Partie 3

Organisation de programmes

7 octobre 2019

Plan

1. Programmation modulaire
2. Masquage de l'information
3. Généricité
4. Compilation
5. Tests et débogage
6. Style

Plan

Programmation modulaire : principes

Un programme (en C ou un autre langage) est souvent découpé en une série de modules indépendants.

Un module est une collection de services, mis à la disposition de clients, c'est-à-dire d'autres modules ou programmes.

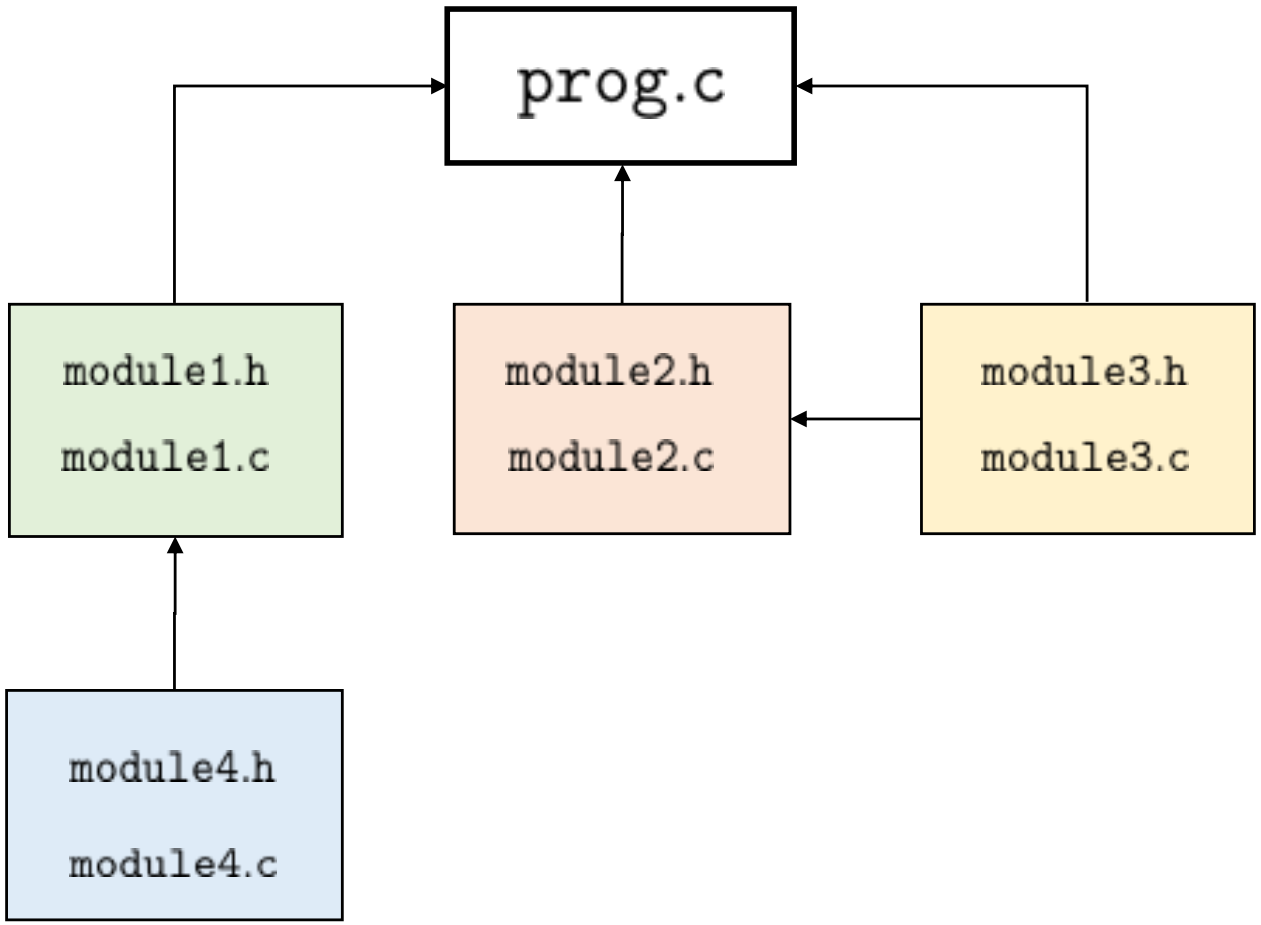
Chaque module a une interface qui décrit précisément les services qu'il fournit.

Les détails du module sont contenus dans son implémentation.

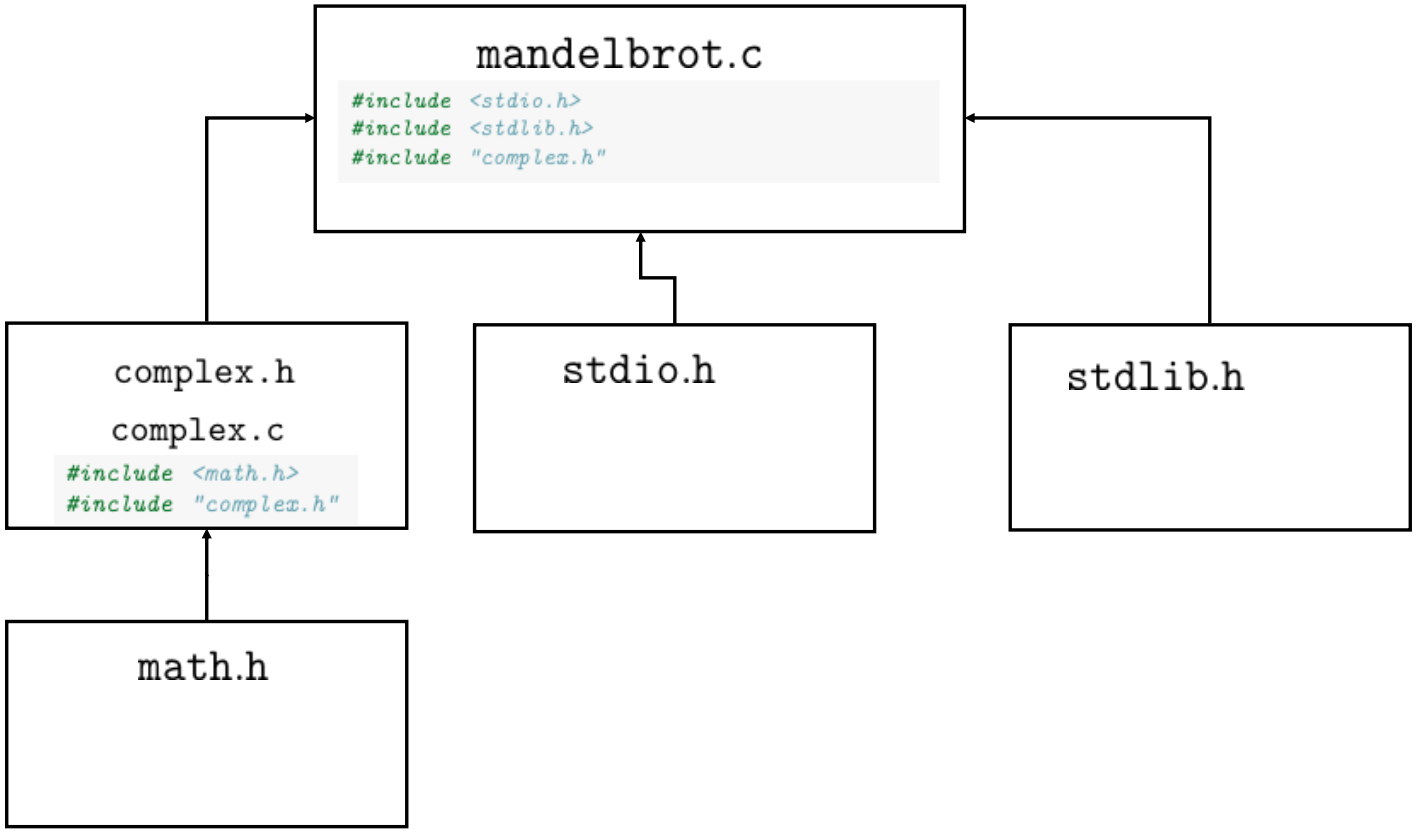
En C :

- Chaque service correspond à une fonction ou procédure.
- L'interface est précisée dans un fichier d'entête (`.h`).
- L'implémentation est fournie dans un fichier source (`.c`).

Programmation modulaire : illustration



Exemple : Mandelbrot



(Code : voir slides 34 et 35)

Avantages de l'approche modulaire

Abstraction :

- Chaque module peut être traité comme une abstraction : on sait ce qu'il fait mais on ne se tracasse pas de comment il le fait.
- Une fois que l'interface du module est bien précisée, on peut séparer les tâches d'implémentation module par module.

Réutilisabilité :

- Tout module est potentiellement réutilisable par d'autres clients.
- Les modules peuvent (devraient) être designés avec cet objectif en tête.

Maintenabilité :

- Rend la maintenance du programme plus aisée.
- On peut corriger, améliorer chaque module séparément.

Découpage en modules

Pas facile de déterminer le découpage optimal. Souvent plusieurs bonnes solutions.

Deux propriétés principales d'un bon module :

- Forte cohésion : les services offerts au sein d'un même module doivent être liés les uns aux autres.
- Couplage faible : les modules doivent être aussi indépendants que possible les uns des autres.

Exemples

- Bons modules : fonctions de manipulations de matrices, fonctions de traitement de chaînes de caractères (`<string.h>`), définition de structures de nombres complexes.
- Mauvais modules : modules 'help', 'truc', ou 'divers' contenant toutes les fonctions auxiliaires d'un programme.

Grands types de modules

Data pool : une collection de variables et de constantes liées entre elles.

- Exemples : `<limits.h>` et `<float.h>`.

Librairie (bibliothèque) : une collection de fonctions liées entre elles

- Exemples : ensemble de fonctions mathématiques `<math.h>`, fonctions de manipulations de chaînes de caractères `<string.h>`.

Objet abstrait : une collection de fonctions opérant sur une structure de données cachées

- Exemples : un dictionnaire de la langue français, une pile.

Type abstrait de données : définition d'un nouveau type de données avec les opérations associées.

- Contraitement à un objet abstrait, on peut créer plusieurs objets de même type.
- Exemples : type complexe, type dictionnaire, type pile.

Data pool : implémentation typique

Juste un fichier .h avec des définitions de constantes (macro or `const`)

Exemple : `limits.h` (extrait)

```
/* Copyright (C) 1991-2018 Free Software Foundation, Inc.
   This file is part of the GNU C Library. */
...
/* Minimum and maximum values a `signed short int' can hold.  */
#  define SHRT_MIN        (-32768)
#  define SHRT_MAX        32767

/* Maximum value an `unsigned short int' can hold.  (Minimum is 0.)  */
#  define USHRT_MAX        65535

/* Minimum and maximum values a `signed int' can hold.  */
#  define INT_MIN        (-INT_MAX - 1)
#  define INT_MAX        2147483647
/* Maximum value an `unsigned int' can hold.  (Minimum is 0.)  */
#  define UINT_MAX        4294967295U

/* Minimum and maximum values a `signed long int' can hold.  */
#  if __WORDSIZE == 64
#   define LONG_MAX        9223372036854775807L
#  else
#   define LONG_MAX        2147483647L
#  endif
#  define LONG_MIN        (-LONG_MAX - 1L)
...
```

Librairie : implémentation typique

Un fichier d'entête contenant les prototypes des fonctions accessibles au client et un fichier source les implémentant.

Exemple : une librairie de fonctions mathématiques

`fctmath.h`

```
#ifndef _FCTMATH_H
#define _FCTMATH_H

...
double cos(double);
float cosf(float);

double sin(double);
float sinf(float);

double pow(double, double);
float powf(float, float);

double sqrt(double);
float sqrtf(float);
...

#endif
```

`fctmath.c`

```
#include "fctmath.h"

...
double cos(double x) {
     ...
};

float cosf( float x) {
     ...
};

double sin( double )  {
     ...
};

float sinf( float )  {
     ...
};
...
```

Objet abstrait : implémentation typique

Idem qu'une librairie mais les fonctions sont toutes relatives à une structure de données abstraite qui est définie concrètement dans le fichier source.

Exemple : un dictionnaire français

`dicofr.h`

```c
#ifndef _DICOFR_H
#define _DICOFR_H

int dicofr_init();
int dicofr_destroy();
int dicofr_search_word(char *word);
char *dicofr_get_definition(char *word);
int dicofr_add_word(char *word,char *definition);
...

#endif
```

`dicofr.c`

```c
#include "dicofr.h"

// définition de la structure interne au module
static struct {
  ...
} dicofr;

// définition des fonctions de l'interface

int dicofr_init() {
  ...
}

int dicofr_destroy() {
  ...
};

int dicofr_search_word(char *word) {
  ...
}
...
```

Type abstrait de données : implémentation typique

Le fichier d'entête définit le nouveau type et fournit le prototype des fonctions opérants sur les objets de ce type. Le fichier source définit concrètement les fonctions.

Exemple 1 : type de données complexe

`complex.h`

```c
#ifndef _COMPLEXE_H
#define _COMPLEXE_H

// définition du nouveau type

typedef struct {
  double re, im;
} complex;

// prototypes des fonctions

complex complex_new(double, double);
complex complex_sum(complex, complex);
complex complex_product(complex, complex);
double  complex_modulus(complex);
...

#endif
```

`complex.c`

```c
#include "complex.h"

// Implémentation des fonctions

complex complex_new(double re, double im) {
  ...
}
complex complex_sum(complex a, complex b) {
  ...
}
complex complex_product(complex a, complex b) {
  ...
}
double complex_modulus(complex c) {
  ...
}
...
```

Type abstrait de données : implémentation typique

Exemple 2 : type de données matrice

matrix.h

```c
#ifndef _MATRIX_H
#define _MATRIX_H

// définition du nouveau type

typedef struct {
  double **el;
  unsigned n, m;
} matrix;

// prototypes des fonctions

matrix *matrix_create(unsigned, unsigned);
void matrix_free(matrix *);
matrix *matrix_sum(matrix *, matrix *);
matrix *matrix_product(matrix *, matrix *);
double matrix_determinant(matrix *);
...

#endif
```

matrix.c

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

// Implémentation des fonctions
matrix *matrix_create(unsigned n, unsigned m) {
  ...
}

void matrix_free(matrix *m) {
  ...
}

matrix *matrix_sum(matrix *a, matrix *b) {
  ...
}
...
```

Remarque 1 : documentation

Il est important de documenter précisément toutes les fonctions de l'interface, dans le fichier d'entête et/ou via un document séparé.

Exemple :

```
/* ------------------------------------------------------------------ *
 * Creates a m by n matrix with all values set to 0. Returns NULL if
 * m <= 0 or n <= 0 and otherwise a pointer to the new matrix.
 *
 * ARGUMENTS
 * m           The number of rows
 * n           The number of columns
 *
 * RETURN
 * matrix      A pointer to the new matrix or NULL
 *
 * NOTE
 * The returned matrix should be cleaned with matrix_free after usage
 * ------------------------------------------------------------------*/

matrix *matrix_create(unsigned m, unsigned n)
```

Remarque 2 : conventions de nommage

Lorsqu'on utilise plusieurs modules, il est possible que des fonctions avec le même nom soient présentes dans différents modules si on n'y prête pas attention.

Particulièrement probable avec des types abstraits de données (exemple : `init`, `create`, `is_full`...).

Pour éviter les conflits de noms, il est utile de mettre le nom de module ou du type de données en préfixe des noms de fonctions du module.

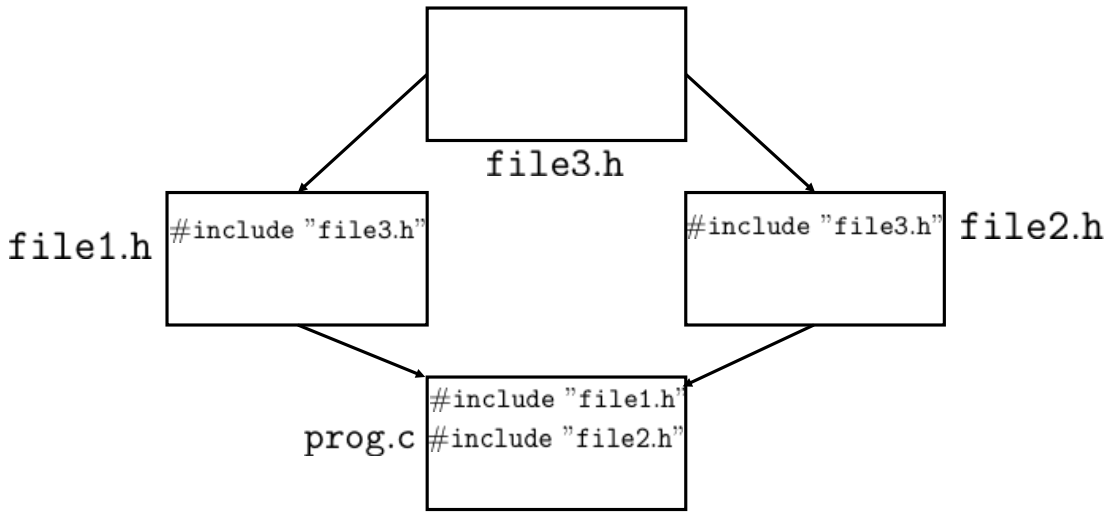
Exemples :

- `complex_new`, `complex_add`, `matrix_create`, `matrix_new`...
- `complexNew`, `complexAdd`, `matrixCreate`, `matrixNew`...

Inclusion gardée

Tout client d'un module doit inclure l'entête de ce module pour pouvoir accéder aux fonctions, variables et types qui y sont définis.

Un fichier d'entête peut inclure un autre fichier d'entête et donc un même fichier d'entête peut être compilé plusieurs fois.



Engendre des erreurs si un fichier d'entête inclus plusieurs fois contient :

- des définitions de types
- des déclarations de variables avec initialisation

Inclusion gardée

Deux solutions :

- S'arranger pour qu'un fichier d'entête ne soit inclus qu'une seul fois (compliqué)
- Utiliser la technique de l'inclusion gardée.

On entoure le fichier d'entête (`module.h`) des instructions suivantes :

```
#ifndef _MODULE_H  // N'inclut ce qui suit que si _MODULE_H n'est pas définie
#define _MODULE_H  // définit la constante _MODULE_H
...
// corps du fichier d'entête
...
#endif
```

Le nom de la constante n'a pas d'importance (`_MODULE_H`, `__MODULE__`...) pour autant qu'il soit le plus éloigné possible de noms communs de constantes (`H`, `N`...).

Pas toujours nécessaire mais autant le faire systématiquement.

Plan

Masquage de l'information

Un bon module garde secret les détails d'implémentation non utiles au client.

Deux avantages :

- Securité : le client ne peut pas corrompre les données. Il est obligé d'utiliser les fonctions de l'interface.
- Flexibilité : le programmeur peut modifier l'implémentation sans devoir en référer au client.

Deux mécanismes sont disponibles en C pour faire ça :

- Déclaration statique des variables globales et fonctions/procédures privées.
- Utilisation d'un type opaque.

Variables globales et fonctions statiques

Le mot-clé `static` permet de rendre les variables globales et les fonctions/procédures inaccessibles en dehors du code source où elles sont définies.

Doit être utilisé pour toutes les fonctions et variables internes au module non destinées à être utilisées par le client.

Exemples :

dicofr.h

```
#ifndef _DICOFR_H
#define _DICOFR_H

int dicofr_init();
...

#endif
```

dicofr.c

```
#include "dicofr.h"

// définition de la structure interne au module
static struct {
  ...
} dicofr;

// définition des fonctions de l'interface

static int load_data() {
  ...
}

static void terminate(char *message) {
  printf("%s\n", message);
  exit(EXIT_FAILURE);
}

int dicofr_init() {
  ...
  if (!load_data()) {
    terminate("error when loading data");
  }
  ...
}
...
```

Remarque : classe de stockage statique

Le mot-clé `static` a une signification tout à fait différente quand il est utilisé pour des variables locales.

```
int func(int n) {
  static int a;
  ...
}
```

Dans ce contexte, `static` indique que la variable est allouée de manière permanente (statique) et garde sa valeur d'un appel à l'autre de la fonction `func`.

Utilisation très spécifique, à éviter dans le contexte du cours.

Types opaques : motivation

Problème de l'implémentation précédente du type de données 'matrix' : la structure est visible dans l'entête et donc est accessible au client :

`matrix.h`

```
#ifndef _MATRIX_H
#define _MATRIX_H

// définition du nouveau type

typedef struct {
  double **el;
  unsigned n, m;
} matrix;
...
```

Le client est autorisé à utiliser `mat.m` ou `mat.n` directement s'il le désire et à créer une matrice sans passer par `matrix_new`.

Quid si on veut changer la structure ? Il faudra modifier tous les clients utilisant l'ancienne structure.

La solution est de rendre la structure opaque.

Types opaques : exemple 1

`matrix.h`

```c
#ifndef _MATRIX_H
#define _MATRIX_H

// définition du nouveau type

typedef struct matrix_t matrix;

// prototypes des fonctions

matrix *matrix_create(unsigned, unsigned);
void matrix_free(matrix *);
double matrix_getelement(matrix *,int,int);
void matrix_setelement(matrix *,int,int);
unsigned matrix_getnbrows(matrix *);
unsigned matrix_getnbcols(matrix *);
...

#endif
```

`matrix.c`

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

// Définition concrète du type
struct matrix_t {
  double **el;
  unsigned n, m;
};

// Implémentation des fonctions
matrix *matrix_create(unsigned n, unsigned m) {
  ...
}

void matrix_free(matrix *m) {
  ...
}
...
```

- 'typedef struct matrix_t matrix' définit le type matrix comme une structure `matrix_t` mais ne la définit pas.
- La définition concrète est déplacée dans le fichier source.
- Les clients, qui incluent `matrix.h`, n'y ont donc plus accès.

Types opaques

Une fois le type rendu opaque, on peut changer l'implémentation de la structure sans affecter les clients.

Restriction : le client n'ayant pas accès aux détails de la structure, il ne peut y accéder que par pointeur.

Exemple :

```
#include "matrix.h"
...
matrix m; // Impossible !
matrix *m; // Ok !
...
```

En conséquence, le code du type `complex` du slide 124 ne peut pas être rendu opaque. Il faut plutôt utiliser l'implémentation par pointeur du slide 47.

Types opaques : exemple 2

complex.h

```
typedef struct complex_t complex;

// constructeur
complex *complex_new(double, double);
// destructeur
void complex_destroy(complex *);
// accesseurs
double  complex_real_part(complex *);
double  complex_imgry_part(complex *);
void complex_set_real_part(complex *,double);
void complex_set_imgry_part(complex *,double);
// opérateurs
void complex_sum(complex *, complex *);
void complex_product(complex *, complex *);
double  complex_modulus(complex *);
...
```

complex.c

```
#include <math.h>
#include "complex.h"

struct complex_t {
  double re, im;
};

complex *complex_new(double re, double im) {
  complex *c = (complex *)malloc(sizeof(complex));
  c->re = re;
  c->im = im;
  return c;
}

double complex_real_part(complex *a) {
  return a->re;
}

void complex_set_real_part(complex *a, double re) {
  a->re = re;
  return a;
}
...
```

Plan

Généricité

Pour augmenter la réutilisabilité d'un module, on souhaite souvent le rendre le plus générique possible.

Exemple : module de tri (basé sur le tri par insertion)

`sort.h`

```
int sort(int array[], int length);
```

`sort.c`

```
#include "sort.h"

void sort(int array[], int length) {
  int i = 1;
  while (i < length) {
    int key = array[i];
    int j = i;
    while (j > 0 && array[j-1]>key) {
      array[j] = array[j-1];
      j--;
    }
    array[j] = key;
    i++;
  }
}
```

Inutilisable si on veut trier des tableaux d'autre chose que des entiers.

Généricité en C

Le C n'a pas de mécanisme simple pour rendre le code générique. Il faut un peu bricoler.

Trois techniques néanmoins :

- Utilisation de macros (ou `typedef`)
- Pointeur de fonctions
- Pointeur sur void

Généricité en C via des macros

On peut paramétriser le type du tableau trié en utilisant un `#define` (ou un `typedef`).

`sort.h`

```
#define SORTTYPE int
[Ou bien: typedef int SORTTYPE;]

SORTTYPE sort(SORTTYPE array[], int length);
```

`sort.c`

```
#include "sort.h"

void sort(SORTTYPE array[], int length) {
  int i = 1;
  while (i < length) {
    SORTTYPE key = array[i];
    int j = i;
    while (j > 0 && array[j-1]>key) {
      array[j] = array[j-1];
      j--;
    }
    array[j] = key;
    i++;
  }
}
```

Le client peut définir `SORTTYPE` au type désiré avant d'inclure `"sort.h"`.

On peut utiliser des macros pour faire des choses plus sophistiquées, au détriment de la lisibilité du code.

Pointeurs de fonctions

Soit l'implémentation suivante de la sécante (voir TP1)

```
double secant_method(double approx0, double approx1, double min_error) {
  double xcurrent, xp, xpp;

  xcurrent = approx1;
  xp = approx0;

  while (fabs(xcurrent-xp)) {
    xpp = xp;
    xp = xcurrent;
    xcurrent = ((xpp*f(xp))-((xp)*f(xpp)))/((f(xp)-f(xpp)));
  }

  return xcurrent;
}
```

Le mécanisme de passage de la fonction `f` dont on veut calculer la racine n'est pas satisfaisant.

Solution plus appropriée : passer la fonction `f` en argument. C'est possible en utilisant un pointeur de fonction.

Pointeurs de fonctions

Comme tout élément en C, une fonction dispose d'une adresse en mémoire. Son nom peut être utilisé pour dénoter cette adresse.

Il est possible de stocker ces adresses dans des variables et de les passer en arguments à des fonctions. Les variables et arguments sont alors de type pointeur de fonction.

La déclaration d'une pointeur de fonction se fait comme suit :

```
type (* id) ([type1[, type2] [,...]]]);
```

Note : les parenthèses autour du * sont nécessaires pour distinguer le pointeur de fonction d'un fonction renvoyant un pointeur.

Exemples :

- `int (*fonction)(int, int);`
- `void (*procedure)(float);`

Pointeurs de fonctions

Une version générique de la méthode de la sécante :

```
double secant_method(double (* f)(double), double approx0,
                     double approx1, double min_error) {
  ...
    xcurrent = ((xpp*f(xp))-((xp)*f(xpp)))/((f(xp)-f(xpp)));
  ...
}
```

L'appel à `f` peut aussi se faire via `(*f)(.)` ou directement via `f(.)`.

Au niveau du client :

```
double myfunction(double x) {
  return (pow(x,3)-18);
}

int main() {
  double root = secant_method(myfunction, -2.0, 2.0, 0.0005);
}
```

Pointeur sur void

Ce mécanisme n'est pas suffisant pour écrire une fonction de tri générique.

Il faut pour cela qu'on puisse manipuler des données dont le type n'est pas connu à l'avance.

Solution : pointeur sur `void` :

- On passe les données via un pointeur sur void qui peut pointer vers n'importe quoi et est déclaré comme suit :

```
void *p;
```

- Quand on veut manipuler réellement la donnée, on doit néanmoins préciser le type en utilisant une conversion de type. Par exemple :

```
(int *)p
```

- Idée : on demande au client de fournir les fonctions de manipulation des données en utilisant des pointeurs de fonctions.

Exemple d'algorithme de tri générique

Manipulation de données dans le contexte du tri : comparaison et échange de valeurs.

```
void sort(void *array, int length, int (*compare)(void*, int, int),
          void (*swap)(void *, int, int)) {
  int i = 1;
  while (i < length) {
    int j = i;
    while (j > 0 && !(compare(array, j-1,j))) {
      swap(array,j-1,j);
      j--;
    }
    i++;
  }
}
```

- `swap(array, i, j)` échange les éléments aux positions i et j dans array.
- `compare(array, i, j)` vaut 1 si l'élément i est avant l'élément j en terme d'ordre, 0 sinon.

Tri générique : application 1

Implémentation de `compare` et `swap` pour le tri d'un tableau d'entiers :

```
void swap_int(void *array, int i, int j) {
  int temp = ((int*)array)[i];
  ((int*)array)[i] = ((int*)array)[j];
  ((int*)array)[j] = temp;
}

int compare_int(void *array, int i, int j) {
  return (((int*)array)[i] <= ((int*)array)[j]);
}
```

Utilisation au niveau du client :

```
int A[5]={5, 4, 3, 2, 1};
sort(A, 5, compare_int, swap_int);
```

Tri générique : application 2

Implémentation de `compare` et `swap` pour trier un tableau de complexes selon leurs modules :

```
void swap_complex(void *array, int i, int j) {
  complex temp = ((complex*)array)[i];
  ((complex*)array)[i] = ((complex*)array)[j];
  ((complex*)array)[j] = temp;
}

int compare_complex_mod(void *array, int i, int j) {
  return (complex_modulus(((complex*)array)[i]) <= complex_modulus(((complex*)array)[j]));
}
```

Utilisation au niveau du client :

```
complex C[5]={{2,5}, {3,4}, {0,3}, {1,2}, {1,0}};
sort(C, 5, compare_complex_mod, swap_complex);
```

La programmation modulaire en C

Les mécanismes de programmation modulaire en C, tels qu'exposés, sont fonctionnels mais relativement rudimentaires.

Dans d'autres langages de programmation, la modularité est gérée de manière plus naturelle, même si les concepts restent les mêmes.

Par exemple :

- En programmation orientée objet (C++, Java, etc.), il existe de nombreux mécanismes pour générer le masquage de l'information, la définition de nouveau type abstrait et la généricité.
- En programmation fonctionnelle (Scheme, Lisp, Haskell), les fonctions sont des valeurs comme les autres qui peuvent être passées comme arguments sans passer par un mécanisme de pointeurs.

Plan

Compilation d'un seul fichier C

Si tout le code source tient dans un seul fichier (`nom_programme.c`), on peut le compiler via la commande suivante :

```
gcc -o nom_executable nom_programme.c
```

Si le code est séparé en différents modules, on peut les compiler en une seule ligne :

```
gcc -o nom_executable nom_programme.c module1.c module2.c
```

Cette commande compile en fait séparément les différents fichiers avant de les lier pour créer l'exécutable.

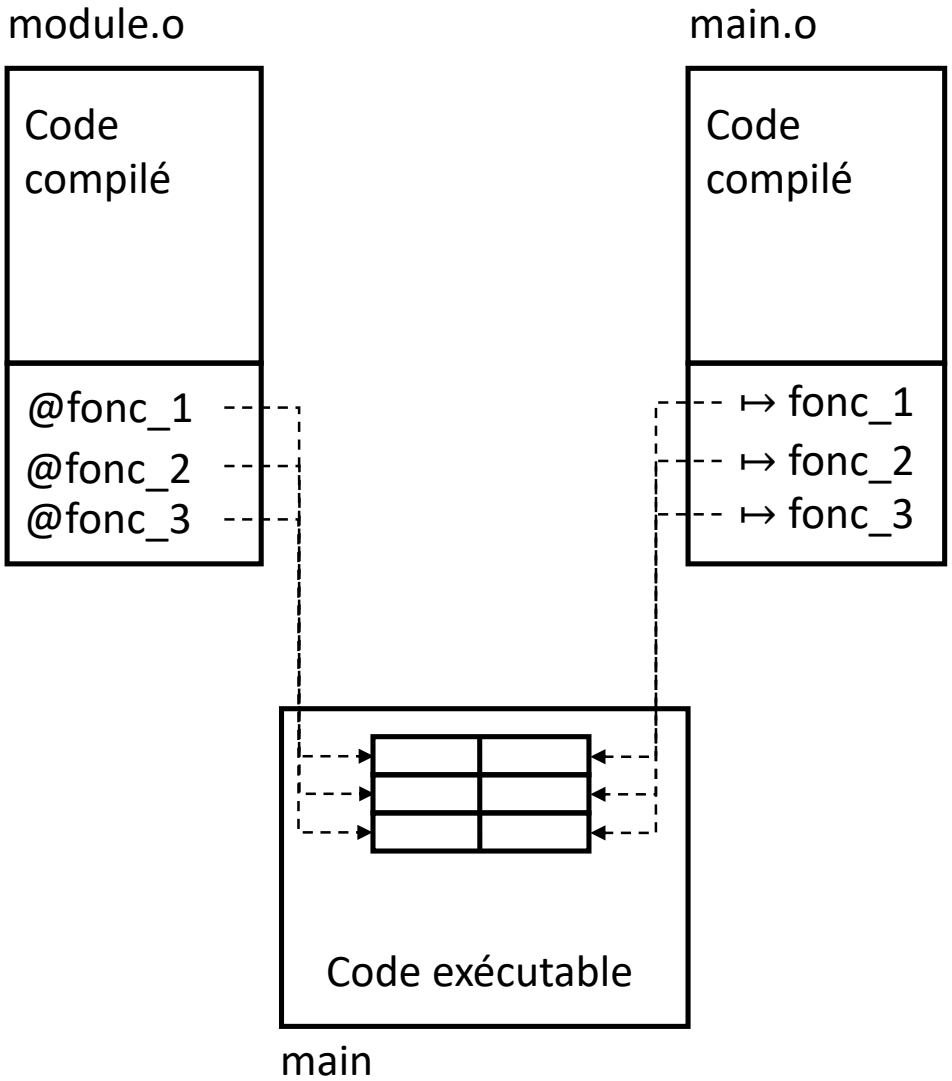
Les différentes étapes de la compilation

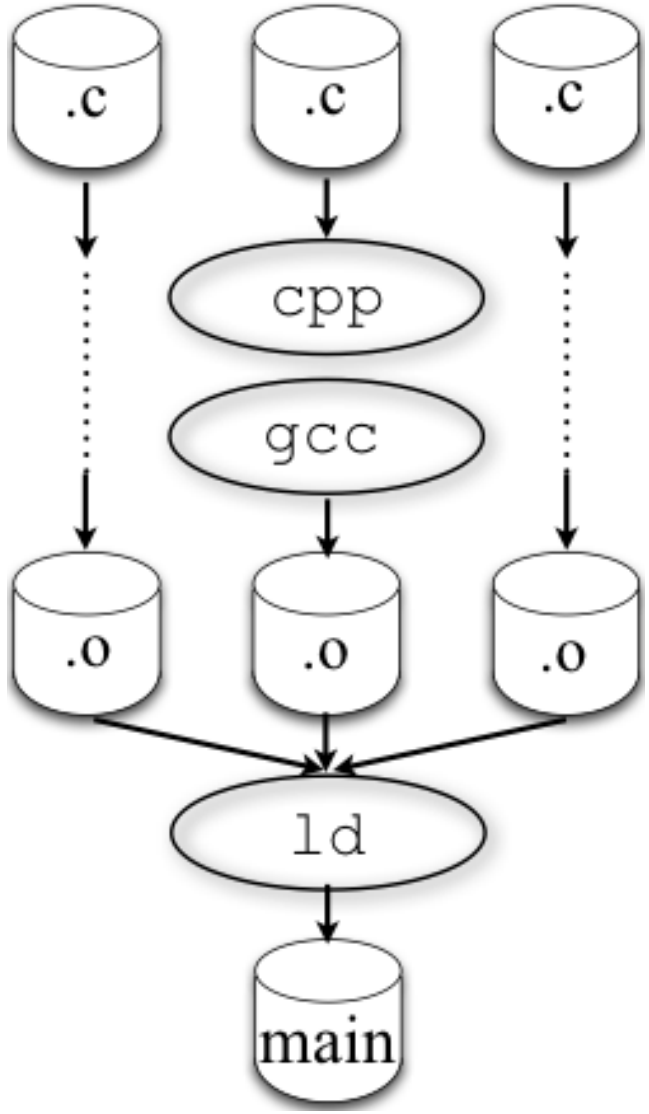
Pour chaque fichier source séparément, on passe par les étapes suivantes :

- Pré-traitement : éliminations des commentaires, remplacement des macros, inclusion des sous-fichiers
- Génération d'un fichier objet (`.o`) contenant :
  - le code compilé en langage d'assemblage
  - La table des liens (les variables/fonctions exportées ou importées par le module)

Ensuite, l'exécutable est obtenu à partir des fichiers objets en utilisant l'éditeur de liens (`ld`). Ce dernier assemble les codes compilés et fait les liens entre les appels de fonctions et de variables.

Les différentes étapes de la compilation





Les différentes étapes de la compilation

Pour faire les étapes manuellement :

```
>  gcc -c nom_programme.c
>  gcc -c module1.c
>  gcc -c module2.c
>  gcc -o nom_executable nom_programme.o module1.o module2.o
```

Avec l'approche manuelle, on ne doit pas recompiler la totalité des modules, seulement ceux qui ont été modifiés.

Par contre, il peut être très fastidieux de gérer les recompilations à la main.

Deux solutions :

- Utiliser un IDE (environnement de développement intégré. Par exemple CodeBlocks)
- Utiliser l'outil unix Make en ligne de commande

Make

Make est un utilitaire permettant de gérer la compilation d'un programme réparti en plusieurs fichiers (pas nécessairement en C).

Make est un outil très puissant dont on va juste voir ici le strict minimum.

Principe :

- On place un fichier appelé `Makefile` (ou `makefile`) dans le répertoire où se trouvent les sources.
- On lance la compilation en faisant :

  ```
  > make [cible]
  ```

  où `cible` est défini dans le fichier `Makefile`.

Fichier `Makefile`

Le fichier est constitué d'une suite de règles :

```
cible: dépendances
[tabulation] actions
```

où :

- `cible` est un nom de fichier ou un simple label
- `dépendances` est une liste de fichiers dont dépend la cible
- `actions` est une liste d'actions à effectuer

Suite à un appel à `make cible`, les actions ne sont effectuées que si la date du fichier cible est moins récente qu'au moins l'un des fichiers de dépendances.

Si les fichiers de dépendances apparaissent comme cibles dans une règle, ils sont mis à jour récursivement selon le même principe avant de gérer la cible courante.

Illustration : version 1

Exemple de `Makefile` dans le cas d'un programme `prog.c` basé sur deux modules, `module1.c` et `module2.c`.

```
1  main: prog.o module1.o module2.o
2      gcc -o main prog.o module1.o module2.o
3  prog.o: prog.c module1.h module2.h
4      gcc -c prog.c
5  module1.o: module1.c module1.h
6      gcc -c module1.c
7  module2.o: module2.c module2.h
8      gcc -c module2.c
```

On construit le programme en faisant `make main` ou plus simplement `make` (qui utilise la première cible du fichier par défaut).

Remarque : on peut obtenir toutes les dépendances dans un répertoire (les lignes 3, 5, et 7) en faisant 'gcc `-MM *.c`'.

Illustration : version 2

- `Make` sait comment obtenir un fichier `o` à partir d'un fichier `c`. On peut enlever les lignes 4, 6, et 8.
- Il faut néanmoins lui dire quel compilateur utiliser via la définition d'une variable `CC` au début du fichier (ligne 1 ci-dessous).
- On peut fournir les flags de compilation via une variable `CFLAGS`.

```
1  CC = gcc
2  CFLAGS = -Wall -Wextra -Wmissing-prototypes --pedantic\
3           -std=c99
4  main: prog.o module1.o module2.o
5      gcc -o main prog.o module1.o module2.o
6  prog.o: prog.c module1.h module2.h
7  module1.o: module1.c module1.h
8  module2.o: module2.c module2.h
```

Remarque sur les flags de compilation

Pour ce cours (devoirs et projets), on utilise les flags suivants ;

- `--std=c99` : spécifie la norme C99
- `-pedantic` : application stricte de la norme C99
- `-Wall` : affiche (presque) tous les warnings
- `-Wextra` : affiche d'autres warnings
- `-Wmissing-prototypes` : affiche un warning pour les prototypes non définis

Une liste complète des warnings générés par les options `-Wall` et `-Wextra` peut être obtenue ici :

https://gcc.gnu.org/onlinedocs/gcc-6.1.0/gcc/Warning-Options.html

Illustration : version 3

On peut encore simplifier le fichier en utilisant des variables, substituées dans les règles via `$(VAR)`, où `VAR` est le nom de la variable.

```
1  OFILES = prog.o module1.o module2.o
2  TARGET = main
3  CC = gcc
4  CFLAGS = -Wall -Wextra -Wmissing-prototypes --pedantic\
5           -std=c99
6  $(TARGET): $(OFILES)
7      $(CC) $(OFILES) -o $(TARGET)
8  prog.o: prog.c module1.h module2.h
9  module1.o: module1.c module1.h
10 module2.o: module2.c module2.h
```

Les seules lignes à écrire sont les lignes 1 et 2. Les lignes 3-7 sont génériques et les lignes 8-10 sont obtenues directement via '`gcc -MM *.c`'.

Illustration : version finale

- Il peut être utile d'ajouter des règles non liées à des fichiers.
- Par ex. pour supprimer les fichiers compilés (`clean`), exécuter le programme (`run`) et créer une archive avec le code (`archive`)

```
1  OFILES = prog.o module1.o module2.o
2  TARGET = main
3  CC = gcc
4  CFLAGS = -Wall -Wextra -Wmissing-prototypes --pedantic\
5           -std=c99
6  .PHONY: all clean run archive
7
8  all: $(TARGET)
9  clean:
10     rm -f $(OFILES) $(TARGET)
11 run: $(TARGET)
12     ./$(TARGET)
13 archive:
14     tar cvfz illustration_makefile.tar.gz *.h *.c Makefile README
15 $(TARGET): $(OFILES)
16     $(CC) $(OFILES) -o $(TARGET)
17 prog.o: prog.c module1.h module2.h
18 module1.o: module1.c module1.h
19 module2.o: module2.c module2.h
```

- La ligne 6 (`.PHONY: ...`) précise que les cibles qui suivent ne correspondent pas à des fichiers.
- La cible `all` en première position est celle qui est exécutée par défaut.

Synthèse

`make` est un outil très puissant et très complexe. Des livres entiers lui sont consacrés.

C'est l'outil de base pour faciliter la distribution de code source sous environnements de type Unix/Linux.

Beaucoup plus portable que de distribuer des projets produits par un IDE.

Pour plus d'information :
`https://www.gnu.org/software/make/manual/make.html`.

Plan

Tests

Une fois qu'un programme compile, il faut le tester, c'est-à-dire vérifier que son comportement est conforme au comportement attendu (test de conformité)

Un des avantages de la programmation modulaire est qu'on peut tester chaque module séparément.

Vérifier le fonctionnement de portions de code (p.ex, une fonction, un module) indépendamment des programmes qui les utilisent est ce qu'on appelle un test unitaire.

Il existe aussi des tests d'intégration, qui vérifient les interactions entre modules, et des tests systèmes, qui vérifient le comportement d'un système complet.

Tests unitaires en pratique

Pour chaque module, on crée un fichier source `module_main_test.c` avec une fonction `main` chargée de réaliser les tests.

Ces tests doivent :

- mettre en œuvre l'ensemble des fonctionnalités décrites dans les spécifications du module
- explorer le fonctionnement du module dans des conditions non-spécifiées

Établir ces tests peut constituer un défi en soi.

- Par exemple, comment tester la validité d'un générateur de nombres aléatoires, une fonction mathématique, une générateur d'images ?

Une bonne pratique est d'écrire les tests avant d'implémenter le module.

Exemple 1

module.h

```
...
int abs(int a);
...
```

module.c

```
#include "module.h"
...
int abs(int a) {
  if (a < 0) return -a;
  return a;
}
...
```

test_main_module.c

```
#include "module.c"
#include <assert.h>

int main() {
  ...
  // tests de la fonction abs
  int x = -3;
  int y = abs(x);
  assert (x == y || -x == y);
  assert (y >= 0);
  x = abs(y);
  assert (y == x);
  ...
  return 0;
}
```

La macro assert

```
void assert(scalar expression)
```

`assert` est une macro définie dans `assert.h`.

Son unique argument est une assertion, c'est-à-dire une expression qu'on suppose être vraie à un moment de l'exécution du programme.

Fonctionnement :

- Si l'expression a une valeur non nulle, `assert` ne fait rien.
- Si l'expression a une valeur nulle, un message est affiché et l'exécution du programme est arrêtée.

Exemple de message :

```
a.out: module.c:9: main: Assertion `y >= 0' failed.
```

Exemple 2

module.h

```
...
int swap(int *x, int *y);
...
```

module.c

```
#include "module.h"
...
int swap(int * x, int * y) {
  if (x == NULL || y == NULL)
     return -1;
  int t = *x;
  *x = *y;
  *y = t;
  return 0;
}
...
```

test_main_module.c

```
#include "module.c"
#include <assert.h>

int main() {
  ...
  // tests de la fonction swap
  int a = 0, b = 1;
  int rv = swap(&a, &b);
  assert (rv == 0);
  assert (a == 1);
  assert (b == 0);
  rv = swap(&a, NULL);
  assert (rv != 0);
  assert (a == 1);
  ...
  return 0;
}
```

Tests unitaires

L'approche informelle manuelle précédente est suffisante dans le cadre de ce cours.

C'est celle que nous utiliserons pour tester vos codes.

Il existe cependant de nombreux outils pour construire ces tests de façon plus systématique. Par exemple : CUnit, cmocka, Seatest...

Débogage

Une fois qu'un test a mis en évidence un problème dans une fonction d'un module, il faut isoler et corriger l'erreur (bogue) dans le programme. C'est ce qu'on appelle le débogage.

Activité assez compliquée en général : le bogue peut être très éloigné du symptôme.

Une programmeur passe en général plus de temps à déboguer du code existant qu'à écrire du nouveau code.

Bogues fréquents en C

- Mauvais cast (ou cast implicite) provoquant une perte de précision
  - `(float)(x/y)` au lieu de `((float) x)/((float y))`
- Accès en dehors des tableaux : en particulier le `'\0'` des chaînes de caractères.
- Variable/mémoire non initialisée
- Confusion entre = et ==
- Ne pas traiter le code de retour d'une fonction pouvant indiquer une erreur
- Boucle infinie

Prévenir les bogues

Pour prévenir les bogues (ou faciliter le débogage) :

- Raisonnez sur papier avant de vous lancer dans l'implémentation (pensez aux invariants)
- Documentez votre code (surtout lorsque vous travaillez à plusieurs)
- Privilégiez toujours la lisibilité et la clarté du code à sa compacité.
  Un mauvais exemple :

```
void my_strcopy(char dest[], char src[]) {
   while ((*dst++ = *src++));
}
```

- Evitez les effets de bord (variables globales, variables locales statiques...)
- Activez (et supprimez) tous les warnings de compilation
- Adoptez une programmation défensive (voir plus loin).

Techniques de débogage

Lecture du code

- En général, inefficace si l'auteur est le lecteur

Exécution instrumentée :

- Ajouts d'assertions
- Ajouts de `printf`

Exécution contrôlée

- Utilisation d'un débogueur, permettant d'exécuter le code pas-à-pas, de mettre des points d'arrêts, de consulter les variables en temps réel...
- Par exemple, `gdb` ou votre IDE préféré (p. ex., CodeBlocks).

Utilisation de `printf` pour le débogage

Approche très simple et facile à mettre en œuvre mais peut être longue et fastidieuse.

Un classique : afficher les indices d'accès à un tableau pour détecter les débordements.

Remarque :

- La commande `printf` est bufferisée : l'appel n'affiche pas tout de suite le résultat à l'écran.
- Le bug pourrait donc suivre un printf avorté plutôt que le précéder.
- Préférez l'instruction `fprintf(stderr,...)` plutôt que `printf(...)`, qui n'est pas bufferisée.

Utilisation de `printf` pour le débogage

Des macros permettent d'afficher de l'information utile lors du débogage :

- `__LINE__` : le numéro de ligne,
- `__FILE__` : le nom de fichier,
- `__FUNCTION__` : le nom de la fonction.

Macro générale d'affichage d'un message de debogage :

```
#define DEBUG(message, indice) \
      fprintf(stderr, "Error (%s%d): ligne %d, fonction \
      [%s], fichier [%s]\n", message, indice, __LINE__,\
        __FUNCTION__,__FILE__)
```

Utilisation :

```
DEBUG("indice i=",i);
```

Programmation défensive

La programmation défensive consiste à penser à tous les cas possibles de mauvaises utilisations de ses fonctions, à ajouter des tests pour détecter ces situations, et à rapporter les erreurs correspondantes.

Elle consiste essentiellement à vérifier la pré-condition des algorithmes mais on peut également tester des invariants.

Elle permet de détecter les symptômes d'un bug au plus tôt.

Deux solutions principales en cas d'erreur :

- Afficher un message et arrêter le programme.
- Prévoir un résultat spécial de la fonction (cfr. le return de la fonction `main()`)

Programmation défensive : un exemple (Bradley, 1998)

```
int reverse(char *in, char *out) {
  if (!in || !out) return -1;

  int i;
  for (i = 0; in[i]; i++);

  assert(in[i] = '\0');

  for (i--; i >= 0; i--) {
    assert(in[i] != '\0');
    *out = in[i];
    out++;
  }
  *out = '\0';
  return 0;
}
```

- La fonction renvoie 0 si tout s'est bien passé, -1 sinon.
- Les `assert` permettent de détecter un problème au niveau de l'algorithme.
- Ce code reste cependant vulnérable à des argument mal formés *(pourquoi ?)*.

Programmation défensive : un exemple (Bradley, 1998)

Une version plus robuste, en ajoutant comme argument la longueur maximale de la chaîne pointée par `in` :

```
int nreverse(char *in, char *out, int maxLength) {
  if (!in || !out) return -1;

  int i;
  for (i = 0; in[i]; i++) {
    if (i > maxLength) return -2:
  }

  assert(in[i] = '\0');

  for (i--; i >= 0; i--) {
    assert(in[i] != '\0');
    *out = in[i];
    out++;
  }
  *out = '\0';
  return 0;
}
```

Programmation défensive : remarques

Faire des tests systématiques des arguments a un coût en terme de temps de calcul.

Généralement seulement utile pour les fonctions de l'interface d'un module auxquels ont accès les utilisateurs. Pas nécessaire pour les fonctions statiques du module.

On réservera l'utilisation de `assert` à des tests du bon fonctionnement de l'algorithme, pas aux tests de vérification des arguments.

La macro `assert` peut être désactivée en faisant.

```
#define NDEBUG
```

Permet de maintenir deux versions du code compilé, une version pour le débogage et une version de production.

Plan

Euh ?

```c
#include "stdio.h"
#define e 3
#define g (e/e)
#define h ((g+e)/2)
#define f (e-g-h)
#define j (e*e-g)
#define k (j-h)
#define l(x) tab2[x]/h
#define m(n,a) ((n&(a))==(a))

long tab1[]={ 989L,5L,26L,0L,88319L,123L,0L,9367L };
int tab2[]={ 4,6,10,14,22,26,34,38,46,58,62,74,82,86 };

main(m1,s) char *s; {
    int a,b,c,d,o[k],n=(int)s;
    if(m1==1){ char b[2*j+f-g]; main(l(h+e)+h+e,b); printf(b); }
    else switch(m1-=h){
    case f:
        a=(b=(c=(d=g)<<g)<<g)<<g;
        return(m(n,a|c)|m(n,b)|m(n,a|d)|m(n,c|d));
    case h:
        for(a=f;a<j;++a)if(tab1[a]&&!(tab1[a]%((long)l(n))))return(a);
    case g:
        if(n<h)return(g);
        if(n<j){n-=g;c='D';o[f]=h;o[g]=f;}
        else{c='\r'-'\b';n-=j-g;o[f]=o[g]=g;}
        if((b=n)>=e)for(b=g<<g;b<n;++b)o[b]=o[b-h]+o[b-g]+c;
        return(o[b-g]%n+k-h);
    default:
        if(m1-=e) main(m1-g+e+h,s+g); else *(s+g)=f;
        for(*s=a=f;a<e;) *s=(*s<<e)|main(h+a++,(char *)m1);
    }
}
```

Style

- Le style de programmation est un ensemble de lignes directrices utilisées lors de l'écriture d'un programme informatique.
- Inclut principalement des questions liées à l'aspect visuel du code mais pas uniquement.
- Suivre un (bon) style de programmation permet de rendre le code source plus lisible par soi-même et par d'autres (y compris les correcteurs) et permet d'éviter les erreurs.
- Important dans la mesure où un programme est souvent développé par plusieurs auteurs et où une grande partie de la vie d'un programme est consacrée à sa maintenance.
- La qualité d'un style est assez difficile à apprécier et subjective.
- Pour ce cours, pas de règles strictes, plutôt une série de conseils.

Identifiants

- Utilisés pour la dénomination de variables, fonctions, types, et constantes.
- Suites de lettres, chiffres et de '_'. Doivent commencer par une lettre ou '_'. La casse (majuscule/minuscule) compte.
- Dénominations en anglais (de préférence) ou en français, pour autant que vous soyez cohérent.
- Différents styles :
  - noms composés séparés par des '_' : `add_interest`, `number_of_days`... ("old school")
  - noms composés séparés par des majuscule : `addInterest`, `numberOfDays`... ("camel case").

Identifiants

En général :

- Ne pas utiliser d'abbréviations :
  - `firstName`, `lastName` au lieu de `fname`, `lname`
- Pas de noms trop longs
  - `setField` au lieu de `setTheLengthField`
- Eviter les dénominations trop proches
  - `thisPerishableProduct` au lieu de `perishableProduct` si `perishableProducts` existe.
- Utiliser des noms informatifs pour que le code soit auto-documenté.

```
double tax1;          // sales tax rate
double tax2;          // income tax rate

double salesTaxRate;
double incomeTaxRate;
```

Identifiants

Convention courante (mais non obligatoire) en fonction de la nature de l'identifiant :

- Variables et fonctions commencent par une minuscule
  - `myVar`, `my_var`, `myFunction`, `my_function`
- Types commencent par une majuscule
  - `MyType`, `My_type`
- Constantes en majuscule et utilisation de '_'
  - `MY_CONST`

Identifiants : fonctions

- Utiliser des verbes d'action.
  - `addInterest`, `convertToAustralianDollars`
- Préfixe `get` et `set` pour obtenir et donner une valeur à une variable.
  - `getBalance`, `setBalance` pour des opérations sur la variable `balance`.
- Préfixe `is` et `has` pour des fonctions retournant un booléen.
  - `isOverdrawn`, `hasCreditLeft`

Formatage du code

- Concerne l'indentation, les alignements, l'utilisation des espaces...
- Affecte uniquement la lisibilité du code.
- Une fois un style choisi, le conserver.
- Il existe des outils de formatage automatique
  - Par exemple, `indent` sous unix/linux.

Formatage : indentation et blocs

- Indentation :
  - Ajouter un nombre fixe d'espaces (2 ou 4 par exemple) à chaque rentrée dans un nouveau bloc
  - Éviter l'utilisation de tabulations (qui dépendent du système)
  - Souvent gérée automatiquement par votre éditeur de texte

- Sous-blocs :
  - Découper le code en sous-blocs en laissant une ligne vide entre ces blocs (composés de quelques lignes)
  - Laisser deux lignes vides entre chaque fonction

Formatage : espaces

`for(int i=0;i<n;i++)` vs. `for (int i = 0; i < n; i++)`

Mettre des espaces :

- Avant et après les opérateurs binaires, arithmétiques, et logiques
  `b = 1 + 2;`
  sauf éventuellement pour mettre en évidence la précédence :
  `a*x + b`
- Après les virgules et les points-virgules
  `myfunction(3, 4, 5)`
- Après les mots-clés réservés (`for`, `if`, `else`, `do`...)
- Après le signe de début de commentaires inline :
  `// this is a comment`
- Pour aligner du code si ça améliore la lisibilité

```
int n      = atoi(argv[1]);      //  size of population
int trials = atoi(argv[2]);      //  number of trials
```

Formatage : espaces

Ne pas en mettre :

- Entre les opérateurs unaires et leur variable
  `*p++ = !a;`
- Autour des parenthèses
  `myfunction(5 * (4+5))`
- Autour des opérateurs de sélection
  `member.data`, `node->next`, `vec[i]`...
- Avant la ponctuation
  `myfunction(3, 4, 5)`

Formatage : accolades

- Deux écoles (parmi d'autres) :

```
int add(int a, int b) {
   int result;
   if (a) {
      result = a + b;
      return result;
   } else {
      return result;
   }
}
```

```
int add(int a, int b)
{
   int result;
   if (a)
   {
      result = a + b;
      return result;
   }
   else
   {
      return result;
   }
}
```

- Les deux sont valides. La première est utilisée dans ces slides pour gagner de la place.

Formatage : les accolades

Les accolades ne sont pas obligatoires lorsque le bloc ne contient qu'une instruction :

```
if (currentHour < AFTERNOON) {
  printf("Morning\n");
} else if (currentHour < EVENING) {
  printf("Afternoon\n");
} else {
  printf("Evening\n");
}
```

```
if (currentHour < AFTERNOON)
  printf("Morning\n");
else if (currentHour < EVENING)
  printf("Afternoon\n");
else
  printf("Evening\n");
```

Attention cependant aux cas ambigüs :

```
if (b1)
  if (b2)
     printf("here");
else
  printf("there");
```

$\Leftrightarrow$

```
if (b1) {
  if (b2)
     printf("here");
  else
     printf("there");
}
```

Documentation et commentaires

Différents types de commentaires :

- Commentaires d'entête : précise l'auteur du code, la date de création/modification, description succincte du contenu du module, copyright, etc.
- Commentaires de documentation : définit le contrat de chaque fonction/procédure de l'interface du module.
- Commentaires de bloc : résume l'action d'une partie de code
- Commentaires *inline* : précise l'intérêt d'une ligne particulière de code

Documentation

```
/* ---------------------------------------------------------------------- *
 * Creates a m by n matrix with all values set to 0. Returns NULL if
 * m <= 0 or n <= 0 and otherwise a pointer to the new matrix.
 *
 * ARGUMENTS
 * m           The number of rows
 * n           The number of columns
 *
 * RETURN
 * matrix      A pointer to the new matrix or NULL
 *
 * NOTE
 * The returned matrix should be cleaned with matrix_free after usage
 * ----------------------------------------------------------------------*/

matrix *matrix_create(unsigned m, unsigned n)
```

Définition non ambigüe des fonctions de l'interface à l'adresse du client.

Un autre programmeur doit être capable de réimplémenter la fonction uniquement sur base des commentaires.

Il existe des outils permettant de générer automatiquement la documentation d'un code source à partir des commentaires et du code.

- Par exemple, doxygen.

Commentaires de bloc et inline

Les commentaires de bloc ou inline doivent décrire principalement le pourquoi d'une portion de code, le comment étant expliqué par le code lui-même.

Ni trop, ni trop peu :

- les réserver aux parties non triviales.
  Un mauvais exemple :

```
i++          // increment i by one
```

- Privilégier l'auto-documentation en utilisant des noms de fonctions et de variables informatifs
- S'il y a besoin d'en mettre trop, c'est probablement que le code manque de clarté. Il vaut mieux alors le réécrire.

Remarque : commentaires imbriqués

En C, on ne peut pas imbriquer des commentaires délimités par `/*...*/`.

Le code suivant génèrera une erreur (pourquoi ?) :

```
/* /* coucou */ */
```

Par contre, cette construction est possible :

```
/*
// coucou
*/
```

Que vaut la variable nest ?

```
int nest = /*/*/0*/**/1;
```

Quelques bonnes pratiques en vrac

- Déclarer des constantes au lieu d'utiliser des nombres/chiffres dans le code :
  Non pas :

```
day = (3 + numberOfDays) % 7;
```

  Mais plutôt :

```
const int WEDNESDAY = 3;
const int DAYS_IN_WEEK = 7;
day = (WEDNESDAY + numberOfDays) % DAYS_IN_WEEK;
```

- Créer des fonctions courtes
  - Si elle ne s'affiche pas sur un écran, il est probablement possible de la découper en plusieurs autres méthodes
- Se limiter à maximum trois niveaux de boucles imbriquées
- Pas plus de 80 caractères par ligne.
- Si nécessaire, aller à la ligne après une virgule ou avant un opérateur.

Quelques bonnes pratiques en vrac

- Éviter les écritures ambiguës même si elles sont autorisées par le langage

```c
int b1 = 1;
int b2 = 2;
if (b1 == b2)
   printf("%d\n", b1);
```

```c
int b1 = 1;
int b2 = 2;
if (b1 = b2)
   printf("%d\n", b1);
```

- Ne pas déclarer les variables sur la même ligne

```c
int a, b, c;
```

$\Rightarrow$

```c
int a;
int b;
int c;
```

- Déclarer les variables juste avant leur utilisation et initialiser les en même temps que leur déclaration.

Quelques bonnes pratiques en vrac

- Ne pas réutiliser une même variable dans des situations différentes
  - Si elle est judicieusement nommée, cela ne devrait pas vous effleurer l'esprit
- Éviter les répétitions de code. Mieux vaut créer une fonction que copier dix fois la même suite d'instructions.
- S'intéresser au langage
  - Utiliser les idiomes appropriés
    - `for` au lieu de `while` si gardien pas trop compliqué, `switch` au lieu de `if` imbriqués, `++`, `+=`...
  - Fouiller pour vérifier l'existence d'une fonction plutôt que de réinventer la roue

Sources

Programmation modulaire :

- C programming : a modern approach, K.N. King, W. W. Norton & Company, Second edition, 2008.
- Slides du cours INFO0030, Benoît Donnet

Style :

- A guide to coding style, Justus Piater, 2005.
- Notes du cours Algorithmique I, Renaud Dumont, 2009-2010.