Outils et règles pour la programmation

LIF7 – Semestre printemps 2013
Julien MILLE
julien.mille@univ-lyon1.fr

LIF7 : Objectifs

- Développer un projet complet
- Outils
- Méthodes
- Travail en groupe
- Règles de programmation

Plan

- **Compilation**
- Compilation de projet (make)
- Débogage (gdb)
- Règles de programmation
- Paramètres de main

Question

- **Quelles sont les 3 étapes réalisées pour générer un exécutable à partir d'un ou plusieurs fichiers source(s) en C ?**

Rappels : compilation

- source C → (pré-processeur) → source C pré-traité (option –E)
- source C pré-traité → (compilateur) → fichier objet (option –c)
- fichiers objets → (éditeur de liens) → exécutable (option –o)

Pré-processeur

- Insère le contenu des fichiers inclus

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

- Interprète les directives de précompilation :

```
#if ...
#endif
#define ...
```

  - Produit un autre source C pré-traité sans directives de précompilation

- Remarque : la précompilation s'apparente à du traitement de texte

Compilateur (compiler)

- Analyse le source C pré-traité (texte)
- Vérifie qu'il n'y a pas d'erreur syntaxique ni d'ambiguïté
- Produit un fichier objet (.o)
  - séquences d'instructions processeur (code machine) pour chaque fonction (le code machine généré dépend de l'architecture)
  - références / lien vers les fonctions non connues (exemple : printf)

7

Editeur de liens (linker)

- Construit l'exécutable
- Assemble tous les fichiers objets et les bibliothèques
- Vérifie que chaque fonction et variable globale est définie une seule fois, soit dans les sources .c, soit dans les bibliothèques (.a ou .so)

8

Un compilateur : GCC

- GCC permet de faire toutes les tâches en une seule fois ou une par une.
- Son comportement est contrôlé par des options:
  - gcc main.c (produit l'exécutable a.out)
  - gcc –E main.c (appelle le pré-processeur, remplace les #include et #define…)
  - gcc –c main.c (produit main.o)
  - gcc main.o (produit l'exécutable a.out)

9

Questions

Erreur de pré-processeur, compilation ou édition de lien ?

- main.c:18: error: `oto' was not declared in this scope
- main.c:18: error: expected `;' before "dansCercle"
- main.c:4:1: unterminated #ifdef
- main.c:16: undefined reference to `poto
- main.c:10: error: too few arguments to function `float calculPi(int)'
- main.c:8:1: warning: "TOTO" redefined
- main.c:16: error: redeclaration of `int TOTO'

10

Questions

- Sens des options de compilation ?

```
-Wall
-c
-o
-ggdb
-lm
-I/usr/include
-L/usr/lib
```

11

GCC : options courantes

- Compilation

```
gcc –Wall –c main.c –o main.o
```

- -Wall affiche tous les messages de prévention (ambigüités, oublis, etc.)
- -c demande la production de fichiers objets
- -o fichier donne le nom du fichier créé.

12

GCC : options courantes

- Édition de lien / création de l'exécutable
  `gcc –o projet main.o –lABC`
- fichier(s).o indique le fichier(s) objet(s) à utiliser
- -o executable[.exe] indique le nom du programme à créer
- **`-lABC`**, indique qu'il faut inclure la bibliothèque (*library*) ABC à l'exécutable.

  Une bibliothèque contient un fichier .a (statique) ou .so (dynamique). Elle correspond souvent avec un en-tête .h

  `-lABC` inclus le fichier libABC.a ou libABC.so au programme (il y a certainement un ou plusieurs **`#include <ABC.h>`** dans les sources)

  Les bibliothèques standards sont dans /usr/lib

13

GCC : options courantes

- Édition de lien / création de l'exécutable
  `gcc –o main main.c –lm –L/usr/lib –I/usr/include`

`-L` indique un chemin vers les bibliothèques (.a)

`-I` indique un chemin vers les fichiers d'en-tête (*headers*) (.h)

Les fichiers d'en-tête (standards sont dans /usr/include

Remarque : –L/usr/lib –I/usr/include est automatiquement ajouté à toute compilation

14

Exemple

- Créer un exécutable à partir des sources main.c vecteur.c matrice.c et triangle.c et en utilisant la bibliothèque m pour les fonctions mathématiques ?

```
gcc -Wall -o main.o -c main.c
gcc -Wall -o vecteur.o -c vecteur.c
gcc -Wall -o matrice.o -c matrice.c
gcc -Wall -o triangle.o -c triangle.c
gcc -o projet main.o vecteur.o matrice.o
  triangle.o -lm
```

Ou en plus court :

```
gcc –o projet main.c vecteur.c matrice.c
   triangle.c -lm
```

15

Plan

- Compilation
- **Compilation de projet (make)**
- Débogage (gdb)
- Règles de programmation
- Paramètres de main

16

Cycle de développement

- Problème → analyse / idées → solution
- Solution → structure de données et algorithmes
- Structure de données et algorithmes → sources C
- Sources C → compilation → exécutable
- Vérification de l'exécutable → (débugger) → modification sources C → compilation

17

Génération de projet : Make

Reconstruire l'exécutable après une modification

- `gcc -o projet *.c` Recompile tous les fichiers
- – On voudrait recompiler uniquement les morceaux nécessaires
  - Recompiler les sources modifiés
  - Reconstruire l'exécutable
- – Exemple : Modification de triangle.c
  - gcc –Wall –o triangle.o –c triangle.c
  - gcc –o projet main.o vecteur.o matrice.o triangle.o –l m

▪ Makefile et la commande make

18

Notions

- Décrire les relations de dépendances entre les fichiers
  - triangle.o est produit par triangle.c et triangle.h
  - projet est produit par triangle.o, vecteur.o …
- Le programme **make** analyse le projet
  - Si un fichier .c est plus récent que le fichier .o
    - Il faut recompiler le fichier .c
  - Si un fichier .o est plus récent que l'exécutable
    - Il faut reconstruire l'exécutable

19

Makefile : description du projet

- Ensemble de règles

```
cible(résultat): sources(dépendances)
<tabulation>commande
```

- Exemple:

```
triangle.o : triangle.c triangle.h
  gcc -ggdb -Wall -o triangle.o -c triangle.c
vecteur.o : vecteur.c vecteur.h
  gcc -ggdb -Wall -o vecteur.o -c vecteur.c
...
projet: main.o vecteur.o matrice.o triangle.o
  gcc -ggdb -o projet main.o vecteur.o \
  matrice.o triangle.o
```

20

Makefile : variables

- On peut aussi déclarer des variables dans un Makefile

```
OBJS=main.o vecteur.o matrice.o triangle.o
FLAGS=-ggdb -Wall -pedantic

projet: $(OBJS)
  gcc $(FLAGS) -o projet $(OBJS)
```

21

Makefile : règles génériques

Pour éviter les répétitions, règles génériques

```
%.o: %.c
     gcc $(CFLAGS) $(INCLUDES) -o $@ -c $<
```

- $@ contient le nom de la cible (résultat)
- $< le nom de la première source
- $^ le nom de toutes les sources

22

Make : utilisation

- **make** tout seul, produit la première règle du Makefile
- **make** projet

  construit "projet", ou exécute la commande de la règle "projet" :
- On peut ajouter des règles

```
clean:
  rm *.o
```

23

Exemple

```
CC = gcc
CFLAGS = -Wall -ggdb
EXEC_NAME = nom_de_votre_exécutable
INCLUDES = -I/usr/include
LIBS = -L/usr/lib -lm
OBJ_FILES = fichier_1.o fichier_2.o

all: $(EXEC_NAME)
$(EXEC_NAME): $(OBJ_FILES)
        $(CC) -o $(EXEC_NAME) $(OBJ_FILES) $(LIBS)
%.o: %.c
        $(CC) $(CFLAGS) $(INCLUDES) -o $@ -c $<
clean:
        rm $(OBJ_FILES)
```

24

Plan

- Compilation
- Compilation de projet (make)
- **Débogage (gdb)**
- Règles de programmation
- Paramètres de main

25

Questions

- Mon programme compile, est-ce gagné ?
- Mon programme ne marche pas, quelle solution ai-je pour le corriger ?
- Ai-je besoin de le comprendre pour le réparer?
- Mon programme fait ceci

```
$ ./bin/prog
Segmentation fault (core dumped)
```

Ca veut dire quoi?

26

Débogage

- Ce n'est pas parce qu'un programme compile qu'il fait ce que l'on veut !
  - 20% du temps à écrire le code
  - 10% du temps à le faire compiler
  - 70% à le déboguer
- 2 manières de déboguer à utiliser en //
  - printf (surtout dans les cas d'une longue itération où tous les cas nous intéressent)
  - **Un débogueur** (ou débugueur, débugger, …)

27

Débogage

Segmentation fault (core dumped)

- **Erreur de segmentation** (en anglais **segmentation fault**, parfois abrégé **segfault**), est un plantage d'une application qui a tenté d'accéder à un emplacement mémoire qui ne lui était pas alloué.
- **Core dump** signifie que le système à sauvegardé tout l'état de la mémoire dans un fichier ./core

28

Débugger / mettre au point avec GDB

- Suivi de l'exécution d'un programme (pas à pas, ou arrêts spécifiques (breakpoints))
- Inspection de la valeur des variables
- Modification de la valeur des variables lors de l'exécution (« en live ») !
- Options de gcc pour utiliser gdb :

  Ajouter –ggdb dans les options de compilation et dans les options de la création de l'exécutable.

```
gcc -Wall -ggdb -o main.o -c main.c
gcc -ggdb -o projet main.o -lm
```

29

GDB

- Utilisation

$ gdb ./prog

(gdb) run (ou juste r)

…

backtrace (bt) pour localiser la fonction et la ligne de l'erreur (si erreur système type SegFault)

list pour afficher la fonction fautive

quit pour sortir de gdb

30

GDB : utilisation courante

Stopper l'exécution du programme

Placer un point d'arrêt

break fonction ou fichier:line

tbreak … (break temporaire)

Reprendre l'exécution

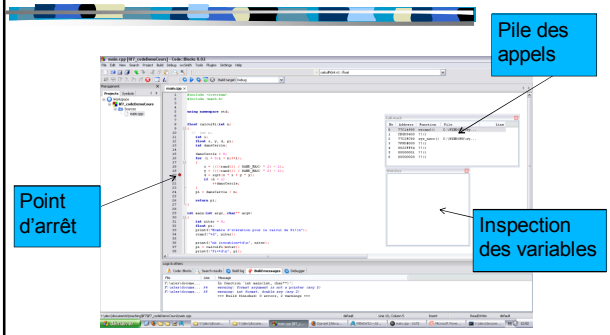
next, **step**, **continue**

Afficher la valeur d'une variable

print variable

31

DEMO codeblocks



32

Plan

- Compilation
- Compilation de projet (make)
- Débogage (gdb)
- **Règles de programmation**
- Paramètres de main

33

Règles de programmation

Objectif : limiter le temps de mise au point et de débogage

Quelques pistes :

- Fonction **assert**
- Qualificatif **const**
- Passage de paramètres
- Initialisation ≠ création
- Accesseurs (get) et mutateurs (set)

34

Question

```
int div(int a, int b)
{
    return a/b;
}

int main()
{
   div(5,0);
   return 0;
}
```

```
int fact (int n)
{
   if (n==0) return 1;
   else return n*fact (n-1);
}

int main()
{
   fact(-4);
   return 0;
}
```

Que se passe-t-il?

35

Robustesse et vérification : *assert*

```
#include <assert.h>
int div(int a, int b)
{
    assert(b!=0);
    return a/b;
}
```

```
int factorielle(int n)
{
    assert(n>=0);
    …
}
```

- But : permet de vérifier une pré-condition de l'algorithmique.
- Le programme se termine si la pré-condition n'est pas vérifiée
- Avantage :
  - assure une sortie « propre » en cas de problème
  - court à écrire
- Inconvénient : message de sortie un peu rude
- **A utiliser sans retenue**

36

Qualificatif **const**

- Rappel : le qualificatif de type **const** déclare au compilateur que cette variable ne sera pas modifiée (lecture seulement).
- Ajoute du sens au code : renseigne sur la manière dont une fonction manipule une variable
- Une « variable constante » peut jouer le même rôle qu'une macro définie à l'aide de la directive #define

Exemple : quelle est la différence ?

```
#define nomFichier "toto.txt"
printf(nomFichier);
```

```
const char *nomFichier ="toto.txt";
printf(nomFichier);
```


Passage de paramètres

- Le paramètre est une donnée (en lecture seule)
  - Passage par valeur : **`void affichage(Tab t)`**
    - Recopie de la variable sur la pile (zone mémoire dédiée, entre autres, aux paramètres et variables locales) → **peut être lourd**
  - Passage par pointeur sur variable constante : **`void affichage(const Tab *pt)`**
    - Seule l'adresse est passée en argument
    - Interdiction de modifier la variable pointée **à la compilation** → **la solution idéale**
- Le paramètre est une donnée-résultat
  - Passage par pointeur : **`void effacer(Tab *pt)`**

  → **pas d'autre choix**

Passage de paramètres

```
typedef struct { int v[10000]; } Tab;

En algo, la fonction d'affichage ne fait que consulter
        => passage de paramètre en 'donnée'
En C
void affichage(Tab t);        // LOURD : 40Ko sur la pile
void affichage(Tab *t);       // Donnée-résultat : efficace mais risqué
void affichage(const Tab *t); // Donnée : efficace et sûr
```

- Passage de paramètres par pointeur (par adresse) quasiment tout le temps
  - const Type * → IN : donnée (lecture seule)
  - Type * → IN-OUT : donnée-résultat (lecture/écriture)
- Remarque : importance du const

Initialisation ≠ création

```
typedef struct {
   int nb_remorques;
   int *poids_remorques; /* le poids de chaque remorque est stocké dans un tableau */
} Camion;

void camInit(Camion *pCam, int nbr)
{
   pCam->nb_remorque = nbr;
   pCam->poids_remorques = (int*)malloc(sizeof(int)*pCam->nb_remorques);
   memset(pCam->poids_remorques, 0, sizeof(int)*pCam->nb_remorques);
}

Camion *camCreer(int nbr)
{
   Camion *pCamNew = (Camion *)malloc(sizeof(Camion));
   camInit(pCamNew, nbr);
   return pCamNew;
}
```

- camInit : initialise une variable de type Camion
- camCreer : crée dans le tas une variable de type Camion puis l'initialise
- La création implique l'initialisation

Initialisation des champs

- L'initialisation d'un composé implique l'initialisation de ses composants

```
typedef struct Roue {...}
typedef struct Cylindre{...};
typedef struct Batterie {...};
typedef struct Alternateur
{...};

typedef struct Moteur {
   Cylindre tabCylindres[4];
   Batterie bat;
   Alternateur alt;
   ...
};

typedef struct Voiture {
   Moteur mot;
   Roue tabRoues[4];
   ...
};
```

```
void initRoue(Roue *pRoue) {...}
void initCylindre(Cylindre *pCyl) {...}
void initBatterie(Batterie *pBat) {...}
void initAlternateur(Alternateur *pAlt) {...}

void initMoteur(Moteur *pMot) {
   int iCyl;
   assert(pMot!=NULL);
   for (iCyl=0; iCyl<4; iCyl++)
      initCylindre(&(pMot->tabCylindres[iCyl]));
   initBatterie(&(pMot->bat));
   initAlternateur(&(pMot->alt));
   ...
};

void initVoiture(Voiture *pVoit) {
   int iRoue;
   assert(pVoit!=NULL);
   initMoteur(&(pVoit->mot));
   for (iRoue=0; iRoue<4; iRoue++)
      initRoue(&(pVoit->tabRoues[iRoue]));
   ...
}
```

- A ré-organiser en plusieurs modules…

Destruction

```
void camLibere(Camion *pCam)
{
   free(pCam->poids_remorques);
   pCam->poids_remorques = NULL;
   pCam->nb_remorques = 0;
}

void camDetruit(Camion **ppCam)
{
   camLibere(*ppCam);
   free(*ppCam);
   *ppCam = NULL;
}

int main()
{
   Camion cam;
   Camion *pCam;
   camInit(&cam, 2);
   pCam = camCreer(2);
   ...
   camLibere(&cam);
   camDetruit(&pCam);
   ...
```

camLibere va avec camInit et

camDetruit va avec camCreer

On doit en appeler le même nombre !

- camLibere : libère les allocations internes à Camion
- camDetruit : libère + détruit aussi la structure

Accesseur et mutateur

- A part dans Camion.c, aucune fonction n'accède directement aux champs de la structure
  - Conserve la cohérence qui peut exister entre plusieurs champs (exemple simple : rayon et aire d'un cercle)
  - Permet de limiter les erreurs et de faciliter le débogage
  - Permet l'écriture de fonctions de haut niveau
  - On peut changer l'implémentation sans changer l'interface… notion de boîte noire

Question : Accesseur et mutateur

```
int main()
{
   Camion cam;
   camInit(&cam, 2);

   cam.nb_remorques = 3; /* nb_remorques est modifiée
                            mais pas la taille réelle du tableau */
   camAffiche(&cam);
}
```

Que se passe t-il?

Accesseur et mutateur

```
int main()
{
   Camion cam;
   camInit(&cam, 2);
   cam.nb_remorques = 3;  /* nb_remorques est changé mais
                             pas la taille réelle du tableau */
   camAffiche(&cam);      /* PLANTAGE!! */
}
```

- Seul le module Camion sait ce qu'il faut faire si le nombre de remorques change !
- Si on le court-circuite → plantage !
- Remarques :
  - le plantage peut intervenir longtemps après
  - le compilateur ne fait aucune vérification
  - le développeur doit se discipliner

Accesseur et mutateur

```
// Dans Camion.h et Camion.c
void camSetNbRemorques(Camion *pCam, int nbr)
{
   /* Allocation d'un tableau à la nouvelle taille */
   int* poidsrem = (int*)malloc(sizeof(int)*nbr);

   /* Recopie des éléments poids_remorques dans un nouveau tableau */
   memcpy( poidsrem, pCam->poids_remorques, sizeof(int)*pCam->nb_remorques);

   /* Mise à jour des champs de *pCam */
   pCam->nb_remorques = nbr;
   free(pCam->poids_remorques);
   pCam->poids_remorques = poidsrem;
}

// Dans main.c
int main()
{
   Camion cam;
   camInit(&cam, 2);
   ...
   camSetNbRemorques(&cam,3); /* nb_remorques change ainsi le tableau */
   camAffiche(&cam);          /* C'est plus juste ! */
}
```

- Les fonctions Set() sont appelées "mutateurs"

Question

```
int main()
{
   Camion cam;
   camInit(&cam, 2);

   printf("poids remorque num 6=%d\n", cam.poids_remorques[6]);
}
```

Que se passe-t-il?

Accesseur et mutateur

```
int main()
{
   Camion cam;
   camInit(&cam, 2);

   printf("poids remorque num 6=%d\n", cam.poids_remorques[6]);
   /* PLANTAGE
      ou alors Affichage étrange
      ou alors Affichage plausible en fonction de la mémoire,
        ce qui rend ce bug non détecté immédiatement!! LE PIRE!! */
}
```

- Idem pour les accesseurs. Seul le module camion sait ce qui est accessible ou non !
- Remarque : dans du code plus important, il est possible que ce bug passe inaperçu!

Accesseur et mutateur

```
int camGetPoidsRemorques(const Camion *pCam, int num_remorque)
{
   assert( num_remorque>=0 && num_remorque<pCam->nb_remorques );
   return pCam->poids_remorques[ num_remorque ];
}

int main()
{
   Camion cam;
   camInit(&cam, 2);

   printf("poids remorque num 6=%d\n", camGetPoidsRemorques(&cam, 6));
   /* Le programme s'arrête plus « proprement » ! */
}
```

- Les fonctions Get() sont appelées “accesseurs”
- « assert » arrête le programme immédiatement lors du problème → Debug beaucoup plus rapide

49

Commentaires

- Aide à la compréhension et à la maintenance du code
- Le développeur commente pour les autres et pour lui-même
- Avant une fonction (en-tête), le commentaire renseigne sur le rôle de la fonction et de ses paramètres
- Dans le corps d’une fonction, il renseigne sur le rôle d’une instruction ou d’un bloc.
- Il ne doit pas être une traduction littérale du code (voir exemples)

50

Commentaires

- Exemple 1 :

```
struct Heros {
   int x, y;
   int nbPointsVie; …};
…
/* Incrémente x (COMMENTAIRE INUTILE) */
/* Déplace le héros vers la droite (COMMENTAIRE UTILE) */
x++;
```

- Exemple 2 :

```
/* Boucle sur i, de 0 à taille (COMMENTAIRE INUTILE) */
/* Compte le nombre d'éléments nuls dans le tableau (COMMENTAIRE
      UTILE) */
for (i=0 ; i<taille ; i++)
   if (tab[i]==0)
      nbZeros++;
```

51

Nommage

- Une variable doit porter un nom explicite, qui donne une indication sur son rôle (quitte à ce que le nom soit plus long)
- Exemple :

Nommage peu explicite :

```
int tab1[…], tab2[…], tab3[…];
```

Nommage explicite :

```
int tabSource[…], tabDest[…], tabTemp[…];
```

- Même remarques pour les fonctions, les structures et les macros

52

Nommage

- Il faut uniformiser la langue le plus possible
- Exemple :

Mélange des langues :

```
bool estValide;   int nbArrays;   char *msgGutenTag;
```

Une seule langue :

```
bool estValide;   int nbTableaux;   char *msgBonjour;
```

ou

```
bool isValid;   int nbArrays;   char *msgGoodMorning;
```

53

Nommage

- Il faut respecter une convention fixe pour la casse (minuscules/majuscules) et la séparation des mots → il existe plusieurs bonnes solutions
- Exemple :

Aucune convention : la casse et les séparations ne sont pas homogènes

```
bool is_valid;
int NBARRAYS;
float CircleRadius;
bool contains_zeros(float *array, int size);
float sum_ofSQUARES(float *Array, int Size);
```

54

Nommage

Respect d'une convention : la casse et les séparations sont homogènes

```
bool is_valid;
int nb_arrays;
float circle_radius;
bool contains_zeros(float *array, int size);
float sum_of_squares(float *array, int size);

ou

bool is_valid;
int nb_arrays;
float circle_radius;
bool ContainsZeros(float *array, int size);
float SumOfSquares(float *array, int size);
```

55

Représentation des données

- Une structure de donnée correspond à une entité propre avec des attributs et des liens éventuels vers d'autres entités
- Les attributs (ou propriétés) et les liens (ou relations) sont les champs de la structure.

```
struct Personne {
    char *nom;
    char *prenom;
    int age;
    enum {HOMME, FEMME} sexe;
    struct Personne *ptrPere;
    struct Personne *ptrMere;
    …
};
```

Attributs

Liens

56

Utilisation des pointeurs

- Dans une structure, un pointeur est utilisé :
  - pour représenter un lien
  - comme un tableau (voir diapo 38 du 1er cours)
- Remarque : le rôle d'un pointeur peut être clarifié par un commentaire

```
struct Personne {
  char *nom; → chaîne (tableau) de caractère
  char *prenom; → chaîne (tableau) de caractère
  …
  struct Personne *ptrPere; → Pointeur (lien) vers le père
  struct Personne *ptrMere; → Pointeur (lien) vers la mère
  struct Personne **tabPtrEnfants; → Tableau de pointeurs (lien) vers les enfants
  int nbEnfants; → taille du tableau tabPtrEnfants
};
```

57

Variables locales ≠ champs

- Dans la structure, seuls les attributs **permanents** doivent apparaître
- Exemple :

```
typedef struct {…} Locomotive;
…
typedef struct {…} Wagon;
void CopierWagon(Wagon *, const Wagon *);
…
typedef struct {
  Locomotive loco;
  Wagon tabWagons[NB_WAGONS_MAX];
  int nbWagons;
} Train;
```

58

Variables locales ≠ champs

- La fonction DupliquerWagonsTrain () a besoin d'un tableau temporaire :

```
void DupliquerWagonsTrain(Train *pTrain)
{
  Wagon tabWagonsTemp[NB_WAGONS_MAX];
  int index;

  for (index=0; index<train.nbWagons; index++)
    CopierWagon(tabWagonsTemp[index], pTrain->tabWagons[index]);

  for (index=0; index<train.nbWagons; index++)
  {
    CopierWagon(pTrain->tabWagons[index*2], tabWagonsTemp[index]);
    CopierWagon(pTrain->tabWagons [index*2+1], tabWagonsTemp[index]);
  }
  pTrain->nbWagons *= 2;
}
```

59

Variables locales ≠ champs

- L'erreur à ne pas commettre est de l'ajouter parmi les champs de la structure
  - Incorrect du point de vue de la conception
  - Complique inutilement la structure

```
typedef struct {
  Locomotive loco;
  Wagon tabWagons[NB_WAGONS_MAX];
  Wagon tabWagonsTemp[NB_WAGONS_MAX]; /* Inutile ici ! */
  int nbWagons;
} Train;
```

- Par définition, une variable temporaire n'est pas un attribut (chaque chose à sa place !)

60

Plan

- Compilation
- Compilation de projet (make)
- Débogage (gdb)
- Règles de programmation
- **Paramètres de main**

Paramètres de main()

```
int main(int argc, char argv[][])
```

- argc = nombre d'arguments
- argv[][] = tableau de tableaux de caractères, chaque ligne du tableau comporte un argument

Paramètres de main()

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

Commande :

```
$ bin/arg -l toto -g truc
```

Résultat :

```
argc=5
arg[0]=bin/arg
arg[1]=-l
arg[2]=toto
arg[3]=-g
arg[4]=truc
```