur sur `var` : `*pvar.champ` est la valeur à l'adresse (?) `pvar.champ` ⇒ utiliser : `(*pvar).champ` ou `pvar->champ`.

Les pointeurs : déréférencement

```
int i = 10, j;
int *pi = &i;
struct {x: double; y:double} point = {1, 1};
point * pp;
j = 2 * (*pi);
(*pi) = j + 1;
if ((*pp).x == pp->y) { ... }
```

- ▶ l'opérateur `*` est l'opérateur de *déréférencement*
- ▶ `*p` est un `int` dont la valeur est lue à l'adresse `p`
- ▶ par récurrence : `int **p`, définit une variable dont la valeur est l'adresse d'une variable de type `int *`...
- ▶ si `var` est une structure contenant champ, et pvar un pointeur sur `var : *pvar.champ` est la valeur à l'adresse (?) `pvar.champ` ⇒ utiliser : `(*pvar).champ` ou `pvar->champ`.

Pas d'abonné à l'adresse demandée...

```
char *pc = NULL;  
int * pi , i ;
```

- ▶ les pointeurs s'initialisent, comme toutes les variables
- ▶ NULL (dans `<stddef.h>`) est une adresse inacessible
- ▶ `char *c=NULL; printf(" (*c) vaut %c\n", *c);` est une erreur à l'exécution :

Bus error

- ▶ déréférencer une adresse inconnue est aussi une erreur
- ▶ pire : écrire à une « mauvaise » adresse ! `pi = &i + 1000; *pi = 12;` est funeste

Segmentation fault

Pas d'abonné à l'adresse demandée...

```
char *pc = NULL;  
int * pi , i ;
```

- ▶ les pointeurs s'initialisent, comme toutes les variables
- ▶ **NULL (dans <stddef.h>) est une adresse inaccessible**
- ▶ char *c=NULL; printf(" (*c) vaut %c\n", *c); est une erreur à l'exécution :

Bus error

- ▶ déréférencer une adresse inconnue est aussi une erreur
- ▶ pire : écrire à une « mauvaise » adresse ! `pi = &i + 1000; *pi = 12;` est funeste

Segmentation fault

Pas d'abonné à l'adresse demandée...

```
char *pc = NULL;  
int * pi , i ;
```

- ▶ les pointeurs s'initialisent, comme toutes les variables
- ▶ NULL (dans `<stddef.h>`) est une adresse inacessible
- ▶ `char *c=NULL; printf(" (*c) vaut %c\n", *c);` est une erreur à l'exécution :

Bus error

- ▶ déréférencer une adresse inconnue est aussi une erreur
- ▶ pire : écrire à une « mauvaise » adresse ! `pi = &i + 1000; *pi = 12;` est funeste

Segmentation fault

Pas d'abonné à l'adresse demandée...

```
char *pc = NULL;  
int * pi , i ;
```

- ▶ les pointeurs s'initialisent, comme toutes les variables
- ▶ NULL (dans `<stddef.h>`) est une adresse inacessible
- ▶ `char *c=NULL; printf(" (*c) vaut %c\n", *c);` est une erreur à l'exécution :

Bus error

- ▶ déréférencer une adresse inconnue est aussi une erreur
- ▶ pire : écrire à une « mauvaise » adresse ! `pi = &i + 1000; *pi = 12;` est funeste

Segmentation fault

Pas d'abonné à l'adresse demandée...

```
char *pc = NULL;  
int * pi , i ;
```

- ▶ les pointeurs s'initialisent, comme toutes les variables
- ▶ NULL (dans `<stddef.h>`) est une adresse inacessible
- ▶ `char *c=NULL; printf(" (*c) vaut %c\n", *c);` est une erreur à l'exécution :

Bus error

- ▶ déréférencer une adresse inconnue est aussi une erreur
- ▶ pire : écrire à une « mauvaise » adresse ! `pi = &i + 1000; *pi = 12;` est funeste

Segmentation fault

Pas d'abonné à l'adresse demandée...

```
char *pc = NULL;  
int * pi , i ;
```

- ▶ les pointeurs s'initialisent, comme toutes les variables
- ▶ NULL (dans `<stddef.h>`) est une adresse inacessible
- ▶ `char *c=NULL; printf(" (*c) vaut %c\n", *c);` est une erreur à l'exécution :

Bus error

- ▶ déréférencer une adresse inconnue est aussi une erreur
- ▶ pire : écrire à une « mauvaise » adresse ! `pi = &i + 1000; *pi = 12;` est funeste

Segmentation fault

Pas d'abonné à l'adresse demandée...

```
char *pc = NULL;  
int * pi , i ;
```

- ▶ les pointeurs s'initialisent, comme toutes les variables
- ▶ NULL (dans `<stddef.h>`) est une adresse inacessible
- ▶ `char *c=NULL; printf(" (*c) vaut %c\n", *c);` est une erreur à l'exécution :

Bus error

- ▶ déréférencer une adresse inconnue est aussi une erreur
- ▶ pire : écrire à une « mauvaise » adresse ! `pi = &i + 1000; *pi = 12;` est funeste

Segmentation fault

La lecture des déclarations

```
1  const char * pc;
2  int **ppi;
3  struct complex *pcx;
4  float *ptf[10];
5  double (*ptd)[10];
6  const int * p1; // *p1 est constant: *p1 = x rate
7  int * const p2; // p2 est constant: p2 = x rate
```

1. pc est un pointeur sur un **const char**
2. ppi est un pointeur sur un **int** * ; en déréférençant *deux fois* on aura un **int**
3. pcx est un pointeur sur un complex
4. ptf est un tableau de 10 pointeurs sur float ;
5. ptd est un pointeur sur un tableau de (10) doubles ;
6. en cas de doute :

▶ play cdecl

int * ≠ char * ! Pourquoi diable ?

```
int i = 10;
char c = 'c';
char * pi = &i; /* ← erreur */
void * ptr;

ptr = &i; printf("%i\n", *((int *)ptr));
ptr = &c; printf("%c\n", *((char *)ptr));
```

- ▶ un pointeur est une adresse : l'interpréter demande de connaître la taille de l'objet pointé !!!
- ▶ **int** i; **char** * c = &i; provoque une erreur
- ▶ **void** * définit un pointeur générique
- ▶ **int** i = 5 ; **void** *p = &i; printf ("%i\n", *p); provoque :

*void.c:9: warning: dereferencing 'void *' pointer
void.c:9: error: invalid use of void expression*

- ▶ printf ("%i\n", *((**int** *)p)); est correct : le pointeur p reçoit un type avant déréférencement ;

int * ≠ char * ! Pourquoi diable ?

```
int i = 10;
char c = 'c';
char * pi = &i; /* ← erreur */
void * ptr;

ptr = &i; printf("%i\n", *((int *)ptr));
ptr = &c; printf("%c\n", *((char *)ptr));
```

- ▶ un pointeur est une adresse : l'interpréter demande de connaître la taille de l'objet pointé !!!
- ▶ **int i; char * c = &i;** provoque une erreur
- ▶ **void *** définit un pointeur générique
- ▶ **int i = 5 ; void *p = &i;** `printf ("%i\n", *p);` provoque :

*void.c:9: warning: dereferencing 'void *' pointer
void.c:9: error: invalid use of void expression*

- ▶ `printf ("%i\n", *((int *)p));` est correct : le pointeur p reçoit un type avant déréférencement ;

int * ≠ char * ! Pourquoi diable ?

```

int i = 10;
char c = 'c';
char * pi = &i; /* ← erreur */
void * ptr;

ptr = &i; printf("%i\n", *((int *)ptr));
ptr = &c; printf("%c\n", *((char *)ptr));

```

- ▶ un pointeur est une adresse : l'interpréter demande de connaître la taille de l'objet pointé !!!
- ▶ **int** i; **char** * c = &i; provoque une erreur
- ▶ **void** * définit un pointeur générique
- ▶ **int** i = 5 ; **void** *p = &i; printf ("%i\n", *p); provoque :

*void.c:9: warning: dereferencing 'void *' pointer
void.c:9: error: invalid use of void expression*

- ▶ printf ("%i\n", *((**int** *)p)); est correct : le pointeur p reçoit un type avant déréférencement ;

int * ≠ char * ! Pourquoi diable ?

```

int i = 10;
char c = 'c';
char * pi = &i; /* ← erreur */
void * ptr;

ptr = &i; printf("%i\n", *((int *)ptr));
ptr = &c; printf("%c\n", *((char *)ptr));

```

- ▶ un pointeur est une adresse : l'interpréter demande de connaître la taille de l'objet pointé !!!
- ▶ **int** i; **char** * c = &i; provoque une erreur
- ▶ **void** * définit un pointeur générique
- ▶ **int** i = 5 ; **void** *p = &i; printf ("%i\n", *p); provoque :

*void.c :9 : warning : dereferencing 'void *' pointer
void.c :9 : error : invalid use of void expression*

- ▶ printf ("%i\n", *((**int** *)p)); est correct : le pointeur p reçoit un type avant déréférencement ;

int * ≠ char * ! Pourquoi diable ?

```

int i = 10;
char c = 'c';
char * pi = &i; /* ← erreur */
void * ptr;

ptr = &i; printf("%i\n", *((int *)ptr));
ptr = &c; printf("%c\n", *((char *)ptr));

```

- ▶ un pointeur est une adresse : l'interpréter demande de connaître la taille de l'objet pointé !!!
- ▶ **int** i; **char** * c = &i; provoque une erreur
- ▶ **void** * définit un pointeur générique
- ▶ **int** i = 5 ; **void** *p = &i; printf ("%i\n", *p); provoque :

*void.c :9 : warning : dereferencing 'void *' pointer
void.c :9 : error : invalid use of void expression*

- ▶ printf ("%i\n", *((**int** *)p)); est correct : le pointeur p reçoit un type avant déréférencement ;

int * ≠ char * ! Pourquoi diable ?

```

int i = 10;
char c = 'c';
char * pi = &i; /* ← erreur */
void * ptr;

ptr = &i; printf("%i\n", *((int *)ptr));
ptr = &c; printf("%c\n", *((char *)ptr));

```

- ▶ un pointeur est une adresse : l'interpréter demande de connaître la taille de l'objet pointé !!!
- ▶ **int** i; **char** * c = &i; provoque une erreur
- ▶ **void** * définit un pointeur générique
- ▶ **int** i = 5 ; **void** *p = &i; printf ("%i\n", *p); provoque :

*void.c :9 : warning : dereferencing 'void *' pointer
void.c :9 : error : invalid use of void expression*

- ▶ printf ("%i\n", *((**int** *)p)); est correct : le pointeur p reçoit un type avant déréférencement ;

La vérité sur les tableaux

```
char s[8] = "tableau";
char *c = s;

printf("le char \`a l'adresse %p : %c \n", s, *s);
```

- ▶ un tableau est un pointeur
- ▶ sa valeur est l'adresse du premier élément : `char *c = &s[0]` équivaut à `char *c=s`
- ▶ `&s[1]` est l'adresse de la cellule à gauche de `0 = &s[0] + sizeof(char)`
- ▶ `int * soustab = &tab[3]` construit un sous-tableau
- ▶ `tab[i] = *(tab + i)` : le compilateur s'occupe des `sizeof()`
- ▶ utiliser les pointeurs est plus efficace, mais moins lisible
- ▶ pathologique : `+ est commutatif : tab[i] = *(tab+i) = i[tab] !!`

La vérité sur les tableaux

```
char s[8] = "tableau";
char *c = s;

printf("le char \`a l'adresse %p : %c \n", s, *s);
```

- ▶ un tableau est un pointeur
- ▶ sa valeur est l'adresse du premier élément : `char *c = &s[0]` équivaut à `char *c=s`
- ▶ `&s[1]` est l'adresse de la cellule à gauche de `0 = &s[0] + sizeof(char)`
- ▶ `int * soustab = &tab[3]` construit un sous-tableau
- ▶ `tab[i] = *(tab + i)` : le compilateur s'occupe des `sizeof()`
- ▶ utiliser les pointeurs est plus efficace, mais moins lisible
- ▶ pathologique : `+ est commutatif : tab[i] = *(tab+i) = i[tab] !!`

La vérité sur les tableaux

```
char s[8] = "tableau";
char *c = s;

printf("le char \`a l'adresse %p : %c \n", s, *s);
```

- ▶ un tableau est un pointeur
- ▶ sa valeur est l'adresse du premier élément : `char *c = &s[0]` équivaut à `char *c=s`
- ▶ **&s[1] est l'adresse de la cellule à gauche de 0 = &s[0] + sizeof(char)**
- ▶ `int * soustab = &tab[3]` construit un sous-tableau
- ▶ `tab[i] = *(tab + i)` : le compilateur s'occupe des `sizeof()`
- ▶ utiliser les pointeurs est plus efficace, mais moins lisible
- ▶ pathologique : + est commutatif : `tab[i] = *(tab+i) = i[tab] !!`

La vérité sur les tableaux

```
char s[8] = "tableau";
char *c = s;

printf("le char \`a l'adresse %p : %c \n", s, *s);
```

- ▶ un tableau est un pointeur
- ▶ sa valeur est l'adresse du premier élément : char *c = &s[0]
équivaut à char *c=s
- ▶ &s[1] est l'adresse de la cellule à gauche de 0 =
&s[0] + sizeof(char)
- ▶ int * soustab = &tab[3] construit un sous-tableau
- ▶ tab[i] = *(tab + i) : le compilateur s'occupe des sizeof()
- ▶ utiliser les pointeurs est plus efficace, mais moins lisible
- ▶ pathologique : + est commutatif : tab[i] = *(tab+i) = i[tab] !!

La vérité sur les tableaux

```
char s[8] = "tableau";
char *c = s;

printf("le char \`a l'adresse %p : %c \n", s, *s);
```

- ▶ un tableau est un pointeur
- ▶ sa valeur est l'adresse du premier élément : `char *c = &s[0]` équivaut à `char *c=s`
- ▶ `&s[1]` est l'adresse de la cellule à gauche de 0 = `&s[0] + sizeof(char)`
- ▶ `int * soustab = &tab[3]` construit un sous-tableau
- ▶ `tab[i] = *(tab + i)` : le compilateur s'occupe des `sizeof()`
- ▶ utiliser les pointeurs est plus efficace, mais moins lisible
- ▶ pathologique : + est commutatif : `tab[i] = *(tab+i) = i[tab] !!`

La vérité sur les tableaux

```
char s[8] = "tableau";
char *c = s;

printf("le char \`a l'adresse %p : %c \n", s, *s);
```

- ▶ un tableau est un pointeur
- ▶ sa valeur est l'adresse du premier élément : `char *c = &s[0]` équivaut à `char *c=s`
- ▶ `&s[1]` est l'adresse de la cellule à gauche de 0 = `&s[0] + sizeof(char)`
- ▶ `int * soustab = &tab[3]` construit un sous-tableau
- ▶ `tab[i] = *(tab + i)` : le compilateur s'occupe des `sizeof()`
- ▶ **utiliser les pointeurs est plus efficace, mais moins lisible**
- ▶ pathologique : + est commutatif : `tab[i] = *(tab+i) = i[tab] !!`

La vérité sur les tableaux

```
char s[8] = "tableau";
char *c = s;

printf("le char \`a l'adresse %p : %c \n", s, *s);
```

- ▶ un tableau est un pointeur
- ▶ sa valeur est l'adresse du premier élément : `char *c = &s[0]` équivaut à `char *c=s`
- ▶ `&s[1]` est l'adresse de la cellule à gauche de 0 = `&s[0] + sizeof(char)`
- ▶ `int * soustab = &tab[3]` construit un sous-tableau
- ▶ `tab[i] = *(tab + i)` : le compilateur s'occupe des `sizeof()`
- ▶ utiliser les pointeurs est plus efficace, mais moins lisible
- ▶ pathologique : + est commutatif : `tab[i] = *(tab+i) = i[tab] !!`

Un point sur les pointeurs

Pointeurs et tableaux

Inspection d'un tableau

▶ play showtab

Respiration

```
#include <stdio.h>

int main() {
    int t [3], int i, j; int * adt;
    for (i=0, j=0; i<3; i++) t[i] = j++ + i;
    for (i=0; i<3; i++) printf ("%d ", t[i]);
    printf ("\n");
    for (i=0; i<3; i++) printf ("%d ", *(t+i));
    printf ("\n");
    for (adt = t; adt < t+3; adt++) printf ("%d ", *adt);
    printf ("\n");
}
return 0;
```

Un point sur les pointeurs

Pointeurs et tableaux

Respirations (A. Feuer)

▶ play feuer57

Un point sur les pointeurs

Pointeurs et tableaux

Respirations (A. Feuer)

▶ play feuer59

Un point sur les pointeurs

Pointeurs et tableaux

Respirations (A. Feuer)

▶ play feuer65

Les pointeurs : des tableaux... en puissance

```
1 int tab[5] = {0, 1, 2, 3, 4};  
2 int *ptab = tab;  
3 for (int i = 0; i < 5; i++, tab++)  
    printf("%d\n", *tab);  
5 for (int i = 0; i < 5; i++, ptab++)  
    printf("%d\n", *ptab);
```

- ▶ (1) crée une variable tableau (pointeur) ET *alloue la mémoire pour le stocker*
- ▶ (2) crée une variable tableau (pointeur) *sans aucun espace de stockage* (autre que la valeur du pointeur)
- ▶ quelle différence entre (3/4) et (5/6) ?

Les pointeurs : des tableaux... en puissance

```
1 int tab[5] = {0, 1, 2, 3, 4};  
2 int *ptab = tab;  
3 for (int i = 0; i < 5; i++, tab++)  
    printf("%d\n", *tab);  
5 for (int i = 0; i < 5; i++, ptab++)  
    printf("%d\n", *ptab);
```

- ▶ (1) crée une variable tableau (pointeur) ET *alloue la mémoire pour le stocker*
- ▶ (2) crée une variable tableau (pointeur) *sans aucun espace de stockage* (autre que la valeur du pointeur)
- ▶ quelle différence entre (3/4) et (5/6) ?

Les pointeurs : des tableaux... en puissance

```
1 int tab[5] = {0, 1, 2, 3, 4};  
2 int *ptab = tab;  
3 for (int i = 0; i < 5; i++, tab++)  
    printf("%d\n", *tab);  
5 for (int i = 0; i < 5; i++, ptab++)  
    printf("%d\n", *ptab);
```

- ▶ (1) crée une variable tableau (pointeur) ET *alloue la mémoire pour le stocker*
- ▶ (2) crée une variable tableau (pointeur) *sans aucun espace de stockage (autre que la valeur du pointeur)*
- ▶ quelle différence entre (3/4) et (5/6) ?

Les pointeurs : des tableaux... en puissance

```
1 int tab[5] = {0, 1, 2, 3, 4};  
2 int *ptab = tab;  
3 for (int i = 0; i < 5; i++, tab++)  
    printf("%d\n", *tab);  
5 for (int i = 0; i < 5; i++, ptab++)  
    printf("%d\n", *ptab);
```

- ▶ (1) créée une variable tableau (pointeur) ET *alloue* la mémoire pour le stocker
- ▶ (2) créée une variable tableau (pointeur) *sans aucun espace de stockage* (autre que la valeur du pointeur)
- ▶ quelle différence entre (3/4) et (5/6) ?

Construire des tableaux de taille variable

```
char *s = NULL;  
int n = 30;  
  
s = (char *) malloc (n * sizeof(char));
```

- ▶ `void * malloc(size_t N)` alloue une zone contigüe de N bytes en mémoire sur le *tas*
- ▶ la place pour n entiers : $(n * \text{sizeof(int)})$ bytes
- ▶ `malloc()` renvoie `NULL` si la place manque
- ▶ n'initialise pas la mémoire ($\Rightarrow \text{void} * \text{calloc}(\text{size_t}, \text{size_t})$)
- ▶ `void *realloc(void * ptr, size_t size)` permet d'agrandir (ou de diminuer) une zone
- ▶ `void free(void *s)` assure le « nettoyage » d'une zone mémoire

Vous êtes responsables de la gestion du tas (allocation, libération)

Construire des tableaux de taille variable

```
char *s = NULL;  
int n = 30;  
  
s = (char *) malloc (n * sizeof(char));
```

- ▶ void * malloc(size_t N) alloue une zone contigüe de *N* bytes en mémoire sur le *tas*
- ▶ la place pour *n* entiers : (n*sizeof(int)) bytes
- ▶ malloc() renvoie NULL si la place manque
- ▶ n'initialise pas la mémoire (\Rightarrow void * calloc(size_t , size_t))
- ▶ void *realloc(void * prt, size_t size) permet d'agrandir (ou de diminuer) une zone
- ▶ void free(void *s) assure le « nettoyage » d'une zone mémoire

Vous êtes responsables de la gestion du tas (allocation, libération)

Construire des tableaux de taille variable

```
char *s = NULL;  
int n = 30;  
  
s = (char *) malloc (n * sizeof(char));
```

- ▶ void * malloc(size_t N) alloue une zone contigüe de N bytes en mémoire sur le *tas*
- ▶ la place pour n entiers : $(n * \text{sizeof(int)})$ bytes
- ▶ malloc() renvoie NULL si la place manque
- ▶ n'initialise pas la mémoire (\Rightarrow void * calloc(size_t, size_t))
- ▶ void *realloc(void * prt, size_t size) permet d'agrandir (ou de diminuer) une zone
- ▶ void free(void *s) assure le « nettoyage » d'une zone mémoire

Vous êtes responsables de la gestion du tas (allocation, libération)

Construire des tableaux de taille variable

```
char *s = NULL;  
int n = 30;  
  
s = (char *) malloc (n * sizeof(char));
```

- ▶ void * malloc(size_t N) alloue une zone contigüe de N bytes en mémoire sur le *tas*
- ▶ la place pour n entiers : $(n * \text{sizeof(int)})$ bytes
- ▶ malloc() renvoie **NULL** si la place manque
- ▶ n'initialise pas la mémoire (\Rightarrow void * calloc(size_t, size_t))
- ▶ void *realloc(void * prt, size_t size) permet d'agrandir (ou de diminuer) une zone
- ▶ void free(void *s) assure le « nettoyage » d'une zone mémoire

Vous êtes responsables de la gestion du tas (allocation, libération)

Construire des tableaux de taille variable

```
char *s = NULL;  
int n = 30;  
  
s = (char *) malloc (n * sizeof(char));
```

- ▶ void * malloc(size_t N) alloue une zone contigüe de N bytes en mémoire sur le *tas*
- ▶ la place pour n entiers : $(n * \text{sizeof}(\text{int}))$ bytes
- ▶ malloc() renvoie NULL si la place manque
- ▶ n'initialise pas la mémoire (\Rightarrow void * calloc(size_t, size_t))
- ▶ void *realloc(void * prt, size_t size) permet d'agrandir (ou de diminuer) une zone
- ▶ void free(void *s) assure le « nettoyage » d'une zone mémoire

Vous êtes responsables de la gestion du tas (allocation, libération)

Construire des tableaux de taille variable

```
char *s = NULL;  
int n = 30;  
  
s = (char *) malloc (n * sizeof(char));
```

- ▶ void * malloc(size_t N) alloue une zone contigüe de N bytes en mémoire sur le *tas*
- ▶ la place pour n entiers : $(n * \text{sizeof(int)})$ bytes
- ▶ malloc() renvoie NULL si la place manque
- ▶ n'initialise pas la mémoire (\Rightarrow void * calloc(size_t, size_t))
- ▶ void *realloc(void * prt, size_t size) permet d'agrandir (ou de diminuer) une zone
- ▶ void free(void *s) assure le « nettoyage » d'une zone mémoire

Vous êtes responsables de la gestion du tas (allocation, libération)

Construire des tableaux de taille variable

```
char *s = NULL;  
int n = 30;  
  
s = (char *) malloc (n * sizeof(char));
```

- ▶ void * malloc(size_t N) alloue une zone contigüe de N bytes en mémoire sur le *tas*
- ▶ la place pour n entiers : $(n * \text{sizeof}(\text{int}))$ bytes
- ▶ malloc() renvoie NULL si la place manque
- ▶ n'initialise pas la mémoire (\Rightarrow void * calloc(size_t, size_t))
- ▶ void * realloc(void * prt, size_t size) permet d'agrandir (ou de diminuer) une zone
- ▶ void free(void *s) assure le « nettoyage » d'une zone mémoire

Vous êtes responsables de la gestion du tas (allocation, libération)

Un tableau de chaînes

```
char line[100];
char * words [10];
int nw = 0, len = 0;
while (fgets(line, 100, stdin) != NULL && nw < 10) {
    len = strlen(line) + 1;
    words[nwords] = (char *) malloc((len * sizeof(char)))
    strcpy(line, words);
}
```

- ▶ un tableau mono-dimensionnel est un pointeur
- ▶ un tableau bi-dimensionnel est un pointeur de pointeur, ou encore un tableau de pointeurs
- ▶ les chaînes ne sont pas nécessairement contigües en mémoire...
- ▶ mais leurs adresses le sont
- ▶ attention au `strlen (line)+1`

Un tableau de chaînes

```
char line[100];
char * words [10];
int nw = 0, len = 0;
while (fgets(line, 100, stdin) != NULL && nw < 10) {
    len = strlen(line) + 1;
    words[nwords] = (char *) malloc((len * sizeof(char)))
    strcpy(line, words);
}
```

- ▶ un tableau mono-dimensionnel est un pointeur
- ▶ un tableau bi-dimensionnel est un pointeur de pointeur, ou encore un tableau de pointeurs
- ▶ les chaînes ne sont pas nécessairement contigües en mémoire...
- ▶ mais leurs adresses le sont
- ▶ attention au `strlen (line)+1`

Un tableau de chaînes

```
char line[100];
char * words [10];
int nw = 0, len = 0;
while (fgets(line, 100, stdin) != NULL && nw < 10) {
    len = strlen(line) + 1;
    words[nwords] = (char *) malloc((len * sizeof(char)))
    strcpy(line, words);
}
```

- ▶ un tableau mono-dimensionnel est un pointeur
- ▶ un tableau bi-dimensionnel est un pointeur de pointeur, ou encore un tableau de pointeurs
- ▶ les chaînes ne sont pas nécessairement contiguës en mémoire...
- ▶ mais leurs adresses le sont
- ▶ attention au strlen (line)+1

Un tableau de chaînes

```
char line[100];
char * words [10];
int nw = 0, len = 0;
while (fgets(line, 100, stdin) != NULL && nw < 10) {
    len = strlen(line) + 1;
    words[nwords] = (char *) malloc((len * sizeof(char)))
    strcpy(line, words);
}
```

- ▶ un tableau mono-dimensionnel est un pointeur
- ▶ un tableau bi-dimensionnel est un pointeur de pointeur, ou encore un tableau de pointeurs
- ▶ les chaînes ne sont pas nécessairement contigües en mémoire...
- ▶ **mais leurs adresses le sont**
- ▶ attention au `strlen (line)+1`

Un tableau de chaînes

```
char line[100];
char * words [10];
int nw = 0, len = 0;
while (fgets(line, 100, stdin) != NULL && nw < 10) {
    len = strlen(line) + 1;
    words[nwords] = (char *) malloc((len * sizeof(char)))
    strcpy(line, words);
}
```

- ▶ un tableau mono-dimensionnel est un pointeur
- ▶ un tableau bi-dimensionnel est un pointeur de pointeur, ou encore un tableau de pointeurs
- ▶ les chaînes ne sont pas nécessairement contigües en mémoire...
- ▶ mais leurs adresses le sont
- ▶ **attention au strlen (line)+1**

J'ai la mémoire qui goutte...



```
void leak (int n) {
    int *t = NULL;
    t = (double *) calloc(n, sizeof(double));
    /* do something */
    ...
}

for (i=1; i < 1000000; i++) {
    leak (i);
}
```

Un terrain miné : l'arithmétique des pointeurs

AVERTISSEMENT

Cette diapositive contient des informations sur le langage C qui sont connues pour être la source d'innombrables erreurs de programmation, à un point tel que les langages "modernes" interdisent de telles manipulations.

En conséquence, les enseignants du cours "Principe de programmation" déclinent toute responsabilité vis-à-vis des erreurs que vous, ou tout membre de votre entourage à qui vous tenteriez d'expliquer le C, allez commettre en utilisant l'arithmétique des pointeurs.

- ▶ Rappel : si int i; alors i++; est un racourci pour i=i+1 ;+
- ▶ int * i;; i++ signifie... i = i + sizeof(int)
- ▶ de même double * i;; i++ signifie... i = i + sizeof(double)
- ▶ cohérent avec tab[i] ⇔ *(tab+i)
- ▶ idem pour i--, --i...

Un point sur les pointeurs

L'arithmétique des pointeurs

Les pointeurs ne sont PAS des entiers

▶ play pnotint

Programmation préventive : Valgrind

- ▶ installer valgrind <http://valgrind.org/> (Linux/x86)
- ▶ compiler « normalement » : gcc –Wall –o mon_prog mon_prog.c
- ▶ lancer valgrind mon_prog
- ▶ ⇒ détecte débordements de tableaux, fuite de mémoires etc.

Programmation préventive : Valgrind

- ▶ installer valgrind <http://valgrind.org/> (Linux/x86)
- ▶ compiler « normalement » : gcc –Wall –o mon_prog mon_prog.c
- ▶ lancer valgrind mon_prog
- ▶ ⇒ détecte débordements de tableaux, fuite de mémoires etc.

Programmation préventive : Valgrind

- ▶ installer valgrind <http://valgrind.org/> (Linux/x86)
- ▶ compiler « normalement » : gcc –Wall –o mon_prog mon_prog.c
- ▶ lancer valgrind mon_prog
- ▶ ⇒ détecte débordements de tableaux, fuite de mémoires etc.

Programmation préventive : Valgrind

- ▶ installer valgrind <http://valgrind.org/> (Linux/x86)
- ▶ compiler « normalement » : gcc –Wall –o mon_prog mon_prog.c
- ▶ lancer valgrind mon_prog
- ▶ ⇒ détecte débordements de tableaux, fuite de mémoires etc.

Appels de fonction : passez les pointeurs !

```
typedef struct {
    double real;
    double imag;
} complex;

double module_1(complex c) {
    return(sqrt(c.real*c.real + c.imag*c.imag));
}

double module_2(complex * c) {
    return(sqrt(c->real*c->real + c->imag*c->imag));
}
```

- ▶ **module_1() et module_2() calculent le module d'un complexe**
- ▶ **module_1(x) recopie un complexe (16 bytes), pas d'indirection**
- ▶ **module_2(&x) recopie un pointeur (4 bytes), 4 indirections**
- ▶ **conclusion ?**

Appels de fonction : passez les pointeurs !

```
typedef struct {
    double real;
    double imag;
} complex;

double module_1(complex c) {
    return(sqrt(c.real*c.real + c.imag*c.imag));
}

double module_2(complex * c) {
    return(sqrt(c->real*c->real + c->imag*c->imag));
}
```

- ▶ module_1() et module_2() calculent le module d'un complexe
- ▶ module_1(x) recopie un complexe (16 bytes), pas d'indirection
- ▶ module_2(&x) recopie un pointeur (4 bytes), 4 indirections
- ▶ conclusion ?

Appels de fonction : passez les pointeurs !

```
typedef struct {
    double real;
    double imag;
} complex;

double module_1(complex c) {
    return(sqrt(c.real*c.real + c.imag*c.imag));
}

double module_2(complex * c) {
    return(sqrt(c->real*c->real + c->imag*c->imag));
}
```

- ▶ module_1() et module_2() calculent le module d'un complexe
- ▶ module_1(x) recopie un complexe (16 bytes), pas d'indirection
- ▶ module_2(&x) recopie un pointeur (4 bytes), 4 indirections
- ▶ conclusion ?

Appels de fonction : passez les pointeurs !

```
typedef struct {
    double real;
    double imag;
} complex;

double module_1(complex c) {
    return(sqrt(c.real*c.real + c.imag*c.imag));
}

double module_2(complex * c) {
    return(sqrt(c->real*c->real + c->imag*c->imag));
}
```

- ▶ module_1() et module_2() calculent le module d'un complexe
- ▶ module_1(x) recopie un complexe (16 bytes), pas d'indirection
- ▶ module_2(&x) recopie un pointeur (4 bytes), 4 indirections
- ▶ conclusion ?

Modifier des paramètres en passant des *valeurs* ?

```
/* croit échanger deux entiers */
void swap(int i1, int i2)
{
    int i = i1;
    i1 = i2;
    i2 = i;
}
int j1 = 5, j2 = 10;
swap(j1, j2);
printf("j1=%i -- j2=%i\n", j1, j2);
```

- ▶ l'appel à swap() *recopie* les valeurs 5 et 10 dans deux nouvelles variables
- ▶ les valeurs de ces variables sont échangées par swap()
- ▶ les variables sont supprimées de l'environnement au terme de swap()
- ▶ les valeurs de j1 et j2 sont inchangées

Pour modifier des paramètres... passez des pointeurs

```
/* \ 'echange deux entiers */
void swap(int * i1, int * i2)
{
    int i = *i1;
    *i1 = *i2;
    *i2 = i;
}
int j1 = 5, j2 = 10;
swap(&j1, &j2);
printf("j1=%i -- j2=%i\n", j1, j2);
```

- ▶ l'appel à swap() recopie les valeurs *des adresses* de i1 et i2 dans deux variables locales
- ▶ le *contenu des cellules* pointées par ces variables sont échangées
- ▶ les variables locales sont supprimées au terme de swap()
- ▶ les valeurs de j1 et j2 ont été échangées (à leur insu !)

Inverse-t-y ou inverse-t-y pas ?

```
/*
    Inverse (?) une cha\^ine :
    invert("invert") = "trevni"
*/
void invert(char * s) {
    char temp;
    int l = strlen(s) - 1;
    for (i = 0; i <= l/2; i++) {
        temp = *(s+i); *(s+i) = *(s+l-i); *(s+l-i) = temp;
    }
}
char invert[7] = "invert";
invert(invert);
printf("%s\n", invert);
```

- ▶ invert (invert) recopie invert dans la variable locale s ;
- ▶ dont la valeur est l'adresse du tableau invert : *s a la valeur 'i'
- ▶ *(si) = *(s+l-i)+ échange les valeurs : l'inversion a lieu
- ▶ le passage des tableaux en argument correspond au passage de pointeurs

Inverse-t-y ou inverse-t-y pas ?

```

/*
    Inverse (?) une cha\^ine :
    invert("invert") = "trevni"
*/
void invert(char * s) {
    char temp;
    int l = strlen(s) - 1;
    for (i = 0; i <= l/2; i++) {
        temp = *(s+i); *(s+i) = *(s+l-i); *(s+l-i) = temp;
    }
}
char invert[7] = "invert";
invert(invert);
printf("%s\n", invert);

```

- ▶ invert(invert) recopie invert dans la variable locale s ;
- ▶ dont la valeur est l'adresse du tableau invert : *s a la valeur 'i'
- ▶ *(si) = *(s+l-i)+ échange les valeurs : l'inversion a lieu
- ▶ le passage des tableaux en argument correspond au passage de pointeurs

Traitement des chaînes : caractère par caractère

```
void mystrcpy1(char *s, char *t) {
    while (*t != 0) { *s = *t; s++; t++; }
}

int mystrcpy2(char *s, char *t) {
    while (*t != 0) { *s++ = *t++; }
}

int mystrcpy3(char *s, char *t) {
    while (*s++ = *t++);
}
```

- ▶ mystrcpy1 sépare copie et incrémantation
- ▶ $*s++ = *t++$ affecte puis incrémenté
- ▶ $(*s++ = *t++)$ vaut la valeur de la partie droite

Allocation d'un tableau : à l'ancienne

```
/*
     Croit allouer un tableau
*/
void alloc_table(int * tab , int size) {
    tab = (int *)malloc(size*sizeof(int));
    if (tab == NULL)
        printf("erreur de l'allocation\n");
}
int * table = NULL;
alloc_table(table);
```

- ▶ l'appel à `alloc_table ()` recopie la valeur de `tab` dans une variable locale `tab` ;
- ▶ dont la valeur est l'adresse du tableau `table`
- ▶ l'affectation `tab = (int *)malloc (...)` alloue une zone de mémoire
- ▶ dont l'adresse est perdue au retour de la fonction !
- ▶ une version correcte : `void alloc_table (int **tab, int size),`
`alloc_table (&table) ;`

Allocation d'un tableau : à l'ancienne

```
/*
     Croit allouer un tableau
*/
void alloc_table(int * tab , int size) {
    tab = (int *)malloc(size*sizeof(int));
    if (tab == NULL)
        printf("erreur de l'allocation\n");
}
int * table = NULL;
alloc_table(table);
```

- ▶ l'appel à alloc_table () recopie la valeur de tab dans une variable locale tab ;
- ▶ dont la valeur est l'adresse du tableau table
- ▶ l'affectation tab = (**int** *)malloc (...) alloue une zone de mémoire
- ▶ dont l'adresse est perdue au retour de la fonction !
- ▶ une version correcte : **void** alloc_table (**int** **tab, **int** size),
alloc_table (&table) ;

Allocation d'un tableau : à l'ancienne

```
/*
     Croit allouer un tableau
*/
void alloc_table(int * tab , int size) {
    tab = (int *)malloc(size*sizeof(int));
    if (tab == NULL)
        printf("erreur de l'allocation\n");
}
int * table = NULL;
alloc_table(table);
```

- ▶ l'appel à alloc_table () recopie la valeur de tab dans une variable locale tab ;
- ▶ dont la valeur est l'adresse du tableau table
- ▶ l'affectation tab = (**int** *)malloc (...) alloue une zone de mémoire
- ▶ dont l'adresse est perdue au retour de la fonction !
- ▶ une version correcte : **void** alloc_table (**int** **tab, **int** size),
alloc_table (&table) ;

Allocation d'un tableau : à l'ancienne

```
/*
     Croit allouer un tableau
*/
void alloc_table(int * tab , int size) {
    tab = (int *)malloc(size*sizeof(int));
    if (tab == NULL)
        printf("erreur de l'allocation\n");
}
int * table = NULL;
alloc_table(table);
```

- ▶ l'appel à alloc_table () recopie la valeur de tab dans une variable locale tab ;
- ▶ dont la valeur est l'adresse du tableau table
- ▶ l'affectation tab = (**int** *)malloc (...) alloue une zone de mémoire
- ▶ **dont l'adresse est perdue au retour de la fonction !**
- ▶ une version correcte : **void** alloc_table (**int** **tab, **int** size),
alloc_table (&table) ;

Allocation d'un tableau : à l'ancienne

```
/*
     Croit allouer un tableau
*/
void alloc_table(int * tab , int size) {
    tab = (int *)malloc(size*sizeof(int));
    if (tab == NULL)
        printf("erreur de l'allocation\n");
}
int * table = NULL;
alloc_table(table);
```

- ▶ l'appel à alloc_table () recopie la valeur de tab dans une variable locale tab ;
- ▶ dont la valeur est l'adresse du tableau table
- ▶ l'affectation tab = (**int** *)malloc (...) alloue une zone de mémoire
- ▶ **dont l'adresse est perdue au retour de la fonction !**
- ▶ **une version correcte :** **void** alloc_table (**int** **tab, **int** size),
alloc_table (&table) ;

Quand les fonctions deviennent variables : pointeurs sur fonctions

```
typedef struct {
    char * nom;
    char * prenom;
} personne;
int personnecmp(const personne *p1, const *p2) {
    int i = strcmp(p1->nom, p2->nom);
    return (i == 0 ? strcmp(p1->prenom, p2->prenom) : i);
}
personne annuaire[100];
qsort(annuaire, 100, sizeof(personne), personnecmp);
```

- ▶ le nom d'une fonction est comme une adresse
- ▶ qsort est une fonction de tri *générique* (dans stdlib.h)
- ▶ pour classer des personnes, faut les comparer, hence personnecmp
- ▶

```
qsort(void *base, size_t nmemb, size_t size,
      int (*compar)(const void *, const void *));
```

Quand les fonctions deviennent variables : pointeurs sur fonctions

```
typedef struct {
    char * nom;
    char * prenom;
} personne;
int personnecmp(const personne *p1, const *p2) {
    int i = strcmp(p1->nom, p2->nom);
    return (i == 0 ? strcmp(p1->prenom, p2->prenom) : i);
}
personne annuaire[100];
qsort(annuaire, 100, sizeof(personne), personnecmp);
```

- ▶ le nom d'une fonction est comme une adresse
- ▶ qsort est une fonction de tri générique (dans stdlib.h)
- ▶ pour classer des personnes, faut les comparer, hence personnecmp
- ▶

```
qsort(void *base, size_t nmemb, size_t size,
      int (*compar)(const void *, const void *));
```

Quand les fonctions deviennent variables : pointeurs sur fonctions

```
typedef struct {
    char * nom;
    char * prenom;
} personne;
int personnecmp(const personne *p1, const *p2) {
    int i = strcmp(p1->nom, p2->nom);
    return (i == 0 ? strcmp(p1->prenom, p2->prenom) : i);
}
personne annuaire[100];
qsort(annuaire, 100, sizeof(personne), personnecmp);
```

- ▶ le nom d'une fonction est comme une adresse
- ▶ qsort est une fonction de tri *générique* (dans stdlib.h)
- ▶ pour classer des personnes, faut les comparer, hence personnecmp

▶

```
qsort(void *base, size_t nmemb, size_t size,
      int (*compar)(const void *, const void *));
```

Quand les fonctions deviennent variables : pointeurs sur fonctions

```
typedef struct {
    char * nom;
    char * prenom;
} personne;
int personnecmp(const personne *p1, const *p2) {
    int i = strcmp(p1->nom, p2->nom);
    return (i == 0 ? strcmp(p1->prenom, p2->prenom) : i);
}
personne annuaire[100];
qsort(annuaire, 100, sizeof(personne), personnecmp);
```

- ▶ le nom d'une fonction est comme une adresse
- ▶ qsort est une fonction de tri *générique* (dans stdlib.h)
- ▶ pour classer des personnes, faut les comparer, hence personnecmp
- ▶ `qsort(void *base, size_t nmemb, size_t size,
 int (*compar)(const void *, const void *));`

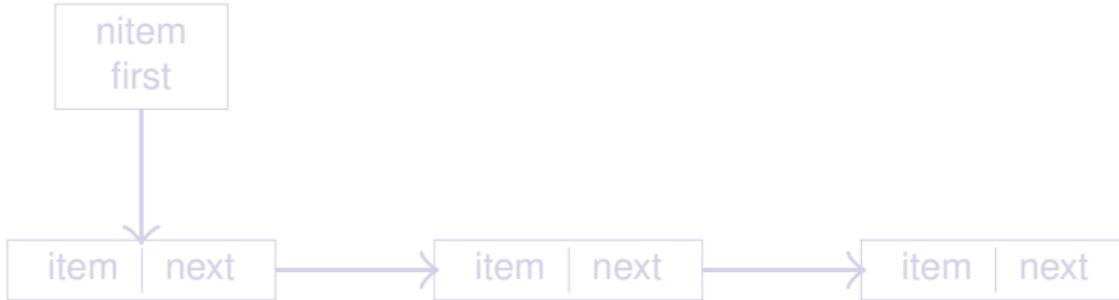
Fonctions au choix

```
double cosinus(double x) { return cos(x);}
double sinus(double) {return sin(x);}
double (*tab)[2](double) = {cosinus, sinus};
int main() {
    double pi = 3.1416;
    char 'c' = (char)getchar();
    switch(c) {
        case ('c'): printf("%f\n", (tab[0])(pi)); break;
        case ('s'): printf("%f\n", (tab[1])(pi)); break;
        default: break;
    }
}
```

Un type dynamique : la liste

La liste

- ▶ représente une séquence ou un ensemble ou une pile ou ...
- ▶ insérer/supprimer un élément
- ▶ chercher un élément dans la liste
- ▶ intersection, union...



Un type dynamique : la liste

La liste

- ▶ représente une séquence ou un ensemble ou une pile ou ...
- ▶ insérer/supprimer un élément
- ▶ chercher un élément dans la liste
- ▶ intersection, union...



Un type dynamique : la liste

La liste

- ▶ représente une séquence ou un ensemble ou une pile ou ...
- ▶ insérer/supprimer un élément
- ▶ chercher un élément dans la liste
- ▶ intersection, union...



Un type dynamique : la liste

La liste

- ▶ représente une séquence ou un ensemble ou une pile ou ...
- ▶ insérer/supprimer un élément
- ▶ chercher un élément dans la liste
- ▶ intersection, union...



Un type dynamique : la liste

La liste

- ▶ représente une séquence ou un ensemble ou une pile ou ...
- ▶ insérer/supprimer un élément
- ▶ chercher un élément dans la liste
- ▶ intersection, union...



Un type dynamique : la liste

La liste

- ▶ représente une séquence ou un ensemble ou une pile ou ...
- ▶ insérer/supprimer un élément
- ▶ chercher un élément dans la liste
- ▶ intersection, union...



Un type dynamique : la liste

La liste

- ▶ représente une séquence ou un ensemble ou une pile ou ...
- ▶ insérer/supprimer un élément
- ▶ chercher un élément dans la liste
- ▶ intersection, union...



Les types de la liste

```
// pour une liste d'int
typedef val int;  
  
typedef struct {  
    val value;  
    cell * next;  
} cell;  
  
typedef struct {  
    int nitem;  
    cell * first;  
} list;
```

Créer et détruire une liste

```
/* NewList(): alloc and init new list */
list * NewList(void) {
    list * mylist = NULL;
    if ((mylist = malloc(sizeof(list))) == NULL) {
        /* Problem with malloc */
    }
    mylist->nitem = 0;
    mylist->first = NULL;
    return mylist;
}
```

```
/* DelList(mylist): empty and free list */
void DelList(list * mylist) {
    if (mylist != NULL) {
        if (nitem !=0 && mylist->head != NULL)
            EmptyList(*mylist);
        free(mylist);
    }
}
```

Créer et détruire une liste

```
/* NewList(): alloc and init new list */
list * NewList(void) {
    list * mylist = NULL;
    if ((mylist = malloc(sizeof(list))) == NULL) {
        /* Problem with malloc */
    }
    mylist->nitem = 0;
    mylist->first = NULL;
    return mylist;
}
```

```
/* DelList(mylist): empty and free list */
void DelList(list * mylist) {
    if (mylist != NULL) {
        if (nitem !=0 && mylist->head != NULL)
            EmptyList(*mylist);
        free(mylist);
    }
}
```

Ajouter et supprimer des maillons

```
/* AddVal(mylist, value): add val in mylist */
void AddVal(list * mylist, int value) {
    cell * newcell;
    if ((newcell = malloc(sizeof(cell))) == NULL) {
        /* Problem with malloc */
    }
    newcell->val = value; newcell->next = list->head;
    list->head = newcell;
    list->nitem++;
}
```

```
/* EmptyList(mylist): empty and free list */
void EmptyList(cell * head) {
    if (head->next != NULL)
        EmptyList(head->next);
    free(head);
}
```

Ajouter et supprimer des maillons

```
/* AddVal(mylist, value): add val in mylist */
void AddVal(list * mylist, int value) {
    cell * newcell;
    if ((newcell = malloc(sizeof(cell))) == NULL) {
        /* Problem with malloc */
    }
    newcell->val = value; newcell->next = list->head;
    list->head = newcell;
    list->nitem++;
}
```

```
/* EmptyList(mylist): empty and free list */
void EmptyList(cell * head) {
    if (head->next != NULL)
        EmptyList(head->next);
    free(head);
}
```

Transformation tableau vers liste

```
/* DelVal(mylist): remove all occurrences of Val */
void DelVal(list * mylist, val value) {
    cell * cell = list->head, father = NULL;
    while (cell != NULL) {
        if (cell->val == value) {
            if (father == NULL) list->head = cell->next;
            else father->cell = cell->next;
            (list->nitem)--; free(cell);
        }
        father = cell; cell = cell->next;
    }
}
```

```
/* Tab2List() : convert a table of int into a list */
list * Tab2List(int * tab, int dim) {
    list * mylist = NewList();
    if (list == NULL) return NULL;
    for (int i=0; i<dim; i++) Addval(mylist, tab[i]);
    return(mylist);
}
```

Transformation tableau vers liste

```
/* DelVal(mylist): remove all occurrences of Val */
void DelVal(list * mylist, val value) {
    cell * cell = list->head, father = NULL;
    while (cell != NULL) {
        if (cell->val == value) {
            if (father == NULL) list->head = cell->next;
            else father->cell = cell->next;
            (list->nitem)--; free(cell);
        }
        father = cell; cell = cell->next;
    }
}
```

```
/* Tab2List() : convert a table of int into a list */
list * Tab2List(int * tab, int dim) {
    list * mylist = NewList();
    if (list == NULL) return NULL;
    for (int i=0; i<dim; i++) Addval(mylist, tab[i]);
    return(mylist);
}
```

Récréations

- ▶ Insérer en fin de liste
- ▶ Concaténer, intersester deux listes
- ▶ Supprimer seulement le premier, le dernier
- ▶ Fusionner deux listes (sans doublon)
- ▶ ...
- ▶ Avec deux successeurs : les arbres

Récréations

- ▶ Insérer en fin de liste
- ▶ Concaténer, intersester deux listes
- ▶ Supprimer seulement le premier, le dernier
- ▶ Fusionner deux listes (sans doublon)
- ▶ ...
- ▶ Avec deux successeurs : les arbres

Récréations

- ▶ Insérer en fin de liste
- ▶ Concaténer, intersester deux listes
- ▶ Supprimer seulement le premier, le dernier
- ▶ Fusionner deux listes (sans doublon)
- ▶ ...
- ▶ Avec deux successeurs : les arbres

Récréations

- ▶ Insérer en fin de liste
- ▶ Concaténer, intersester deux listes
- ▶ Supprimer seulement le premier, le dernier
- ▶ Fusionner deux listes (sans doublon)
- ▶ ...
- ▶ Avec deux successeurs : les arbres

Récréations

- ▶ Insérer en fin de liste
- ▶ Concaténer, intersester deux listes
- ▶ Supprimer seulement le premier, le dernier
- ▶ Fusionner deux listes (sans doublon)
- ▶ ...
- ▶ Avec deux successeurs : les arbres

Récréations

- ▶ Insérer en fin de liste
- ▶ Concaténer, intersester deux listes
- ▶ Supprimer seulement le premier, le dernier
- ▶ Fusionner deux listes (sans doublon)
- ▶ ...
- ▶ Avec deux successeurs : les arbres

Le(s) thème(s) du jour: La chaîne de production

Du programme à l'exécutable

1. Suppression des espaces, des lignes vides, des commentaires
2. Préprocesseur : inclusion des fichiers `#include`, expansion des *macros* et des *directives de compilation conditionnelles*
`gcc -E monprog.c > toutmonprog.c` (cf. aussi `cpp`)
3. Compilation et assemblage : création des objets en langage machine (plusieurs phases)
`gcc monprog.c -c -o monprog.o`
4. Edition de liens : calcul d'adresses, résolution des références, inclusion du code des fonctions "externes"
`gcc monprog.o -o monprog` (cf. aussi `ld`)
5. Chargement en mémoire : l'action du chargeur

Du programme à l'exécutable

1. Suppression des espaces, des lignes vides, des commentaires
2. Préprocesseur : inclusion des fichiers **#include**, expansion des *macros* et des *directives de compilation conditionnelles*
gcc –E monprog.c > toutmonprog.c (cf. aussi cpp)
3. Compilation et assemblage : création des objets en langage machine (plusieurs phases)
gcc monprog.c –c –o monprog.o
4. Edition de liens : calcul d'adresses, résolution des références, inclusion du code des fonctions “externes”
gcc monprog.o –o monprog (cf. aussi ld)
5. Chargement en mémoire : l'action du chargeur

Du programme à l'exécutable

1. Suppression des espaces, des lignes vides, des commentaires
2. Préprocesseur : inclusion des fichiers **#include**, expansion des *macros* et des *directives de compilation conditionnelles*
`gcc -E monprog.c > toutmonprog.c` (cf. aussi `cpp`)
3. **Compilation et assemblage : création des objets en langage machine (plusieurs phases)**
`gcc monprog.c -c -o monprog.o`
4. Edition de liens : calcul d'adresses, résolution des références, inclusion du code des fonctions "externes"
`gcc monprog.o -o monprog` (cf. aussi `ld`)
5. Chargement en mémoire : l'action du chargeur

Du programme à l'exécutable

1. Suppression des espaces, des lignes vides, des commentaires
2. Préprocesseur : inclusion des fichiers **#include**, expansion des *macros* et des *directives de compilation conditionnelles*
`gcc -E monprog.c > toutmonprog.c` (cf. aussi `cpp`)
3. Compilation et assemblage : création des objets en langage machine (plusieurs phases)
`gcc monprog.c -c -o monprog.o`
4. Edition de liens : calcul d'adresses, résolution des références, inclusion du code des fonctions "externes"
`gcc monprog.o -o monprog` (cf. aussi `ld`)
5. Chargement en mémoire : l'action du chargeur

Du programme à l'exécutable

1. Suppression des espaces, des lignes vides, des commentaires
2. Préprocesseur : inclusion des fichiers **#include**, expansion des *macros* et des *directives de compilation conditionnelles*
`gcc -E monprog.c > toutmonprog.c` (cf. aussi `cpp`)
3. Compilation et assemblage : création des objets en langage machine (plusieurs phases)
`gcc monprog.c -c -o monprog.o`
4. Edition de liens : calcul d'adresses, résolution des références, inclusion du code des fonctions "externes"
`gcc monprog.o -o monprog` (cf. aussi `ld`)
5. Chargement en mémoire : l'action du chargeur

Finition du code : Les œuvres du préprocesseur

- ▶ Insertion littérale (réursive) des fichiers inclus
 - ▶ `#include <header.h>` (accessibles via le PATH ou via `gcc -Iincludepath`)
 - ▶ `#include "header.h"` (dans le répertoire courant)
 - ▶ `#include "../include/header.h"` (...)
- ⇒ connaissance des prototypes
- ▶ Expansion des macros
- ▶ Traitement des directives de précompilation (`#ifdef`, `#ifndef`, `#endif`)

Finition du code : Les œuvres du préprocesseur

- ▶ Insertion littérale (réursive) des fichiers inclus
 - ▶ **#include** <header.h> (accessibles via le PATH ou via gcc –Iincludepath)
 - ▶ **#include** "header.h" (dans le répertoire courant)
 - ▶ **#include** "../include/header.h" (...)
- ⇒ connaissance des prototypes
- ▶ Expansion des macros
- ▶ Traitement des directives de précompilation (**#ifdef**, **#ifndef**, **#endif**)

Finition du code : Les œuvres du préprocesseur

- ▶ Insertion littérale (réursive) des fichiers inclus
 - ▶ #include <header.h> (accessibles via le PATH ou via gcc –Iincludepath)
 - ▶ #include "header.h" (dans le répertoire courant)
 - ▶ #include "../include/header.h" (...)
- ⇒ connaissance des prototypes
- ▶ Expansion des macros
- ▶ Traitement des directives de précompilation (#ifdef, #ifndef, #endif)

Finition du code : Les œuvres du préprocesseur

- ▶ Insertion littérale (réursive) des fichiers inclus
 - ▶ **#include** <header.h> (accessibles via le PATH ou via gcc –lincludepath)
 - ▶ **#include** "header.h" (dans le répertoire courant)
 - ▶ **#include** "../include/header.h" (...)
- ⇒ connaissance des prototypes
- ▶ Expansion des macros
- ▶ Traitement des directives de précompilation (**#ifdef**, **#ifndef**, **#endif**)

Finition du code : Les œuvres du préprocesseur

- ▶ Insertion littérale (réursive) des fichiers inclus
 - ▶ **#include** <header.h> (accessibles via le PATH ou via gcc –Iincludepath)
 - ▶ **#include** "header.h" (dans le répertoire courant)
 - ▶ **#include** "../include/header.h" (...)
- ⇒ connaissance des prototypes
- ▶ Expansion des macros
- ▶ Traitement des directives de précompilation (**#ifdef**, **#ifndef**, **#endif**)

Finition du code : Les œuvres du préprocesseur

- ▶ Insertion littérale (réursive) des fichiers inclus
 - ▶ #include <header.h> (accessibles via le PATH ou via gcc –lincludepath)
 - ▶ #include "header.h" (dans le répertoire courant)
 - ▶ #include "../include/header.h" (...)
⇒ connaissance des prototypes
- ▶ Expansion des macros
- ▶ Traitement des directives de précompilation
(#ifdef, #ifndef, #endif)

Le langage des macros

```
#define PI 3.1416
#define FAC "Orsay"
#define SQUARE(x) (x*x)

...
char * faculte = FAC;      /* faculte = "Orsay"; */
float circ = 2 * PI;        /* circ = 2 * 3.1416   */
float surf = PI * SQR(R); /* surf = 3.1416 * (R*R) */
```

- ▶ `#define NOM stuff` : remplace NOM par stuff
- ▶ stuff doit être au moins un *token* complet
- ▶ `#a` crée une chaîne ; `a ## b` fusionne a et b ;
- ▶ `#define NOM(x) stuff(x)` : remplace NOM(X) par stuff (x)
- ▶ les remplacements sont récursifs
- ▶ ⇒ utile pour la définition de constantes ou métavariables

Le langage des macros

```
#define PI 3.1416
#define FAC "Orsay"
#define SQUARE(x) (x*x)

...
char * faculte = FAC;      /* faculte = "Orsay"; */
float circ = 2 * PI;        /* circ = 2 * 3.1416   */
float surf = PI * SQR(R); /* surf = 3.1416 * (R*R) */
```

- ▶ **#define** NOM stuff : remplace NOM par stuff
- ▶ stuff doit être au moins un *token* complet
- ▶ #a crée une chaîne ; a ## b fusionne a et b ;
- ▶ **#define** NOM(x) stuff(x) : remplace NOM(X) par stuff (x)
- ▶ les remplacements sont récursifs
- ▶ ⇒ utile pour la définition de constantes ou métavariables

Le langage des macros

```
#define PI 3.1416
#define FAC "Orsay"
#define SQUARE(x) (x*x)

...
char * faculte = FAC;      /* faculte = "Orsay"; */
float circ = 2 * PI;        /* circ = 2 * 3.1416   */
float surf = PI * SQR(R); /* surf = 3.1416 * (R*R) */
```

- ▶ **#define** NOM stuff : remplace NOM par stuff
- ▶ stuff doit être au moins un *token* complet
- ▶ #a crée une chaîne ; a ## b fusionne a et b ;
- ▶ **#define** NOM(x) stuff(x) : remplace NOM(X) par stuff (x)
- ▶ les remplacements sont récursifs
- ▶ ⇒ utile pour la définition de constantes ou métavariables

Le langage des macros

```
#define PI 3.1416
#define FAC "Orsay"
#define SQUARE(x) (x*x)

...
char * faculte = FAC;      /* faculte = "Orsay"; */
float circ = 2 * PI;        /* circ = 2 * 3.1416   */
float surf = PI * SQR(R); /* surf = 3.1416 * (R*R) */
```

- ▶ **#define** NOM stuff : remplace NOM par stuff
- ▶ stuff doit être au moins un *token* complet
- ▶ **#a** créé une chaîne ; **a ## b** fusionne a et b ;
- ▶ **#define** NOM(x) stuff(x) : remplace NOM(X) par stuff (x)
- ▶ les remplacements sont récursifs
- ▶ ⇒ utile pour la définition de constantes ou métavariables

Le langage des macros

```
#define PI 3.1416
#define FAC "Orsay"
#define SQUARE(x) (x*x)

...
char * faculte = FAC;      /* faculte = "Orsay"; */
float circ = 2 * PI;        /* circ = 2 * 3.1416   */
float surf = PI * SQR(R); /* surf = 3.1416 * (R*R) */
```

- ▶ **#define** NOM stuff : remplace NOM par stuff
- ▶ stuff doit être au moins un *token* complet
- ▶ **#a** créé une chaîne ; a ## b fusionne a et b ;
- ▶ **#define** NOM(x) stuff(x) : remplace NOM(X) par stuff (x)
- ▶ les remplacements sont récursifs
- ▶ ⇒ utile pour la définition de constantes ou métavariables

Le langage des macros

```
#define PI 3.1416
#define FAC "Orsay"
#define SQUARE(x) (x*x)

...
char * faculte = FAC;      /* faculte = "Orsay"; */
float circ = 2 * PI;        /* circ = 2 * 3.1416   */
float surf = PI * SQR(R); /* surf = 3.1416 * (R*R) */
```

- ▶ **#define** NOM stuff : remplace NOM par stuff
- ▶ stuff doit être au moins un *token* complet
- ▶ **#a** créé une chaîne ; a ## b fusionne a et b ;
- ▶ **#define** NOM(x) stuff(x) : remplace NOM(X) par stuff (x)
- ▶ les remplacements sont récursifs
- ▶ ⇒ utile pour la définition de constantes ou métavariables

Le langage des macros

```
#define PI 3.1416
#define FAC "Orsay"
#define SQUARE(x) (x*x)

...
char * faculte = FAC;      /* faculte = "Orsay"; */
float circ = 2 * PI;        /* circ = 2 * 3.1416   */
float surf = PI * SQR(R); /* surf = 3.1416 * (R*R) */
```

- ▶ **#define** NOM stuff : remplace NOM par stuff
- ▶ stuff doit être au moins un *token* complet
- ▶ **#a** créé une chaîne ; a ## b fusionne a et b ;
- ▶ **#define** NOM(x) stuff(x) : remplace NOM(X) par stuff (x)
- ▶ les remplacements sont récursifs
- ▶ ⇒ utile pour la définition de constantes ou métavariables

Respirations

▶ play feuer71

Récréations

- ▶ Définir une macro swap(*t*, *x*, *y*) qui échange deux arguments *x* et *y* de type *t*.



Récréations

- ▶ Définir une macro swap(*t*, *x*, *y*) qui échange deux arguments *x* et *y* de type *t*.

Macros ou fonctions ?

```
#define MeanMac(x,y) (x+y/(double)2)
double MeanFun(double x, double y) {
    return (x+y)/2;
}
```

► Avantage macro :

- ▶ l'expansion est faire à la compilation : gain de temps
- ▶ "marche" pour tous les types d'arguments

► Avantage fonction :

- ▶ lisibilité, contrôle des types, prototypage
- ▶ possibilité d'*Inlining* à la compilation

Macros ou fonctions ?

```
#define MeanMac(x,y) (x+y/(double)2)
double MeanFun(double x, double y) {
    return (x+y)/2;
}
```

► Avantage macro :

- ▶ l'expansion est faire à la compilation : gain de temps
- ▶ "marche" pour tous les types d'arguments

► Avantage fonction :

- ▶ lisibilité, contrôle des types, prototypage
- ▶ possibilité d'*Inlining* à la compilation

Macros ou fonctions ?

```
#define MeanMac(x,y) (x+y/(double)2)
double MeanFun(double x, double y) {
    return (x+y)/2;
}
```

► Avantage macro :

- ▶ l'expansion est faire à la compilation : gain de temps
- ▶ "marche" pour tous les types d'arguments

► Avantage fonction :

- ▶ lisibilité, contrôle des types, prototypage
- ▶ possibilité d'*Inlining* à la compilation

Macros ou fonctions ?

```
#define MeanMac(x,y) (x+y/(double)2)
double MeanFun(double x, double y) {
    return (x+y)/2;
}
```

► Avantage macro :

- ▶ l'expansion est faire à la compilation : gain de temps
- ▶ "marche" pour tous les types d'arguments

► Avantage fonction :

- ▶ lisibilité, contrôle des types, prototypage
- ▶ possibilité *d'inlining* à la compilation

Macros ou fonctions ?

```
#define MeanMac(x, y) (x+y/(double)2)
double MeanFun(double x, double y) {
    return (x+y)/2;
}
```

► Avantage macro :

- ▶ l'expansion est faire à la compilation : gain de temps
- ▶ "marche" pour tous les types d'arguments

► Avantage fonction :

- ▶ lisibilité, contrôle des types, prototypage
- ▶ possibilité *d'inlining* à la compilation

Macros ou fonctions ?

```
#define MeanMac(x, y) (x+y/(double)2)
double MeanFun(double x, double y) {
    return (x+y)/2;
}
```

► Avantage macro :

- ▶ l'expansion est faire à la compilation : gain de temps
- ▶ "marche" pour tous les types d'arguments

► Avantage fonction :

- ▶ lisibilité, contrôle des types, prototypage
- ▶ possibilité *d'inlining* à la compilation

Compilation à la demande : l'art du portage

```
#ifdef __HAVE_STDDEF_H__  
#include <stddef.h>  
#else  
#define NULL ('\0')  
#endif
```

- ▶ #ifdef VAR est vrai si la variable VAR a été définie (contraire : #ifndef)
- ▶ #if CONDITION ... #elif ... #endif avec COND s'évaluant à une constante entière (après expansion)
- ▶ #error MSG permet de provoquer des erreurs du préprocesseur

▶ voir /usr/include/stdio.h

Compilation à la demande : l'art du portage

```
#ifdef __HAVE_STDDEF_H__  
    #include<stddef.h>  
#else  
    #define NULL ('\0')  
#endif
```

- ▶ **#ifdef** VAR est vrai si la variable VAR a été définie (contraire : **#ifndef**)
- ▶ **#if** CONDITION ... **#elif** ... **#endif** avec COND s'évaluant à une constante entière (après expansion)
- ▶ **#error** MSG permet de provoquer des erreurs du préprocesseur

▶ voir /usr/include/stdio.h

Compilation à la demande : l'art du portage

```
#ifdef __HAVE_STDDEF_H__  
    #include<stddef.h>  
#else  
    #define NULL ('\0')  
#endif
```

- ▶ **#ifdef** VAR est vrai si la variable VAR a été définie (contraire : **#ifndef**)
- ▶ **#if** CONDITION ... **#elif** ... **#endif** avec COND s'évaluant à une constante entière (après expansion)
- ▶ **#error** MSG permet de provoquer des erreurs du préprocesseur

▶ voir /usr/include/stdio.h

Compilation à la demande : l'art du portage

```
#ifdef __HAVE_STDDEF_H__  
#include <stddef.h>  
#else  
#define NULL ('\0')  
#endif
```

- ▶ **#ifdef** VAR est vrai si la variable VAR a été définie (contraire : **#ifndef**)
- ▶ **#if** CONDITION ... **#elif** ... **#endif** avec COND s'évaluant à une constante entière (après expansion)
- ▶ **#error** MSG permet de provoquer des erreurs du préprocesseur

voir /usr/include/stdio.h

Compilation à la demande : l'art du portage

```
#ifdef __HAVE_STDDEF_H__  
#include <stddef.h>  
#else  
#define NULL ('\0')  
#endif
```

- ▶ **#ifdef** VAR est vrai si la variable VAR a été définie (contraire : **#ifndef**)
- ▶ **#if** CONDITION ... **#elif** ... **#endif** avec COND s'évaluant à une constante entière (après expansion)
- ▶ **#error** MSG permet de provoquer des erreurs du préprocesseur

► voir /usr/include/stdio.h

Compilation à la demande : l'art du portage

```
#ifdef __HAVE_STDDEF_H__  
#include <stddef.h>  
#else  
#define NULL ('\0')  
#endif
```

- ▶ **#ifdef** VAR est vrai si la variable VAR a été définie (contraire : **#ifndef**)
- ▶ **#if** CONDITION ... **#elif** ... **#endif** avec COND s'évaluant à une constante entière (après expansion)
- ▶ **#error** MSG permet de provoquer des erreurs du préprocesseur

▶ voir /usr/include/stdio.h

Quelques constantes “offertes”

- ▶ `__DATE__` : date de la compilation
- ▶ `__TIME__` : heure de la compilation
- ▶ `__FILE__` : nom du fichier en cours de compilation
- ▶ `__LINE__` : ligne courante
- ▶ plus de nombreuses constantes dépendant de l'OS et de l'environnement
- ▶ et les constantes passées à la compilation gcc –DVAR=Valeur

```
printf("error on line __LINE__");
```

Quelques constantes “offertes”

- ▶ `__DATE__` : date de la compilation
- ▶ `__TIME__` : heure de la compilation
- ▶ `__FILE__` : nom du fichier en cours de compilation
- ▶ `__LINE__` : ligne courante
- ▶ plus de nombreuses constantes dépendant de l'OS et de l'environnement
- ▶ et les constantes passées à la compilation `gcc -DVAR=Valeur`

```
printf("error on line __LINE__");
```

L'assemblage final : l'édition des liens

- ▶ Fusion des tables de symboles, recalcul des adresses
- ▶ Résolution des symboles externes : exploration des bibliothèques
- ▶ Une bibliothèque = répertoire de fonctions définies par leurs E/S
- ▶ Une bibliothèque = un ou plusieurs fichiers de prototypes
- ▶ Trois types de bibliothèque :
 - ▶ la bibliothèque *standard*, explorée par défaut
 - ▶ les bibliothèques non-standard `libpath/libXXXX` : utiliser l'option `(g)cc --Libpath --IXXXX`. Exemple avec la bibliothèque mathématique `(g)cc --Im`

L'assemblage final : l'édition des liens

- ▶ Fusion des tables de symboles, recalcul des adresses
- ▶ Résolution des symboles externes : exploration des bibliothèques
 - ▶ Une bibliothèque = répertoire de fonctions définies par leurs E/S
 - ▶ Une bibliothèque = un ou plusieurs fichiers de prototypes
 - ▶ Trois types de bibliothèque :
 - ▶ la bibliothèque *standard*, explorée par défaut
 - ▶ les bibliothèques non-standard `libpath/libXXXX` : utiliser l'option `(g)cc --Libpath --IXXXX`. Exemple avec la bibliothèque mathématique `(g)cc --lm`

L'assemblage final : l'édition des liens

- ▶ Fusion des tables de symboles, recalcul des adresses
- ▶ Résolution des symboles externes : exploration des bibliothèques
- ▶ Une bibliothèque = répertoire de fonctions définies par leurs E/S
- ▶ Une bibliothèque = un ou plusieurs fichiers de prototypes
- ▶ Trois types de bibliothèque :
 - ▶ la bibliothèque *standard*, explorée par défaut
 - ▶ les bibliothèques non-standard `libpath/libXXXX` : utiliser l'option `(g)cc --Libpath --IXXXX`. Exemple avec la bibliothèque mathématique `(g)cc --Im`

L'assemblage final : l'édition des liens

- ▶ Fusion des tables de symboles, recalcul des adresses
- ▶ Résolution des symboles externes : exploration des bibliothèques
- ▶ Une bibliothèque = répertoire de fonctions définies par leurs E/S
- ▶ Une bibliothèque = un ou plusieurs fichiers de prototypes
- ▶ Trois types de bibliothèque :
 - ▶ la bibliothèque *standard*, explorée par défaut
 - ▶ les bibliothèques non-standard *libpath/libXXXX* : utiliser l'option `(g)cc --Libpath --IXXXX`. Exemple avec la bibliothèque mathématique `(g)cc --Im`

L'assemblage final : l'édition des liens

- ▶ Fusion des tables de symboles, recalcul des adresses
- ▶ Résolution des symboles externes : exploration des bibliothèques
- ▶ Une bibliothèque = répertoire de fonctions définies par leurs E/S
- ▶ Une bibliothèque = un ou plusieurs fichiers de prototypes
- ▶ Trois types de bibliothèque :
 - ▶ la bibliothèque *standard*, explorée par défaut
 - ▶ les bibliothèques non-standard `libpath/libXXXX` : utiliser l'option `(g)cc -Llibpath -IXXXX`. Exemple avec la bibliothèque mathématique `(g)cc -lmath`
 - ▶ les bibliothèques statiques (`.a`) : recopie intégrale du code des fonctions utilisées
 - ▶ les bibliothèques dynamiques (`*.dll`, `*.so`, `*.dylib`) : copie d'un pointeur vers le code des fonctions utilisées

L'assemblage final : l'édition des liens

- ▶ Fusion des tables de symboles, recalcul des adresses
- ▶ Résolution des symboles externes : exploration des bibliothèques
 - ▶ Une bibliothèque = répertoire de fonctions définies par leurs E/S
 - ▶ Une bibliothèque = un ou plusieurs fichiers de prototypes
 - ▶ Trois types de bibliothèque :
 - ▶ la bibliothèque *standard*, explorée par défaut
 - ▶ les bibliothèques non-standard `libpath/libXXXX` : utiliser l'option `(g)cc -Llibpath -IXXXX`. Exemple avec la bibliothèque mathématique `(g)cc -lmath`
 - ▶ les bibliothèques statiques (`.a`) : recopie intégrale du code des fonctions utilisées
 - ▶ les bibliothèques dynamiques (`*.dll`, `*.so`, `*.dylib`) : copie d'un pointeur vers le code des fonctions utilisées

L'assemblage final : l'édition des liens

- ▶ Fusion des tables de symboles, recalculation des adresses
- ▶ Résolution des symboles externes : exploration des bibliothèques
- ▶ Une bibliothèque = répertoire de fonctions définies par leurs E/S
- ▶ Une bibliothèque = un ou plusieurs fichiers de prototypes
- ▶ Trois types de bibliothèque :
 - ▶ la bibliothèque *standard*, explorée par défaut
 - ▶ les bibliothèques non-standard `libpath/libXXXX` : utiliser l'option `(g)cc -Llibpath -IXXXX`. Exemple avec la bibliothèque mathématique `(g)cc -lm`
 - ▶ les bibliothèques statiques (`.a`) : recopie intégrale du code des fonctions utilisées
 - ▶ les bibliothèques dynamiques (`*.dll`, `*.so`, `*.dylib`) : copie d'un pointeur vers le code des fonctions utilisées

L'assemblage final : l'édition des liens

- ▶ Fusion des tables de symboles, recalculation des adresses
- ▶ Résolution des symboles externes : exploration des bibliothèques
- ▶ Une bibliothèque = répertoire de fonctions définies par leurs E/S
- ▶ Une bibliothèque = un ou plusieurs fichiers de prototypes
- ▶ Trois types de bibliothèque :
 - ▶ la bibliothèque *standard*, explorée par défaut
 - ▶ les bibliothèques non-standard `libpath/libXXXX` : utiliser l'option `(g)cc -Llibpath -IXXXX`. Exemple avec la bibliothèque mathématique `(g)cc -lm`
 - ▶ les bibliothèques statiques (`.a`) : recopie intégrale du code des fonctions utilisées
 - ▶ les bibliothèques dynamiques (`*.dll`, `*.so`, `*.dylib`) : copie d'un pointeur vers le code des fonctions utilisées

L'assemblage final : l'édition des liens

- ▶ Fusion des tables de symboles, recalcul des adresses
- ▶ Résolution des symboles externes : exploration des bibliothèques
- ▶ Une bibliothèque = répertoire de fonctions définies par leurs E/S
- ▶ Une bibliothèque = un ou plusieurs fichiers de prototypes
- ▶ Trois types de bibliothèque :
 - ▶ la bibliothèque *standard*, explorée par défaut
 - ▶ les bibliothèques non-standard `libpath/libXXXX` : utiliser l'option `(g)cc -Llibpath -IXXXX`. Exemple avec la bibliothèque mathématique `(g)cc -lm`
 - ▶ les bibliothèques statiques (`.a`) : recopie intégrale du code des fonctions utilisées
 - ▶ les bibliothèques dynamiques (`*.dll`, `*.so`, `*.dylib`) : copie d'un pointeur vers le code des fonctions utilisées

Les arguments du main

```
#include <stdio.h>
int main(int argc, char *argv[])
{
    int i = 0;
    printf("%s a %d args.\n", argv[0], argc);

    for(i=0; i < argc && argv[i] != NULL; i++)
        printf("argument %d: %s\n", i);

}
```

[▶ play main](#)

- ▶ `argc` compte le nombre d'arguments
- ▶ `argv[]` : tableau des arguments de la commande débutant par la commande elle-même, terminé par un `NULL`.
- ▶ un argument supplémentaire `char ***` : l'environnement

Les arguments du main

```
#include <stdio.h>
int main(int argc, char *argv[]) {
    int i = 0;
    printf("%s a %d args.\n", argv[0], argc);

    for(i=0; i < argc && argv[i] != NULL; i++)
        printf("argument %d: %s\n", i);

}
```

[▶ play main](#)

- ▶ **argc** compte le nombre d'arguments
- ▶ **argv[]** : tableau des arguments de la commande débutant par la commande elle-même, terminé par un **NULL**
- ▶ un argument supplémentaire **char ***** : l'environnement

Les arguments du main

```
#include <stdio.h>
int main(int argc, char *argv[]) {
    int i = 0;
    printf("%s a %d args.\n", argv[0], argc);

    for(i=0; i < argc && argv[i] != NULL; i++)
        printf("argument %d: %s\n", i);

}
```

[▶ play main](#)

- ▶ argc compte le nombre d'arguments
- ▶ argv[] : tableau des arguments de la commande débutant par la commande elle-même, terminé par un NULL
- ▶ un argument supplémentaire char *** : l'environnement

Les arguments du main

```
#include <stdio.h>
int main(int argc, char *argv[]) {
    int i = 0;
    printf("%s a %d args.\n", argv[0], argc);

    for(i=0; i < argc && argv[i] != NULL; i++)
        printf("argument %d: %s\n", i);

}
```

[▶ play main](#)

- ▶ argc compte le nombre d'arguments
- ▶ argv[] : tableau des arguments de la commande débutant par la commande elle-même, terminé par un NULL
- ▶ un argument supplémentaire char *** : l'environnement

Des options standardisées : getopt()

```
#include<getopt.h>
int main(int argc, char * argv[]) {
    extern char * optarg; // cf. getopt.h
    extern int optind;    // idem
    while((c = getopt(argc, argv, "gc:o:")) != -1) {
        switch(c) {
            case 'c' : /* ... */ break;
            case 'g' : /* ... */ break;
            case 'o' : /* ... */ break;
            default:
                printf("unknown option for %s\n", argv[0]);
                break;
        }
    }
}
```

- ▶ Usage prog -g -c param2 -o param2
- ▶ optarg est un pointeur sur les paramètres
- ▶ optind donne le nombre d'arguments supplémentaires (accessibles par argv[optind]).

Des options standardisées : getopt()

```
#include<getopt.h>
int main(int argc, char * argv[]) {
    extern char * optarg; // cf. getopt.h
    extern int optind;    // idem
    while((c = getopt(argc, argv, "gc:o:")) != -1) {
        switch(c) {
            case 'c' : /* ... */ break;
            case 'g' : /* ... */ break;
            case 'o' : /* ... */ break;
            default:
                printf("unknown option for %s\n", argv[0]);
                break;
        }
    }
}
```

- ▶ Usage prog -g -c param2 -o param2
- ▶ optarg est un pointeur sur les paramètres
- ▶ optind donne le nombre d'arguments supplémentaires (accessibles par argv[optind]).

Le(s) thème(s) du jour: Visites de bibliothèques

Utiliser le travail des autres : les bibliothèques



La bibliothèque standard

- ▶ Les fonction d'IO : stdio.h
- ▶ Manipulation de la mémoire ; conversions diverses : stdlib.h
- ▶ Tests unitaires de char : ctype.h
- ▶ Manipulation des chaînes : string.h
- ▶ Manipulations de l'horloge time.h
- ▶ Constantes entières et réelles limits.h et float.h
- ▶ Entiers étendus inttypes.h, stdint.h
- ▶ Des contrôles : assert.h
- ▶ Les signaux : signal.h
- ▶ Gestion des erreurs : errno.h ...
- ▶ une implantation++ : la glib.c

Accès aux fichiers : concepts de base

- ▶ deux types pour les fichiers : « bas niveau » (**descripteur, int**) ; « haut-niveau » (**stream, FILE ***), à privilégier
- ▶ FILE * est une structure complexe, qui enregistre le descripteur, la position courante, les droits d'accès, la position physique et symbolique du fichier, le tampon... (dépend de l'OS)
- ▶ les fonctions de stdio.h interagissent avec le système (ouverture, fermeture, création, tubes (*pipes*)...)
- ▶ mode d'accès privilégié : lecture/écriture séquentielle
- ▶ pas de contrôle des accès concurrents
- ▶ possibilité d'avoir plusieurs stream pour un même fichier
- ▶ les opérations passent par un *tampon*

Accès aux fichiers : concepts de base

- ▶ deux types pour les fichiers : « bas niveau » (descripteur, `int`) ; « haut-niveau » (`stream`, `FILE *`), à privilégier
- ▶ `FILE *` est une structure complexe, qui enregistre le descripteur, la position courante, les droits d'accès, la position physique et symbolique du fichier, le tampon... (dépend de l'OS)
- ▶ les fonctions de `stdio.h` interagissent avec le système (ouverture, fermeture, création, tubes (*pipes*)...)
- ▶ mode d'accès privilégié : lecture/écriture séquentielle
- ▶ pas de contrôle des accès concurrents
- ▶ possibilité d'avoir plusieurs `stream` pour un même fichier
- ▶ les opérations passent par un *tampon*

Accès aux fichiers : concepts de base

- ▶ deux types pour les fichiers : « bas niveau » (descripteur, `int`) ; « haut-niveau » (`stream`, `FILE *`), à privilégier
- ▶ `FILE *` est une structure complexe, qui enregistre le descripteur, la position courante, les droits d'accès, la position physique et symbolique du fichier, le tampon... (dépend de l'OS)
- ▶ les fonctions de `stdio.h` interagissent avec le système (ouverture, fermeture, création, tubes (*pipes*)...)
- ▶ mode d'accès privilégié : lecture/écriture séquentielle
- ▶ pas de contrôle des accès concurrents
- ▶ possibilité d'avoir plusieurs `stream` pour un même fichier
- ▶ les opérations passent par un *tampon*

Accès aux fichiers : concepts de base

- ▶ deux types pour les fichiers : « bas niveau » (descripteur, `int`) ; « haut-niveau » (`stream`, `FILE *`), à privilégier
- ▶ `FILE *` est une structure complexe, qui enregistre le descripteur, la position courante, les droits d'accès, la position physique et symbolique du fichier, le tampon... (dépend de l'OS)
- ▶ les fonctions de `stdio.h` interagissent avec le système (ouverture, fermeture, création, tubes (*pipes*)...)
- ▶ mode d'accès privilégié : lecture/écriture séquentielle
- ▶ pas de contrôle des accès concurrents
- ▶ possibilité d'avoir plusieurs `stream` pour un même fichier
- ▶ les opérations passent par un *tampon*

Accès aux fichiers : concepts de base

- ▶ deux types pour les fichiers : « bas niveau » (descripteur, `int`) ; « haut-niveau » (`stream`, `FILE *`), à privilégier
- ▶ `FILE *` est une structure complexe, qui enregistre le descripteur, la position courante, les droits d'accès, la position physique et symbolique du fichier, le tampon... (dépend de l'OS)
- ▶ les fonctions de `stdio.h` interagissent avec le système (ouverture, fermeture, création, tubes (*pipes*)...)
- ▶ mode d'accès privilégié : lecture/écriture séquentielle
- ▶ pas de contrôle des accès concurrents
- ▶ possibilité d'avoir plusieurs `stream` pour un même fichier
- ▶ les opérations passent par un *tampon*

Accès aux fichiers : concepts de base

- ▶ deux types pour les fichiers : « bas niveau » (descripteur, `int`) ; « haut-niveau » (`stream`, `FILE *`), à privilégier
- ▶ `FILE *` est une structure complexe, qui enregistre le descripteur, la position courante, les droits d'accès, la position physique et symbolique du fichier, le tampon... (dépend de l'OS)
- ▶ les fonctions de `stdio.h` interagissent avec le système (ouverture, fermeture, création, tubes (*pipes*)...)
- ▶ mode d'accès privilégié : lecture/écriture séquentielle
- ▶ pas de contrôle des accès concurrents
- ▶ possibilité d'avoir plusieurs `stream` pour un même fichier
- ▶ les opérations passent par un *tampon*

Accès aux fichiers : concepts de base

- ▶ deux types pour les fichiers : « bas niveau » (descripteur, `int`) ; « haut-niveau » (`stream`, `FILE *`), à privilégier
- ▶ `FILE *` est une structure complexe, qui enregistre le descripteur, la position courante, les droits d'accès, la position physique et symbolique du fichier, le tampon... (dépend de l'OS)
- ▶ les fonctions de `stdio.h` interagissent avec le système (ouverture, fermeture, création, tubes (*pipes*)...)
- ▶ mode d'accès privilégié : lecture/écriture séquentielle
- ▶ pas de contrôle des accès concurrents
- ▶ possibilité d'avoir plusieurs `stream` pour un même fichier
- ▶ les opérations passent par un *tampon*

Manipulations élémentaires des fichiers

► ouvrir, fermer un flot (*stream*)

- ▶ `FILE * fopen(const char * path, const char * mode);`. Modes possibles : lecture (mode = "r"), écriture (mode="w") ...
- ▶ `fclose(FILE *fp)` : ferme le flux

► lecture, écriture caractère par caractère dans fp

`int fgetc(FILE *fp), int fputc(char c, FILE * fp)`

► lecture, écriture formatée dans fp

`fscanf(FILE *fp, ..), fprintf (FILE *fp, ...)` (idem `sscanf, sprintf`)

► lecture, écriture ligne par ligne fp

`int fgets(FILE *fp, ..), int fputs(FILE *fp, ..)`

► lecture, écriture binaire dans fp

`fread(fp, ..), fwrite (fp, ...)`

► pour contourner la séquentialité : déplacement, repositionnement

`fseek(fp, ..), ftell (fp, ...)`

Manipulations élémentaires des fichiers

- ▶ ouvrir, fermer un flot (*stream*)
 - ▶ `FILE * fopen(const char * path, const char * mode);`. Modes possibles : lecture (mode = "r"), écriture (mode="w") ...
 - ▶ `fclose(FILE *fp)` : ferme le flux
- ▶ lecture, écriture caractère par caractère dans fp
 - `int fgetc(FILE *fp)`, `int fputc(char c, FILE * fp)`
- ▶ lecture, écriture formatée dans fp
 - `fscanf(FILE *fp, ...)`, `fprintf (FILE *fp, ...)` (idem `sscanf`, `sprintf`)
- ▶ lecture, écriture ligne par ligne fp
 - `int fgets(FILE *fp, ...)`, `int fputs(FILE *fp, ...)`
- ▶ lecture, écriture binaire dans fp
 - `fread(fp, ...)`, `fwrite (fp, ...)`
- ▶ pour contourner la séquentialité : déplacement, repositionnement
 - `fseek(fp, ...)`, `ftell (fp, ...)`

Manipulations élémentaires des fichiers

- ▶ ouvrir, fermer un flot (*stream*)
 - ▶ `FILE * fopen(const char * path, const char * mode);`. Modes possibles : lecture (mode = "r"), écriture (mode="w") ...
 - ▶ `fclose(FILE *fp)` : ferme le flux
- ▶ lecture, écriture caractère par caractère dans fp
`int fgetc(FILE *fp), int fputc(char c, FILE * fp)`
- ▶ lecture, écriture formatée dans fp
`fscanf(FILE *fp, ...), fprintf (FILE *fp, ...)` (idem `sscanf, sprintf`)
- ▶ lecture, écriture ligne par ligne fp
`int fgets(FILE *fp, ...), int fputs(FILE *fp, ...)`
- ▶ lecture, écriture binaire dans fp
`fread(fp, ...), fwrite (fp, ...)`
- ▶ pour contourner la séquentialité : déplacement, repositionnement
`fseek(fp, ...), ftell (fp, ...)`

Manipulations élémentaires des fichiers

- ▶ ouvrir, fermer un flot (*stream*)
 - ▶ `FILE * fopen(const char * path, const char * mode);`. Modes possibles : lecture (mode = "r"), écriture (mode="w") ...
 - ▶ `fclose(FILE *fp)` : ferme le flux
- ▶ lecture, écriture caractère par caractère dans fp
 - `int fgetc(FILE *fp)`, `int fputc(char c, FILE * fp)`
- ▶ lecture, écriture formatée dans fp
 - `fscanf(FILE *fp, ...)`, `fprintf (FILE *fp, ...)` (idem `sscanf`, `sprintf`)
- ▶ lecture, écriture ligne par ligne fp
 - `int fgets(FILE *fp, ...)`, `int fputs(FILE *fp, ...)`
- ▶ lecture, écriture binaire dans fp
 - `fread(fp, ...)`, `fwrite (fp, ...)`
- ▶ pour contourner la séquentialité : déplacement, repositionnement
 - `fseek(fp, ...)`, `ftell (fp, ...)`

Manipulations élémentaires des fichiers

- ▶ ouvrir, fermer un flot (*stream*)
 - ▶ `FILE * fopen(const char * path, const char * mode);`. Modes possibles : lecture (mode = "r"), écriture (mode="w") ...
 - ▶ `fclose(FILE *fp)` : ferme le flux
- ▶ lecture, écriture caractère par caractère dans fp
`int fgetc(FILE *fp)`, `int fputc(char c, FILE * fp)`
- ▶ lecture, écriture formatée dans fp
`fscanf(FILE *fp, ...)`, `fprintf (FILE *fp, ...)` (*idem* `sscanf`, `sprintf`)
- ▶ lecture, écriture ligne par ligne fp
`int fgets(FILE *fp, ...)`, `int fputs(FILE *fp, ...)`
- ▶ lecture, écriture binaire dans fp
`fread(fp, ...)`, `fwrite (fp, ...)`
- ▶ pour contourner la séquentialité : déplacement, repositionnement
`fseek(fp, ...)`, `ftell (fp, ...)`

Manipulations élémentaires des fichiers

- ▶ ouvrir, fermer un flot (*stream*)
 - ▶ `FILE * fopen(const char * path, const char * mode);`. Modes possibles : lecture (mode = "r"), écriture (mode="w") ...
 - ▶ `fclose(FILE *fp)` : ferme le flux
- ▶ lecture, écriture caractère par caractère dans fp
`int fgetc(FILE *fp)`, `int fputc(char c, FILE * fp)`
- ▶ lecture, écriture formatée dans fp
`fscanf(FILE *fp, ...)`, `fprintf (FILE *fp, ...)` (idem `sscanf`, `sprintf`)
- ▶ lecture, écriture ligne par ligne fp
`int fgets(FILE *fp, ...)`, `int fputs(FILE *fp, ...)`
- ▶ lecture, écriture binaire dans fp
`fread(fp, ...)`, `fwrite (fp, ...)`
- ▶ pour contourner la séquentialité : déplacement, repositionnement
`fseek(fp, ...)`, `ftell (fp, ...)`

Manipulations élémentaires des fichiers

- ▶ ouvrir, fermer un flot (*stream*)
 - ▶ `FILE * fopen(const char * path, const char * mode);`. Modes possibles : lecture (mode = "r"), écriture (mode="w") ...
 - ▶ `fclose(FILE *fp)` : ferme le flux
- ▶ lecture, écriture caractère par caractère dans fp
 - `int fgetc(FILE *fp)`, `int fputc(char c, FILE * fp)`
- ▶ lecture, écriture formatée dans fp
 - `fscanf(FILE *fp, ...)`, `fprintf (FILE *fp, ...)` (idem `sscanf`, `sprintf`)
- ▶ lecture, écriture ligne par ligne fp
 - `int fgets(FILE *fp, ...)`, `int fputs(FILE *fp, ...)`
- ▶ lecture, écriture binaire dans fp
 - `fread(fp, ...)`, `fwrite (fp, ...)`
- ▶ pour contourner la séquentialité : déplacement, repositionnement
 - `fseek(fp, ...)`, `ftell (fp, ...)`

Manipulations élémentaires des fichiers

- ▶ ouvrir, fermer un flot (*stream*)
 - ▶ `FILE * fopen(const char * path, const char * mode);`. Modes possibles : lecture (mode = "r"), écriture (mode="w") ...
 - ▶ `fclose(FILE *fp)` : ferme le flux
- ▶ lecture, écriture caractère par caractère dans fp
`int fgetc(FILE *fp)`, `int fputc(char c, FILE * fp)`
- ▶ lecture, écriture formatée dans fp
`fscanf(FILE *fp, ...)`, `fprintf (FILE *fp, ...)` (idem `sscanf`, `sprintf`)
- ▶ lecture, écriture ligne par ligne fp
`int fgets(FILE *fp, ...)`, `int fputs(FILE *fp, ...)`
- ▶ lecture, écriture binaire dans fp
`fread(fp, ...)`, `fwrite (fp, ...)`
- ▶ pour contourner la séquentialité : déplacement, repositionnement
`fseek(fp, ...)`, `ftell (fp, ...)`

Dump d'un tableau de structures

```
#define N_ELEVE (10)
struct eleve {
    char * nom;
    int age;
    float moyenne
};
int neleve = N_ELEVE;
struct eleve[neleve] classe;
FILE * fp = fopen("classe.dmp", "w");
if (fp == NULL) { /* error */ }
fwrite(&neleve, sizeof(int), 1, fp);
for (int i=0; i<neleve; i++) {
    int l = strlen(eleve[i].nom);
    fwrite(&l, sizeof(int), 1, fp);
    fwrite(eleve[i].nom, sizeof(char), l+1, fp);
    fwrite(&(eleve[i].age), sizeof(int), 1, fp);
    fwrite(eleve[i].moyenne, sizeof(float), 1, fp);
}
```

Construction d'un index

```
#define MAX_LINE (256)
#define N_LINE (500)

long start[N_LINE];
char line[MAX_LINE];
int nline = 0;
start[0] = 0;
while(fgets(line, MAX_LINE, fp) != NULL) {
    start[++nline] = ftell(fp);
    if (nline >= N_LINE) /* error */
}
/* debut du fichier */
void rewind(fp);
/* accede a la 10e ligne */
fseek(fp, start[10], SEEK_SET);
```

Fichiers et tampons

- ▶ FILE * est un type complexe qui interface avec les opérations du système. Contient au moins un descripteur de fichiers, un tampon (buffer) et des données de gestion
- ▶ les lectures et écritures *physique* sont sous le contrôle du système ; toute E/S du programmeur se fait dans un tampon
- ▶ des FILE * offerts : stdin, stdout, stderr
- ▶ vidange du tampon : fflush ()
- ▶ choisir son tampon : setvbuf() et la politique de vidange associée

Interagir avec le shell

- ▶ terminer l'exécution du programme :
`exit(int), abort(), atexit (void (*f)())`
- ▶ passer sa commande au shell : `system(const char *)`, voire
`popen(FILE *, char *)`
- ▶ lire/écrire l'environnement :
`getenv(char *), putenv(char *), (un)setenv (...)`

Interagir avec le shell

- ▶ terminer l'exécution du programme :
exit (int), abort (), atexit (void (*f)())
- ▶ passer sa commande au shell : system(const char *), voire
popen(FILE *, char *)
- ▶ lire/écrire l'environnement :
getenv(char *), putenv(char *), (un)setenv (...)

Interagir avec le shell

- ▶ terminer l'exécution du programme :
exit (int), abort(), atexit (void (*f)())
- ▶ passer sa commande au shell : system(const char *), voire
popen(FILE *, char *)
- ▶ lire/écrire l'environnement :
getenv(char *), putenv(char *), (un)setenv (...)

Une utilisation des tubes

▶ play tube

Exploration d'un répertoire

▶ play dirent

Manipulation des chaînes [string.h]

- ▶ copie : strcpy, strncpy, memcpy, memmove
- ▶ caténation : strcat, strncat
- ▶ comparaison : strcmp, strncmp, memcmp, strcoll
- ▶ recherche d'un char : strchr, memchr, strrchr, index !
- ▶ recherche de plusieurs char : strspn, strcspn, strpbrk
- ▶ recherche de sous-chaînes : strstr
- ▶ segmentation de l'entrée : strtok
- ▶ à savoir : recherche de *motifs* (dans la *glibc* :
regcomp(), regmatch(), regfree()))

Découper l'entrée avec strtok

▶ play strtok

Explication des erreurs

```
if ((fp = fopen("toto", "w")) == NULL) {
    perror("*** Erreur *** fopen (read)");
    /* Variante */
    fprintf("Echec lecture de %s: %s\n",
           (char *)"toto", strerror(errno));
}
```

- ▶ errno (dans errno.h) : numéro de la dernière erreur
- ▶ perror (dans stdio.h) : impression simple du message d'erreur
- ▶ strerror (dans string.h) : récupération du message d'erreur
- ▶ ?? retour des erreurs à l'appelant

Explication des erreurs

```
if ((fp = fopen("toto", "w")) == NULL) {
    perror("*** Erreur *** fopen (read)");
    /* Variante */
    fprintf("Echec lecture de %s: %s\n",
           (char *)"toto", strerror(errno));
}
```

- ▶ **errno (dans errno.h) : numéro de la dernière erreur**
- ▶ **perror (dans stdio.h) : impression simple du message d'erreur**
- ▶ **strerror (dans string.h) : récupération du message d'erreur**
- ▶ ?? retour des erreurs à l'appelant

Explication des erreurs

```
if ((fp = fopen("toto", "w")) == NULL) {  
    perror("*** Erreur *** fopen (read)");  
    /* Variante */  
    fprintf("Echec lecture de %s: %s\n",  
           (char *)"toto", strerror(errno));  
}
```

- ▶ **errno** (dans `errno.h`) : numéro de la dernière erreur
- ▶ **perror** (dans `stdio.h`) : impression simple du message d'erreur
- ▶ **strerror** (dans `string.h`) : récupération du message d'erreur
- ▶ ?? retour des erreurs à l'appelant

Explication des erreurs

```
if ((fp = fopen("toto", "w")) == NULL) {  
    perror("*** Erreur *** fopen (read)");  
    /* Variante */  
    fprintf("Echec lecture de %s: %s\n",  
           (char *)"toto", strerror(errno));  
}
```

- ▶ errno (dans errno.h) : numéro de la dernière erreur
- ▶ perror (dans stdio.h) : impression simple du message d'erreur
- ▶ strerror (dans string.h) : récupération du message d'erreur
- ▶ ?? retour des erreurs à l'appelant

Explication des erreurs

```
if ((fp = fopen("toto", "w")) == NULL) {  
    perror("*** Erreur *** fopen (read)");  
    /* Variante */  
    fprintf("Echec lecture de %s: %s\n",  
           (char *)"toto", strerror(errno));  
}
```

- ▶ errno (dans errno.h) : numéro de la dernière erreur
- ▶ perror (dans stdio.h) : impression simple du message d'erreur
- ▶ strerror (dans string.h) : récupération du message d'erreur
- ▶ ?? retour des erreurs à l'appelant

Récréation

Soit un fichier de données structuré en une suite de lignes contenant chacune un nom de personne, un nom de pièce, un nombre et un prix. Exemple :

```
dupond vilebrequin 10 1000
```

Écrire une procédure main dans laquelle on déclarera les variables suivantes :

- ▶ nom et article : tableaux de 80 caractères
- ▶ nombre et prix : entiers

Le corps de la procédure consistera en une boucle dont chaque itération lira une ligne et l'imprimera :

- ▶ la lecture d'une ligne se fera par un appel à scanf affectant les 4 champs de la ligne aux 4 variables nom, article, nombre et prix.
- ▶ l'écriture consistera à imprimer nom, article et le produit $\text{nombre} \times \text{prix}$.

Récréations

- ▶ Ecrire un programme qui lit un verbe régulier en "er" au clavier et qui en affiche la conjugaison au présent de l'indicatif de ce verbe. Contrôlez s'il s'agit bien d'un verbe en "er" avant de conjuguer. Utiliser les fonctions gets, puts, strcat et strlen
- ▶ Écrire un programme qui supprime les commentaires de vos programmes (`/* */`).
- ▶ Écrire deux programmes cesar[de]code –n qui [dé]code (avec un code cyclique : $c \rightarrow \dots \rightarrow c + n$) les lignes en entrée et les affiche sur stdin En enchaînant les deux par un pipe, on doit retomber sur ses pieds.

Récréations

- ▶ Ecrire un programme qui lit un verbe régulier en "er" au clavier et qui en affiche la conjugaison au présent de l'indicatif de ce verbe. Contrôlez s'il s'agit bien d'un verbe en "er" avant de conjuguer. Utiliser les fonctions gets, puts, strcat et strlen
- ▶ Écrire un programme qui supprime les commentaires de vos programmes /* */.
- ▶ Écrire deux programmes cesar[de]code –n qui [dé]code (avec un code cyclique : $c \rightarrow - > c + n$) les lignes en entrée et les affiche sur stdin En enchaînant les deux par un pipe, on doit retomber sur ses pieds.

Récréations

- ▶ Ecrire un programme qui lit un verbe régulier en "er" au clavier et qui en affiche la conjugaison au présent de l'indicatif de ce verbe. Contrôlez s'il s'agit bien d'un verbe en "er" avant de conjuguer. Utiliser les fonctions gets, puts, strcat et strlen
- ▶ Écrire un programme qui supprime les commentaires de vos programmes /* */.
- ▶ Écrire deux programmes cesar[de]code –n qui [dé]code (avec un code cyclique : $c \rightarrow - > c + n$) les lignes en entrée et les affiche sur stdin En enchaînant les deux par un pipe, on doit retomber sur ses pieds.

L'emploi du time

```
#include <time.h>
clock_t start = clock();
time_t temps;
temps = time();
char * ctime(&temps);
clock_t end = clock();
printf("temps total %d\n",
      (double)(end - start)/CLOCK_PER_SEC);
```

- ▶ les objets `clock_t` : permettent de mesurer le nombre de « ticks » de l'horloge interne de la machine
- ▶ `time_t time(time_t *p_temps);` : renvoie un objet de type `time_t`
- ▶ `char *ctime(time_t *p_temps);` : produit une chaîne de caractère (en anglais !)
- ▶ `gmtime, localtime` qui manipulent des `struct tm` (un champ pour les secondes, les minutes, les heures, les jours...)

L'emploi du time

```
#include <time.h>
clock_t start = clock();
time_t temps;
temps = time();
char * ctime(&temps);
clock_t end = clock();
printf("temps total %d\n",
      (double)(end - start)/CLOCK_PER_SEC);
```

- ▶ les objets `clock_t` : permettent de mesurer le nombre de « ticks » de l'horloge interne de la machine
- ▶ `time_t time(time_t *p_temps);` : renvoie un objet de type `time_t`
- ▶ `char *ctime(time_t *p_temps);` : produit une chaîne de caractère (en anglais !)
- ▶ `gmtime, localtime` qui manipulent des `struct tm` (un champ pour les secondes, les minutes, les heures, les jours...)

L'emploi du time

```
#include <time.h>
clock_t start = clock();
time_t temps;
temps = time();
char * ctime(&temps);
clock_t end = clock();
printf("temps total %d\n",
      (double)(end - start)/CLOCK_PER_SEC);
```

- ▶ les objets `clock_t` : permettent de mesurer le nombre de « ticks » de l'horloge interne de la machine
- ▶ `time_t time(time_t *p_temps)` : renvoie un objet de type `time_t`
- ▶ **char** *`ctime(time_t *p_temps)` : produit une chaîne de caractère (en anglais !)
- ▶ `gmtime`, `localtime` qui manipulent des `struct tm` (un champ pour les secondes, les minutes, les heures, les jours...)

L'emploi du time

```
#include <time.h>
clock_t start = clock();
time_t temps;
temps = time();
char * ctime(&temps);
clock_t end = clock();
printf("temps total %d\n",
      (double)(end - start)/CLOCK_PER_SEC);
```

- ▶ les objets `clock_t` : permettent de mesurer le nombre de « ticks » de l'horloge interne de la machine
- ▶ `time_t time(time_t *p_temps)` : renvoie un objet de type `time_t`
- ▶ `char *ctime(time_t *p_temps)` : produit une chaîne de caractère (en anglais !)
- ▶ `gmtime, localtime` qui manipulent des `struct tm` (un champ pour les secondes, les minutes, les heures, les jours...)

Caractères : compléments

- ▶ conversion char * → float, double, int ... :
atof, atoi, strtod, strtol ... ou (meilleur) strtod (), strtodf (), strtol ...
- ▶ classes de caractères dans ctype.h :
isalpha, isnum, isspace, isdigit, isalphanum...
- ▶ comportement standard pour $c < 128$: au delà ??? localiser (y compris le point décimal)
- ▶ pour des jeux de caractères multiples : les wchar (wide char)
 - ▶ wctype.h : classes de caractères
 - ▶ wchar.h : manipulation de chaînes
 - ▶ ⇒ imprimer des wchar : %ls

Caractères : compléments

- ▶ conversion char * → float, double, int ... :
atof , atoi , strtod , strtol ... ou (meilleur) strtod (), strtodf (), strtol ...
- ▶ classes de caractères dans ctype.h :
isalpha, isnum, isspace, isdigit , isalphanum...
- ▶ comportement standard pour $c < 128$: au delà ??? localiser (y compris le point décimal)
- ▶ pour des jeux de caractères multiples : les wchar (wide char)
 - ▶ wctype.h : classes de caractères
 - ▶ wchar.h : manipulation de chaînes
 - ▶ ⇒ imprimer des wchar : %ls

Caractères : compléments

- ▶ conversion char * → float, double, int ... :
- atof , atoi , strtod , strtol ... ou (meilleur) strtod (), strtodf (), strtol ...
- ▶ classes de caractères dans ctype.h :
- isalpha, isnum, isspace, isdigit , isalphanum...
- ▶ comportement standard pour $c < 128$: au delà ??? localiser (y compris le point décimal)
- ▶ pour des jeux de caractères multiples : les wchar (wide char)
 - ▶ wctype.h : classes de caractères
 - ▶ wchar.h : manipulation de chaînes
 - ▶ ⇒ imprimer des wchar : %ls

Caractères : compléments

- ▶ conversion char * → float, double, int ... :
atof , atoi , strtod , strtol ... ou (meilleur) strtod (), strtodf (), strtol ...
- ▶ classes de caractères dans ctype.h :
isalpha, isnum, isspace, isdigit , isalphanum...
- ▶ comportement standard pour $c < 128$: au delà ??? *localiser* (y compris le point décimal)
- ▶ pour des jeux de caractères multiples : les wchar (wide char)
 - ▶ wctype.h : classes de caractères
 - ▶ wchar.h : manipulation de chaînes
 - ▶ ⇒ imprimer des wchar : %ls

Caractères : compléments

- ▶ conversion char * → float, double, int ... :
atof , atoi , strtod , strtol ... ou (meilleur) strtod (), strtodf (), strtol ...
- ▶ classes de caractères dans ctype.h :
isalpha, isnum, isspace, isdigit , isalphanum...
- ▶ comportement standard pour $c < 128$: au delà ??? *localiser* (y compris le point décimal)
- ▶ pour des jeux de caractères multiples : les wchar (wide char)
 - ▶ wctype.h : classes de caractères
 - ▶ wchar.h : manipulation de chaînes
 - ▶ ⇒ imprimer des wchar : %ls

Caractères : compléments

- ▶ conversion char * → float, double, int ... :
atof , atoi , strtod , strtol ... ou (meilleur) strtod (), strtodf (), strtol ...
- ▶ classes de caractères dans ctype.h :
isalpha, isnum, isspace, isdigit , isalphanum...
- ▶ comportement standard pour $c < 128$: au delà ??? *localiser* (y compris le point décimal)
- ▶ pour des jeux de caractères multiples : les wchar (wide char)
 - ▶ wctype.h : classes de caractères
 - ▶ **wchar.h : manipulation de chaînes**
 - ▶ ⇒ imprimer des wchar : %ls

Caractères : compléments

- ▶ conversion char * → float, double, int ... :
atof , atoi , strtod , strtol ... ou (meilleur) strtod (), strtodf (), strtol ...
- ▶ classes de caractères dans ctype.h :
isalpha, isnum, isspace, isdigit , isalphanum...
- ▶ comportement standard pour $c < 128$: au delà ??? *localiser* (y compris le point décimal)
- ▶ pour des jeux de caractères multiples : les wchar (wide char)
 - ▶ wctype.h : classes de caractères
 - ▶ wchar.h : manipulation de chaînes
 - ▶ ⇒ imprimer des wchar : %ls

Localisation : principes

- ▶ consultation de l'environnement ou mise à jour explicite de variables « pays » via setlocale () dans locale.h
- ▶ influence :
 - ▶ les classes de caractères is_alpha (), toupper (...) ... [LC_TYPE] et leur ordre [LC_COLLATE] (utiliser strcoll ())
 - ▶ l'affichage des réels (séparateur de milliers, point décimal) [LC_NUMERIC]
 - ▶ l'affichage des durées [LC_TIME]
 - ▶ l'affichage des montants via lconv [LC_MONETARY]
 - ▶ l'affichage des messages systèmes [LC_MESSAGES]... et des vôtres (création de catalogues)

Localisation : des logiciels polyglottes !

▶ play locale

La bibliothèque mathématique

► fonctions usuelles (`sqrt`, `cbrt`,`hypot`)

- partie entière, arrondi and such : (`ceil` , `floor` , `mod`, `frexp` ...)
- fonctions trigonométriques (`cos`, `acos`, `sin`, `asin`, `tan`, `atan`, ...)
- fonctions exponentielle et logarithme
(`exp`, `exp2`, `pow`, `log`, `log2`, `log10` ...)
- constantes usuelles (`INFINITY`, `M_PI`, `M_E`, `MSQRT_2...`)
- travailler en double (`cosf` pour les `float`, `cosl` pour les `long doubles`)
- pour aller plus loin : bibliothèque scientifique *gsl* (calcul vectoriel et matriciel, probabilités, statistiques, optimisation...)

La bibliothèque mathématique

- ▶ fonctions usuelles (`sqrt`, `cbrt`,`hypot`)
- ▶ partie entière, arrondi and such : (`ceil` , `floor` , `mod`, `frexp` ...)
- ▶ fonctions trigonométriques (`cos`, `acos`, `sin`, `asin`, `tan`, `atan`, ...)
- ▶ fonctions exponentielle et logarithme
(`exp`, `exp2`, `pow`, `log`, `log2`, `log10` ...)
- ▶ constantes usuelles (`INFINITY`, `M_PI`, `M_E`, `MSQRT_2...`)
- ▶ travailler en double (`cosf` pour les `float`, `cosl` pour les `long doubles`)
- ▶ pour aller plus loin : bibliothèque scientifique *gsl* (calcul vectoriel et matriciel, probabilités, statistiques, optimisation...)

La bibliothèque mathématique

- ▶ fonctions usuelles (`sqrt`, `cbrt`,`hypot`)
- ▶ partie entière, arrondi and such : (`ceil` , `floor` , `mod`, `frexp` ...)
- ▶ fonctions trigonométriques (`cos`, `acos`, `sin`, `asin`, `tan`, `atan`, ...)
- ▶ fonctions exponentielle et logarithme
(`exp`, `exp2`, `pow`, `log`, `log2`, `log10` ...)
- ▶ constantes usuelles (`INFINITY`, `M_PI`, `M_E`, `MSQRT_2`...)
- ▶ travailler en double (`cosf` pour les `float`, `cosl` pour les `long doubles`)
- ▶ pour aller plus loin : bibliothèque scientifique *gsl* (calcul vectoriel et matriciel, probabilités, statistiques, optimisation...)

La bibliothèque mathématique

- ▶ fonctions usuelles (`sqrt`, `cbrt`,`hypot`)
- ▶ partie entière, arrondi and such : (`ceil` , `floor` , `mod`, `frexp` ...)
- ▶ fonctions trigonométriques (`cos`, `acos`, `sin`, `asin`, `tan`, `atan`, ...)
- ▶ **fonctions exponentielle et logarithme**
(`exp`, `exp2`, `pow`, `log`, `log2`, `log10` ...)
- ▶ constantes usuelles (`INFINITY`, `M_PI`, `M_E`, `MSQRT_2...`)
- ▶ travailler en double (`cosf` pour les `float`, `cosl` pour les `long doubles`)
- ▶ pour aller plus loin : bibliothèque scientifique *gsl* (calcul vectoriel et matriciel, probabilités, statistiques, optimisation...)

La bibliothèque mathématique

- ▶ fonctions usuelles (`sqrt`, `cbrt`,`hypot`)
- ▶ partie entière, arrondi and such : (`ceil` , `floor` , `mod`, `frexp` ...)
- ▶ fonctions trigonométriques (`cos`, `acos`, `sin`, `asin`, `tan`, `atan`, ...)
- ▶ fonctions exponentielle et logarithme
(`exp`, `exp2`, `pow`, `log`, `log2`, `log10` ...)
- ▶ **constantes usuelles (INFINITY, M_PI, M_E, MSQRT_2...)**
- ▶ travailler en double (`cosf` pour les float, `cosl` pour les long doubles)
- ▶ pour aller plus loin : bibliothèque scientifique *gsl* (calcul vectoriel et matriciel, probabilités, statistiques, optimisation...)

La bibliothèque mathématique

- ▶ fonctions usuelles (`sqrt`, `cbrt`,`hypot`)
- ▶ partie entière, arrondi and such : (`ceil` , `floor` , `mod`, `frexp` ...)
- ▶ fonctions trigonométriques (`cos`, `acos`, `sin`, `asin`, `tan`, `atan`, ...)
- ▶ fonctions exponentielle et logarithme
(`exp`, `exp2`, `pow`, `log`, `log2`, `log10` ...)
- ▶ constantes usuelles (`INFINITY`, `M_PI`, `M_E`, `MSQRT_2...`)
- ▶ travailler en double (`cosf` pour les float, `cosl` pour les long doubles)
- ▶ pour aller plus loin : bibliothèque scientifique `gsl` (calcul vectoriel et matriciel, probabilités, statistiques, optimisation...)

La bibliothèque mathématique

- ▶ fonctions usuelles (sqrt, cbrt, hypot)
- ▶ partie entière, arrondi and such : (ceil, floor, mod, frexp ...)
- ▶ fonctions trigonométriques (cos, acos, sin, asin, tan, atan, ...)
- ▶ fonctions exponentielle et logarithme
(exp, exp2, pow, log, log2, log10 ...)
- ▶ constantes usuelles (INFINITY, M_PI, M_E, MSQRT_2...)
- ▶ travailler en double (cosf pour les float, cosl pour les long doubles)
- ▶ pour aller plus loin : bibliothèque scientifique *gsl* (calcul vectoriel et matriciel, probabilités, statistiques, optimisation...)

Le(s) thème(s) du jour: S'outiller pour développer

Organiser le code, séparer les problèmes

- ▶ séparer le code en plusieurs fichiers le plus autonome possible
/Rightarrow structuration du code, simplification du développement
- ▶ séparation des espaces de visibilité des variables, distinction public/privé
- ▶ chaque fichier compilé séparément produit un fichier objet (gcc -c)
- ▶ l'assemblage est réalisé par l'éditeur de lien
- ▶ plusieurs objets rassemblés dans une bibliothèque ar, ranlib
- ▶ inspection des objets : objdump, nm, otool...

Organiser le code, séparer les problèmes

- ▶ séparer le code en plusieurs fichiers le plus autonome possible
/Rightarrow structuration du code, simplification du développement
- ▶ séparation des espaces de visibilité des variables, distinction public/privé
- ▶ chaque fichier compilé séparément produit un fichier objet (gcc -c)
- ▶ l'assemblage est réalisé par l'éditeur de lien
- ▶ plusieurs objets rassemblés dans une bibliothèque ar, ranlib
- ▶ inspection des objets : objdump, nm, otool...

Organiser le code, séparer les problèmes

- ▶ séparer le code en plusieurs fichiers le plus autonome possible
/Rightarrow structuration du code, simplification du développement
- ▶ séparation des espaces de visibilité des variables, distinction public/privé
- ▶ chaque fichier compilé séparément produit un fichier objet (`gcc -c`)
- ▶ l'assemblage est réalisé par l'éditeur de lien
- ▶ plusieurs objets rassemblés dans une bibliothèque `ar`, `ranlib`
- ▶ inspection des objets : `objdump`, `nm`, `otool...`

Organiser le code, séparer les problèmes

- ▶ séparer le code en plusieurs fichiers le plus autonome possible
/Rightarrow structuration du code, simplification du développement
- ▶ séparation des espaces de visibilité des variables, distinction public/privé
- ▶ chaque fichier compilé séparément produit un fichier objet (`gcc -c`)
- ▶ l'assemblage est réalisé par l'éditeur de lien
- ▶ plusieurs objets rassemblés dans une bibliothèque `ar`, `ranlib`
- ▶ inspection des objets : `objdump`, `nm`, `otool...`

Organiser le code, séparer les problèmes

- ▶ séparer le code en plusieurs fichiers le plus autonome possible
/Rightarrow structuration du code, simplification du développement
- ▶ séparation des espaces de visibilité des variables, distinction public/privé
- ▶ chaque fichier compilé séparément produit un fichier objet (`gcc -c`)
- ▶ l'assemblage est réalisé par l'éditeur de lien
- ▶ plusieurs objets rassemblés dans une bibliothèque `ar`, `ranlib`
- ▶ inspection des objets : `objdump`, `nm`, `otool...`

Organiser le code, séparer les problèmes

- ▶ séparer le code en plusieurs fichiers le plus autonome possible
/Rightarrow structuration du code, simplification du développement
- ▶ séparation des espaces de visibilité des variables, distinction public/privé
- ▶ chaque fichier compilé séparément produit un fichier objet (`gcc -c`)
- ▶ l'assemblage est réalisé par l'éditeur de lien
- ▶ plusieurs objets rassemblés dans une bibliothèque `ar`, `ranlib`
- ▶ inspection des objets : `objdump`, `nm`, `otool...`

Débogguer

- ▶ Un programme qui compile sans erreur n'est pas nécessairement correct
- ▶ Avant toute chose : vérifier l'utilisation de la mémoire avec par ex. valgrind ou équivalent
- ▶ Si les problèmes persistent : utilisation d'un débogueur (par ex gdb) :
 - ▶ exécution pas à pas
 - ▶ positions de points d'arrêts
 - ▶ accès à toutes les variables
 - ▶ déplacement dans la pile d'appel
 - ▶ ...
- ▶ compilation et édition de lien doivent maintenir des informations auxiliaires : gcc -g
- ▶ incompatible avec l'optimisation du code (-O2)

Débogguer

- ▶ Un programme qui compile sans erreur n'est pas nécessairement correct
- ▶ Avant toute chose : vérifier l'utilisation de la mémoire avec par ex. valgrind ou équivalent
- ▶ Si les problèmes persistent : utilisation d'un débogueur (par ex gdb) :
 - ▶ exécution pas à pas
 - ▶ positions de points d'arrêts
 - ▶ accès à toutes les variables
 - ▶ déplacement dans la pile d'appel
 - ▶ ...
- ▶ compilation et édition de lien doivent maintenir des informations auxiliaires : gcc -g
- ▶ incompatible avec l'optimisation du code (-O2)

Débogguer

- ▶ Un programme qui compile sans erreur n'est pas nécessairement correct
- ▶ Avant toute chose : vérifier l'utilisation de la mémoire avec par ex. valgrind ou équivalent
- ▶ Si les problèmes persistent : utilisation d'un débogueur (par ex gdb) :
 - ▶ exécution pas à pas
 - ▶ positions de points d'arrêts
 - ▶ accès à toutes les variables
 - ▶ déplacement dans la pile d'appel
 - ▶ ...
- ▶ compilation et édition de lien doivent maintenir des informations auxiliaires : gcc -g
- ▶ incompatible avec l'optimisation du code (-O2)

Débogguer

- ▶ Un programme qui compile sans erreur n'est pas nécessairement correct
- ▶ Avant toute chose : vérifier l'utilisation de la mémoire avec par ex. valgrind ou équivalent
- ▶ Si les problèmes persistent : utilisation d'un débogueur (par ex gdb) :
 - ▶ exécution pas à pas
 - ▶ positions de points d'arrêts
 - ▶ accès à toutes les variables
 - ▶ déplacement dans la pile d'appel
 - ▶ ...
- ▶ compilation et édition de lien doivent maintenir des informations auxiliaires : gcc -g
- ▶ incompatible avec l'optimisation du code (-O2)

Débogguer

- ▶ Un programme qui compile sans erreur n'est pas nécessairement correct
- ▶ Avant toute chose : vérifier l'utilisation de la mémoire avec par ex. valgrind ou équivalent
- ▶ Si les problèmes persistent : utilisation d'un débogueur (par ex gdb) :
 - ▶ exécution pas à pas
 - ▶ positions de points d'arrêts
 - ▶ accès à toutes les variables
 - ▶ déplacement dans la pile d'appel
 - ▶ ...
- ▶ compilation et édition de lien doivent maintenir des informations auxiliaires : gcc -g
- ▶ incompatible avec l'optimisation du code (-O2)

Débogguer

- ▶ Un programme qui compile sans erreur n'est pas nécessairement correct
- ▶ Avant toute chose : vérifier l'utilisation de la mémoire avec par ex. valgrind ou équivalent
- ▶ Si les problèmes persistent : utilisation d'un débogueur (par ex gdb) :
 - ▶ exécution pas à pas
 - ▶ positions de points d'arrêts
 - ▶ accès à toutes les variables
 - ▶ déplacement dans la pile d'appel
 - ▶ ...
- ▶ compilation et édition de lien doivent maintenir des informations auxiliaires : gcc -g
- ▶ incompatible avec l'optimisation du code (-O2)

Débogguer

- ▶ Un programme qui compile sans erreur n'est pas nécessairement correct
- ▶ Avant toute chose : vérifier l'utilisation de la mémoire avec par ex. valgrind ou équivalent
- ▶ Si les problèmes persistent : utilisation d'un débogueur (par ex gdb) :
 - ▶ exécution pas à pas
 - ▶ positions de points d'arrêts
 - ▶ accès à toutes les variables
 - ▶ déplacement dans la pile d'appel
 - ▶ ...
- ▶ compilation et édition de lien doivent maintenir des informations auxiliaires : gcc -g
- ▶ incompatible avec l'optimisation du code (-O2)

Débogguer

- ▶ Un programme qui compile sans erreur n'est pas nécessairement correct
- ▶ Avant toute chose : vérifier l'utilisation de la mémoire avec par ex. valgrind ou équivalent
- ▶ Si les problèmes persistent : utilisation d'un débogueur (par ex gdb) :
 - ▶ exécution pas à pas
 - ▶ positions de points d'arrêts
 - ▶ accès à toutes les variables
 - ▶ déplacement dans la pile d'appel
 - ▶ ...
- ▶ compilation et édition de lien doivent maintenir des informations auxiliaires : gcc -g
- ▶ incompatible avec l'optimisation du code (-O2)

Débogguer

- ▶ Un programme qui compile sans erreur n'est pas nécessairement correct
- ▶ Avant toute chose : vérifier l'utilisation de la mémoire avec par ex. valgrind ou équivalent
- ▶ Si les problèmes persistent : utilisation d'un débogueur (par ex gdb) :
 - ▶ exécution pas à pas
 - ▶ positions de points d'arrêts
 - ▶ accès à toutes les variables
 - ▶ déplacement dans la pile d'appel
 - ▶ ...
- ▶ compilation et édition de lien doivent maintenir des informations auxiliaires : gcc -g
- ▶ incompatible avec l'optimisation du code (-O2)

Débogguer

- ▶ Un programme qui compile sans erreur n'est pas nécessairement correct
- ▶ Avant toute chose : vérifier l'utilisation de la mémoire avec par ex. valgrind ou équivalent
- ▶ Si les problèmes persistent : utilisation d'un débogueur (par ex gdb) :
 - ▶ exécution pas à pas
 - ▶ positions de points d'arrêts
 - ▶ accès à toutes les variables
 - ▶ déplacement dans la pile d'appel
 - ▶ ...
- ▶ compilation et édition de lien doivent maintenir des informations auxiliaires : gcc -g
- ▶ incompatible avec l'optimisation du code (-O2)

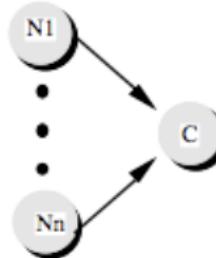
Gestion des dépendances : Make

- ▶ Make est un utilitaire de *gestion des dépendances* dans un ensemble de sources

- Graphe de dépendance :

si le fichier A dépend du fichier B, il y aura un arc de B vers A.

Par exemple, si C dépend de N1, N2, ... Nn :



- ▶ les dépendances sont décrites dans le Makefile
- ▶ Make prend en charge la reconstruction de cibles en fonction des dates de modification

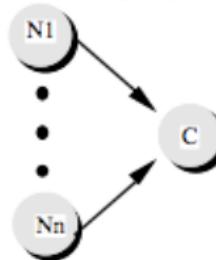
Gestion des dépendances : Make

- ▶ Make est un utilitaire de *gestion des dépendances* dans un ensemble de sources

- Graphe de dépendance :

si le fichier A dépend du fichier B, il y aura un arc de B vers A.

Par exemple, si C dépend de N1, N2, ... Nn :

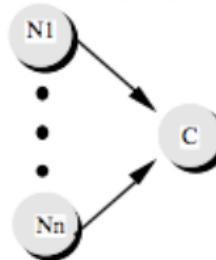


- ▶ les dépendances sont décrites dans le Makefile
- ▶ Make prend en charge la reconstruction de cibles en fonction des dates de modification

Gestion des dépendances : Make

- ▶ Make est un utilitaire de *gestion des dépendances* dans un ensemble de sources
 - Graphe de dépendance :

si le fichier A dépend du fichier B, il y aura un arc de B vers A.
Par exemple, si C dépend de N1, N2, ... Nn :



- ▶ les dépendances sont décrites dans le Makefile
- ▶ Make prend en charge la reconstruction de cibles en fonction des dates de modification

Structure du Makefile

```
# construction de l'objet classe.o
headers=students.h menu.h
classes.o: classes.c ${headers}
<TAB>gcc -c -o classes.o classes.c
```

- ▶ les dépendances cible : liste_des_sources
- ▶ les actions de mise à jour sous la forme de commandes du shell
(attention à la tabulation !!)
- ▶ des commentaires (lignes débutant par #)
- ▶ des définitions de macros (var=value), résolues par \${var}
- ▶ make est récursif : make [-f Makefile] cible :
 - ▶ vérifie si cible est à jour (= plus récent que ces sources)
 - ▶ si non, vérifie que les sources sont à jour, puis reconstruit cible

Structure du Makefile

```
# construction de l'objet classe.o
headers=students.h menu.h
classes.o: classes.c ${headers}
<TAB>gcc -c -o classes.o classes.c
```

- ▶ les dépendances cible : liste_des_sources
- ▶ les actions de mise à jour sous la forme de commandes du shell
(attention à la tabulation !!)
- ▶ des commentaires (lignes débutant par #)
- ▶ des définitions de macros (var=value), résolues par \${var}
- ▶ make est récursif : make [-f Makefile] cible :
 - ▶ vérifie si cible est à jour (= plus récent que ces sources)
 - ▶ si non, vérifie que les sources sont à jour, puis reconstruit cible

Les variables automatiques du Makefile

nom	contient	exemple
Variables globales		
CC	compilateur	CC=gcc
CFLAGS	options de compilation	CFLAGS=-g -Wall
LD	éditeur de liens	CC=gcc
RM	effacement de fichiers	RM=/bin/rm
LDFLAGS	options de l'édition de liens	LDFLAGS=-g -Wall-g -lm
...	+ les variables d'environnement	
Variables locales (à une cible)		
\$@	cible à reconstruire	\$(CC) -\$(CFLAGS) -o \$@
\$<	première dépendance	\$(CC) \$< -o \$@
\$^	liste des dépendances	\$(CC) \$^ -o \$@

pour une liste complète : make -p

Les variables automatiques du Makefile

nom	contient	exemple
Variables globales		
CC	compilateur	CC=gcc
CFLAGS	options de compilation	CFLAGS=-g -Wall
LD	éditeur de liens	CC=gcc
RM	effacement de fichiers	RM=/bin/rm
LDFLAGS	options de l'édition de liens	LDFLAGS=-g -Wall -g -lm
...	+ les variables d'environnement	
Variables locales (à une cible)		
\$@	cible à reconstruire	\$(CC) -\$(CFLAGS) -o \$@
\$<	première dépendance	\$(CC) \$< -o \$@
\$^	liste des dépendances	\$(CC) \$^ -o \$@

pour une liste complète : make -p

Manipuler Make : définition de variables

- ▶ les substitutions ont lieu à l'utilisation

```
OPTIONS=-g $(WARNING)
WARNING=Wall -pedantic
gcc $(OPTIONS)
```

- ▶ := impose les substitutions immédiates ; :+= des concaténations
- ▶ des opérateurs de substitution (ici pour GnuMake)

```
SRC = ( wildcard *.c )
OBJ = $(SRC:.c=.o)
OBJECTS = $(subst %.c,%.o,$(SRC))
```

- ▶ des conditionnels (!)

```
libs_for_gcc = -lgnu
normal_libs =
foo: $(objects)
ifeq ($(CC),gcc)
    $(CC) -o foo $(objects) $(libs_for_gcc)
else
    $(CC) -o foo $(objects) $(normal_libs)
endif
```



Les règles de make

- ▶ commande pour refaire des objets :

```
%.o : %.c
# commands to execute (built-in):
$(COMPILE.c) $(OUTPUT_OPTION) $<
```

- ▶ commande pour refaire des exécutables

```
% : %.c
# commands to execute (built-in):
$(LINK.c) $^ $(LOADLIBES) $(LDLIBS) -o $@
```

- ▶ de nouveau : make -p
- ▶ Des règles avec cibles fictives

```
.PHONY: clean
clean:
  $(RM) -f *.o *
```

Quelques options de Make

- ▶ `-n` simule la reconstruction
- ▶ `-i` ignore les erreurs
- ▶ `-f [file]` le makefile est dans `file`
- ▶ `-q` état de la cible
- ▶ `gcc -MM` analyse les dépendances

Quelques options de Make

- ▶ `-n` simule la reconstruction
- ▶ `-i` ignore les erreurs
- ▶ `-f [file]` le makefile est dans `file`
- ▶ `-q` état de la cible
- ▶ `gcc -MM` analyse les dépendances

Quelques options de Make

- ▶ `-n` simule la reconstruction
- ▶ `-i` ignore les erreurs
- ▶ `-f [file]` le makefile est dans `file`
- ▶ `-q` état de la cible
- ▶ `gcc -MM` analyse les dépendances

Quelques options de Make

- ▶ **-n** simule la reconstruction
- ▶ **-i** ignore les erreurs
- ▶ **-f [file]** le makefile est dans `file`
- ▶ **-q** état de la cible
- ▶ **gcc –MM** analyse les dépendances

Quelques options de Make

- ▶ `-n` simule la reconstruction
- ▶ `-i` ignore les erreurs
- ▶ `-f [file]` le makefile est dans `file`
- ▶ `-q` état de la cible
- ▶ `gcc -MM` analyse les dépendances

Le(s) thème(s) du jour: void

Un tableau de chaînes

