ENSEIRB 2006-2007 - Première année
PG101 - Programmation en C

TD 6 : Compilation séparée

Cette feuille de TD traite du processus de compilation, de la compilation séparée et de la modularisation du code, au travers de l'implantation d'une structure de liste chaînée

Dans votre répertoire nommé `TDc/`, créez un sous-répertoire `TDc/td6/` (avec la commande `mkdir` dans un terminal). Positionnez-vous dans ce sous-répertoire (avec la commande `cd`), vous y resterez toute cette séance.

1 Le processus de compilation

Exercice 1: Pré-compilation

Copiez-collez le programme suivant dans un buffer `emacs` nommée `prog1.c`.

```
#include <stdio.h>

#define N 10

int main() {
  int t[N], i;

  /* Remplissage de t */
  for (i=0; i<N; i++)
    t[i] = i;

  /* Affichage de t */
  printf("t: ");
  for (i=0; i<N; i++)
    printf("%d ", t[i]);
  printf("\n");

  return 0;
}
```

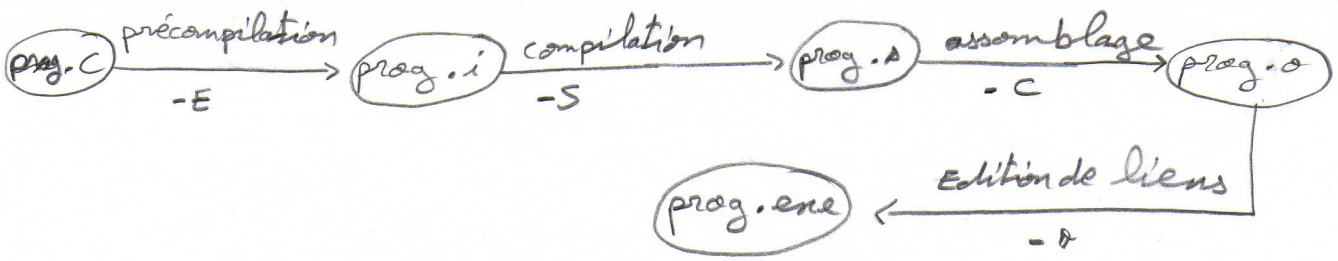
Exécutez la commande : `gcc -E prog1.c -o prog1.i`. Ouvrez le fichier `prog1.i` dans `emacs` et expliquez ce que vous observez (vous pourrez vous servir de la page `man` de `gcc` pour connaître la signification de l'option `-E`) précompilation

Exercice 2: Compilation

1. Exécutez maintenant la commande : `gcc -S prog1.i -o prog1.s`. Ouvrez le fichier `prog1.s` dans `emacs` et expliquez ce que vous observez (là encore, pensez à utiliser la page `man` de `gcc`).
2. La même commande a été exécutée sur une autre machine, voici un extrait du fichier `prog1.s` obtenu. À quoi les différences observées sont-elles dues ?

```
.section __TEXT,__text,regular,pure_instructions
.section __TEXT,__picsymbolstub1,symbol_stubs,pure_instructions,32
.machine ppc
.cstring
.align 2
LC0:
.ascii "t: \0"
.align 2
LC1:
.ascii "%d \0"
.align 2
LC2:
.ascii "\12\0"
.text
.align 2
.globl _main
_main:
mflr r0
stmw r30,-8(r1)
stw r0,8(r1)
stwu r1,-128(r1)
mr r30,r1
bcl 20,31,"L00000000001$pb"
"L00000000001$pb":
mflr r31
li r0,0
stw r0,56(r30)
b L2
L3:
lwz r0,56(r30)
slwi r2,r0,2
addi r0,r30,56
add r2,r2,r0
addi r2,r2,4
lwz r0,56(r30)
stw r0,0(r2)
lwz r2,56(r30)
addi r0,r2,1
stw r0,56(r30)
L2:
lwz r0,56(r30)
```

```
cmpwi cr7,r0,9
ble cr7,L3
addis r2,r31,ha16(LC0-"L00000000001$pb")
la r3,lo16(LC0-"L00000000001$pb")(r2)
bl L_printf$LDBLStub$stub
li r0,0
stw r0,56(r30)
b L5
L6:
lwz r0,56(r30)
slwi r2,r0,2
addi r0,r30,56
add r2,r2,r0
addi r2,r2,4
lwz r0,0(r2)
addis r2,r31,ha16(LC1-"L00000000001$pb")
la r3,lo16(LC1-"L00000000001$pb")(r2)
mr r4,r0
bl L_printf$LDBLStub$stub
lwz r2,56(r30)
addi r0,r2,1
stw r0,56(r30)
L5:
lwz r0,56(r30)
cmpwi cr7,r0,9
ble cr7,L6
addis r2,r31,ha16(LC2-"L00000000001$pb")
la r3,lo16(LC2-"L00000000001$pb")(r2)
bl L_printf$LDBLStub$stub
li r0,0
mr r3,r0
lwz r1,0(r1)
lwz r0,8(r1)
mtlr r0
lmw r30,-8(r1)
blr
.section __TEXT,__picsymbolstub1,symbol_stubs,pure_instructions,32
.align 5
L_printf$LDBLStub$stub:
.indirect_symbol _printf$LDBLStub
mflr r0
bcl 20,31,"L00000000001$spb"
"L00000000001$spb":
mflr r11
addis r11,r11,ha16(L_printf$LDBLStub$lazy_ptr-"L00000000001$spb")
mtlr r0
lwzu r12,lo16(L_printf$LDBLStub$lazy_ptr-"L00000000001$spb")(r11)
```




```
mtctr r12
bctr
.lazy_symbol_pointer
L_printf$LDBLStub$lazy_ptr:
.indirect_symbol _printf$LDBLStub
.long dyld_stub_binding_helper
.subsections_via_symbols
```

Exercice 3: Assemblage

Exécutez la commande : `gcc -c prog1.s -o prog1.o`. Que contient le fichier `prog1.o` ? Que fait l'option `-c` de `gcc` ?

Exercice 4: Édition des liens

Exécutez la commande : `gcc -o prog1 prog1.o`. Que contient le fichier `prog1` ?

Exercice 5: Le processus de compilation en pratique

Résumez le processus de compilation utilisé en pratique pour la compilation de programmes C

2 Code modulaire

Lors de la séance précédente, nous avons vu que le langage C permet de définir de nouveaux types. D'un point de vue syntaxique, ces nouveaux types sont de simples agglomérations de valeurs, et il est possible de les manipuler directement via les opérateurs du langages (`[.]` pour les tableaux, `.` ou `->` pour les structures et les pointeurs sur les structures, etc).

Par contre, d'un point de vue algorithmique et méthodologique, ceci n'est pas souhaitable. Prenons l'exemple de la structure `point` :

```
struct point {
  float x, y;
};
```

Tout programme peut afficher ou modifier une valeur de type `struct point` en accédant directement aux champs `x` et `y`. Supposons que nous souhaitions enrichir le type point avec une information de couleur :

```
enum couleur {
  ROUGE, BLEU, VERT, BLANC, NOIR, JAUNE
};

struct point {
  float x, y;
  enum couleur c;
};
```

Il faut alors reprendre tout les programmes utilisant des variables de type `struct point` pour qu'ils affichent, modifient, etc, le nouveau champ `c` ajouté à la structure.

Ceci n'est évidemment pas souhaitable. À la manière des «Types Abstraits de Données (TAD)» vus en algorithmique, nous associons à un type non seulement une représentation en mémoire (son type du point de vue du langage C), mais aussi une liste d'opérations qui permettent d'agir sur les valeurs de ce type. Tout comme pour les TAD, il est alors implicite que l'utilisateur *doit* exclusivement utiliser ces opérations pour agir sur ces valeurs.

Au travers des fichiers d'interface `.h` et d'implantation `.c`, le langage C offre les outils nécessaire à une méthodologie d'analyse descendante et modulaire de la programmation.

Exercice 6: Une première approche modulaire

Soit le programme suivant :

```
#include <stdio.h>

struct point {
  float x, y;
};

struct point point__creer(float x, float y) {
  struct point p;

  p.x = x;
  p.y = y;

  return p;
}

struct point point__copier(struct point p) {
  return p;
}

int point__egal(struct point p1, struct point p2) {
return ((p1.x == p2.x) && (p1.y == p2.y));
}

void point__afficher(struct point p) {
  printf("(%f, %f)\n", p.x, p.y);
}

int main() {
  struct point p1, p2;

  p1 = point__creer(3.2, 4.7);
  point__afficher(p1);
  p2 = point__copier(p1);
  point__afficher(p2);
```

① gcc -Wall -c point.c
↓
point.o

compilation
② gcc -Wall -c prog3.c
↓
prog3.o

Edition des liens
③ gcc -o prog3 prog3.o point.o

```
  return 0;
}
```

1. Nous voulons créer une bibliothèque dédiée au type `struct point`. En vous basant sur le code existant, écrivez le fichier de declaration `point.h` et le fichier d'implantation `point.c` de ce type. Donnez la commande de compilation pour la bibliothèque
2. Modifiez alors le programme pour qu'il utilise votre bibliothèque. Donnez la commande de compilation du programme, puis celle d'édition des liens
3. Exécutez le programme d'origine et le programme modularisé et vérifiez qu'ils produisent bien le même résultat
4. Pourquoi avons-nous écrit une fonction `point_copie` alors que le type `struct point` est une structure et qu'il suffirait d'utiliser l'opérateur d'affectation `=` pour obtenir le même résultat ?

Exercice 7: Réutilisation du code et approche modulaire

Écrivez un programme utilisant la bibliothèque `point` écrite à l'exercice précédent qui ne comprend qu'une fonction principale `main` qui crée puis affiche un point. Donnez les commandes de compilation et d'édition des liens pour ce programme

Exercice 8: Maintenance et approche modulaire

Nous souhaitons maintenant ajouter une information de couleur à la structure `point` comme indiqué précédemment.

1. Que faut-il modifier ? Dans le cadre de cette approche modulaire, que doit-on préserver autant que possible ?
2. Effectuez les modifications les plus conservatives possibles
3. Donnez les commandes de compilation et d'édition des liens nécessaires à l'obtention des exécutables pour nos deux programmes

Exercice 9: Un type liste

Lancez le programme `xfig` depuis un terminal. Sélectionnez l'outil POLYLINE. Vous pouvez alors dessiner une ligne polygônale : le premier clic de souris positionne le premier point, le second clic positionne un second point, et ainsi de suite jusqu'à ce que vous cliquiez sur le bouton du milieu pour positionner le dernier point. Une fois la ligne tracée, il est possible de supprimer des points et de les déplacer.

Il est nécessaire au programme `xfig` de stocker en mémoire l'image tracée par l'utilisateur. En particulier, il stocke la séquence des points qui définissent la ligne polygônale. Nous cherchons maintenant à développer un type permettant de stocker une ligne polygônale comme le ferait `xfig`.

1. En supposant que nous ayons défini un type `ligne` qui permet de représenter une ligne polygônale, donnez les prototypes des fonctions qu'il est utile d'implanter sur cette structure de données (inspirez-vous du paragraphe précédent, des exercices précédents, et des questions suivantes)

2. Sachant qu'une ligne est une séquence de points, nous proposons de définir le type `ligne` comme :

   ```
   #define MAX_PT  30
   typedef struct point ligne[MAX_PT];
   ```

   Expliquez pourquoi ceci n'est pas satisfaisant

3. Proposez une structure de données plus adaptée et la définition du type `ligne` correspondant. Vous écrirez alors le fichier d'interface `ligne.h` de votre bibliothèque

4. Implantez les fonctions décrites à la première question dans le fichier implantation `ligne.c` de votre bibliothèque

5. Écrivez un programme constitué d'une fonction principale, qui crée une ligne polygônale par ajouts successifs de 5 points, affiche cette ligne, retire le 3ème point et affiche la ligne ainsi obtenue

6. Donnez les commandes de compilation et d'édition des liens pour votre programme. Compilez-le, et expliquez l'erreur signalée.

7. Corrigez cette erreur, compilez votre programme, et vérifiez que le résultat produit est convenable