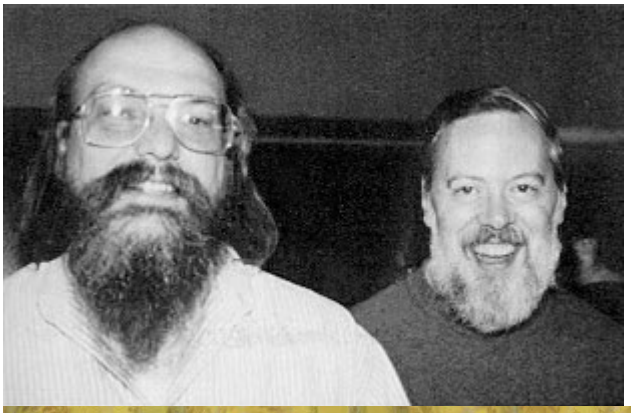
Langage C

Introduction au langage C

Le langage C a été inventé au cours de l'année 1972 dans les laboratoires Bell (le Google de l'époque) par Dennis Ritchie et Ken Thompson pour développer le système d'exploitation Unix.

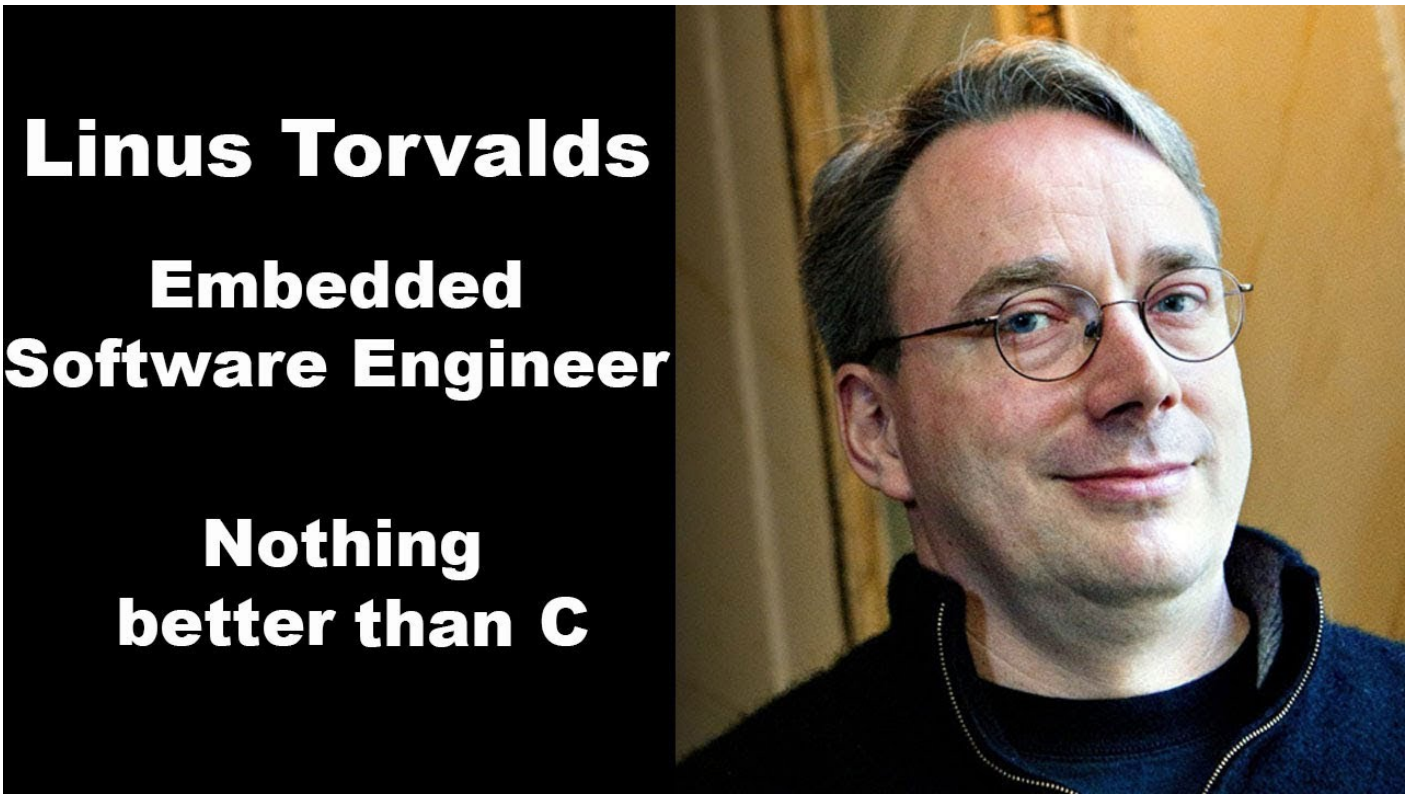
Brian Kernighan contribua avec Dennis Ritchie à populariser le langage C avec le livre "The C programming Langage", encore appelé le "C K&R" pour les initiales des deux auteurs.



Le langage C pour développer des OS (Kernel) / Drivers

C est un langage proche du matériel, adapté pour programmer les sytèmes d'exploitation (OS), les drivers, les micro-contrôleurs, les DSP, les logiciels embarqués, mais pas seulement; l'interpréteur Python, la machine virtuelle Java ou le "moteur" PHP sont également écrits en C/C++.

Linus Torvalds (le créateur de Linux) répondait ainsi à la question de savoir s'il voyait un autre langage que le C pour le développement de systèmes d'exploitation.



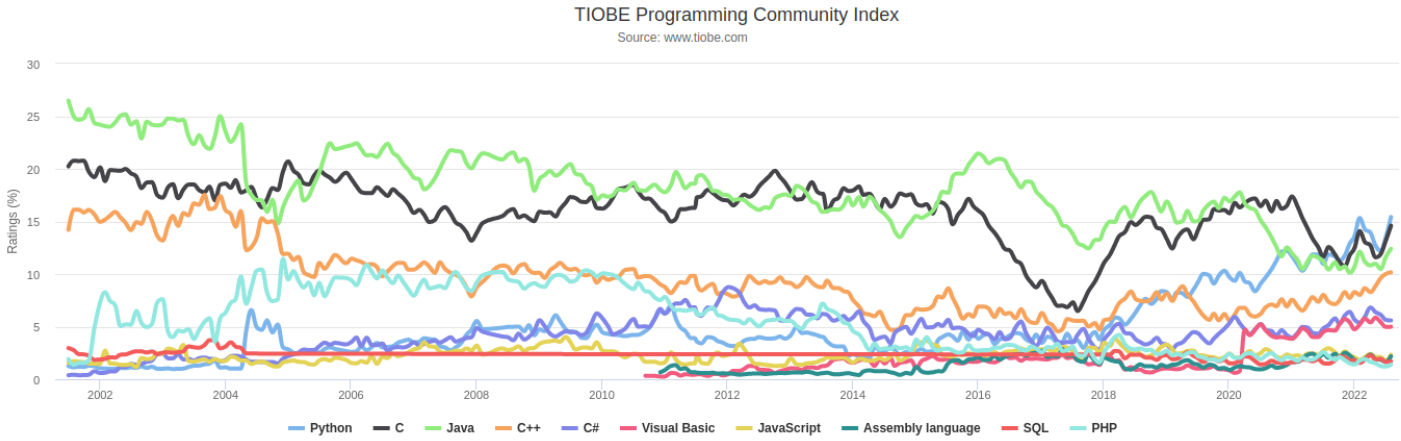
La raison pour laquelle je me suis lancé dans Linux et les systèmes d'exploitation en général est que j'aime vraiment le hardware. Vu sous cet angle, je n'ai pas encore vu un langage de programmation qui approche seulement le langage C. Cette affirmation ne tient pas uniquement à ce que le C soit utile pour générer du bon code pour piloter le matériel. Ce qu'il faut dire en plus c'est que l'usage du C fait sens pour des personnes qui pensent comme un ordinateur. Je crois que la raison pour laquelle il en est ainsi est que les personnes qui ont conçu le langage C l'ont fait à un moment où les compilateurs devaient être simples ; à un moment où le langage devait être adapté à la sortie ou au résultat attendu. Donc lorsque je lis du code en langage C, je sais à quoi va ressembler le code assembleur et c'est ce qui m'intéresse. (extraits de 2012)

Le langage C dans l'embarqué

Connaître le langage C reste un pré-requis pour qui souhaite développer des systèmes embarqués en programmation Bare Metal (sans OS) comme sur les Microchip PIC ou STM32. Sur les systèmes avec un OS embarqué comme le Raspberry Pi, on préfère souvent pour des questions de rapidité de développement, programmer dans un langage de plus haut niveau (C++ / Python) mais Les drivers restent développés en C.

Classement TIOBE des langages de programmation

L'index TIOBE mesure la popularité des langages de programmation en se basant sur le nombre de pages web retournées par les principaux moteurs de recherche lorsqu'on leur soumet le nom du langage de programmation.

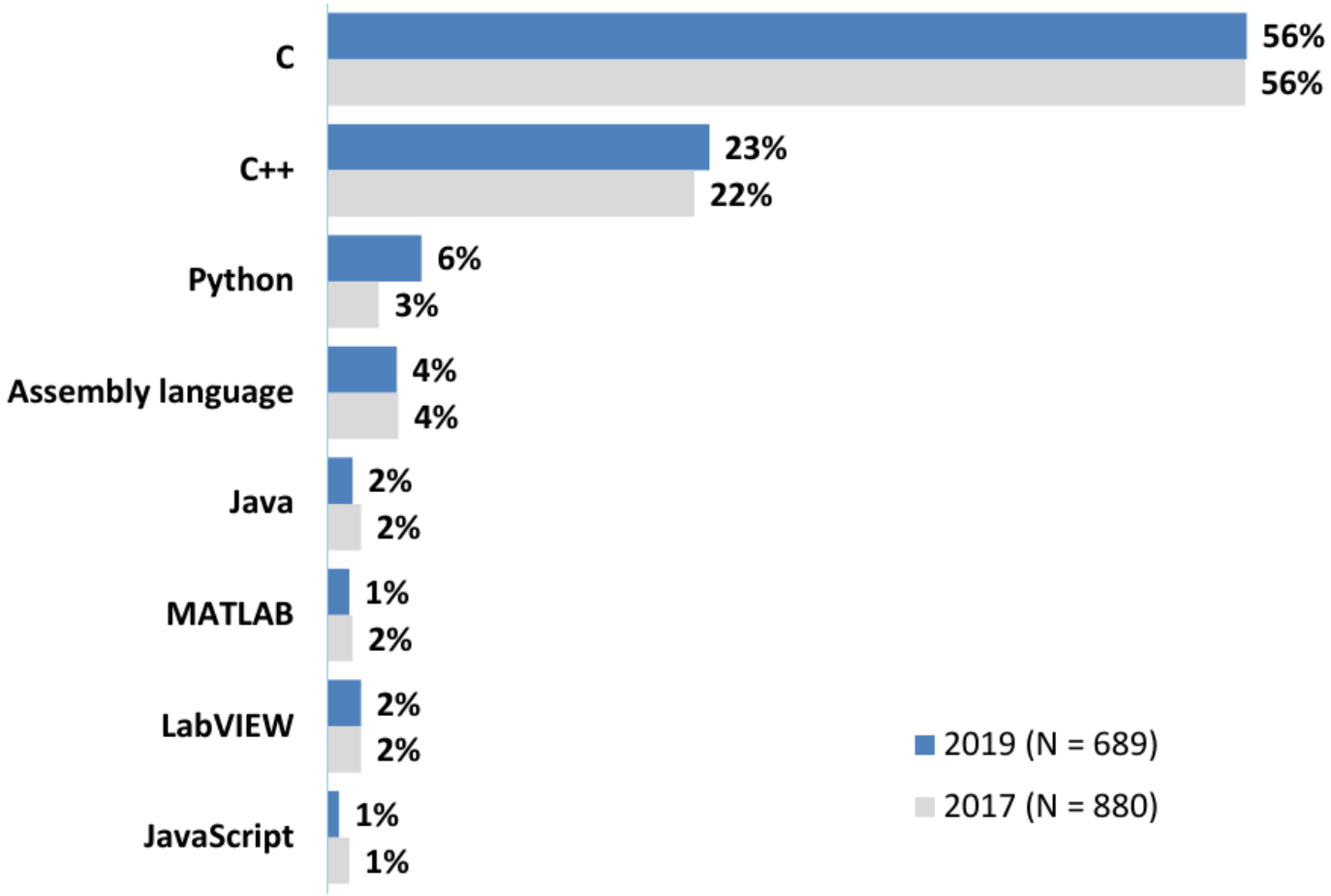


Le langage C (en noir) continue depuis 50 ans à truster les 2 premières places du classement. En 2022, le langage C se situe derrière Python,

Aug 2022	Aug 2021	Change	Programming Language	Ratings	Change
1	2	^	Python	15.42%	+3.56%
2	**1**	v	**C**	**14.59%**	**+2.03%**
3	3		Java	12.40%	+1.96%
4	4		C++	10.17%	+2.81%
5	5		C#	5.59%	+0.45%

Classement Embedded Markets des langages de programmation

Dans l'enquête de 2019 Embedded Markets Study pour les systèmes embarqués et IoT, le langage C est largement premier avec 56%, suivi de C++ avec 23%. Python arrive loin derrière avec 6%.



Les concurrents du C dans l'embarqué

- Rust conçu pour la sécurité et les hautes performances. Il est souvent décrit comme l'un des successeurs potentiels de C et C++ et commence à être utilisé sur des cibles STM32.
- Micropython est un portage du langage Python 3 incluant un sous-ensemble de la bibliothèque standard Python et prévu pour fonctionner sur certains micro-contrôleurs 32 bits (STM32, Raspberry Pi Pico, ESP32 ...)
- Le langage GO et tinyGo est parfois utilisé comme preuve de concept sur système embarqué. Il reste plus utilisé sur des projets en concurrence avec le C++ (Docker ou Grafana sont écrits en GO)
- Kotlin est utilisé pour Les applications Android (historiquement Java). Les couches basses (Kernel, Runtime) restent en C/C++.

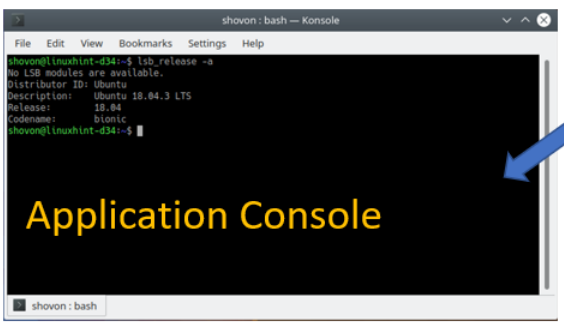
Chaque décennie apporte son nouveau langage censé remplacer le C, mais force est de constater que le langage C, qui fête ses 50 ans en 2022, n'a pas fini de nous étonner par sa longévité et pertinence dans l'embarqué.

Pertinence du C comme langage de programmation

1. Est-il pertinent d'apprendre le C en 2022 ?
   - Oui pour de la programmation de micro-contrôleurs et pour l'embarqué
   - Oui, la notion de pointeur se retrouve dans les automates, variateurs de fréquence,
   - Oui pour la programmation de Drivers / modules du Kernel Linux
2. Est-il pertinent d'apprendre le C en Génie Électrique ?
   - Oui, voir 1.
3. Quels autres langages apprendre ?
   - pour l'IA, la science des données : Python 3 est très utilisé pour sa simplicité d'utilisation. Dans les faits, on utilise Python pour manipuler des fonctions performantes écrites en C/C++ sans avoir à maîtriser le C++.
   - pour l'embarqué avec FPGA : Verilog ou VHDL

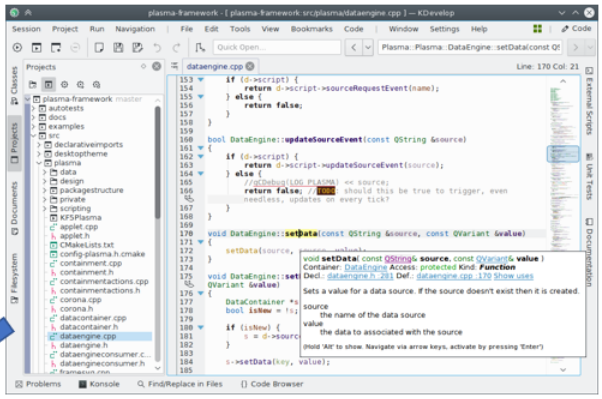
Développer en C, la chaîne de compilation

Ordinateur de développement et d'exécution. Dispose d'une chaîne de compilation.



Développement C / C++ classique

Environnement de développement intégré (IDE)



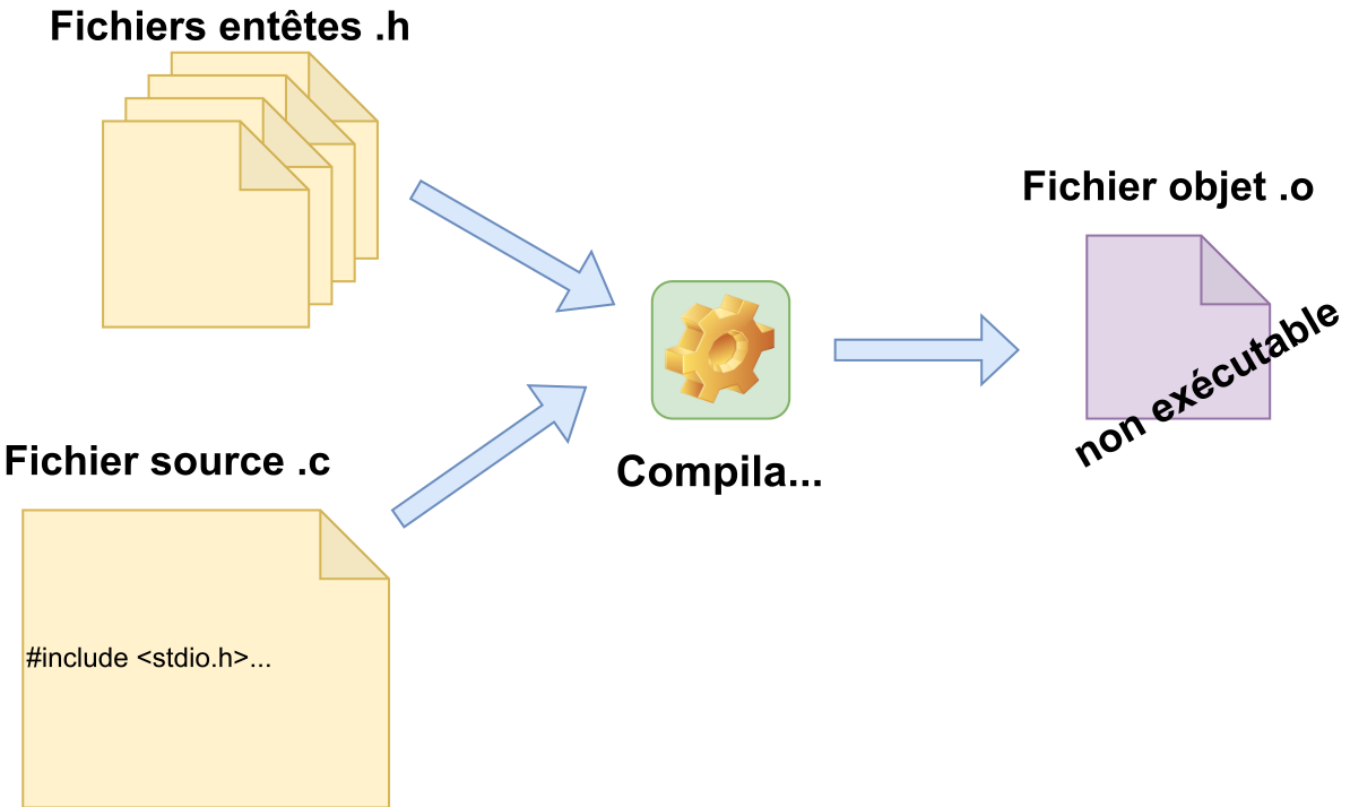
Compilation et exécution (débogage, ...)

La Toolchain

1. L'éditeur de texte : permet de créer et de modifier les fichiers *sources*
2. Le compilateur : qui permet de passer d'un fichier source à un fichier objet. Cette transformation se fait en réalité en plusieurs étapes grâce à différents composants (préprocesseur C, compilateur, assembleur).

Remarques :

- Compiler n'est pas « construire un fichier exécutable »
- Produire un fichier exécutable = compilation + édition des liens (link) + ...



Le fichier **source :** est un fichier texte contenant le code C du programme

Le fichier **objet :** est un fichier binaire contenant les instructions (code machine) et données provenant du processus de traduction du langage

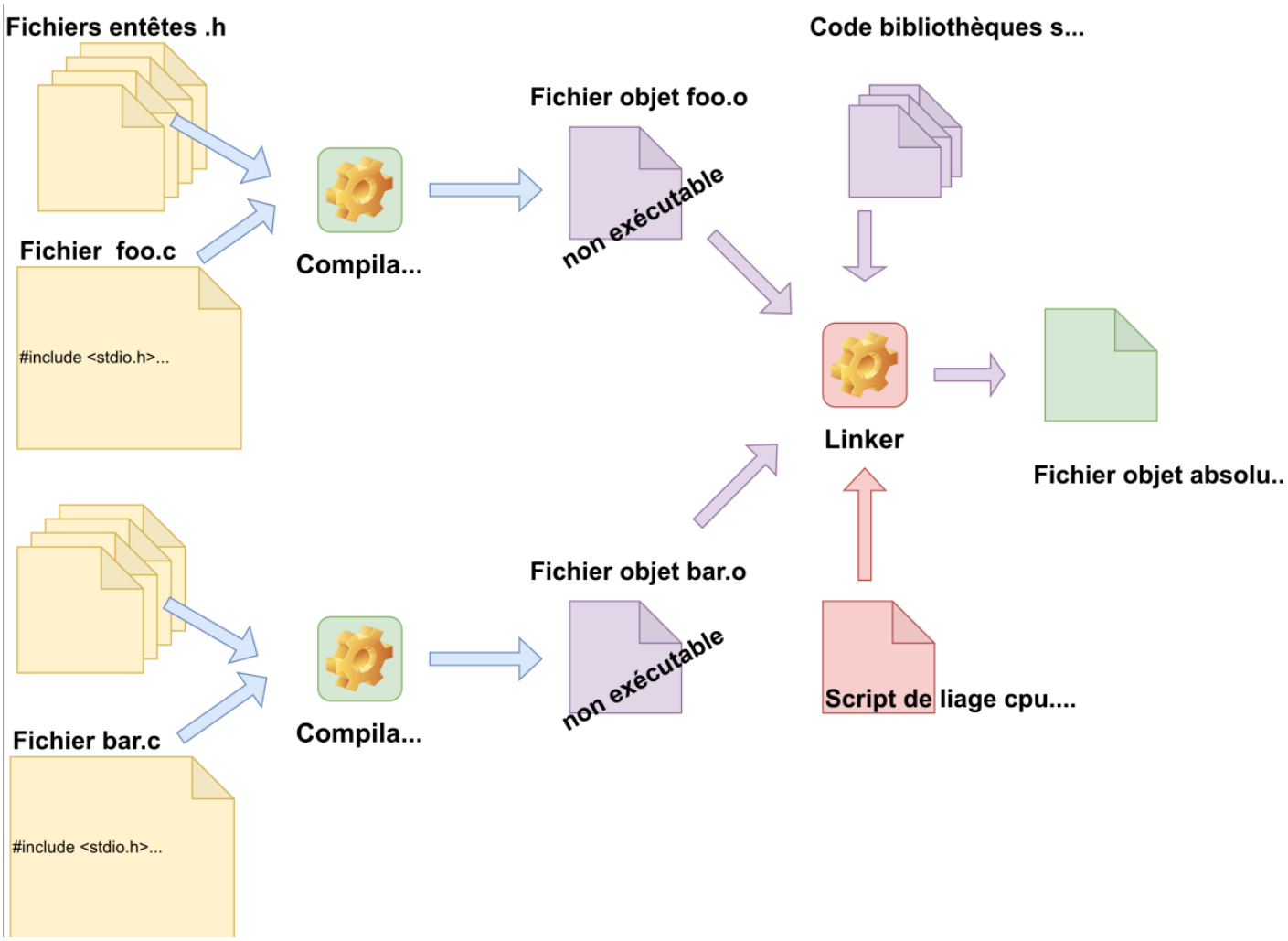
- Contient le code exécutable par un processeur mais fichier non exécutable
- Organisé en sections (.text, .data, .bss, .debug, ...)
- Différents formats : ELF, COFF, PE, ...

Pour obtenir un fichier exécutable, il manque : **l'édition des liens et la localisation**

- Les liens vers les fonctions et variables externes (symboles)
- Le positionnement des fonctions et variables dans les mémoires

- Les étapes de lancement et de fin d'exécution

3. L'éditeur de liens (Linker), qui assure le regroupement des fichiers objet provenant des différents modules et les associe avec les bibliothèques utilisées pour l'application. Nous obtenons ici un fichier exécutable.



4. Locator: localisation des fonctions et variables
5. Libraries : bibliothèques

- pour interfacer l'environnement (stdlib, newlib): printf, malloc, ...
- Pour les fonctions usuelles (math.h, ...)

6. Le débogueur, qui peut alors permettre l'exécution pas à pas du code, l'examen des variables internes, etc. Pour cela, il a besoin du fichier exécutable et du code source.

Notons également l'emploi éventuel d'utilitaires annexes travaillant à partir du code source, le vérificateur de code, les enjoliveurs (beautifier), les outils de documentation automatique, etc.

ToolChain manuelle vs IDE

On remarque que la chaîne de compilation fait intervenir de nombreux outils et peut rapidement devenir complexe si le programme est constitué de nombreux fichiers source et de bibliothèques :

Deux écoles de programmeurs coexistent :

- ceux qui préfèrent disposer d’un environnement intégrant tous les outils de développement, on parle d'EDI pour Environnement de Développement Intégré ou d'IDE en anglais.
- ceux qui utilisent les différents utilitaires de manière séparée, configurant manuellement un fichier Makefile pour recompiler leur application.

Les IDE permettent de configurer le compilateur de générer automatiquement un Makefile, de réaliser la mise en format du code et générer documentation, etc. Ils sont très efficace, mais pour le débutant, on occulte les différentes étapes qui permettent de générer le programme. Comprendre à minima ces étapes vont nous aider pour la cross-compilation ou la résolution de bugs générés par les automatismes de l'IDE.

Les élements de la chaîne de compilation

L'éditeur de texte :

Quelques éditeurs :

- VSCode : un bon choix multiplateforme. VSCode est cependant un hybride car l'ajout de plugins le transforme en IDE pour tous les langages.
- Notepad++ : éditeur très rapide pour Windows
- Gedit ou Kate : les éditeurs pour Linux en version Gnome ou KDE
- Vim / Emacs : les historiques, puissants et complexes à maîtriser, ont perdu grandement de leur intérêt depuis quelques années.
- Nano : quand on est obligé d'éditer du code en mode Console, plus simple que Vim.

Le compilateur :

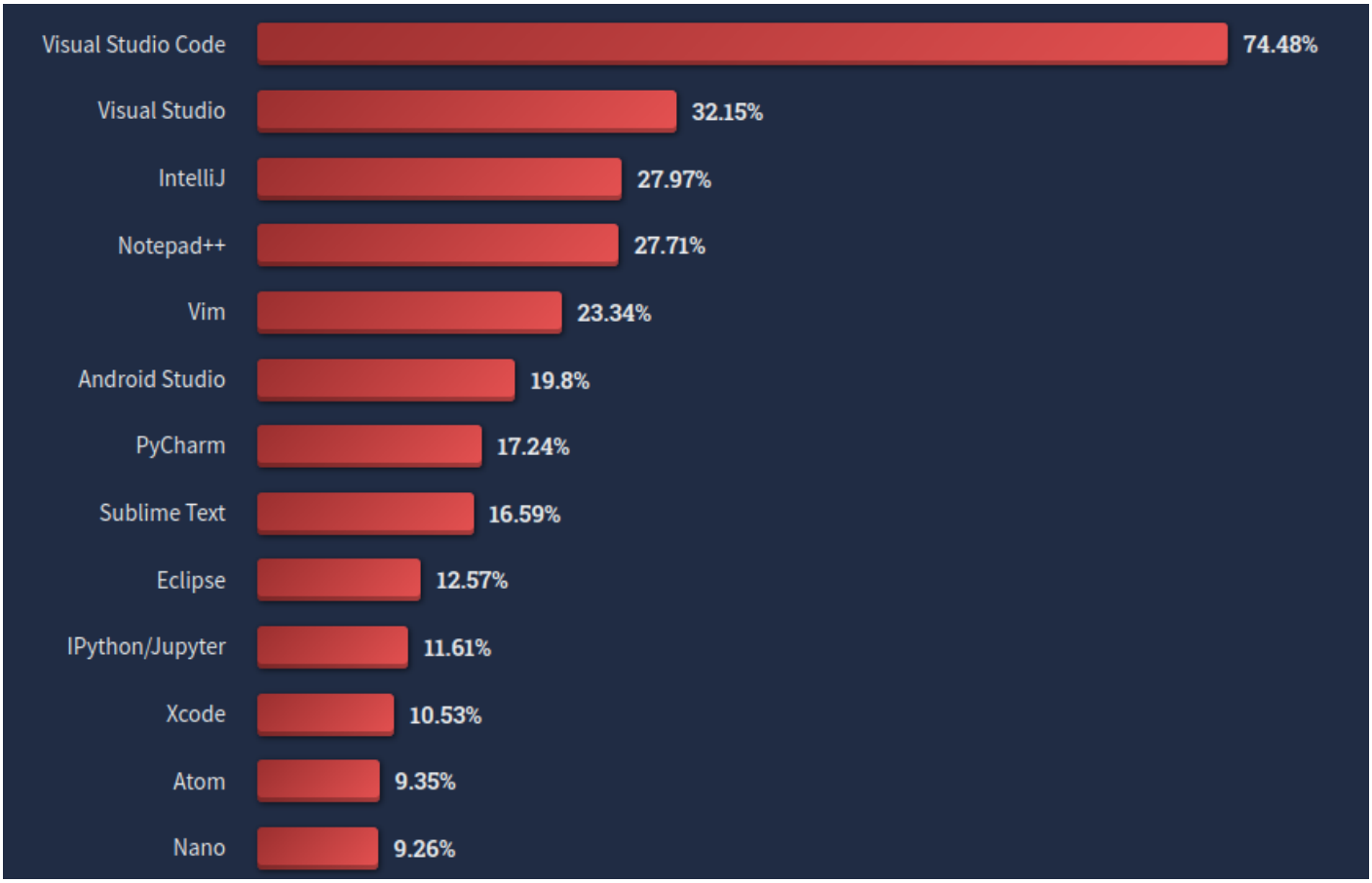
- GCC : pour GNU Compiler Collerction qui contient gcc (en minuscule) qui est le gnu c compiler pour le C, g++ pour le c++, gccgo pour GO, ... GCC fonctionne sur Linux et BSD
- MinGW[-w64] : est un portage de GCC pour un environnement Windows
- Microsoft Visual C++ : compilateur C/C++ de Microsoft
- LLVM et CLANG: interface de compilation C, C++, Objectif C, alternative à GCC.

Les autres outils :

- Make : un utilitaire qui permet d’automatiser la compilation. Pour fonctionner, il a besoin d’un fichier Makefile qui contiendra la cible, les dépendances et les commandes à exécuter.
- CMake : un moteur de production (build automaton) de plus haut niveau que Make ou SCons. Il est à placer au même niveau qu'Autotools puisqu'il permet de générer des Makefiles.
- GDB (GNU Debugger) : le débogueur standard du projet GNU.
- Doxygen : pour générer de la documentation (format HTML ou LaTeX par exemple) pour plusieurs langages de programmation (C, C++, Java, VHDL...).
- Git : logiciel de gestion de versions décentralisé, le standard du marché, inventé par Linus Torvalds également.

Les IDE

Les IDE possèdent l'avantage d'intégrer l'ensembles de outils nécessaires au développement (éditeur, compilateur, debugger, ...) souvent optimisé pour un langage donné. Le graphique ci-dessous représente les parts d'utilisation des différents IDE et éditeurs de textes configurables selon un sondage de 2022 du site StackOverflow.



On remarque que VSCode est en tête, les plugins disponibles permettent de transformer l'éditeur de texte en IDE et de l'adapter à de nombreux langages de programmation (pareil pour Notepad++, Vim, Sublim Text).

Ce sondage est cependant à prendre avec beaucoup de hauteur car si l'on regarde en fonction des langages de programmation:

- PyCharm est très utilisé en Python
- Visual Studio est le standard quand on veut développer dans les technos Microsoft,
- Eclipse ou IntelliJ pour du Java (Android Studio est basé sur IntelliJ)

Les fabriquants de micro-contrôleurs se basent sur ces IDE pour développer leurs outils de programmation, par exemple:

- STM32CubeIDE est basé sur Eclipse pour programmer les STM32 en C
- l'Arduino IDE 2.0 est basé sur Eclipse également
- MPLAB X est basé sur NetBeans (IDE pour Java) pour programmer les PIC en C
- VSCode + le Plugin PlateformIO est utilisé pour programmer les ESP32

IDE pour le langage C

A l'IUT, le choix s'est porté sur l'IDE Code::Blocks avec le compilateur MinGW pour Windows :

- logiciel libre -> gratuit
- multiplateforme
- léger et facile à installer
- simplicité d'utilisation qui le rend particulièrement adapté pour l'apprentissage.

Pour la programmation en C sur Linux, vous pouvez utiliser VirtualBox sur Windows et y installer Ubuntu. Dans Ubuntu, il restera à installer le paquet build-essential pour avoir accès aux compilateurs et VSCode si vous souhaitez travailler avec cet éditeur.

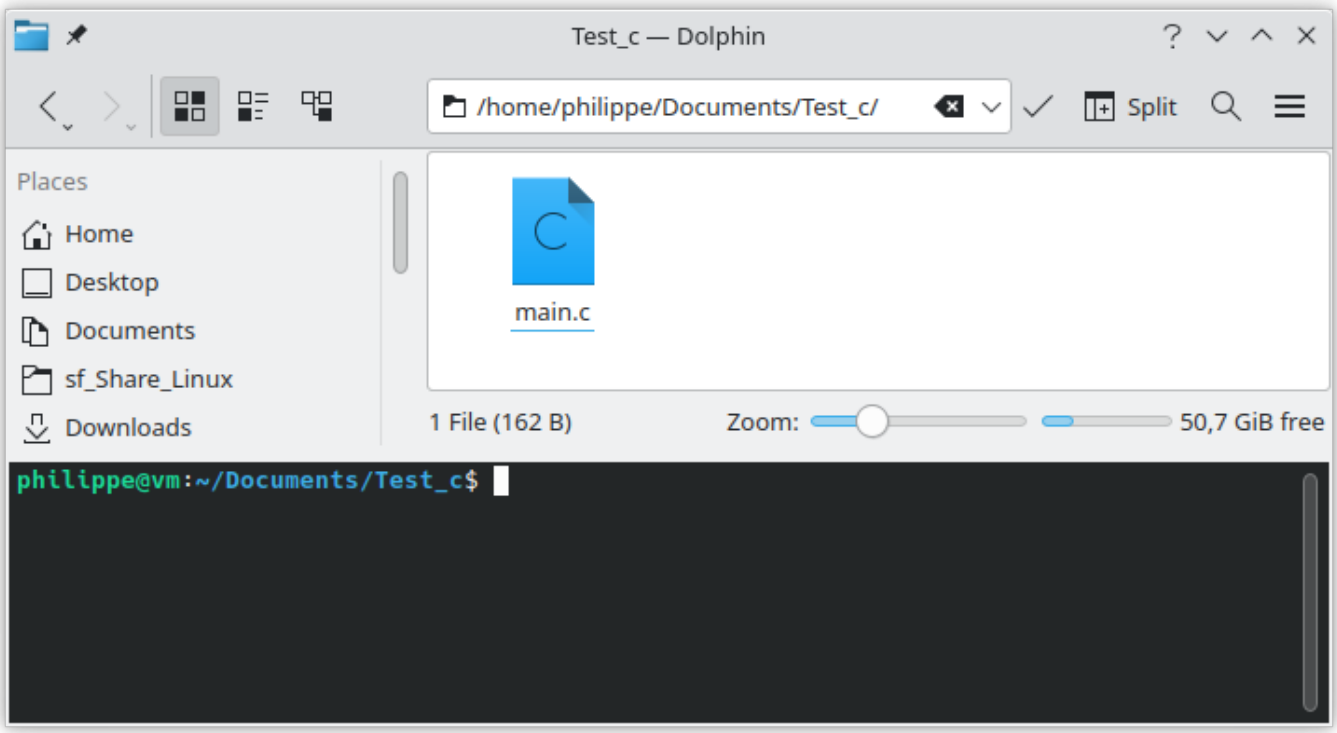
Premier programme en C et Compilation sous Linux

En hommage au Livre "The C programming Langage de K&R", il est de tradition de commencer l'apprentissage de la programmation en affichant Hello world! sur l'écran.

```
/*
Date    : 2022 08 09
Auteur  : Philippe Celka
Nom     : main.c
Résumé  : Affiche Hello world! à l'écran
*/

#include <stdio.h> //Bibliothèque pour la fonction printf

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

Compilation et Édition de liens

Dans cet exemple, le code source hello.c est enregistré dans le répertoire /home/philippe/Documents/Test_c/



Nous pouvons nous placer dans ce répertoire et ouvrir un terminal Linux. Nous utiliserons gcc pour compiler notre programme et générer le fichier exécutable.

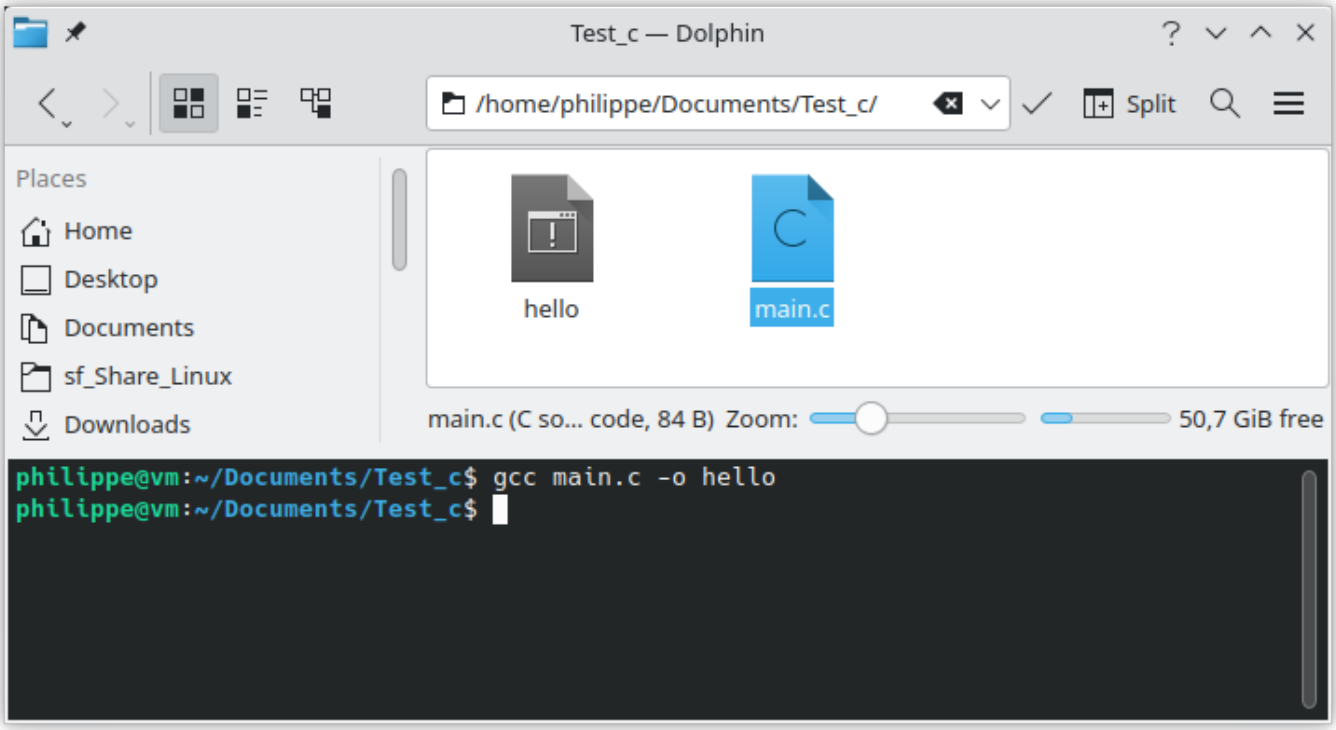
```
GCC(1)                                    GNU                                    GCC(1)

NAME
       gcc - GNU project C and C++ compiler

SYNOPSIS
       gcc [-c|-S|-E] [-std=standard]
           [-g] [-pg] [-Olevel]
           [-Wwarn...] [-Wpedantic]
           [-Idir...] [-Ldir...]
           [-Dmacro[=defn]...] [-Umacro]
           [-foption...] [-mmachine-option...]
           [-o outfile] [@file] infile...

       Only the most useful options are listed here; see below for the remainder.  g++ accepts mostly the same
       options as gcc.

Manual page gcc(1) line 1 (press h for help or q to quit)
```

gcc s'utilisera de la manière suivante :

gcc [fichier c à compiler] -o [exécutable]

-o correspond à l'option outfile et permet de spécifier le nom pour le fichier de sortie.

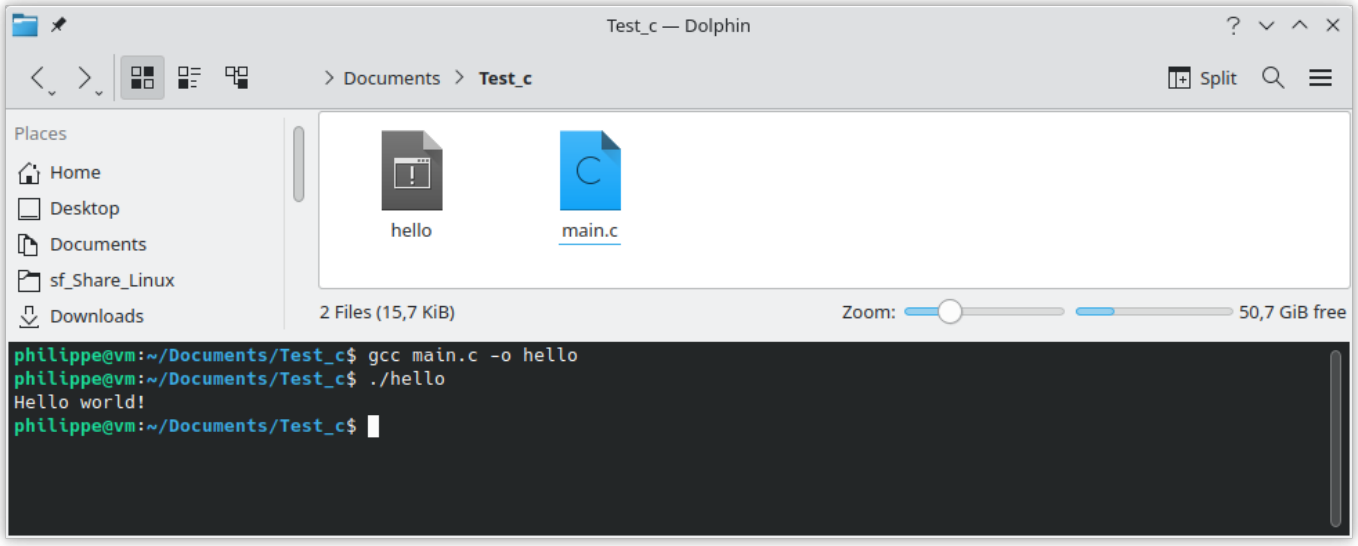
```
gcc main.c -o hello
```



On remarque que dans le répertoire, un nouveau fichier apparaît, un fichier exécutable.

Pour exécuter ce fichier, depuis le terminal, nous utiliserons la commande

```
./hello
```



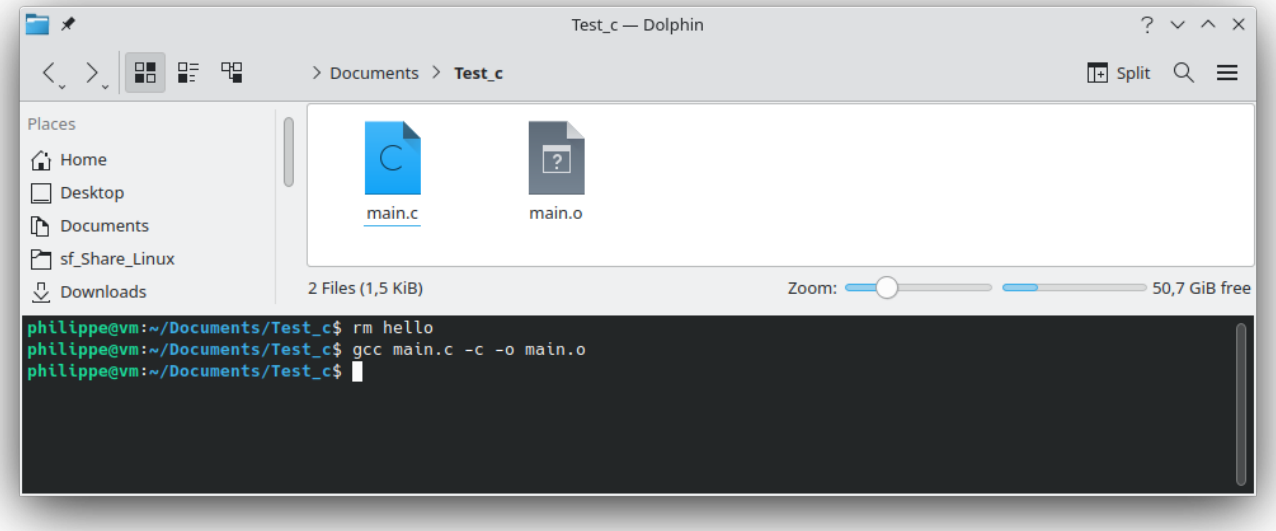
Hello world! s'affiche dans la console, vous avez réalisé votre premier programme.

Remarque : gcc dans ce cas de figure, n'effectue pas seulement la compilation mais également l'édition des liens pour générer l'exécutable.

Compilation seule et fichier objet

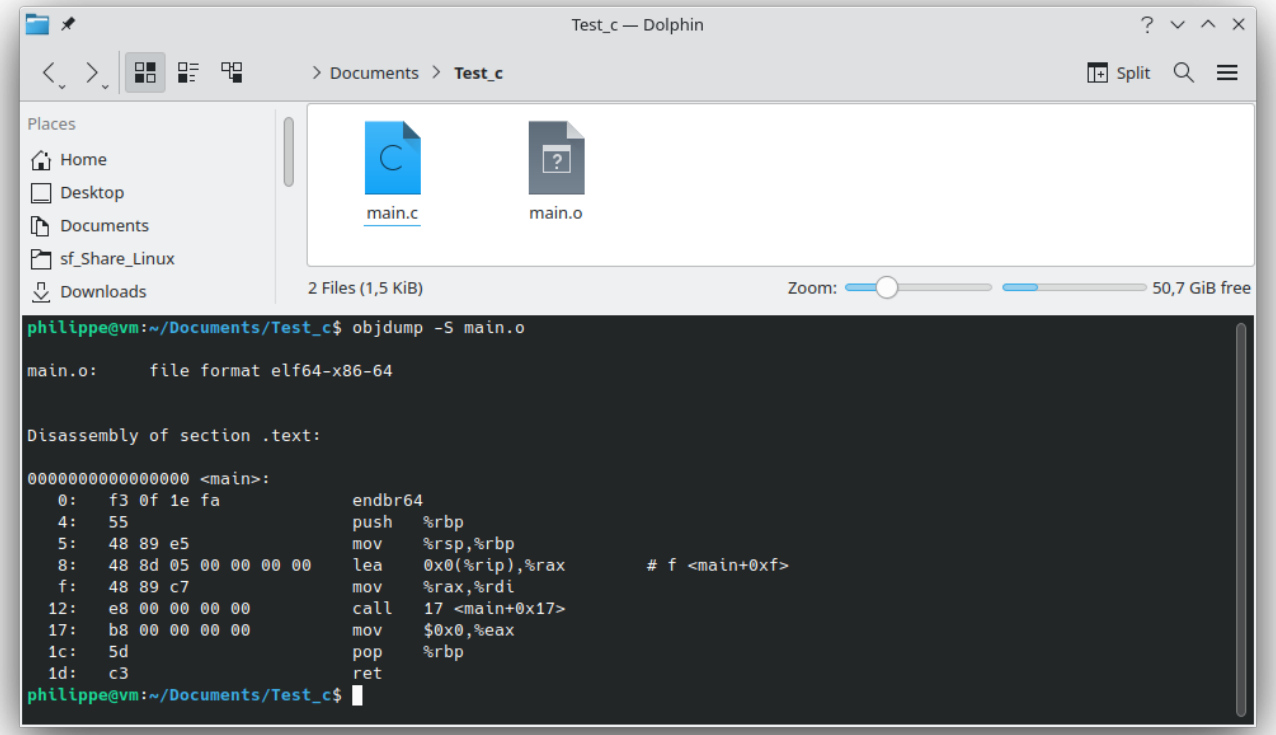
Pour utiliser uniquement le compilateur de gcc et générer le fichier objet, on utilise l'option -c

```
gcc main.c -c -o main.o
```



Pour afficher le contenu du fichier objet, nous utiliserons objdump

```
objdump -S main.o
```



Nous obtenons le code machine et le code assembleur suivant :

```
main.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <main>:
   0: f3 0f 1e fa           endbr64
   4: 55                    push   %rbp
   5: 48 89 e5              mov    %rsp,%rbp
   8: 48 8d 05 00 00 00 00  lea    0x0(%rip),%rax        # f <main+0xf>
   f: 48 89 c7              mov    %rax,%rdi
  12: e8 00 00 00 00        call   17 <main+0x17>
  17: b8 00 00 00 00        mov    $0x0,%eax
  1c: 5d                    pop    %rbp
  1d: c3                    ret
```

Il s'agit d'une section du code. Le linker réalisera ensuite le lien entre les différents fichiers objet et bibliothèques pour générer l'exécutable. (Je ne développerai pas ces opérations dans cet article)

Compilation et édition de liens avec plusieurs fichiers

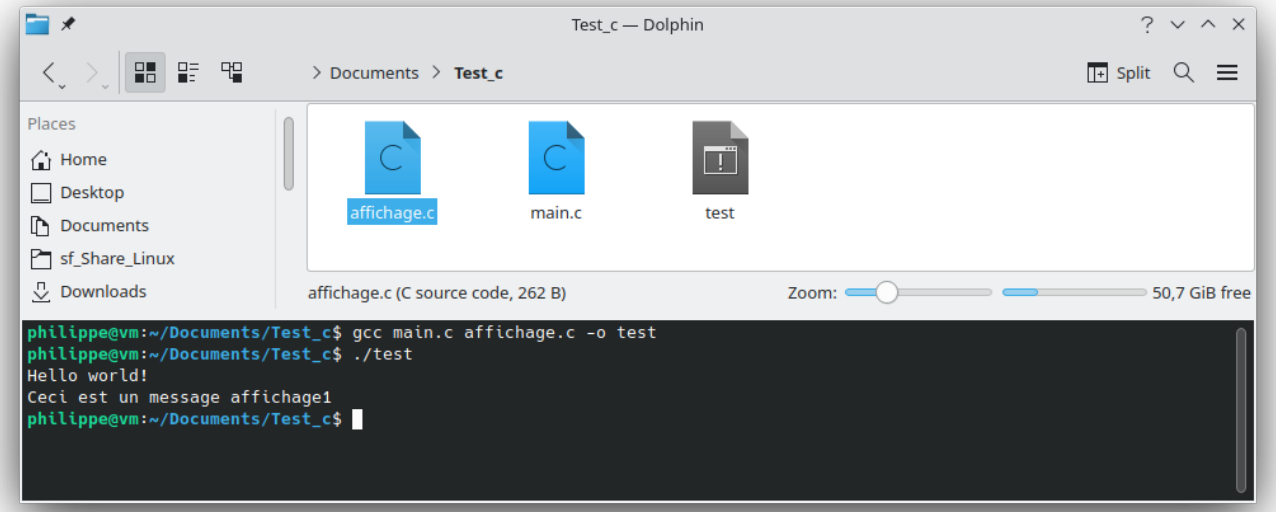
Nous souhaitons maintenant compiler deux fichiers c pour obtenir un exécutable :

```
/*
Nom     : main.c
*/

#include <stdio.h>

int affichage1(void); //  prototype de la fonction affichage1

int main(void)
{
  printf("Hello world!  \n");
  affichage1();  // appel de la fonction affichage1
  return 0;
}
```

```
/*
Nom     : affichage.c
*/

#include <stdio.h> //Bibliothèque pour la fonction printf

int affichage1(void)
{
  printf("Ceci est un message affichage1  \n");
  return 0;
}
```

Pour générer l'exécutable, gcc va compiler main.c pour en faire un main.o (objet), affichage.c devient un affichage.o et le linker va lier les deux fichiers pour générer l'exécutable test.

```
gcc main.c affichage.c -o test
```



Fichier .h

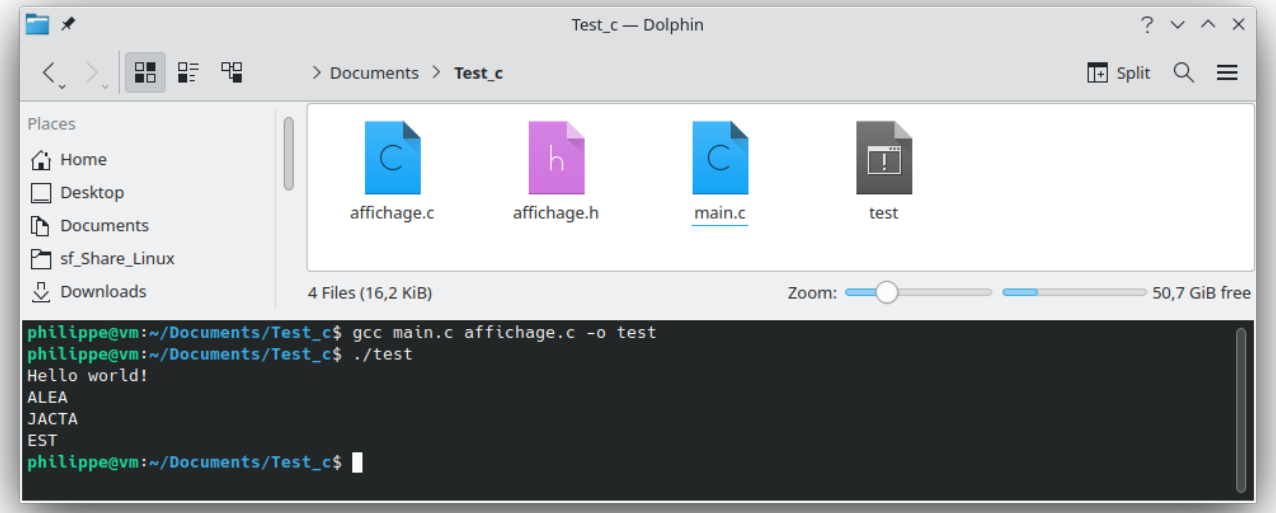
```
//Nom     : main.c
#include <stdio.h>
#include "affichage.h" //fichier header

int main(void)
{
  printf("Hello world! \n");
  affichage1();
  affichage2();
  affichage3();
  return 0;
}
```

```
//Nom     : affichage.c

#include <stdio.h> //Bibliothèque pour la fonction printf

int affichage1(void)
{
```

```
  printf("ALEA  \n");
  return 0;
}

int affichage2(void)
{
  printf("JACTA \n");
  return 0;
}

int affichage3(void)
{
  printf("EST  \n");
  return 0;
}
```

```
//Nom     : affichage.h

#ifndef   _AFFICHAGE_H_
#define   _AFFICHAGE_H_

int affichage1(void);
int affichage2(void);
int affichage3(void);

#endif
```

```
gcc main.c affichage.c -o test
```



ALEA, JACTA, EST s'affiche correctement, notre chaîne de compilation fonctionne avec plusieurs fichiers.

Pour le moment, nous avons deux fichiers c à compiler. Un programme complet va en contenir plusieurs dizaines ou beaucoup plus. Le moteur de jeu comme Unity dépasse le Million de lignes de code, des centaines de fichiers, il est alors nécessaire de structurer le projet et d'automatiser le processus de compilation.

Présentation de Makefile

Présentation

Un Makefile est un fichier constitué de plusieurs règles de la forme :

Sélectionnez

```
cible: dependance
    commandes
```

Chaque commande est précédée d'une **tabulation**. Lors de l'utilisation d'un tel fichier via la commande make la première règle rencontrée, ou la règle dont le nom est spécifié, est évaluée.

L'évaluation d'une règle se fait en plusieurs étapes :

- Les dépendances sont analysées, si une dépendance est la cible d'une autre règle du Makefile, cette règle est à son tour évaluée.

- Lorsque l'ensemble des dépendances est analysé et si la cible ne correspond pas à un fichier existant ou si un fichier dépendance est plus récent que la règle, les différentes commandes sont exécutées.

###Makefile minimal

Pour notre exemple, le makefile minimal est :

```
test: affichage.o main.o
  gcc -o test affichage.o main.o

affichage.o: affichage.c
  gcc -o affichage.o -c affichage.c

main.o: main.c affichage.h
  gcc -o main.o -c main.c -W
```

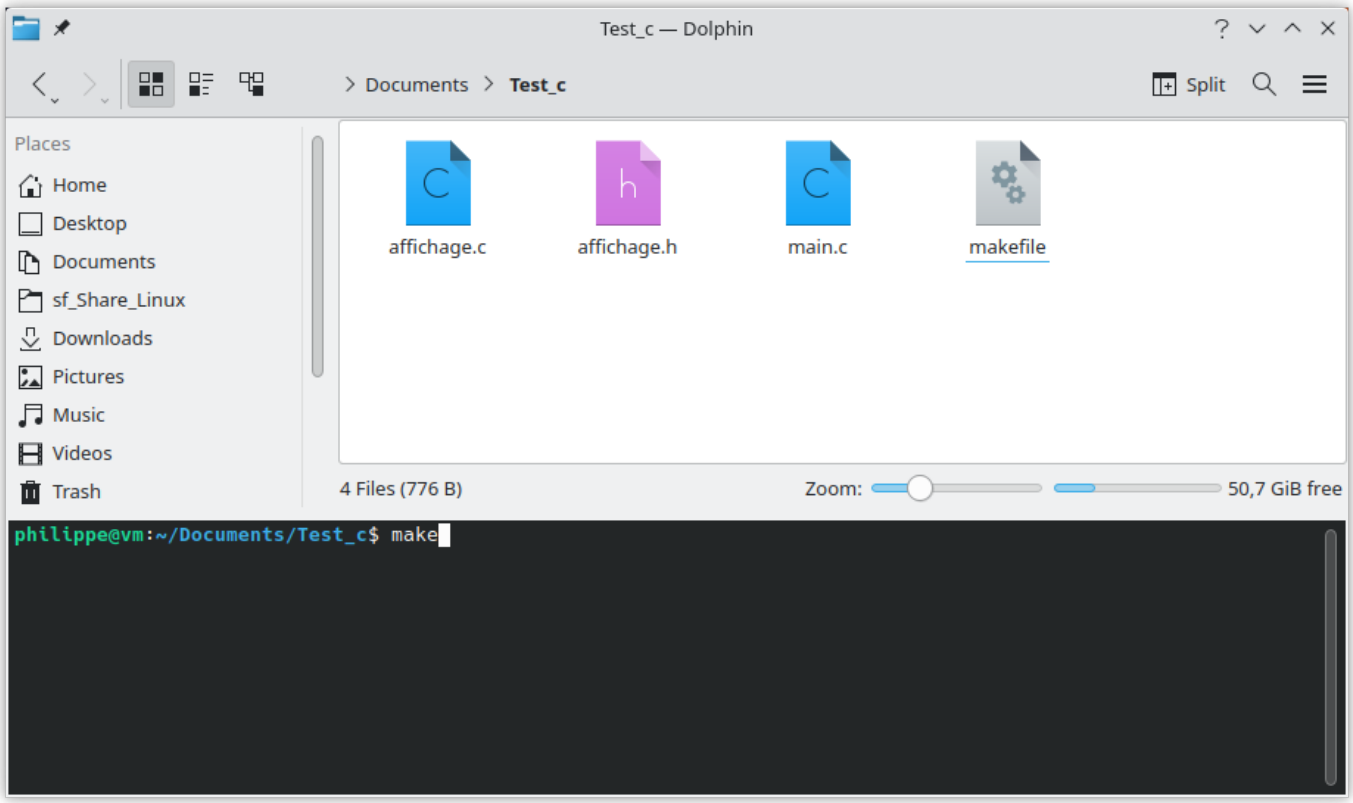
Regardons comment fonctionne ce Makefile :

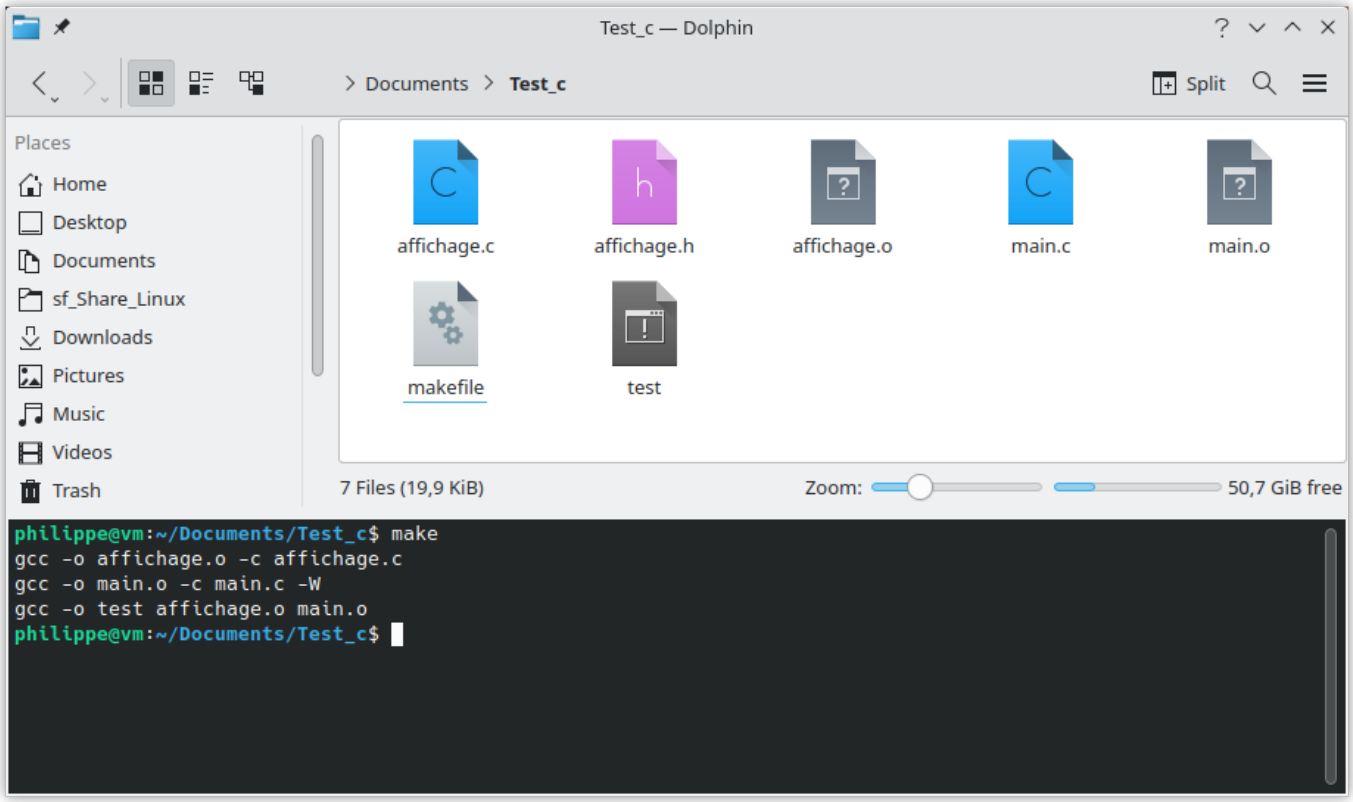
Nous cherchons à créer le fichier exécutable test, la première dépendance est la cible d'une des règles de notre Makefile, nous évaluons donc cette règle. Comme aucune dépendance de affichage.o n'est une règle, aucune autre règle n'est à évaluer pour compléter celle-ci.

Deux cas se présentent ici : soit le fichier affichage.c est plus récent que le fichier affichage.o, la commande est alors exécutée et affichage.o est construit, soit affichage.o est plus récent que affichage.c est la commande n'est pas exécutée. L'évaluation de la règle affichage.o est terminée.

Les autres dépendances de test sont examinées de la même manière puis, si nécessaire, la commande de la règle test est exécutée et test est construit.

On crée le makefile avec un éditeur de texte, en faisant attention d'utiliser des tabulations et non des espaces pour les règles. On lance la commande make qui va exécuter les règles de compilation issues du makefile.





Makefile "enrichi"

Plusieurs cas ne sont pas gérés dans l'exemple précédent :

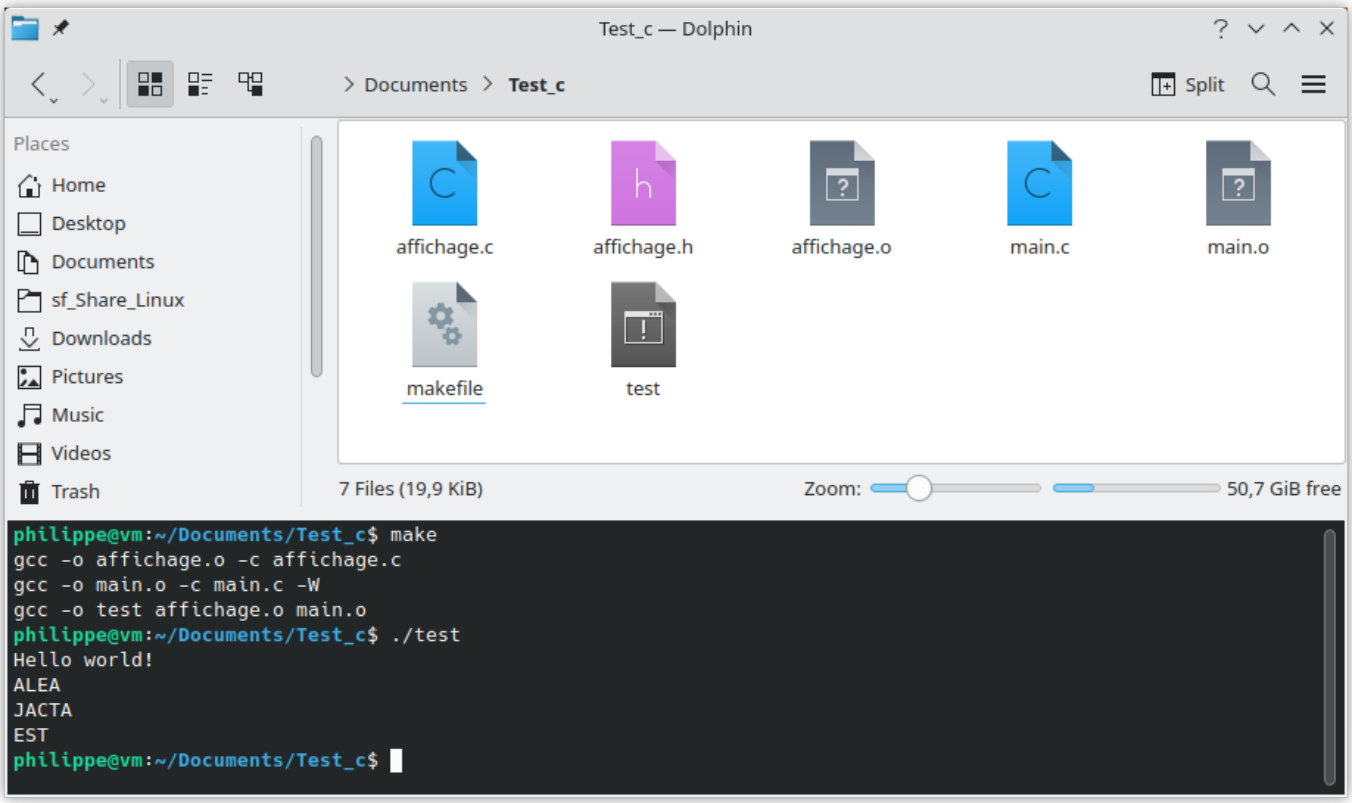
- Un tel Makefile ne permet pas de générer plusieurs exécutables distincts.
- Les fichiers intermédiaires restent sur le disque dur même lors de la mise en production.
- Il n'est pas possible de forcer la régénération intégrale du projet

Ces différents cas conduisent à l'écriture de règles complémentaires :

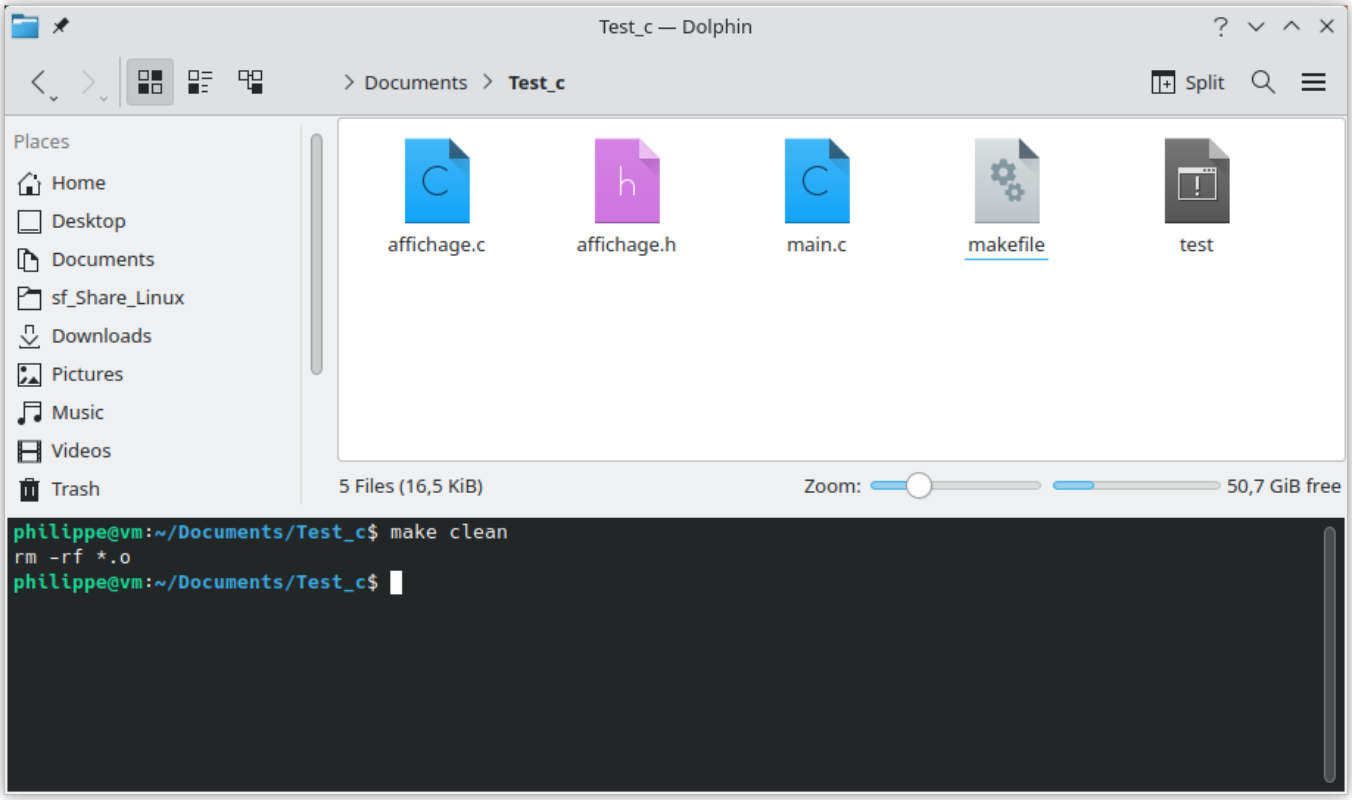
- all : généralement la première du fichier, elle regroupe dans ces dépendances l'ensemble des exécutables à produire.
- clean : elle permet de supprimer tous les fichiers intermédiaires.
- mrproper : elle supprime tout ce qui peut être régénéré et permet une reconstruction complète du projet.

En ajoutant ces règles complémentaires, notre Makefile devient donc :

```
all: test

test: affichage.o main.o
  gcc -o test affichage.o main.o

affichage.o: affichage.c
  gcc -o affichage.o -c affichage.c

main.o: main.c affichage.h
  gcc -o main.o -c main.c -W

clean:
    rm -rf *.o

mrproper: clean
    rm -rf test
```



Après avoir généré l'exécutable, on peut lancer la commande make clean pour supprimer les fichiers objets.



La réalisation d'un makefile mérite un article spécifique. On n'a même pas affleuré les possibilités de cet outils. L'objectif de cette présentation de makefile était de présenter les différentes étapes que gère un IDE quand vous appuyez sur le bouton play.

Conclusion

Compiler un programme avec les outils gcc et make peut rapidement devenir complexe avec l'augmentation du nombre de fichiers.

L'utilisation d'un IDE comme Code::Blocks va générer de manière transparente le bon makefile pour compiler votre projet. Par contre, comme dans tout automastisme, il y a des cas de figures où cela ne fonctionne pas. Maîtriser à minima les différentes étapes de compilation permet de mieux comprendre les erreurs remontées par l'IDE et trouver une solution.

Lorsque l'on appuye sur le bouton play de l'IDE, de nombreuses étapes sont effectuées en toute transparence pour l'utilisateur pour générer l'exécutable. Le programmeur peut se concentrer sur la qualité de son code.

Pour le développement des systèmes embarqués, maîtriser un peu plus en profondeur les étapes de compilation est nécessaire.

Le C pour l'embarqué

Avant-propos

L'objectif de cette série d'articles sur le langage C, n'est pas de faire un cours exhaustif, mais de réaliser un focus sur certains aspects techniques, pas toujours suffisament développés dans les cours, mais nécessaires quand on travaille sur des systèmes embarqués.

Pour traiter les exemples présentés, nous pouvons utiliser un simulateur de code C en ligne https://www.onlinegdb.com/online_c_compiler.

Manipulation sur les Bits

La technique du masquage consiste à modifier individuellement des bits sans affecter les autres bits.

- sur un octet, on est sur 8 bits, bit#7 ... bit#0
- sur un int, on est sur 4 octets, bit#31 ... bit#0

Opérateurs de manipulation de bits

```
&      // AND bit à bit
|      // OR bit à bit
^      // OU Exclusif bit à bit
~      // NOT bit à bit (complément à 1)
<<     // décalage à gauche
>>     // décalage à droite
```

- Dans l'exemple ci-dessous, j'utilise le type uint8_t pour travailler sur 8 bits en non-signé
- Pour utiliser uint8_t, il faut inclure <stdint.h>
- 0x18 correspond à la valeur 18 en hexadécimal, soit 24 en décimal, ou 0b0001 1000 en binaire.
- Pour afficher le contenu d'un uint8_t dans un printf on utilise %hhx à la place de %d pour un int, pour afficher la valeur en hexa.

Exemple d'utilisation

A partir de ce programme C, essayez dans un premier temps d'anticiper les résultats à obtenir avant d'éxecuter le code.

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

int main()
{
  uint8_t A=0x18,B=0x57, Res;
  Res=A&B;
  printf("A=0x %hhx & B=0x %hhx donne Res=0x %hhx \n",A,B,Res);
  Res=B>>3;
  printf("Un décalage à droite de B>>3 donne Res=0x %hhx \n",Res);
  Res=~A;
  printf("Le complément à 1 de A donne  Res=0x %hhx \n",Res);
  return 0;
}
```

Solution

- A=0x18 & B=0x57 donne Res=0x10, en binaire 0b0001 1000 & 0b0101 0111 donne 0b0001 0000 soit 0x10
- Un décalage à droite de B>>3 donne Res=0xa, en binaire 0b0101 0111 décalé de 3 à droite donne 0b0000 1010 soit 0x0a ou 0xa
- Le complément à 1 de A donne Res=0xe7, en binaire, le complément à 1 de 0b0001 1000 donne 0b1110 0111 soit 0xe7

Rappel: un décalage de 1 bit vers la gauche correspond à une multiplication par 2, 2 bits à gauche correspond à une multiplication par 4 .... Pour un décalage à droite, le principe est identique mais engendre une division.

Bit SET

Pour faire une mise à 1 de bit (SET), nous utiliserons l'operateur OR qui est représenté par `|` et qui effectue une opération OR bit à bit.

```
// Nous voulons faire un SET du Bit#7 d'un registre nommé: REG
REG = REG |  0x80;

// Pour fair un SET du bit#31:
```

```
REG = REG |  0x80000000;

// Simplifions :
// (1 << 31) signifie que 1 sera décalé à gauche 31 fois pour produire 0x80000000
REG = REG | (1 << 31);

// Simplifions et compactons:
REG |= (1 << 31);

// Un SET du Bit#21 et du Bit#23:
REG |= (1 << 21) | (1 << 23);
```

Exemple

```
#include <stdio.h>

int main()
{
  int REG=5;
  printf("Au départ REG = %d\n",REG);
  REG |= (1<<7);
  printf("REG |= (1<<7) = %d",REG);
  return 0;
}
```

Solution

- REG = 8 soit en binaire 0b0000_0101
- on met le bit#7 à 1
- REG = 0b1000_0101 soit 133 en décimal

Bit CLEAR

Pour mettre un bit à 0, on parle de Reset ou de Clear.

- Une fonction AND (ET) notée `&` sera utilisée en association avec 0 pour effectuer une mise à 0.
- Une fonction AND associée avec 1 ne change rien (élément neutre)

L'opérateur tilde (`~`) peut nous aider pour simplifier cette opération

- pour rappel, l'opérateur tilde (`~`) correspond à la fonction NOT

```
// Nous voulons faire un reset du Bit#7 d'un registre nommé: REG
REG = REG &   0x7F;
REG = REG & ~(0x80); // Même chose, mais en utilisant ~ pour simplifier

// Faison un reset du bit#31:
REG = REG & ~(0x80000000);

// Simplifions :
REG = REG & ~(1 << 31);

// Simplify et compactons :
REG &= ~(1 << 31);

// Reset Bit#21 et Bit# 23:
REG &= ~( (1 << 21) | (1 << 23) );
```

Exemple

```
#include <stdio.h>

int main()
{
  int REG=133;
  printf("Au départ REG = %d\n",REG);
REG &= ~(1<<7);
printf("REG &= ~(1<<7) = %d",REG);
  return 0;
}
```

Solution

- REG = 133 soit en binaire 0b1000_0101
- on met le bit#7 à 0
- REG = 0b0000_0101 soit 5 en décimal

Bit TOGGLE

Réaliser un Toggle d'un bit consiste à inverser l'état actuel du bit. On utiliser la fonction XOR notée `^`

```
// On utilise XOR pour faire un toggle du bit#5
REG ^= (1 << 5);

// Inversion du bit#3 et du bit#5
REG ^= ((1 << 3) | (1 << 5));
```

Exemple

```
#include <stdio.h>

int main()
{
  int REG=133;
  printf("Au départ REG = %d\n",REG);
  REG ^= (1 << 7);
  printf("Toggle du bit#7 de REG  = %d\n",REG);
  REG ^= (1 << 7);
  printf("on refait un Toggle du bit#7 de REG  = %d\n",REG);
  return 0;
}
```

Solution

- REG = 133 soit en binaire 0b1000_0101
- Le bit#7 est à 1, si l'on fait un Toggle, le bit#7 passe à 0 càd REG = 0b0000_0101 = 5
- REG = 5 soit en binaire 0b0000_0101
- Le bit#7 est à 0, si l'on fait un Toggle, le bit#7 passe à 1 càd REG = 0b1000_0101 = 133

Bit CHECK

Supposons que nous souhaitons vérifier que le bit 7 d'un registre est à 1 (set) :

- deux métodes

```
// Si bit#7 est à 1, on lance la fonction DoAThing()
if(REG & (1 << 7))
{
```

```
 DoAThing();
}

// On reste bloqué dans la boucle while tant que bit#7 est à 0
while( ! (REG & (1 << 7)) ) {
  ;
}
```

Voici un autre exemple dans lequel nous souhaitons attendre jusqu'à ce que le bit#9 soit à 0

```
// Tant que bit#9 n'est pas à zero (tant que bit#9 est set)
while((REG & (1 << 9)) != 0) {
  ;
}

// Tant que bit#9 est set
while(REG & (1 << 9)) {
  ;
}
```

Brainstorming

Combien de manière pouvons nous trouver pour tester si l'entier value est une puissance de deux en utilisant les manipulations de bits.

```
(value | (value + 1)) == value
(value & (value + 1)) == value
(value & (value - 1)) == 0
(value | (value + 1)) == 0
(value >> 1) == (value/2)
((value >> 1) << 1) == value
```

Exercice

Que réalise cette fonction ?

```
boolean foo(int x, int y) {
  return ((x & (1 << y)) != 0);
}
```

Solution

- la fonction foo reçoit deux entiers x et y en paramètres
- on créer un masque avec 1 qui est décalé y fois
  - exemple y=7 alors 0b1000 0000 ou 0x80 (le bit#7 est mis à 1)
- on fait un AND avec x, ce qui veut dire que tous les bits de x sont mis à 0, sauf le bit#7 qui conserve sa valeur (0 ou 1).
- deux cas de figure :
  - bit#7 de x vaut 0
    - la fonction foo retourne 0
  - bit#7 de x vaut 1
    - la fonction foo retourne 1

Cette fonction permet de tester si le y ème bit du nombre x est Set ou Reset.

Opérateurs logiques

```
&&      // Logical AND
||      // Logical OR
!       // Logical NOT
```

Attention

Historiquement, C ne possède pas de type Booléan (True or False)

- En C, 0 signifie "False", et non-zéro signifie "True"
- Ainsi 1 est True. Tout comme -37 est True et 3.1415 est True. Seul 0 est False.

Quand on débute en C, on fait souvent l'erreur d'oublier que -37 est équivalent à True sur une opération logique. Vigilence !

Une possibilité d'introduire une notion de booléen plus lisible est d'inclure `#include <stdbool.h>` On peut ainsi accéder à des noms symboliques de type true ou false, mais c'est uniquement de l'affichage, dans une opération logique en C, -54 sera toujours vue comme True...

```
#include <stdio.h>
#include <stdbool.h>
int main(void) {
  bool x = true;
  if (x) {
    printf("x is true!\n");
  }
}
```

Logical AND

De manière simple `&&` retourne vrai (1) si les deux côtés de l'expression sont différents de 0.

L'expression est évaluée de 'Gauche vers Droite'. Si une des partie de l'expression vaut ZERO - l'évaluation se termine.

Par exemple :

```
/* These all return TRUE (1) */

if (4 && 5) return();

i=3;
j=2;
return( i && j);
```

Exemple

```
//For example:
#include <stdio.h>

int main()
{
  int k=0;
int i=3;
int j=2;
if ( i-i && j++) k=1;
  printf("i=%d, j=%d, k=%d",i,j,k);
  return 0;
}
```

Solution

- Le côté gauche (i-i) donne 0, l'évaluation se termine, ce qui fait que j n'est pas postincrémentée et k n'est pas changé.
- i=3, j=2, k=0

WARNING : ici on sort directement de l'évaluation sans incrémenter j. C'est spécifique à &&

Si à la place de la condition `( i-i && j++)`, on avait `( i && j++)`

- le côté gauche est différent de 0, le côté droit est différent de 0, la condition de boucle est valide, on incrémente j, on affecte 1 à k
- i=3, j=3, k=1

Logical OR

De manière simple `||` retourne vrai (1) quand un des deux côté vaut vrai. OR s'évalue également de 'gauche à droite' et va stopper quand une expression returne true.

Exemple

```
//For example:
#include <stdio.h>

int main()
{
  int k=0;
int i=3;
int j=2;
if ( i-i || j++) k=1;
  printf("i=%d, j=%d, k=%d",i,j,k);
  return 0;
}
```

Solution

- Le côté gauche (i-i) donne 0, on évalue à droite j=2, l'opération || retourne True, la variable j est post-incrémentée, la boucle if est validée, et k=1
- i=3, j=3, k=1

Exemple avec subtilité

```
//For example:
#include <stdio.h>

int main()
{
  int k=0;
 int i=3;
 int j=0;
 if ( i-i || j++) k=1;
  printf("i=%d, j=%d, k=%d",i,j,k);
  return 0;
}
```

Solution

- Le côté gauche (i-i) donne 0, on évalue à droite j=0, l'opération || retourne False, la variable j n'est pas post-incrémentée, la boucle if n'est pas validée.
- i=3, j=0, k=0

Autre exemple avec subtilité

```
//For example:
#include <stdio.h>

int main()
{
  int k=0;
 int i=3;
 int j=0;
 if ( i-i || ++j) k=1;
  printf("i=%d, j=%d, k=%d",i,j,k);
  return 0;
}
```

Solution

- Le côté gauche (i-i) donne 0, on pré-incrémente à droite j, j vaut 1, l'opération || retourne True, la boucle if est validée.
- i=3, j=1, k=1

Logical NOT

NOT inverse l'état logique de son opérande. Si l'opérande est 0, 1 est retourné, sinon 0 est retouné. Si l'opérande est différent de 0, il est considéré comme True, et 0 est retourné.

```
!4 /* Returns 0 */
!-4 /* Returns 0 */
!1 /* Returns 0 */
!0 /* Returns 1 */
```

il ne faut pas confondre l'opérateur `~` et l'opérateur `!`

~ vs !

- Dans l'exemple ci-dessous, j'utilise le type uint8_t pour travailler sur 8 bits en non-signé
- Pour utiliser uint8_t, il faut inclure <stdint.h>
- Pour afficher le contenu d'un uint8_t dans un printf on utilise %hhu à la place de %d pour un int.

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

int main()
{
  uint8_t REG=0;
 printf("~REG = %hhu (Not bit à bit)\n",~REG);
 printf("!REG = %hhu (Not logique)\n",!REG);
  return 0;
}
```

Solution

- ~REG = 255 (Not bit à bit) car REG = 0 = 0b0000_0000 -> ~REG = 0b1111_1111 = 255
- !REG = 1 (Not logique) car REG = 0 et !REG = 1

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

int main()
{
```

```
  uint8_t REG=156;
printf("~REG = %hhu (Not bit à bit)\n",~REG);
printf("!REG = %hhu (Not logique)\n",!REG);
  return 0;
}
```

Solution

- ~REG = 99 (Not bit à bit) car REG = 156 = 0b1001_1100 -> ~REG = 0b0110_0011= 99
- !REG = 0 (Not logique) car !256 = 0

Opérateurs relationnels (de test)

```
==      // Egal à
!=      // Différend de
>       // supérieur
<       // inférieur
>=      // supérieur ou égal
<=      // inférieur ou égal
```

Une des erreurs classique en C est de confondre `= et ==`

- `i = i + 1;` correspond à une fonction d'affectation de valeur, on affecte la valeur i+1 à la variable i,
  - `i += 1;` correspond à l'écriture compacte,
  - `i++;` on parle également de post incrément de i,
- `if (i==9)` == correspond à une fonction de test, si i égale 9, alors on exécute le contenu de la boucle,

Post incrémentation et Pré incrémentation

Une des difficulté quand on débute, est de comprendre la différence entre une pré-incrémentation `++i` et une post-incrémentation `i++`

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

```
int main()
{
  int i=41;
  int k=0;
  if( 42 == i++) k=1;
  printf("i= %d, k= %d",i,k);
  return 0;
}
```

Le résultat de ce code est i=42 et k=0, alors que k devrait valoir 1 d'après le code.

Solution

- dans `( 42 == i++)` 42 est évalué, puis i, i=41, comme 42 n'est pas égale à 41, la boucle n'est pas évaluée, on post incrémente i qui passe à 42
- k reste à 0

```
#include <stdio.h>

int main()
{
  int i=41;
  int k=0;
  if( 42 == ++i) k=1;
  printf("i= %d, k= %d",i,k);
  return 0;
}
```

Le résultat de ce code est i=42 et k=1.

Solution

- dans `( 42 == i++)` 42 est évalué, puis on pré incrémente i, i=42, comme 42 est égale à 42, la boucle est évaluée
- k passe à 1

Expressions conditionnelles

On peut compacter les écritures de boucles if-else par des expressions conditionnelles.

```
// if - else  en C
  if ( x == 1 )
    y = 10;
  else
    y = 20;

// l'expression conditionnelle équivalente
  y = (x == 1) ? 10 : 20;
```

à droite, on évalue la première expression `(x ==1 )` et si elle est vraie, on évalue la seconde `10`. Si fausse, la troisième expression est évaluée `20`.

```
// if - else  en C
 if ( x == 1 )
   printf("take car");
 else
   printf("take bike");

// l'expression conditionnelle équivalente
 (x == 1) ? printf("take car") : printf("take
bike");
// ou bien
 printf( (x == 1) ? "take car" : "take bike");
```

Il paraîtrait que les structures conditionnelles soient plus efficaces au niveau de la compilation pour les sections de code critique en temps réel. On retrouve les expressions conditionnelles chez de nombreux programmeurs adeptes de la compacité du code au détriment de la lisibilité.