http://nic.volanschi.free.fr/teaching/c/

Projet de programmation (PG110-CI)

N. Volanschi *

Outils et techniques de programmation impérative en langage C

14 janvier 2020

* En collaboration avec G. Mercier, L. Réveillère

RAPPELS SUR LE C

Opérateurs logiques

Ligne de commande

Lecture sur l’entrée standard

Expressions

		Assoc
x( )	Appel de fonction	
( )	Groupe	
* / %	Multiplier, diviser, modulo (reste de la div)	G
+ -	Addition et soustraction (binaire)	G
> >= < <=	Plus grand (ou égal) Plus petit (ou égal)	G
== !=	Égalité, différent	G
=	Affectation	D

Priorité des opérateurs

Priorité	Opérateurs	Description	Associativité
15	() [ ] -> . ++ --	opérateurs postfix	-
14	++ --	incrément/décrément (prefix)	-
	~	complément à un (bit à bit)	
	!	non unaire	
	& *	adresse et valeur (pointeurs)	
	(*type*)	conversion de type (cast)	
	+-	plus/moins unaire (signe)	
13	* / %	opérations arithmétiques	->
12	+ -	""	->
11	<< >>	décalage bit à bit	->
10	< <= > >=	opérateur relationnels	-> (louche!)
9	== !=	""	-> (louche!)
8	&	et bit à bit	->
7	^	ou exclusif bit à bit	->
6	|	ou bit à bit	->
5	&&	et logique	->
4	||	ou logique	->
3	?:	conditionnel	<-
2	= += -= *= /= %= >>= <<= &= ^= |=	assignations	<-
1	,	séquence	->

Associativité

- Présupposés: opérateurs sur un domaine clos:
  - Op : D x D -> D
  - Alors x Op y Op z représente (x Op y) Op z, ou x Op (y Op z) ?
  - Associativité gauche (« -> »): (x OP y) OP z
  - Associativité droite (« <- »): x OP (y OP z)
- Ex: + : int x int -> int, associativité « -> »
  - x + y + z représente: (x + y) + z, et non x + (y + z)
- Contre-ex: = : Lval x Rval -> Rval,
  - Son associativité « <- » est un abus de langage!
  - x = y = z représente: x = (y = z)
  - … car de toute manière, (x = y) = z, est erroné !
- Contre-ex: < : int x int -> {0, 1}
  - Son associativité « -> » est louche!
  - x < y < z représente (x < y) < z, mais compare z avec 0 ou 1 !
- Contre-ex (opérateurs unaires):
  - Op Op x représente Op (Op x), et x Op Op représente (x Op) Op
- Contre-ex (opérateur ternaire): ? : {0,1} x T x T -> T
  - q1? q2? x : y : z représente q1? (q2? x : y) : z, sans ambiguité
  - q1? x : q2? y : z représente q1? x : (q2? y : z), et non (q1? x : q2)? y : z, louche

Exercice (logic.c)

- Compléter les points de suspension par les valeurs affichées
  - int x = 3, y = 2;
  - printf(”%i\n", x && y); // affiche ...
  - printf(“%i\n", x || y); // affiche ...
  - printf(”%i\n", x & y); // affiche ...
  - printf(“%i\n", x | y); // affiche ...
  - printf(”%i\n", x << y); // affiche ...
  - printf(”%i\n", x >> y); // affiche ...
  - printf(”%i\n", x <= y); // affiche ...
  - printf(”%i\n", x >= y); // affiche ...
  - printf(”%i\n", x % y); // affiche ...
  - printf(”%i\n", x / y); // affiche ...
  - printf(”%i\n", x %= y); // affiche ...
  - printf(”%i\n", x /= y); // affiche ...

Exercice (evens.c)

- Ecrire un programme qui affiche le 1er et le dernier nombres pairs passés sur la ligne de commande, ou un message d'échec si aucun nombre pair ne s'y trouve.
- Exemples:
  - ex/$ ./pairs 1
    no even number
  - ex/$ ./pairs 1 2 3
    1st even = 2
    last even = 2
  - ex/$ ./pairs 1 2 3 4
    1st even = 2
    last even = 4

Exercice (join.c)

- Ecrire un programme (join.c) qui affiche les nombres lus sur l'entrée standard, séparés par virgule et finis par point.
- Exemples:
  - ex/$ echo | ./join
    .
  - ex/$ echo 11 | ./join
    11.
  - ex/$ echo 11 22 33 | ./join
    11,22,33.

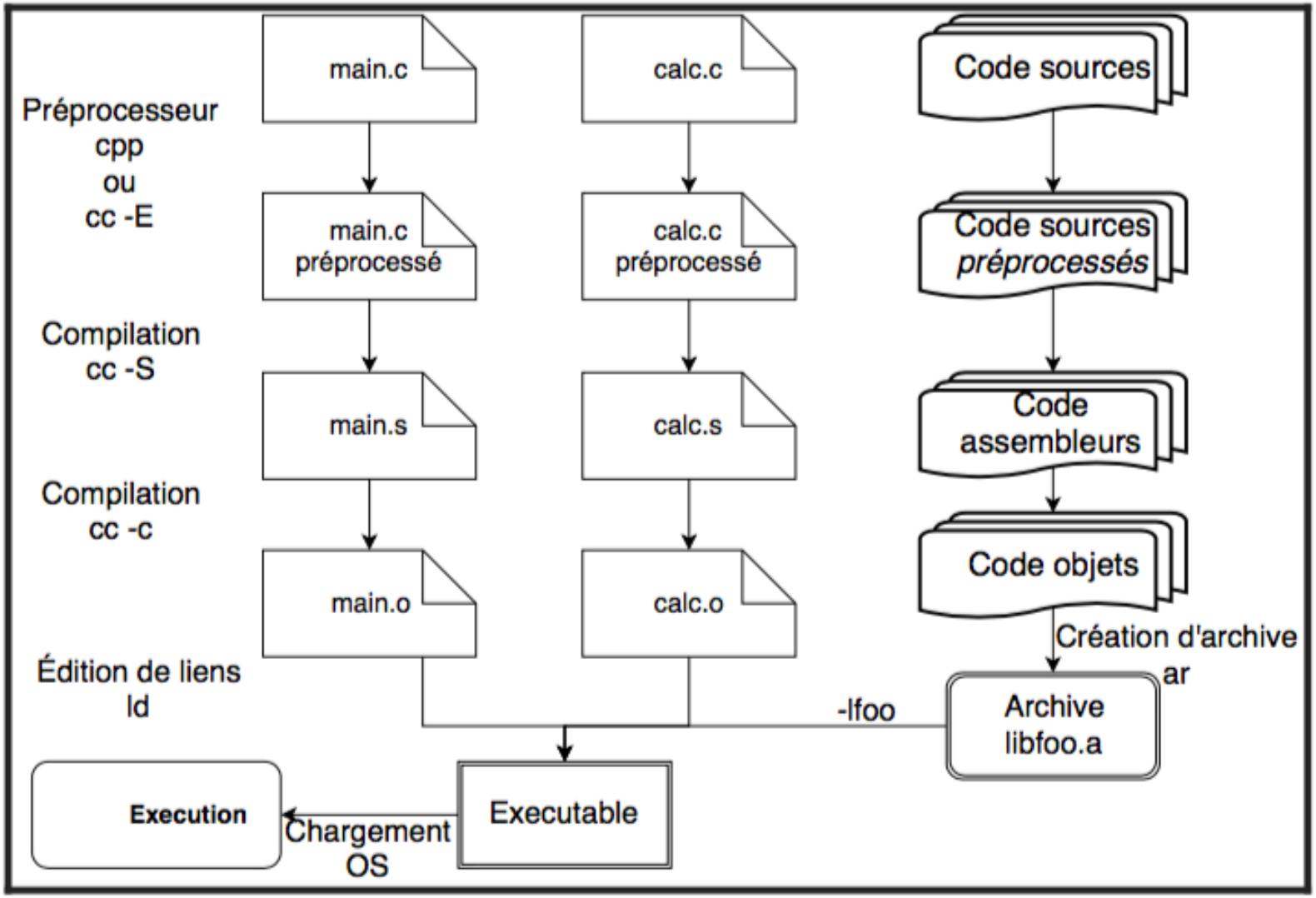
OUTILS

Make

Gdb

Valgrind

CHAÎNE DE COMPILATION



Make

- Utilitaire pour piloter la chaîne de compilation d'une application donnée (cf. un « makefile »)
- Repose sur un langage dédié (domain-specific language, ou DSL)
- Règles de construction (build) d'une application
  - Implicites (prédéfinies)
  - Explicites (définies par le programmeur – plus tard)
- Build incrémental:
  - On reconstruit juste ce qui a changé

```
# Makefile
CC=gcc
CFLAGS=-Wall –std=c99
```

```
ex/$ make example-03
gcc -Wall    example-03.c   -o example-03
ex/$ make example-03
make: `example-03' is up to date.
ex/$ rm example-03
ex/$ make example-03
gcc -Wall    example-03.c   -o example-03
ex/$ make example-03
make: `example-03' is up to date.
ex/$ touch example-03.c
ex/$ make example-03
gcc -Wall    example-03.c   -o example-03
```

Make

- Règles de construction explicites (build) d'une application
  - Partie déclarative: de quels fichiers (« sources ») dépend chaque fichier (« cible »)
  - Partie impérative: comment produire les fichiers cible

```
cible: source …
        commande1
        ...
```

- Variables:
  - Définies par l'utilisateur:
    - Déclaration: NOM=...
    - Utilisation: $(NOM)
  - Prédéfinies: CC (compilateur), CFLAGS (opts. compil.), …
  - Automatiques: cible ($@), 1er source ($<), liste des sources ($^)

Exemple: pile

```
# Makefile
CC=gcc
CFLAGS=-Wall –std=c99
OBJS=stack.o main.o

stack: $(OBJS)
	$(CC) -o $@ $^
stack.o: stack.c stack.h
	$(CC) -c $(CFLAGS) $<
main.o: main.c stack.h
	$(CC) -c $(CFLAGS) $<
clean:
	rm -f $(OBJS) stack
```

```
stack/$ make
gcc -c -Wall stack.c
gcc -c –Wall main.c
gcc -o stack stack.o main.o
stack/$ make clean
rm -f stack.o main.o stack
```

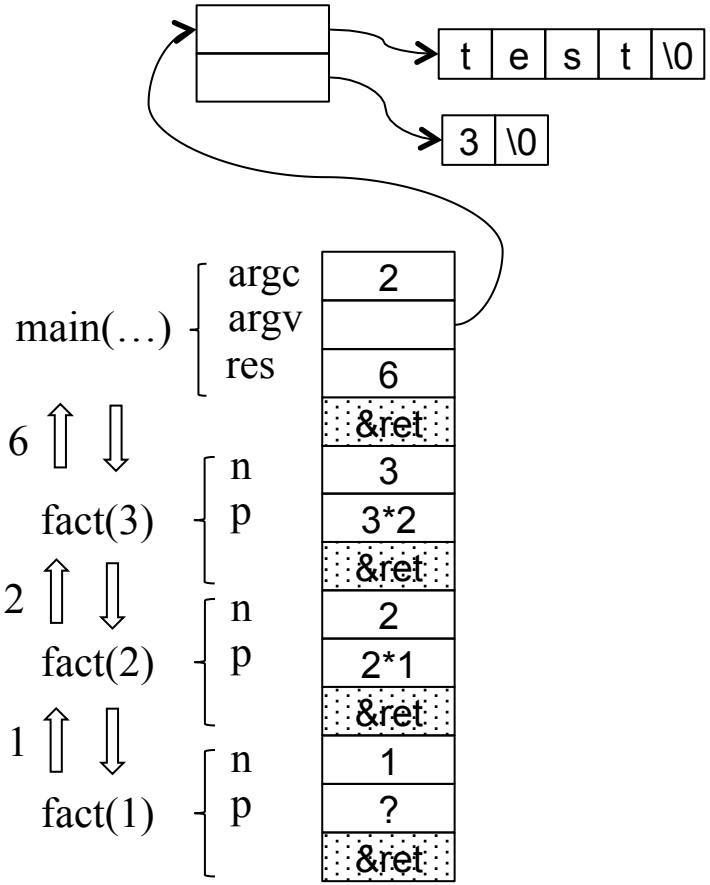
Débogguer: GDB

- But: contrôler l'exécution d'un programme, suivre son état
  - Lancement: « **gdb** prog arg …» ou simplement « **gdb** »
  - NB: il faut avoir compilé le programme avec l'option « -g »
- Commandes:
  - **Help**: Afficher l'aide
  - **Run** prog arg …: (re)Lancer un programme
  - **List**: Afficher le source
  - **Break** ligne/fonction: Définir un point d'arrêt
  - **Continue**: Continuer l'exécution
  - **Print** expr: Évaluer une expression
  - **Display** var: Afficher une variable à chaque pas
  - **Backtrace**: Affiche la pile de fonctions en cours et le no de ligne
  - **Step**: Continuer jusqu'à la ligne suivante
  - **Next**: Continuer jusqu'à la ligne suivante dans la même fonction
  - **Finish**: Continue jusqu'à sortir de la fonction courante

Mécanisme d'appel d'une fonction. Récursion

```
int fact(int n) {
  int p;
  if (n<=1) return (1);
  p = n * fact(n-1); // appel récursif
  return p;
}
int main (int argc, char *argv[]) {
  int res = fact(atoi(argv[1]));
  printf("%i\n", res);
}
```

```
$ make test
gcc -o test test.c
$ ./test 3
6
```



- Empiler:
  - Les arguments
  - Les variables locales
  - L'adresse de retour

Instrumenteur: Valgrind

- But: analyser l'exécution d'un programme pour détecter des bugs, mesurer les ressources (temps, mémoire …), ...
  - Ici: détecter des débordements de mémoire
- Moyen: modifie le programme exéctutable pour y insérer des vérifications, sondes, ...
- Lancement
  - "valgrind commande arg ..."
  - NB: il faut avoir compilé le programme avec l'option « -g »

TECHNIQUES DE PROGRAMMATION LANGAGE

Un langage dédié

Parsing descendant-récursif

Evaluation d’expressions

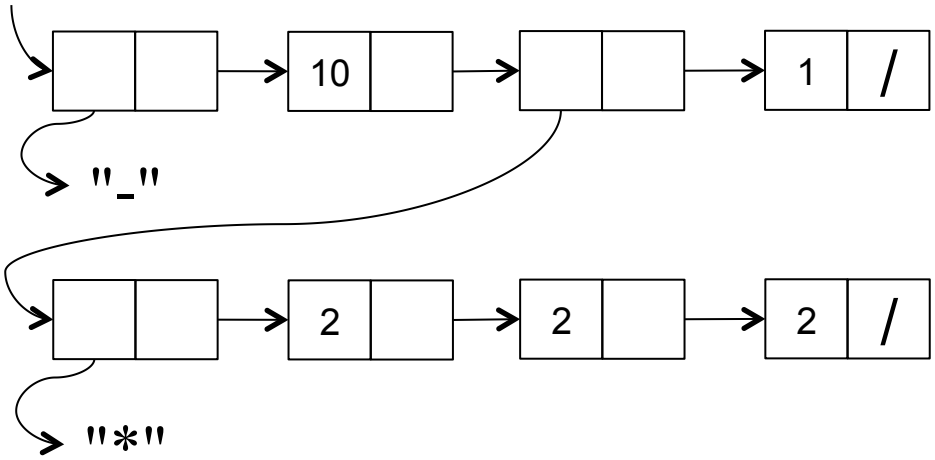
Génération de code d’expressions en format « infix »

Exercice (eval.c)

- Écrire un programme qui:
  - Lit un arbre parenthésé:
    tree ::= int | string | **(** tree* **)**
  - Vérifie si l'arbre est une expression valide:
    expr ::= int | **(** op expr expr+ **)**
    op ::= **+** | **-** | * | /
  - Affiche une expression valide en format infix
  - Évalue une expression valide et affiche son résultat
- Exemples
  - ./eval "(manque parenthese fermante" ➔ parse error
  - ./eval "(ini (mini mani) mo)" ➔ invalid expression
  - ./eval "(- 10 (* 2))" ➔ invalid expression
  - ./eval "(- 10 (* 2 2 2) 1)" ➔ (10-(2*2*2)-1) = 1

Indices (eval.c)

(- 10 (* 2 2 2) 1)



- Représentation expr:
  union(int, char*, cell*);
- Analyse lexicale:
  string | (|)
  char *token();
- Lexeme suivant:
  char lookahead()
- Analyse syntaxique:
  expr parse();
- Affichage:
  void print(expr e);
- Analyse sémantique:
  int valid(expr e);
- Évaluation:
  int eval(expr e);

Exercice (extension de eval.c)

- Étendre l'évaluateur pour:
  - Admettre des expressions valides plus générales:
    expr ::= int | id| **(** op expr expr+ **)** | **(** op1 expr **)** | (= id expr)
    id ::= letter (digit | letter)*
    op ::= **+** | **-** | * | / |
    == | **<** | **>** | **<=** | **>=** | **!=** |
    && | **||**
    op1 ::= **+** | **-** | **!**
  - Afficher une expression valide en format infix
  - Évaluer une expression valide et affiche son résultat
    - La valeur par défaut d'un id est 0
- Exemples
  - ./eval "(|| (< 2 3 1) (== 4 4 3))" ➔ ((2<3<1)||(4==4==3)) = 0
  - ./eval "(= 2 3)" ➔ invalid expression
  - ./eval "(+ (= x 2) (= y 3) (* x y))" ➔ ((x=2)+(y=3)+(x*y)) = 11

Conclusion

- La technique pour implanter un langage dédié consiste en:
  - Analyse lexicale (lexemes, lookahead)
  - Analyse syntaxique (parsing) descendante-récursive:
    - régles déclaratives de grammaire
    - vérification d'un texte par rapport à une grammaire,
    - construction de l'arbre de syntaxe abstraite
  - Analyse statique (validation, typage): expression ayant un sens
  - Évaluation (interprétation)
  - Génération de code (compilation)
- Modulariser les programmes à l'aide d'un Makefile
  - Assure la cohérence entre les sources et l'exécutable produit
  - Permet de re-fabriquer facilement un exécutable pour:
    - débogage ou analyses dynamiques
    - production (optimisé pour la performance)