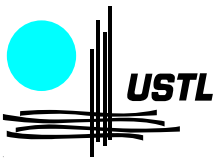

Licence d'informatique
Module de Pratique du C

Travaux dirigés

Convention d'appel de fonction

Philippe MARQUET

Décembre 2008

1 Convention d'appel et organisation de la pile

Une convention d'appel est une méthode qui assure, lors d'un appel de fonction, la cohérence entre ce qui est réalisé par la fonction appelante et la fonction appelée. En particulier, une convention d'appel précise comment les paramètres et la valeur de retour sont passées, comment la pile est utilisée par la fonction.

Plusieurs conventions sont utilisées :

- `__cdecl` est la convention qui supporte la sémantique du langage C et en particulier l'existence de fonctions variadiques (fonctions à nombre variable de paramètres, par exemple `printf()`). Cette convention est le standard de fait, en particulier sur architecture x86 ;
- `__stdcall` est une convention qui suppose que toutes les fonctions ont un nombre fixe de paramètres. Cette convention simplifie le nettoyage la pile après un appel de fonction ;
- `__fastcall` est une convention qui utilise certains registres pour passer les premiers paramètres de la fonction.

Sur architecture Intel x86, la pile croit suivant les adresses décroissantes.

Suite à l'appel d'une fonction `f()` tel le suivant

```
int
f(int a, int b, int c)
{
    int i=123, j=456;

    return a+b+c+i+j;
}
```

```
int
main()
{
    int r;

    r = f(1, 2,3);

    ...
```

suivant la convention `__cdecl`, l'organisation de la pile contient selon les adresses décroissantes :

- les valeurs effectives des paramètres déposés par `main()`, dans l'ordre inverse de leur apparition. On a donc la valeur 3, puis la valeur 2, et la valeur 1 ;
- une sauvegarde de la valeur du registre d'instruction et du registre de contexte qui seront utilisées pour retourner dans le contexte de `main()` suite à l'appel de `f()` ;
- la mémoire réservée pour l'allocation des variables automatiques de la fonction `f()` : `i`, puis `j`.

(La valeur de retour de la fonction est transmise de `f()` à `main()` via le registre `%eax`.)

Exercice 1 (Un dessin vaut...)
Dessinez l'état de la pile suite à l'appel de la fonction `f()`. □

2 Fonctions à nombre variable de paramètres

Une fonction variadique est une fonction à nombre variable de paramètres. La fonction `printf()` est un exemple de telle fonction. D'une invocation à l'autre, la fonction sera appelée avec un nombre de paramètres différent. Typiquement, la valeur du premier paramètre permet de connaître le nombre de paramètres effectivement passés.

Si l'on connaît la convention d'appel utilisée (qui peut dépendre de l'architecture, du système et du compilateur), il est possible de déterminer l'adresse d'un paramètre donné en fonction de

- l'adresse du premier paramètre ;
- la taille des paramètres précédents.

Exercice 2 (Somme des paramètres, à la main)
Donnez la définition de la fonction variadique

```
int sum_them_all(int nargs, ...);
```

qui retourne la somme des `nargs` entiers passés en paramètre. La convention `__cdecl` sur Intel X86 sera supposée utilisée. □

(• ***Éléments de solution 2***

- Syntaxe `...` pour les paramètres variadiques.
- Adresse du premier paramètre suivant `nargs` :

  ```
  &nargs + 1
  ```

  arithmétique sur les pointeurs d'entiers.
- Définition de la fonction

  ```
  int
  sum_them_all(int nargs, ...)
  {
      int sum = 0;
      int *p = &nargs+1;

      while (nargs--)
          sum += *p++;
      return sum;
  }
  ```

•)

Exercice 3 (Somme flottante)
Traitez de même de la fonction travaillant sur des valeurs flottantes :

```
float f_sum_them_all(unsigned nargs, ...);
```

□

(• ***Éléments de solution 3***

- L'adresse du premier paramètre flottant :

  ```
  float *pf = (float *) (nargs +1);
  ```

  les parenthèses sont nécessaires ; `(float *) nargs +1` étant lu `((float *) nargs) +1;` . Or on veut de l'arithmétique sur les pointeurs d'*entiers*.
- Et le code de la fonction

  ```
  float
  f_sum_them_all(int nargs, ...)
  {
      float sum = 0.0;
      float *pf = (float $)(&nargs+1);

      while (nargs--)
          sum += *pf++;
      return sum;
  }
  ```

•)

3 Bibliothèque `stdarg.h`

Afin de garantir la portabilité de cette gestion d'un nombre variable de paramètres, on utilise habituellement les macros définies dans `<stdarg.h>`.

```
float f_sum_them_all_std(int nargs, ...)
{
    float sum = 0.0;
    int i;
    va_list ap;

    va_start(ap, nargs);
    for (i=0; i<nargs; i++)
        sum += va_arg(ap, float);
    va_end(ap);
    return sum;
}
```

Le type `va_list` permet de déclarer une variable qui référencera le prochain argument de la liste des paramètres. Cette variable est initialisée par `va_start()` et référencera l'argument suivant celui donné en paramètre. La macro `va_arg()` permet de récupérer la valeur du paramètre suivant dont on spécifie le type; `va_arg()` avance aussi la référence sur l'argument suivant. Enfin, `va_end()` assure le nettoyage.

Exercice 4 (Des macros ?)
Expliquez pourquoi `va_start()`, `va_arg()`, et `va_end()` sont des macros et non pas des fonctions. □

(• ***Éléments de solution 4*** Des tas de plus ou moins bonnes rasons, mais le fait de devoir passer un type en paramètre est bloquant. •)

Exercice 5 (Mon `stdarg.h` pour `__cdecl` sur x86)
Donnez une définition du type `va_list` et des macros `va_start()`, `va_arg()`, et `va_end()`. □

(• ***Éléments de solution 5***

```
#define va_list             char *
typedef char * va_list;

#define va_start(ap,arg)    ap = (char*) ((&arg) + 1)
#define va_start(ap,arg)    ap = ((char*) &arg) + sizeof(arg)
```

En deux étapes pour `va_arg()` :

```
1-        "return"   *(type*) ap
2-        (char *) ap += sizeof (type)
2 bis-    (type *) ap ++
```

et donc

```
#define va_arg(ap,type)     *((type*) ap++)
```

•)