

EPU - Informatique ROB4

Informatique Système

Cours Introductif

Miranda Coninx

Presented by

Ludovic Saint-Bauzel

ludovic.saint-bauzel@sorbonne-universite.fr

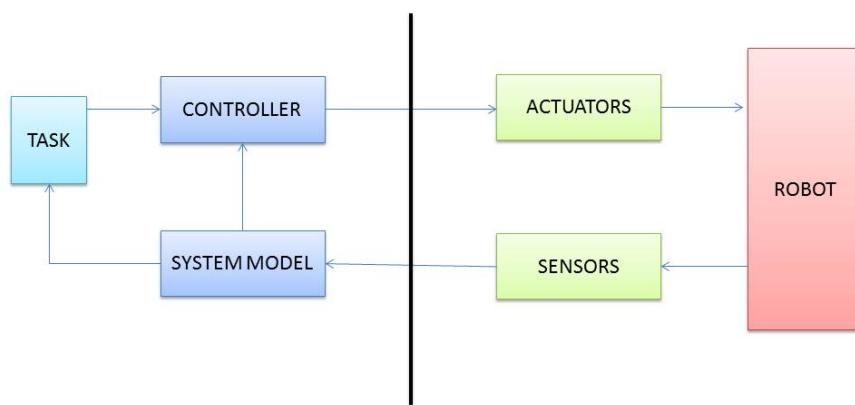
Sorbonne Université

Institut des Systèmes Intelligents et de Robotique (CNRS UMR 7222)

2023-2024



A classical control loop in robotics



Plan de ce cours

Introduction

Robotique et Programmation Système

Le système d'exploitation

Définition

Les différents types d'OS

Les particularités de Linux

Couches d'abstractions logicielles

Mécanismes gérés par l'OS

Vue générale

Le processeur

Les interruptions

Les Entrées/Sorties (IO)

Technique de production de programmes

Rappels de C

La chaîne de compilation

Make et Makefile

Outils de débogage

Mémoire et pointeurs en C

Allocation mémoire

Les pointeurs

↳ Comment synthétiser le contrôleur ?

Premières solutions : DSPs et μ contrôleurs (Arduino, etc.)

- ▶ Nombre d'E/S (relativement) limité
- ▶ Types de protocole de communication limités
- ▶ Capacité de calcul "faible"

Secondes solutions : architectures "PC" Intel ix86, PowerPC, Motorola 68K, ...

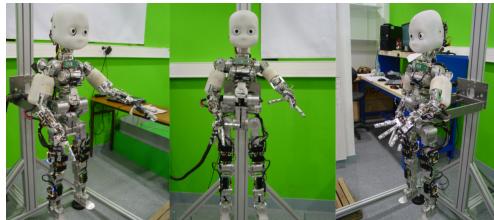
- ▶ Solutions versatiles et adaptables
- ▶ Un grand nombre de configurations matérielles possibles : PC classique, ordinateurs monocartes (Raspberry Pi, etc.), PC104 | multi-processeurs, mémoire partagée ...
- ▶ Systèmes d'Exploitations généralistes (Linux) ou dédiés, souvent temps-réel (VxWorks, QNX, RTlinux, Xenomai ...)

↪ Comment synthétiser le contrôleur ?

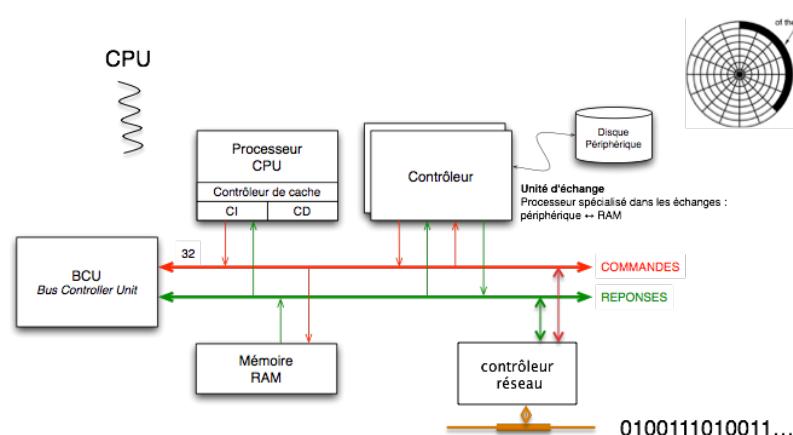
Solutions hybrides

- ▶ Contrôleurs locaux (au niveau des actionneurs) basés sur des DSPs ou des μ contrôleurs
 - ↪ Déport d'une partie des calculs (asservissement bas-niveau, traitement des signaux des capteurs).
 - ▶ Architecture de type PC au niveau central
 - ↪ Centralisation des informations et calculs haut-niveau (coordination des mouvements, fusion données capteurs).
 - ▶ Communication entre le haut et le bas niveau au travers de bus avec protocoles standardisés.

↪ Approche justifiée pour des robots à grand nombre de ddl



OS et abstraction



→ **Système d'exploitation** (Operating System, OS) : ensemble de programmes à l'interface entre le matériel de l'ordinateur et les utilisateurs.

Définition générale

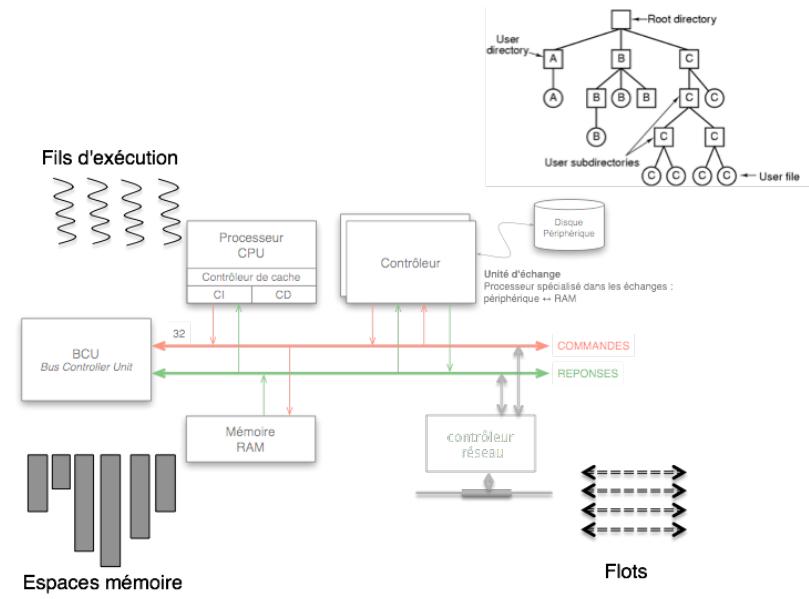
- ▶ Prise en charge de la gestion des ressources (processeur, mémoire et périphériques) et de leur partage
 - ▶ Fournit une **abstraction** de ces ressources, "au-dessus du matériel", facile d'emploi et conviviale (requêtes standardisées indépendantes du type de matériel).

Rôles principaux

- ▶ Partage du processeur : parmi tous les programmes chargés en mémoire centrale, lequel doit s'exécuter à un instant donné ?
 - ▶ Allocation de la mémoire centrale aux différents programmes : comment assurer la protection des zones mémoires entre différents programmes en cours d'exécution ?
 - ▶ Traitements des requêtes d'accès (E/S) aux différents périphériques ?

→ **L'OS doit assurer l'équité d'accès à la machine physique et aux ressources matérielles entre les différents programmes et la cohérence de ces accès.**

OS et abstraction



⇒ Quels types d'OS pour quels types de besoin ?

Les "ancêtres" : OS à traitement par lots (batch)

- ▶ Maximisation de l'utilisation du processeur
- ▶ Pas d'interactions possibles avec l'utilisateur au cours de l'exécution.

OS interactifs (Windows, Unix, Linux, BSD,...)

- ▶ Interaction entre le(s) utilisateur(s) et le système même au cours de l'exécution d'un programme
- ▶ Utilisation multi-utilisateurs " transparente "
- ▶ Partage du temps processeur entre les différents processus et les différents utilisateurs (système en temps partagé).

OS temps-réel (VxWorks, RTlinux, QNX, Xenomai ...)

- ▶ Réagissent en un temps adapté aux événements externes ;
- ▶ Fonctionnent en continu sans réduire le débit du flot d'information à traiter ;
- ▶ Temps de calcul connus et modélisables pour permettre l'analyse de la réactivité ;
- ▶ Latence maîtrisée.

⇒ Quels types d'OS pour quels types de besoin ?

Les "ancêtres" : OS à traitement par lots (batch)

...

OS interactifs (Windows, Unix, Linux, BSD,...)

...

OS temps-réel (VxWorks, RTlinux, QNX ...)

- ▶ Réagissent en un temps adapté aux événements externes ;
- ▶ Fonctionnent en continu sans réduire le débit du flot d'information à traiter ;
- ▶ Temps de calcul connus et modélisables pour permettre l'analyse de la réactivité ;
- ▶ Latence maîtrisée.

OS temps-réel : remarques

- ▶ Utile dans le cas de **contraintes temporelles** : " j'ai absolument besoin du résultat de cette opération à tel instant ".
- ▶ Ce n'est pas temps le temps d'exécution qui est primordial mais la capacité à fournir un résultat à un instant précis ou à un **intervalle de temps régulier** (qui peut aller de la ms à l'heure en fonction du système physique associé).
- ▶ Un **mécanisme de préemption** permet l'interruption à tout moment de l'exécution d'un processus si un processus de priorité plus élevée apparaît.

⇒ UNIX

Développement commencé en été 1969 (54 ans !).



Ken Thompson (seated) and Dennis Ritchie in 1972, shortly after they and their colleagues in Bell Labs created Unix.

```
11/3/71                                SYS SEEK (II)

NAME      seek -- move read/write pointer
SYNOPSIS  (file descriptor in fd)
          sys seek; offset; pathname / seek > 19.
DESCRIPTION The file descriptor refers to a file open for
reading or writing. The (or write) pointer
position is set as follows:
  if pathname is 0, the pointer is set to offset;
  if pathname is 1, the pointer is set to its
current location plus offset;
  if pathname is 2, the pointer is set to the
size of the file plus offset.

FILES    --
SEE ALSO tell
DIAGNOSTICS The error bit (error) is set for an undefined
file descriptor.
BUGS     A file can conceptually be as large as 2**30
bytes. Clearly only 2**16 bytes can be addressed
by offset. If offset is negative, it wraps
around. If offset is greater than the size of the
file and R and RF, something is going to be
done about that.
OWNER   Ken, der
```

A page from the first Unix manual or *man* (released November 1971), by Thompson (ken) and Ritchie (dmr).

From "The strange birth and long life of Unix", by W. Toomey, IEEE Spectrum, Dec. 2011.

⇒ Quelques faits à propos de Linux (que vous utiliserez en TP) :

- ▶ Système compatible avec la norme POSIX pour les OS.
- ▶ Système né en 1991 (Linus Torvalds) et membre de la famille des systèmes de type Unix.
- ▶ Code source libre d'accès (licence publique GNU) et donc système vivant s'enrichissant constamment.
- ▶ Programmé dans un langage puissant et "haut niveau" : C.
- ▶ Système interactif (multi-utilisateurs, multi-tâches) avec des aspects compatibles avec la problématique du temps réel faiblement contraint.
- ▶ Le système s'organise autour du noyau (kernel) qui représente l'ensemble des programmes nécessaires au bon fonctionnement du système.
- ▶ Le noyau peut être vu comme un gestionnaire de processus.
- ▶ Le noyau est modulaire et peut être modifié hors ligne (recompilation) et dynamiquement (ajout de modules).

→ Noyau, OS, distribution :

► Au sens strict, Linux est un **noyau** (kernel)

- Au plus près du matériel
- Gestion mémoire, entrées/sorties, ordonnancement
- Accès direct au matériel, en "mode réel" (dangereux !)

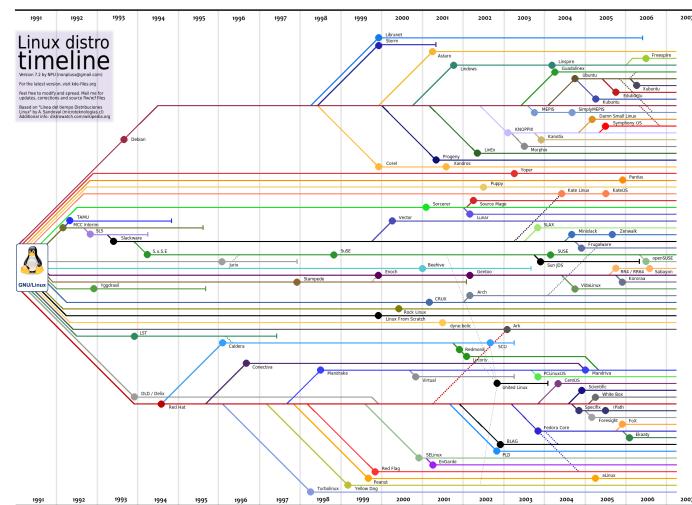
► OS = noyau + collection d'outils en **espace utilisateur**

- Sous Linux : beaucoup d'outils du **projet GNU** (d'où : GNU/Linux)
- Collection d'outils sous licence libre
- Projet démarré en 1983 par Richard Stallman
- Indépendant de Linux (un autre noyau, Hurd, existe)
- Outils console (ls, cd, etc.), GRUB, gcc, bureau GNOME, etc.

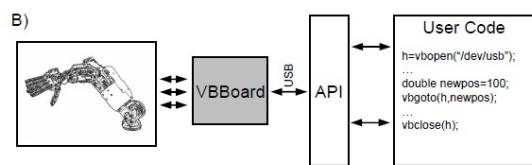
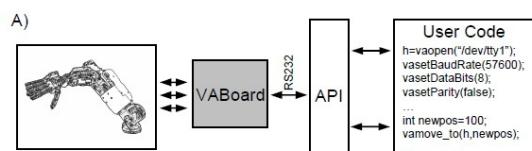
► Distribution = ensemble cohérent de logiciels autour d'un noyau

- Va au delà de l'OS à proprement parler (noyau + outils userspace + programmes variés)
- Gestion des versions des logiciels, des dépendances, des mises à jour, ...
- Utilise généralement un **système de gestion de paquets** (ex : APT)
- Exemples : Debian, Ubuntu, Mint, etc.

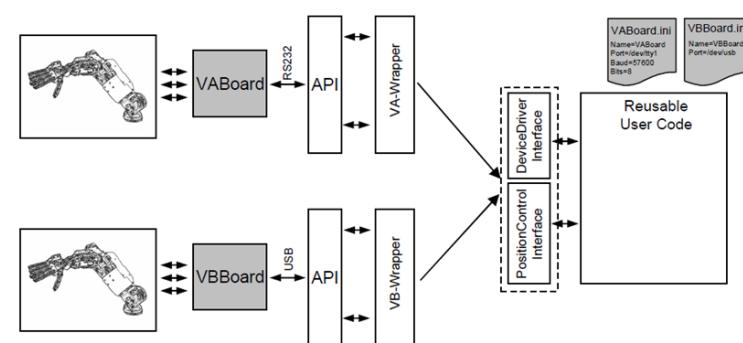
↪ Ne pas confondre Linux (le noyau) et les distributions Linux (Debian, Mandriva, Ubuntu, Gentoo, Fedora, Suse ...).



Q : why do we need an abstraction layer in robotics ?



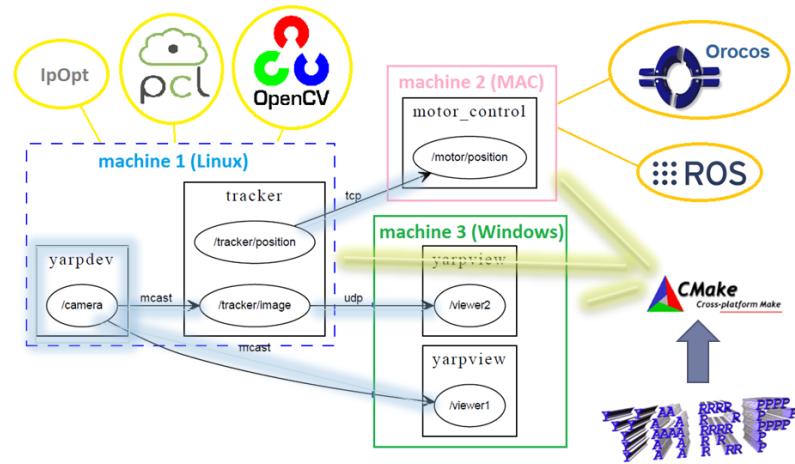
A : code re-use, abstraction from hardware, ...



B : capability to support multiple robots..



C : interoperability of multiple OS..



→ Au delà des OS au sens classique du terme, des couches d'abstraction ont été développées spécifiquement pour la Robotique :

Example : ROS

" ROS is an open-source, **meta-operating system** for your robot. It provides the services you would expect from an operating system, including **hardware abstraction**, low-level device control, implementation of **commonly-used functionality**, **message-passing** between processes, and package management. It also provides tools and libraries for obtaining, building, writing, and running code across **multiple computers**."

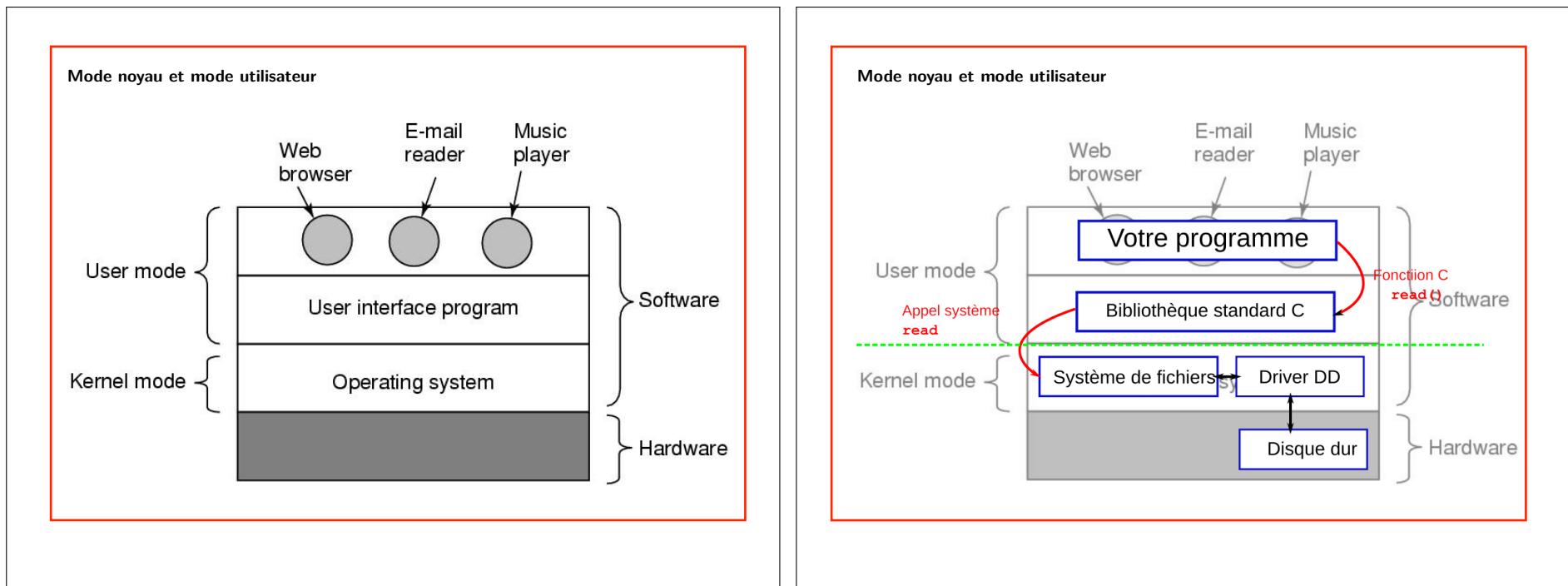
ROS wiki

ROS is not an operating system ! But...

- ▶ abstraction from hardware : code behaviors and controls without worrying to change sensors, actuators, processors, and networks
- ▶ commonly-used functionality : promote code re-use, make robot software more stable and long-lasting
- ▶ message-passing : helps communication (loose coupling) between sensors, processors, and actuators to ease gradual system evolution (i.e. changes in devices less disruptive)
- ▶ multiple computers : minimize problems of incompatible architectures, frameworks, and middlewares
- ▶ free, opensource (license BSD principalement)

→ L'OS réalise une couche logicielle placée entre la machine matérielle et les applications. Il peut être découpé en plusieurs grandes fonctionnalités :

- ▶ **Gestion du processeur** : l'allocation du temps processeur aux différents programmes en cours d'exécution est réalisé par un ordonneur qui planifie l'exécution des programmes (il existe différentes politiques d'ordonnancement : FIFO, priorités, tourniquet...).
- ▶ **Gestion des objets externes** : la mémoire centrale est volatile et cela nécessite l'utilisation de périphériques de mémoire de masse (disques durs....). La gestion de l'allocation des mémoires de masse ainsi que l'accès aux données stockées s'appuie sur le système de gestion de fichiers.
- ▶ **Gestion des accès E/S aux périphériques** : gestion du lien entre les appels de haut niveau des programmes utilisateurs (par exemple getchar()) et les opérations de bas niveau de l'unité d'échange responsable du périphérique (ici le clavier). Ce lien est assuré par les pilotes d'E/S (drivers).
- ▶ **Gestion de la mémoire centrale** : la mémoire physique étant limitée, seuls les codes en cours d'exécution sont placés en mémoire centrale. Les autres données en mémoire sont placées en mémoire virtuelle.
- ▶ **Gestion de la concurrence** : accès cohérents aux données partagées par plusieurs processus au travers d'outils de synchronisation et de communications entre processus.
- ▶ **Gestion de la protection** : limite les accès possibles des programmes utilisateurs afin de préserver les ressources (processeur, mémoire, fichiers). Modes d'exécution utilisateur et superviseur.
- ▶ **Gestion de l'accès au réseau** : couche de protocole permettant les échanges avec des machines distantes.



→ Le processeur a pour rôle l'exécution des programmes. Les programmes **utilisateur** accèdent aux fonctionnalités de l'OS au travers d'appels aux **routines systèmes**.

Caractéristiques

- ▶ Vitesse d'horloge
- ▶ Jeu d'instruction (propre à l'architecture : x86_64, mips, arm, etc...)
- ▶ Etat observable entre l'exécution de 2 instructions uniquement
- ▶ Zone mémoire dédiée (la **pile**) pour l'exécution des programmes (appel de sous-programmes, traitement d'interruptions).

Contexte d'exécution d'un programme

- | | |
|-------------------------|----------------------|
| ▶ Compteur de programme | ▶ Etat du processeur |
| ▶ Pointeur de pile | ▶ Registres généraux |

Il existe deux modes d'exécution possibles

- ▶ **Superviseur** : toutes les instructions et toute la mémoire sont accessibles ;
- ▶ **Utilisateur** : accès restreint à certaines instructions et adresses mémoire.

↪ **Processus** : instance d'exécution d'un programme par le processeur.

↪ **processus** ≠ **programme**

→ Les interruptions permettent la prise en compte asynchrones d'événements externes au processeur.

- ▶ La prise en compte d'une interruption provoque l'arrêt (pas nécessairement définitif) du programme en cours et l'exécution du sous-programme associé à l'interruption.
- ▶ La gestion des interruptions peut être **hiérarchisée** : dans ce cas les interruptions de priorité basse ne peuvent pas interrompre les interruptions de priorité plus élevée.
- ▶ Certaines interruptions sont **masquables**, ie elles doivent être ignorées par certains programmes.
- ▶ La table de pointeurs vers les sous-programmes de traitement associés aux interruptions est appelée **vecteur d'interruption**.
- ▶ Les déroulements sont des interruptions non masquables internes au processeur. Ils permettent la prise en compte d'erreurs du type : violation de mode d'exécution, violation d'accès mémoire, erreur d'adressage, erreur arithmétique (division par 0, débordement de registre...),...

→ La gestion des I/O se fait au travers d'un *bus*. Le rôle du bus est d'assurer la connexion des périphériques au processeurs. Il existe différents protocoles de bus, les plus standards étant :

- ▶ PCI : bus parallèle 32 ou 64 bits, standardisé par Intel. Nombreuses variantes : PCI-Express, Compact-PCI, Mini-PCI,...
- ▶ USB : Universal Serial Bus. Bus d'I/O série grand public.
- ▶ I2C : Inter-Integrated Circuit bus, standardisé par Philips. Bus série 2 fils simple pour interconnexion de périphériques. Utilisé pour capteurs de température, information sur les moniteurs,...
- ▶ CAN Controller Area Network. Conçu par l'industrie automobile. Communication temps-réel entre µcontrôleurs et capteurs/actionneurs.

C programming language

History

- ▶ 1972 : Dennis Ritchie - AT&T Bell Labs
- ▶ 1978 : "The C programming language" published
- ▶ 1989 : C89 standard, aka ANSI C or standard C
- ▶ 1990 : C90 is adopted by ISO
- ▶ 1999 : C99 standard
- ▶ C11 (2011), C17 (2018; current latest standard), C2x (in development expected 2023)

Widely used

- ▶ OS, like Linux
- ▶ µcontrollers
- ▶ embedded processors
- ▶ DSP processors

Features

- ▶ Pro : few keywords, structures, pointers (memory, arrays, ...), external libraries, faster code
- ▶ Cons : no exceptions, no object-oriented programming, no polymorphism



Beware! C is inherently unsafe!

- ▶ Low-level language
- ▶ No exceptions
- ▶ No range checking
- ▶ No type checking at runtime

Writing C code

Editing : hello.c

```
/* hello.c - the simplest program */
#include<stdio.h>
int main(int argc, char* argv[])
{
    printf("Hello!");
    return 0;
}
```

Compiling :

```
gcc -Wall hello.c -o hello
```

Includes et prototypes

Les directives `#include` permettent d'inclure des fichiers d'en-tête (*headers*)

contenant les **prototypes** des fonctions

Exemple : dans `#include <stdio.h>` :

- ▶ Ecriture formatée : `int printf(const char *format, ...);`

Dans `#include <stdlib.h>` :

- ▶ Conversion d'une chaîne en entier : `int atoi(const char *nptr);`
- ▶ Conversion d'une chaîne en double : `double atof(const char *nptr);`

man

Pour connaître dans quel header chercher une fonction, on peut utiliser le manuel (man). Par exemple (dans le shell) : `man atof`

```
ATOF(3)          Manuel du programmeur Linux          ATOF(3)
NOM           atof - Convertir une chaîne en réel double précision (double)
SYNOPSIS      #include <stdlib.h>
              double atof(const char *nptr);
DESCRIPTION    La fonction atof() convertit le début de la chaîne pointée par entré en
```

Writing C code

Le manuel

- ▶ **man** est très utile. Il a aussi un manuel (`man man`)

Pages divisées en 9 sections, les plus utiles :

- ▶ 1 : commandes shell standard (ex : `ls`)
- ▶ 8 : commandes shell admin (ex : `adduser`)
- ▶ 3 : fonctions C fournies par les bibliothèques (ex : `atof`)
- ▶ 2 : fonctions C correspondant à des appels système (fournies par l'API du noyau ; ex : `read`)

▶ Pour chercher une page dans une section particulière, donner son numéro avant le nom (ex : `man 3 printf`) Sinon, renvoie l'entrée dans la première section où elle est trouvée.

▶ Pour les fonctions C, donne (en général) :

- ▶ Description, prototype, header correspondant
- ▶ Fonctionnement, valeur de retour, arguments
- ▶ Gestion d'erreurs
- ▶ Parfois, exemples de code

▶ Qualité et quantité très variables (de très sommaire à beaucoup trop détaillé !)

↪ Prototype générique de la fonction main : `int main(int argc, char* argv[]);`

- ▶ La fonction main d'un programme peut prendre des arguments en ligne de commande.
- ▶ Exemple : `> cp fichier1.tex fichier2.tex`
- ▶ La récupération de ces arguments se fait au moyen des arguments argc et argv.
- ▶ argc : nombre d'arguments passés au main + 1
- ▶ argv : tableau de pointeurs sur des chaînes de caractères :
 - ▶ le premier élément argv[0] pointe sur la chaîne qui contient le nom du fichier exécutable du programme.
 - ▶ les éléments suivants, argv[1], argv[2], ..., argv[argc-1] pointent sur les chaînes correspondants aux arguments passés en ligne de commande.

↪ Les arguments du main offrent de nombreuses possibilités.

TEST 1

Given test.c

```
/* a test */
#include<stdio.h>
int main(int argc, char* argv[])
{
    puts("Hello students!"); //output text to stdout and end line
    return 0;
}
```

and the following definition :

```
#define DMSG "Hello students!"
How do we rewrite test.c?
```

```
/* a test using define */
#define DMSG "Hello students!"
#include<stdio.h>
int main(int argc, char* argv[])
{
    const char msg[] = DMSG;
    puts(msg);
    return 0;
}
```

TEST 2

```
#define DMSG = "Hello students!"
```

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

```
int function (void arg)
{ return arg-1; }
```

Find the error in the statements above, and fix it :

```
#define DMSG "Hello students!"
```

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

```
int function (int arg)
{ return arg-1; }
```

→ Les étapes de compilation et d'édition de liens sont précédée d'un passage du préprocesseur. Le préprocesseur permet des manipulations pre compilation et édition de liens.

- ▶ Inclusion de fichiers à l'aide de l'instruction `#include` :
 - ▶ Cette instruction permet l'inclusion de fichiers d'en-tête .h;
 - ▶ Ces fichiers permettent notamment de faire référence à des fonctions dont le symbole ne sera résolu qu'au moment de l'édition de liens;
 - ▶ Ils permettent aussi la définition de types, de structures...;
- ▶ Remplacement de morceaux de code par des constantes ou des macros en utilisant l'instruction `#define`;

```
#define PI 3.14159
#define norm(x,y) (sqrt(pow(x,2.0)+pow(y,2.0)))
#define myfabs(x) ((x < 0) ? -x : x)
    ▶ la prise en compte conditionnelle de morceaux de code à la compilation avec
    #ifdef...#endif ou #ifndef...#endif;

#ifndef MON_HEADER_H
#define MON_HEADER_H
....bloc pris en compte de manière conditionnelle...
#endif
```

→ La production d'un programme exécutable à partir d'un code écrit en C répond à plusieurs règles.

- ▶ Un et un seul des fichiers sources utilisés contient une fonction `main()` ;
- ▶ Cette fonction est la fonction principale du programme ;
- ▶ C'est à partir de `main()` que l'ensemble des traitements sera effectué soit directement, soit par appel à d'autres fonctions ;
- ▶ Ces autres fonctions peuvent être :
 - ▶ des fonctions standard et en général portables du C ;
 - ▶ des fonctions écrites par le programmeur.

Trois étapes

1. préprocesseur
2. compilation
3. édition de liens

→ La production du programme passe par une étape de *compilation*.

- ▶ Elle permet, à partir d'un fichier source .c, de produire un fichier objet .o ;
- ▶ Le fichier objet contient "une traduction" du code source C en un code compréhensible par le processeur (assembleur) ;



- ▶ GCC (GNU C Compiler) créé en 1985 par Richard Stallman
- ▶ `gcc -c foo.c` produit le fichier objet `foo.o` ;
- ▶ Les options classiquement passées à `gcc` à l'étape de compilation sont les suivantes :
 - ▶ `-g` pour rendre le débogage possible;
 - ▶ `-Wall` pour afficher tous les avertissements;
 - ▶ `-O0` pour imposer la non optimisation du code (par défaut)
 - ▶ ...

→ **La compilation doit être suivie d'une édition de liens.**

- ▶ Elle permet le rassemblement de l'ensemble des fichiers objet qui composent le programme ;
- ▶ Son rôle est la résolution de l'ensemble des références aux symboles non définis des différents fichiers objets ;
- ▶ L'éditeur de liens GNU est le programme **ld** qui est en général appelé de manière implicite via **gcc** ;
- ▶ **gcc -o mon_exec main.o titi.o tutu.o tata.o** produit l'exécutable **mon_exec** ;
- ▶ Les fichiers objets peuvent être regroupé en bibliothèques statiques .a à l'aide des commandes **ar** et **ranlib** :

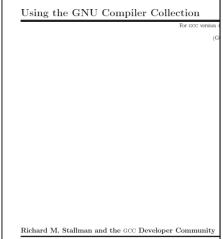
```
ar cr libtoto.a titi.o tutu.o tata.o
ranlib libtoto.a
```

- ▶ L'appel d'une bibliothèque à l'édition de liens se fait comme suit :

 gcc -o mon_exec main.o -L./lib -ltoto
- ▶ L'appel à la bibliothèque standard de math se fait par **-lm**, **libm.a** étant le nom de cette bibliothèque ;
- ▶ L'option **-L chemin_de_la_lib** permet d'indiquer les chemins de bibliothèques qui ne seraient pas des bibliothèques standards du C (**/usr/lib**) ;
- ▶ Il existe aussi des bibliothèques qui sont liées dynamiquement (.so ou .dll sous Windows) ;

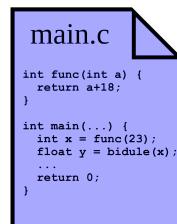
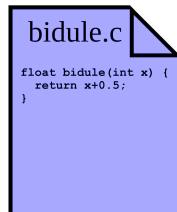
Common options :

- ▶ **Generic**
 - ▶ **-S** stop after compilation, do not assemble
 - ▶ **-E** stop after preprocessing, do not compile
 - ▶ **-o file** place output in file **file** (default **a.out**)
- ▶ **Optimization**
 - ▶ **-O0** no optimization
 - ▶ **-O1** optimize and minimize compile time
 - ▶ **-O2** more costly optimization
 - ▶ **-Os** optimize for code size
- ▶ **C language**
 - ▶ **-ansi** , **-std=c90** , **-std=iso9899:1990** to select the standard C
 - ▶ **-pedantic** for all the diagnostics required by the specific standard
 - ▶ **-pedantic-errors** to treat warning as errors
- ▶ **Debugging and errors**
 - ▶ **-g** enables generation of debugging information, that can be used by **gdb** (works with **-O0**)
 - ▶ **-Werr** make all warnings into errors
 - ▶ **-Wall** enables all (main) warnings
 - ▶ **-Wextra** enables extra warnings not enabled by **-Wall** (for example **-Wsign-compare**)
- ▶ **Threads**
 - ▶ **-pthread** support multithreading using POSIX threads library
- ▶ **Linking**
 - ▶ **-llibrary** search the library named **library** when linking (default **llibrary.a**), for example **-lm** links the math library
 - ▶ **-Ldir** add **dir** to the list of directories to be searched for **-l**

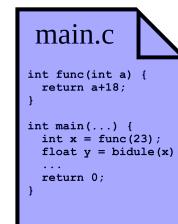
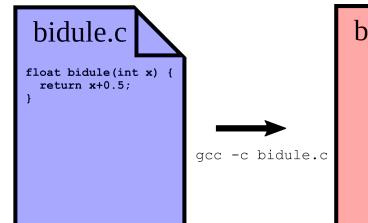


the manual can be downloaded from gcc.gnu.org

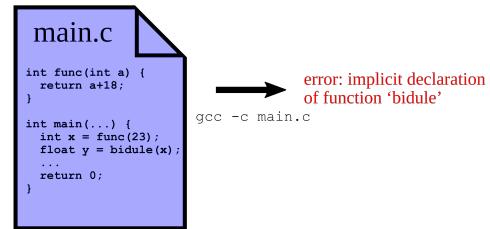
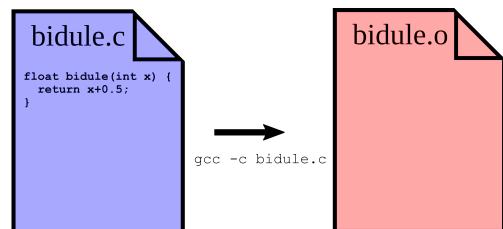
Exemple : compilation



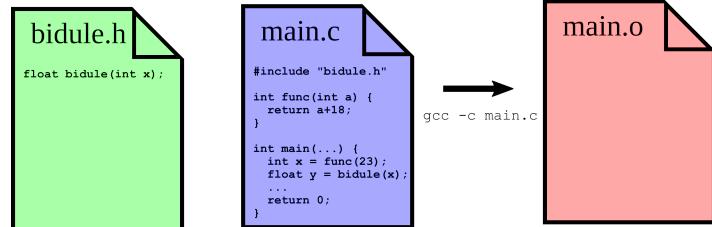
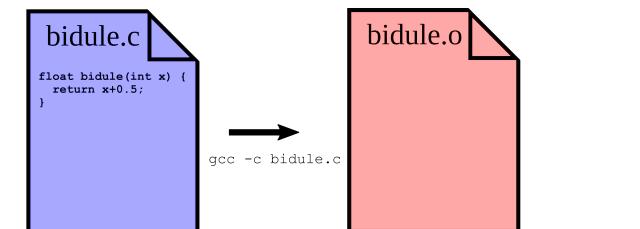
Exemple : compilation



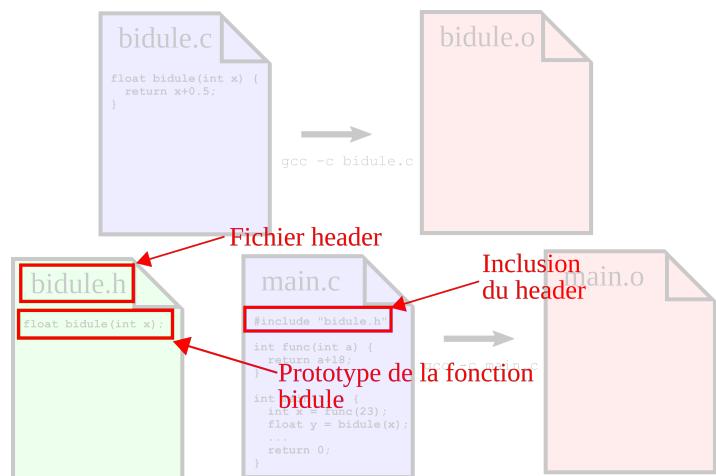
Exemple : compilation



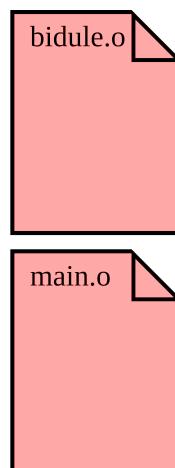
Exemple : compilation



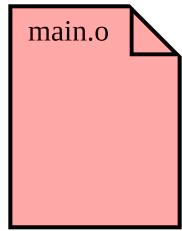
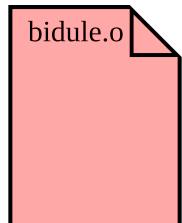
Exemple : compilation



Exemple : édition de liens



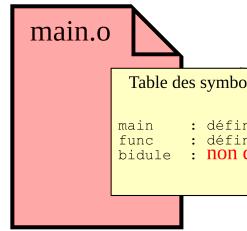
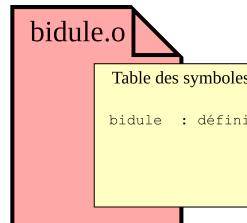
Exemple : édition de liens



→
gcc -o monprog main.o

error: undefined reference
to 'bidule'

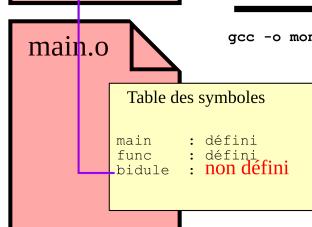
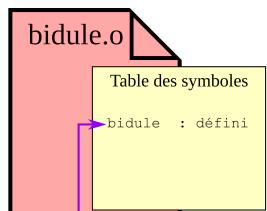
Exemple : édition de liens



→
gcc -o monprog main.o

error: undefined reference
to 'bidule'

Exemple : édition de liens



→
gcc -o monprog bidule.o main.o



TEST 3

Editing : test.c

```
/* a test */  
#include<stdio.h>  
int main(int argc, char* argv[]){  
    puts("Hello students!"); //output text to stdout and end line  
    return 0;  
}
```

How do we compile ?

gcc -Wall test.c -o test

Or :

avant il y a les pre-processing

gcc -Wall -c test.c **precompilation & compilation**
gcc -Wall -o test test.o **edition des liens**

↪ Exemple simple

- ▶ Création de l'exécutable `my_exec`
 - ▶ Fonction `main()` déclarée dans `main.c`
 - ▶ `main()` appelle les fonctions `titi()`, `tata()` et `tutu()` respectivement déclarées dans `titi.c`, `tata.c` et `tutu.c`
 - ▶ `titi()` et `tata()` appellent `tutu()`
- ↪ Comment générer l'exécutable `my_exec` ?
↪ Avec ou sans bibliothèque ?

Solution basique

Compilation

```
» gcc -g -c titi.c  
» gcc -g -c tutu.c  
» gcc -g -c main.c  
» gcc -g -c tata.c
```

Edition de liens

```
» gcc -g -o my_exec main.o tutu.o tata.o titi.o
```

↪ Exemple simple

Solution basique

Compilation

```
» gcc -g -c titi.c  
» gcc -g -c tutu.c  
» gcc -g -c main.c  
» gcc -g -c tata.c
```

Edition de liens

```
» gcc -g -o my_exec main.o tutu.o tata.o titi.o
```

Αυτό δεν το καταλαβαίνω και πολύ

Solution utilisant une bibliothèque

Compilation

```
» gcc -g -c titi.c  
» gcc -g -c tutu.c  
» gcc -g -c main.c  
» gcc -g -c tata.c
```

Création de la bibliothèque

```
» ar cr libtoto.a tutu.o titi.o tata.o  
» ranlib libtoto.a
```

Edition de liens

```
» gcc -g -o my_exec2 main.o -L . -ltoto
```

Exemple simple ... mais quand-même ...

Compilation

```
» gcc -g -c titi.c  
» gcc -g -c tutu.c  
» gcc -g -c main.c  
» gcc -g -c tata.c
```

Création de la bibliothèque

```
» ar cr libtoto.a tutu.o titi.o tata.o  
» ranlib libtoto.a
```

Edition de liens

```
» gcc -g -o my_exec2 main.o -L . -ltoto
```

Remarques

- ▶ Long et fastidieux au delà de 2 fichiers à compiler (souvent un projet peut en compter plusieurs centaines) !
 - ▶ Quand un fichier source est modifié, il n'est pas nécessaire de tout recompiler : par souci d'efficacité il faut donc connaître les **dépendances** et ne recompiler et lier ce qui est nécessaire. Ceci n'est pas humainement faisable au delà de 5 fichiers sources.
- ↪ Il existe des outils qui permettent d'automatiser le processus de création de programme : `make` et pour les projets plus importants `CMake`, `autotools`, ...

↪ `make` est un programme permettant d'exécuter de manière automatisée l'ensemble des opérations nécessaires à l'obtention d'un exécutable

- ▶ Les instructions exécutées par `make` sont placées dans un fichier **Makefile**
- ▶ En utilisant les dates d'accès aux fichiers, `make` est capable de déterminer quels fichiers doivent être recompilés ou non → **gestion des dépendances**
- ▶ La seule connaissance strictement nécessaire pour l'écriture de fichiers `Makefile` "simples" est celle de l'écriture des règles :
 - ▶ CIBLE : DEPENDANCES
COMMANDÉ
 - ...
 - ▶ CIBLES peut représenter le nom d'un fichier à produire (un fichier `.o`, un exécutable...) ou de manière plus générale, un nom quelconque (`question2` par exemple)
 - ▶ DEPENDANCES représente l'ensemble des règles qui doivent avoir été exécutées et/ou l'ensemble des fichiers qui doivent exister afin de pouvoir lancer la commande COMMANDÉ
- ▶ `make` est en fait beaucoup plus général que cela. Par exemple, ce document a été réalisé en utilisant `make`. (cadre général de la compilation en C).

↪ Un bon document de référence sur `make` peut être trouvé à cette adresse :
<http://www.laas.fr/~matthieu/cours/make>

↪ `Make` automatise la chaîne de compilation...`CMake`, `Autotools` et certaines IDE automatisent la production de `Makefile` (ou de tout autre mécanisme standardisé de description efficace de la chaîne de compilation)

→ Exemple simple

- ▶ L'exemple qui suit produit un exécutable `monprog` à partir des fichiers `main.o` et `fichier1.o`.
- ▶ Ces deux fichiers `.o` dépendent de leurs fichiers source respectifs. De plus `main.o` dépend de `fichier1.h`.
- ▶ La règle `clean` permet d'effacer les fichiers `.o`. La règle `vclean` applique la règle `clean` puis efface l'exécutable `monprog`.

```
monprog: main.o fichier1.o
          gcc -o monprog main.o fichier1.o

fichier1.o: fichier1.c
            gcc -c fichier1.c

main.o: main.c fichier1.h
        gcc -c main.c

clean:
        rm -f *.o

vclean: clean
        rm -f monprog
```

→ Des erreurs syntaxiques ou conceptuels peuvent faire échouer la phase de compilation et/ou d'édition des liens. Par ailleurs, une fois l'exécutable généré, son comportement peut ne pas être celui attendu : résultats faux, comportement non conforme, erreurs de segmentation...

→ Nécessite le recours à des outils de débogage :

- ▶ Le plus simple : la fonction `int printf(const char *format, ...)` qui permet notamment d'afficher à l'écran des chaînes de caractères.
- ▶ La fonction `void assert(scalar expression)` qui termine le programme en cas d'échec (0) du test.
- ▶ Le débogueur `gdb` (et éventuellement son interface graphique `ddd`) qui permet l'exécution d'un programme pas à pas tout en visualisant l'évolution des valeurs des variables.

» `gdb ./my_exec`
`(gdb) run arguments_du_main`

...

- ▶ L'outil `valgrind` qui permet notamment de vérifier que la mémoire allouée dynamiquement est bien libérée par le programme (il existe un grand nombre d'autres possibilités offertes par valgrind).

» `valgrind ./my_exec run arguments_du_main ...`

- ▶ Les outils `gdb` et `valgrind` nécessitent la compilation et l'édition de liens avec l'option `-g`.

→ A propos d'allocation mémoire...

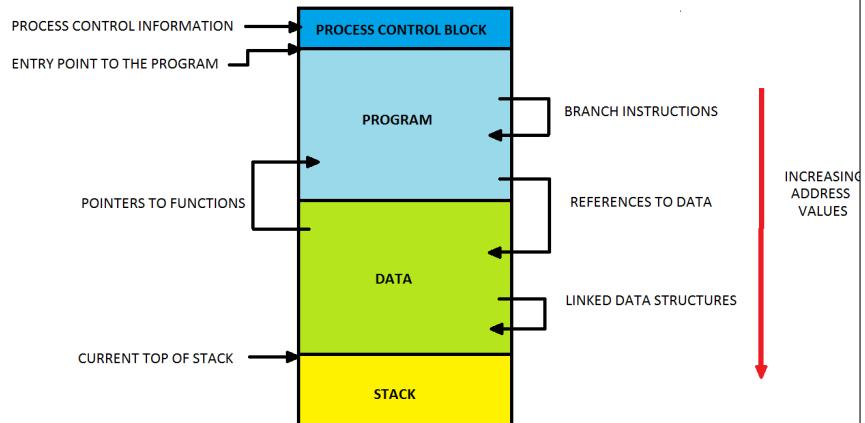
Differentes méthodes d'allocation

- ▶ De manière **statique**, au lancement du programme ; C'est le cas des variables globales ;
- ▶ De manière **automatique**, dans la pile (*stack*) d'exécution du processus correspondant au programme en cours ; C'est le cas des variables locales aux fonctions dont la taille est connue et spécifiée au moment de l'écriture du programme ; C'est aussi le cas de la mémoire allouée pour le passage des arguments à une fonction.
- ▶ De manière **dynamique**, dans le tas (*heap*) autrement dit dans une zone mémoire du système non spécifiquement dédiée au programme en cours d'exécution.

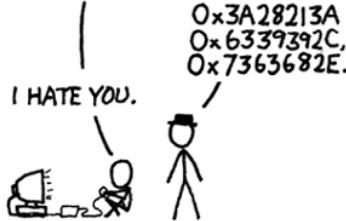
Quelques remarques

- ▶ L'allocation dynamique permet une grande souplesse de programmation car elle n'impose pas l'allocation de morceaux arbitrairement grands de mémoire au lancement du programme ;
- ▶ Elle reste cependant relativement coûteuse en temps d'exécution, peut parfois échouer (difficile à rattraper) et est la source d'un grand nombre de bogues.
- ▶ Dans la mesure du possible, il est préférable d'éviter les instructions d'allocation dynamique dans des zones critiques du programme ;
- ▶ On évitera par exemple d'allouer dynamiquement de la mémoire dans la partie du code qui réalise une boucle d'asservissement à une fréquence élevée.

→ a process in memory



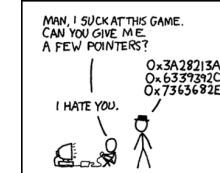
MAN, I SUCK AT THIS GAME.
CAN YOU GIVE ME
A FEW POINTERS?



(original comics on xkcd.com)

pointers are memory addresses

symbol	example	meaning
&	&x	the address of x
*	*p	the object pointed by p



```
int *a, *b; declare int pointers a,b
int c; declare int c
a = &c; a has the address of c
*b = *a; the object pointed by b has the same value as the one pointed by a
b = a; b also points to the object pointed by a
```

→ Un pointeur est une variable contenant une adresse mémoire.



- ▶ Notation : type *p ; -> "p est un pointeur sur type"
Exemples : char *p, int *p, POINT *p, struct data *p, etc.
- ▶ Accès à la donnée pointée par p : *p;
- ▶ Accès au champ d'une structure pointée par p :
p->champ ou (*p).champ;
- ▶ Accès à l'adresse de la variable var : &var;
- ▶ void *p : pointeur sur un type indéfini (cf. malloc());
- ▶ p = &data; stocke dans p l'adresse de data;
- ▶ *p = data; copie dans la variable désignée par p la valeur de la variable data;
- ▶ Arithmétique sur les pointeurs (sauf void *) :
 - ▶ Addition/soustraction d'un entier : déplace le pointeur en mémoire de la taille du désigné;
Exemples :


```
int *p; /* p est un pointeur sur entiers */
p = &i; /* p reçoit l'adresse de i */
p++; /* p pointe sur l'entier suivant i en mémoire */
```
 - ▶ Soustraction de pointeurs vers même type : retourne le nombre d'éléments entre les deux adresses;
 - ▶ ! Toute autre opération est interdite car dénuée de sens !

→ Allocation dynamique (dans le tas = heap) en C (#include <stdlib.h>)

- ▶ Allocation :
 - ▶ void *malloc(size_t taille);
 - ▶ void *calloc(size_t nmemb, size_t taille);
 - ▶ void *realloc(void *ptr, size_t size);
- ▶ Désallocation :
 - ▶ void free(void *ptr)

```
typedef struct point {
    double x, y;
} POINT;
POINT *p;
...
p = (POINT *)malloc(sizeof(POINT));
p->x = 0.0;
p->y = 0.0;
...
free(p);
```

→ Equivalence tableau / pointeur

- ▶ La notation [] peut à la fois être utilisée pour un tableau ou pour une zone désignée par un pointeur.

```
int* p;
p = (int*) malloc(sizeof(int)*100);


[i] est équivalent à *(p+i) et p est équivalent à &p[0]


▶ Inversement, un tableau peut être manipulé comme un pointeur (sur la première case du tableau).
```



```
int tab[100];
*(tab+i) est équivalent à tab[i] et tab est équivalent à &tab[0]
```

- ▶ Attention : `sizeof(p)` ≠ `sizeof(tab)`. `p` a la taille d'une adresse en mémoire : 4 octets pour un processeur 32 bits, 8 octets pour un processeur 64 bits. `tab` a la taille de 100 entiers : 400 octets.

→ Pointeurs sur des pointeurs

- ▶ Il est souvent utile de définir des pointeurs sur des pointeurs sur de type.
- ▶ Exemple : Stocker une image de 10×10 pixels codés en niveau de gris par des entiers.

```
int** p;
/* Allocation */
p = (int **) malloc(sizeof(int*) * 10);
for (i = 0; i < 10; i++) {
    *(p+i) = (int *) malloc(10 * sizeof(int));
}
...
/* Déallocation */
for (i = 0; i < 10; i++) {
    free(*(p+i));
}
free(p);
```

TEST 4

Given the following definition :

```
int n = 4;
double p = 3.14;
```

How do we find the address of variables `n, p`?

```
int *pn = &n;
double *pp = &p;
```

How can we obtain 7.14 using the pointers `pn, pp`?

```
*pp = *pp + *pn;
```

TEST 5

Given :

```
int a = 5, b = 7;
```

we want to write a function to swap the integers, such that by calling

```
swap(&a, &b);
```

we have $a = 7, b = 5$.

What is the prototype of the swap function ?

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

How is swapping done ?

```
void swap(int **x, int **y)
{
    int temp = *x;
    *x = *y;
    *y = temp;
}
```

TEST 6

Given :

```
int a [] = { 11, 24, 37 };
int *p;
p=a;
```

what is the result of the following operations ?

```
(p+2) - p =
*p =
*(p+1) =
*(p+2) - *p =
Answer :
```

```
(p+2) - p = 2
*p = 11
*(p+1) = 24
*(p+2) - *p = 26
```

TEST 7

Given the following strings of different length :

```
char str1[] = "hello";
char str2[] = "death star";
char str3[] = "obi wan";
declare an array of strings containing all of them :
```

```
char * strArray[] = { str1, str2, str3 };
```

Note : strArray contains only pointers, not the characters themselves !

TEST 8

```
#include <stdio.h>
char* getMessage()
{
    char msg[] = "Pointers are fun";
    return msg;
}
int main ()
{
    char* string = getMessage();
    puts(string);
    return 0;
}
```

What is wrong with this code ?

The pointer is invalid after the variable passes out of scope !

TEST 9 (this is difficult)

```
#include<stdio.h>
int main()
{
    static int i,j,k;
    int *(*ptr)[] ;
    int *array[3]={&i,&j,&k};
    ptr=&array;
    j=i+++k+10;
    ++(**ptr);
    printf("Output = %d",***ptr);
    return 0;
}
```

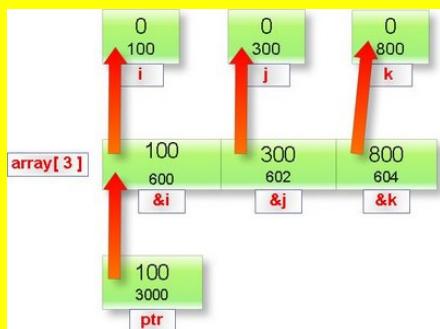
What is the output of this code ?

Output = 10

Let's try to understand...

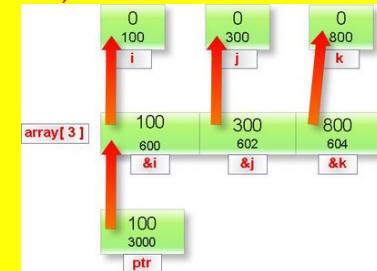
TEST 9 (this is difficult)

```
static int i,j,k;  
int *(*ptr)[];  
int *array[3]={&i,&j,&k};  
ptr=&array;
```

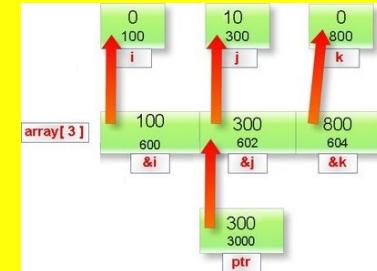


TEST 9 (this is difficult)

```
j=i+++k+10;  
j=i++ +k +10  
j=0 +0 +10 = 10  
note that i++ is resolved after the sum !
```



```
++(**ptr);  
uses the rules : array[0] = *(array+0)  
++(**ptr) => **ptr = **ptr +1  
=> ***ptr = *(array[1]) = *&j = j  
  
printf("Output = %d",***ptr);  
=> ***ptr = *(array[1]) = *&j = j = 10
```



Questions ?

