Mise à niveau en C

Pointeurs et allocation dynamique

Enseignant: P. Bertin-Johannet

Mémoire et addressage

- Dans la mémoire, les octets sont tous identifiés par une addresse
- Par exemple, le code ci-dessous pourrait être représenté ainsi :

```
int i = 5;
char a = 'a';
int j = 24;
```

Valeurs	Addresses
5	0xff17065
	0xff17066
97	0xff17067
24	0xff17068
	0xff17069

Pointeurs

- On peut créer des variables contenant une addresse en mémoire, elles sont alors appelées pointeur
- Pour déclarer une variable de type de pointeur on ajoute une étoile (∗) après le type :

```
int i ; // i est un int
int* iptr ; // iptr est un pointeur vers un int
char* cptr ; // cptr est un pointeur vers un char
```

Pour récupérer l'addresse d'une variable on utilise l'opérateur &

```
int i ; // i est un int
int* iptr = &i; // iptr contient l'addresse de i
```

Exemple

Programme:

```
int i = 5;
char a = 'a';
int* iptr = &i;
char* aptr = &a;
```

Mémoire:

Valeurs	Addresses
	0xff150
	0xff151
	0xff152
	0xff153
	0xff154
	0xff155
	0xff156
	0xff157
	0xff158
	0xff159
	0xff15a

Exemple

Programme:

```
int i = 5;
char a = 'a';
int* iptr = &i;
char* aptr = &a;
```

Mémoire:

Valeurs	Addresses
5	0xff150
	0xff151
	0xff152
	0xff153
	0xff154
	0xff155
	0xff156
	0xff157
	0xff158
	0xff159
	0xff15a

Exemple

Programme:

```
int i = 5;
char a = 'a';
int* iptr = &i;
char* aptr = &a;
```

Mémoire:

Valeurs	Addresses
5	0xff150
	0xff151
97	0xff152
	0xff153
	0xff154
	0xff155
	0xff156
	0xff157
	0xff158
	0xff159
	0xff15a

Exemple

Programme:

```
int i = 5;
char a = 'a';
int* iptr = &i;
char* aptr = &a;
```

Mémoire:

Valeurs	Addresses
5	0xff150
	0xff151
97	0xff152
0xff150	0xff153
	0xff154
	0xff155
	0xff156
	0xff157
	0xff158
	0xff159
	0xff15a

Exemple

Programme:

```
int i = 5;
char a = 'a';
int* iptr = &i;
char* aptr = &a;
```

Mémoire:

Valeurs	Addresses
5	0xff150
	0xff151
97	0xff152
0xff150	0xff153
	0xff154
	0xff155
	0xff156
0xff153	0xff157
	0xff158
	0xff159
	0xff15a

Utilisation des pointeurs

- À partir d'un pointeur, on peut accèder à la valeur vers laquelle il pointe avec l'opérateur $*$

```
int* iptr  = &i; // iptr pointe vers un entier
int j = *iptr;  // j est la valeur vers laquelle pointe iptr
```

- On peut afficher l'addresse avec printf en utilisant le format `"%p"` :

```
int i ; // i est un int
int* iptr  = &i; // iptr contient l'addresse de i
printf("l'addresse de i est %p et sa valeur est%d", iptr, *iptr) ;
```

Pourquoi utiliser un pointeur

Un pointeur peut permettre de:

- Créer une fonction qui modifie une variable exterieure à la fonction
- Créer une fonction qui renvoie plusieurs valeurs
- Acceder à de la mémoire hors de la pile de fonctions
- Utiliser des tableaux de taille dynamique

- Maitriser précisément l’usage de la mémoire

Exemples d'usage

- Une fonction qui modifie les arguments passés en paramètres

```c
void set_to_n(int n, int* a, int* b, int* c){
  *a = n; // on écrit la valeur de n à l'addresse vers laquelle pointe a
  *b = n; // idem à l'addresse vers laquelle pointe b
  *c = n; // idem pour c
}

int main(){
  int a, b, c;
  set_to_n(5, &a, &b, &c); // on passe les addresses de a, b et c
  printf("values of a, b, c : %d, %d, %d\n", a, b, c); // affiche 5, 5, 5
}
```

Exemples d'usage

- Une fonction qui modifie les arguments passés en paramètres afin de renvoyer plusieurs valeurs

```c
void division_avec_reste(int a, int b, int* resultat, int* reste){
  *resultat = a / b;
  *reste = a % b;
}

int main(){
  int resultat, reste;
  division_avec_reste(35, 8, &resultat, &reste);
  printf("35/8 = %d reste %d\n", resultat, reste);
  // affiche "35/8 = 4 reste 3"
}
```

Arithmétique de pointeurs

- Additionner un nombre à un pointeur revient à faire avancer le pointeur dans la mémoire.
- Le pointeur avance du nombre d'octet occupé par le type vers lequel il pointe
- Si un int occupe 4 octets, ajouter 1 à un pointeur vers un int le déplacera de quatres cases mémoire.

Exemple

Programme:

```
int i = 5;
int j;
int* iptr = &i;
int* jptr = iptr + 1;
*jptr  = 4;
```

Mémoire:

Valeurs	Addresses
	0xff150
	0xff151
	0xff152
	0xff153
	0xff154
	0xff155
	0xff156
	0xff157
	0xff158
	0xff159
	0xff15a
	0xff15b

Exemple

Programme:

```
int i = 5;
int j;
int* iptr = &i;
int* jptr = iptr + 1;
*jptr  = 4;
```

Mémoire:

Valeurs	Addresses
5	0xff150
	0xff151
	0xff152
	0xff153
	0xff154
	0xff155
	0xff156
	0xff157
	0xff158
	0xff159
	0xff15a
	0xff15b

Exemple

Programme:

```
int i = 5;
int j;
int* iptr = &i;
int* jptr = iptr + 1;
*jptr  = 4;
```

Mémoire:

Valeurs	Addresses
5	0xff150
	0xff151
??	0xff152
	0xff153
	0xff154
	0xff155
	0xff156
	0xff157
	0xff158
	0xff159
	0xff15a
	0xff15b

Exemple

Programme:

```
int i = 5;
int j;
int* iptr = &i;
int* jptr = iptr + 1;
*jptr  = 4;
```

Mémoire:

Valeurs	Addresses
5	0xff150
	0xff151
??	0xff152
	0xff153
0xff150	0xff154
	0xff155
	0xff156
	0xff157
	0xff158
	0xff159
	0xff15a
	0xff15b

Exemple

Programme:

```
int i = 5;
int j;
int* iptr = &i;
int* jptr = iptr + 1;
*jptr  = 4;
```

Mémoire:

Valeurs	Addresses
5	0xff150
	0xff151
??	0xff152
	0xff153
0xff150	0xff154
	0xff155
	0xff156
	0xff157
0xff152	0xff158
	0xff159
	0xff15a
	0xff15b

Exemple

Programme:

```
int i = 5;
int j;
int* iptr = &i;
int* jptr = iptr + 1;
*jptr  = 4;
```

Mémoire:

Valeurs	Addresses
5	0xff150
	0xff151
4	0xff152
	0xff153
0xff150	0xff154
	0xff155
	0xff156
	0xff157
0xff152	0xff158
	0xff159
	0xff15a
	0xff15b

Tableaux et pointeurs

- Un tableau est un pointeur vers son premier élément.
- Accéder à l'élément n d'un tableau revient à y additionner n puis récupérer la valeur vers laquelle pointe l'addresse.

```
int tab[4];
tab[2] = 0; // cette ligne
*(tab + 2) = 0; // est transformée en cette ligne par le compilateur
```

Allocation statique de la mémoire

- On parle d'allocation quand on réserve une partie de la mémoire
- Jusqu'a présent nous avons pu réserver des octets dans la mémoire en déclarant des variables

```
int i ; // des octets sont reservés pour i
char c ; // un octet est reservé pour c
int* ptr; // des octet sont reservé pour ptr
```

- Chaque fonction, lorsqu'elle est appelée, réserve juste assez de mémoire pour contenir toutes ses variables ainsi que les arguments.
- Le calcul de la mémoire utilisée par une fonction est habituellement fait **au moment de la compilation**.

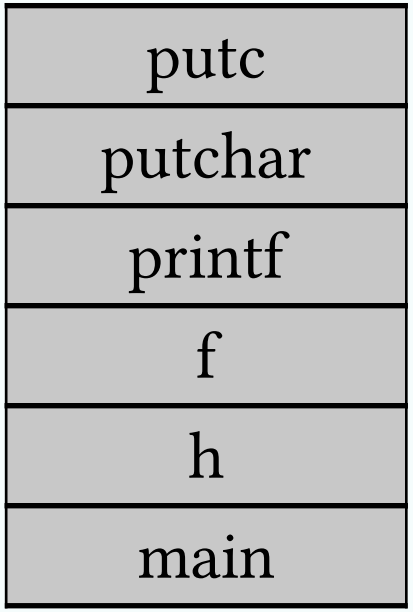
- Il est donc difficile de changer dynamiquement la mémoire occupée par les variables d'une fonction.

La pile d'appels

- Chaque appel de fonction alloue de la mémoire pour ses variables à la suite de la mémoire utilisée par la fonction précédente
- Elle alloue aussi de la mémoire pour stocker des informations permettant de reprendre l'execution de la fonction précédente (addresse de la prochaine instruction, addresse de début de la fonction, etc...)
- La partie de la mémoire ainsi utilisée est appelée la pile (ou stack en anglais)

```
void f(){printf();}
void h(){f();}
int main(){h();}
```

Mémoire



Allocation dynamique sur la pile

L'allocation dynamique de mémoire dans la pile est possible en utilisant principalement deux méthodes.

- Les tableaux de tailles variables (Variable Length Array ou VLA en anglais). La mémoire reservée l'est uniquement tant que la variable existe.
- La fonction `alloca()`. La mémoire reservée l'est jusqu'a la fin de l'execution de la fonction.

```
int n = rand();
if (n > 0){
  char tableau[n]; // ici la mémoire reservée par tableau est calculée pendant
l'execution
  char* tableau2  = alloca(n); // idem
} // la mémoire reservée par tableau n'est plus valide à partir d'ici
// la mémoire resrvée par tableau2 sera valide jusqu'a la fin de la fonction
```

Désavantages de l'allocation dynamique sur la pile

- Le nombre d'instructions utilisées augmente.
- La pile est limitée en mémoire (consultable avec : `ulimit -a`) et la nature imprévisible de l'allocation dynamique entraine un fort risque d'erreur stack overflow.
- La mémoire allouée n'est reservée qu'un temps et le risque de l'utiliser après sa validité est accru.
- Pour ces raisons, l'utilisation de tableaux de tailles variables est interdite ou fortement déconseillée dans de nombreux projets (par exemple le noyau linux).

Allocation dynamique sur le tas

- Il est possible de réserver de la mémoire dans une autre zone du programme appelée le **tas** (en anglais **heap**).
- La fonction `malloc()` accepte en argument le nombre d'octets à reserver.
- Elle renvoie un pointeur vers le début de la mémoire reservée.
- La mémoire réservée est valide **jusqu'a la fin de l'execution du programme**.

```
int* a = malloc(8); // on reserve 8 octets pour un tableau d'entiers
a[0] = 5;
a[1] = 4;
printf("%d\n", *(a + 1));
// affiche 4
```

L'opérateur sizeof

- Pour calculer la taille mémoire occupée par un type, on peut utiliser l'opérateur `sizeof`.
- sizeof prend en argument un **type** et renvoie le nombre d'octets qu'il occupe.
- Cela permet de facilement allouer un tableau sur le tas.

```
float* tab = malloc(9 * sizeof(float));
// ici on reserve assez de place pour 9 floats quelque soit leur taille mémoire.
```

Libération de la mémoire

- Si l'on ne fait rien, la mémoire reservé par malloc l'est jusqu'à la fin de l'execution.
- Lorsqu'on ne s'en sert plus, on doit manuellement la "libérer" en utilisant `free()`.
- Tant qu'on ne la libère pas, la mémoire reservée ne sera pas utilisable ailleurs.

```
float* tab = malloc(9 * sizeof(float)); // on reserve
...
free(tab); // on libère
```

Fuite mémoire

- Il est possible de ne plus avoir accès au pointeur renvoyé par malloc
- La mémoire reste reservée mais on ne peut plus la libérer
- On appelle ça une fuite mémoire

```
if (1){
  char* tab = malloc(9); // on reserve de la place pour 9 chars
  ...
}
int a = 12;
free(??); // on ne peut plus libérer la mémoire car on n'a plus
accès au pointeur
```

Mise en pratique

```
printf("

         _______
        < TP 4 >
         -------
                \   ^__^
                 \  (oo)\_______
                    (__)\       )\/\
                        ||----w |
                        ||     ||

        ")
```