CHARLES CARNUS
LYCÉE TECHNIQUE
ENSEIGNEMENT SUPÉRIEUR

C++, C

Compilation

Prof : **Kamal Boudjelaba**

10 novembre 2022

BTS SN

LTP Charles Carnus
Avenue de Saint Pierre - Bourran - 12000 RODEZ, France

IR–EC

Table des matières

Liste des figures

Liste des tableaux

1. La fonction `main`

La fonction principale `main` est une fonction comme les autres. On peut la considérer de type `void`, ce qui est toléré par le compilateur. Toutefois l'écriture `main()` provoque un message d'avertissement :
`warning: type specifier missing, defaults to 'int'`

En fait, la fonction main est de type `int`. Elle doit retourner un entier dont la valeur est transmise à l'environnement d'exécution. Cet entier indique si le programme s'est ou non déroulé sans erreur.

— La valeur de retour 0 correspond à une terminaison correcte,

— toute valeur de retour non nulle correspond à une terminaison sur une erreur.

La fonction main peut également posséder des paramètres formels. En effet, un programme C ou C++ peut recevoir une liste d'arguments au lancement de son exécution. La ligne de commande qui sert à lancer le programme est, dans ce cas, composée du nom du fichier exécutable suivi par des paramètres. La fonction main reçoit tous ces éléments de la part de l'interpréteur de commandes.

La fonction main possède deux paramètres formels, appelés par convention **`argc`** (argument count) et **`argv`** (argument vector (value)).

- **`argc`** est une variable de type `int` dont la valeur est égale au nombre de mots composant la ligne de commande (y compris le nom de l'exécutable). Elle est donc égale au nombre de paramètres effectifs de la fonction +1.
- **`argv`** est un tableau de chaînes de caractères correspondant chacune à un mot de la ligne de commande. Le premier élément argv[0] contient donc le nom de la commande (du fichier exécutable), le second **`argv[1]`** contient le premier paramètre ...

Le second prototype valide de la fonction **`main`** est donc : `int main (int argc, char *argv[])`

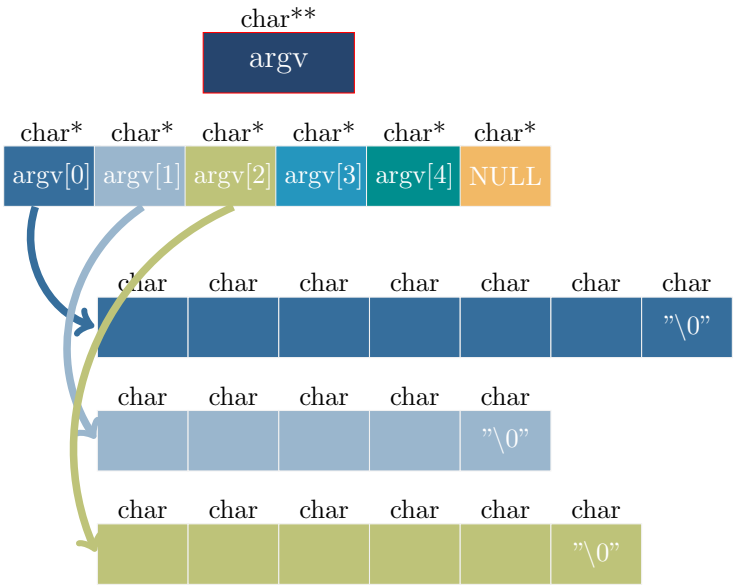

Figure 1. Les arguments de la fonction `main`

Exemple 1.1

Le programme suivant calcule le produit de deux entiers, entrés en arguments de l'exécutable :

```c
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
    int a, b;
    if (argc != 3)
    {
        printf("\nErreur : nombre invalide d'arguments");
        printf("\nUsage: %s int int\n",argv[0]);
        return(EXIT_FAILURE);
    }
    a = atoi(argv[1]);
    b = atoi(argv[2]);
    printf("\nLe produit de %d par %d vaut : %d\n", a, b, a * b);
    return(EXIT_SUCCESS);
}
```

Dans le terminal, taper (après compilation) : `./NomProExe 15 10`
La console affiche : `La somme de 15 et 10 vaut : 25`

Exemple 1.2

Cet exemple affiche les arguments de la fonction `main`

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

int main(int argc, char *argv[])
{
int a, b;
  if (argc != 3)
    {
      printf("\nErreur : nombre invalide d'arguments");
      printf("\nUsage: %s int int\n",argv[0]);
      return(EXIT_FAILURE);
    }
  a = atoi(argv[1]);
  b = atoi(argv[2]);
  printf("\nLa somme de %d et %d vaut : %d\n", a, b, a + b);
  int i;
    for (i=0; i < argc; i++)
        printf("Argument %d : %s\n", i+1, argv[i]);
  return(EXIT_SUCCESS);
}
```

La console affiche : (après la commande `./NomProExe 15 10`)

```
La somme de 15 et 10 vaut : 25
Argument 1 : ./NomProExe
Argument 2 : 15
Argument 3 : 10
```

2. Les directives au préprocesseur

Le préprocesseur est un programme exécuté lors de la première phase de la compilation. Il effectue des modifications textuelles sur le fichier source à partir de directives. Les différentes directives au préprocesseur, introduites par le caractère `#`, ont pour but :

— l'incorporation de fichiers source (*#include*)

— la définition de constantes symboliques et de macros (*#define*)

— la compilation conditionnelle (*#if, #ifdef,* ...)

2.1 La directive `#include`

Elle permet d'incorporer dans le fichier source, le texte figurant dans un autre fichier. Ce dernier peut être un fichier en-tête de la librairie standard (`stdio.h, math.h, ...`) ou n'importe quel autre fichier. La directive *#include* possède deux syntaxes voisines :

#include <nom-de-fichier> Recherche le fichier mentionné dans un ou plusieurs répertoires systèmes définis par l'implémentation (par exemple, /usr/include/).

#include "nom-de-fichier" Recherche le fichier dans le répertoire courant (celui où se trouve le fichier source). On peut spécifier d'autres répertoires à l'aide de l'option `-I` du compilateur.

La première syntaxe est généralement utilisée pour les fichiers en-tête de la librairie standard, tandis que la seconde est plutôt destinée aux fichiers créés par l'utilisateur.

2.2 La directive `#define`

La directive *#define* permet de définir :

— Des constantes symboliques

— Des macros avec paramètres

Remarque 2.1 :

La directive *#undef* supprime une macro (constante) définie.

Définition des constantes symboliques

La directive *#define nom reste-de-la-ligne* demande au préprocesseur de substituer toute occurence de `nom` par la chaîne de caractères `reste-de-la-ligne` dans la suite du fichier source. Son utilité principale est de donner un nom parlant à une constante, qui pourra être aisément modifiée.

```
#define NB_LIGNES 10
#define NB_COLONNES 33
#define TAILLE_MATRICE NB_LIGNES*NB_COLONNES
```

Définition des macros

Une macro avec paramètres se définit de la manière suivante :

```
#define nom(liste-de-paramètres) corps-de-la-macro
```

Par exemple :

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

Le processeur remplacera dans la suite du code toutes les occurences du type `MAX(x,y)` où x et y sont des symboles quelconques par `(x>y?x:y)`
Une macro a donc une syntaxe similaire à celle d'une fonction, mais son emploi permet en général d'obtenir de meilleures performances en temps d'exécution.
La distinction entre une définition de constante symbolique et celle d'une macro avec paramètres se fait sur le caractère qui suit immédiatement le nom de la macro : si ce caractère est une parenthèse ouvrante, c'est une macro avec paramètres, sinon c'est une constante symbolique. Il ne faut donc jamais mettre d'espace entre le nom de la macro et la parenthèse ouvrante.
Il faut toujours garder à l'esprit que le préprocesseur n'effectue que des remplacements de chaînes de caractères.

En particulier, il est conseillé de toujours mettre entre parenthèses le corps de la macro et les paramètres formels qui y sont utilisés. Par exemple, si l'on écrit sans parenthèses : `#define CARRE(a) a*a` le préprocessuer remplacera `CARRE(a+b)` par $a+b*a+b$ et non par $(a+b)*(a+b)$. De même `!CARRE(x)` sera remplacé par $!x*x$ et non par $!(x*x)$.

2.3 La compilation conditionnelle

La compilation conditionnelle a pour but d'incorporer ou d'exclure des parties du code source dans le texte qui sera généré par le préprocesseur. Elle permet d'adapter le programme au matériel ou à l'environnement sur lequel il s'exécute, ou d'introduire dans le programme des instructions de débogage.
Les directives de compilation conditionnelle se répartissent en deux catégories, suivant le type de condition invoquée :

— la valeur d'une expression

— l'existence ou l'inexistence de symboles

Condition liée à la valeur d'une expression

```
#if condition-1
  partie-du-programme-1
#elif condition-2
  partie-du-programme-2
  ...
#elif condition-n
  partie-du-programme-n
#else
  partie-du-programme-N
#endif
```

Le nombre de `#elif` est quelconque et le `#else` est facultatif. Chaque condition-i doit être une expression constante. Une seule partie-du-programme sera compilée : celle qui correspond à la première condition-i non nulle, ou bien la partie-du-programme-N si toutes les conditions sont nulles.
Par exemple, on peut écrire :

```
#define PROCESSEUR ALPHA
#if PROCESSEUR == ALPHA
  taille_long = 64;
#elif PROCESSEUR == PC
  taille_long = 32;
#endif
```

Condition liée à l'existence d'un symbole

```
#ifdef symbole
  partie-du-programme-1
#else condition-2
  partie-du-programme-2
#endif
```

Si symbole est défini au moment où l'on rencontre la directive `#ifdef`, alors partie-du- programme-1 sera compilée et partie-du-programme-2 sera ignorée. Dans le cas contraire, c'est partie-du-programme-2 qui sera compilée. La directive `#else` est évidemment facultative. De façon similaire, on peut tester la non-existence d'un symbole par :

```
#ifndef symbole
  partie-du-programme-1
#else condition-2
  partie-du-programme-2
#endif
```

Ce type de directive est utile pour rajouter des instructions destinées au débogage du programme :

```
#define DEBUG
  ....
#ifdef DEBUG
  for (i = 0; i < N; i++)
    printf("%d\n",i);
#endif /* DEBUG */
```

Il suffit alors de supprimer la directive `#define DEBUG` pour que les instructions liées au débogage ne soient pas compilées. Cette dernière directive peut être remplacée par l'option de compilation -Dsymbole, qui permet de définir

un symbole.

3. Découper le programme en plusieurs fichiers

Le C (C++) permet de découper le programme en plusieurs fichiers source. Chaque fichier contient une ou plusieurs fonctions. On peut ensuite inclure les fichiers dont on a besoin dans différents projets.
Les fichiers d'en-tête contiennent les déclarations des types et fonctions que l'on souhaite créer.
Un programme écrit en C se compose généralement de plusieurs fichiers-sources. Il y a deux sortes de fichiers-sources :

— ceux qui contiennent effectivement des instructions ; leur nom possède l'extension `.c`,

— ceux qui ne contiennent que des déclarations ; leur nom possède l'extension `.h` (header ou en-tête).

Un fichier `.h` sert à regrouper des déclarations qui sont communes à plusieurs fichiers `.c`, et permet une compilation correcte de ceux-ci. Dans un fichier `.c` on prévoit l'inclusion automatique des fichiers `.h` qui lui sont nécessaires, grâce aux directives de compilation *#include*.
En supposant que le fichier à inclure s'appelle **entete.h**, on écrira *#include <entete.h>* s'il s'agit d'un fichier de la bibliothèque standard du C, ou *#include "entete.h"* s'il s'agit d'un fichier écrit par nous-mêmes.

Exemple 3.1

```c
/***** Prog.c original *****/
#include <stdio.h>
#define J 5

// Fonction main
int main()
{
    // Déclaration des variables
    int i = 10, k;
    k = i+J;
    printf("k = %d\n", k);
    return 0;
}
```

```c
/********** Prog.c **********/
#include <stdio.h>
#define J 5

// Fonction main
int main()
{
    // Déclaration des variables
    int i = 10, k;
    k = i+J;
    printf("k = %d\n", k);
    return 0;
#include "entete.h"
```

```c
/********** entete.h **********/
}
```

Exemple 3.2

```c
/****** main.c ******/
#include <stdio.h>
#include "test.h"

int main()
{
    int const x(5);
    printf("Voici un calcul : %d", x+fonction());

    return 0;
}
```

```c
/****** test.c ******/
#include "test.h"

int fonction()
{
    return 12;
}
```

```c
/****** test.hpp ******/
#ifndef TEST_H
#define TEST_H

int fonction();

#endif
```

```
Générer les fichiers objet : gcc -c test.c main.c
Lier les fichiers :          gcc test.o main.o -o mon_programme.out
Exécuter :                   ./mon_programme.out

Ou
En une seule étape : gcc test.c main.c -o programme.out
Exécuter :           ./programme.out
```

4. Compilation

4.1 A partir de l'IDE

Lancement de la phase de compilation

Pour compiler le fichier source, utilisez l'icône Build. Si vous n'avez pas d'erreur, vous pouvez passer à la phase suivante. Sinon, corrigez les erreurs ...

Exécution du programme

L'exécution n'est possible que si la compilation a été faite sans erreurs.

Les messages d'erreurs de compilation permettent de corriger les fautes de syntaxe. Corrigez les erreurs et relancez jusqu'à obtenir une compilation sans erreurs. Les warnings sont de simples avertissements et n'empêchent pas l'exécution. Faites attention néanmoins à ces warnings.
Le lancement de l'exécution se fait par l'icône Run

Remarque : l'icône Build and Run (compiler et exécuter) lance la compilation et l'exécution par la suite sauf en cas d'erreur de compilation bien sûr.

Dans la section Build log, l'IDE affiche quelques messages en bas de l'IDE. Et si tout va bien, une console apparaît avec le résultat de l'execution du programme.

4.2 Les étapes de la compilation

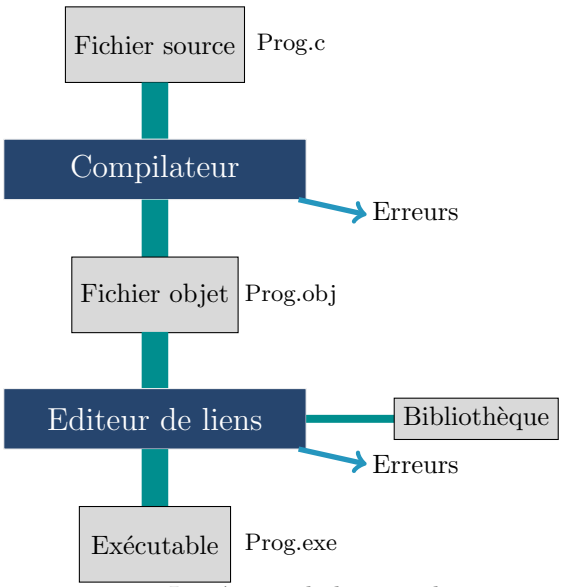

Figure 2. Les étapes de la compilation

La compilation consiste en une série d'étapes de transformation du code source en du code machine exécutable sur un processeur cible.
Le langage C fait partie des langages compilés : le fichier exécutable est produit à partir de fichiers sources par un compilateur.
La compilation passe par différentes phases :

Le préprocessing : Le compilateur analyse le langage source afin de vérifier la syntaxe et de générer un code source brut (s'il y a des erreurs de syntaxe, le compilateur est incapable de générer le fichier objet). Les commentaires sont enlevés et les directives de compilation commençant par # sont d'abord traités pour obtenir le code source brut.

La compilation en fichier objet : Les fichiers de code source brut sont transformés en un fichier dit objet, c'est-à-dire un fichier contenant du code machine ainsi que toutes les informations nécessaires pour l'étape suivante (édition de liens).

L'édition de liens : L'éditeur de liens (linker) s'occupe d'assembler les fichiers objet en une entité exécutable et doit pour ce faire résoudre toutes les adresses non encore résolues. C'est à dire que lorsqu'il fait appel dans le fichier objet à des fonctions ou des variables externes, l'éditeur de liens recherche les objets ou bibliothèques concernés et génère l'exécutable (Il se produit une erreur lorsque l'éditeur de liens ne trouve pas ces références).

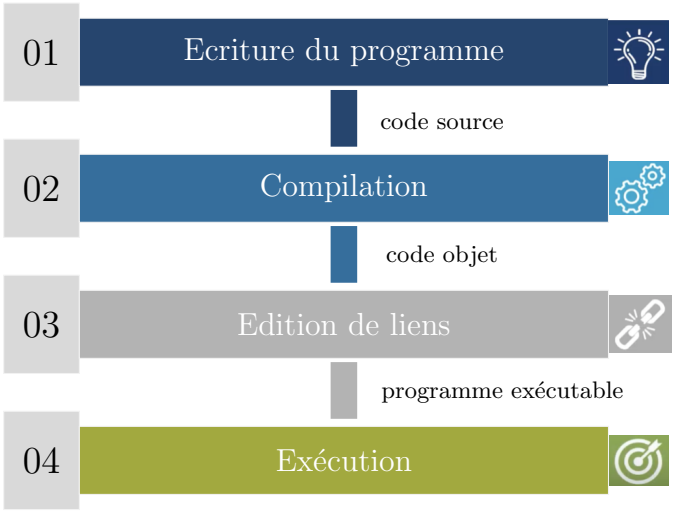

Figure 3. Les étapes pour exécuter un programme

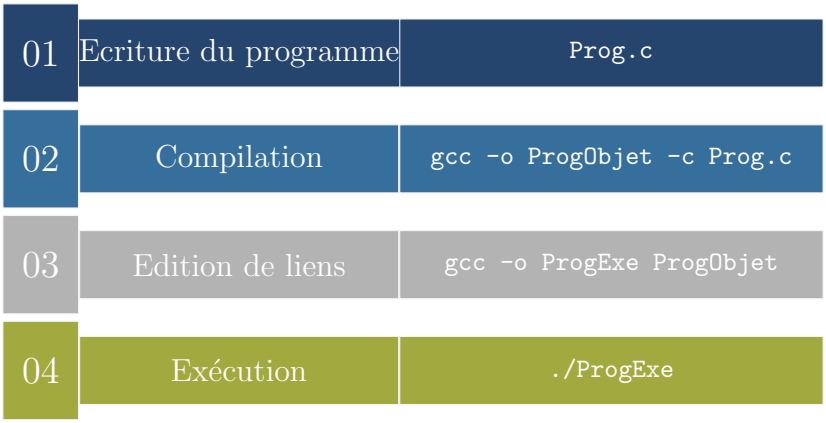

Figure 4. Les commandes pour exécuter un programme

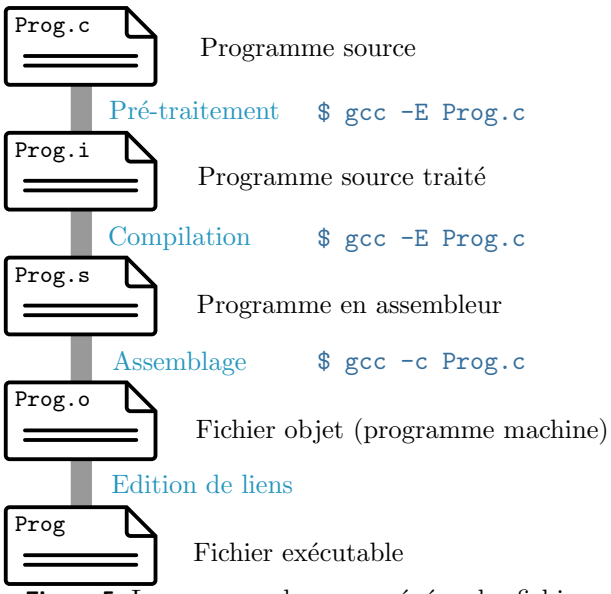

Figure 5. Les commandes pour générer les fichiers

Table 1. Méthode 1 - Commandes pour exécuter un programme en C

Commande[1]	Type du fichier généré	Nom du fichier généré
`gcc Prog.c -o Prog`	Exécutable	Prog

`./Prog`	Pour exécuter le programme

[1] Les commandes doivent être réalisées dans l'ordre.

Table 2. Méthode 2 - Commandes pour exécuter un programme en C

Commande[2]	Type du fichier généré	Nom du fichier généré
`gcc -o ProgObjet -c Prog.c`	Objet	ProgObjet
`gcc -o ProgExe ProgObjet`	Exécutable	ProgExe

`./ProgExe`	Pour exécuter le programme

[2] Les commandes doivent être réalisées dans l'ordre.

Table 3. Méthode 3 - Commandes pour exécuter un programme en C

Commande[3]	Type du fichier généré	Nom du fichier généré
`gcc -save-temps Prog.c -o ProgExe`	PS* traité	Prog.i
	Assembleur	Prog.s
	Objet	Prog.o
	Exécutable	ProgExe

`./ProgExe`	Pour exécuter le programme

[3] Les commandes doivent être réalisées dans l'ordre.

* PS : Programme Source.

Table 4. Méthode 4 - Commandes pour exécuter un programme en C

Commande[4]	Type du fichier généré	Nom du fichier généré
`gcc -E Prog.c`	Affiche le programme source traité	
`gcc -c Prog.c`	Objet	Prog.o
`gcc -o Prog Prog.o`	Exécutable	Prog
`./Prog`	Pour exécuter le programme	

[4] Les commandes doivent être réalisées dans l'ordre.

Remarque 4.1 :

Pour l'exécution des programmes C++, il suffit de remplacer `gcc` par `g++`

Le contenu du dossier

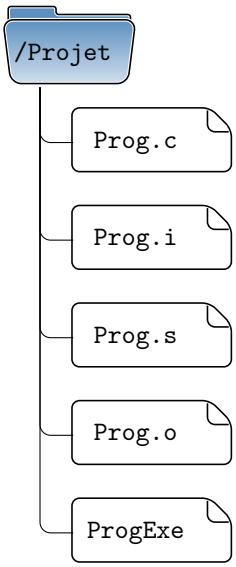

Figure 6. Fichiers générés en utilisant les commandes du tableau 3

Les fichiers générés

Contenu des fichiers générés en utilisant les commandes du tableau 3 :

Programme source : Prog.c Taille = 170 octets

```c
#include <stdio.h>
#define J 5

// Fonction main
int main()
{
    // Déclaration des variables
    int i = 10, k;
    k = i+J;
    printf("k = %d\n", k);
    return 0;
}
```

Programme source traité : Prog.i Taille = 23 ko

```
# 1 "Prog.c"
# 1 "<built-in>" 1
# 1 "<built-in>" 3
# 363 "<built-in>" 3
# 1 "<command line>" 1
# 1 "<built-in>" 2
# 1 "Prog.c" 2
# 1 "/Library/Developer/CommandLineTools/SDKs/MacOSX.sdk/usr/include/stdio.h" 1 3 4
.
.
500 lignes plus bas
.
.
extern int __vsnprintf_chk (char * restrict, size_t, int, size_t,
       const char * restrict, va_list);
# 408 "/Library/Developer/CommandLineTools/SDKs/MacOSX.sdk/usr/include/stdio.h" 2 3 4
# 2 "Prog.c" 2



int main()
{

    int i = 10, k;
    k = i+5;
    printf("k = %d\n", k);
    return 0;
}
```

Programme en assembleur : Prog.s Taille = 474 octets

```
.section __TEXT,__text,regular,pure_instructions
.build_version macos, 10, 15, 4 sdk_version 10, 15, 4
.globl _main                    ## -- Begin function main
.p2align 4, 0x90
_main:                                  ## @main
.cfi_startproc
## %bb.0:
pushq %rbp
.cfi_def_cfa_offset 16
.cfi_offset %rbp, -16
movq %rsp, %rbp
.cfi_def_cfa_register %rbp
subq $16, %rsp
movl $0, -4(%rbp)
movl $10, -8(%rbp)
movl -8(%rbp), %eax
addl $5, %eax
movl %eax, -12(%rbp)
movl -12(%rbp), %esi
leaq L_.str(%rip), %rdi
movb $0, %al
callq _printf
xorl %ecx, %ecx
movl %eax, -16(%rbp)         ## 4-byte Spill
movl %ecx, %eax
addq $16, %rsp
popq %rbp
retq
.cfi_endproc
                                        ## -- End function
.section __TEXT,__cstring,cstring_literals
L_.str:                                 ## @.str
.asciz "k = %d\n"


.subsections_via_symbols
```

Programme objet : Prog.o Taille = 792 octets

```
cffa edfe 0700 0001 0300 0000 0100 0000
0400 0000 0802 0000 0020 0000 0000 0000
1900 0000 8801 0000 0000 0000 0000 0000
.
.
40 lignes plus bas
.
.
0000 0000 0100 0006 0100 0000 0f01 0000
0000 0000 0000 0000 0700 0000 0100 0000
0000 0000 0000 0000 005f 6d61 696e 005f
7072 696e 7466 0000
```

Programme Exécutable : ProgExe.out Taille = 13 ko

```
cffa edfe 0700 0001 0300 0000 0200 0000
1000 0000 5805 0000 8500 2000 0000 0000
1900 0000 4800 0000 5f5f 5041 4745 5a45
524f 0000 0000 0000 0000 0000 0000 0000
.
.
800 lignes plus bas
.
.
0300 0000 2000 5f5f 6d68 5f65 7865 6375
7465 5f68 6561 6465 7200 5f6d 6169 6e00
5f70 7269 6e74 6600 6479 6c64 5f73 7475
625f 6269 6e64 6572 005f 5f64 796c 645f
7072 6976 6174 6500 0000 0000
```

4.3 Organisation d'un projet : Exemple

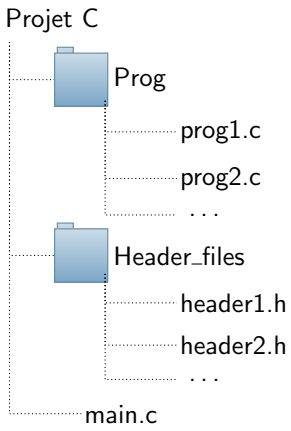

Figure 7. Contenu du répertoire `Projet C`

4.4 Résumé

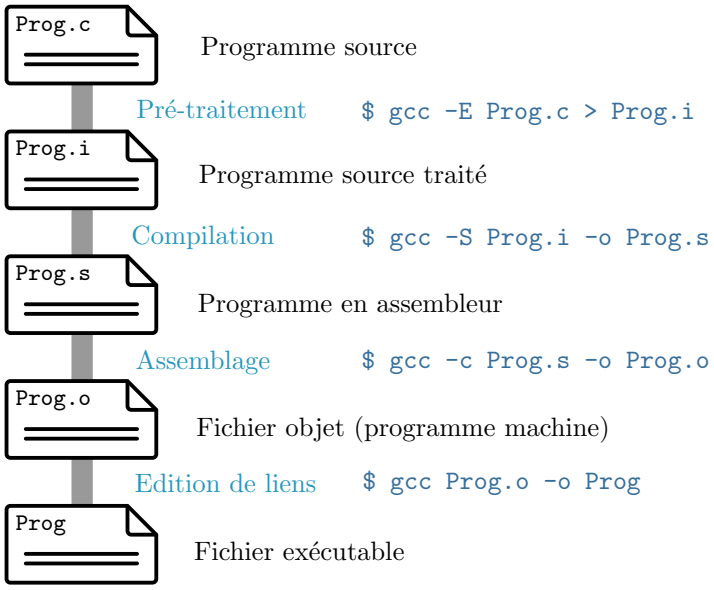

Figure 8. Les commandes pour compiler un programme étape par étape

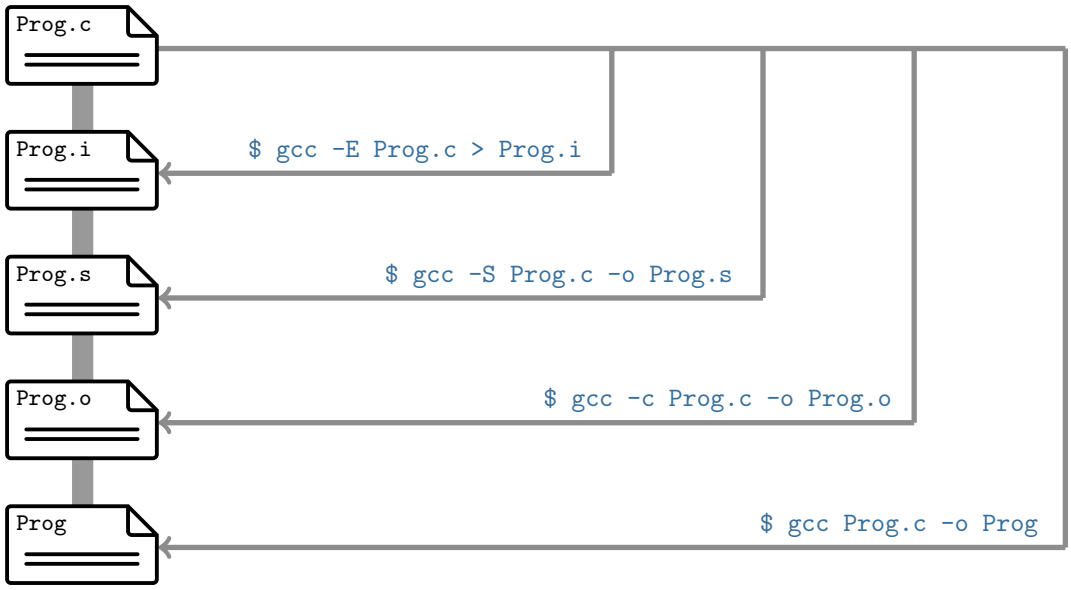

Figure 9. Les commandes pour compiler un programme

Commande pour générer tous les fichiers lors de la compilation : `$ gcc --save-temps Prog.c -o Prog`

4.5 Les options du compilateur

- `gcc source.c` : Crée un fichier binaire exécutable `a.out`
- `-o` (`gcc source.c -o prog`) : Crée un fichier binaire exécutable nommé `Prog` (Spécifie le nom de l'exécutable à générer)
- `-v` : Visualise toutes les actions effectuées lors d'une compilation
- `-c` : Supprime l'édition de lien. Seul(s) le(s) fichier(s) `.o` est (sont) généré(s) pour réaliser des bibliothèques pré-compilées par exemple
- `-S` : Ne génère pas le fichier exécutable et crée un fichier `.s` contenant le programme en assembleur
- `-L` : Indique un chemin de recherche supplémentaire à l'éditeur de liens pour d'autres librairies
- `-l` : Indique une nouvelle librairie à charger

Exemple 4.1

— Créer le fichier source `Prog.c` qui contient le code en langage C suivant :

```c
#include <stdio.h>
int main(void)
{
    printf("Bonjour\n");
    return(0);
}
```

— Compiler le programme pour créer l'exécutable : `gcc Prog.c`

— Lancer l'exécutable : `./a.out`

— `gcc Prog.c -o ProgExe.out` remplacera `a.out` par `ProgExe.out`

— Lancer l'exécutable : `./ProgExe.out`

— Recompiler ce programme en spécifiant l'option `-v` et commenter ce qui s'affiche (`gcc Prog.c -v`)

4.6 La compilation séparée

Lorsqu'on réalise de gros programmes (projets), on doit découper ceux-ci en plusieurs programmes sources. L'avantage est un gain de temps au moment de la compilation. Si une application est constituée des 3 fichiers sources `source1.c`, `source2.c` et `source3.c` la compilation d'un exécutable nommé `exec` est effectuée par la ligne suivante :

```
gcc -o exec source1.c source2.c source3.c
```

L'option `-c` du compilateur permet de générer des fichiers objets sans effectuer une édition de lien. Si on modifie un des fichiers sources (par exemple, `source2.c`), on utilise cette option pour ne compiler que le fichier modifié :

```
gcc -c source2.c
gcc -o exec source2.o source1.o source3.o
```

Remarque 4.2 : Compilation séparée

Cette technique devient indispensable lors de la création de gros logiciels qui peuvent nécessiter des temps de compilation de plusieurs minutes, ou dans le cas d'un travail collaboratif (projet par exemple).

Exemple

On réalise un programme qui demande de saisir le rayon d'un cercle, calcul et affiche le périmètre et la surface de ce cercle. Le programme source est découpé en trois fichiers sources :

— `main.c` : Programme principal

— `perimetre.c` , `surface.c` : Programmes contenant les fonctions `peri` et `surf` pour le calcul du périmètre et de la surface du cercle respectivement

— `cercle.h` : Le fichier d'en-tête (header) contenant le prototype des fonctions `peri` et `surf`

- Créer les trois programmes sources
- Créer le fichier d'en-tête `cercle.h`
- Compiler les fichiers `main.c perimetre.c surface.c` de manière à obtenir l'exécutable nommé `calcul`
  ```
  $ gcc -o calcul main.c perimetre.c surface.c
  ```
- Compiler `perimetre.c` et `surface.c` de manière à créer les fichiers objets `perimetre.o` et `surface.o`
  ```
  $ gcc -c perimetre.c surface.c
  ```
- Re-créer `calcul` en compilant et liant `main.c perimetre.o` et `surface.o`
  ```
  $ gcc main.c perimetre.o surface.o                Exécutable généré ⟶ a.out
  $ gcc -o calcul main.c perimetre.o surface.o      Exécutable généré ⟶ calcul
  ```

On souhaite modifier le programme `surface.c` de manière à retourner toujours la surface en cm^2 ($1\ m^2 = 10^4\ cm^2$)

- Modifier uniquement `surface.c`
- Créer uniquement un nouveau programme objet `surface.o` et re-compiler l'ensemble comme précédemment. Conclure.
  ```
  $ gcc -c surface.c
  $ gcc -o calcul surface.o main.c perimetre.o
  $ ./calcul
  ```

Fichier main.c

```c
#include <stdio.h>
#include <math.h>
#include "cercle.h"

int main(void)
{
double r, p, s;
      printf("Entrer le rayon du cercle : ");
      scanf("%lf",&r);

      p = peri(r);
      s = surf(r);
      printf("\nLe périmètre du cercle de rayon %lf est : %lf",r,p);
      printf("\nLa surface du cercle de rayon %lf est : %lf\n",r,s);
      return(0);
}
```

Fichier perimetre.c

```c
#include <math.h>

double peri(double x)
{
      return 2*M_PI*x;
}
```

Fichier surface.c

```c
#include <math.h>

double surf(double x)
{
      //return M_PI*pow(x,2)
      return M_PI*pow(x,2)*10000;
}
```

Fichier cercle.h

```c
#ifndef CERCLE_H
#define CERCLE_H

double peri(double r);
double surf(double r);

#endif
```

5. Construction de bibliothèque (Librarie en anglais)

Il est intéressant de se construire des bibliothèques contenant les fonctions les plus fréquemment utilisées plutôt que de les réécrire à chaque projet ou programme. Il suffit ensuite d'indiquer vos librairies au moment de la compilation. Pour cela, les options `-L` et `-l` permettent respectivement d'inclure un nouveau chemin de recherche pour l'éditeur de lien et d'indiquer le nom de librairie.

5.1 Bibliothèque statique (.lib, .a) et bibliothèque partagée (.dll, .so)

Une bibliothèque est une collection de fichiers objets pré-compilés qui peuvent être liés à vos programmes via l'éditeur de liens. Par exemple, les fonctions système telles que `printf()` et `sqrt()`.

Il existe deux types de bibliothèques externes : la bibliothèque statique et la bibliothèque partagée.

— Une bibliothèque statique a l'extension de fichier `.a` (fichier d'archive) sous Unix ou `.lib` (bibliothèque) sous Windows. Lorsque votre programme est lié à une bibliothèque statique, le code machine des fonctions externes utilisées dans votre programme est copié dans l'exécutable. Une bibliothèque statique peut être créée via le programme d'archivage `ar.exe`.

— Une bibliothèque partagée a l'extension de fichier `.so` (shared objects : objets partagés) sous Unix ou `.dll` (dynamic link library : bibliothèque de liens dynamiques) sous Windows. Lorsque votre programme est lié à une bibliothèque partagée, seule une petite table est créée dans l'exécutable. Avant que l'exécutable ne commence à s'exécuter, le système d'exploitation charge le code machine nécessaire aux fonctions externes - un processus connu sous le nom de liaison dynamique. La liaison dynamique réduit la taille des fichiers exécutables et économise de l'espace disque, car une copie d'une bibliothèque peut être partagée entre plusieurs programmes. De plus, la plupart des systèmes d'exploitation autorisent l'utilisation d'une copie d'une bibliothèque partagée en mémoire par tous les programmes en cours d'exécution, économisant ainsi de la mémoire. Les codes de la bibliothèque partagée peuvent être mis à jour sans qu'il soit nécessaire de recompiler votre programme.

En raison de l'avantage de la liaison dynamique, GCC établit par défaut un lien vers la bibliothèque partagée si elle est disponible.
Vous pouvez lister le contenu d'une bibliothèque via `nm filename`.

5.2 Recherche de fichiers d'en-tête et de bibliothèques (-I(i), -L et -l)

Lors de la compilation du programme, le compilateur a besoin des fichiers d'en-tête pour compiler les codes sources ; l'éditeur de liens a besoin des bibliothèques pour résoudre les références externes à partir d'autres fichiers objets ou bibliothèques. Le compilateur et l'éditeur de liens ne trouveront pas les en-têtes/bibliothèques à moins que vous ne définissiez les options appropriées.

Pour chacun des fichiers headers (en-têtes) utilisés dans votre programme (via les directives `include`), le compilateur recherche les chemins d'inclusion pour ces en-têtes. Les chemins d'inclusion sont spécifiés via l'option `-Idir` (ou la variable d'environnement CPATH). Comme le nom de fichier de l'en-tête est connu (par exemple, `iostream`, `stdio.h`), le compilateur n'a besoin que des répertoires.

L'éditeur de liens recherche dans les chemins de bibliothèque les bibliothèques nécessaires pour lier le programme à un exécutable. Le chemin de la bibliothèque est spécifié via l'option `-Ldir` ('L' majuscule suivi du chemin du répertoire) (ou la variable d'environnement LIBRARY_PATH). De plus, vous devez également spécifier le nom de la bibliothèque. Sous Unix, la bibliothèque `libxxx.a` est spécifiée via l'option `-lxxx` (lettre minuscule 'l', sans le préfixe "lib" et l'extension ".a"). Sous Windows, indiquez le nom complet tel que `-lxxx.lib`. L'éditeur de liens doit connaître à la fois les répertoires et les noms des bibliothèques. Par conséquent, deux options doivent être spécifiées :

5.3 Exemple

On reprend les fichiers de l'exemple précédent :

- Créer un nouveau dossier (répertoire) nommé `TestLib`
- Copier et coller dans ce dossier uniquement les programmes `perimetre.c` et `surface.c`
- On compile les fichiers `perimetre.c` , `surface.c` pour générer les fichiers `objet` : `perimetre.o` , `surface.o`
  ```
  $ gcc -c perimetre.c surface.c
  ```
  Des avertissements (warning) peuvent apparaitre mais les fichiers `objet` seront générés normalement.
  Il est maintenant possible de lier ces fichiers `objet` lors d'une compilation.

Pour cette fois, il suffira de les déclarer dans un fichier d'en-tête (header), la compilation ne sera plus nécessaire d'où un gain de temps.

Fichier `libcercle.h`

```
#ifndef CERCLE_H
#define CERCLE_H

double peri(double r);
double surf(double r);

#endif
```

- Copier et coller dans le dossier `TestLib`, le programme `main.c` de l'exemple précédent
- Renommer ce programme `main2.c`
- Modifier-le comme montré ci-dessous

Fichier `main2.c`

```
#include <stdio.h>
#include <math.h>
#include "libcercle.h"

int main(void)
{
double r, p, s;
      printf("Entrer le rayon du cercle : ");
      scanf("%lf",&r);

      p = peri(r);
      s = surf(r);
      printf("\nLe périmètre du cercle de rayon %lf est : %lf",r,p);
      printf("\nLa surface du cercle de rayon %lf est : %lf\n",r,s);
      return(0);
}
```

- La compilation peut se faire par la commande : `$ gcc main2.c perimetre.o surface.o`
  Seul `main2.c` est compilé pour générer `main2.o` qui sera lié à `perimetre.o` et `surface.o`
  Les fichiers `objet *.o` peuvent être regroupés dans une bibliothèque.
- Créer la librairie `libcercle.a` (les noms des bibliothèques commencent toujours par `lib`) à l'aide de la commande `ar` qui crée un fichier archive :
  `ar r libcercle.a perimetre.o surface.o`
- On peut vérifier le contenu de la librairie en tapant :
  `ar t libcercle.a`

5.4 Exemple (suite) : Utilisation de la librairie

A partir de cette étape, on peut déjà utiliser ces fonctions en respectant quelques conditions.

— La première condition est de placer `libcercle.a` dans le répertoire où on effectue la compilation.

— La seconde condition est de compiler le programme en spécifiant le nom de la librairie.

- Compiler le programme en exécutant la ligne suivante :
  `$ gcc -o test main2.c libcercle.a`

— On peut également placer toutes nos librairies dans un de nos répertoires (par exemple, `"librairies"` qu'on créera par la commande `$ mkdir librairies`).

- On lance alors la compilation de la manière suivante :
  `$ gcc -o test2 main2.c -L librairies -l cercle`
  - `-L` indique au linker un nouveau chemin pour les bibliothèques
  - `-l` indique le nom de la bibliothèque. On remarque que le nom de la bibliothèque `libcercle.a` devient `cercle` dans la ligne de commande.

6. Création d'une librairie (classe) pour Arduino

6.1 Ecriture de la classe

On va implémenter les programmes dans une classe nommée Diode.

— Nom de la classe : `Diode`

— Attribut : `_pin` (privé)

— Méthodes :

  — `Diode(int pin)` : Constructeur de la classe (public)

  — `oncourt()` : void (public)

  — `onlong()` : void (public)

L'attribut (privé) de cette classe est `_pin` qui représente la broche où sera connectée la LED.
Les méthodes (publiques) sont `Diode(int pin)` le constructeur de la classe, `oncourt()` la méthode qui permet d'allumer la LED pendant une durée courte et `onlong()`, la méthode qui permet d'allumer la LED pendant une durée longue.

- Création du fichier d'en-tête (header) nommé `Diode.h`

```
#ifndef DIODE_H
#define DIODE_H
#include "Arduino.h"

class Diode
{
    public:
        Diode(int pin);
        void oncourt();
        void onlong();
    private:
        int _pin;
};

#endif
```

La ligne `#include "Arduino.h"` permet d'avoir accès aux variables et constantes spécifiques à Arduino.
Une classe doit posséder une méthode spécifique (une fonction) appelée constructeur qui est chargé d'initialiser une instance (objet) de la classe. Le constructeur est appelé systématiquement au moment de la création de l'objet, il porte le même nom que la classe et ne retourne aucun type.
Les méthodes `oncourt()` et `onlong()` sont définies comme public.
L'attribut `_pin` est déclaré comme privé, cette variable contiendra le numéro du pin où la LED est branchée. On utilise généralement le caractère `_` pour distinguer l'appellation de l'attribut de l'argument de la fonction.

- Création du fichier nommé `Diode.cpp`
  Ce fichier contiendra la description des méthodes.

```
#include "Arduino.h"
#include "Diode.h"

Diode::Diode(int pin)
{
    pinMode(pin, OUTPUT);
    _pin = pin;
}

void Diode::oncourt()
{
    digitalWrite(_pin, HIGH);
    delay(250);
    digitalWrite(_pin, LOW);
    delay(250);
}

void Diode::onlong()
{
```

```
    digitalWrite(_pin, HIGH);
    delay(750);
    digitalWrite(_pin, LOW);
    delay(250);
}
```

Le fichier contient sur les deux premières lignes : `#include "Arduino.h"` (accès au fonctions Arduino) et `#include "Diode.h"` (définition de l'en-tête associé).
Puis vient la définition des méthodes.
Toutes les méthodes commencent par `Diode::`
On retrouve le constructeur (qui effectue aussi l'initialisation de la broche en sortie) puis les deux méthodes (`oncourt` et `onlong`) qui correspondaient aux fonctions dans le cas d'une "implémentation normale". On note la différence entre `pin` et `_pin`. Les méthodes ne "manipulent pas" `pin` mais sa "réplique" privée `_pin`.
`_pin` prend la valeur de `pin` au moment de l'instanciation de l'objet.

- Placer les deux fichiers créés précédemment dans un répertoire `Diode` (à créer) puis déplacer ce répertoire dans le repertoire `libraries` d'Arduino.
- Relancer Arduino et vérifier que le répertoire est bien vu par Arduino en effectuant l'opération `Croquis -> Importer bibliothèque`.
- Saisir le programme `ProgTest.ino` permettant de tester la classe réalisée (voir code ci-dessous)

```
#include <Diode.h>

Diode diode(13);

void setup() {
  // put your setup code here, to run once:

}

void loop() {
  // put your main code here, to run repeatedly:
  diode.oncourt(); diode.oncourt(); diode.oncourt();
  delay(500);
  diode.onlong(); diode.onlong(); diode.onlong();
  delay(500);
  diode.oncourt(); diode.oncourt(); diode.oncourt();
  delay(1500);
}
```

- Vérifier le bon fonctionnement du programme `ProgTest.ino`

Il est souhaitable lorsque l'on récupère une bibliothèque (une classe) d'avoir des exemples de mise en œuvre de cette bibliothèque.

- Il faut placer le répertoire `ProgTest` contenant le fichier `ProgTest.ino` (programme Arduino) dans un répertoire `examples` au sein de notre répertoire `Diode`.
  - `Arduino/Libraries/Diode/examples/ProgTest/ProgTest.ino`
  - `Diode.h` et `Diode.cpp` sont placés dans `Arduino/Libraries/Diode/`
- Relancer Arduino.
- Vérifier que le fichier `ProgTest.ino` est bien accessible à partir du menu Arduino `Fichier->Exemples->Diode`
- Pour réaliser notre librairie dans les règles de l'art, on doit spécifier dans un fichier les propriétés de la librairie créée (voir l'exemple ci-dessous nommé `library.properties`) qui doit être inclus dans le répertoire `Diode`

```
name=Diode
version=1.0.0
author=Kamal B., LTP Charles Carnus
maintainer=LTP <carnuslab@carnus.fr>
sentence=Allows Arduino boards to control a variety of LEDs.
paragraph=This library can control a great number of LEDs.<br />
```

```
category=Device Control
url=https://www.carnus.fr/
architectures=avr,megaavr,sam,samd,nrf52,stm32f4,mbed,mbed_nano,mbed_portenta,mbed_rp2040
```

- Si on utilise les anciennes versions d'Arduino, il est possible que le nom de la classe et des méthodes soit coloré en noir. On peut obtenir la coloration syntaxique du programme réalisé en créant un fichier nommé `keywords.txt` (voir l'exemple ci-dessous).

```
#######################################
# Syntax Coloring Map LED
#######################################

#######################################
# Datatypes (KEYWORD1)
#######################################

Diode KEYWORD1 Diode

#######################################
# Methods and Functions (KEYWORD2)
#######################################
oncourt KEYWORD2
onlong KEYWORD2

#######################################
# Constants (LITERAL1)
#######################################
```

Chaque ligne porte le nom d'un mot clé suivi d'une tabulation (pas d'espaces) suivi de KEYWORD1 pour une classe ou de KEYWORD2 pour une méthode. La classe sera ainsi coloré en bleu-vert (ou orange), les méthodes en marron (la différence de couleur est faible). Il faut là aussi re-démarrer Arduino pour que la modification soit effective.

Exercice

Réaliser une classe permettant d'allumer une LED, de l'éteindre ou de la faire clignoter.

- Nom de la classe : `Led`
- Attributs :
  - `_pin` (privé)
  - `_etat` (privé)
- Méthodes :
  - `Led(int pin)` : Constructeur de la classe (public)
  - `ledOn()` : void (public)
  - `ledOff()` : void (public)
  - `ledClign(int T, int N)` : void (public)
  - `ledChange()` : void (public)
  - `ledEtat()` : void (public)

Définition des attributs

`_pin` : Correspond au numéro de la broche.
`_etat` : Correspond à l'état de la LED (false : LED éteinte ; true : LED allumée).

Définition des méthodes

`Led(int pin)` : Constructeur de la classe.
`ledOn()` : Allume la LED et renseigne son état.
`ledOff()` : Eteint la LED et renseigne son état.
`ledClign(int T, int N)` : Fait clignoter la LED N fois, la LED est allumée de 0 à $\frac{T}{2}$, éteinte de $\frac{T}{2}$ à T.
`ledChange()` : Allume la LED si elle est éteinte, éteint la LED si elle est allumée et renseigne l'état de la LED.
`ledEtat()` : Permet de récupérer l'état de la LED.

Remarque : `ledClign(int T, int N)` et `ledChange()` pourront utiliser `ledOn()` et `ledOff()`.

Ecrire tous les fichiers (`Led.h`, `Led.cpp`, fichier exemple, fichier keyword, fichier `library.properties`) permettant de faire fonctionner cette classe et de valider son bon fonctionnement.
Le fichier exemple devra tester toutes les méthodes et utilisera le moniteur série (en particulier) pour valider la méthode `ledEtat()`.

7. Résumé

7.1 Compilation directe

`gcc source.c -o sourceExe`
`./sourceExe` pour exécuter le programme.

7.2 Compilation par étapes ordonnées

Table 5. Commandes pour la compilation d'un programme en C par étapes ordonnées

Commande	Action	Fichier généré
`gcc -E source.c > prog.i`	Pré-traitement	prog.i
`gcc -S prog.i -o prog.s`	Compilation	prog.s
`gcc -c prog.s -o prog.o`	Assemblage	prog.o
`gcc prog.o -o progExe`	Edition des liens	progExe

7.3 Compilation par étapes

Table 6. Commandes pour la compilation d'un programme en C par étapes

Commande	Action	Fichier généré
`gcc -E source.c > prog.i`	Pré-traitement	prog.i
`gcc -S source.c -o prog.s`	Compilation	prog.s
`gcc -c source.c -o prog.o`	Assemblage	prog.o
`gcc source.c -o progExe`	Edition des liens	progExe

7.4 Commande pour générer tous les fichiers

`gcc -save-temps Prog.c -o progExe`

7.5 Options

- `gcc source.c` : Crée un fichier binaire exécutable `a.out`
- `-o (gcc source.c -o progExe)` : Crée un fichier binaire exécutable nommé `ProgExe` (Spécifie le nom de l'exécutable à générer)
- `-v` : Visualise toutes les actions effectuées lors d'une compilation
- `-c` : Supprime l'édition de lien. Seul(s) le(s) fichier(s) `.o` est (sont) généré(s) pour réaliser des bibliothèques pré-compilées par exemple
- `-S` : Ne génère pas le fichier exécutable et crée un fichier `.s` contenant le programme en assembleur
- `gcc -o exec source1.c source2.c source3.c` compilation et génération de l'exécutable `exec`
- Si on modifie un seul des fichiers :
  - `gcc -c source2.c`
  - puis `gcc -o exec source2.o source1.o source3.o`

7.6 Les librairies

- `-L` : Indique un chemin de recherche supplémentaire à l'éditeur de liens pour d'autres librairies
- `-l` : Indique une nouvelle librairie à charger
- `ar r libcercle.a perimetre.o surface.o` ensuite on vérifie le contenu de la librairie `ar t libcercle.a`
- `gcc -o test main2.c libcercle.a`
- `gcc -o test2 main2.c -L librairies -l cercle`

7.7 Autres options

— `$ g++ -Wall -g -o progExe source.cpp`

— `-o` : spécifie le nom du fichier exécutable de sortie.

— `-Wall` : imprime "tous" les messages d'avertissement.

— `-g` : génère des informations de débogage symboliques supplémentaires à utiliser avec le débogueur gdb.

```
$ gcc -c hello.c
$ gcc -o hello.exe hello.o

$ file hello.c
hello.c: C source, ASCII text, with CRLF line terminators

$ file hello.o
hello.o: data

$ (ou >) file hello.exe
hello.exe: PE32 executable (console) x86-64, for MS Windows

$ nm hello.o

$ nm hello.exe | grep main
```

8. makefile

Ci-dessous un exemple de fichier `makefile` qui doit être nommé `makefile` ou `Makefile` ou `GCCMakefile` sans aucune extension.

```
all: hello.exe

hello.exe: hello.o
 gcc -o hello.exe hello.o

hello.o: hello.c
 gcc -c hello.c

clean:
 rm hello.o hello.exe
```

Pour exécuter le programme `hello.c`, on utilise la commande : `$ make`
Résultat : lancement de la compilation et le terminal affiche :

```
gcc -c hello.c
gcc -o hello.exe hello.o
```

8.1 Syntaxe

```
target: pre-req-1 pre-req-2 ...
    command
```

Target et pre-requisites sont séparés par deux-points (:). La commande doit être précédée d'une tabulation (PAS d'espaces).

Lorsqu'il est demandé à `make` d'évaluer une règle, il commence par rechercher les fichiers dans les pré-requis.

Dans l'exemple ci-dessus, la règle "all" a un pré-requis `hello.exe`. `make` ne trouve pas le fichier `hello.exe`, il recherche donc une règle pour le créer. La règle `hello.exe` a un pré-requis `hello.o`. Encore une fois, il n'existe pas, donc `make` cherche une règle pour le créer. La règle `hello.o` a un pré-requis `hello.c`. vérifie que `hello.c` existe et qu'il est plus récent que la cible (qui n'existe pas). Il exécute la commande `$ gcc -c hello.c`. La règle `hello.exe` lance alors sa commande `$ gcc -o hello.exe hello.o`. Enfin, la règle "all" ne fait rien.

Plus important encore, si le pré-requis n'est pas plus récent que la cible, la commande ne sera pas exécutée. En d'autres termes, la commande ne sera exécutée que si la cible est périmée par rapport à son pré-requis. Par exemple, si nous réexécutons la commande `make` :

```
make: Nothing to be done for 'all'.
```

Vous pouvez également spécifier la cible à créer dans la commande `make`. Par exemple, la cible `clean` supprime les `hello.o` et `hello.exe`. Vous pouvez ensuite exécuter le `make` sans cible, ce qui revient au même que `make all`.

```
$ make clean

rm hello.o hello.exe
```

8.2 Cibles factices (ou cibles artificielles)

Une cible qui ne représente pas un fichier est appelée une fausse cible. Par exemple, le "clean" dans l'exemple ci-dessus, qui est juste une étiquette pour une commande. Si la cible est un fichier, il sera vérifié par rapport à ses pré-requis pour l'obsolescence. La fausse cible est toujours obsolète et sa commande sera exécutée. Les fausses cibles standard sont : `all, clean, install`.

8.3 Variables

Une variable commence par un `$` et est entourée de parenthèses (...) ou d'accolades {...}. Les variables à caractère unique n'ont pas besoin de parenthèses. Par exemple, `$(CC)`, `$(CC_FLAGS)`, `$@`, `$^`.

8.4 Variables automatiques

Les variables automatiques sont définies par make après la correspondance d'une règle. Il comprend :

— `$@` : le nom du fichier cible.

— `$*` : le nom du fichier cible sans l'extension de fichier.

— `$<` : le premier nom de fichier pré-requis.

— `$^` : les noms de fichiers de tous les pré-requis, séparés par des espaces, éliminent les doublons.

— `$+` : similaire à `$^`, mais inclut les doublons.

— `$?` : les noms de tous les pré-requis qui sont plus récents que la cible, séparés par des espaces.