INFO0030 – Initiation au langage C

Justus H. Piater

INFO0030[1] – Initiation au langage C

Justus H. Piater

[1] http://www.montefiore.ulg.ac.be/~piater/courses/INFO0030/plan.php

Table des matières

1. Séance 1 – Organisation et introduction au langage C 1
    1. (Introduction) 1
    2. Introduction 1
    3. Quelques instructions et expressions 3
2. Séance 2 – Style, opérateurs, types, flot de contrôle, tableaux 7
    1. (Introduction) 7
    2. Style, documentation, astuces 7
    3. Opérateurs 8
    4. Types et valeurs 9
    5. Davantage sur le flot de contrôle 10
    6. Tableaux 11
3. Séance 3 – Types dérivés, fonctions, préprocesseur 15
    1. Types dérivés 15
    2. Fonctions 16
    3. Variables 18
    4. Le préprocesseur 19
    5. Le mot final 20
4. Séance 4 – Pointeurs, chaînes 21
    1. Pointeurs 21
    2. Pointeurs et structures 23
    3. Pointeurs et tableaux 24
    4. Chaînes 26
    5. Résumé 27
5. Séance 5 – Pointeurs et tableaux avancés ; librairie standard 29
    1. (Introduction) 29
    2. Pointeurs avancés 29
    3. La librairie standard 32
    4. Résumé 34

Chapitre 1. Séance 1 – Organisation et introduction au langage C

1. (Introduction) 1
1.1. Organisation du cours 1
2. Introduction 1
2.1. Une petite histoire du langage C 1
2.2. Notre premier programme en C 2
2.3. Des sources à l'exécutable 2
2.4. Nombres 3
2.5. Entrée 3
2.6. Comment trouver les détails? 3
3. Quelques instructions et expressions 3
3.1. if 3
3.2. Expressions booléennes 4
3.3. while et do...while 4
3.4. Expressions arithmétiques 4
3.5. for 4
3.6. Un petit projet 5

1. (Introduction)

1.1. Organisation du cours

Vous trouvez tout sur la page Web du cours (http://www.montefiore.ulg.ac.be/~piater/courses/INFO0030/) :

- actualités
- détails, infos
- notes de cours
- énoncés, critères d'évaluation

Notre charge : que vous avanciez !

Parcourir la page Web du cours

Qui ne dispose pas d'un

- compte sur les machines d'algorithmique ?
- ordinateur personnel pour réaliser les travaux ?

La première séance de programmation supervisée

L'organiser !

2. Introduction

2.1. Une petite histoire du langage C

- ~1970 : Bell Laboratories a besoin d'un langage de programmation pour des implémentations portables du nouveau système d'exploitation UNIX.
- Dennis M. Ritchie et Brian W. Kernighan définissent la première version du langage C en se basant sur les langages BCPL et B.
- 1978 : Kernighan et Ritchie publient « The C programming language ». (Cette première édition est périmée.)

- Depuis 1983, Bjarne Stroustrup (Bell Labs) développe une extension orientée-objet nommée C++.
- 1989 : l'« American National Standards Institute » standardise le langage ANSI C. Le même standard est adopté en 1990 par l'« International Standards Organisation » (ISO-C90).
- 1999 : l'ISO définit un nouveau standard (ISO-C99) en intégrant des leçons apprises de C++.

2.2. Notre premier programme en C

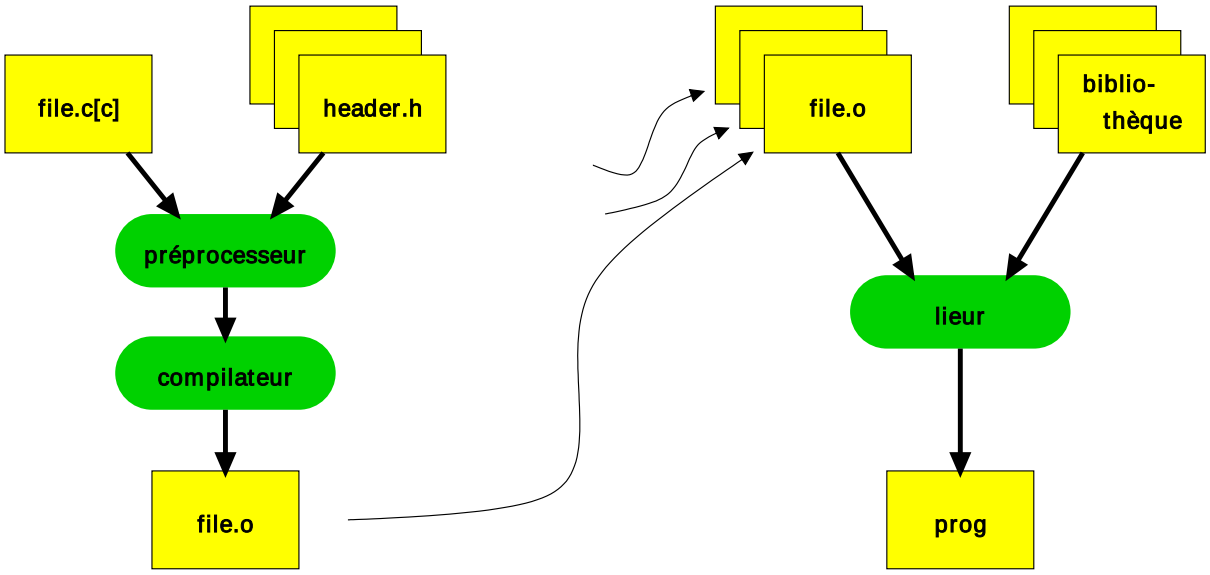
```c
#include <stdio.h>

/* Display the string "Hello World!" */

int main() {
  printf("Hello World!\n");
  return 0;
}
```

- `/* Voici un commentaire. */`
- L'unique fonction `main()` est toujours le point de départ d'un programme.
- `printf()` est une fonction pour afficher quelque chose sur la *sortie standard*.
- Il vaut mieux déclarer toute fonction avant de l'utiliser. La déclaration de `printf()` se trouve dans le *fichier d'en-tête* `stdio.h`.
- Le `"\n"` dénote un retour chariot.
- La valeur de retour est rendue au système d'exploitation. Par convention, 0 signifie succès.
- Chaque instruction est terminée par un point-virgule.
- Observez le bon style de codage (ici : pas plus d'une instruction sur une ligne ; indentation cohérente).

2.3. Des sources à l'exécutable



2.4. Nombres

```
#include <stdio.h>

int main() {
  int a = 5;
  int b = 7;
  int c = a + b;
  printf("%d + %d = %d\n", a, b, c);
  return 0;
}
```

- Le premier argument de `printf()` est une *chaîne de format* ; les arguments suivants y sont intégrés par les *spécificateurs de format*.

2.5. Entrée

```
#include <stdio.h>

int main() {
  int a;
  printf("Enter the first value: ");
  scanf("%d", &a);

  int b;
  printf("Enter the second value: ");
  scanf("%d", &b);

  printf("%d + %d = %d\n", a, b, a + b);
  return 0;
}
```

- `scanf()` est plus ou moins l'inverse de `printf()`.
- On l'utilise ici parce que c'est simple, mais la gestion d'erreurs est difficile.
- Il faut donner l'*adresse* de chaque élément lu.

2.6. Comment trouver les détails?

- Une référence en ligne[1]
- Pour les fonctions, consultez aussi les pages **man** de UNIX, Section 3, p.ex., **`man 3 printf`**.
- Des ouvrages de référence sur ANSI C ou ISO C

3. Quelques instructions et expressions

3.1. `if`

```
if (a < b) {
  …               // block
}
```

- Le **`if`** est suivi d'une expression *scalaire* entre parenthèses.
- Si le ***bloc*** contrôlé par le **`if`** ne contient qu'une seule instruction, les accolades ne sont pas nécessaires.

[1] http://ccs.ucsd.edu/c/

```
if (a < b) …
else …

if (a < b) …
else if …
else if …
else …
```

3.2. Expressions booléennes

Une expressions conditionnelle est « vraie » si sa valeur est différente de zéro.

Les opérateurs principaux : **<**, **<=**, **==**, **!=**, **>=**, **>**, **&&**, **||**, **!**

> ***Note***
>
> **=** est un opérateur d'affectation.

3.3. `while` et `do…while`

```
while (a < b) {
  …
}


do {
  …
} while (a < b);
```

3.4. Expressions arithmétiques

Les opérateurs principaux : **+**, **−**, *****, **/**, **%**

L'opérateur principal d'affectation : **=**

```
a = a + 1;
```

Il y a aussi des opérateurs qui combinent une opération arithmétique et une affectation :

```
a += 1;
a++;
a--;
```

3.5. `for`

La boucle **`for`** est une syntaxe abrégée pour un type répandu de boucle **`while`** :

```
int i = 0;
while (i < k) {
  …
  i++;
}


for (int i = 0; i < k; i++) {
  …
}
```

3.6. Un petit projet

```c
/* Project 0 by Justus Piater 2007-02-07
 *
 * This program reads a number from standard input,
 * and prints the sum of 0, 1, …, number.
 */

#include <stdio.h>

int main() {
  int max;
  printf("Enter a number: ");
  scanf("%d", &max);

  int sum = 0;
  for (int i = 0; i <= max; i++)
    sum += i;

  printf("The sum of the numbers from 0 to %d is %d.\n",
         max, sum);
  return 0;
}
```

Chapitre 2. Séance 2 – Style, opérateurs, types, flot de contrôle, tableaux

1. (Introduction) 7
    1.1. Le premier projet 7
2. Style, documentation, astuces 7
    2.1. Le style de codage : Quelques conseils 7
    2.2. Documenter un programme : où ? 8
    2.3. Outils 8
    2.4. Vérifier le succès d'un scanf() 8
    2.5. Récupérer d'un scanf() échoué 8
3. Opérateurs 8
    3.1. Évaluation raccourcie des opérateurs logiques 8
    3.2. L'opérateur conditionnel ?: 9
    3.3. Les opérateurs et leurs précédences 9
4. Types et valeurs 9
    4.1. Types primitifs : Les types de base 9
    4.2. Types primitifs : variantes 10
    4.3. Conversion de types 10
    4.4. Les valeurs littérales 10
    4.5. Quelques séquences d'échappement 10
5. Davantage sur le flot de contrôle 10
    5.1. for avancée et l'opérateur virgule 10
    5.2. break et continue 11
    5.3. switch 11
6. Tableaux 11
    6.1. Déclarer un tableau 11
    6.2. Utiliser un tableau 12
    6.3. Exemple : Bubble Sort 12
    6.4. Tableaux multidimensionnels 12
    6.5. Initialisation lors de la définition 13

1. (Introduction)

1.1. Le premier projet

La séance de programmation supervisée était-elle utile ?

Voici une bonne solution.

La montrer et discuter

Suivez l'énoncé méticuleusement, y compris les chaînes de caractères affichées ; sinon vous cassez notre évaluation automatique !

2. Style, documentation, astuces

2.1. Le style de codage : Quelques conseils

Objectif : la *lisibilité*

Principe	**Suggestion**
Choisissez un style, et *adhérez-y* !	à l'aide d'un bon éditeur
indentation systématique	2 espaces par niveau
position des accolades	voir exemples au tableau
espaces, lignes vides	`1 + 1`, `method()`, *structurer*
minuscules/majuscules	`MY_CONST`, `MyClasse`, `myVar`, `myMethod()`
noms des identifiants	les plus parlants possible
langue	anglais préféré

Consultez mon *Guide to Coding Style*[1].

2.2. Documenter un programme : où ?

- En *blocs de commentaires* :
  - Spécifier le contrat de chaque méthode.
  - Préciser le rôle de chaque champ.
  - Si non évident, expliquer chaque algorithme.
- En *petits commentaires* :
  - Annoter toutes les opérations qui ne sont pas immédiatement évidentes.
- Vos commentaires feront partie de la cotation de vos projets.

2.3. Outils

- Utilisez un bon éditeur qui automatise beaucoup de choses pour vous.
  (Emacs vous offre un tutoriel dans le menu Help.)
- Apprenez à utiliser un débogueur (voir le site du cours).

2.4. Vérifier le succès d'un `scanf()`

```c
#include <stdio.h>

int main() {
  int a, b;
  printf("Enter two numbers: ");
  int result = scanf("%d%d", &a, &b);
  printf("Result: %d\n", result);
}
```

- Dans **`man 3 scanf`**, il y a une section « RETURN VALUE ».

2.5. Récupérer d'un `scanf()` échoué

Tampon d'entrée : ***Toute entrée*** par l'utilisateur y est enregistrée, indépendamment des `scanf()` etc.

Suite à l'entrée par l'utilisateur, le tampon d'entrée contient tous les caractères entrés par l'utilisateur, y compris le retour à la ligne (`'\n'`).

Toutes les conversions (sauf `"%c"`) s'arrêtent avant le `'\n'`, qui reste dans le tampon d'entrée.

La prochaine lecture retrouve ce `'\n'`, mais toutes les conversions le sautent avec tout *espace blanc* initial (sauf `"%c"`), y compris `" %c"`.

Par conséquent, pour supprimer des caractères erronés du tampon d'entrée, bouclez en lisant un caractère à la fois, jusqu'à la lecture du premier `'\n'`.

3. Opérateurs

3.1. Évaluation raccourcie des opérateurs logiques

Si `a` est vraie, l'expression `b` n'est pas évaluée :

[1] http://www.montefiore.ulg.ac.be/~piater/courses/#coding-style

```
a || b
```

Si `c` est fausse, l'expression `d` n'est pas évaluée :

```
c && d
```

3.2. L'opérateur conditionnel ?:

```
min = (a < b) ? a : b;
```

- L'expression avant le **?** doit être d'un type scalaire.
- Les types des deux expressions alternatives doivent être identiques.

3.3. Les opérateurs et leurs précédences

postfixe	***[] . -> (***params***)*** *expr***++** *expr***--**
préfixe	**++***expr* **--***expr* **+***expr* **-***expr* **~ ! &***expr* *******expr* ***sizeof (***type***)***expr*
multiplicatifs	*** / %**
additifs	**+ -**
décalages	**<< >>**
comparaisons	**< > <= >=**
égalité	**== !=**
ET binaire	**&**
OU exclusif binaire	**^**
OU binaire	**|**
ET logique	**&&**
OU logique	**||**
conditionnel	**?:**
affectations	**= += -= *= /= %= <<= >>= &= ^= |=**
comma	**,**

4. Types et valeurs

4.1. Types primitifs : Les types de base

`bool`	**`true`** ou **`false`** (ISO-C99, avec **`#include`** `"stdbool.h"`)
`char`	caractère signé en 1 *byte* (**`sizeof`**(**`char`**) ≡ 1), ≥8 bit (voir aussi **`wchar_t`**)
`int`	entier signé, ≥16 bit
`float`	flottant, ≥6 chiffres décimaux

double flottant, ≥10 chiffres décimaux

4.2. Types primitifs : variantes

- Les spécificateurs : **unsigned short long**
  - **unsigned char|short|int|long**
  - **short int**
  - **long int|double**
- Les qualificateurs :

  const la valeur stockée est fixe

  volatile la valeur stockée est susceptible d'être changée par des moyens hors du contrôle du compilateur

4.3. Conversion de types

Implicite (***promotion***) et explicite (***cast***) :

```
int a = 3, b = 2;

a / b          // 1

(float)a / b   // 1.5
```

4.4. Les valeurs littérales

bool : **true**, **false** (ISO-C99)

entiers : 29 = 035 = 0x1D = 0X1d

flottants : 18. = 1.8e1 = .18E2 = 18e0 ; 1d, 1F, −0d ; 3.14L

caractères : `'a'`, `'\n'` = `'\x0A'` = `'\X0a'`

chaînes : `"quelques caractères"`

4.5. Quelques séquences d'échappement

`\"` " (à l'intérieur d'une chaîne)

`\'` ' (le caractère `'\''`)

`\n` nouvelle ligne

`\t` caractère de tabulation

`\x1A` la valeur 26 (1 ou 2 chiffres hexadécimaux)

`\0` la valeur 0

5. Davantage sur le flot de contrôle

5.1. for avancée et l'opérateur virgule

```
for (/* statement, evaluated once before entering */;
     /* conditional, evaluated before each loop */;
```

```
     /* statement, evaluated after each loop */) …

// Possible, but barely readable:
for (a = 4, printf("Here we go:\n");
     printf("Let's see: "), z < 3;
     printf("We made it!\n"), z += a);
```

On peut omettre n'importe quelle des trois expressions. Pour le gardien, une expression absente est « vraie ».

5.2. break et continue

```
#include <stdbool.h>  // ISO-C99
…

  bool quit = false;
  bool relevant = true;

  while (true) {        // = for (;;)
    …
    if (quit)
      break;
    if (!relevant)
      continue;
    …
  }
```

5.3. switch

```
switch (a % 4) {
case 0:
  printf("4 divides %d\n", a);
  // fall-through
case 2:
  printf("%d is pair\n", a);
  break;
case 1:
case 3:
  printf("%d is impair\n", a);
  break;
default:
  printf("How on earth did I get here?\n");
}
```

6. Tableaux

6.1. Déclarer un tableau

```
int a[5];

a[0] = 12;
a[4] = 3;

a[5] = 2;   // legal, but bug!
a[-1] = 1;  // legal, but bug!


int size = …;
```

```
int b[size];     // C99
```

- En pré-C99, la taille doit être spécifiée par une expression constante (c'est-à-dire, sa valeur est connue par le compilateur).

6.2. Utiliser un tableau

```
#include <stdio.h>

int main() {
  // specify the size at a single location:
  const int N_ELEMS = 5;
  int a[N_ELEMS];         // C99

  for (int i = 0; i < N_ELEMS; i++)
    a[i] = i;
  for (int i = 0; i < N_ELEMS; i++)
    printf("a[%d] = %d\n", i, a[i]);

  return 0;
}
```

6.3. Exemple : *Bubble Sort*

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

int main() {
  const int N_ELEMS = 10;
  int a[N_ELEMS];           // C99

  for (int i = 0; i < N_ELEMS; a[i++] = rand()); // poor style

  // bubble-sort the array:
  for (int i = 0; i < N_ELEMS - 1; i++) {
    for (int j = 0; j < N_ELEMS - i - 1; j++) {
      if (a[j] > a[j+1]) {
        int t = a[j];
        a[j] = a[j + 1];
        a[j + 1] = t;
      }
    }
  }

  for (int i = 0; i < N_ELEMS; i++)
    printf("a[%d] = %d\n", i, a[i]);

  return 0;
}
```

6.4. Tableaux multidimensionnels

```
double mat[3][4];

mat[0][1] = 0;  // the second element
mat[2][3] = 0;  // the last element


// don't try this:
```

```
int arr[1000][1000][1000];
```

6.5. Initialisation lors de la définition

```
int ints[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

int pascalsTriangle[][5] = {
  { 1 },            // the others are zero
  { 1, 1 },
  { 1, 2, 1 },
  { 1, 3, 3, 1 },
  { 1, 4, 6, 4, 1 }
};
```

Chapitre 3. Séance 3 – Types dérivés, fonctions, préprocesseur

1. Types dérivés 15
1.1. Constantes discrètes typées 15
1.2. Structures 15
1.3. Unions 16
1.4. Variantes syntaxiques 16
1.5. Définition de nouveaux types 16
2. Fonctions 16
2.1. Exemple : factorielle 16
2.2. Les Fonctions 17
2.3. Exemple : factorielle avec déclaration préalable 17
2.4. Exemple : factorielle récursive 17
2.5. return void 17
2.6. Passage par valeur 18
2.7. Listes d'arguments variables 18
2.8. Modulariser le code 18
3. Variables 18
3.1. Variables automatiques 18
3.2. Variables statiques 18
4. Le préprocesseur 19
4.1. Inclure du code C 19
4.2. Définir des macros 19
4.3. Compilation conditionnelle 19
4.4. Contrôler l'inclusion 19
5. Le mot final 20
5.1. Comment implémenter un programme modulaire 20
5.2. Un Makefile simple et générique 20
5.3. Un Makefile pour un projet modulaire 20
5.4. Résumé 20

1. Types dérivés

1.1. Constantes discrètes typées

```
enum season_t {
  SPRING, SUMMER, FALL, WINTER
} season;

if (season == FALL) …

enum corners_t { TRIANGLE = 3, QUADRANGLE };
enum corners_t shape = QUADRANGLE;

shape = FALL;    // compile-time error
```

1.2. Structures

```
struct Complex {
  float real, imaginary;
};


  …
  struct Complex a, b;
  a.real = 1.23;
  b = a;
```

1.3. Unions

```
union number {
  int i;
  float f;
};

  …
  union number n;
  n.i = 3;
  float f = n.f;   // bug!
```

1.4. Variantes syntaxiques

```
struct Complex {
  float real, imaginary;
} a, b;


struct {
  float real, imaginary;
} c;


struct {
  struct Complex c[4];
  int id;
} nested;
```

1.5. Définition de nouveaux types

```
typedef struct Complex_t {
  float real, imaginary;
} Complex;

typedef struct {
  float real, imaginary;
} Complex;

typedef int integer;


  …
  Complex c;
  integer b;
```

2. Fonctions

2.1. Exemple : factorielle

```
#include <stdio.h>

static int factorial(int n) {
  int fac = 1;
  for (int i = 1; i <= n; i++)
    fac *= i;
  return fac;
}
```

```c
int main() {
  const int VALUE = 4;
  printf("%d! = %d\n", VALUE, factorial(VALUE));
  return 0;
}
```

2.2. Les Fonctions

- Chaque fonction doit être *déclarée* (ou définie) avant son utilisation.
- Chaque fonction prend zéro, un ou plusieurs arguments.
- Les arguments sont passés *par valeur*.
- Chaque fonction renvoie une valeur d'un type donné, ou **void**.
- Déclarez **static** toute fonction qui n'est utilisée que dans ce fichier source.

2.3. Exemple : factorielle avec déclaration préalable

```c
#include <stdio.h>

static int factorial(int n);

int main() {
  const int VALUE = 4;
  printf("%d! = %d\n", VALUE, factorial(VALUE));
  return 0;
}

static int factorial(int n) {
  int fac = 1;
  for (int i = 1; i <= n; i++)
    fac *= i;
  return fac;
}
```

2.4. Exemple : factorielle récursive

```c
#include <stdio.h>

static int factorial(int n) {
  if (n <= 0)
    return 1;
  return n * factorial(n - 1);
}


int main() {
  const int VALUE = 4;
  printf("%d! = %d\n", VALUE, factorial(VALUE));
  return 0;
}
```

Pourquoi <= au lieu de == ?

2.5. return void

```c
// This function is stupid:
```

```
void count(int from, int to) {
  for (; from <= 9999; from++)
    if (from == to)
      return;
}
```

Noter l'instruction vide dans la boucle **for**.

2.6. Passage par valeur

```
// This function has no effect:
void swap(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
}
```

2.7. Listes d'arguments variables

```
printf("Hello\n");
printf("Hello %s\n", name);
printf("%d + %d = %d\n", a, b, a + b);
```

Il y a une syntaxe pour déclarer des fonctions telles que `printf()` ; voir `stdarg.h`.

C'est dangereux parce qu'on contourne le *contrôle des types*.

2.8. Modulariser le code

- Mettre les fonctions *généralement utiles* dans un ou plusieurs fichiers.
- Déclarer les fonctions *auxiliaires* **static**.
- Mettre toutes les *déclarations publiques* dans un ou plusieurs fichiers `.h`.
- Simplifiez votre vie avec l'outil **make** !

3. Variables

3.1. Variables automatiques

Les variables ***automatiques*** n'*existent* que dans leur ***portée***.

```
void procedure() {
  int i = 5;

  for (int k = 0; k < 1000; k++) {
    int i = k;
  }

  // k == ?
  // i == ?
}
```

3.2. Variables statiques

Les variables ***statiques*** ne sont *accessibles* que dans leur portée, mais elles *existent toujours*.

L'initialisation est calculée lors de la compilation.

```
static int k = 42;   // file scope
```

```
int counter() {
  static int c = 0;  // different from the statement c = 0
  return ++c;
}
```

4. Le préprocesseur

4.1. Inclure du code C

```
#include <stdlib.h>
#include "mylib.h"
```

Note

Dans un fichier ainsi inclu, on ne met que des *déclarations* (pas de *définitions*), sauf parfois des définitions **static**.

4.2. Définir des macros

```
#define MAXVAL 3
#define MAX(a,b) ((a) > (b) ? (a) : (b))


  …
  c = MAX(d++, sqrt(e))  // ouch!
```

Note

Évitez **#define** autant que possible. Préférez les valeurs typées **const** et les fonctions **inline** :

```
static const int MAXVAL = 3;

static inline int max(int a, int b) {
  return a > b ? a : b;
}
```

Les macros sont indépendants des types – avantage ou inconvénient ?

4.3. Compilation conditionnelle

```
#ifdef DEBUG      // or: #if defined DEBUG
  printf("Got here safely\n");
#endif

#if 0
  // exclude this code (quick and dirty)
#else
  // use this instead
#endif

#if 0
  // exclude this
#  if 1
  // this is excluded too
#  endif
#endif
```

4.4. Contrôler l’inclusion

```
// myheader.h
```

```
#ifndef _MYHEADER_H_
#define _MYHEADER_H_

…

#endif
```

5. Le mot final

5.1. Comment implémenter un programme modulaire

Illustrer par le prochain projet.

- « Implémenter » veut toujours dire « implémenter et ***tester*** ».
  Commencez par l'implémentation des tests.
- Implémentez *une* fonctionnalité à la fois, en ordre croissant d'envergure.
- *Décomposez* les calculs complexes en morceaux simples (du haut vers le bas), et réalisez l'implémentation en *intégrant* les morceaux (du bas vers le haut).
- Procédez par *petits* morceaux, recompilant et testant souvent.

5.2. Un `Makefile` simple et générique

Évitez de retaper les commandes de compilation.

Avec un `Makefile` contenant

```
CFLAGS = --std=c99 --pedantic -Wall -W -Wmissing-prototypes
```

tapez **make primetest** pour créer `primetest` à partir de `primetest.c`.

Afin de pouvoir déboguer votre programme (p.ex., avec **gdb**), utilisez plutôt

```
CFLAGS = --std=c99 --pedantic -Wall -W -Wmissing-prototypes -g
LDFLAGS = -g
```

5.3. Un `Makefile` pour un projet modulaire

Avec un `Makefile` contenant

```
CFLAGS = --std=c99 --pedantic -Wall -W -Wmissing-prototypes -g
LDFLAGS = -g

ratiocalc: Rational.o ratiocalc.o
```

tapez **make** pour créer `ratiocalc` à partir de `Rational.c` et `ratiocalc.c`.

5.4. Résumé

- Types dérivés (**`enum`**, **`struct`**, **`typedef`**)
- Fonctions ; **`static`**
- Variables et leurs portées ; **`static`**
- Préprocesseur (**`#include`**, **`#define`**, **`#ifdef`**, **`#if`**)
- **make**

Chapitre 4. Séance 4 – Pointeurs, chaînes

1. Pointeurs 21
1.1. Pointeurs – à quoi bon ? 21
1.2. Exemple 21
1.3. Variables et adresses 22
1.4. Des pointeurs multiples 22
1.5. Passage par référence (adresse) 22
1.6. Pointeurs : adresses typées 22
1.7. Pointeurs constants 23
2. Pointeurs et structures 23
2.1. Allouer et libérer de la mémoire 23
2.2. Pointeurs valides 23
2.3. Allouer une structure 24
2.4. Utiliser une structure allouée 24
2.5. Structures liées 24
3. Pointeurs et tableaux 24
3.1. Allouer un tableau 24
3.2. Adresser un tableau 25
3.3. Tableaux automatiques et dynamiques 25
3.4. Tableaux comme paramètres 25
4. Chaînes 26
4.1. Les chaînes de caractères 26
4.2. Les arguments sur la ligne de commande 26
4.3. Pointeurs et tableaux 26
4.4. Copier une chaîne 26
4.5. Illustration : strcpy() 26
4.6. Illustration : strlen() 27
5. Résumé 27
5.1. Résumé 27

1. Pointeurs

1.1. Pointeurs – à quoi bon ?

Un pointeur est une *adresse*.

Quelques cas d'utilisation :

- des structures de données dynamiques
- passage de paramètres modifiables (« par référence »)
- accès flexible aux tableaux
- efficacité

1.2. Exemple

```
#include <stdio.h>

int main() {
  int i, j;
  int* p;   // a pointer to an integer

  p = &i;
  *p = 5;
  j = i;
```

```
  printf("%d %d %d\n", i, j, *p);
  return 0;
}
```

dessiner un schématique

1.3. Variables et adresses

- L'*identifiant* d'une *variable* dénote un *emplacement*, à une *adresse* donnée, contenant une *valeur* d'un certain *type*.
- Un ***pointeur*** est une *variable* dont la *valeur* est une *adresse* (d'une autre variable, ...)

1.4. Des pointeurs multiples

```
#include <stdio.h>

int main() {
  int i = 0, j;
  int* p = &i;
  int* q = &i;
  int* r = q;

  (*p)++;
  j = *r;

  printf("%d = %d = %d = %d = %d\n",
         i, *p, *q, *r, j);

  return 0;
}
```

1.5. Passage par référence (adresse)

```
#include <stdio.h>

void swap(int* a, int* b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}

int main() {
  int a, b;
  printf("Enter two integers: ");
  scanf("%d%d", &a, &b);

  printf("a = %d, b = %d\n", a, b);
  swap(&a, &b);
  printf("a = %d, b = %d\n", a, b);

  return 0;
}
```

1.6. Pointeurs : adresses typées

```
  int* pi;
  float* pf;
  …
```

```
  pi = pf;            // illegal
  pi = (int*)pf;      // legal, but usually bug or poor design

  int i = *pf;        // OK - type conversion
  int j = (int)*pf;   // type conversion made explicit

  int k = *(int*)pf;  // legal, but usually bug or poor design

  void* pv;           // generic (typeless) pointer
  void v = *pv;       // 2x illegal

  int** ppi = &pi;    // pointer to a pointer
  int l = **ppi;
```

1.7. Pointeurs constants

```
int* const pi;          // pi cannot be changed
int const* pi;          // elements at *pi cannot be changed
const int* pi;          // ditto (preferred)
const int* const pi;

int* const* ppi;
const int* const* const ppi;

void f(const int* pi) {
  …
}
```

2. Pointeurs et structures

2.1. Allouer et libérer de la mémoire

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


int main() {
  int* p = (int*)malloc(sizeof(*p)); // see also calloc()
  if (!p) {
    printf("ERROR: Out of memory\n");
    return 1;
  }

  *p = 5;
  printf("%d\n", *p);

  free(p);
  return 0;
}
```

2.2. Pointeurs valides

- Un pointeur doit contenir une adresse valide pour que l'on puisse le déréférencer.
- Après un `free()`, une adresse n'est plus valide.
- L'adresse `0` (`NULL`) n'est jamais valide.
- Par convention, un pointeur `NULL` signifie un pointeur qui ne se réfère à rien.

- `NULL` est défini par inclusion de presque n'importe quel fichier en-tête standard (sinon, `stddef.h`).

2.3. Allouer une structure

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

struct rec {
  int i;
  char c;
};

int main() {
  struct rec* p = (struct rec*)malloc(sizeof(struct rec));

  (*p).i = 10;
  (*p).c = 'a';
  printf("%d %c\n", (*p).i, (*p).c);

  free(p);
  return 0;
}
```

Il y a un problème avec ce code. Lequel ? (Manque la vérification de `p`)

2.4. Utiliser une structure allouée

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

struct rec {
  int i;
  char c;
};

int main() {
  struct rec* p = (struct rec*)malloc(sizeof(struct rec));

  p->i = 10;
  p->c = 'a';
  printf("%d %c\n", p->i, p->c);

  free(p);
  return 0;
}
```

2.5. Structures liées

```c
typedef struct node_t {
  void* content;
  struct node_t* next;
} Node;
```

3. Pointeurs et tableaux

3.1. Allouer un tableau

```c
#include <stdlib.h>
```

```
int main() {
  const int TABSIZE = 42;
  int* p = (int*)malloc(TABSIZE * sizeof(*p));

  for (int i = 0; i < TABSIZE; i++)
    p[i] = 2 * i;

  free(p);
  return 0;
}
```

3.2. Adresser un tableau

```
  const int TABSIZE = 42;
  int* p = (int*)malloc(TABSIZE * sizeof(*p));

  for (int i = 0; i < TABSIZE; i++)
    p[i] = 2 * i;

  for (int i = 0; i < TABSIZE; i++)
    *(p + i) = 2 * i;

  for (int* q = p; q < p + TABSIZE; q++)
    *q = 2 * (q - p);

  free(p);
```

3.3. Tableaux automatiques et dynamiques

```
  int ints[5];
  char* argv[argc];
  struct complex c[size];

  int* const ints = (int*)malloc(5 * sizeof(*ints));
  char** const argv = (char**)malloc(argc * sizeof(*argv));
  struct complex* const c =
    (struct complex *)malloc(size * sizeof(*c));

  for (int i = 0; i < size; i++)
    c[i].imaginary = 0;
```

La sémantique de `ints` est la même partout – une *adresse typée* (avec des règles de calcul d'indices) :

`ints = &ints[0]`

3.4. Tableaux comme paramètres

```
int a[5];                          // sizeof(a) = 5 * sizeof(int)
int* const b = (int*)malloc(5 * sizeof(*b));

f(a, 5);
f(b, 5);

void f(int ints[], int nElems);
void f(int ints[5], int nElems);  // sizeof(ints) = sizeof(int*)
void f(int* const ints, int nElems);
```

4. Chaînes

4.1. Les chaînes de caractères

Une ***chaîne*** n'est rien d'autre qu'un *tableau de caractères*, terminé par la valeur de *zéro*, et un peu de syntaxe pratique.

Un tas de fonctions dans la librairie standard : `strncat()`, `strchr()`, `strrchr()`, `strcmp()`, `strncpy()`, `strlen()`, `strstr()`, `atoi()`, ...

4.2. Les arguments sur la ligne de commande

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


int main(int argc, char* argv[]) {
  for (int i = 0; i < argc; i++)
    printf("Arg %d = %s\n", i, argv[i]);

  return 0;
}
```

4.3. Pointeurs et tableaux

```
const char* const s = "hello";
const char t[] = "hello";

return t; // bug!


const char* s = "hello";
char t[] = "hello";

s = "hello again";        // OK
t = "hello again";        // illegal


char* s = "hello";        // poor style
```

4.4. Copier une chaîne

```
  char dest[STRLEN];
  strncpy(dest, src, sizeof(dest));
  dest[sizeof(dest) - 1] = 0;

  char* dest = (char*)malloc(STRLEN * sizeof(*dest));
  if (dest) {
    strncpy(dest, src, STRLEN);
    dest[STRLEN - 1] = 0;
  }

  char* dest = (char*)malloc(strlen(src) + 1);
  if (dest)  strcpy(dest, src);

  char* dest = strdup(src);
```

4.5. Illustration : `strcpy()`

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

```
char* strcpy(char* dest, const char* src) {
  char* d = dest;
#ifdef CLASSICAL
  while (*src)
    *d++ = *src++;
  *d = 0;
#else
  do {
    *d++ = *src;
  } while (*src++); // no *d = 0 at the price of readability
#endif

  return dest;
}
```

4.6. Illustration : `strlen()`

```
#include <string.h>


size_t strlen(const char* str) {
  const char* s = str;

  while (*s++);

  return s - str - 1;
}
```

5. Résumé

5.1. Résumé

Pointeurs :

- accès par référence
- allocation dynamique de mémoire
- tableaux
- chaînes

Chapitre 5. Séance 5 – Pointeurs et tableaux avancés ; librairie standard

1. (Introduction) 29
    1.1. Un bogue 29
2. Pointeurs avancés 29
    2.1. Pointeurs opaques 29
    2.2. Implémentation du type opaque (Treasure.c) 30
    2.3. Les tableaux multidimensionnels 30
    2.4. Les tableaux de pointeurs 30
    2.5. L'accès aux éléments 31
    2.6. Pointeurs aux fonctions 31
    2.7. Gestionnaire de signaux 31
    2.8. Polymorphisme : du code utilisateur 32
    2.9. Polymorphisme : la librairie générique 32
3. La librairie standard 32
    3.1. Caractères : is...() 32
    3.2. Constantes utiles 33
    3.3. Math 33
    3.4. Entrée / sortie 33
    3.5. Manipulation de chaînes 33
    3.6. Fonctions utiles 33
    3.7. Gestion d'erreurs 33
    3.8. assert() 34
4. Résumé 34
    4.1. Résumé 34

1. (Introduction)

1.1. Un bogue

```c
#include <stdio.h>

int main() {
  int s[4], t[4];

  for (int i = 0; i <= 4; i++)
    s[i] = t[i] = i;

  printf("i:s:t\n");
  for (int i = 0; i <= 4; i++)
    printf("%d:%d:%d\n", i, s[i], t[i]);

  return 0;
}
/*  0:4:0
    1:1:1
    2:2:2
    3:3:3
    4:4:4  */
```

2. Pointeurs avancés

2.1. Pointeurs *opaques*

• Dans l'interface (`Treasure.h`) :

```
typedef struct Treasure_t Treasure;

Treasure* newTreasure();
void deleteTreasure(Treasure* t);
void fillTreasure(Treasure* t, int euros);
```

- Pour l'utilisateur :

```
#include "Treasure.h"

void f() {
  Treasure* t = newTreasure();
  fillTreasure(t, 42);
  deleteTreasure(t);
}
```

2.2. Implémentation du type opaque (`Treasure.c`)

```
#include "Treasure.h"
#include <stdlib.h>

struct Treasure_t {
  int accessCount;
  int euros;
};


Treasure* newTreasure() { return malloc(sizeof(Treasure)); }

void deleteTreasure(Treasure* t) { free(t); }

void fillTreasure(Treasure* t, int euros) {
  t->euros += euros;
  t->accessCount++;
}
```

2.3. Les tableaux multidimensionnels

```
double mat[3][4];
double (*const mat)[4] =
  (double(*)[4])malloc(3 * sizeof(*mat));

mat[2][3] = 0.0;

f(mat);
void f(double m[][4]);
void f(double (*const m)[4]);
```

- La sémantique de `mat` est la même partout (une adresse avec des règles de calcul d'indices).
- *Stockage :* Le dernier indice varie le plus vite.
- Pour le C pré-ISO-1999, les tailles de toutes les dimensions devaient être connues lors de la compilation, sauf la première (pour le calcul des indices).

2.4. Les tableaux de pointeurs

```
double* const tab[3];
```

```
double** const tab = (double**)malloc(3 * sizeof(*tab));

for (int i = 0; i < 3; i++)
  tab[i] = (double*)malloc(4 * sizeof(**tab));

tab[2][3] = 0.0;
```

Note

```
void f(double m[][4]);
f(tab);  // error

void g(double** const m);
void g(double* const m[]);
g(tab);
```

2.5. L'accès aux éléments

Syntaxe identique, logique différente :

- calcul d'indices
- suivi de pointeurs

2.6. Pointeurs aux fonctions

```
// In stdlib.h:
void qsort(void* base, size_t nmemb, size_t size,
           int(*compar)(const void*, const void*));

// Your code:

static int compareStrings(const void* p1, const void* p2) {
  /* p1 and p2 are "pointers to pointers to char",
     but strcmp(3) arguments are "pointers to char": */
  return strcmp(*(char* const*)p1, *(char* const*)p2);
}

int main(int argc, char* argv[]) {
  qsort(&argv[1], argc - 1, sizeof(argv[1]), compareStrings);
  for (int i = 1; i < argc; i++)
    puts(argv[i]);
  return 0;
}
```

2.7. Gestionnaire de signaux

```
// In signal.h:

typedef void (*sighandler_t)(int);

sighandler_t signal(int signum, sighandler_t handler);
void (*signal(int signum, void (*handler)(int)))(int);

#define SIGINT 15


// Your code:

void myHandler(int signum) { /* … */ }
```

```
void f() {
  sighandler_t prevHandler = signal(SIGINT, myHandler);
  (*prevHandler)(SIGINT);
}
```

2.8. Polymorphisme : du code utilisateur

```
#include <stdio.h>
#include "Object.h"

static void displayInteger(const void* integerPtr) {
  printf("%d\n", *(const int*)integerPtr);
}

static void displayString(const void* stringPtr) {
  printf("%s\n", (const char*)stringPtr);
}

int main(int argc, char* argv[]) {
  Object integer = { &argc, displayInteger };
  Object string = { argv[0], displayString };

  displayObject(&integer);
  displayObject(&string);
  return 0;
}
```

2.9. Polymorphisme : la librairie générique

• Interface `Object.h` :

```
typedef struct {
  void* data;
  void (*display)(const void* data);
} Object;


void displayObject(const Object* obj);
```

• Implémentation `Object.c` :

```
#include "Object.h"

void displayObject(const Object* obj) {
  obj->display(obj->data);
}
```

3. La librairie standard

3.1. Caractères : is…()

```
// defined in <ctype.h>:
int isalnum(int c);
int isalpha(int c);
int iscntrl(int c);
int isdigit(int c);
int isgraph(int c);
int islower(int c);
```

```
int isprint(int c);
int ispunct(int c);
int isspace(int c);
int isupper(int c);
int isxdigit(int c);
int tolower(int c);
int toupper(int c);
```

3.2. Constantes utiles

```
// in <float.h>:
#define DBL_EPSILON … // min eps with 1.0 + eps != 1.0
#define DBL_MAX …
#define DBL_MIN …
#define FLT_…

// in <limits.h>:
#define INT_MAX …
#define INT_MIN …
#define LONG_…
#define U…

// in <stddef.h>:
#define NULL …        // null pointer
typedef … size_t;     // for sizes of C objects
typedef … ptrdiff_t;  // for differences of two pointers
```

3.3. Math

Définies dans `math.h` :

`fabs()`, `ceil()`, `floor()`, `exp()`, `pow()`, `log()`, `sin()`, `cos()`, `tan()`, `atan2()`, …

3.4. Entrée / sortie

Définies dans `stdio.h` :

`fopen()`, `fclose()`, `fgets()`, `fputs()`, `fprintf()`, `fread()`, `fwrite()`, `getc()`, `printf()`, `sprintf()`, `scanf()`, …

`stdin`, `stdout`, `stderr`, …

3.5. Manipulation de chaînes

Définies dans `string.h` :

`memcpy()`, `memset()`, `strncat()`, `strchr()`, `strrchr()`, `strcmp()`, `strncpy()`, `strlen()`, `strstr()`, …

3.6. Fonctions utiles

Définies dans `stdlib.h` :

`abs()`, `atoi()`, `malloc()`, `free()`, `qsort()`, `rand()`, …

3.7. Gestion d’erreurs

Beaucoup de fonctions de la librairie standard placent un code d’erreur dans la variable globale `errno` (`errno.h`).

Avant d’appeler une telle fonction, il faut mettre `errno` explicitement à zéro.

On peut afficher un message d’erreur correspondant sur `stderr` avec `perror()` (`stdio.h`).

On peut récupérer le message d’erreur avec `strerror()` (`string.h`).

3.8. assert()

```
// #define NDEBUG
#include <assert.h>

  …
  assert(k > 0);
  …
```

Si la condition n'est pas vraie, un message est affiché sur `stderr`, et `abort()` est appelée.

Astuce

Génial pour *tester* et *clarifier* le code en même temps.

4. Résumé

4.1. Résumé

- Pointeurs opaques
- Tableaux multidimensionnels
- Pointeurs aux fonctions
- Survol extrêmement incomplet de la librarie standard
  (Regardez-la.)